FACULTY OF ENGINEERING AND ARCHITECTURE

Investigation and simulation of a decentralized social network

Karel Haerens
Student number: 01408954

Supervisors: Dr. ir. Ruben Taelman, Prof. dr. ir. Ruben Verborgh

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

Academic year 2019-2020

GHENT UNIVERSITY

Acknowledgments

First, I would like to thank my supervisor, Dr. ir. Ruben Taelman, for his helpful insights, his intellectual and moral support, and his fast email replies.

Next, my thanks go out to my supervisor Prof. Dr. ir. Ruben Verborgh, without whom I would have never discovered the incredible world of linked data.

I would also like to thank my friends and family, for their support throughout this journey.

Karel Haerens
August 16, 2020

Usage

Karel Haerens
August 16, 2020

Investigation and simulation of a decentralized social network

Karel Haerens

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

Supervisors: Dr. ir. Ruben Taelman, Prof. dr. ir. Ruben Verborgh

Department of Electronics and Information Systems
Chair: Prof. Dr. Ir. K. De Bosschere
Faculty of Engineering and Architecture
Academic year 2019-2020

Summary
This dissertation investigates the effects of decentralization on social network applications with respect to functionality and time to complete actions of users. In a first step we propose a hypothetical social network application based on linked data and compare its functionality to a centralized social network. Next, a social graph generating algorithm is implemented and the results compared to the Facebook social graph. The social network application and network generator are combined to create an agent-based discrete event simulation, which is subsequently used to test the behaviour of decentralized social networks under various circumstances and configurations.

Keywords: Semantic web, Solid, Linked data, Discrete event simulation, Social network

Investigation and simulation of a decentralized social network

Karel Haerens

Supervisors: Dr. ir. Ruben Taelman, Prof. dr. ir. Ruben Verborgh

Abstract— **This paper investigates the effects of decentralization on social network applications with respect to functionality and time to complete actions of users. In a first step we propose a hypothetical social network application based on linked data and compare its functionality to a centralized social network. Next, a social graph generating algorithm is implemented and tested. The social network application and network generator are combined to create an agent-based discrete event simulation, which is subsequently used to test the behaviour of decentralized social networks under various circumstances and configurations.**

Keywords—**Semantic web, Solid, Linked data, Discrete event simulation, Social network**

I. Introduction

In the past ten years the world has flocked to online social media platforms to connect and share their lives with others [1]. These enormous platforms all share one common characteristic: large scale collection and analysis of user data for the purpose of targeted advertisement as a business model. Recent events such as the whistle blowing of Edward Snowden in 2014 [2] and the Facebook-Cambridge Analytica scandal in 2018 [3] have made it clear that this precarious situation, where the data of millions is controlled by a few tech giants, is unmaintainable. To break these data monopolies, a bottom-up revolution is needed in the form of *decentralization*. Tim Berners-Lee, the creator of the World Wide Web, has proposed a new decentralized platform called Solid [4], based on Linked Data, that gives users back control of the information they share, and provides a uniform API for developers to build applications that can *reuse* existing user data, and as such reduce applications to *views* of data [5]. These benefits come with the downside that querying data becomes more complex in decentralized systems, since a group of uncoordinated sources need to be searched in order to find information, whereas centralized systems can make use of highly optimized databases to resolve queries. In this work we consider a social application based on linked data, inspired by Solid, and implement it into a discrete event simulation in order to test how server response times change under various conditions.

II. Related Work

A. Linked Data

Linked Data is a cornerstone technology of the Semantic Web, an idea proposed by Tim Berners-Lee with the goal to make data on the web accessible to machines [6][7][8]. Linked Data is published in RDF [9], a schemaless a graph-based data model that represents information in the form of *triples*, that have the following structure:

$$<subject> <predicate> < object>$$

where subject and predicate are HTTP URIs that can be resolved on the web to provide more information, and the object can either be an HTTP URI or a typed literal. RDF can be queried in a language called SPARQl [10]. Combining many of these triples creates *knowledge graphs* and through the usage of URIs as names for things, linked data essentially creates a knowledge graph that spans the entire internet, often referred to as the *Web of data*.

B. Solid

Solid is a decentralized platform proposed by Tim Berners-Lee upon which applications involving user data can be built [11] [5]. Solid is based on *personal online datastores* or pods: these are containers that hold all personal information of a user. Pods can be located on the servers of a *pod-provider* of the user's choosing, or be hosted by users themselves. Pods are implemented according to the linked data platform (LDP) specification [12], and contain user information in the form of linked data. The triples in a pod are stored in *linked data documents* fashion, i.e. all triples *about* a certain URI are stored in the file pointed to *by* that URI. The servers themselves do not execute queries for specific triples, rather they serve the linked data documents, that can subsequently be queried at client-side. In this paper we will propose a hypothetical application based on these same principles, to then implement into a discrete event simulation.

C. Social Graph Characteristics

A social network is a structure made up of people that maintain one-to-one social relationships. Such a network can be modeled by a *social graph*: The nodes in the graph represent people, while the the vertices represent the presence of some predefined social relationship between two persons. We briefly describe a number of mathematical graph properties which are relevant to the study of social graphs:

- **Degree** The degree k of a node is the amount other nodes it is connected to, thus in a social graph, it represents the amount of people an individual is connected to.
- **Degree distribution**: Let p_k be the fraction of the network that has degree k. The degree distribution shows the values p_k for every value of k [13].
- **Hop distance**: The minimum amount of connections one has to traverse to reach one node starting from another.
- **Neighbourhood function**: The neighbourhood function $N(h)$ of a graph describes the number of pairs (u, v) such that u is reachable starting from v, taking a path that counts h edges or

less. In other words, it shows what portion of the graph we can cover in h hops on average, starting from a random node. This measure is more robust than the average hop distance [13].

- **Degree assortativity**: The correlation between the degree of a random node, and the degree of the nodes it is connected to. For social networks, a positive assortativity means that people with many friends are connected to other people with many friends.
- **Local clustering coefficient**: The ratio of the number of connections in the *neighborhood* of a node (i.e. the induced subgraph consisting of a node and all its one-hop neighbors) over the amount of *possibles* connections in that neighborhood.

Following characteristics are typical for social graphs [13], [14]:

- Sparseness
- A fat-tailed degree distribution
- A positive degree assortativity
- A Short average hop distance
- A high average local clustering coefficient

We will propose an algorithm to generate social graphs that display these properties.

D. Discrete Event Simulation

The discrete event simulation (DES) is a popular method for modeling systems wherein behaviour is partly stochastic, and changes to the system happen at discrete moments in time, such as requests arriving at servers [15] . Because of this latter property, it is not necessary to simulate the system in intervals *between* changes. The simulation thus "jumps" through time, from one state change to the next. These state-changing events are kept in a chronologically sorted *event list*, which is traversed throughout the simulation. State changes are handled one by one, and any state-changing events they might *cause* are appended to the event list. An approach to model Poisson processes in a DES is the following: whenever some type of event is encountered, the *same* type of event is appended to the event list, but at a time increment that is drawn from an exponential distribution.

III. Social Application Model

In this section we lay out a design for a hypothetical social network application based on pods. This design will then later be implemented into a discrete event simulation. The model we present is only theoretical, and as such does not implement real-world design criteria such as privacy.

A. Functionality

Starting from the definition of a social network site in [16] and expanding upon it, we present the following list of desired functionalities:

1. Users have a profile that contains personal information.
2. Users maintain a list of profiles to whom they are connected. It would be possible to create a Twitter-like application with "followers", i.e. unidirectional connections, but we choose to make the connections bidirectional, in the style of Facebook "friendships", as we have data on the graph properties of the Facebook social graph [13].
3. Users can create content in the form of text posts. The post should reside in the pod of the creator.
4. Users can leave comments on content their connections have created, which can itself consist of a comment or a text post. These comments should reside in the pod of the creator, while a link *to* this comment should be added to the original post.
5. Users can get an overview of the recent posts and comments made by their connections, together with the comments other people have created in reaction to those posts (cfr. Facebook's *news feed*) .

B. File Structure

To achieve the proposed functionality, we propose the following set of files and folders, kept in the datapod of each user:

1. A profile, a file containing all personal information of a user, and a list of connected users.
2. A folder called "posts", containing a users text posts. Each post gets it's own file within this folder. A post keeps a link to the creator, and keeps links to any comments that might have been left by other users in reaction to the post.
3. A folder called "comments", again containing a single file per comment created by the user. A comment file keeps reference to the creator and to the original content it has been created for.
4. An *event file*, a file that contains most recent *events* a user has generated. An event keeps a reference to the event source. Only a limited number of the most recent events is kept.

C. User Actions

Within the application a user can perform the following actions: creating posts and comments, and compiling news feeds by collecting *event files* of their connections. Note that compiling a news feed is a sequential process, where first the event files of connected users, then the event sources of those events, and then the comments on the event sources must be collected. This process thus requires requests to the datapods of all connected users, and any users that have left comments on the posts of those users. Note that although these *stages* of news feed collection are sequential, the individual requests per stage are issued in parallel, e.g. once an event file arrives, all events sources it contains are fetched at the same time.

D. Discussion

The system we propose in this section offers basic social network functionality: creating content such as text posts and leaving comments is still possible while keeping all data in the pods of the creators through linking. Through the use of "events", which keep track of the most recent actions a user has undertaken, compiling of news feeds becomes tractable, and avoids the need to gather and filter needlessly large amounts of data. In our implementation these events are only used to represent the creation of a new text post or a new comment, but they could just as well be used to represent videos, links, articles, birthdays, etc.

One feature a decentralized system doesn't offer is the ability to easily *look up* profiles and content in a single request, since data is spread out over multiple servers that aren't necessarily indexed. Two approaches are possible to resolve this problem: we can either aggregate and index all servers through web crawling, or use link traversal strategies, possible aided by data sum-

maries, to find relevant sources on the fly [17]. Data summaries allow for a quick evaluation of whether a source can contribute to a given query, and thus speeds up finding sources through link traversal.

IV. Social Network Simulation

In this section we describe the component of the discrete event simulation. This simulation is implemented in Javascript using the Node.js framework.

A. Social Graph Generation

The first step in simulating a social network is to generate a realistic social graph structure, exhibiting the properties in section II-C. To this end we implemented an algorithm that creates a *hierarchy of clusters*. At each step of the algorithm, a set of clusters of nodes is generated according to input parameters. In the next iteration, these clusters themselves are treated as the nodes that in turn get clustered. The connections between clusters are then finally resolved into inter-person connections. To achieve a positive degree assortativity, we select users to connect randomly, but with a probability proportional to the amount of connections they have. This has the effect that highly connected users are again connected to highly connected users, and that highly connected users become even more connected, thus creating a heavy-tailed degree distribution. The validity of this network is tested in section V. User activity is modeled as a Poisson process with a rate proportional to the amount of connections a user has [18].

B. Server Model

In the simulation all datapods of users are stored on simulated servers: simple datastructures with following fields: a request queue, i.e. a list holding pending requests, latency and bandwidth. All users are distributed evenly across the servers. Servers distribute available bandwidth equally over all pending requests. Server traffic is dominated by read request, thus write request are assumed to happen instantly, without going through the server queue [18].

V. Experiments

A. Experiment 1: Network Generator Validation

In this experiment we investigate how the properties of the generated networks compare to random networks and the Facebook social graph properties [13]. We generate 50 networks of 480 people, analyze them in Python using the Networkx library, and average out the results.

B. Experiment 2: Decentralization With Constant Bandwidth

In this experiment we investigate how distributing a fixed set of user pods over an increasing amount of servers affects the server response time, and the average time it takes to compile news feeds, when total bandwidth in the system remains constant, i.e. the more servers there are, the less bandwidth each server receives. The experiment is conducted with a network size of 120, and the amount of servers is varied from 1 to 100. The simulated period is 1 day. A total bandwidth of 1 Gbps is present in the system. This experiment is run 5 times and the results are averaged out.

C. Experiment 3: Increasing User Activity

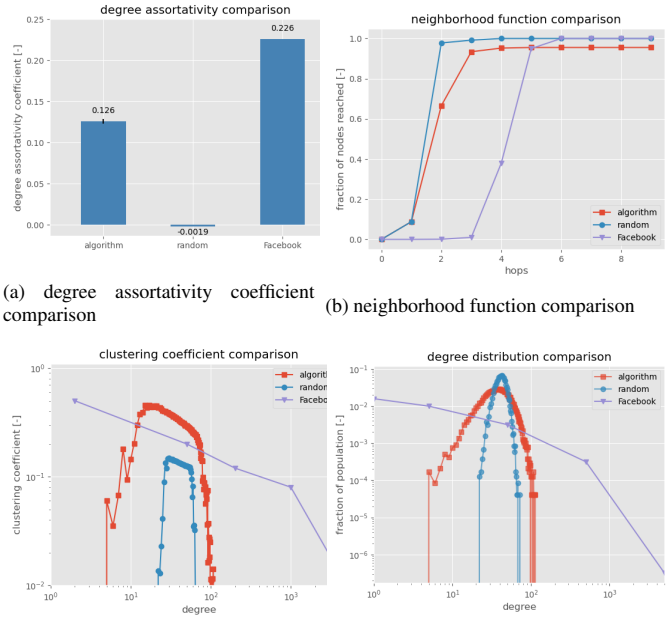
In this experiment we investigate the change in server response time and average time it takes to compile a news feed when user activity increases. The experiment is conducted with a network size of 120, and the average frequency of requesting a news feed varies form 3 to 215 times a day. The simulated period is 1 day. This experiment is run 5 times and the results are averaged out.

