Developing Peer-To-Peer Web Applications

Toni Ruottu

Helsinki September 19, 2010

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Laitos — Institution — Department Department of Computer Science

Tekijä — Författare — Author Toni Ruottu
Työn nimi — Arbetets titel — Title Developing Peer-To-Peer Web Applications
Oppiaine — Läroämne — Subject Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year September 19, 2010	Sivumäärä — Sidoantal — Number of pages 55 pages + 2 appendices

Tiivistelmä — Referat — Abstract

As the virtual world grows more complex, finding a standard way for storing data becomes increasingly important. Ideally, each data item would be brought into the computer system only once. References for data items need to be cryptographically verifiable, so the data can maintain its identity while being passed around. This way there will be only one copy of the users family photo album, while the user can use multiple tools to show or manipulate the album. Copies of users data could be stored on some of his family members computer, some of his computers, but also at some online services which he uses. When all actors operate over one replicated copy of the data, the system automatically avoids a single point of failure. Thus the data will not disappear with one computer breaking, or one service provider going out of business. One shared copy also makes it possible to delete a piece of data from all systems at once, on users request.

In our research we tried to find a model that would make data manageable to users, and make it possible to have the same data stored at various locations. We studied three systems, Persona, Freenet, and GNUnet, that suggest different models for protecting user data. The main application areas of the systems studied include securing online social networks, providing anonymous web, and preventing censorship in file-sharing. Each of the systems studied store user data on machines belonging to third parties. The systems differ in measures they take to protect their users from data loss, forged information, censorship, and being monitored. All of the systems use cryptography to secure names used for the content, and to protect the data from outsiders.

Based on the gained knowledge, we built a prototype platform called Peerscape, which stores user data in a synchronized, protected database. Data items themselves are protected with cryptography against forgery, but not encrypted as the focus has been disseminating the data directly among family and friends instead of letting third parties store the information. We turned the synchronizing database into peer-to-peer web by revealing its contents through an integrated http server. The REST-like http API supports development of applications in javascript.

To evaluate the platform's suitability for application development we wrote some simple applications, including a public chat room, bittorrent site, and a flower growing game. During our early tests we came to the conclusion that using the platform for simple applications works well. As web standards develop further, writing applications for the platform should become easier. Any system this complex will have its problems, and we are not expecting our platform to replace the existing web, but are fairly impressed with the results and consider our work important from the perspective of managing user data.

ACM Computing Classification System (CCS):
D.2.11 [Software Architectures]

Avainsanat — Nyckelord — Keywords
content centric, peer-to-peer, browser, javascript

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

– When you stare up into the snowfall, what do you see?
– I see moments.
– Memories from across my life, floating in the dark.
– And do you see a story, Joseph?
– I'm looking for one.

from the award winning computer game Spectre, by Vaguely Spectacular

Acknowledgements

I would like to thank the European Union, for funding a large part of the research that lead into development of Peerscape. I owe a great deal to GNUnet developers, who introduced me to the academic side of peer-to-peer systems. I am standing on the shoulders of my fellow Peerscape developers, who did a large part of the work I am presenting in my thesis. I would specially like to thank Ken Rimey, who gave me the opportunity to work in the project, but also taught me half of the computer science I know. My everyday work would not have been the same without my HIIT roommates Harri Hämäläinen, Ville Mäntysaari, Jani Turunen, and Konrad Markus. I am also grateful towards many other HIIT researchers, who were always happy to comment on the timely puzzle. While working on the prototype we received lots of useful feedback from the Aalto University, School of Art and Design, Media Laboratories, who used Peerscape as a foundation for their own research prototypes. As for the thesis itself, I'd like to thank professors Jussi Kangasharju and Sasu Tarkoma for supervising the work. Last, but not least, I would want to thank my family and friends, who helped me maintain my mental health by interrupting my work during public holidays and such.

I would also like to clarify my contribution to Peerscape platform a bit. I participated in the design of Peerscape from the beginning. Most of the work done within the project was built as collaborations of multiple researchers. In addition to general testing of the platform, I took part in developing a chat application prototype, a file-sharing application prototype, and some of the developer tools used for storing files into the system. I also built a simple game prototype on my own.

Some parts highlighted in my thesis were written by my co-workers alone. These include the webdav hooks, written by Ville Mäntysaari, and the ftp front end, written by Konrad Markus.

Contents

1 Introduction

In colloquial language the word *internet*, referring to the world wide web, is often used as a place. "We haven't been on the Internet lately", and thus we have not been able to check our email. Urls are thought to be names for information. We surf to http://www.smart.com/ because the advertisement on the bus said we could "go there and receive more information on a car that fits in any parking space imaginable", and visit Facebook to look at our family photographs. Most of the time this transparency is unbreakable. In reality however, the Internet is run by a bunch of companies who only care about storing our family photographs until they close their doors. When this happens worlds collide, and users find out in an unpleasant way how things really are. The problem can be mitigated by making computer systems, owned by the involved companies and users, understand they are all working around copies of the same data.

Our research is all about abstracting location of content. Making content itself addressable removes the coupling between information and the computers hosting it. If we can talk about the information directly, it becomes possible to retrieve it from anyone having a copy. This means that we are no longer dependent on any company that might close its doors. Thus, in the future, Smart drivers may be able to retrieve the service manual directly from one another even if Smart moves on to the field of motor sports, and stop having an interest in the consumer cars they used to produce. Addressing content should also make it easier for us to have straight forward copies of our digital life on multiple devices we own, instead of making backup copies of files we'd rather just have a backup computer.

Getting user computers involved in the Internet is another goal we are chasing. An example would include storing family photographs, not only on computers owned by some company, but also on computers of family members. In the current model family members may already store copies of their photographs on some of their laptops or home computers. Thus some of the more important photographs may survive the case of a photograph site going out of business. This however does not fully remove the problem of securing a digital life, as the family members may have used a web site to organize their photographs. When the site closes its doors, they have to find a new site, recollect the photographs from local computers, and add the same metadata again.

To gain understanding over the subject, we study some existing schemes of naming and protecting user data. These systems were designed to protect users in everyday use cases like browsing the web, sharing files, or using online social networks, such as Facebook [fac10]. The systems have their go at protecting users from data loss, forged information, censorship, and being monitored by some third party. Incentives and commitment from people and companies involved in sharing and dissemination of the data online plays an important role when considering such trust questions. Ensuring low cost for development and maintenance of the systems, and the requirement for a high quality user experience, add to the challenge. Nevertheless they remain essential for wide adoption of such a platform.

To experiment with some related ideas, we go on to build a prototype of our own. The result, a software platform called Peerscape, allows web site developers to create web applications that do not require any infrastructure run by a company. It works entirely on computers operated by its users, except for a general peer discovery service run by a federated group of infrastructure providers. While developing the platform, we tried to keep the user interface easy enough for an average grandparent. Most importantly, we ended up tying content to people, so we could help people locate interesting content, and do simple access control management.

To try out the platform we developed a few example applications for it. Before we really got into developing applications we needed some tools for injecting files from file system into our platform. As our first actual application, we developed a public chat room where users exchange real-time textual messages with one another. Then we developed a file sharing site for sharing files and attaching meta data to shared files. We also developed a simple flower growing game to see, if developing games for the platform was possible. Finally, we spent some time analysing different opportunities for building a multi-user text area.

The rest of the thesis is organized as follows. In Section 2 we take a closer look at the concepts of content centric and peer-to-peer networking. In section 3 we explore some previously existing systems that attempt to protect user data. In section 4 we introduce Peerscape, the platform prototype we built ourselves. In section 5 we discus some of the applications we wrote to validate our platform. Finally, section 6 concludes by looking back at the work done and discussing future research directions.

2 Background

When trying to design future networking protocols it may be helpful to take a look at the past [Jac06]. Internet was originally built on top of a phone network. At the time most professionals in communication worked for telephone companies, and they did not believe Internet could work too well. In those days routing was done with phone numbers. The setup cost for creating a route between two telephones was high, but worked well for phone conversations that would typically take several minutes. For small data packets the case was different. It was observed, however, that keeping the phone lines open and doing the routing between end-points worked relatively well. The telephone communication workers saw this as an inefficient way of using their network, and refused to think of the Internet itself as a network. The world's first overlay network was born, and with it the first debate whether doing overlay networks is a good idea. Overlay networks are a topic still visible in the modern research, although the discussion has moved to upper levels [NEKB$^+$08].

The new routing paradigm introduced packet switching, and addresses for the communication end-points. Thus it became possible to have conversations between computers using data packets. *Content centric networking* (CCN) takes the stand point that doing conversations between end-points is not always the best way of doing networking [Sta09]. When networking is seen as a series of conversations the trust relations are seen to exist between the client and the server. In CCN, the trust relations and the technical means of delivery are being treated separately. Instead of naming end-points the focus is on naming the data. Naming the data, so the names can be proven to be correct without contacting a server, requires some cryptography. In an ideal content centric world all data, both online and offline, would be addressed in an equivalent manner, regardless of its physical storage location.

While CCN attempts to make the data addressable, another area called peer-to-peer networking asks us to make use of all computers involved in the system rather than concentrating a large part of the system on a centralized server. The term peer-to-peer was popularised by an online music file locating service called Napster. Napster had a centralised database containing music file meta data published by users. Users used the service with a proprietary Napster client. Given permission, the client would report users music files to the database, and set up a server for sharing the files to the Internet with minimal user interaction. Other users would then use the same client software to search the database and initiate downloads from the user running the server. As a consequence nearly all Napster users were running servers, which helped the system scale beyond capacity of the centralized parts run by the company involved. Napster initiated file transfers would take place between users – from peer to peer.

The next step, in the development of peer-to-peer applications, was the introduction of Gnutella. Gnutella users would similarly share files and transfer them to each other directly, but it had no central database listing the content shared by its users. Instead each Gnutella application would contact a few other Gnutella applications, to form a network. Peers broadcasted search queries regarding some content they were looking for. Broadcasting search queries worked well at first, but as the number of users grew, the network started failing. The load grew too high because all queries were forwarded to all nodes. Never the less, the Gnutella system can be considered more peer-to-peer than Napster since it distributed more functionality among peers.

As usage rose, peer-to-peer system developers started adding structure to their networks. Gnutella developers had their go detecting powerful nodes, and forming a core network out of such super peers. Although all peers were equal, some peers were "more equal" than others. Previously peer-to-peer networks had been said to exist as long as one of the peers continued operating normally. Exercising with super peers lead to the understanding that even though some systems did not introduce super peers explicitly, the nodes would often be unequal because of different network location, or different computing power. This brought attention to the problem that attacking certain well positioned nodes might hurt the system more than bringing down an arbitrary peer would.

After the few early file sharing applications, peer-to-peer application development forked to multiple different areas. The applications ranged from a virtual peer-to-peer aquarium, in DALiWorld [Ber01], to peer-to-peer telephony, in Skype [sky10]. Other areas included transferring distributed hash tables into peer-to-peer applications, implementing a file system in a peer-to-peer manner, and delivering content efficiently to multiple receivers.

The term peer-to-peer has lots of different interpretations. Sometimes peer-to-peer is confused with the openness of a protocol. Skype peers communicate using a secret protocol, known only to the company manufacturing Skype peers [BS06]. Still Skype is usually considered to be a peer-to-peer application. Sometimes it is seen as an ethical decision of not having a server. This interpretation is getting closer, and might be a true requirement for a pure peer-to-peer system, but it is quite common to have a central server, helping the cloud of peers. Some might say peer-to-peer is all about not requiring too much configuration. This is due to the Napster heritage, where servers were automatically configured to run on user machines. However, the amount of work required for setting up a server is a quality attribute. Failing to make a *good* peer-to-peer system should not take away from the peer-to-peerness of the system.

We could define peer-to-peer systems, as systems where the application developer makes use of client side computing power. As such this definition is not strict enough. Even a desktop solitaire application would match that definition, not to mention web sites that transparently execute pieces of javascript on the users computer. We can make our definition more accurate by requiring communication between the clients. This means that javascript can not be used today, by itself, to implement peer-to-peer systems. This is due to it being unable to receive information directly from pieces of javascript running on other computers. For peer-to-peer web applications to be possible, we would need to either implement a peer-to-peer web server, or enable javascript to listen for incoming traffic in the users browser.

Some preceding research has analysed the idea of separating content and servers, without focusing too much on peer-to-peer aspects of the problem [KYB$^+$07]. We add to their work by exploring the user side of such systems.

3 Protecting Users

In this section we examine three systems that attempt to protect their users from various threats available on the open Internet. An adversary who wishes to distort normal operation of the system, may attack either the publisher of the information, the store where the publisher stores the information, the system that enables interested users to locate the information, or the users who have retrieved the information. The factors that alter the difficulty of an attack include design of the storage system, as well as design of the system used for locating information. Incentives of the people operating the systems play an important role in retaining user's privacy and keeping data accessible and intact.

An adversary may try to gain undesired access to the shared information, or trace down the publisher, or other users interested in the information. He may take efforts to censor some content, or forge the information to force misinformation on the audience. He may try to corrupt the information, or make the datastore fail in delivering the information in its original form.

We will start the tour in subsection 3.1 by looking at Persona, which introduces a model for implementing online social networks in a way, that allows users to have a third party store their data, preventing misuse on the social network providers part. In subsection 3.2 we will look at Freenet, and its efforts in bringing privacy to the web by providing an anonymous datastore for storing web sites. Finally, in subsection 3.3 we will examine GNUnet, and its anonymous file sharing service.

3.1 Persona

Persona is a data protection model where publishers host information in encrypted form [BBS+09]. Each user is responsible for hosting their own data. It is assumed that the users are willing to take the responsibility of hosting their own data, in return for increased protection of private information. Persona users have cryptographic identities. In addition to addressing individuals directly, Persona supports user defined attributes, and some simple set operations on the said attributes. To become part of an average users Internet experience some integration with existing web standards, and web services is required. Persona attempts to minimize access rights given to the web service, to return users control of their personal data items.

Each Persona user is responsible for hosting their own data. This is different from peer-to-peer systems where it is common for users to help host content of other users who may be offline when the information is requested. It is also different from the traditional web service model, where online services host the data to get control over user's data or user experience. Such an online service might profit from access to users' private information, or the possibility to show advertisements to the user. Being responsible for hosting his own data the user may choose to nuke his entire digital life by deleting all the encrypted items from his datastore. This does not remove all copies that someone might have made, but breaks all the existing references to the data. The approach raises questions about whether or not users can be trusted with the responsibility of hosting their own data.

It is assumed that the users are either willing to take the burden of hosting their own data, or to pay for such protection, in return for increased protection of private information. The data stored in Persona is protected with AES encryption. Encrypting the data makes it possible to outsource datastore maintenance to an untrusted third party. For example Internet service providers might want to compete by providing a Persona datastore for their users. The data is given out from the datastore to anyone knowing its public name. Taking measures to protect the public name would be possible, and might in some cases actually happen, but the name is not assumed to be kept secret.

Persona users have cryptographic identities. A public/private RSA key pair is created for each user, and the public key is distributed to friends of that user. Online social networks, such as Facebook, are seen as a potential channel for distributing the public keys. While references to the encrypted data are public, the symmetric key for decrypting the hidden content is not. Persona does access control at the level of delivering the symmetric encryption keys to appropriate receivers. RSA is used to secure such key delivery to individuals.

In addition to addressing individuals directly, Persona supports user defined attributes. Persona users may tag other users with attributes, such as *customer*, *enemy*, *friend*, *trusted*, *untrusted*. Once such attributes are in place, users can use them to encrypt data for a set of users tagged with some specific attribute. Thus, once I have tagged all my friends with attribute *friend*, I can reuse the set of people defined by that attribute instead of selecting individual users repeatedly each time I wish to publish something to them. Users are not limited to using attributes defined by themselves, but can also use attributes defined by other people. It would for example be possible to restrict access to a user's mother, provided that the user has tagged his mother with attribute *mother*.

Persona also supports some simple set operations on the said attributes. This allows users to define richer access rules for their data. One could for example use the access control when sending out invitations for a cocktail party. Targeting the group of people having both the attribute *customer* and the attribute *trusted*, might help keep out some hostile customers. Having to create RSA keys separately for all attribute combinations would have lead into an explosion in the amount of keys, while encrypting the data separately for each recipient would have caused a large amount of encrypted data pieces. Thus Persona uses Attribute-Based Encryption (ABE), instead of RSA for securing key delivery, when the receiver is defined by attributes.

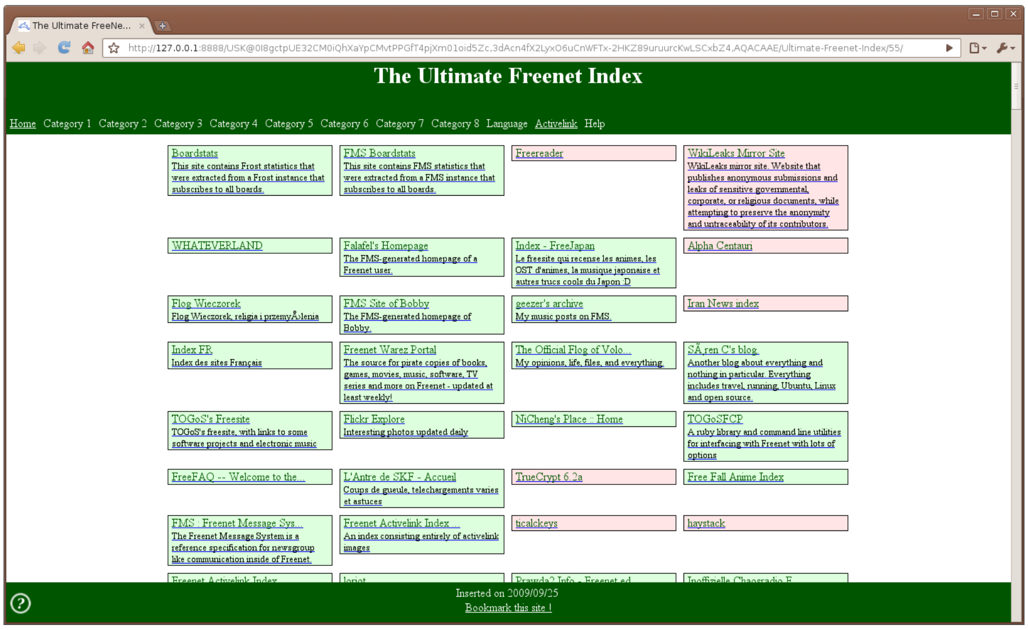
It is suggested that Persona would be integrated with existing web standards. As parts of web pages would in future only exist in protected Persona stores, web services need a way to insert those parts into web pages without retrieving and decrypting the content. A way for attaching crypto references to pages is required. A Persona-enabled browser would then retrieve those parts of the web page once the web page with Persona references has been loaded. Special support is also needed for publishing data through the user's datastore. The browser would upload the content into the user's datastore at publication time, and send a crypto reference instead of the content to the web service. In a transitional phase, support for the said standards could be implemented as browser extensions.

