Decentralized Hosting And Preservation Of Open Data

Samuel Goebert (1,2), Bettina Harriehausen-Mühlbauer (1), Christoph Wentzel (1), Steven Furnell (2) ; (1) University of Applied Sciences Darmstadt (Germany), (2) Plymouth University (UK)

Abstract

Open Data Initiatives are hosted in a centralized way despite their encouragement of sharing. Synchronizing the data in a multi master database replication setup enables user contributions from multiple sources. This paper details a proof of concept implementation of a peer-to-peer protocol that enables a decentralized archiving network which does not need a central synchronization database. The protocol provides auditable, tamperproof multi master database replication via a distributed log file. This provides a permanent and formalized way to access the data. All participants can edit their locally and changes are synchronized with the network.

Introduction

The Internet is changing from a web of documents to a web of data. Open data initiatives like Wikipedia, Internet Archive, Stack Exchange or OpenStreetMap have become important sources of global knowledge. While mainly built by volunteers the quality of the data has reached and in some cases exceeded proprietary offerings. Although open data is freely available and everybody is invited to contribute, update operations have to happen on the main database or will be ignored. The data is locked in a centralized architecture.

Current efforts to archive open data is based on mirroring offside copies of the main database [9]. This prevents the raw data binary stream from getting lost but the data in the archives is disconnected. The copies must be updated in regular intervals to reflect the state of the main database. Direct editing of a mirror is possible but creates a fork as update operations are not synchronized back to the main project. With many forks of the same data it becomes difficult to determine the leading fork that should be preserved and contributed to.

This paper discusses a novel approach to preservation and synchronization of open data which does not require a single point of truth. Every participant hosts a complete version of a decentralized transaction log. The transaction log is replicated and synchronized in a peer-to-peer network and contains the history of update operations from every node. The participants together form the main database.

This paper gives the syntax and semantics of a proof of concept implementation of a peer-to-peer protocol that enables decentralized archiving networks. Its features are:

- no single point of truth needed
- auditable and temper proof data storage
- easy to join an existing network via a meta data file
- provides a permanent infrastructure to access the data as long as one machine remains in the network
- mirrors can edit local data and changes are synchronized with the network

Concept

To preserve a website it might be sufficient to store snapshots of a page if the web page follows the document model of visualization. The approach might not work as well for data drive applications where visualization is dependent on the state of the interaction. A blog type website displays the same content for a given URL, while a search engine displays different results depending on the search terms. To preserve this type of dynamic websites, an approach would be to preserve the data behind the search engine and recreate the presentation layer by creating a website that uses the data as source. Open data initiatives are willingly giving access to the structured data behind their web application.

The structured data is the culmination of the history of changes to the data set. A snapshot of only the current state of the database might be sufficient for a mirror who wants to display the latest changes only but for preserving the data valuable information is sacrificed in the long term. A distributed network archive is a data structure with no single point of truth that focuses on storing the history of the structured data. Due to the temper proof nature of the protocol it is possible to store a copy of the history from any participant in the network. The distributed nature of the protocol makes it possible to also contribute to the history from any participant in the network moving the current data set from a single machine into a network.

The difference from a distributed archiving network to existing synchronization approaches is the data structure used to hold and distribute the history. An archiving network is synchronized using a temper proof, cryptographically linked chain of blocks. A block groups single change operations to a local database together. To form a time line, the blocks are linked by hash values. To seal a block, a cryptographic puzzle has to be solved and the solution is signed with cryptographic keys. Every node is able to verify a sealed block by validating the hash chain and the corresponding puzzles. If the sealed block satisfies the validation criteria the log is advanced, broadcasted to the other participants and used as the base for the next puzzle.

Decentralized hosting is truly democratic since there is no single point of authority that decides the validation rules for a block or update operation. The rules are decided by the majority of members, not the host of the main database. The entry barrier for contributing data is lowered as infrastructure and contributors are separated. Fluctuation of members does not have an impact on availability, accessibility of the data or the corresponding infrastructure since all machines are connected but can advance independently from each other.

By providing a standardized way of access to open data, archives are able to become a connected part of the hosting infrastructure instead of stale backups of the main database. By contributing to a decentralized data set, participants prevent knowledge islands. The global data set is improved, instead of only one

of many forks. Reviving an abandoned project becomes effortless since the infrastructure is still in place as long as one machine in the network holds the history.

An archive network can be seen as an extension to the 3-tier web architecture, acting independently from the web application. This 4th-tier is responsible for handling the communication with the network, creating blocks and validating blocks or transactions. The web application pushes data into the 4th-tier which becomes part of the network knowledge and data must be pulled from the network and imported back into the web application, to become part of the local knowledge. This approach makes it possible to decide if local data should be overwritten or not with data from the network.

The creation of new network results in two artifacts: The genesis block which is the root node of the block chain and a meta data file, with all information to distinguish the network from other networks. The genesis blocks is a special block since it does not have a precedent block and no transaction. All validation must start from this block hence the hash to identify this block is very important and should be well known in the network to make sure everybody validates from the same base. Similar to bittorrent [7] the meta data file contains information about how to identify a network. For example the root block hash, tracker links where peers might be found, a title of the network and a description of the network.

Bittorrent uses a special purposed software, so called tracker software, to collect information about machines like the IP address and a port who want to participate in a torrent. Machines that want to join a torrent ask the tracker if it has other machines or peers that are running the same torrent. If this is the case information is send back and the machine is able to contact the network. The archive network protocol piggybacks on this mechanism and thus uses an existing infrastructure to connect machines with each other. It let others find and join a network via a bittorrent tracker found in the meta data file.