D. Experiment 4: Increasing Network Size With Self-Hosting Users

In this experiment we investigate the effects of increasing the size of a network, where all users host *their own* datapods on smaller servers, i.e. the number of servers in this experiment is always equal to the amount of users in the network. We vary this number from 60 to 240, and assume each user has a personal server with a bandwidth of 0.01 Gbps. Again we measure server response time and average time it takes to compile a news feed. The simulated period is 1 day. This experiment is run 5 times and the results are averaged out.

VI. Results

A. Experiment 1: Network Generator Validation



(a) degree assortativity coefficient comparison

(b) neighborhood function comparison

(c) clustering coefficient in function of degree comparison

(d) degree distribution comparison of algorithmically generated networks

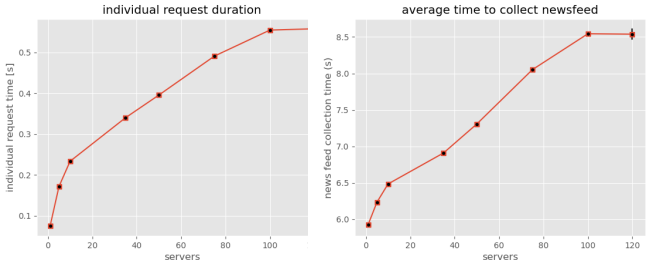
Fig. 1: Comparison of characteristics between algorithmically generated networks, randomly generated networks, and the Facebook social graph

The overall results are shown in figure 1. From this experiment we learn that our algorithm outperforms random networks

for all measured properties, but still doesn't fully capture the characteristics of a real social graph. The steep drop-offs in figure 1c and 1d are due to the smaller network size. In general we note that it is hard to precisely tune the properties of a network, as these aren't orthogonal to one another.

B. Experiment 2: Decentralization with constant bandwidth

The results of this experiment are shown in figure 2 and show that the time to resolve individual requests and time to load news feeds do increase with increasing decentralization, with 625% and 23% respectively. The difference between these two percentages is due to the fact that many requests are issued in parallel. The strong increase in server response time is due to a "popularity" effect; a higher degree of decentralization means less users, and thus less bandwidth per server. If some of those few users is very popular, and its files are requested often, this server is flooded with requests, slowing down average response times.

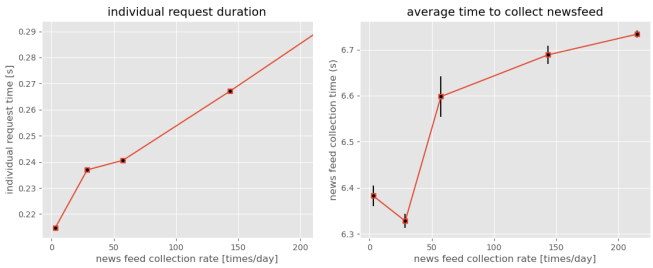


(a) Average time between start and completion of an individual request (b) Average time between start and completion of gathering a news feed

Fig. 2: Results of experiment 2: varying amount of servers w/ constant bandwidth in system

C. Experiment 3: Increasing User Activity

Again we see in figure 3 that the server response time and the total time to collect news feeds increases, but less severely than the increase seen in the previous experiment: an increase in request frequency a factor 200 only increases the server response time by $\approx 30\%$. The time to collect news feeds is again lower, and increases by 4%.



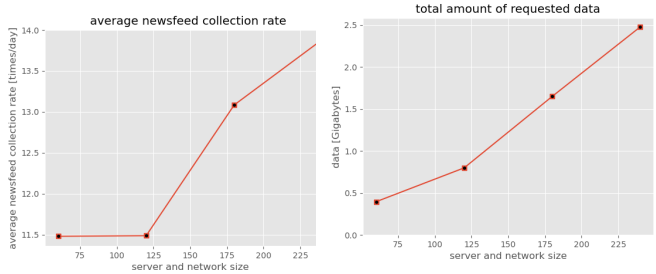
(a) Average time between start and completion of an individual request (b) Average time between start and completion of gathering a news feed

Fig. 3: Results of experiment 3

D. Experiment 4: Increasing Network Size With Self-Hosting Users

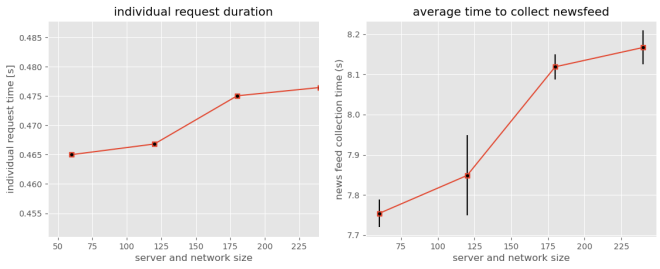
In this experiment we change the *size* of the network, and by doing so slightly alter some important network characteristics, most notably the average amount of connections per user and, as a consequence, the user activity rate, and thus the load on the system. These latter two values are shown in figure 4, and show that the load increases slightly with increased network size. This needs to be taken into account when looking at the change in server response times and average time to compile a news feed shown in figure 5.

In figure 5a we see that the time to resolve an individual request at the servers barely changes, from 0.465 to 0.476 seconds with increasing network size, i.e. an increase of 2% . The average time to collect a news feed changes from 7.75 to 8.15 seconds, shown in figure 5b. This increase is at a higher percentage than the individual request times, by 5%. This is likely due to the increased average amount of friends, as users need to collect more files on average per news feed collection.



(a) Average news feed request rate (b) Total amount of data requested throughout simulation

Fig. 4: network characteristics and server load of experiment 4



(a) Average time per individual request (b) Average news feed collection time

Fig. 5: Results of experiment 4

VII. Discussion

We first note that the simplifications made in the experiment set-ups and small network sizes might affect numerical results, yet the general trends are expected to stay valid for larger, more complex, networks.

The first experiment shows that the proposed hierarchical clustering algorithm is a viable approach to create social graph struc-

tures, yet more research in optimal parameters is needed to truly mimic networks such as Facebook.
The second experiment shows that decentralization of data increases the time it takes to complete actions if bandwidth remains constant. The extrapolated implication of this finding is that a decentralized network requires *more* bandwidth to offer the same response times when compared to a centralized system. The third experiment shows that the time to execute user actions increases with increased frequency of those actions, albeit not by much. The implication for pod providers is that they should be prepared to scale up in bandwidth when Solid were to suddenly become a popular platform.
The final experiment indicates that self-hosting is a valid strategy for a decentralized network in terms of time to complete user actions. If a Solid-like network were to use self-hosting as a selling point and emphasize the complete data control that is created by storing the data *physically* in the homes of users, a sudden increase in popularity wouldn't affect the server response times as much as in a system that relies on pod-providers that need to scale up with the increased network traffic, as shown in experiment 2 and 3.

VIII. Conclusion

In this paper we have described a hypothetical decentralized social network application, inspired by the Solid platform. We implemented this social network application into a discrete event simulation, combined with an algorithm to generate realistic social graph structures, to test such a decentralized system under various configurations and circumstances regarding network size, number of servers and user activity.

We found that basic social network functionalities can be mapped onto decentralized networks, but that there exist features that are more easily achieved in a centralized system, such as looking up specific content, which require smart querying tactics in a decentralized system.

The proposed network generating algorithm was validated against a random baseline and the Facebook social graph. It was shown that the algorithm is a viable approach, yet more research is needed to determine optimal parameters to completely capture the characteristics of an online social graph.

The proposed discrete event simulation was used to test a number of set-ups: varying network size with constant total bandwidth, varying user activity rates and varying network size with self-hosting users. We showed that for the first two experiments, the time it takes to complete user actions goes up, while for the last experiment this time remained constant.

IX. Future Work

A. Expanding Functionality

The application simulated in this work only implements very basic features such as creating posts and comments, and compiling news feeds. Future work could expand upon this set of features to mimic social networks more closely. Features such as uploading images, uploading videos, chatting with other users, pages for events and organizations attracting, following profiles instead of bidirectional friendships, etc. could all be added.

B. Experimenting With Load Distribution

The generated network traffic was assumed to be static over time in this thesis. In future work it would be interesting to see how decentralized networks withstand sudden surges in traffic, for example when a piece of content goes "viral".

C. Querying Social Networks

One of the features that a decentralized social network does not offer is a straightforward way to *look up* specific profiles or content on social media. It would be interesting to investigate ways to make this possible, for example through data summaries and link traversal query execution.

D. Social Application Benchmark

In this work we opted to simulate simple servers in a discrete event simulation. As a general remark on that approach, it was found that discrete event simulations - although allowing for fine control simulation parameters, and giving clear insight in the simulated system - is costly in terms of runtime, and it might be worthwhile to investigate alternative simulation methods, such as simulating the network with *real* servers. The work presented in this dissertation could be used as a benchmark generator, that generates the load under which the servers operate.

E. Facilitate self-hosting

In experiment 4 we show that the approach of self-hosting pods with small servers scales well with a growing network size. Self-hosting of pods is possible on the Solid platform, yet requires a prohibitive amount of technological knowledge to be accessible to a large public[1]. It would be interesting to investigate the development of physical *plug-and-play* data pods that anyone could easily set up at home. This could also be a selling point of the Solid platform: anecdotal evidence of the author proves that it's easier to explain to a layman that his data is on a pod he has at home, than the subtle difference between what server ("what's a server?") his data resides on.

References

[1] Hannah Ritchie Max Roser and Esteban Ortiz-Ospina, "Internet," *Our World in Data*, 2015, https://ourworldindata.org/internet.
[2] Bryan Burrough, Sarah Ellison, and Suzanna Andrews, "The Snowden Saga: A shadowland of secrets and light," *Vanity Fair*, apr 2014.
[3] Nicholas Confessore, "Cambridge Analytica and Facebook: The Scandal and the Fallout So Far," *The New York Times*, apr 2018.
[4] Andrei Vlad Sambra, Essam Mansour, Sandro Hawke, Maged Zereba, Nicola Greco, Abdurrahman Ghanem, Dmitri Zagidulin, Ashraf Aboulnaga, and Tim Berners-Lee, "Solid: A Platform for Decentralized Social Applications Based on Linked Data," Tech. Rep., 2016.
[5] Ruben Verborgh, "Paradigm shifts for the decentralized Web," Tech. Rep., dec 2017.
[6] David Wood, Marsha Zaidman, Luke Ruth, Michael Hausenblas, *Linked Data, structured data on the web*, Manning, 2014.
[7] Tim Berners-Lee, "Linked Data - Design Issues," 2006.
[8] Christian Bizer, Tom Heath, and Tim Berners-Lee, "Linked data - The story so far," *International Journal on Semantic Web and Information Systems*, vol. 5, no. 3, pp. 1–22, jul 2009.

[1] https://solidproject.org/use-solid/

[9] Richard Cyganiak, David Wood, and Markus Lanthaler, "RDF 1.1 Concepts and Abstract Syntax," Tech. Rep., W3C, 2014.
[10] Steve Harris and Andy Seaborne, "SPARQL 1.1 Query Language," Tech. Rep., W3C, mar 2013.
[11] Tim Berners-Lee, James Hendler, and Ora Lassila, "The Semantic Web," vol. 284, pp. 34–43, 2001.
[12] Steve Speicher, John Arwe, and Ashok Malhotra, "Linked Data Platform 1.0," Tech. Rep., W3C, feb 2015.
[13] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow, "The anatomy of the facebook social graph," *arXiv preprint*, vol. 1111.4503, 11 2011.
[14] Lynne Hamill Gilbert and Nigel, "Social Circles: A Simple Structure for Agent-Based Social Network Models," *Journal of Artificial Societies and Social Simulation*, mar 2009.
[15] Antuela A. Tako and Stewart Robinson, "Model development in discrete-event simulation and system dynamics: An empirical study of expert modellers," *European Journal of Operational Research*, vol. 207, no. 2, pp. 784–794, dec 2010.
[16] Danah Boyd and Nicole Ellison, "Social Network Sites: Definition, History, and Scholarship," *Journal of Computer-Mediated Communication*, vol. 13, no. 1, pp. 210–230, oct 2007.
[17] Jürgen Umbrich, Katja Hose, Marcel Karnstedt, Andreas Harth, and Axel Polleres, "Comparing data summaries for processing live queries over Linked Data," *World Wide Web*, vol. 14, no. 5, pp. 495–544, oct 2011.
[18] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan, "LinkBench: a Database Benchmark Based on the Facebook Social Graph," Tech. Rep., Facebook, 2013.

Contents

List of Figures

List of Tables

Acronyms

API application program interface. 2

CSV comma separated values. 7

DES discrete event simulation. 13

GDPR General Data Protection Regulation. 1

HTML hyper text mark-up language. 8

HTTP hyper text transfer protocol. 6

JSON JavaScript object notation. 8, 28

LDP linked data platform. 11

LOD linked open data. 7

pod personal online datastore. 11

RDF resource description framework. 6, 8

RDFa RDF attributes. 8

SPARQL SPARQL Protocol and RDF Query Language. 7, 15

UML unified modeling language. 23

URI uniform resource identifier. 6, 28

XML extensible mark-up language. 8

Chapter 1

Preface

1.1 Introduction

"Data is the new oil". When this phrase was first coined in 2006, the person who did so -Clive Humby, a data scientist who helped create the world's first loyalty card at Tesco [1]- could have never predicted how right he would turn out to be: at the time, Facebook counted only twelve million users, both Twitter and Reddit had just been launched, and overall only 18% of the world was online. In comparison, today 46% of the world uses the internet, over 300 million people are on Twitter and Reddit respectively, and Facebook counts 2.38 billion users as of 2019 [2].