Persona also requires some support from the web services. While all of the cryptography is handled by the browser, Persona will not work with just any web service. The user's browser can provide access control settings together with ordinary web forms, and posting the form can trigger encryption, but the server needs to promise it will store the reference in a sensible way. In case of a legacy browser the user's data would remain protected, but communicating the reference forward would fail. Another problem is introduced by hostile web sites that might try to mime the Persona publishing user interface, but provide an insecure form instead. Thus the user needs at least partial trust in the web service.

By default Persona does not grant the web service any rights for accessing user data. Thus the web service cannot do any server-side manipulations on the data. However the web service continues to have a monopoly in providing access to the user's data as the entity storing the crypto reference. A user can provide the web service additional access rights, and might want to do so if the web service can demonstrate additional value to the user. More common tasks such as rendering the web page based on the encrypted content could be implemented by the service as client-side scripts that would act on the data after it has been decrypted by the browser. The script could leak a user's private information to the web service, as the current browsers allow javascript to call home. Thus the browser security model may need rethinking if users are to be protected with Persona.

3.2 Freenet

Freenet is a storage service, used for building anonymity enhancing censorship-resistant distributed applications [fre10]. Freenet nodes collaborate in a friend-to-friend overlay network where users explicitly choose nodes they connect to [CS05]. Nodes have no knowledge of other nodes behind the trusted neighbour nodes they interface with. Disk space allocation and load balancing are based on a virtual ring where nodes are placed. Nodes improve their relative position on the ring by swapping positions with their neighbouring nodes. Storing some data in the network is done by splitting the data into small pieces, and sending out storage requests for an encrypted representation of each individual piece separately. Piece retrieval is done by sending out retrieval requests for each encrypted piece. The system is vulnerable to a known attack, where an evil user tampers with the load balancing features by introducing misbehaving nodes into the network [EGG07].


The Ultimate FreeNe...
http://127.0.0.1:8888/USK@0I8gctpUE32CM0iQhXaYpCMvtPPGfT4pjXm01oid5Zc,3dAcn4fX2LyxO6uCnWFTx-2HKZ89uruurcKwLSCxbZ4,AQACAAE/Ultimate-Freenet-Index/55/

The Ultimate Freenet Index

Home Category 1 Category 2 Category 3 Category 4 Category 5 Category 6 Category 7 Category 8 Language Activelink Help

Boardstats This site contains Frost statistics that were extracted from a Frost instance that subscribes to all boards.	FMS Boardstats This site contains FMS statistics that were extracted from a FMS instance that subscribes to all boards.	Freereader	WikiLeaks Mirror Site WikiLeaks mirror site. Website that publishes anonymous submissions and leaks of sensitive governmental, corporate, or religious documents, while attempting to preserve the anonymity and untraceability of its contributors.
WHATEVERLAND	Falafel's Homepage The FMS-generated homepage of a Freenet user.	Index - FreeJapan Le freesite qui recense les animes, les OST d'animes, la musique japonaise et autres trucs cools du Japon :D	Alpha Centauri
Flog Wieczorek Flog Wieczorek, religia i przemyÅ›lenia	FMS Site of Bobby The FMS-generated homepage of Bobby.	geezer's archive My music posts on FMS.	Iran News index
Index FR Index des sites Français	Freenet Warez Portal The source for pirate copies of books, games, movies, music, software, TV series and more on Freenet - updated at least weekly!	The Official Flog of Volo... My opinions, life, files, and everything.	SÃ¸ren C's blog. Another blog about everything and nothing in particular. Everything includes travel, running, Ubuntu, Linux and open source.
TOGoS's Freesite TOGoS's freesite, with links to some software projects and electronic music	Flickr Explore Interesting photos updated daily	NiCheng's Place :: Home	TOGoSFCP A ruby library and command line utilities for interfacing with Freenet with lots of options
FreeFAQ -- Welcome to the...	L'Antre de SKF - Accueil Coups de gueule, telechargements varies et astuces	TrueCrypt 6.2a	Free Fall Anime Index
FMS : Freenet Message Sys... The Freenet Message System is a reference specification for newsgroup like communication inside of Freenet.	Freenet Activelink Index ... An index consisting entirely of activelink images	ticalckeys	haystack
Freenet Activelink Index	loriot	Prawda2 Info - Freenet ed	Inoffizielle Chaosradio F

Inserted on 2009/09/25
Bookmark this site !

Figure 1: An example of a *freesite*, retrieved from Freenet.

Freenet is used for building applications in an anonymity enhancing censorship-resistant manner. Existing applications include websites, usenet news, file-transfer and electronic mail. The websites stored in Freenet, or so called *freesites*, are accessed through a local web server that retrieves the required files from the datastore. It displays a progress bar while retrieving the content. Figure 1 is screen shot of a freesite displayed in a web browser. Some of the other services use specialized application software written for Freenet alone. One of the news service implementations has specialized application software that is used for reading news. Another news service implementation interfaces with standard usenet news readers by providing an interface similar to that of a regular usenet news server.

Freenet is a friend-to-friend network where users explicitly choose trusted nodes to connect to. This was not the case before version 0.7. In earlier versions forming connections between any two nodes on-demand was accepted, as long as the actual network environment did not block the connection by accident. While the new Freenet does not support on-demand connections, it has an automated dating service that nodes may use to trade security for performance. The dating service finds new connections for nodes to increase the amount of overall connectivity. The user can control the ratio of connections accepted through the dating service. Despite the dating service, algorithms running on the platform are not provided with information about arbitrary nodes, nor the ability to form connections to them. Instead the algorithms are forced to live in the restricted graph.

Freenet nodes do not have knowledge of nodes that might exist behind the trusted nodes they interface with. Synchronizing all content to each node provides a trivial way to share content in such dark networks. Doing so, however, requires huge disks from each participant. Freenet defines transitive rules that are used to determine where an arbitrary piece of content should be stored. Similarly rules are defined for locating the content from the location where it got stored. Each node allocates a fixed amount of disk space for the content, and some of the least popular content is dropped from the network as the datastore gets full. Because of the distribution and disk space limit enforcement, only statistical guarantees can be given for the storage.

Disk space allocation and load balancing are based on a virtual ring. Both nodes and data items are distributed evenly on the ring. Data item locations are decided by feeding encrypted representations of the data items to a hash function. Nodes with a closer position to a data item, consider the item more important, and are thus more likely to store and keep the item. Node locations are random. Random placement distributes the nodes on the ring evenly, but does not guarantee means for efficient routing towards specific positions on the ring. Thus, after initial placement, nodes swap places with each other to optimize routing performance. This behaviour does not change the distribution on the ring, but only affects which node is responsible for the original random positions.

Nodes improve their relative position on the ring by swapping positions with the nodes they connect to. Swapping does not affect topology of the network, but only the ring positions assigned to the two nodes involved in a swap. A node is said to be in a good position when most of its neighbours are close on the ring, and a few neighbours are far away. The small amount of long links helps to route rapidly towards the correct direction, while the larger amount of short links are useful for more accurate routing once requests approach the right direction. When negotiating a swap, two nodes both check separately that the swap is useful for them. In the case where both are happy with the swap, they proceed to actually changing identifiers.

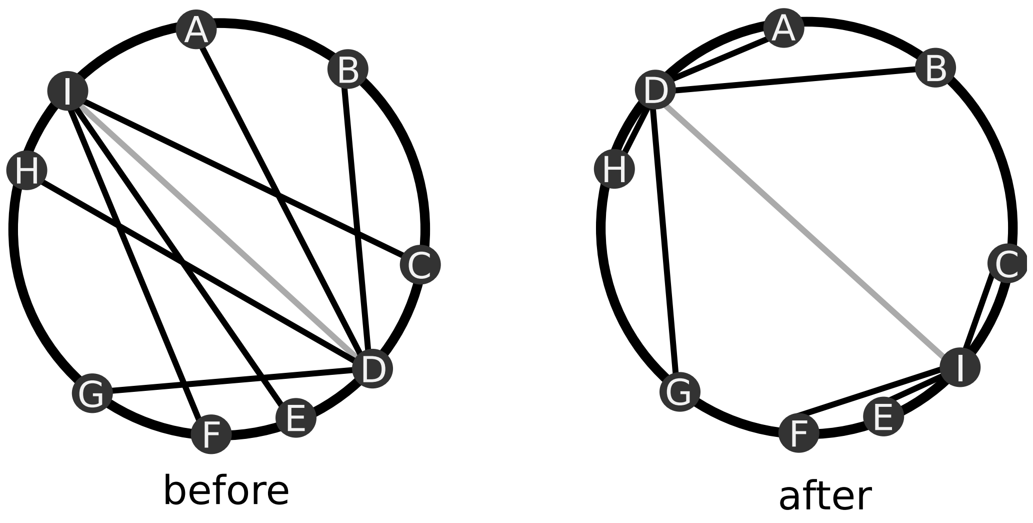


Figure 2: Freenet ring position swap. Lines depict connections between nodes. The nodes D, and I consider a swap. Left image shows the situation before, and right image after, swapping positions. The swap moves neighbours closer to each other, and is found to be beneficial.

Figure 2 shows an example where two nodes consider swapping places, find the swap beneficial, and finally swap places. Nodes store their ring position in long term memory. When a node restarts it joins at the same ring position which it left.

Storing some data in the network is done by splitting the data into small pieces, and requesting storage for an encrypted representation of each piece. The requests are routed at each hop towards the node whose id is closest to the hash of the encrypted piece. Thus each piece has a predetermined point on the ring towards which it is attracted to. While making routing decisions the node ignores the path where it received the piece from, as well as any congested neighbours. Routing continues until the requests get dropped from the network for being around too long. The requests are passed on even in a case where the current node is closer to the piece than any of its neighbours. While the request is being routed around the network, some nodes that pass the request forward may store the piece. Nodes whose id is close to the hash of the piece are more likely to store the piece than nodes further away on the ring.

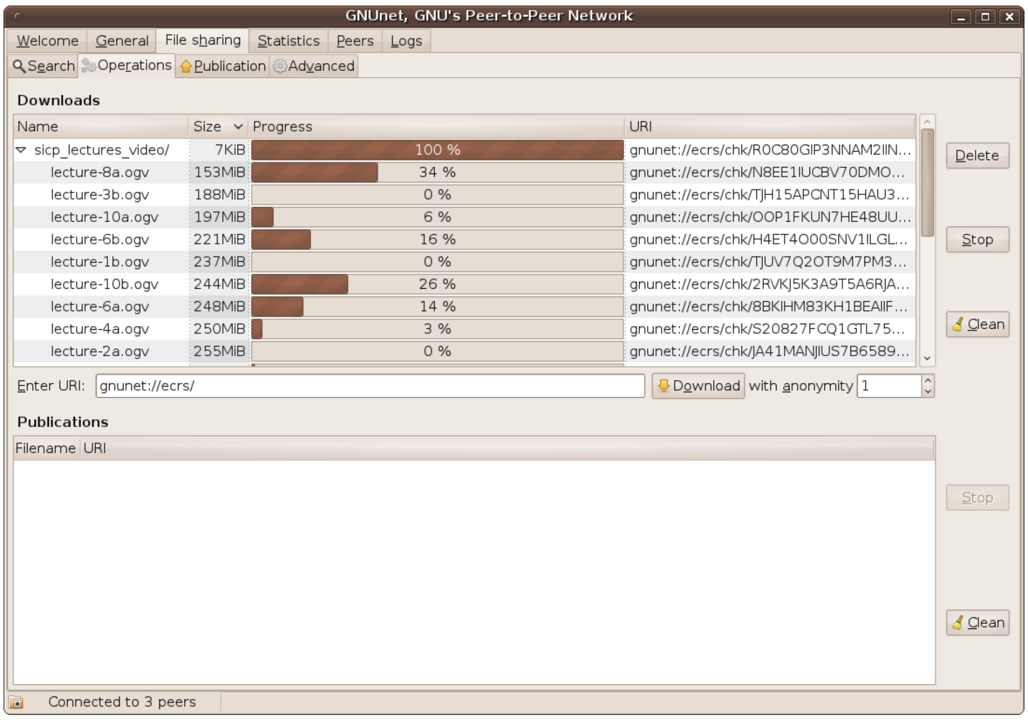
To retrieve some data from the network one needs to ask for each of its encrypted pieces separately. Routing of the retrieval requests proceeds in a similar manner to the routing of storing requests. If a node storing the requested piece is encountered before the request gets dropped from the network, the piece is returned by the same path that the retrieval request traversed, only backwards. Each node forwards the response to the previous requester until it reaches the original requester. While forwarding the response some or all of the nodes may store the encrypted piece. Again, the nodes closer to the content on the ring are more likely to store the piece.

An evil user may attack the load balancing by introducing misbehaving nodes to the network. The attack targets the node introducing algorithm, and the position swapping algorithm. When a new node joins the network it is supposed to enter at a random location. However it is the node's own responsibility to randomize its starting position. Thus it becomes possible to introduce an attacker node at a chosen location instead of a random location. The attack node may then wait for a swap to happen. Once a swap has happened, the attack node may choose to remove the gained node location from the network by quitting. As it is also responsible for storing its previous location, it may rejoin at a location of its choice. By repeating these steps it is possible to make node distribution on the ring unbalanced, degrading routing performance to that of random walk. Preventing such misbehaviour remains an open research problem.

3.3 GNUnet

GNUnet is a peer-to-peer framework that attempts to retain anonymity of its users and takes some measures to thwart censorship. One of the first applications implemented on GNUnet, shown in Figure 3, is an anonymous censorship-resistant file-sharing application. The framework provides its applications with services for building an overlay network, discovering new nodes, keeping trust records, anonymizing downloads, and locating content. The file-sharing application encodes content with Encoding for Censorship Resistant Sharing (ECRS) to protect the content in highly untrusted environments [BGHL03]. ECRS is used to encode shared files into small encrypted pieces, with an index structure built out of similar encrypted pieces. In addition to protecting the actual payload, ECRS is used to provide secured freeform names for shared content.

GNUnet is an overlay network where the nodes use multiple different protocols to communicate with each other [FGR03]. The protocols include UDP, TCP, HTTP, and SMTP, both on IPv4 and IPv6. Supporting multiple protocols helps to circumvent censorship attempts and limitations of restrictive network environments. The



Downloads

Name	Size	Progress	URI
sicp_lectures_video/	7KiB	100 %	gnunet://ecrs/chk/R0C80GIP3NNAM2IIN...
lecture-8a.ogv	153MiB	34 %	gnunet://ecrs/chk/N8EE1IUCBV70DMO...
lecture-3b.ogv	188MiB	0 %	gnunet://ecrs/chk/TJH15APCNT15HAU3...
lecture-10a.ogv	197MiB	6 %	gnunet://ecrs/chk/OOP1FKUN7HE48UU...
lecture-6b.ogv	221MiB	16 %	gnunet://ecrs/chk/H4ET4O00SNV1ILGL...
lecture-1b.ogv	237MiB	0 %	gnunet://ecrs/chk/TJUV7Q2OT9M7PM3...
lecture-10b.ogv	244MiB	26 %	gnunet://ecrs/chk/2RVKJ5K3A9T5A6RJA...
lecture-6a.ogv	248MiB	14 %	gnunet://ecrs/chk/8BKIHM83KH1BEAIIF...
lecture-4a.ogv	250MiB	3 %	gnunet://ecrs/chk/S20827FCQ1GTL75...
lecture-2a.ogv	255MiB	0 %	gnunet://ecrs/chk/JA41MANJIUS7B6589...

Enter URI: gnunet://ecrs/ with anonymity 1

Publications

Filename	URI

Connected to 3 peers

Figure 3: Downloading programming lecture videos with gnunet-gtk.

sent SMTP messages have an answering machine flag set, to make sure mailing lists can not be tricked into forwarding peer advertisements. Nodes are identified by the hash of their public key, and all traffic between peers is encrypted with public key cryptography. Each node forms as many connections as possible. The number of connections is limited by connection table size, which is set based on the amount of resources available on the platform. Nodes proven good in previous sessions have a higher probability of getting a connection. Some connections are swapped from time to time, lowering the odds of becoming surrounded by hostile nodes, but also to provide new nodes opportunities to prove their worth.

A couple of mechanisms are used to learn about new nodes. A few relatively stable node caches are preconfigured in the client, and the client will try to use as many as possible to discover an initial set of peer advertisements. A peer advertisement is signed and contains an address, and the public key of the peer that created the advertisement. Nodes verify peer advertisements by sending an encrypted random number to the node mentioned in the advertisement, and expect to get the same number back in return. In case of no response, no further communication will happen. As all traffic is encrypted with the receivers public key, it is impossible for the receiver to return the number unless it has the private key for the advertised key pair. Nodes will then forward verified peer advertisements to their neighbours during idle periods. Any node may also be turned into a public node cache, by turning on an integrated http node cache server. Using statistical models for finding other nodes, by port scanning likely IP ranges, has also been researched [GG08].

Each GNUnet node keeps a local trust value record for each of its neighbouring peers [Gro03]. The value is initialized at zero and can never go below that. Thus nodes can never improve their position by creating new identities. GNUnet applications use the trust value when they consider interactions with other peers. They may also increase or decrease the value when they observe good or bad behaviour. The protocol does not make it possible to ask neighbouring peers about the trust one has received. A node may compute estimates, based on the amount of trust it would have given itself, if it had been in the position of the other node. A trustful node may also notice an increase in the quality of service received from neighbours. All trust values decrease over time, and nodes lower their trust towards neighbours who request service during busy moments. If a node has excess resources it does not lower its neighbours trust when the neighbour requests service. This feature helps the network build up trust during idle periods. It is possible for a node to turn off the trust tracking system. Turning the trust system off decreases performance of the node, as it is not able to bias its queries towards efficient nodes. In future its neighbours may also assign more trust to their other neighbours because of the weakened performance.

To download a piece of data from the network, a node asks some of its neighbours to provide the piece [BG03]. Some measures are taken to protect the anonymity of the original publisher and the actual downloader. To anonymize its own traffic, a node collects requests from other nodes, and sends out its own requests among the ones it is forwarding. The recipient does not know which messages originated from the neighbouring node it is communicating with. Responses to queries are delayed a random amount of time to protect the sharing node from revealing that it had the content. Once the response gets sent, it is routed backwards hop-by-hop until it reaches the originator of the request. While forwarding queries, a congested node may drop itself out of the forwarding path by asking its follower to respond directly to its predecessor. Such path shortening decreases the anonymity of the drop out, as it loses some of its cover traffic, but does not affect anonymity of its predecessor which continues to have the same amount of cover traffic. A node controls the level of anonymity by controlling the ratio between cover traffic and payload. This is a local operation.

Queries are forwarded after each hop to a random neighbour using a biased random function. The bias is based on three factors. Nodes who have bigger trust values are preferred over nodes with low trust values. The bias for high trust helps the queries to get around evil and broken nodes. Nodes that have been able to provide us with correct answers lately are preferred over nodes that have not been able to respond to our recent queries. The bias towards nodes that are able to respond, helps us direct our queries towards the parts of the network that seem to have the information we are currently looking for. Finally, nodes whose identifier is closer to the identifier of the requested piece are preferred. The biasing based on node identifier makes similar queries take similar paths, which makes caching pieces at intermediaries more efficient. Caching pieces is not compulsory, but allows one to gain trust multiple times for a piece of content, and only lose trust one time when the piece was originally downloaded.

Having multiple pieces instead of one large record helps to parallelize work. It becomes possible to query for multiple pieces of a file at once. The queries may take different routes, and the responses may come from multiple sources. Distributing the work among multiple nodes adds fault tolerance to the system, but also speeds up operation by balancing load between nodes. Using small pieces is also considered safer for node operators who forward and cache encrypted content that could turn out to be illegal under their legislation. Handling small pieces is seen as less risky because one intermediate node does not usually store or forward all pieces belonging to some controversial content.

Encryption protects the payload from intermediaries, who participate in content delivery. It is desirable for the same file to always result in the exact same set of encrypted pieces. This makes it possible for multiple users to co-operate in sharing a common file, without communicating, or even knowing about each other. Because GNUnet anonymizes its user it is also important that each piece can be validated against a given query at each hop. With such verifiable pieces, any node in the routing chain, receiving an invalid piece from one of its neighbours, can tell that the preceding node is either broken, trying to carry out an attack, or lazy in validating incoming packets. This prevents attackers from hiding in the anonymity cloud.

To encode a file, we first chop the data into pieces, and then encrypt each piece separately. We start by examining the length of the data. The length of the data

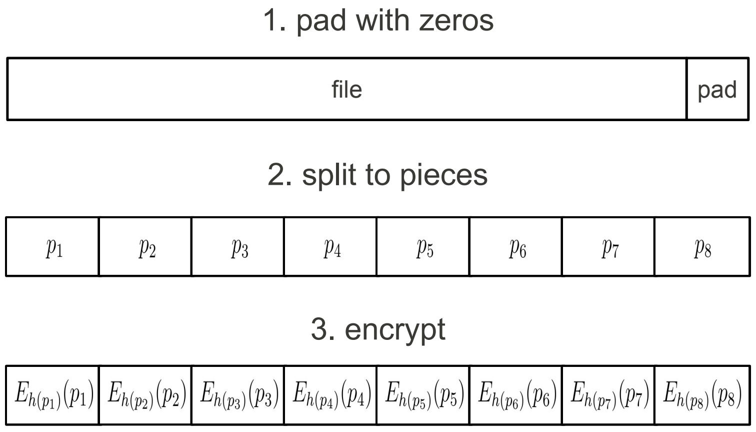


Figure 4: Turning a file into DBlocks.

needs to be divisible by an implementation wide constant piece length l. In case it is not divisible we pad it, with the minimum amount of zeros required, to make it divisible. Then we start taking pieces of length l from the beginning of the padded data d, and continue until we have wholly consumed d. This should result in c pieces $\{p_i : i \in \mathbb{Z}, 0 < i \leq c\}$. After that, for each piece p_i, we use an irreversible hash function h to extract a key $h(p_i)$. Using the key, we encrypted the piece, with symmetric encryption E, turning it into a DBlock $E_{h(p_i)}(p_i)$. We refer to a *DBlock* with its hash $h(DBlock)$.

When we wish to retrieve a file, we need to retrieve the encrypted pieces, decrypt them, and combine them in the correct order. To send out queries for piece p_i, we need its reference $r_i := h(E_{h(p_i)}(p_i))$, and to decrypt the piece, we need its encryption key $k_i := h(p_i)$. Thus a block reference $b_i := (r_i, k_i)$, with said two details, is all we need to request a single piece, and reveal the original plain text. To retrieve a file with c pieces, we need a c-tuple $(b_1, b_2, ..., b_c)$ holding block references for all pieces in correct order.

With large files, the block reference tuple grows inconveniently large, and therefore most of it is stored in the datastore along with the actual data. While it would be possible to store the references in a file for uploading them to the network, a slightly optimized way of building index structures is used. We split the c-tuple into m-tuples, m being the maximum number of block references fitting in one piece without compression. The last tuple may have null items at the end, in the case where c is not divisible by m. We then serialize each m-tuple, and pad the serialized representations with zeros to create a plain text of length l. We then turn each plain text into an IBlock, by encrypting it like we would encrypt a piece to produce a single DBlock.

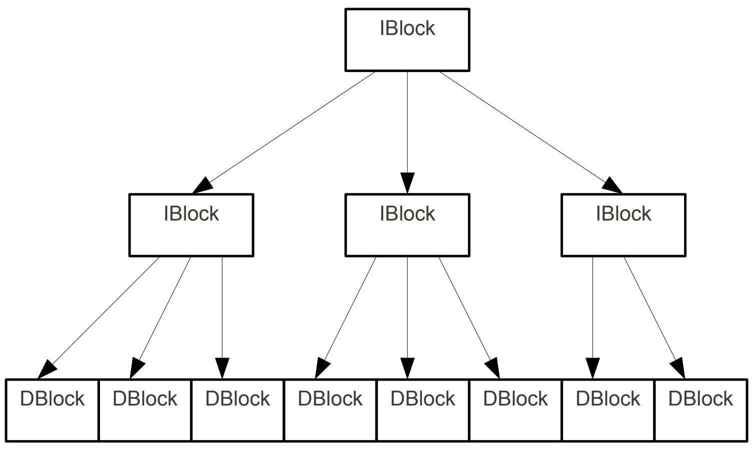


Figure 5: An example reference tree for a stored file.

By storing block references into the network one level at a time, we may build a reference tree for the shared file. After producing IBlocks for a set of encrypted pieces we are left with another level of block references, pointing at the newly created IBlocks. With any reasonable piece length, the number of remaining block references should be smaller than the number of block references we started with. We repeat the process turning those block references into another set of IBlocks, and again, until we are left with a single block reference. We use the block reference (ref, key) of this, root IBlock, as a reference for the whole tree. To download the file, we additionally need to know the length len of the shared file. Without knowing the length we are unable to remove the padding that got added to the file before turning it into DBlocks. To advertise a shared file, we advertise its file reference $(id, key, length)$.

In addition to protecting the actual payload, ECRS is used to provide secured freeform names for shared content. To name files with arbitrary strings we need a format for storing file/name associations. Each association maps a name n, and a file reference r. We could store the two details in a pair (n, r). With such pairs an attacker could filter out all associations for a specific file, but also filter out all results with some substring in the name. In addition to censorship resistance, we also need to be able to validate publishers of associations to guard against adversaries attempting to create answers to queries on the fly. The anonymizing functionality of the network hides origins of any attack that cannot be detected by the attacker's immediate neighbours. Without publisher validation the fake responses might even get cached at some intermediate nodes, supporting the attacker's cause. SBlocks are introduced to solve the fake response problem for queries that target associations verified by a known pseudonym. KBlocks are introduced solving the fake response problem for queries targeting a global namespace. Once protected, the associations may be used to implement versioning, protected spaces, or keyword search.

To make associations censorship resistant, we need to protect both the name and the reference. Protecting the reference is easy. We hash the name to create an encryption key $h(n)$, and encrypt the file reference turning it into a cipher $E_{h(n)}r$. The cipher is decryptable by anyone knowing the name part of the association. Thus an attacker wanting to filter out all associations for a certain file, needs to know all the names used in the related associations. Protecting the name part is tricky, as third parties need to be able to figure out matching associations for a given query. We protect the name, by hashing it twice to create a query identifier $h(h(n))$. The query identifier does not reveal encryption key used for the reference, and censoring all results that match a given substring becomes impossible. Finally, we pair the query identifier with the protected reference to create a protected association $(h(h(n)), E_{h(n)}r)$. The encrypted part of an association may be used to store additional metadata for the file along the reference. GNUnet stores thumbnail images and textual meta data for the referred file in the associations.

The problem of fake responses is dealt with by adding pseudonyms to the system. A user can create a new pseudonym by creating a random seed s, and feeding the seed into a deterministic key generator. Given the seed, the key generator returns a public/private key pair (Pub_s, Prv_s). The key pair is a cryptographic identity of a pseudonym. We refer to the pseudonym, with the hash of its public key $h(Pub_s)$. The private key may be used to approve associations, while the public key may be used for verifying authenticity of such approvals. Approving names as some specific pseudonym without the private key of that pseudonym is considered impossible.

To approve an association with her pseudonym, Alice creates a protected association a as described above, and signs it with her pseudonym's private key. The result $[a]_{Prv_s}$ can be verified to have originated from the pseudonym by anyone holding the public key of her pseudonym. She combines the signed association with the public key to form an SBlock $([a]_{Prv_s}, Pub_s)$. As the SBlock includes the public key, anyone running into it can verify that the association stored within was approved by the pseudonym. As long as queries target a specific publisher, the publisher's identifying hash may be included in the query, and verified to match suggested responses at each hop. Generating responses on the fly for such queries becomes impossible. When associations are signed by pseudonyms, it also becomes possible to form an opinion of a given pseudonym. Using reputation systems, such as Credence [WS05], to collaboratively rate pseudonyms might be possible, but implementing reputation systems in a censorship resistant manner remains an open challenge.

As queries against the names verified by pseudonyms need to target a specific publisher, another format is required to store associations in a global space. The problem is solved by using the name part of the association as a key for gaining publishing rights. To retrieve signing keys for some name n, we hash the name, creating a seed $h(n)$, and give it to the same key generator used for generating regular pseudonyms. The key generator returns a public/private key pair $(Pub_{h(n)}, Prv_{h(n)})$, which works as a cryptographic identity for the anonymous group of users aware of the name. Similar to pseudonyms, we refer to identities of names with a hash of their public key $h(Pub_{h(n)})$. Approving an association with a group identity is similar to approving an association with a regular identity, but the result $([(h(h(n)), E_{h(n)}r)]_{Prv_{h(n)}}, Pub_{h(n)})$ is called a KBlock because of its special properties.

The name part of an association may be used to implement versioning, protected spaces, or keyword search. A name could have a version number, and updates could be provided by sharing the update with the version number changed. One could use easily guessable names to implement search, or one could choose to use names that can not be guessed, and use them as passwords. For example Alice and Bob, being members of the same cooking club, could come up with name "recipe" for exchanging some of their newly found recipes. Eve, who likes stealing 0-day recipes, might be able to guess the name "recipe" and gain rights equal to those of Alice and Bob. If Alice and Bob wished to protect their content, they could use a password generator to create a secure name for their content.

4 Peerscape Platform

Peerscape [pee09] started as a library for connecting some desktop applications to form a peer-to-peer system. To get the nodes connected we created a distributed database. To experiment with the database, we wrote a web server that revealed contents of the database through a web browser. Some of the problems with peer-to-peer web seemed to be related to navigation and making decisions about which content to trust. We assumed that tying the content to people might make these questions more conceivable to a regular user. For a picture of a Peerscape user profile see Appendix 1.

The remainder of this chapter is organized as follows. In subsection 4.1 we discus the issue of loosening the tie between content and its storage location. In subsection 4.2 we introduce our distributed datastore. In subsection 4.3 we explain how we can harness the datastore for building distributed web applications. Finally, in subsection 4.4, we put Peerscape into perspective by comparing it with systems introduced in the previous section.

4.1 Encoding for Tamper Resistant Sharing

Peerscape stores data in datasets belonging to pseudonymous users. Each dataset has a root item that works as the dataset's creation certificate. Each pseudonym is identified by a special profile dataset, that may be used to publish public information about the pseudonym. A pseudonym may create one or more additional datasets, which are initially kept private from other users. Individual data items are stored in records that represent updates to key-value items in a dataset. The structure of a dataset is rather general and allows a multitude of different applications.

Each dataset has a root item that works as the datasets creation certificate. The root item identifies a single pseudonym who is considered the owner of the dataset. Datasets are referred to by their information identifier (IID), which may be computed from the root item. Thus it is possible to check if an IID matches an arbitrary root-item, by recomputing the IID. The IID is computed in a way that makes an IID mismatch likely for corrupted root items. The owner approves ownership by signing the root item. The owner's public key is attached to the root item, making it possible to check the signature with no prior knowledge of the owner.

A new pseudonym is introduced by creating a public profile dataset for the pseudonym. There is no limit on the amount of pseudonyms one user may have. To create a profile dataset for her pseudonym $Ecila$, Alice first creates a public/private RSA key pair $(Pub_{Ecila}, Prv_{Ecila})$. She hashes her public key to produce a core $h(Pub_{Ecila})$. She then adds her pseudonym's signature to the core, turning the core into a profile dataset creation certificate $[h(Pub_{Ecila})]_{Prv_{Ecila}}$, which she pairs with her pseudonym's public key, to form a profile dataset root item $([h(Pub_{Ecila})]_{Prv_{Ecila}}, Pub_{Ecila})$. We hash the core to retrieve the dataset's IID $h(h(Pub_{Ecila}))$.

Once a pseudonym has been introduced by creating the profile dataset, one or more additional datasets may be created. Unlike the profile dataset, the additional datasets are not computationally revealed to other users who encounter the pseudonym. To create an additional dataset, Alice takes the hash of her pseudonym's public key $h(Pub_{Ecila})$ and creates a random seed s. She then encloses the two in a pair, forming a core $(h(Pub_{Ecila}), s)$. She then adds her signature to the core turning it into a dataset creation certificate $[(h(Pub_{Ecila}), s)]_{Prv_{Ecila}}$, and pairs the certificate with her public key to form a dataset root item $([(h(Pub_{Ecila}), s)]_{Prv_{Ecila}}, Pub_{Ecila})$. We hash the core to compute the dataset's IID $h((h(Pub_{Ecila}), s))$.

A pseudonym may create records for any dataset, as long as he knows the information identifier (IID) for that dataset. Dictatorship is not required for creating records. Having a dataset's IID d, the user may bind some name n within the dataset to version v of some content c, by enclosing the said details into a record (d, n, v, c). The publisher is made accountable by requiring the record to be extended with the pseudonym's signature and public key, leading to a key-value item $([(d, n, v, c)]_{Prv_{Ecila}}, Pub_{Ecila})$. By verifying the signature, we can also detect corruption that might have happened during the transfer. Two items, one with record $(d_1, n_1, \_, \_)$, and another with record $(d_2, n_2, \_, \_)$, are considered to be versions of the same item, iff $d_1 = d_2$ and $n_1 = n_2$. The records have a total order, which determines which version of the item has the latest version of the content. Items with a higher version number are more recent, than items with a lower version number.

The structure of a dataset is rather general and allows a multitude of different applications. Figure 6 presents an example where two pseudonyms, Ariadne A and Theseus T, store messages for each other. The messaging takes place in Theseus's profile dataset, which has the IID $p_T = h(h(Pub_T))$. First Adriane stores a message to Theseus's profile dataset with name "m1", asking him about his rendezvous with Maxotaur. Theseus replies by storing his reply with name "r1". While reading the response, Adriane notices she wrote Maxotaur where she should have written Minotaur. To avoid further embarrassment, she decides to store a corrected version of the message into Theseus's profile with version number two. This minimal example demonstrates some key features of the encoding. We present some real world Peerscape applications in Chapter 5.

$$([(p_T, "m1", 1, "Did\ you\ deal\ with\ that\ nasty\ Maxotaur\ already?")]_{Prv_A}, Pub_A)$$
$$([(p_T, "r1", 1, "Not\ yet!\ The\ Minotaur\ is\ hiding\ in\ the\ Labyrinth.")]_{Prv_T}, Pub_T)$$
$$([(p_T, "m1", 2, "Did\ you\ deal\ with\ that\ nasty\ Minotaur\ already?")]_{Prv_A}, Pub_A)$$

Figure 6: ETRS application

4.2 Synchronizing Data Store

While working on the synchronizing datastore, we designed a protocol for synchronizing datasets, defined a write access policy-mechanism, and implemented a prototype of the system as overlay networks on top of the public Internet. Users of the datastore subscribe to a number of datasets. To synchronize a dataset we form a network by connecting the dataset's subscribers to each other. World-writeable datasets have some problems, which can be mitigated by introducing write access control to the datastore. Initially, only records created by the owner of a dataset are stored and synchronized to other nodes. Once the users have defined access policies, we may enforce them at each node, to save some resources and to protect the users from questionable content. Each node in an overlay targets two outgoing connections, and accepts a number of incoming connections as well. To advertise the local node and to learn about other nodes, we exchange advertisements in an external federated datastore. While working on the overlay, we had some problems with restrictive network environments. As we were researching peer-to-peer systems we considered some social solutions in addition to purely technical ones.