Those billions of people create an unrelenting stream of data, ranging from the holiday pictures they upload to the links they carelessly click, a stream that pours into the server farms of companies like Google and Facebook, where it is subsequently analyzed and put to use for targeted advertisement. If data is the new oil, the user bases of large social media platforms have become the new oil fields, and tech giants are squeezing every drop out of them.

Events such as the whistle blowing of Edward Snowden in 2014 [3], where it was revealed that the American government has access to private data on the servers of American social media companies, and the Facebook-Cambridge Analytica scandal in 2018 [4], in which the data of over 80 million users was harvested and analyzed for political campaign purposes (without permission of those users), have made it painfully clear how valuable, but more importantly, how *vulnerable* our personal data has become in recent times.

In response to these growing concerns the European union implemented a set of laws in 2018, known as the General Data Protection Regulation (GDPR) [5], to rein in companies regarding privacy and personal data of their users. Top-down regulations like this are of course strides in the right direction, yet pose only a

partial solution to the wider problem of data monopolies; to truly address the issue, a bottom-up revolution is required, in the form of *decentralization*.

1.2 problem Statement

Today, the choice presented to users when signing up for Facebook is to either consent to the stated privacy policy "as is", or simply not to see what their friends and family share of their lives. Users that became fed up with Facebook's stance on hate speech in the recent "stop hate for profit" [1] campaign had no way to easily switch to a different network without having to completely recreate their profiles and connections (assuming those connections also made a switch to the *same* network, otherwise that online friendship would become impossible to maintain). Current social networks are intentionally designed in such a way that our data becomes "locked in". Developers that do want to access this data in their own applications are subjected to the whims of social network APIs, that can change policies and restrict access at any time.

Such problems are solved by *distributed* or *decentralized* social networks, networks in which there exist many providers, which might all offer different privacy policies and user experiences, but are compatible with one another. Examples of such networks are Friendica[2], Mastodon[3] and Diaspora[4], which are all part of the Fediverse, a collection of interoperable social networks.

In 2016, a remarkable decentralized social network initiative was launched by the creator of the World Wide Web, Tim Berners-Lee, called Solid (initially derived from "Social Linked Data") [6]. Rather than a social network like Facebook or Twitter, the project comprises a set of standards upon which social applications -or any type of application involving user data- can be built. Solid stands out because of it's aim to provide complete data control for users, and it's radically open attitude towards application developers. Data control is achieved through the use of *pods*, containers in which users store data, at a provider of their choosing, in a country with adequate privacy laws for example. The openness toward developers follows from the use of Linked Data, allowing developers to create applications that can interpret and *reuse* existing data in pods, which stands in stark contrast to the data *lock-in* mentioned before.

This data reuse allows users to switch platforms instantly, essentially reducing applications to *views*[7], and thus reinstating users as the primary consumers of social platforms, instead of the products. If data is the new oil, Social Linked Data might just be the renewable we need.

[1] www.stophateforprofit.org
[2] https://friendi.ca/
[3] https://mastodon.online/about
[4] https://diasporafoundation.org/

Yet, there's no such thing as a free lunch. Together with the many advantages of decentralizing data come some disadvantages, which will need to prove surmountable if Solid is to be adopted by a wider public. The most obvious drawback of decentralization is the fact that querying data becomes significantly more complicated. Whereas centralized systems can make use of highly optimized database systems to resolve queries, executing a query over decentralized data might require many *different* uncoordinated sources to be consulted through the web, which typically results in a longer execution time. Efficiently executing these so-called "federated" queries is still a matter of ongoing research. A notable advancement in this field is link traversal based query execution [8], in which new data sources are found "on the fly" by following links in intermediate results. This approach allows for unexpected sources to contribute to the query results, fully exploiting the *discoverability* of the semantic web.

For applications where time isn't a critical factor, this drawback might be acceptable. Yet for others, such as social media applications, which are nowadays used by a public that has grown accustomed to very fast loading web pages, the effects of this disadvantage could make or break social applications built upon Solid. It is precisely this problem that is addressed in this thesis.

We consider a decentralized social network application, where each user has it's own datapod, which might be stored at the server of a provider, or hosted by the user itself (potentially negatively affecting uptime and bandwidth). In the spirit of Solid, users in this network have complete control over their data, thus all content they create should reside in their own pod, and be linked to by others. In such a system, simple operations such as reading a friend's new post and the comments left by other people would require queries over many datapods, since the post itself and all comments are stored in the pods of their creators. Whereas centralized applications such as Facebook are able to serve a news feed in a single request, which is subsequently processed server-side, a decentralized application needs to request each piece of content individually. In this thesis we investigate how this decentralization affects waiting times for users.

1.3 Research Questions

In this thesis the effects of decentralization on social network applications through linked data, as is used in Solid, is investigated, summarized in the research question

- *How does decentralization affect social network applications?*

and more specifically

1. *How does decentralization affect social network applications in terms of functionality?*

2. *How does decentralization affect social network applications in terms of time to complete user operations?*

The first sub-question serves to investigate the network dynamics of a decentralized social network built upon linked data. The findings of this research will then be used to answer the second sub-question, namely how these dynamics affect the time it takes to complete user actions such as loading a news feed or posting a microblog.

1.4 Hypotheses

For these research questions we present the following hypotheses:

1. It's possible to map the functionality of a centralized social network (e.g. Facebook) onto a decentralized social network based on data pods.
2. The average time to complete user operations will increase with increasing decentralization of data when keeping total bandwidth of all servers constant.
3. The average time to complete user operations will increase with increasing frequency of these operations.
4. The average time to complete user operations will not increase significantly with increasing network size when each user hosts it's own pod.

These hypotheses will be either accepted or rejected at the end of this dissertation.

1.5 Outline

The remainder of this work is structured as follows: in chapter 2 we discuss related work, next, in chapter 3 we propose a hypothetical decentralized social network and discuss its functionality, we then describe how this social network application is simulated in a discrete event simulation in chapter 4, where we also lay out the algorithm used to generate realistic social graph structures. Then in chapter 5 we describe the experiments we conducted in order to validate the network generation algorithm and test our hypotheses together, we also report the results of these experiments here, which we discuss in chapter 6. We formulate conclusions in chapter 7, and discuss future work in chapter 8.

Chapter 2

Related Work

In this chapter we will discuss a number of subjects which proved relevant to the design and implementation of the experiments conducted in later chapters.

2.1 The Semantic Web

2.1.1 Origins

The world wide web as most people know and use it today consists of a large collection of documents that are connected through the use of *hyperlinks*, which in itself don't mean anything more than "this page has something to do with that page". For humans this vagueness is hardly ever a problem, as we can read the context in which a link appears and deduce what the relation between the sources is. Yet before the recent developments in natural language processing through machine learning, it was inconceivable that computers would ever be able to do the same. Thus for a program, this so-called *web of documents* was a heap meaninglessly linked pages, and the information that lay within was mostly inaccessible without a human mind to parse it.

In his 2001 article [9] Tim Berners-Lee introduced a concept that was designed to counteracts this ambiguity and finally allow machines to comprehend and act upon the wealth of data present on the internet: the semantic web. By meaningfully *annotating* pieces of information and the links between them with specific names and labels, computers would be able to unambiguously interpret data and thus become intelligent agents, capable of performing complex "human" tasks.

2.1.2 Linked Data

In order to realize the semantic web, a couple of things we're needed: firstly, a set of technological standards and conventions upon which the network itself

would be built were required. These standards are described in [9], and the most fundamental ones are listed here:.

Technologies and Standards

- The hyper text transfer protocol (HTTP), which is also used by the web of documents to transfer web pages.
- the use of URIs, or uniform resource identifiers, unique identifiers for resources, of which the URL, or uniform resource locator, is a better known subset, it being a URI that can be resolved on the web through the HTTP protocol)
- resource description framework (RDF), a simple data model in which all data is represented in the form $< subject > < predicate > < object >$. These simple statements are called *triples*.
- the use of *vocabularies* (or *ontologies*) to annotate concepts and the links between them. Vocabularies are collections of labels which have clear definitions, expressed in RDF.

the latter two of these technologies will be discussed further in this chapter.

Linked Data Principles

In order to allow for the creation of useful semantic web applications, lots of data complying to the standards first had to be made available. In 2006, Tim Berners-Lee published a set of principles as a guide for publishing data so that could be useful for the semantic web, knows as the *Linked Data principles* [10]:

1. Use URIs as names for things.
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information.
4. Include links to other URIs. so that they can discover more things.

In 2010, these principles were appended with a 5-star rating scheme:

1. ★
   Available on the web (whatever format) but with an open licence, to be Open Data

2. ★★
   Available as machine-readable structured data (e.g. excel instead of image scan of a table)

3. ★★★
   as (2) plus non-proprietary format (e.g. CSV instead of excel)

4. ★★★★
   All the above plus, use open standards from W3C (RDF and SPARQL) to identify things, so that people can point at your stuff

5. ★★★★★
   All the above, plus: Link your data to other people's data to provide context

Guided by these principles, and encouraged by the *linking open data project*[1], a vast amounts of data have been published in the past 20 years by governments, research institutions and companies alike [11]. The *Linked Open Data cloud*, or *LOD-cloud* [2], gives an overview of published data sets and the links between them.

The web of data today covers a plethora of diverse disciplines and topics, such as people, companies, books, scientific publications, films, music, television and radio programs (most notably those of the BBC [12]), genes, proteins, drugs and clinical trials, online communities, statistical data, census results, and reviews. [11]

Some notable data "hubs" -data sets linked to by many other data sets- on the web of data include:

- DBpedia[3], a data set extracted from Wikipedia's "info boxes", which are usually found in the upper right-hand corner of a Wikipedia page, consisting of more than 103 million RDF triples [13].
- GeoNames[4], a data set containing information on over 25 million geographical names.

[1] https://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData
[2] https://lod-cloud.net/
[3] https://wiki.dbpedia.org/
[4] http://www.geonames.org/

RDF

The resource description framework (RDF) is a data format for representing schemaless data on the Web [14]. All information in RDF is expressed in the form of *triples*, which take on the shape

$$<subject> < predicate> < object>$$

Where the subject and predicate positions are URIs, and the object position can either be a URI or a typed literal, e.g. a string, a date, or a number.

A statement like this simply describes two things and the relationship between them. Multiple RDF statements connect to form *graphs* of information. It should be noted that RDF is a data *model*, not a data format. RDF can be serialized into several data formats, which are all useful in different contexts. The most common serializations are given below [15]:

1. turtle: a human readable form
2. RDF/XML: the original RDF format in XML
3. RDFa: RDF embedded in HTML attributes
4. JSON-LD: RDF serialized in JSON

As an example, let's say Alice and Bob both have their own web pages, "http://alice.com/me" and "http://bob.com/me" respectively, which they use to refer to themselves on the web. If Alice would like to state that she knows Bob on her page, she can do so by using a triple. However, in order for the triple to be interpretable by humans and machines that don't understand English (or language as a concept, for that matter), she needs to use a label that is recognized by others as well. This is where ontologies come in, and it's the subject of the next section. For now, let's assume there's a universally accepted predicate "http://example.com/knows" to express that the subject knows the object. Note that this predicate is also a URI that can be resolved, ideally to find more information about what it really means to "know" someone. This statement now takes on the form:

<http://alice.com/me> < http://example.com/knows> <http://bob.com/me>

And results in the graph shown in figure 2.1:

Figure 2.1: an RDF statement as a graph

Now let's say Bob wants people to know that his nickname is "Bertrand" and that his birthday is on the 16th of August. (Again, we assume there are predicates "http://example.com/nickname" and "http://example.com/birthday", that are understood by everyone). The graph would then take on the shape shown in figure 2.2:

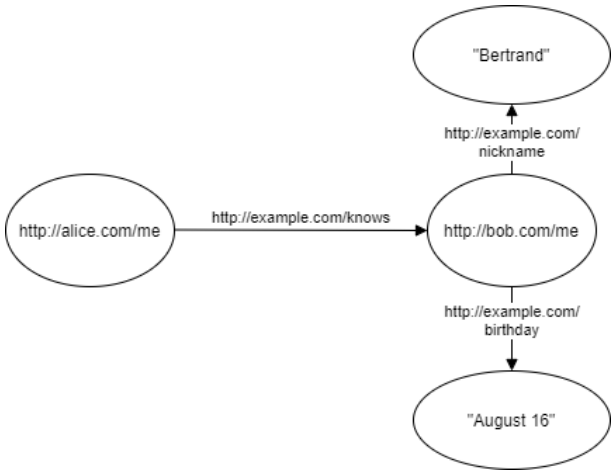


Figure 2.2: a set of RDF statements as a graph

Note that the values "Bertrand" and "August 16" are not URIs but literals of the types string and date respectively.