Each Peerscape user is subscribed to a number of datasets. The goal of the synchronization protocol is to synchronize each dataset with other users who are subscribed to that dataset. Subscription to a dataset is a local operation, where the user's node is configured to exchange records for the given dataset. The architecture does not support transitive requests that pass multiple hops, or requesting individual records. These limitations simplify the design, as we do not need to do routing, or keep track of the nodes having a specific record. The synchronization protocol also does not provide a method for closing down connections. Refusing to provide a method for clean shutdown may result in more robust applications, as it increases the amount of field testing for crash recovery code [CF03][CCF04].

To synchronize a dataset, we form a network by connecting the dataset's subscribers to each other. When two nodes connect, they check that the other node is aware of the IID of that dataset. The dataset's IID works as a passphrase for the dataset. Care is taken to prevent the IIDs from leaking in the process. After initial security checks, the freshly connected nodes compare the records they have for the dataset. After comparing records the nodes fill each other in by supplying each other with the missing records. The nodes keep supplying each other with any new updates they receive after the initial synchronization, as long as the connection stays open. As a transitive effect of such synchronizing connections the whole network becomes synchronized.

World-writeable datasets have some problems, but they can be partially solved by introducing some write access control to the datastore. As a dataset grows huge, it starts eating up lots of disk space, its synchronization starts to consume lots of network bandwidth, and the related computations may be slowed down by the sheer amount of data. Some users may also store questionable content in a dataset. As a consequence of synchronization all subscribed users will, not only store, but also help disseminate the content. By introducing write access control, we reduce the risks involved with subscribing to a dataset. To limit write access, we automatically enforce policy documents that users store in the datasets. By doing so we make the owner accountable for the content stored in his dataset. Having a single authority for each dataset helps the user decide which datasets he is willing to trust based on his knowledge about the different authorities.

Initially, only records created by the owner of a dataset are stored and synchronized to other nodes. The owner may grant write permissions to other users by storing policy documents in the dataset. The owner might make some users administrators by giving those users full permissions to the dataset. Such administrators would have near equal rights to the owner. The ownership however, would be permanent, while administratorship could be lost with a change of policy. The owner might give some users only limited rights by describing limitations in the policy document. The limitations could include limitations on the key or limitations on the stored content. The limitations could be described as regular expressions, or they could state the maximum size for the stored value. Such limitations could be used to prevent users from writing further access policies, or to restrict their write access to one specific bit under a given key.

We enforce the policy documents at each node, to save some resources, and to protect the users from questionable content. With such enforcement, content that has not been approved, either directly or indirectly, by the owner gets dropped after traversing one hop. By validating locally inserted data against the policies we can be sure that our node does not disseminate invalid records even in the case our application is broken. A policy may sometimes be changed to revoke permissions that were previously granted to a user. The datastore keeps track of the order of items it has received. When someone gets deauthorized, the future updates that appear in the item log after the new policy document, will get dropped. Order of items is preserved when items are transfered over the network from one node to another. As the order in which different nodes receive their items may vary, different nodes may have slightly different perception about the items that are permitted.

Each node in an overlay targets two outgoing connections, and accepts a number of incoming connections as well. As we are talking about synchronizing connections, one connection to the network should be enough. As long as no one quits. When some nodes quit, the network where each node took only one connection, is likely to split into multiple separate networks. A higher number of connections adds to redundancy making network splits less likely, while a lower number of connections saves resources. We decided to go with two out going connections, as it gives each node four connections on average. In future, the targeted number of connections could be optimised by measuring performance of different alternatives. While we did not perform such experiments, we optimized the number of connections between two nodes by delivering updates for all common datasets through one connection. This may not seem too significant, but users with common interests might have a huge number of common datasets.

To advertise the local node and to learn about other nodes, we decided to exchange advertisements in an external datastore called OpenLookup, a federated datastore with a DHT-like API[ope09]. The nodes participating in the overlay network would use multiple alternative ways to obtain their local IP-addresses. They would ask the system to provide some addresses, but they would also ask their neighbouring peers to provide some. At startup, and every so often while running, they would store a tuple with all of their supposed addresses to the datastore. When forming new connections, the nodes would query the datastore to get addresses for nodes participating in synchronization of that particular dataset. Storing the addresses as a tuple would allow the connecting node to do some reasoning about the advertised nodes location in the network. In some cases the connecting node could reason that the other node is in the same local area network. The node might then choose to form a local connection instead of circulating traffic through the Internet.

We experienced some problems with restrictive network environments that do not allow direct connections between nodes. Ideally any node should be able to connect to any other node in the Internet. In practice many nodes do not have a public IP address, but they share one with others through network address translation (NAT). When using IPv4, that is. IPv6 makes the hosts addressable by introducing new addresses that combine the IPv4 address of the NAT box with the local IPv4 address of the actual host. As IPv6 infrastructure is not yet available everywhere, Teredo [ter07] can be used for tunneling the IPv6 packages over IPv4. Teredo relies on some general NAT-traversal infrastructure to get past the IPv4 NAT. Operating system vendors provide the Teredo infrastructure for their customers, but some parts of Teredo-systems are also run by volunteers who wish to help in moving the Internet to the next level. While we acknowledged the potential in moving to IPv6, we did not go through the efforts of porting our code to IPv6.

Some problems present in a synchronizing peer-to-peer datastore may have social solutions. As interest groups arise around some content, the more advanced users may help the Sunday users by making sure their node has a public IP address which is accessible, and having their node accessible more often. Our datastore does not store content on hosts that are not subscribed to the dataset. Instead anyone, being aware of the dataset, may opt in at any time by subscribing to the dataset. Subscribers might include individuals, but they might also include companies. Google for example might want to subscribe to as many datasets as possible and index them to provide search. They might learn about new datasets by crawling Peerscape links from public web and other datasets. On the other hand Amazon might want to subscribe to a dataset, if their customer paid them to make backups of a dataset or keep it online when they close their laptop lid.

4.3 Peer-To-Peer Web

When we had our datastore working, we wanted to make writing applications for the datastore as easy as possible. We ended up implementing support for web applications, as web browsers provided a good selection of application programming support and were widely deployed every where. We turned the synchronizing datastore into peer-to-peer web by revealing its contents through an integrated http server. Goal of the REST-like http API, was to support development of applications in javascript. The localhost urls used to address the server turned out to be nontransferable, and failed to protect our web applications from each other. We designed a virtual url scheme that would both sandbox the applications and make the urls transferable. Supporting virtual urls required intercepting the actual requests, and serving contents from the synchronizing datastore instead. Finally, we added some support for instantiating new servers from existing ones.

We turned the synchronizing datastore into peer-to-peer web by revealing its contents through an integrated http server. The value matching *key* in the dataset *IID* would be available at http://localhost:*port*/collection/*IID*/*key* given that we had the server running on port *port*. We stored a dataset starting page with key *web/index.html*, and it was thus available at http://localhost:*port*/collection /*IID*/web/index.html. As the sites were stored in a synchronizing database, all changes done to the local copy of the web site were automatically propagated to all copies of the database. The platform supported all client side web technologies. Writing to an address from the browser would cause the server to write the alternated value to the synchronizing datastore, which would automatically make it available in every other instance of that web application. Supporting various serverside technologies would have forced us to implement a secure sandbox for those technologies. The related challenges were equal to the challenges in supporting those technologies on the client side in general, and were beyond the scope of our research.

The localhost urls turned out to be nontransferable, and failed to protect our web applications from each other. We hoped that Peerscape users could send url addresses to each other off-band. The localhost url however, contained the port number of the local server, which broke them on machines that had Peerscape running on another port. We thought Peerscape might one day become a feature built into browsers. Standardising the port would not have worked, as one host can have multiple browsers and might thus end up running multiple instances of Peerscape. With a standardised port only one Peerscape-enabled browser would work in one host at a time. We also wanted to prevent applications by different authors from accessing each others data. Javascripts coming from different servers, or origins, can only communicate through carefully protected channels [JW07], while scripts coming from the same server have much more liberty. With our localhost urls, a javascript stored in one dataset, had access to all datasets as it was being hosted from the same server.

To fix the said problems, we designed a virtual url scheme that would both make the urls transferable, and cause the applications to be isolated from each other. We registered the domain *hexlabel.net* to make sure it was not used for anything else. We decided that each dataset would be accessible under domain IID.hexlabel.net, IID being the information identifier of the dataset. From now on each dataset would have its own domain. This would restrict javascript from accessing content stored in another dataset. The new url would not contain a port number, but figuring out the correct port number was left to the Peerscape implementation. We still wanted to leave some room for extensibility, and decided to have the raw content available under subpath *content*. Hexlabel url for a dataset *IID* would from now on be available at http://*IID*.hexlabel.net/content/web/index.html. While we were at it, we made accessing the front page easier by showing it to the user in case he left the key part unspecified. We also had some plans for running a webserver at hexlabel.net to provide our users with fall back content, in case they ended up using a system without Peerscape support.

Supporting virtual urls required intercepting the actual requests to hexlabel.net, and serving contents from the synchronizing datastore instead. We experimented with three ways of intercepting the requests. We started by turning our webserver into a proxy server. Once a user configured Peerscape as his proxy, all his traffic would be sent to our proxy. In this model we took the responsibility of serving the user all regular web pages in addition to the ones stored in Peerscape. In our second attempt we generated a proxy auto-configuration (pac) file for the user. The pac file would configure the users browser to only send hexlabel.net requests to our proxy server, and use direct Internet connection for the rest of the traffic. Still our pac file replaced any existing proxy configurations, which in some highly restricted environments would prevent the user from accessing the Internet. Our final version had a browser extension for intercepting the connections, and forwarding them to our datastore. This would not break any existing proxy configurations that the user might have. As the user now had to install a browser extension, we decided to package the datastore inside the same extension. By doing so, we were able to provide our users with a single, easily installable package containing everything required for running Peerscape.

Peerscape platform also provides some support for instantiating new web sites from existing ones. A Peerscape site may provide a link to another dataset that contains the basic files that each instance of that web service is supposed to have. A Peerscape user may copy those files over to an empty dataset to get a fresh instance of the site. Thus, to set up a photograph sharing site, a Peerscape user would go to a photograph site his friend is running and request the site to be cloned. A new dataset would then be created for that user, and the photograph site functionalities would be copied to that dataset. His friends photographs would not be copied, as they do not exist in the instantiation dataset. As a result, the user would get his own personal photograph sharing site, without any photos on it.

4.4 In Comparison

In this subsection we compare Peerscape and the three systems introduced in previous section with each other. We proceed by taking a quick recap for each system in turn considering certain features for each one of them. The features we have selected for consideration are: the storage location where the content is served from, incentive for the party storing and sharing the content, type of user interface, the way regular users locate content, possible ways how undesired access to the content might be gained, tracing down the publisher of the content, tracing down users who are accessing the content, efforts required to censor some content, efforts required to forge content published by someone else, efforts required to break content. Figure 7 summarizes the comparison.

In Persona the content is stored at a third-party datastore provider (DSP), who receives monetary payment from the user. The users use the service through some online social network (OSN). The means of locating content would then be OSN specific. Adversaries can access the content by tricking the publisher into giving access to them. This might involve getting a fake identity at the OSN. Protection against tracing publishers, downloaders, or censoring content depends on OSN and DSP. Both of them are to some degree capable of doing any of those. The ease of forcing some content on users depends on the OSN used. Content can disappear from the network if providers go out of business.

	Persona	Freenet	GNUnet	Peerscape
who store content	data store provider (DSP)	random peers	publisher and random peers	publisher and audience
sharing incentive	receiving monetary payment	helping real life friends	download performance improvements	personal interest in the content
user interface	online social network (OSN)	daemon serving web sites	desktop client	browser extension serving web sites
locating content	*OSN specific*	pick one of a few listed free sites, follow links	guess a keyword	integrated feed
undesired access	tricking publisher into giving access	through a leaked link	guessing a keyword	evesdropping transfers
tracing publisher	*OSN/DSP specific*	control lots of nodes at publication time	control lots of nodes at download time	look at the publishers public profile
tracing downloader	*OSN/DSP specific*	control lots of nodes at download time	control lots of nodes at download time	ask the peer discovery service for subscribers
censoring content	*OSN/DSP specific*	control large number of nodes	control large number of nodes	control each subscribers
counterfeit content	*OSN specific*	promoting a fake version of the free site	flooding keywords	creating a fake identity
corrupting content	providers may go out of business	data may disappear under heavy load	data may seem to be available after it has disappeared	peeers may stop being interested in the content and delete it

Figure 7: A Comparison between different protective platforms.

In Freenet the content is stored on random peers in the network. Nodes are expected to keep participating because they join the network to build an anonymous network with their real life friends. The system is accessed through a daemon serving web sites from the network. The web pages are viewable with a regular web browser. To find content a user picks one of a few free sites listed on the front page provided by the software publisher. Further content may be discovered by following links. The browsing experience is not too good, as retrieving web pages from the network is slower than retrieving web pages from the regular web. Friends who are building their own part of the network may have their own secret free site address passed among friends.

Adversaries of a free site can gain access to the free site by someone leaking a link to the site. To catch a publisher adversaries need to control a large number of nodes at publication time, and delimit possible origination places to some small part of the network. They will also need to monitor connections of their suspects to make sure they have managed to drive the suspects into a corner, and finally they need to go through the small amount of suspects by gaining control of their computers and examining them with care. Even so the published content is encrypted and the adversaries need to gain access to the link before upload of the content happens. This is practically impossible, and tracing down the publishers off-band is probably easier in most cases. Tracing down users retrieving the content may be easier, as the adversary may already know about the content, but control over a large part of the network is still required to reason about the way content is moving within the network. Censoring content is possible by planting the network full of censoring nodes which filter content. Censoring the network might not be possible in practice if the node operators actually know each other in real life, and connect directly to each other. Adversaries can try to forge free sites by creating fake versions of them and making them easier to find than the actual free site. Again, this does not work against a determined group of friends. The adversary can try to corrupt content by flooding the network. There is no guarantee for any data to remain stored in Freenet. Every stored bit will eventually get over written.

In GNUnet a publisher keeps sharing the content after publishing it. Random peers in the network cache the data and it might be available in the network for a while after the publisher has stopped sharing it. The nodes participating in storing and sharing content are rewarded by download performance improvements. Users access the system through a desktop client, and locate content by guessing a keyword used by the publisher. A url scheme has been defined, but there is no way of using this from a web browser today.

Adversaries may gain access to some content in GNUnet by guessing a keyword. Adversaries may try to locate a publisher by attempting to download the published file, taking control of the neighbouring node that seems to respond most eagerly to the requests, and continuing to compromise nodes in a similar manner until they reach a node that is sharing the content. This is by no means accurate as there might be areas in the network that have cached lots of data belonging to the shared content, and the original publisher might have stopped sharing the content. Locating a downloader requires following requests instead of responses. If the adversary has the possibility of sharing the targeted file, they should be able to get more requests, which helps in tracking down the origin of the requests. Still, tracking a downloader requires compromising all nodes on the path. Censoring content in GNUnet requires the adversary to control a large number of nodes, splintering the network into multiple subnets between which he can then filter content. Still filtering the content requires the original plain text. The attacker can try to force his content on users by flooding keywords for his content, making it impossible to find the actual content the users were searching for. Doing this requires the adversary to guess the keyword. While corrupting content in GNUnet is hard, the content may seem to be available after it has disappeared, as there may still be parts of the content available from caches. Even in cases where the missing piece of a file can finally be found in some hardly locatable cache, the decreasing speed towards the end of the download may disturb users because of the non-linear time conception of human beings [HAKB07].

In Peerscape the content is stored and shared by its publisher and audience. By looking at the content one joins the group of nodes sharing it. There is no technology to enforce this. It is thus possible to build a custom client that does not share any content, without any indirect drawbacks. The audience is expected to share the content because they have a personal interest in it. For example family members are expected to participate in sharing a family photo album. Peerscape users are required to install a browser extension which enables them to access web pages stored within Peerscape. Users get notified about new content in their subscribed datasets through a feed on their Peerscape front page, where they can follow links to access the content.

Adversaries can gain access to content in Peerscape by eavesdropping connections, as the traffic is not encrypted. They can identify a publisher by looking at his public profile. A typical Peerscape user reveals some identifying information to the public, which makes tracing the real life person possible. It is in theory possible to have an empty profile with no public information. In such cases there are probably other means of finding out who the publisher is. To locate downloaders of some content an adversary could join them in sharing, and continue asking the peer discovery service for subscribers to that content. To censor content the adversary would need to control all the subscribers. To force some content on users an adversary would need to create a fake copy of the publishers profile, and try to duplicate as much of the identifying information as possible. He could then compete with the actual publisher by publishing competing content. Keeping the original publisher off-line would probably help his cause. Content may disappear from the network, as users may stop being interested in the content and delete it from their node explicitly.

There are multiple different models one could consider for storing the data. The incentives proposed for different models differ a lot. All explored schemes had some connections to the web. The means used to locate content securely varied a lot. Figuring out the preferable ways remains an open challenge for future research. Freenet and GNUnet take the challenge of securing users much more seriously than Persona and Peerscape. Making the system really secure seems to require blurring the concept of content availability in the system. With Persona having lots of undefined parts, and no implementation, it is hard to say much about how various details in it would work out in practice. Persona is designed to prevent undesired access to content, which it seems to be doing. Peerscape takes measures to protect names and user identities, but provides very limited protection against hostile network environments.