Ontologies

In the previous section we assumed the existence of some universally understood terms such as "http://example.com/knows" to express certain concepts. In reality, we don't expect people and machines who stumble across this data to just know what those terms mean. Rather, we *define* what they mean in so-called vocabularies (also called ontologies, though this name is usually reserved for more complex vocabularies [16]). Vocabularies are *themselves* described in RDF triples. This provides linked data with one of it's greatest assets: *discoverability*, i.e. the possibility for people and machines to follow links, and learn what those links mean "on the fly". Many vocabularies for different purposes have been created, some of which are reused so often they have been dubbed "core vocabularies" [15], such as

- FOAF[5] for describing people
- vCard[6] for describing people and addresses

[5] http://xmlns.com/foaf/spec/
[6] https://www.w3.org/TR/vcard-rdf/

- DOAP[7] for describing projects
- RDFS[8] for describing ontologies themselves

In 2011, The companies behind the largest browsers on the web, Google, Bing and Yahoo! published a new vocabulary called "schema.org"[9]. Terms from this vocabulary are used to add structured mark-up to web pages, and allows search engines to better interpret the content of those pages [17].

2.1.3 Querying Linked Data

There are several ways in which linked data is offered on the web, with variable degrees of queryability: most data sets are not live queryable, i.e. they cannot be searched without first downloading the entire knowledge graph [18]. They are offered as data dumps, i.e. large files containing all triples of a data set. This type of data set puts the complete burden of query execution on the client, and might cause a lot of network overhead, since a client who might look for only a single triple is still required to download the whole set.

An approach that gives slightly more responsibility to the server is the so-called *linked data documents* method. In this paradigm the triples of a data set are distributed over *several* files instead of one monolithic file. The files are structured in such a way that the URI of a file is either subject or object of the triple it contains. Structuring triples *about* a URI in the file pointed to *by* that URI is in accordance with the Linked Data principles mentioned in subsection 2.1.2. This type of structuring also allows for *link traversal based query execution*, a querying strategy in which new data sources are discovered at runtime by resolving URIs in (partially) matching triples that have already been encountered [8].

On the opposite side of the spectrum we find SPARQL-endpoints, a type of endpoint where the client submits queries in the SPARQL-language, a query language designed to query RDF [19]. Thus, in this paradigm queries are completely executed server-side, incurring a high server cost. As a consequence, the majority of linked data is not offered in this way, and the SPARQL endpoints that do exist suffer from frequent downtime [20].

An approach that balances the load of query execution between client and server are *triple pattern fragments* [18], a query interface where servers are tasked with matching triples according to a pattern, given by the client, and the client with breaking down complex queries into triple patterns, optimally scheduling the requests of these triple patterns, and finally aggregating these partial pattern matches into the final results of the complete query.

[7] http://usefulinc.com/ns/doap
[8] https://www.w3.org/TR/rdf-schema/
[9] https://schema.org/

2.1.4 Solid

Solid is a decentralized platform upon which applications involving user data can be built, based on RDF and Semantic web technologies. In Solid each user can create one or multiple *personal online datastores* pod, which can reside on either the server of a provider or on a personal server. Solid applications are implemented as client-side web applications that directly read and write data from pods. As such Solid applications essentially become "views" of the data residing in the pod [7]. Data and application are decoupled from that data *by design* to allow for data reuse and seamless switching between applications [6], without the need to migrate or copy data. Users in Solid are identified through a globally unique URL called a *WebId* [21]. A solid pod itself is structured according to the *linked data documents* paradigm, in which triples are distributed over files to which they are relevant, as explained above. A solid server is implemented according to the linked data platform (LDP) specification [22], which specifies how clients can interact with the server.

In order to test the hypotheses proposed in chapter 1, we will simulate a social network that operates in a similar fashion to a Solid application.

2.2 Social Network Structure

A social network is a structure made up of people that maintain one-to-one social relationships. Such a network can be modeled by a *social graph*: The nodes in the graph represent people, while the the vertices represent the presence of some predefined social relationship between two persons. Social graphs can be used to model many types of social networks, but in the context of this thesis, we will focus on the modeling of *online* social networks. In section 2.2.1 we describe a number of mathematical graph properties, which we use in section 2.2.2 to characterise online social graphs.

2.2.1 Graph Properties

Some fundamental concepts and properties that are relevant for describing social graphs are given here:

- **Degree** The degree k of a node is the amount other nodes it is connected to, thus in a social graph, it represents the amount of people an individual is connected to.
- **Degree distribution**: Let p_k be the fraction of the network that has degree k. The degree distribution shows the values p_k for every value of k [23].
- **Hop distance**: The minimum amount of connections one has to traverse to reach to get from one node to another node, averaged out over all nodes

as start and end point. In a social graph this would represent the number intermediate acquaintances two people have between them. (On Facebook, the average hop distance is 4.3. [23]) This value is unfortunately only defined in *connected* graphs, i.e. graphs where any pair of nodes is connected though some path. Furthermore it can easily be skewed by a single ill-connected path [23]. The networks generated in later chapters aren't always guaranteed to be connected, which is why we resort to measuring the neighborhood function, described in the next bullet point.

- **Neighbourhood function**: The neighbourhood function $N(h)$ of a graph describes the number of pairs (u, v) such that u is reachable starting from v, taking a path that counts h edges or less. In other words, it shows what portion of the graph we can cover in h hops on average, starting from a random node. This measure is more robust than the hop distance.

- **Degree assortativity**: The correlation between the degree of a random node, and the degree of the nodes it is connected to. For social networks, a positive assortativity means that people with many friends are connected to other people with many friends.

- **Local clustering coefficient**: The ratio of the number of connections in the *neighborhood* of a node (i.e. the induced subgraph consisting of a node and all nodes connected to that node) over the amount of possibles connections in the neighborhood.

2.2.2 Social Graph Characteristics

A social network site is defined in [24] as a site that allows users to create public or semi-public profiles, to create a list of other users with whom they share a connection, and view and traverse their list of connections and those made by others on the website. The characteristics of social networks have been the subject of study for quite some time, with the well known assertion that all people in the world are connected through "six degrees of separation" going back to 1929, and Milgram studying the same phenomenon in 1967 [25]. Yet it is only since the advent of social network sites that researchers have been able to map and study large scale social networks such as Facebook [24].

Some key characteristics of social graphs in general are presented in [26], and are confirmed to be present in the Facebook social graph in [23]. These characteristics are summarized here:

1. Sparseness, i.e. only a fraction of all possible links in the network actually exist, reflecting that people only know few people compared to the total amount of people in the network.

2. A heavy-tailed degree distribution, i.e. the there are some nodes that are very well connected compared to the mean, reflecting the fact some

individuals have very large social "bubbles" (outside of pandemics, that is).

3. A positive degree assortativity, i.e. people are mostly connected to people who have a similar amount of friends.
4. A Short average hop distance.
5. A high average local clustering coefficient, meaning that although the network as a whole is sparsely connected, the "social bubble" of individual people is highly connected

2.3 Discrete Event Simulation

The discrete event simulation (DES) is a popular method for modeling systems where behaviour is (partly) stochastic in nature, and changes to the system happen at discrete moments in time. The technique is used in various domains such as economic behaviour modeling, energy and environmental problems, supply chain management, healthcare modelling, project management and queuing theory [27].

The basic assumption in a DES is that the state of the system only changes at discrete instances in time. The state remains unchanged in the time between those instances, and therefore must not be simulated. The simulation thus "jumps" through time, from one state change to the next. As such DES stands in contrast to continuous-time simulations, where the state of a system is usually evaluated at small, fixed time increments.

2.3.1 Flow of execution

Discrete event simulations simulate the evolution of a system in a *sequential* manner. At the core of the simulation lies a variable *clock* that keeps track of the simulated time. The simulation also maintains an *event list*, which contains *events* scheduled in the future. An event is anything and everything that changes the state of the system, e.g. the arrival of a message in a queue or the sudden crash of a server. The event list is stored in chronological order, so that the earliest future event sits at the top. A single simulation step is comprised of the following operations:

1. The top event is popped from the event list.
2. The system state is updated according to the event that has been taken from the list in (1).
3. Any new events that might have been generated because of the state change in (2) are added to the list at their respective chronological position.

The simulation ends when a certain criterion, usually a certain simulation time limit, has been reached [28].

2.3.2 Stochastic Systems

Discrete event simulations are often used to model stochastic systems, where certain event parameters or the frequency of occurrence of events are distributed according to a probability distribution [29].
The most commonly used process to model occurrences of events in a system, such as the arrival of requests at servers, is the *Poisson process*. A Poisson process is a counting process where occurrences of a certain event within an interval of length t are counted. The occurrences are independent of one another. Given that there are on average $\lambda \cdot t$ occurrences in the interval, the probability of k occurrences in the interval is given by the following probability mass distribution:

$$Pr(X = k) = \frac{(\lambda t)^k \cdot e^{-\lambda t}}{k!}$$

Where λ is called the *rate parameter*.

Another way to define a Poisson process is as follows: given is a process where events occur at discrete points in time. If the intervals in between subsequent events constitute a sequence of independent *exponentially distributed* random variables with rate λ, then the process is a Poisson process with rate λ. The exponential probability density function is given by:

$$f(x; \lambda) = \lambda \cdot e^{-\lambda \cdot x}$$

This second definition is quite useful in a discrete event simulation: if we wish to generate a Poisson process with rate λ, all we need to do is make sure that the time between two events is drawn from an exponential distribution with rate λ. In practice, when an event is popped from the event list at time t_0, we draw an interval of length l_0 from an exponential distribution with rate λ, and add a new event to the event list at time $t_0 + l_0$, thus creating a Poisson process. Such an approach is shown in figure 2.3

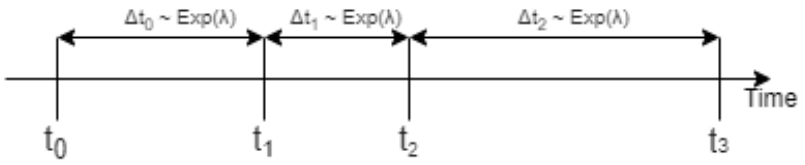


Figure 2.3: Schematic of a Poisson process

Chapter 3

Social Application Model

In this chapter we will describe a model for a hypothetical decentralized social media application. The focus lies on designing a system that provides basic social media functionality through the use of Linked data, based on the premise that every user has a *pod* in which data is stored. We assume that these pods reside in simple document-serving servers, as is the case on the Solid platform, and are *not* accessible through SPARQL-endpoints. This hypothetical system will serve as the basis for simulations in later chapters. Therefore, in order to keep these simulations tractable, we omit real-world design criteria such as privacy.

3.1 Functionality

Starting from the definition of a social network site in [24] and expanding upon it, we present the following list of desired functionalities:

1. Users have a profile that contains personal information.
2. Users maintain a list of profiles to whom they are connected. It would be possible to create a Twitter-like application with "followers", i.e. unidirectional connections, but we choose to make the connections bidirectional, in the style of Facebook "friendships", as we have data on the graph properties of the Facebook social graph [23].
3. Users can create content in the form of text posts. The post should reside in the pod of the creator.
4. Users can leave comments on content their connections have created, which can itself consist of a comment or a text post. These comments should reside in the pod of the creator, while a link *to* this comment should be added to the original post.

5. Users can see an overview of the posts their connections have recently made, together with the comments other people have created in reaction to those posts (cfr. Facebook's *news feed*) .

3.2 File Structure

To achieve aforementioned functionality, we propose the following set of files and folders, kept in the datapod of each user:

1. A profile, a file containing all RDF triples describing the user, as well as the triples describing his or her connections to other users. The file also contains references to all files mentioned below, so that these files do not have to reside in a fixed place.

2. A folder called "posts", containing this users text posts. Each post gets it's own file within this folder. A post keeps a link to the creator, and keeps links to any comments that might have been left by other users in reaction to the post.

3. A folder called "comments", again containing a single file per comment created by the user. A comment file keeps reference to the creator and to the original content it has been created for. Note that this content can itself be a comment.

4. An *event file*, this file contains most recent *events* a user has generated. An event keeps a reference to the event source (which in this simple application can only consist of the creation of a post or comment by the user, but could be cover more types of event sources in future work). The event file contains only a limited number of the most recent events, i.e. when a new event arrives, the oldest events gets pushed out. This is inspired by the Facebook news feed, which is also limited in number of events that are displayed. In a real implementation of a social application, it would of course also be possible to retrieve *all* posts or comments a user has made, but we choose to omit this functionality in favour of simplicity for our simulated application.

Posts and comments are all structured in *linked data document* fashion, i.e. the URI of a post or comment points to a file containing triples that have that URI as subject or object. Note that alternatively we could have chosen to maintain one large file containing *all* posts, and let individual posts be identified through so-called "hash URIs" [30]. This would however require other users to download this entire file each time a single post is requested, which could cause significant network overhead.

3.3 User Actions

Within the application a user can perform the following actions:

1. **Creating posts:** creating a post causes a new file to be added in the "posts" folder, en a new event to be added to the event file. This event is of the type "writeAction" in the schema.org vocabulary.

2. **Creating comments:** creating a comment causes a new file to be added in the "comments" folder, en a new event to be added to the event file. This event is also of the type "writeAction" in the schema.org vocabulary.