5 Application Prototypes

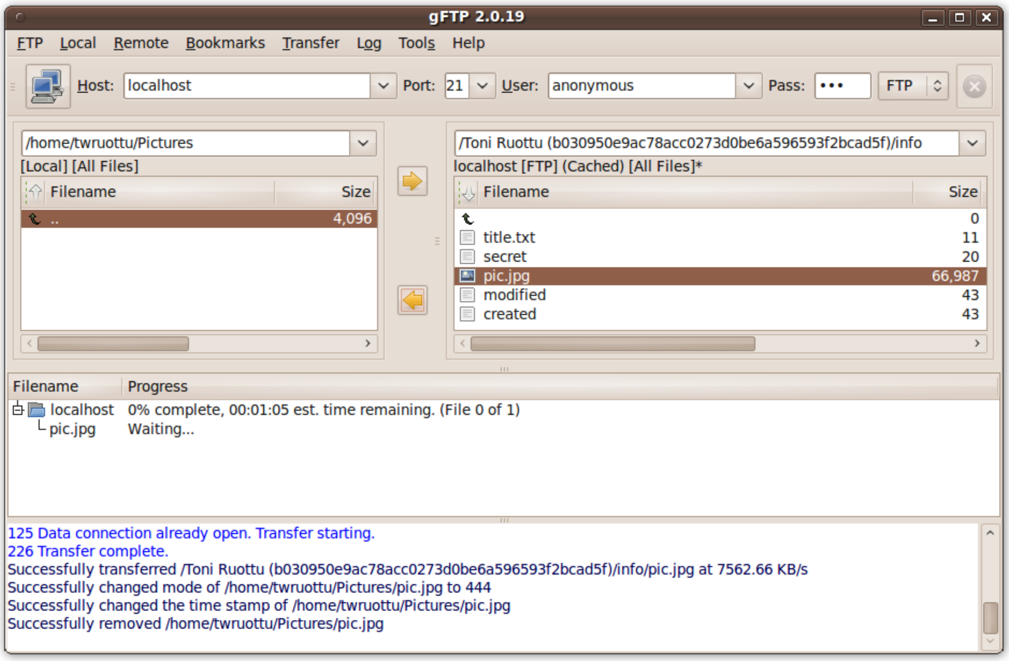
As we got the Peerscape platform up and running, it was finally possible to get into writing applications for it. Actual Peerscape applications were supposed to run in the browser sandbox. We did not want to force our users to make decisions about which software to trust. Users already had to make such decisions with desktop software and Facebook applications. We wanted to minimize the decisions instead of introducing new ones. As we were writing our prototype applications, we occasionally allowed ourselves to extend the platform. Mainly so we could find out what the platform might need to support.

We will continue by discussing development of four example applications. We start in subsection 5.1 by developing an application used for storing files in datasets. In subsection 5.2 we implement our first distributed application, a simple public chat room. In subsection 5.3 we go on and try to develop a distributed Bittorrent site. In subsection 5.4 we write a minimal game for the platform. Finally, in subsection 5.5 we discus how removed items might affect application design.

5.1 Storing Websites in a Database

Once we started creating applications for the peer-to-peer web we noticed that storing files into datasets was cumbersome. Transferring large amounts of files by writing code in the Python shell was not convenient. We also needed an easy way for transferring web pages to datasets, so we could test them out in the wild immediately after we had written our changes to them. During the project there were various attempts to solve this particular issue.

First we wrote a command line tool called Subfusion (sfn). The name was a word



Host: localhost Port: 21 User: anonymous Pass: •••

/home/twruottu/Pictures
[Local] [All Files]

Filename	Size
..	4,096

/Toni Ruottu (b030950e9ac78acc0273d0be6a596593f2bcad5f)/info
localhost [FTP] (Cached) [All Files]*

Filename	Size
	0
title.txt	11
secret	20
pic.jpg	66,987
modified	43
created	43

Filename	Progress
localhost	0% complete, 00:01:05 est. time remaining. (File 0 of 1)
pic.jpg	Waiting...

```
125 Data connection already open. Transfer starting.
226 Transfer complete.
Successfully transferred /Toni Ruottu (b030950e9ac78acc0273d0be6a596593f2bcad5f)/info/pic.jpg at 7562.66 KB/s
Successfully changed mode of /home/twruottu/Pictures/pic.jpg to 444
Successfully changed the time stamp of /home/twruottu/Pictures/pic.jpg
Successfully removed /home/twruottu/Pictures/pic.jpg
```

Figure 8: Transferring files between local file system and Peerscape using gFTP ftp client.

play on P2P-Fusion, the EU project that funded most of our work, and Subversion which we used for version control [fus10a][svn10]. The name was a bit misleading since the tool had very little to do with version control. In fact the biggest similarities were cosmetic. We would "checkout" a dataset into the local file system and "commit" our changes back into the dataset.

Subfusion made moving files into datasets easier, but you would still have to commit after each change. Thus we moved on and wrote Fuse support for Peerscape. Fuse would allow us to mount a dataset as a part of the local file system in Ubuntu[fus10b]. However, majority of people within the project were using Apple computers. Google had ported Fuse to the Mac, but installing Fuse on Mac was not too easy at the time. To make dataset mounting work on a wider scale of platforms without lots of extra work, it was decided that Webdav would be used instead of Fuse. An existing Webdav server was ported to a Peerscape python module. But it never became too popular amongst our ranks.

At times we discussed integrated development environments (IDE), that would allow users to develop the platform from within the platform. We made some short lived tests by creating a web site that had a web form for editing its own html code. Every once in a while it would break, and needed fixing with the other tools that happened to be around. Creating high quality tools takes time, and we already had our favourite tools in the desktop world. Thus, we ended up using file transfer mechanisms more than our early IDE prototypes.

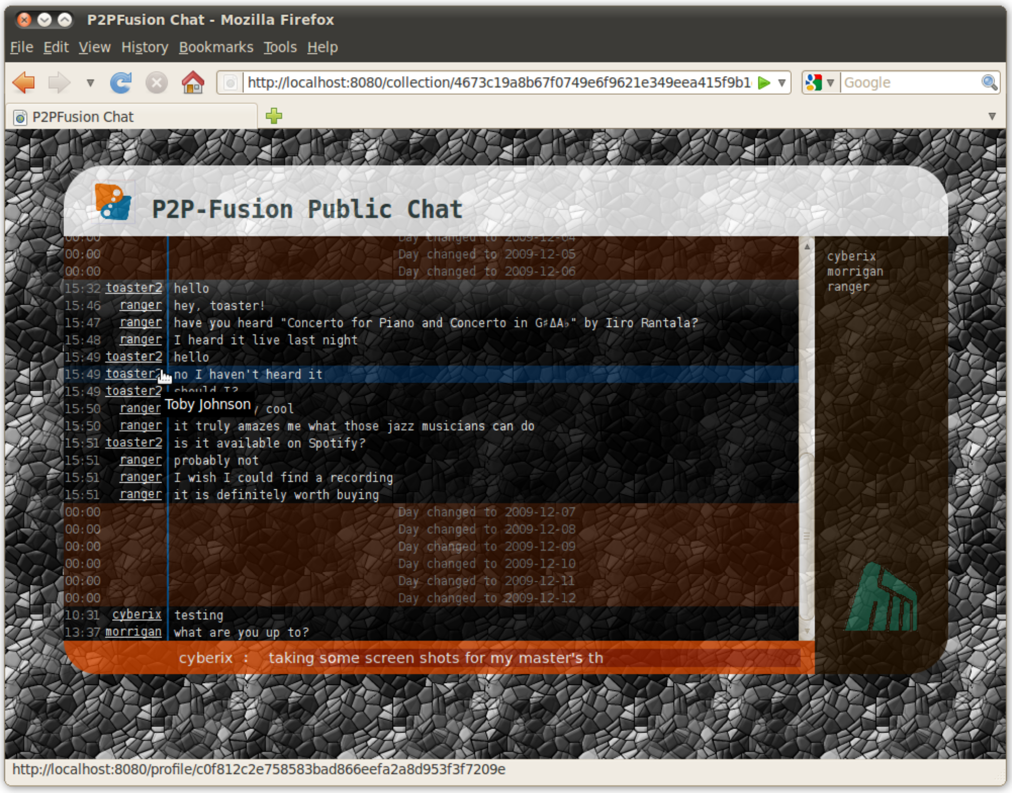
Finally, we wrote a minimalistic ftp server module that enabled us to transfer files into datasets with a standard ftp client. While mounting ftp servers is possible, the mounting services turned out to rely on some advanced features that our implementation did not have. Using ftp still worked fine for file transfers, and the Peerscape daemon would conveniently carry the ftp server around. On a sudden urge to upload a file, the ftp server would always be there. Figure 8 show a screen shot of transferring files to Peerscape using ftp.

The described solutions for storing files in a database do not follow the security model that Peerscape applications typically would. Being trusted and written in Python they are more like parts of the infrastructure. However studying them gives us some idea about the ways in which the platform itself may evolve. As we continue to explore further applications we try to minimize the use of trusted Python components, and find out how hard living inside the browser sandbox really is.

5.2 Persistent Real-time Messaging

Peerscape chat was among the first prototype applications that we wrote to experiment with the platform. We were expecting to encounter some problems, as our system did not have a conception of time. Once we realized the messages could appear early or late during normal operation, we altered the chat to support updating of the past message log. We added pseudo system messages to underline lack of recent activity in a chat. While we had cryptographic identities for our users, protecting user identities in the user interface turned out to be problematic.

Figure 9 is a screen shot of the end result. The white bar at the top shows the title and icon for the chat room. Below that we have a message area on the left, and a list of users currently present in the chat room on the right. At the bottom we display the user's nickname, and provide a box for writing chat messages. User nickname can be edited in place by clicking on it. The message area has a list of messages. Each message starts with a timestamp, and sender nickname, and is followed by


P2PFusion Chat - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/collection/4673c19a8b67f0749e6f9621e349eea415f9b1
Google
P2PFusion Chat
P2P-Fusion Public Chat

```
00:00                Day changed to 2009-12-05
00:00                Day changed to 2009-12-06
15:32 toaster2 hello
15:46   ranger hey, toaster!
15:47   ranger have you heard "Concerto for Piano and Concerto in G♯ΔA♭" by Iiro Rantala?
15:48   ranger I heard it live last night
15:49 toaster2 hello
15:49 toaster2 no I haven't heard it
15:49 toaster2 should I?
15:50   ranger Toby Johnson, cool
15:50   ranger it truly amazes me what those jazz musicians can do
15:51 toaster2 is it available on Spotify?
15:51   ranger probably not
15:51   ranger I wish I could find a recording
15:51   ranger it is definitely worth buying
00:00                Day changed to 2009-12-07
00:00                Day changed to 2009-12-08
00:00                Day changed to 2009-12-09
00:00                Day changed to 2009-12-10
00:00                Day changed to 2009-12-11
00:00                Day changed to 2009-12-12
10:31  cyberix testing
13:37 morrigan what are you up to?

cyberix  :  taking some screen shots for my master's th
```

```
cyberix
morrigan
ranger
```

http://localhost:8080/profile/c0f812c2e758583bad866eefa2a8d953f3f7209e

Figure 9: Chat Prototype

a message from that user. Holding the cursor on top of a nickname will show the user's real name in a tooltip. Clicking on the nickname of a user takes you to his profile. The blue highlight under one of the messages is a ruler helping the user match nickname links with the messages. Some messages lack nicknames. These are system messages, and are coloured brownish to be easily distinguishable.

We were expecting some problems, as our system did not have a conception of time. The receiving end would not be able to tell when the messages had been written to the database. To cope with this limitation we stored publication time in the key of each chat message. We would then poll for all message keys at certain intervals, and filter out the messages that we had already printed out. We also wanted to display who was currently online, so we stored the current time at an interval under a presence key that identified the user. We would then repeatedly load all presence keys to see the time when the users had been online. Having all presence keys, we would list the users who had recently updated their presence time stamp, and show that to the user.

When we realized the messages could appear late during normal operation, we altered the chat to support updating of the past message log. Thus while looking at Peerscape chat one could observe messages popping up in the middle of the chat screen, or even in the message history not currently visible on screen. Some big changes were required in the core functionality of our original chat engine to support messages appearing in the middle. Even after these modifications our chat application would not visualize all possible state transformations of the underlying data model. Another chat application could support removing messages from the chat history. Our webpage would not hide such deleted messages until the webpage got reloaded.

We added pseudo system messages to underline lack of recent activity in a chat. As the chat messages were stored in a database, they persisted over restarting the application. Low volume chat windows would occasionally contain messages from multiple days, with the most recent message being sent one month ago. As a result it was hard to see how active the chat had recently been. We wanted the system to add notifications to the message log when day changed, except that the peer-to-peer philosophy was against such centralized trusted parts. We solved the problem by computing the "day has changed" messages for each chat client separately. We had to call the function twice for each arriving message to determine preceding and following day changes for that message.

While we had cryptographic identities for our users, protecting user identities turned out to be problematic. Messages were always signed by the system, and we linked to the authors profile from his nickname, yet it was up to users to follow that link for verification. Even, when the user would do just that, an attacker could have forged the linked profile by copying over as much identifying information as possible from the real profile. We felt that truly securing identities would require further measures, like building a cryptographic web of trust between users. We considered using different colours for similar nicknames with different profiles, but researching the subject at this level was beyond the scope of our research.

5.3 Distributed Bittorrent Tracker

Peerscape bittorrent tracker consisted of some python code that modified the Peerscape platform to support the bittorrent tracker standard, and a Peerscape application for browsing/sharing torrent files.

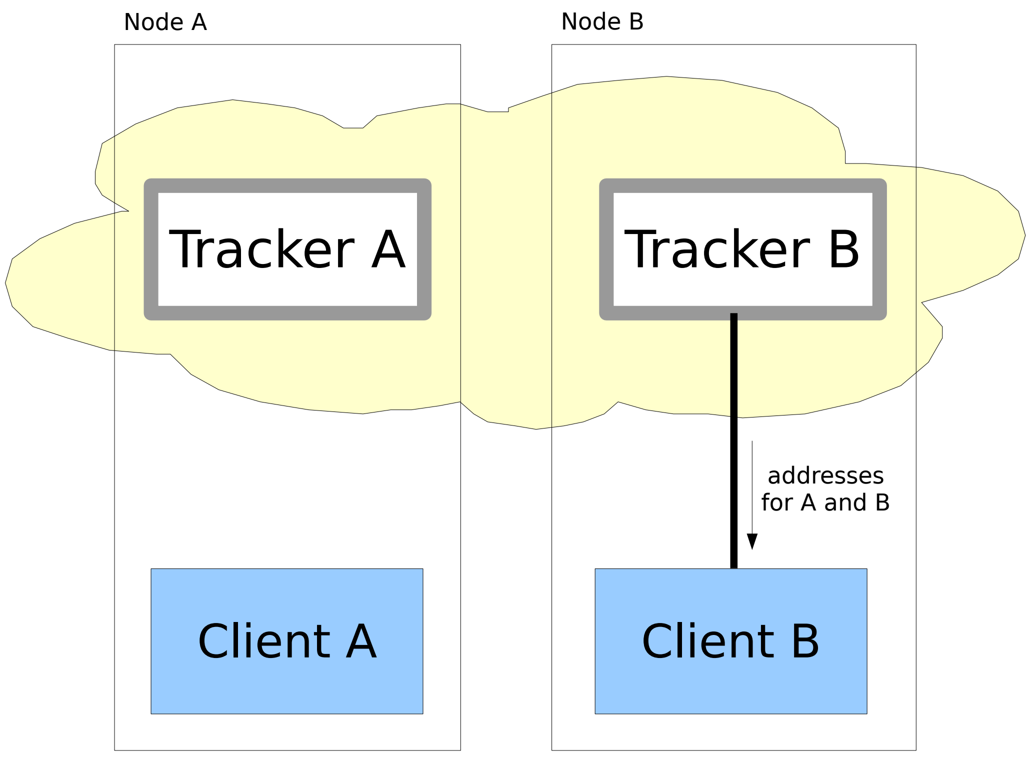


Figure 10: Distributed Tracker

The architecture of our distributed tracker is depicted in Figure 10. There are two nodes A and B who wish to share certain content with each other. Both nodes run a bittorrent client, for file sharing, and an instance of the distributed tracker, for discovering other clients. In the figure, Client B has just formed a connection to its local tracker and is about to receive peer advertisements about, both itself, and Client A. The trackers are in a synchronizing yellow cloud that represents Peerscape datastore. It is the responsibility of the datastore to make sure all trackers have all peer advertisements.

The tracker code listens for client connections under a specific tracker subpath of the http server. At connection time it stores the local peer's address into the distributed data base, and returns a set of addresses from the database. We identified a few challenges that we believe to be different from the centralized world.

Bittorrent trackers often detect client IP address by looking at the incoming connections. This does not work when the client is making a connection from 127.0.0.1, as it is a private IP address which is not useful to other nodes on the public Internet. Luckily the localhost address indicates that the client is running on the same host as the tracker. The tracker can then find out its own external IP address and replace 127.0.0.1 with that. Some bittorrent clients also support resolving IP address and suggesting that to the client. Relying on the client for IP address detection may sometimes be considered harmful, as this gives the user opportunity to advertise some innocent 3rd party nodes in order to cause them unwanted traffic. This is of course not the case when the user is running his own bittorrent tracker himself, as he can alter the address at will.

Unfortunately bittorrent trackers can not advertise multiple addresses for a single peer. Some of the addresses that the Peerscape knows about might not work. It may also be impossible for Peerscape to reason about the best possible address to be advertised. Choosing the best set of addresses to be advertised is left as an open research question. It may turn out that architectural changes in bittorrent trackers are preferable, as the client using the address has usually better opportunity to reason about which addresses are likely to be useful.

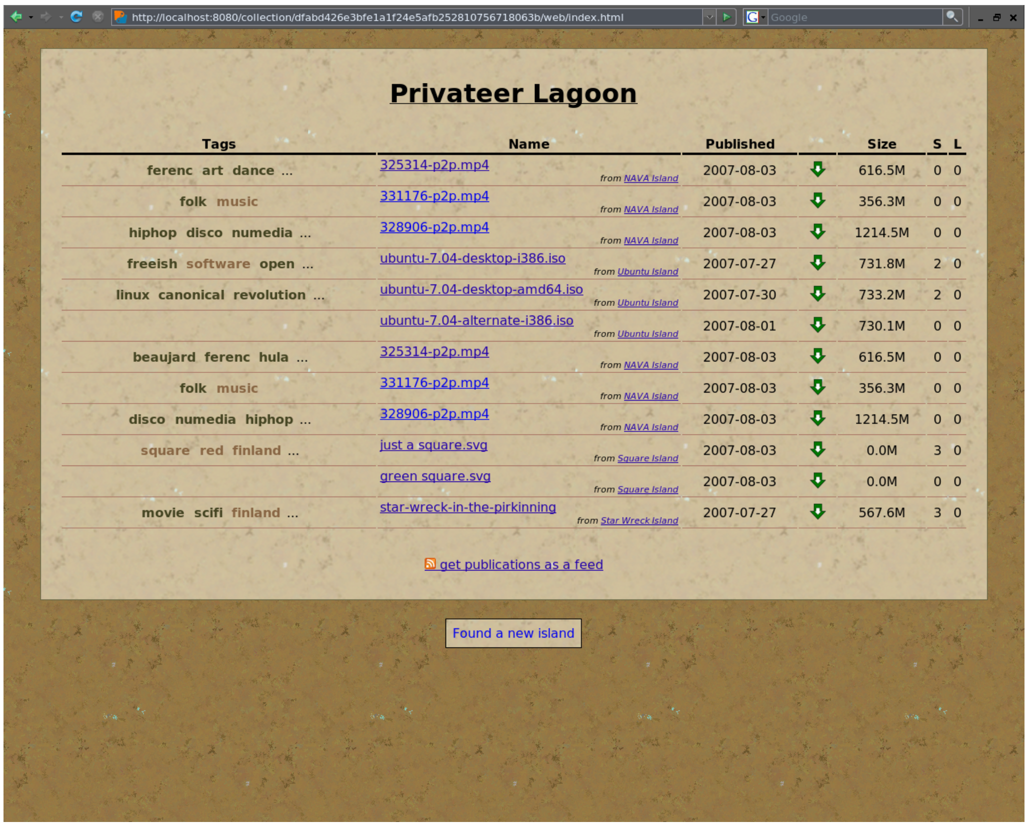
On occasions the contents of a distributed database change more rapidly than the contents of a centralised datastore used by centralised bittorrent trackers. Lets consider a use case where the user has received a torrent file that points to a Peerscape bittorrent tracker. The tracker dataset containing the node information might not exist in the local database. The user opens the torrent file with his bittorrent client. The client connects to the local tracker. The tracker subscribes to the tracker dataset and starts receiving the dataset from the network. The tracker starts receiving oldest database entries containing old peer advertisements, that have been time-outed and are no longer valid. Since the tracker already received authorization he stores his IP address into the database. Then the user receives a list of available nodes. The list contains only the users own node, as the synchronization has not finished. The issue is made worse by many bittorrent clients having relatively long update frequencies. In the centralized case multiple people would be using the same tracker and aggressive polling might cause an unintended distributed denial of service attack against the server. When everyone runs their own tracker there is no such problem. Also network capacity is not a problem when both server and client use localhost interface for communicating to each other. Thus the original precautions are unnecessary, or even counter productive.

The distributed datastore may be detrimental to user privacy. Any user could modify his node to record all peer advertisements. While using a centralized tracker an attacker would need to poll the tracker constantly and he could never be sure if he had actually received all possible advertisements, as the centralized tracker could play tricks on him. The tracker could repeatedly give him some specific set of nodes chosen by a secret algorithm. With a distributed database such facades do not exist. Using a distributed datastore for tracking may also, in some cases, improve user privacy by making it easy to run private trackers. Private trackers are possible with centralized solutions as well, but a peer-to-peer design requires user-friendly administration which may lower the skill requirement for starting new private trackers.

Even when there is no one recording tracker history, some peer advertisements might be left in the database by accident. Ordinary users do not have rights to remove peer advertisements added by other users. The advertisements are removed whenever the local client reports a finished download. Reporting this is optional by the specification, and some clients might not do it. Some clients might also crash before asking the tracker to remove the advertisement. The tracker does not know about ongoing file transfers, so it cannot remove advertisements as the transfers complete. While it is possible for the tracker owners to remove any advertisement they wish at any time, this is in no way peer-to-peer, and the incentives for garbage collection as a service are beyond the scope of this thesis.

Bittorrent based publishing services are typically implemented as centralized web services known as torrent sites. Examples of such sites include The Pirate Bay. We wrote two torrent site applications. One was designed for sharing one's personal files with a small trusted group. We ended up calling this application *island*, due to its small and isolated nature. The other application was used for facilitating public file sharing. This application was named *coast*, representing a place where "the islanders" could publish their works to a wider audience. The island application would contain all the meta data and do the actual bittorrent tracking. The coast application would simply store links to islands and display aggregates of the meta data stored on those islands. Running a bittorrent tracker for the coast was not required as all the shared files already had a tracker on their home island. A coast tracker might have helped in some corner cases where the tracking dataset for some obscure island would have zero copies online.

Figure 11 is a screen shot of an example coast called *Privateer Lagoon*. It is a web page containing a table of files shared by users. Right below the list of files is a link


http://localhost:8080/collection/dfabd426e3bfe1a1f24e5afb252810756718063b/web/index.html

Privateer Lagoon

Tags	Name	Published		Size	S	L
ferenc art dance ...	325314-p2p.mp4 from NAVA Island	2007-08-03	⬇	616.5M	0	0
folk music	331176-p2p.mp4 from NAVA Island	2007-08-03	⬇	356.3M	0	0
hiphop disco numedia ...	328906-p2p.mp4 from NAVA Island	2007-08-03	⬇	1214.5M	0	0
freeish software open ...	ubuntu-7.04-desktop-i386.iso from Ubuntu Island	2007-07-27	⬇	731.8M	2	0
linux canonical revolution ...	ubuntu-7.04-desktop-amd64.iso from Ubuntu Island	2007-07-30	⬇	733.2M	2	0
	ubuntu-7.04-alternate-i386.iso from Ubuntu Island	2007-08-01	⬇	730.1M	0	0
beaujard ferenc hula ...	325314-p2p.mp4 from NAVA Island	2007-08-03	⬇	616.5M	0	0
folk music	331176-p2p.mp4 from NAVA Island	2007-08-03	⬇	356.3M	0	0
disco numedia hiphop ...	328906-p2p.mp4 from NAVA Island	2007-08-03	⬇	1214.5M	0	0
square red finland ...	just a square.svg from Square Island	2007-08-03	⬇	0.0M	3	0
	green square.svg from Square Island	2007-08-03	⬇	0.0M	0	0
movie scifi finland ...	star-wreck-in-the-pirkinning from Star Wreck Island	2007-07-27	⬇	567.6M	3	0

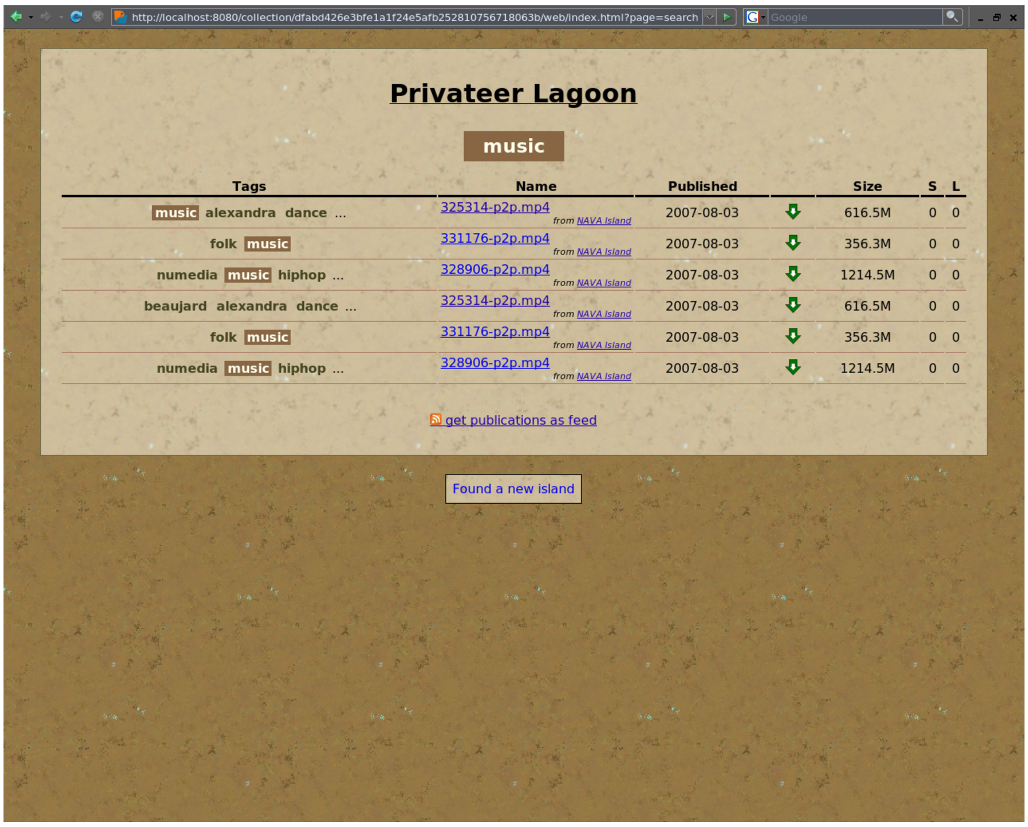
get publications as a feed

Figure 11: Bittorrent Index

pointing to an atom feed of shared files. On the bottom is a link for founding new islands. Islands that are created from a coast are automatically linked with it.

The file table itself is organised as follows. For each shared file it contains some of the most common user assigned tags, the name of the shared file, the home island of the file, date when the file was first introduced to the system, a torrent-download icon, size of the file, number of users currently seeding the file, and number of leechers in the middle of downloading the file.

Clicking on some of the more active looking parts will cause things to happen. Clicking on a tag filters out files not carrying the tag in question. A filtered view can be seen in Figure 12. Clicking on the file name displays a more detailed file view, described in the next paragraph. Clicking on the home island link takes user to the tracker site where the file was originally introduced. Clicking the torrent-download icon will download the torrent-file used for retrieving the content.


http://localhost:8080/collection/dfabd426e3bfe1a1f24e5afb252810756718063b/web/index.html?page=search

Privateer Lagoon

music

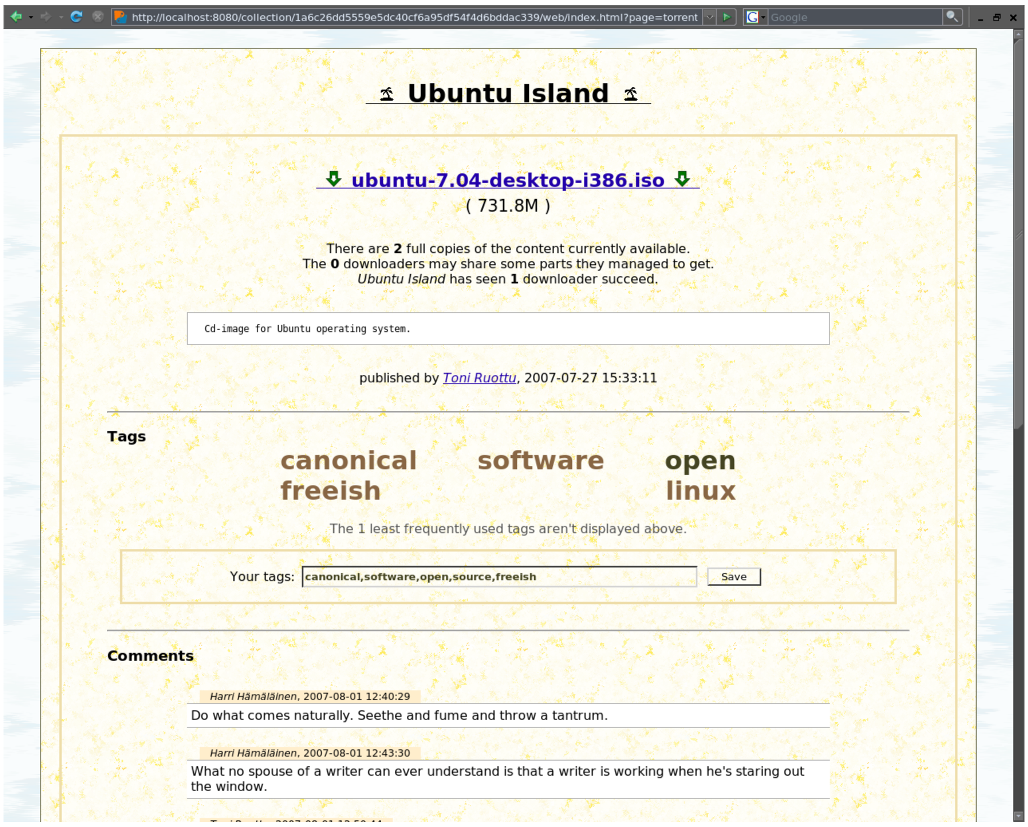
Tags	Name	Published		Size	S	L
music alexandra dance ...	325314-p2p.mp4 from NAVA Island	2007-08-03	⬇	616.5M	0	0
folk music	331176-p2p.mp4 from NAVA Island	2007-08-03	⬇	356.3M	0	0
numedia music hiphop ...	328906-p2p.mp4 from NAVA Island	2007-08-03	⬇	1214.5M	0	0
beaujard alexandra dance ...	325314-p2p.mp4 from NAVA Island	2007-08-03	⬇	616.5M	0	0
folk music	331176-p2p.mp4 from NAVA Island	2007-08-03	⬇	356.3M	0	0
numedia music hiphop ...	328906-p2p.mp4 from NAVA Island	2007-08-03	⬇	1214.5M	0	0

get publications as feed

Figure 12: Filtering The View

Figure 13 shows the file view. The file view is located on the file's home island even when it is reached through a link from a coast. Thus, for each published torrent file, there is only one copy of the file view. At the top of a file view there is a prominent link for downloading the torrent-file. Below the link we display the file size and some simple statistics regarding status of sharing. Then a white text box with information provided by the sharing entity, and a publishing signature. The rest of the page contains social enrichment features. There is a tag cloud showing some of the most popular tags added to the file by users. The current user is also given the chance to provide tags for the file. Below the tag area there is a comment area displaying user provided comments. Finally at the bottom, not visible in the screen shot, the user is given a chance to add comments.

Figure 14 is a screen shot of an island. An island has a list of published files like a coast does, with the exception that the home island of a file is never displayed as the current island always serves as the home island for the files. The list of files


http://localhost:8080/collection/1a6c26dd5559e5dc40cf6a95df54f4d6bddac339/web/index.html?page=torrent

Ubuntu Island

ubuntu-7.04-desktop-i386.iso

(731.8M)

There are **2** full copies of the content currently available.
The **0** downloaders may share some parts they managed to get.
Ubuntu Island has seen **1** downloader succeed.

Cd-image for Ubuntu operating system.

published by *Toni Ruottu*, 2007-07-27 15:33:11

Tags

canonical software open
freeish linux

The 1 least frequently used tags aren't displayed above.

Your tags: canonical,software,open,source,freeish Save

Comments

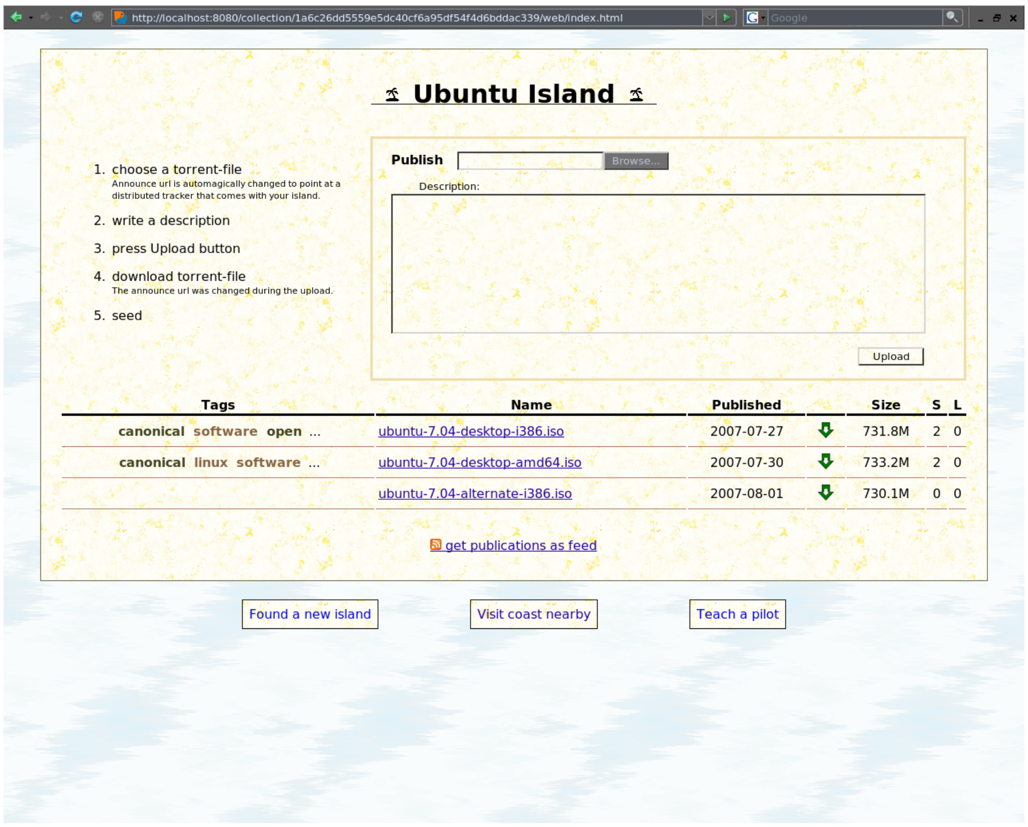
Harri Hämäläinen, 2007-08-01 12:40:29
Do what comes naturally. Seethe and fume and throw a tantrum.

Harri Hämäläinen, 2007-08-01 12:43:30
What no spouse of a writer can ever understand is that a writer is working when he's staring out the window.