3. **Compiling a news feed:** As mentioned above, a news feed is created by gathering the event files of all connections, and subsequently collecting the associated event sources. Note that the traffic generated by this action depends on the amount of connections one has. Having many connections requires more data to be collected than having few.

A schematic overview of the file system is given in figure 3.1. A schematic of the graph structure of a post created by user Alice and a comment created by user Bob is shown in 3.2, where all terms except for "a" are in the schema.org vocabulary ("a" is shorthand for "rdf:type"). The dashed line represents the boundary between data pods.

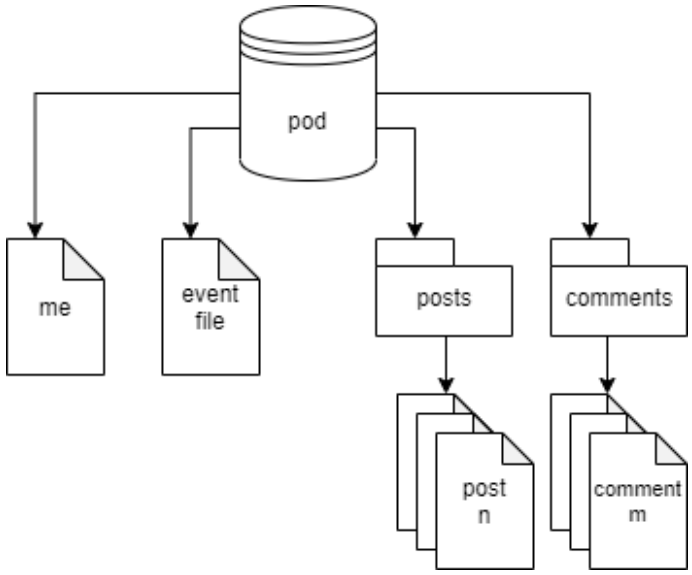


Figure 3.1: Schematic overview of the application file system

3.4 Discussion

In this section we consider whether it's possible for a decentralized social network to offer all features of a centralized network.

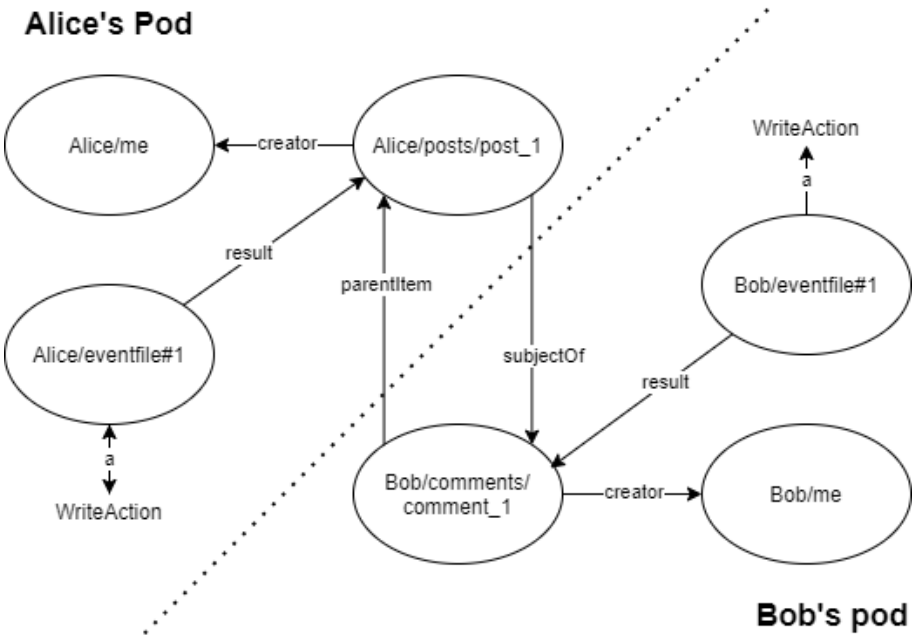


Figure 3.2: Example Graph structure of a post and comment.

The system we propose in this chapter clearly offers the most basic social network functionality: maintaining lists of connections and creating content such as text posts and comments is still possible while keeping all data in the pods of the creators through linking. Through the use of "events", which keep track of the most recent actions a user has undertaken, the creation of news feeds becomes tractable, and avoids the need to gather and filter needlessly large amounts of data. In our implementation these events are only used to represent the creation of a new text post or a new comment, but they could just as well be used to represent videos, links, articles, etc.

One feature a decentralized system doesn't offer is the ability to easily *look up* profiles and content in a single request, since data is spread out over multiple sources that aren't necessarily indexed. Two approaches are possible to resolve this problem: we can either aggregate and index all servers through web crawling, or use link traversal strategies, possible aided by data summaries, to find relevant sources on the fly [31]. Data summaries allow for a quick evaluation of whether a source can contribute to a given query, and thus speeds up finding sources through link traversal.

Chapter 4

Social Network Simulation

In this chapter we describe how the simulation we use is designed. We present the algorithm used to generate social graphs, and explain the design choices made in the various components of the simulation.

4.1 Overview

In order to test the proposed hypotheses we will run a discrete event simulation of the social network application described in the previous chapter. The system we simulate is agent-based, meaning we will simulate individual users, of whom the data is distributed over pods residing in a number of simulated servers. These users can perform three types of action: create posts, create comments and collect news feeds containing the most recent actions of their friends. Since the traffic generated by a user depends on the amount of connections to other users he or she has, as will be explained later, it is important to create social graphs that display to the characteristics laid out in section 2.2 as much as possible. In the next section we describe how such graphs are generated in our simulation.

4.2 Social Graph Generation

We repeat the most important properties of an online social network here for convenience:

1. Sparseness
2. Heavy-tailed degree distribution
3. Positive degree assortativity
4. Short average hop distance (or a neighborhood function that covers most of the graph in few hops)

5. High average local clustering coefficient

4.2.1 Algorithm

The algorithm employed to generate social graph structures that display aforementioned characteristics is designed to iteratively generate a *hierarchy of clusters*, where clusters are densely connected sets of nodes. The core idea is that, at each step of the algorithm, a set of clusters of nodes is generated according to input parameters. In the next iteration, these clusters themselves are treated as *nodes* that in turn get clustered.

An example of this hierarchy structure in the real world would be a school of children: children have two or three best friends in their classroom. These are small, very dense clusters, as all children in a group of best friends know one another very well. Of course, there is quite some overlap between these small clusters, forming a larger, less densely connected cluster, namely the class itself. Many children also get along with children from other classes in the same grade, forming an even larger cluster of classes, the entire grade. Finally, a few children even know kids in *other grades*, forming the largest cluster: the whole school.

In this example the highest level of the hierarchy is the school. The school itself is made up of a cluster of loosely connected grades. Grades are made up of a slightly stronger connected cluster of classes, classes are clusters made up of even more strongly connected friend groups, and these friend groups are small, extremely densely connected groups of children.

Here we explain how the algorithm operates by going through the input parameters and how they are used, together with a numerical example, of which the steps are shown in figure 4.1.

1. **Cluster sizes**: a list that holds the sizes of clusters at each level of the hierarchy, from lowest level to highest level.

   Example: the network generated by list [2, 3, 4] would contain

   - 2 top-level clusters, both consisting of:
   - 3 sub-clusters, which both consist of:
   - 4 users, since at the lowest level, nodes of the network are simply users.

   The total network size can thus be calculated by taking the product of the cluster sizes, in this case $2 \cdot 3 \cdot 4 = 24$. The resulting (not yet connected) cluster hierarchy is shown in figure 4.1a.

2. **Cluster densities**: a list that hold the *densities* of the clusters at each level of the hierarchy. By density we mean the amount of existing connec-

tions divided by the amount of possible connections in a cluster.

Example: Continuing with the network generated by cluster sizes $[2, 3, 4]$ from (1), we now assume the cluster densities $[1, \frac{1}{3}, \frac{2}{3}]$. This means that

- the top level of 2 clusters should have a density of 1. As the amount of possible connections is 1, this simple means the two clusters should be connected.
- One level down, we find that the clusters consisting of 3 nodes should have $\frac{1}{3}$ of all possible connections realized. The total amount of possible connections is 3, and $3 \cdot \frac{1}{3} = 1$, so one connection should be created.
- On the lowest level, a cluster density of $\frac{2}{3}$ is given, so $\frac{2}{3}$ of all 6 possible connections, resulting in 4 connections being made.

The created connections between clusters are shown in figure 4.1b, note that the connections between high-level clusters aren't resolved to user connection yet.

3. **Connection densities**: a list that holds the densities of connections between the clusters at each level of the hierarchy. In (2) we describe how clusters themselves can be connected. Of course, this operation is not yet well-defined, i.e. it does not specify whether *all* people belonging to a cluster should be connected to *all* people belonging to another cluster in case these two clusters are connected, or only a fraction thereof. The amount of connections that should be created between two clusters is given by the entries of this list, again as fractions of the total amount of possible connections.

   Example: We resume the example from (1) and (2), and add the input connection densities $[\frac{1}{48}, \frac{1}{8}, 1]$.

   - The two top clusters are connected, and both clusters contain 12 users. The total possible amount of connections is thus $12 \cdot 12 = 144$. the top level input connection density is $\frac{1}{48}$, so 3 users from one cluster should be connected to 3 users from the other cluster.
   - The next level has cluster density $\frac{1}{8}$, and the clusters on this level all contain 4 users, thus if two of those clusters are connected, this results in $4 \cdot 4 = 16$ possible connections, and thus $16 \cdot \frac{1}{8} = 2$ created connections.
   - On the lowest level, the concept of connection density doesn't make sense, since a connection between two individual users is binary: it either exists or it doesn't. Therefore the last number in the connection density list does not affect the algorithm. It is always interpreted as being 1.

In figure 4.1c we show the network in which connections between clusters have been resolved to connections between individual users. The connections pass through the hierarchy level where the original cluster connection was created.

An important thing to note is the way in which users are selected to form a connection: one possibility would be to *randomly* select users with a uniform probability, yet this would result in a normal degree distribution, and no significant degree assortativity.

Therefore we choose a different method: we still randomly select users, but with probabilities *proportional* to the amount of connections a user already has. The effect of this approach is twofold: on one hand, it causes highly connected users to become *even more connected*, ensuring a heavy-tailed degree distribution. On the other hand, the connection that is formed is likely to be between two highly connected users, thus creating a positive degree assortativity.
This algorithm is tested in chapter 5.

We also mention that in earlier iterations of the simulation the Havel-Hakimi algorithm was used to create graphs [32]. This algorithm generates a graph according to a given degree sequence (i.e. a sorted list of the degrees of all nodes). However, not all degree sequences are *graphical*, i.e. for some sequences of numbers there exist no graphs that have this sequence as degree sequence, which made this into a cumbersome manual search for graphical sequences of various sizes. Furthermore, there exists no intuitive way to control graph parameters like degree assortativity and clustering in this algorithm. The only control one has is over the degree sequence. These difficulties sparked the search for a new, more tuneable algorithm, which we have presented in this section.

4.2.2 User Activity

Another important aspect of the network is the *user activity rate*, i.e. how often a user performs certain actions such as creating posts and comments, and compiling their news feeds. In [23] a positive correlation is found between the amount of connections a user has, and how active this user is on the application. Intuitively this makes sense: The more friends one has one the network, the more new events there are to catch up on, thus creating an incentive to be more active on the network.

To achieve this effect, we simulate user activity through a Poisson process of which the rate parameter is proportional to the amount of connections a user has. Note that user activity on social networks most likely does *not* follow a Poisson distribution in reality. Yet, to the best of our knowledge, no work has been published on the what an actual user activity distribution looks like. Therefore we use the Poisson distribution as an approximation, since it is easily

generated in a discrete event simulation and allows for fine control of event occurrence rates.

4.2.3 User Action Handling

A Facebook benchmark study found that the server load in social networks is dominated by read operations [33]. Therefore we only simulate read requests at the servers; write operations are executed, i.e. posts and comments are created, but this happens *instantly* and does not create simulated server traffic.

The way in which the three possible user actions is handled is described here:

- **Posting:** creating a post simply creates a new file in the folder "posts" of the user performing the action. The content of this post is a string of a randomly chosen length. This action also pushes a new "WriteAction" event in the users event file. No server traffic is generated.
- **Commenting:** Creating a comment works analogous to creating a post, with the only difference that it a URI reference pointing to the newly created comment is appended to the file of the post or comment the comment has been created for. Again, this creates no load on the simulated servers.
- **Compiling news feeds**: This action is handled in a sequence of steps:
  1. The profiles of all connected users ("friends" in Facebook-speak) are collected from the servers where the data pods of the connections reside. From these profiles the URIs of the respective eventfiles are extracted.
  2. As soon as a profile arrives, the eventfile URI is extracted, and is collected next. These eventfiles contain the most recent events of users, pointed to by a source URI.
  3. As soon as an eventfile arrives, the post or comment pointed to by the event source URI is fetched for each event in the eventfiles.
  4. If the retrieved post or comment itself has comments, these are gathered next. This is an recursive process, in order to collect complete "threads".

  This sequence is also shown in the UML-diagram in figure 4.2

4.2.4 Server model

The data pods of users are stored on simulated servers. These servers are implemented as simple document-serving servers, and have the following fields:

- Request queue: in this queue pending requests are stored.

- Latency, expressed in seconds: this value models any latency a request might experience.
- Bandwidth, expressed in bytes per seconds.

Servers are implemented in such a way that the complete bandwidth is evenly distributed over *all* pending requests. Since we actually generate all user data, and thus create all files that can be requested from the server, we simply inspect these generated RDF-documents to find the size of a requested document in the simulation.

Data pods are stored and queried according to the *linked data documents* paradigm, the server thus stores triples regarding a certain URI in the file pointed to by that URI. The server itself does not query the documents, it only serves them to clients.

4.3 Discrete Event Simulation

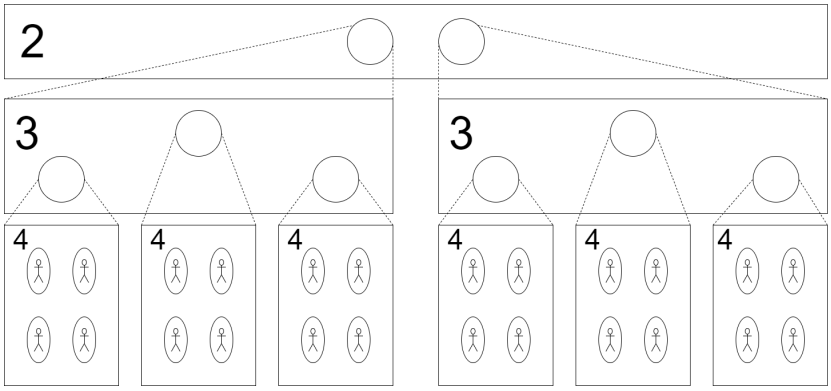
The discrete event simulation consists of the following steps:

1. A social graph is generated according to input parameters.
2. For each user in this social graph, the corresponding files are created (cfr. figure 3.2).
3. A set of servers is generated according to input parameters.
4. The individual users are equally distributed over the servers. (In future work unequal distribution might be interesting to investigate.)
5. The simulation event queue is initialized with the three possible user actions for each user, i.e. posting, commenting and compiling a *news feed*, as described in chapter 3.
6. The simulation warms up by filling up the folders of users with a number of posts and comments. The content of the posts is dummy text of variable length.
7. The simulation iterates over the event queue. The way in which actions are handled is described earlier in this chapter. Whenever a user action has been encountered, the *same* action for the *same* user is appended to the event queue at an exponentially distributed time increment to create the effect of a Poisson process, as described in section 2.3. The time between arrival and completion of a request at the simulated servers is recorded in log files.
8. the simulation runs until the desired simulated time has passed.

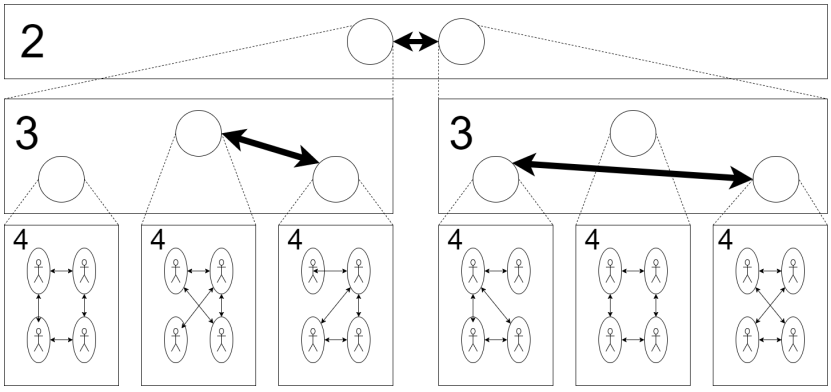
4.4 Implementation

The network generator and the discrete event simulation have been written in Javascript using the Node.js framework, and have been implemented from the ground up. For manipulating RDF triples, the rdflib.js[1] library has been used. Analysis of the results have been done in Python using the Pandas and Networkx libraries.

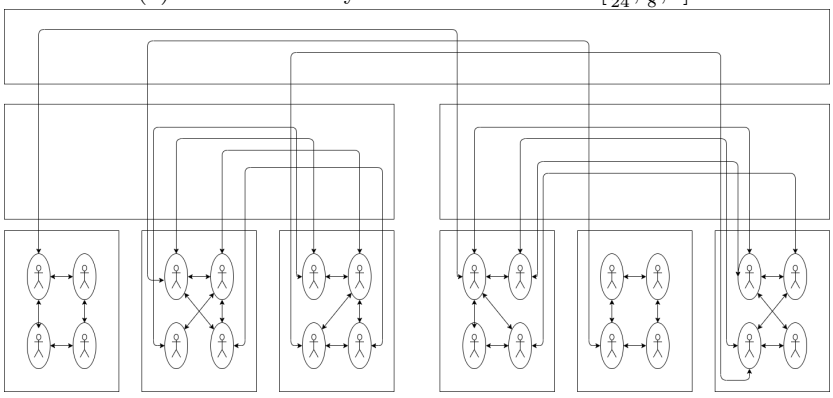
[1] https://linkeddata.github.io/rdflib.js/doc/



(a) Unconnected cluster hierarchy with cluster sizes [2, 3, 4]



(b) cluster hierarchy with cluster densities $[\frac{1}{24}, \frac{1}{8}, 1]$.



(c) cluster hierarchy with connection densities $[\frac{1}{48}, \frac{1}{8}, 1]$.

Figure 4.1: example of hierarchy clustering

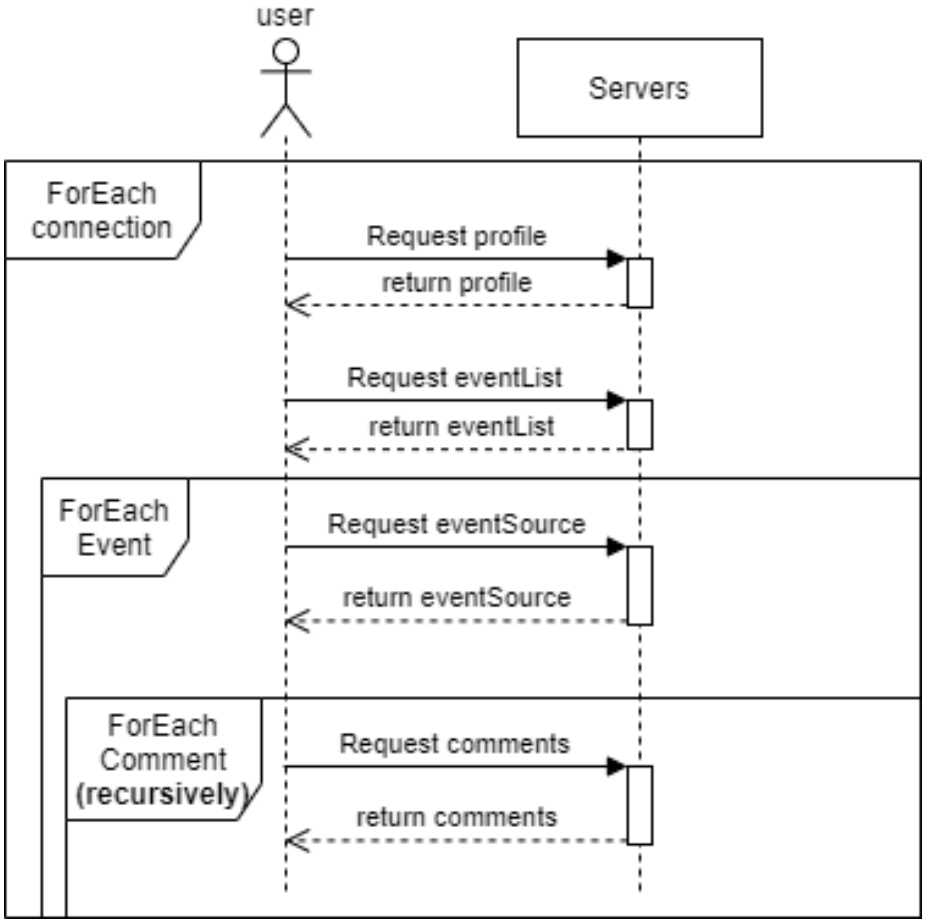


Figure 4.2: UML-diagram representing the process of compiling a news feed

Chapter 5

Experiments

In this chapter we present the experiments we have conducted to assess the validity of the network generation algorithm, and to test the hypotheses formulated in chapter 1.

5.1 General Notes

The main limiting factor when running these experiments is time: the inherent sequential nature of a discrete event simulation doesn't allow for parallelization, and since we actually generate all user files, a lot of slow IO-operations are required. The implementation of this simulation has been an iterative process throughout which improvements in runtime have been made, such as avoiding sorting the *event list* of the DES as much as possible, creating separate event lists per server (which are itself easier to keep sorted), since sorting lists accounts for a large part of the computational cost, and keeping a a JSON-object to link posts URIs to any comment URIs that they might link to, to avoid having to extract this information from the RDF-files themselves. Despite these efforts, the experiments are still conducted with small population sizes, as the runtime scales exponentially with the number of users in the simulated network.

Furthermore we would like to note that these experiments do not serve to make predictions about real-world systems in terms of absolute numbers; too many assumptions and simplifications are made in the server model, the application model, and the network model to accurately predict real server response times. The main focus lies on modeling how the results *change* under changing configurations and circumstances.

5.2 Assumptions

In this section we repeat the assumptions made in the following experiments:

- User activity follows a Poisson process with a rate that is proportional to the amount of connections a user has. We assume that per extra connection, the frequency of collecting a news feed goes up by a certain fixed amount.
- Servers distribute bandwidth equally over all pending requests. This implies that if the amount of requests in a server queue changes, the time to completion for all requests needs to be updated because of the changing bandwidth.
- Users are distributed equally over the servers.
- Network *latency*, i.e. the time it takes for data to travel from client to server and vice versa, is equal for all requests
- Servers do not compress files.
- Clients do not cache files.
- Servers do not experience downtime.
- Server traffic is dominated by read request, thus write request are assumed to happen instantly, without going through the server queue. This choice is made to keep the simulation tractable in terms of complexity and runtime duration.
- Recursion of requested comments is limited in depth. As described in earlier chapters, comments themselves can have comments, creating a nested structure. When collecting a news feed, users will not gather comments that are nested beyond a certain depth. This is copied from social network sites like Reddit.

5.3 Experiment 1: Network Generator Validation

5.3.1 Set-up

To ensure that the algorithm proposed in chapter 4 creates networks with the desired properties, we generate a sample set of 50 networks with empirically determined inputs, and compare them to a baseline of 50 randomly generated networks with the same amount of connections, and to properties of the Facebook social graph, analyzed in [23]. We compare the networks with respect to degree assortativity, neighborhood function, local clustering coefficient in function of degree, and degree distribution. Analysis is done in Python using the Networkx library.

5.3.2 Parameters

The networks are generated with the parameters shown in table 5.1

Parameter	Value
Cluster sizes	[8, 5, 4, 3] (network size = 480)
Cluster densities	[0.4, 0.8, 0.9, 1]
Connection densities	[0.1, 0.5, 0.9, 1]

Table 5.1: Parameters of experiment 1

5.3.3 Results

The results of this experiment are shown in figure 5.1.

- Figure 5.1a shows that our algorithm outperforms random networks in terms of degree assortativity, i.e. the correlation between the degrees of connected nodes. The algorithm however does not reach the assortativity of a real network.

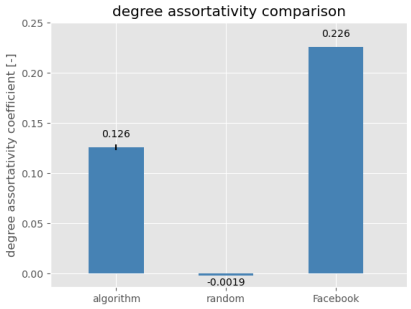
- Figure 5.1b compares the neighborhood functions, i.e. it plots the fraction of nodes that can be reached in function of the amount of hops taken, averaged out over all nodes as starting points. Our algorithm outperforms the random networks, albeit only slightly, and the generated networks are noticeably stronger connected than the Facebook graph, as it takes less hops to reach the whole graph. The fact that our algorithm never completely reaches a value of 1 indicates that some algorithmic networks consist of multiple disjoint subgraphs. This is also the case for the Facebook social graph, but 99.91 percent of users belong to the same subgraph, so that the effects are less noticeable. This disjointness however does not have a consequence for simulations.

- In figure 5.1c we plot the average local clustering coefficient in function of node degree. Note the logarithmic axes. We can again see that out algorithm outperforms random networks in terms of clustering, as clustering is higher for every degree. Compared to Facebook, our algorithm clusters too strongly for some degrees, while for others, mainly smaller degrees, it does not cluster enough. The steep drop-off at $degree = 100$ is caused by the fact that our generated network is only 480 nodes large, and thus doesn't have nodes with a degree over 100. Facebook enforced a maximum of 5000 connections at the time [23] was published.

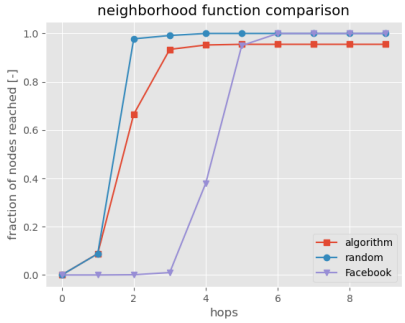
- In figure 5.1d we compare the degree distributions. Note the logarithmic axes. Again, our algorithm generates networks with heavier tails than random networks, yet still diverges significantly from a real social network. The most important difference is that our algorithm doesn't generate enough nodes with low degrees. Again, the sudden drop-off at 100 is caused by the smaller size of our generated network.