Figure 13: Filtering The View

published on an island is also available as an atom feed. In addition, when viewed by the owner, an island provides a web form for publishing new files. An island has three administrational links at the bottom of the view. The one on the left is used for founding new sister islands — islands that share the same coast for making files visible to a larger audience. The one in the middle takes you to the coast where the island was originally created. The one on the right enables one to publish the files on some additional coasts as well.

Making the applications efficient turned out to be difficult. Regular bittorrent sites can do heavy calculations beforehand and store the results in a local database. Our javascript application ended up calculating tags on each page refresh as the user did not have sufficient rights for storing the computed results in the application dataset. Local caching of the results might have saved some resources, but keeping the cache fresh at each node separately would have eaten a serious amount of resources. Implementing such a cache could have benefited from support for exe-


http://localhost:8080/collection/1a6c26dd5559e5dc40cf6a95df54f4d6bddac339/web/index.html

Ubuntu Island

1. choose a torrent-file
   Announce url is automagically changed to point at a distributed tracker that comes with your island.
2. write a description
3. press Upload button
4. download torrent-file
   The announce url was changed during the upload.
5. seed

Publish Browse...

Description:

Upload

Tags	Name	Published		Size	S	L
canonical software open ...	ubuntu-7.04-desktop-i386.iso	2007-07-27	⬇	731.8M	2	0
canonical linux software ...	ubuntu-7.04-desktop-amd64.iso	2007-07-30	⬇	733.2M	2	0
	ubuntu-7.04-alternate-i386.iso	2007-08-01	⬇	730.1M	0	0

get publications as feed

Figure 14: Personal Bittorrent Tracker Prototype

cuting cache updating functions in the background, while the user was looking at other content. With browsers lacking support for such background processes, cache updates would happen when the user first entered the page.

Developing the bittorrent site application required some special support from the platform. The required features were very bittorrent specific, and never made it to the later versions of the platform. The features included the actual bittorrent tracking, and providing status information for the shared torrents. Implementing bittorrent trackers might not be a good area for applying Peerscape, because it would need to be implemented as a part of the platform. From a technological point of view it seemed to work. We tested simple sharing and downloading with a few clients. We did not measure the tracker for performance. One possible way of taking the tracker work forward might include implementing a federated tracker as a standalone application based on similar principles. Unlike the tracker, the bittorrent index did not require special support from the platform, and rewriting the index for Peerscape might make sense. In such a case the index web pages would be served from a different origin than the tracker. Thus the javascript that implemented the index could only access sharing statistics of trackers that made the statistics available with Cross-Origin Resource Sharing[vK10].

5.4 Non-Competitive Real-time Game

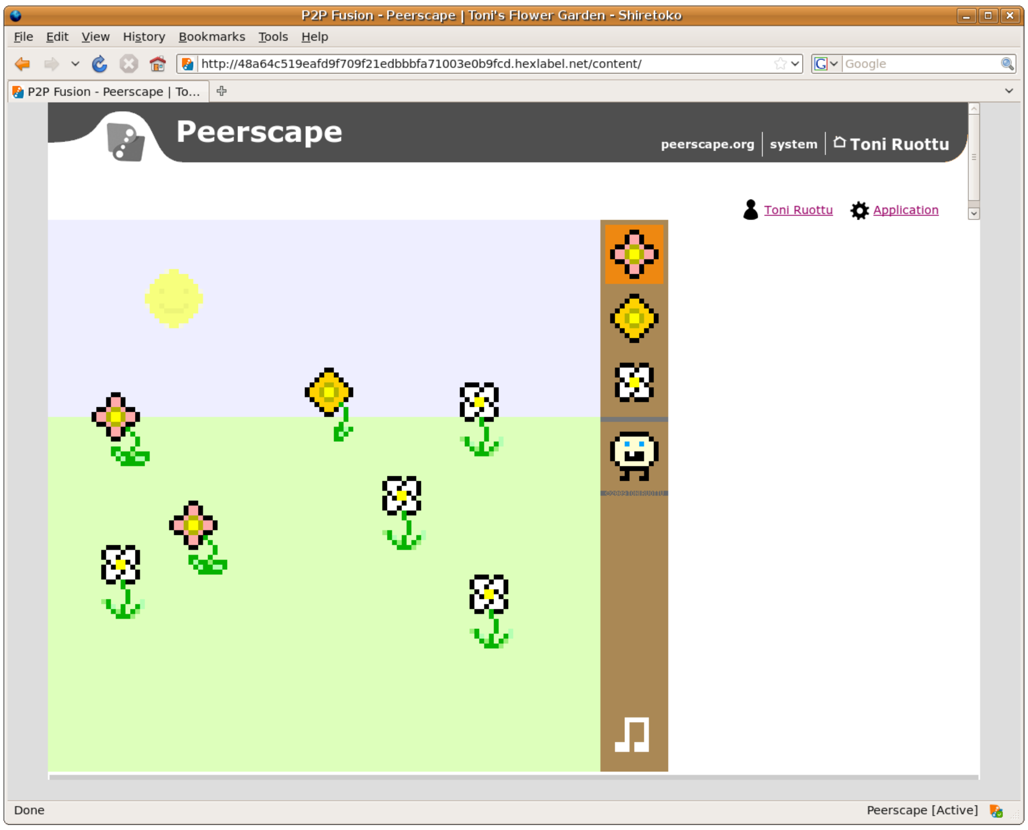

P2P Fusion - Peerscape | Toni's Flower Garden - Shiretoko
File Edit View History Bookmarks Tools Help
http://48a64c519eafd9f709f21edbbbfa71003e0b9fcd.hexlabel.net/content/
Google
P2P Fusion - Peerscape | To...
Peerscape
peerscape.org
system
Toni Ruottu
Toni Ruottu
Application
Done
Peerscape [Active]

Figure 15: Adding Flowers

We wanted to find out how hard it would be to implement a simple game for Peerscape. We decided to go for a collaborative flower garden that would not provide a final goal for winning the game, but would allow the users to define their own goals. The word *game* is heavily overloaded. In this context we define a game as an interactive multimedia experience, where users interact with each other and a system that implements a simple set of rules. We concentrate on the immersive aspects, and worry less about bringing up interesting game theoretical puzzles that other types of games might have.

Figure 15 is a screen shot of the resulting game. The game lives inside an iframe on a Peerscape page. The right-hand side of the game screen has a brown tool bar. The remainder of the game screen is divided into two parts. Top part is blue and represents sky. The bottom part is green, and represents a grass field where flowers can be planted.

The tool bar contains three different flowers that users can plant to the garden. Any one of the three flowers can be selected at any time. Moving the cursor over an unselected flower changes the cursor into a finger, encouraging the user to select the tool. Selecting the tool will result in a dark orange box being drawn behind the icon of the selected tool. In Figure 15 the top most flower has been selected. The currently selected tool is stored in a cookie and remembered next time the user opens up the game. Moving the cursor above grass, while a flower tool is selected, makes the cursor turn into a cross hair, making it easier to select the exact location for planting a flower. Clicking on the grass will make a flower appear, as if it had grown from the point where the user clicked. When the new flower appears a sigh is played, providing the user with audial feedback. Care is taken to make sure the view presented to the user is in sync with the model stored in the database. Thus adding a flower may sometimes take a moment.

Below the three flower tools there is a flower eater who can be used to remove flowers from the garden. The flower eater is selected in a similar manner to selecting one of the flower tools. Selecting the flower eater deselects any previously selected flower tool. In Figure 16 the flower eater has been selected. Selecting the flower eater causes boxes to appear around flowers, turning the flowers into white, partially transparent, cards. Moving mouse cursor on a card will then turn the card red, which is what has happened to one of Ken's flowers in the example screen shot. Clicking on the card will turn the colour into a more intense red and bring the card to the front if there are other cards obscuring it. Upon release the click will cause the flower to disappear and a clomp sound from the flower eaters bite can be heard.

At the bottom of the tool bar is a picture representing beamed pair of eighth notes. The picture serves as a toggle button allowing user to turn music on and off. Depending on the local system clock, a different song is played. One of the songs, titled *Flower Growing Days*, is played during day time, from 9:00 to 21:00. It is rather happy, crisp, and fast march music. The other song, titled *Flower Growing Nights*, is played during night time, from 21:00 to 9:00. It is slower, calmer and more jazzy

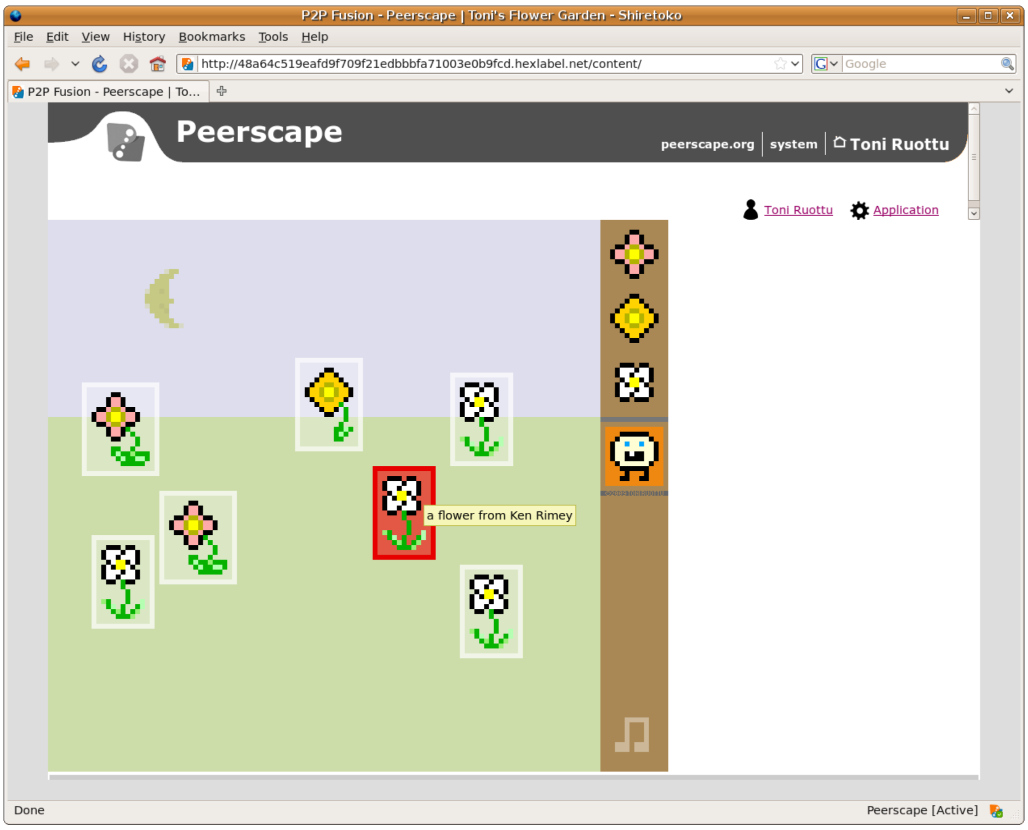

P2P Fusion - Peerscape | Toni's Flower Garden - Shiretoko
File Edit View History Bookmarks Tools Help
http://48a64c519eafd9f709f21edbbbfa71003e0b9fcd.hexlabel.net/content/
Google
P2P Fusion - Peerscape | To...
Peerscape
peerscape.org
system
Toni Ruottu
Toni Ruottu
Application
a flower from Ken Rimey
Done
Peerscape [Active]

Figure 16: Removing Flowers

than the one played during day time. Both songs are included as sheet music in Appendix 2.

In addition to the changes in music, parts of the colour scheme, and the displayed orbital also change accordingly. The screen shot for Figure 15 was taken during day time. It has bright colours and the sun visible in the sky. Screen shot for Figure 16 was taken during night time. It has dimmer colours, and a moon instead of the sun is visible in the sky. The beamed pair of eighth notes is also dimmer, but this is because the user has turned off the music, and not an effect of the day time change.

We wanted users to experience the presence of other users, while they were playing the game. When some user added a flower, it would pop up on screens of other users who were looking at the same instance of the game. Because of occasional network splits some flower updates got cached on some nodes, but were not able to get to the other side of the split. When the split ended the old flowers would get synchronized to the other side. Some existing games, like the ESP game [vA06a][vA06b], record multi-player games played by real users, and use those to simulate user behaviour when no players are available. ESP game displays an image to two randomly selected players and the players have to find a common tag for the image. When there are no free players available, old games from real players are replayed providing the user with an opponent. The behaviour of cached flower garden updates is similarly replaying actions made by an actual human player, and may help by making the game more interesting.

When the players are actually in the same room they may notice odd behaviour, but most of the time the game is either working properly or not at all. If the computers are able to connect to each other the game works. If the computers are not able to connect to each other it does not. An exception is made if there are third parties online who share the same dataset. If the amount of connectable third parties is somewhat stable the game will either work or not. When there is only one reachable third party who keeps leaving and joining the network, the experience may feel inconsistent.

While our flower game does not in itself have any prescribed goal, users may create goals for themselves. One user might decide to trim the garden, by removing flowers from the chaotic flower benches. Some evil user might try to distort the feng shui of the garden by adding millions of red flowers on top of other flowers, while another might hate flowers all together and try to remove them all. In such situations the network conditions may support some goals over others, but this is not a problem as the users who chose hard goals knew what they were opting for.

Our goal was to create a simple game for Peerscape. By implementing the flower game we have shown that it is possible, with relatively small effort, to create an audio visual multi-player game. While there are various restrictions compared to developing regular web applications, lot of the knowledge gained by developing traditional web applications may be reused. We have avoided the problem of making the game fair by turning the game into an innovation platform where user created games can be played out. We realize that this small experiment does not prove all types of games to be well suited for our target platform. We, however, hope that the experiment has shed some light to the possibilities of using Peerscape as a platform for implementing simple games.

5.5 Minding Tombstones

Items stored in a Peerscape dataset can never truly be deleted. The API supports deleting items, but in the background a tombstone is always kept around. A tombstone contains a key, but has no value. As a result the amount of space required to store the key is irreversibly wasted when content is written to the key for the first time. This may not become a problem, unless the system is attacked by using huge keys. An answer to such attack for everyone storing the dataset is to remove that dataset. Users need to be able to remove datasets from their disks anyway, so this is not much of a problem. However choosing suitable keys may become a problem when designing some applications.

Lets consider the case of storing some text into Peerscape. Storing all of the text under one key would cause simultaneous users to overwrite each others changes to the content. Because of how Peerscape is implemented, one of the users would win the conflict. The situation would be boring for the other user. With longer edits the users would be more likely to overwrite each others changes. It seems likely that losing results of longer edits would be more painful than losing results of short edits. Storing shorter pieces, say paragraphs, would probably make the collisions less frequent.

If we decide to split the content into paragraphs, we have to deal with keeping the paragraphs in the right order. We could easily store paragraphs in the right order by having the alphabetic order of the keys tell the order of the paragraphs on the screen. Removing paragraphs would simply be done by deleting the corresponding entry from the database. Adding paragraphs would require finding a key which would position the new paragraph alphabetically in the correct place. Moving a paragraph would require deleting the paragraph from its original key and rewriting it to another key corresponding to its new location. While looking at the new and old version of the text does not tell us which parts have been moved, we can use the *longest common subsequence* algorithm to decide this. As a result we might end up moving different, but from the data models perspective more optimal, paragraphs than the ones the user actually moved.

Lets take a deeper look into assigning keys for paragraphs, and see just how bad the situation is. To simplify the issue into a more general one, let us assume the paragraphs are placed on a line segment between 0 and 1. We use the position on the segment as a key when storing the paragraph. A paragraph can never have the index 0, as that would make it the top most paragraph and placing paragraphs above it would be impossible. Similarly, a paragraph can never have the index of 1, as it would then be the last possible paragraph, placing paragraphs after it being impossible.

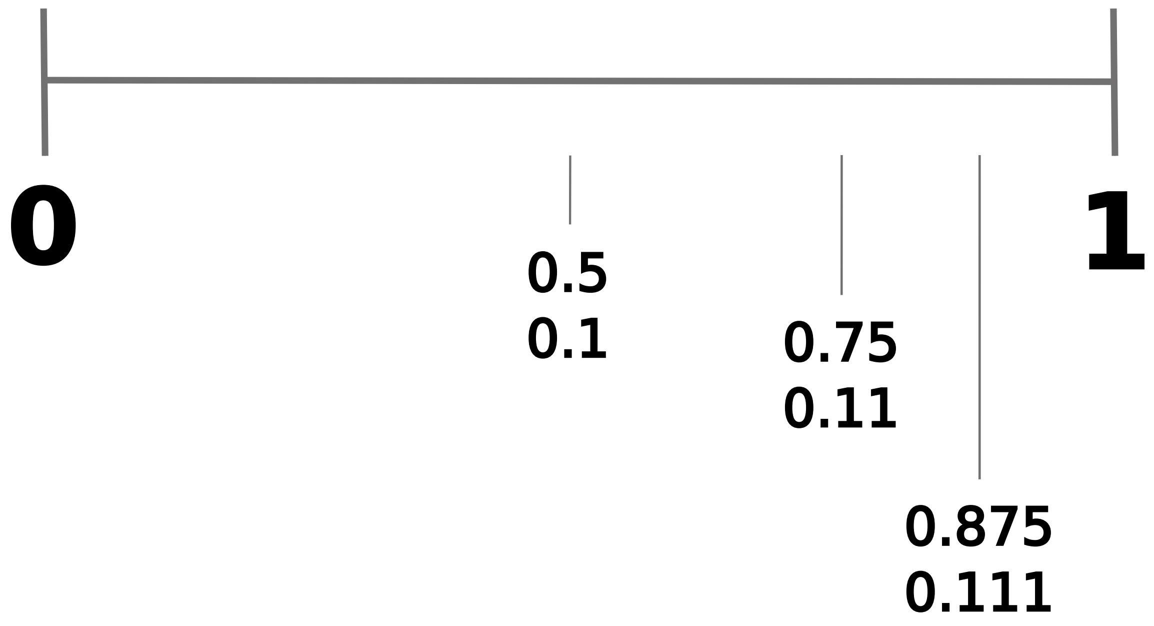


Figure 17: Key distribution with author adding paragraphs one after another.