5.3.4 Conclusion

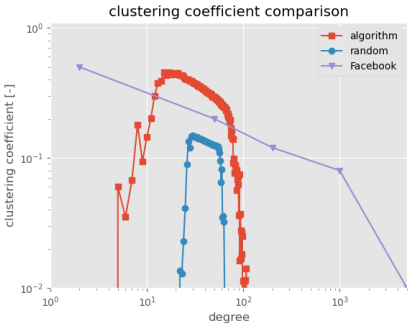
For this experiment we conclude that our algorithm outperforms random networks, yet still does not fully capture the characteristics of a real social network like Facebook. Furthermore, tuning the input parameters of the algorithm is far from easy, as the characteristics are not orthogonal; e.g. shifting the degree distribution to include more nodes of low degree directly impacts clustering, etc. Designing an algorithm that does capture all characteristics perfectly could likely fill up a thesis of it's own, and is not the main focus of this dissertation. We make a trade-off between accuracy and complexity by opting for this network generation algorithm.



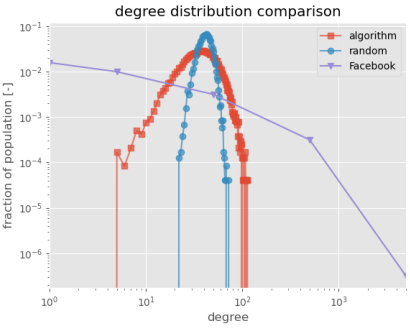
(a) degree assortativity coefficient comparison



(b) neighborhood function comparison



(c) clustering coefficient in function of degree comparison



(d) degree distribution comparison of algorithmically generated networks

Figure 5.1: Comparison of characteristics between algorithmically generated networks, randomly generated networks, and the Facebook social graph

5.4 Experiment 2: Decentralization with constant bandwidth

5.4.1 Set-up

In this experiment we aim to test hypothesis 2, which we repeat here:

The average time to complete user operations will increase with increasing decentralization of data, keeping bandwidth constant in the network.

The reasoning behind this hypothesis is as follows:

1. Because of the nature of linked data, actions such as compiling news feeds are executed *sequentially* per connected user, as described in chapter 4, (i.e. for one connection, first the profile is requested, *then* the eventfiles, *then* the posts and comments, because the URIs of later requests need to be extracted from previous requests. Note that the requests constituting these sequential steps *are* executed in parallel, e.g. all requests for the event sources of a user are issued at the same time). Consequently, if any of these intermediate steps is delayed, the entire action takes longer.

2. The files of "popular", i.e. well-connected users, are requested significantly more often than those of less connected users, since well-connected users not only have more connections, but are connected to well-connected - and thus more active - users.

3. When collecting many users in large servers with corresponding bandwidth, this popularity-factor is averaged out; a server will likely contain a mix of pods of highly connected and less connected users.

4. By distributing users over more servers, there are less user *per* server and as such less averaging over popularity, creating a group of popular servers which are often "congested", and as such delay the operations of all users needing files from them.

In order to prove or disprove this hypothesis, we run a series of simulations, in which we distribute a fixed amount of users over an increasing amount of servers, while keeping the total bandwidth in the system constant, by distributing it equally over the servers.

5.4.2 Parameters

The parameters for this experiment are given in table 5.2.

type	parameter	value(s)
fixed	network size	120 users
	simulated period	1 day
	average news feed collection rate	14.35 times/day
	average post creation rate	2 times/day
	average comment creation rate	4 times/day
	max. comment recursion depth	3
	max. number of events in user eventfiles	5
variable	number of servers	1, 5, 10, 50, 100
	bandwidth per server	1 Gbps/(number of servers)

Table 5.2: Parameters of experiment 2

We run this experiment 5 times and average out the results, to account for possible variability due to the network structure and the randomly generated intervals between events.

5.4.3 Results

The results are shown in figure 5.2. Note that there are error bars drawn to show the variance between the 5 runs of this experiment, but the variance is very small.

- Figure 5.2a shows the total amount of requested data throughout the simulation. This is measured to ensure that any increase in server response times is due to decentralization and not due to an increased load on the servers.
  It is clear that this number remains approximately constant for all simulations, as is expected.

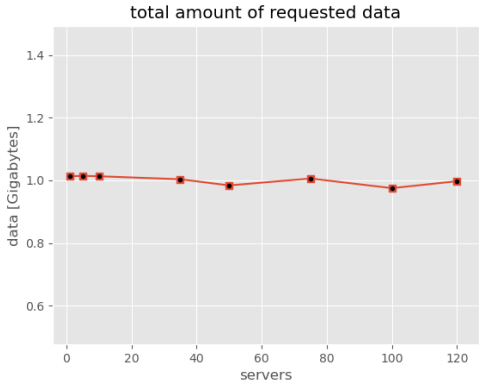
- Figure 5.2b shows the average time between arrival and completion of an individual request at the server.
  We can see that the server response time increases with increased decentralization of the users. The increase starts out steep, but slows down for larger numbers of servers, likely this is due to the "popularity" effect, as described above, leveling out at a higher level of decentralization. The time for resolving individual request increases significantly, with a factor of approximately 7.25.

- Figure 5.2c shows the average time between the start of the first request and finish of last request of collecting a news feed.

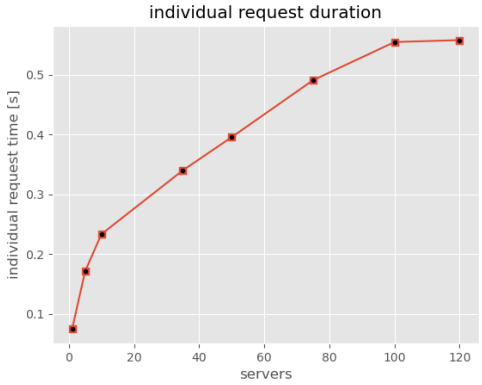
The time to collect news feeds also increases, yet with a much lower factor than the individual requests of only 1.23. This is due to the fact that, although the different steps of news feed gathering are sequential, many of the requests for content that constitute these steps are issued in parallel, as shown in figure 4.2, thus reducing the effects on total news feed collection time.

5.4.4 Conclusion

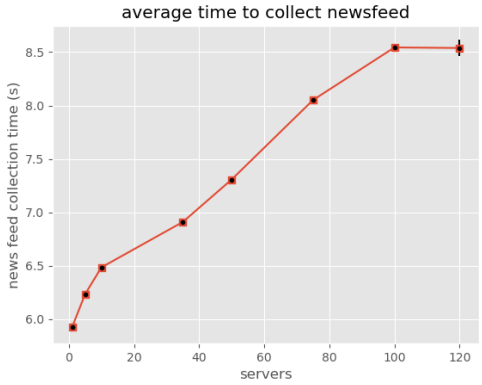
The results from this experiment seem to suggest that hypothesis 2 is valid: the time to collect a news feed does increase due to decentralization, since no other parameters were changes in this experiments, only the amount of servers and their respective bandwidths.



(a) Total amount of requested data vs. amount of servers



(b) Average time between start and completion of an individual request



(c) Average time between start and completion of gathering a news feed

Figure 5.2: Results of experiment 2

5.5 Experiment 3: Increasing User Activity

5.5.1 Set-up

In this experiment we aim to test hypothesis 4, which we repeat here:

The average time to complete user operations will increase with increasing frequency of these operations in the network

We note that in our simulation, the only type of action that generates traffic at the servers is the news feed collecting action, as explained earlier.

5.5.2 Parameters

The parameters for this experiment are given in table 5.3.

type	parameter	value(s)
fixed	network size	120 users
	simulated period	1 day
	average post creation rate	2 times/day
	average post creation rate	2 times/day
	average comment creation rate	4 times/day
	max. comment recursion depth	3
	max. number events in user eventfiles	5
	number of servers	10
	bandwidth per server	0.1 Gbps
variable	average news feed collection rate	2.9, 28.7, 57.5, 143.5, 215.4 times/day

Table 5.3: Parameters of experiment 3

We run this experiment 5 times and average out the results, to account for possible variability due to the network structure and the randomly generated intervals between events.

5.5.3 Results

The results of this experiment are shown in figure 5.3.

- Figure 5.3a shows the total amount of requested data throughout the simulation. The load on the servers clearly increases linearly with increasing frequency of news feed collection.
- Figure 5.3b shows the average time between arrival and completion of an individual request at the server.

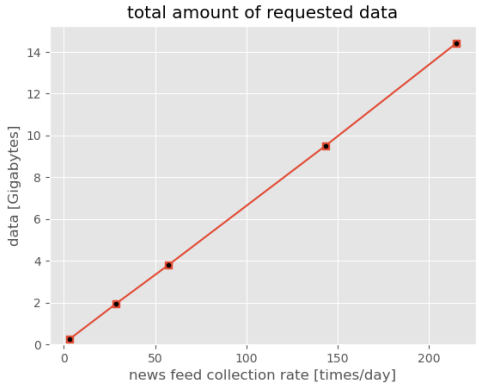
We see that this number rises, as expected, but less severely than the increase in the previous experiment: an increase in request frequency a factor 200 only increases the server response time by a factor of $\approx 30\%$.

- Figure 5.3c shows the average time between the start of the first request and finish of last request of collecting a news feed.

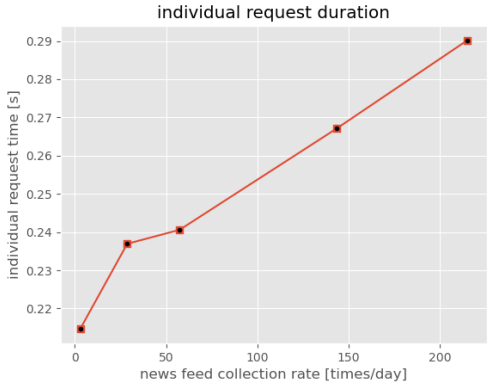
  The time it takes to collect a news feed also increases, but - as was the case in experiment 2 - with a lower factor than the individual requests of around 1.04, due to the parallel issuing of many requests.

5.5.4 Conclusion

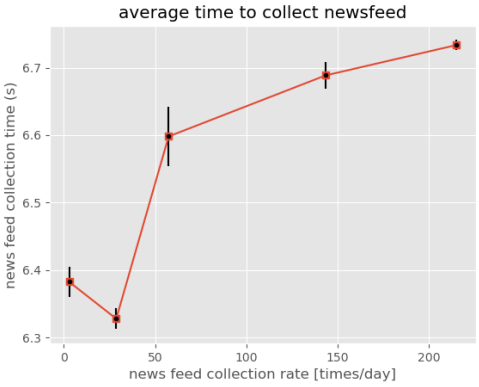
This experiment seems to indicate that the proposed hypothesis is valid; increasing the user frequency does increase the time it takes to collect a news feed, albeit not by a large factor.



(a) Total amount of requested data vs. news feed request frequency



(b) Average time between start and completion of an individual request



(c) Average time between start and completion of gathering a news feed

Figure 5.3: Results of experiment 3

5.6 Experiment 4: Increasing Network Size with Self-Hosting Users

5.6.1 Set-up

In this experiment we aim to test hypothesis 3, which we repeat here:

The average time to complete user operations will not increase significantly with increasing network size, when each user hosts it's own pod.

This hypothesis is interesting with regard to Solid, since it's possible for users is to host their own data pods on the platform. In this experiment we test whether self-hosting a server is a scalable approach with increasing network size. In this experiment we assume that users host servers with a lower bandwidth of 0.01 Gbps.

5.6.2 Parameters

The parameters for this experiment are shown in table 5.4

type	parameter	value(s)
fixed	simulated period	1 day
	average post creation rate	2 times/day
	average comment creation rate	4 times/day
	max. comment recursion depth	3
	max. number of user events	5
	bandwidth per server	0.1 Gbps
variable	network size	60, 120, 180, 240 users
	number of servers	equal to network size

Table 5.4: Parameters of experiment 4

Again we run the experiment 5 times, and average out over the results.

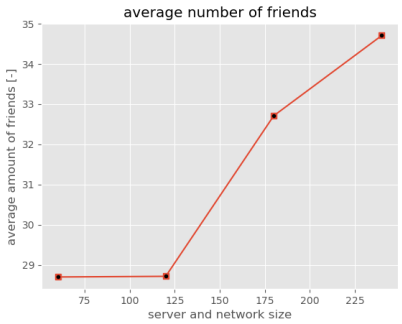
5.6.3 Results

First off all we note that, by changing the network size, we unavoidably also slightly change some graph properties. Most importantly, we see in figure 5.4a that the average amount of friends a user goes up slightly. As the user activity rate is based on the amount of friends a user has, this number also goes up, as shown in figure 5.4b. This implies that the load on the servers will increase slightly faster than linear with increasing network size, as can be seen in figure 5.4c.

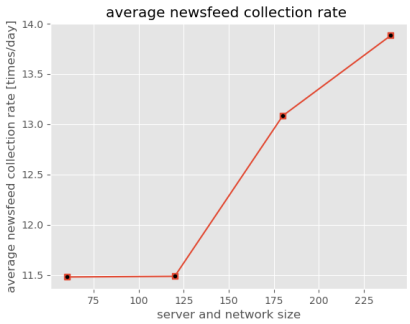
In figure 5.5a we see that the time to resolve an individual request at the servers barely changes, from 0.465 to 0.476 seconds with increasing network size, i.e. an increase of 2% . Consequently, the time to collect an entire news feed also only changes slightly in figure 5.5b, from 7.75 to 8.15 seconds, although at a higher percentage than the individual request times, by 5%. This is likely due to the increased average amount of friends, since users need to collect more files on average per news feed collection.

5.6.4 Conclusion

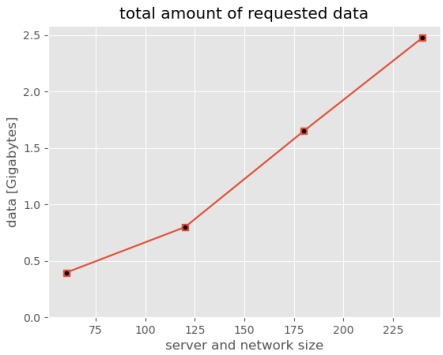
This experiment seems to validate hypothesis 4: scaling up a network of self-hosting users doesn't seem to significantly impact the time it takes collect news feeds, and as such self-hosting appears to be a viable approach for a Solid-like platform.



(a) Average amount of connections

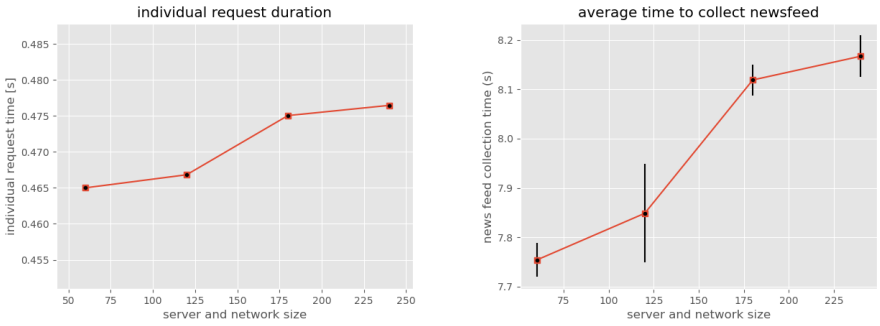


(b) Average news feed request rate



(c) Total amount of data requested throughout simulation

Figure 5.4: network characteristics and server load of experiment 4



(a) Average time per individual request (b) Average news feed collection time

Figure 5.5: Results of experiment 4

Chapter 6

Discussion

The conducted experiments validate the proposed hypotheses in as far as our simulation is able to capture the essence of a real-life social network application. The simplifications made in the experiment set-ups and small network sizes might affect numerical results, yet the general trends are expected to stay valid.

The first experiment in section 5.3 shows that the proposed network generation algorithm is a viable approach to create social graph structures, yet more research in optimal parameters is needed to truly mimic networks like Facebook.

The second experiment in section 5.4 shows that decentralization of data increases the time it takes to complete actions if bandwidth remains constant. The extrapolated implication of this finding is that a decentralized network will require *more* bandwidth to offer the same response times when compared to a centralized system. In this simulation users and servers acted in a very naive way, i.e. for each document a separate request was issued, even if all requested documents resided in the same server. A system that *does* take advantage of partially centralized data would likely offer even better response times.

The third experiment in section 5.5 shows that the time to execute user actions increases with increased frequency of those actions, albeit not by much. The (obvious) implication for pod providers is that they should be prepared to scale up bandwidth when Solid were to suddenly become a popular platform.

The final experiment shown in section 5.6 indicates that self-hosting is a valid strategy for a decentralized network in terms of time to complete user actions. If a Solid-like network were to use self-hosting as a selling point and emphasize the complete data control that is created by storing the data *physically* in the homes of users, a sudden increase in popularity wouldn't affect the server response times as much as in a system that relies on pod-providers that need to scale up with the increased network traffic, as shown in experiment 2 and 3.

Chapter 7

Conclusion

In this dissertation we have laid out a hypothetical decentralized social network application, inspired by the Solid platform. We implemented this social network application into a discrete event simulation, combined with an algorithm to generate realistic social graph structures, to test such a decentralized system under various configurations and circumstances regarding network size, number of servers and user activity.

We found that basic social network functionalities can be mapped onto decentralized networks, but that there are still features that are more easily achieved in a centralized system, such as looking up specific content, which require smart querying tactics in a decentralized system.

The proposed network generating algorithm was validated against a random baseline and the Facebook social graph. It was shown that the algorithm is a viable approach, yet more research is needed to determine optimal parameters to completely capture the characteristics of an online social graph.

The proposed discrete event simulation was used to test a number of set-ups: varying network size with constant total bandwidth, varying user activity rates and varying network size with self-hosting users. We showed that for the first two experiments, the time it takes to complete user actions goes up, while for the last experiment this time remained constant. These findings confirmed our hypotheses, albeit for networks of small sizes.

Chapter 8

Future Work

8.1 Expanding Functionality

The application simulated in this work only implements very basic features such as creating posts and comments, and compiling news feeds. Future work could expand upon this set of features to mimic social networks more closely. Features such as uploading images, uploading videos, chatting with other users, pages for events and organizations attracting, following profiles instead of bidirectional friendships, etc. could all be added.

8.2 Experimenting With Load Distribution

The generated network traffic was assumed to be static over time in this thesis. In future work it would be interesting to see how decentralized networks withstand sudden surges in traffic, for example when a piece of content goes "viral".

8.3 Querying Social Networks

One of the features that a decentralized social network does not offer is a straightforward way to *look up* specific profiles or content on social media. It would be interesting to investigate ways to make this possible, for example through data summaries and link traversal query execution.

8.4 Social Application Benchmark

In this work we opted to simulate simple servers in a discrete event simulation. As a general remark on that approach, it was found that discrete event simulations - although allowing for fine control simulation parameters, and giving

clear insight in the simulated system - is costly in terms of runtime, and it might be worthwhile to investigate alternative simulation methods, such as simulating the network with *real* servers. The work presented in this dissertation could be used as a benchmark generator, that generates the load under which the servers operate.

8.5 Facilitate self-hosting in Solid

In experiment 4 we show that the approach of self-hosting pods with small servers scales well with a growing network size. Self-hosting of pods is possible on the Solid platform, yet requires a prohibitive amount of technological knowledge to be accessible to a large public[1]. It would be interesting to investigate the development of physical *plug-and-play* data pods that anyone could easily set up at home. This could also be a selling point of the Solid platform: anecdotal evidence of the author proves that it's easier to explain to a layman that his data is on a pod he has at home, than the subtle difference between what server ("what's a server?") his data resides on.

[1] https://solidproject.org/use-solid/

Bibliography

[1] C. Arthur, "Tech giants may be huge, but nothing matches big data," *The Guardian*, aug 2013. [Online]. Available: https://www.theguardian.com/technology/2013/aug/23/tech-giants-data

[2] H. R. Max Roser and E. Ortiz-Ospina, "Internet," *Our World in Data*, 2015, https://ourworldindata.org/internet.

[3] B. Burrough, S. Ellison, and S. Andrews, "The Snowden Saga: A shadowland of secrets and light," *Vanity Fair*, apr 2014. [Online]. Available: https://www.vanityfair.com/news/politics/2014/05/edward-snowden-politics-interview

[4] N. Confessore, "Cambridge Analytica and Facebook: The Scandal and the Fallout So Far," *The New York Times*, apr 2018. [Online]. Available: https://www.nytimes.com/2018/04/04/us/politics/cambridge-analytica-scandal-fallout.html

[5] European Commission, "Data protection," Tech. Rep. [Online]. Available: https://ec.europa.eu/info/law/law-topic/data-protection_en

[6] A. V. Sambra, E. Mansour, S. Hawke, M. Zereba, N. Greco, A. Ghanem, D. Zagidulin, A. Aboulnaga, and T. Berners-Lee, "Solid: A Platform for Decentralized Social Applications Based on Linked Data," Tech. Rep., 2016.

[7] R. Verborgh, "Paradigm shifts for the decentralized Web," Tech. Rep., dec 2017. [Online]. Available: https://ruben.verborgh.org/blog/2017/12/20/paradigm-shifts-for-the-decentralized-web/#apps-become-views

[8] O. Hartig and J.-C. Freytag, "Foundations of traversal based query execution over linked data," *Proceedings of 23rd ACM Conference on Hypertext and Social Media*, 05 2012.

[9] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," vol. 284, pp. 34–43, 2001.

[10] T. Berners-Lee, "Linked Data - Design Issues," 2006. [Online]. Available: https://www.w3.org/DesignIssues/LinkedData.html

[11] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data - The story so far," *International Journal on Semantic Web and Information Systems*, vol. 5, no. 3, pp. 1–22, jul 2009.

[12] G. Kobilarov, T. Scott, Y. Raimond, S. Oliver, C. Sizemore, M. Smethurst, C. Bizer, and R. Lee, "Media meets semantic web - How the bbc uses dbpedia and linked data to make connections," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5554 LNCS. Springer, Berlin, Heidelberg, 2009, pp. 723–737.

[13] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "DBpedia: A nucleus for a Web of open data," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4825 LNCS. Springer, Berlin, Heidelberg, nov 2007, pp. 722–735. [Online]. Available: http://www.mediawiki.org

[14] R. Cyganiak, D. Wood, and M. Lanthaler, "RDF 1.1 Concepts and Abstract Syntax," W3C, Tech. Rep., 2014. [Online]. Available: http://www.w3.org/TR/rdf11-concepts/

[15] David Wood, Marsha Zaidman, Luke Ruth, Michael Hausenblas, *Linked Data, structured data on the web.* Manning, 2014.

[16] W3C, "Ontologies - W3C," W3C, Tech. Rep., 2012. [Online]. Available: http://www.w3.org/standards/semanticweb/ontology.html

[17] K. Goel and P. Gupta, "Official Google Webmaster Central Blog: Introducing schema.org: Search engines come together for a richer web," Google, Tech. Rep., 2011. [Online]. Available: http://googlewebmastercentral.blogspot.com/2011/06/introducing-schemaorg-search-engines.html

[18] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert, "Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web," *Journal of Web Semantics*, pp. 37–38, 2016.

[19] S. Harris and A. Seaborne, "SPARQL 1.1 Query Language," W3C, Tech. Rep., mar 2013. [Online]. Available: https://www.w3.org/TR/sparql11-query/

[20] C. Buil-Aranda, A. Hogan, J. Umbrich, and P. Y. Vandenbussche, "SPARQL web-querying infrastructure: Ready for action?" in *Lecture*

Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 8219 LNCS, no. PART 2. Springer, Berlin, Heidelberg, oct 2013, pp. 277–293.

[21] A. Sambra, H. Story, and T. Berners-Lee, "WebID 1.0," W3C, Tech. Rep., mar 2014. [Online]. Available: https://www.w3.org/2005/Incubator/webid/spec/identity/

[22] S. Speicher, J. Arwe, and A. Malhotra, "Linked Data Platform 1.0," W3C, Tech. Rep., feb 2015. [Online]. Available: https://www.w3.org/TR/ldp

[23] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow, "The anatomy of the facebook social graph," *arXiv preprint*, vol. 1111.4503, 11 2011.

[24] D. Boyd and N. Ellison, "Social Network Sites: Definition, History, and Scholarship," *Journal of Computer-Mediated Communication*, vol. 13, no. 1, pp. 210–230, oct 2007.

[25] S. Milgram, "The Small World Problem," *Psychology Today*, 1967.

[26] L. H. Gilbert and Nigel, "Social Circles: A Simple Structure for Agent-Based Social Network Models," *Journal of Artificial Societies and Social Simulation*, mar 2009.

[27] A. A. Tako and S. Robinson, "Model development in discrete-event simulation and system dynamics: An empirical study of expert modellers," *European Journal of Operational Research*, vol. 207, no. 2, pp. 784–794, dec 2010.

[28] J. Misra, "Distributed Discrete-Event Simulation," *ACM Computing Surveys (CSUR)*, vol. 18, no. 1, pp. 39–65, mar 1986.

[29] L. Leemis, "Input modeling techniques for discrete-event simulations," in *Winter Simulation Conference Proceedings*, vol. 1, 2001, pp. 62–73.

[30] T. Heath and C. Bizer, "Linked Data: Evolving the Web into a Global Data Space," *Synthesis Lectures on the Semantic Web: Theory and Technology*, vol. 1, no. 1, pp. 1–136, feb 2011.

[31] J. Umbrich, K. Hose, M. Karnstedt, A. Harth, and A. Polleres, "Comparing data summaries for processing live queries over Linked Data," *World Wide Web*, vol. 14, no. 5, pp. 495–544, oct 2011. [Online]. Available: https://link.springer.com/article/10.1007/s11280-010-0107-z

[32] S. L. Hakimi, "On realizability of a set of integers as degrees of the vertices of a linear graph ii. uniqueness," *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, no. 1, pp. 135–147, 1963. [Online]. Available: http://www.jstor.org/stable/2098770

[33] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "LinkBench: a Database Benchmark Based on the Facebook Social Graph," Facebook, Tech. Rep., 2013. [Online]. Available: http://github.com/facebook/linkbench