While adding the first paragraph to the datastore we cannot know on which side of that paragraph most of the remaining paragraphs will be added. We want to have an equal amount of remaining space on both sides of the paragraph, so we decide to place the paragraph in the middle of the segment. Its index is thus 0.5. Let us say the user keeps adding paragraphs one after another. He is using the text area for writing a novel and keeps adding paragraphs to the end, one after another. The system is unaware that he is writing a novel, so it will keep placing the paragraphs in the middle of the remaining space. His paragraphs will get coordinates 0.5, 0.75, 0.875, etc. As the coordinates grow bigger using binary decimal will be a lot clearer. Binary coordinates for the same paragraphs are 0.1, 0.11, 0.111, etc. Figure 17 attempts to visualize this.

For each added paragraph, our novel author makes the key size grow one bit. This bit will be present in the dataset forever, because of the tombstones. It will also add one bit to the key length of all paragraphs the author will write after that paragraph. Figure 18 shows the growth curves for key size and space consumed, as the author keeps adding paragraphs at the end. If we knew that the author was writing a novel, we could measure how novels get written, and optimize the way we reserve space from the key space to match that model.

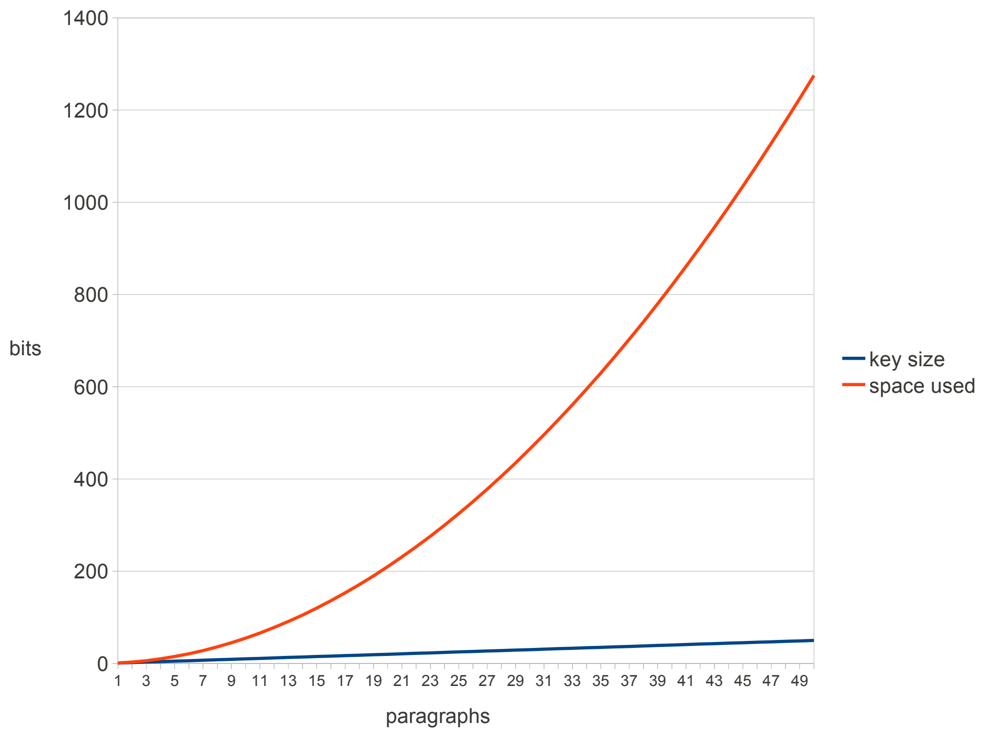


Figure 18: Growth of key size and the total amount of space consumed by keys, as paragraphs get added to the end of the text stored.

Another way to think about the problem is to think of the address space as a binary tree. When the tree grows unbalanced, you would normally do tree rotation operations. Rotations however can not be done, as removing keys leaves tombstones behind. Thus the rotations need to be done beforehand, jumping at insertion time into the node where the data would end up after all balancing operations to the final form of data had been carried out. The likely addition order for future data cannot be extrapolated from the order of past additions, as we do not store the order of additions. Studying the possible ways of optimising the key space based on the order of past additions, or whether such optimisations are possible at all, are beyond the scope of the thesis.

One solution to the problem is to return where we started from. Storing all text under one key. By doing that, we can at least be sure that two incompatible versions of the document will not get merged and the information loss will at least be evident to the user who looses the conflict. In future it would be possible to alter Peerscape to support a separate tombstone format that would store a hash of the key instead of the full key. Or we could define a time-out for the tombstones, taking the risk that the items might pop back into existence if a living copy of the key should join the network after the tombstone had timed out. For now however, we are sticking with the naive approach. We realize it has its problems, but at least they are straight forward ones we can understand.

6 Conclusion

We have explored a few data protection models from the perspective of content centric and peer-to-peer networking. Based on what we have learned we have implemented our own platform for developing content centric peer-to-peer web applications. We have evaluated this platform's suitability for application development by developing a few example applications, including a public chat room, a Bittorrent index and a flower game. From development of the example applications we have come to the conclusion that developing simple applications for the platform is possible. Being browser centric, the platform shares some challenges with general client side web programming. The decision to use the browser makes application development easier for people with former experience in web development. The constant improvement of browsers and web standards should also improve potential of browser based applications in the near future. We see no major obstacles for using the knowledge gained from the prototype in developing consumer systems of the same nature.

The work done was scoped down to peer-to-peer networking. One of the future research directions might include studying how we could allow companies to join the network. Companies could improve the system by providing persistent data storage and hosting. Using Peerscape's data model would be an improvement to the old world where companies have monopolies over user data. Working on standardization of the data models used in Peerscape datasets would allow, for example multiple image service companies to compete in making the best site for accessing family photographs. Some companies might want to compete on creating the best user interface, while some others could compete on data storage or hosting. Having competition at different levels separately should result in better options for end users.

References

BBS+09	Baden, R., Bender, A., Spring, N., Bhattacharjee, B. and Starin, D., Persona: An online social network with user-defined privacy. *SIGCOMM '09: Proceedings of the annual ACM Special Interest Group on Data Communication Conference*, Barcelona, Spain, August 2009, ACM, pages 135–146.
Ber01	Berman, A., Daliworld: From info highway to digital ocean, 2001. URL `http://www.usatoday.com/tech/2001-09-06-net-interest.htm`.
BG03	Bennett, K. and Grothoff, C., gap - practical anonymous networking. *PETS '03: Proceedings of the 3rd Workshop on Privacy Enhancing Technologies Symposium*, Dresden, Germany, March 2003, Springer-Verlag, pages 141–160, URL `http://gnunet.org/download/aff.ps`.
BGHL03	Bennett, K., Grothoff, C., Horozov, T. and Lindgren, J. T., An encoding for censorship-resistant sharing. Technical Report, 2003. URL `http://gnunet.org/download/ecrs.ps`.
BS06	Baset, S. and Schulzrinne, H., An analysis of the skype peer-to-peer internet telephony protocol. *INFOCOM '06: Proceedings of the 25th international conference on Computer Communications*, Barcelona, Catalunya, Spain, April 2006, IEEE.
CCF04	Candea, G., Cutler, J. and Fox, A., Improving availability with recursive microreboots: a soft-state system case study. *Performance Evaluation*, 56, pages 213–248.
CF03	Candea, G. and Fox, A., Crash-only software. *HotOS '03: Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, USA, May 2003, The USENIX Association, pages 67–72.
CS05	Clarke, I. and Sandberg, O., Small world talk, video. *C3 '05: Proceedings of the 22nd Chaos Communication Congress*, Berlin, Germany, December 2005, Chaos Computer Club.

EGG07	Evans, N. S., GauthierDickey, C. and Grothoff, C., Routing in the dark: Pitch black. *ACSAC '07: Proceedings of the* 23rd *annual Computer Security Applications Conference*, Los Alamitos, California, USA, December 2007, IEEE Computer Society, pages 305–314.
fac10	2010. URL `http://facebook.com/`.
FGR03	Ferreira, R. A., Grothoff, C. and Ruth, P., A transport layer abstraction for peer-to-peer networks. *CCGrid '03: Proceedings of the* 3rd *International Symposium on Cluster Computing and the Grid.* IEEE Computer Society, May 2003, pages 398–403, URL `http://gnunet.org/download/transport.ps`.
fre10	2010. URL `http://freenetproject.org/`.
fus10a	2010. URL `http://p2p-fusion.org/`.
fus10b	2010. URL `http://fuse.sourceforge.net/`.
GG08	GauthierDickey, C. and Grothoff, C., Bootstrapping of peer-to-peer networks. *SAInt '08: Proceedings of the* 8th *annual International Symposium on Applications and the Internet*, Turku, Finland, August 2008, IEEE, pages 205–208.
Gro03	Grothoff, C., An excess-based economic model for resource allocation in peer-to-peer networks. *Wirtschaftsinformatik*, 3. URL `http://www.ovmj.org/GNUnet/download/ebe.ps`.
HAKB07	Harrison, C., Amento, B., Kuznetsov, S. and Bell, R., Rethinking the progress bar. *UIST '07: Proceedings of the* 20th *annual ACM symposium on User Interface Software and Technology*, Newport, Rhode Island, USA, October 2007, ACM, pages 115–118.
Jac06	Jacobson, V., A new way to look at networking, video. Google Tech Talks, August 2006, URL `http://video.google.com/videoplay?docid=-6972678839686672840`.
JW07	Jackson, C. and Wang, H. J., Subspace: secure cross-domain communication for web mashups. *WWW '07: Proceedings of the 16th international conference on World Wide Web*, Banff, Alberta, Canada, May 2007, ACM, pages 611–620.

KYB^{+}07 Krohn, M., Yip, A., Brodsky, M., Morris, R. and Walfish, M., A world wide web without walls. *Hotnets '07: Proceedings of the 6th ACM Workshop on Hot Topics in Networking*, Atlanta, Georgia, USA, November 2007, ACM.

NEKB^{+}08 Niebert, N., El Khayat, I., Baucke, S., Keller, R., Rembarz, R. and Sachs, J., Network virtualization: A viable path towards the future internet. *Wireless Personal Communications*, 45,4(2008), pages 511–520.

ope09 2009. URL `http://openlookup.net/`.

pee09 2009. URL `http://peerscape.org/`.

sky10 2010. URL `http://skype.com/`.

Sta09 Stanik, J., A conversation with van jacobson. *Queue*, 7,1(2009), pages 8–16.

svn10 2010. URL `http://subversion.tigris.org/`.

ter07 Teredo overview. Technical Report, Microsoft TechNet, January 2007. URL `http://technet.microsoft.com/en-us/library/bb457011.aspx`.

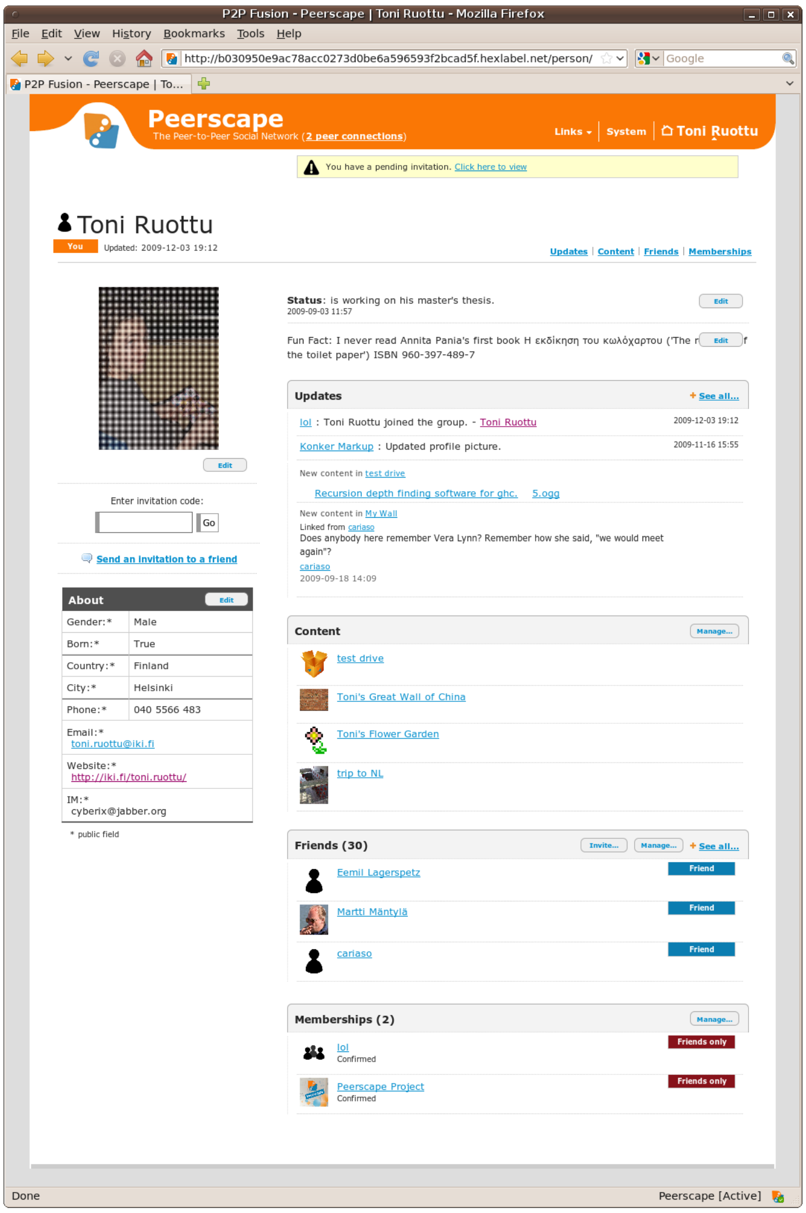
vA06a von Ahn, L., Games with a purpose. *Computer*, 39,6(2006), pages 92–94.

vA06b von Ahn, L., Human computation, video. Google Tech Talks, July 2006, URL `http://video.google.com/videoplay?docid=-8246463980976635143`.

vK10 van Kesteren, A., Cross-origin resource sharing. Technical Report, The World Wide Web Consortium, July 2010. URL `http://www.w3.org/TR/cors/`.

WS05 Walsh, K. and Sirer, E. G., Thwarting p2p pollution using object reputation. Technical Report, Cornell University, February 2005.

Appendix 1. A Peerscape Profile



P2P Fusion - Peerscape | Toni Ruottu - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://b030950e9ac78acc0273d0be6a596593f2bcad5f.hexlabel.net/person/

Google

P2P Fusion - Peerscape | To...

Peerscape
The Peer-to-Peer Social Network (2 peer connections)

Links System Toni Ruottu

You have a pending invitation. Click here to view

Toni Ruottu
You Updated: 2009-12-03 19:12

Updates | Content | Friends | Memberships

Status: is working on his master's thesis.
2009-09-03 11:57

Fun Fact: I never read Annita Pania's first book Η εκδίκηση του κωλόχαρτου ('The r Edit f the toilet paper') ISBN 960-397-489-7

Enter invitation code:
Go

Send an invitation to a friend

About Edit

Gender:*	Male
Born:*	True
Country:*	Finland
City:*	Helsinki
Phone:*	040 5566 483
Email:* toni.ruottu@iki.fi	
Website:* http://iki.fi/toni.ruottu/	
IM:* cyberix@jabber.org	

* public field

Updates + See all...

lol : Toni Ruottu joined the group. - Toni Ruottu 2009-12-03 19:12

Konker Markup : Updated profile picture. 2009-11-16 15:55

New content in test drive

Recursion depth finding software for ghc. 5.ogg

New content in My Wall
Linked from cariaso
Does anybody here remember Vera Lynn? Remember how she said, "we would meet again"?
cariaso
2009-09-18 14:09

Content Manage...

test drive

Toni's Great Wall of China

Toni's Flower Garden

trip to NL

Friends (30) Invite... Manage... + See all...

Eemil Lagerspetz Friend

Martti Mäntylä Friend

cariaso Friend

Memberships (2) Manage...

lol
Confirmed
Friends only

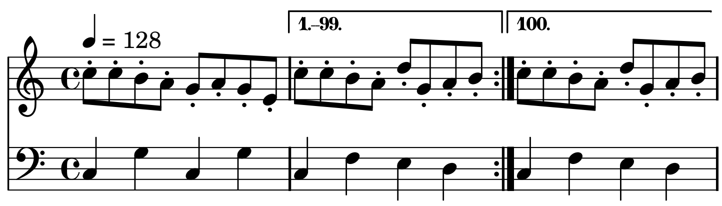
Peerscape Project
Confirmed
Friends only

Done Peerscape [Active]

Appendix 2. Songs From The Flower Garden

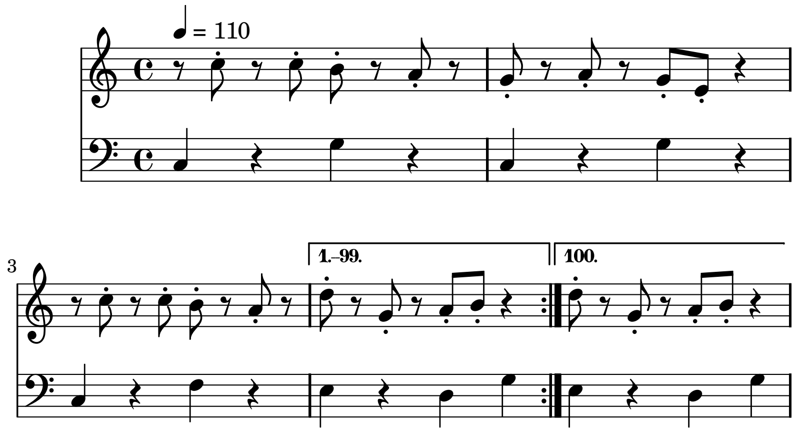
Flower Growing Days

Toni Ruottu



Flower Growing Nights

Toni Ruottu



♩ = 110
3
1.–99.
100.