When a machine receives information about other peers from the tracker, it downloads all the blocks these machines have stored. It then starts to validate if the genesis block is the same as in the meta data file. Subsequent all blocks are validated and if this succeeds all transactions in these blocks are validated. By validating all blocks from the root block them self the application can be sure that it has the same block chain as all other machines. When a machine creates a new block it distributes them to their peer which validates the block. Following a successful validation the block is passed on until all machines have the new block in their chain.

Protocol Format

This section details the primitive data types that are necessary for the protocol. While all examples use the *XML* format, other data representations like *JSON*, *bencode* or *binary* formats are possible. The choice of data format must be decided by the creator of the network. The corresponding software must simply be able to read and write the data format.

Change Set

A change set is a single transformation of the applications database wrapped into a predefined form. The format depends on the data that needs to be exchanged with other machines to recreate the database. For example, a network is created to let machines share their search index for books. If a book is added to the local database, the corresponding change set would contain the information that it was an addition and the fields that have been added to the database. For book modification the format would consist of the primary key identifying the book and the fields that have changed. A book that was deleted would simply be identified by the primary key. The primary key is generated in the form of a Universally Unique Identifier (UUID) to ensure it is unique.

Listing 1. *Change set format in XML*

```
<change-set>
 <addition>
 <book>
  <uuid>550e8400-e29b ...</uuid>
  <author>Goehte</author>
  <title>Faust</title>
  </book>
 </addition>
</change-set>
```

With these three operations we are able to write a log file that when replayed, results in the current state of the local database. The fields that a change set must have are predefined by the creator of the network. Syntax validation is applied to make sure that a change set is in the format the creator of the network intended it to be. Validation rules might include the length, the content, the uniqueness or regular expressions. The change set must obey the format. A machine that receives a change set that is not in the correct format, should drop it and the encapsulating block.

Transaction

A transaction is a container type that holds a change set. It has additional data information about who created the change set, and when as well as a hash of the content. To make a change temper proof the hash is build from the data of the change set and the date when the set was created. By rehashing the change set data and the date, it is possible for other machines to validate that the content has not been changed.

Listing 2. *Transaction format in* XML

```
<transaction>
 <hash>709813209487</hash>
 <created-at>
   2013-05-30T09:30:10Z
 </created-at>
 <change-set>
   ...
 </change-set>
</transaction>
```

Block

A block is a container type holding transactions. Additionally to the transactions, a block contains the following data:

- Hash of the previous block
- Hash for this block
- Date when mining was started
- Creation date of this block

- Difficulty for the cryptographic puzzle
- Offset (nonce) for the cryptographic puzzle
- A date when the correct nonce was found
- A hash build by the hashes of all transactions in this block in lexicographic order
- The public key of the machine

Listing 3. *Block format in* XML

```
<block>
  <previous-hash>
    0023987...
  </previous-hash>
  <hash>0002390d...</hash>
  <transactions-hash>
    3246234...
  </transactions-hash>
  <started-mining-at>
    2013-05-30T09:30:10Z
  </started-mining-at>
  <difficulty>4</difficulty>
  <nonce>439</nonce>
  <found-nonce-at>
    2013-05-30T09:45:47Z
  </found-nonce-at>
  <transactions>
  <transaction>
    ...
  </transaction>
  <transaction>
    ...
  </transaction>
  </transactions>
</block>
```

To seal a block with all transactions a cryptographic puzzle must be solved. To solve the puzzle, the hash of the block must begin with zeros. How many numbers must be zero is determined by the difficulty level. If the level is four, the first four numbers have to be zero. The values for a block a static. To generate a different input for the hashing function the nonce value is changed with random characters until the puzzle is solved.

The difficulty must not be a fixed value. A formula containing the average time between the last 10 blocks might suite better if many machines take part in the network. The value or the formula to get the difficulty is decided by the creator of the network.

The root hash of the transactions is build by hashing all hashes of the transactions in lexicographic order. This saves up valuable time while solving the puzzle since only one hash has to be taken into consideration instead of all transaction hashes for every pass.

The hash for a block is build by hashing the values of the previous hash, the root hash for the transactions, date of beginning of puzzle solving, the difficulty and the nonce together. If the hash does not solve the cryptographic puzzle, the nonce value is changed. This creates a new hash that might solve the puzzle. This is done in a loop until the puzzle is solved.

To provide information who created the block, it is digitally signed using asymmetric keys. The block is signed with a private key of the user. The public key is used to verify the signature. If a system like PGP is used, additional information about the user can be stored while creating the keys. PGP maintains a public infrastructure storing public keys. Those can be retrieved from the service and validate that the transaction could only be signed by someone who owns the private key. This makes it possible to identify the source of a transaction. As a backup method the public key should be stored with every block to prevent that a public service for the keys vanishes and hurts the auditability this way.

Block Chain

The block chain is not an element in itself but is build by the blocks and their connection with each other. The block chain is a tree structure with a single root branch, the genesis block. The number of nodes that a tree can posses is not fixed. Blocks can come from all machines in a network and the tree structure accommodates for this by supporting branches.

To seal a new block, the machine takes the previous block with the greatest depth in the tree as a basis for the new block. If the final node has two blocks the application should start with the older block. When a new block is received from the network and the depth is greater than the current one the application is working on, it should stop and take the new block as the predecessor. Orphaned blocks, blocks received from the network which have no previous block that is in the chain yet, are stored separately as the missing block might arrive later.

The block chain can be validated by following the hashes and signatures in the chain. This might take some time depending on the number of blocks.

To accommodate for orphaned or split branches only the mainline branch should be applied to the application database. The mainline branch is the chain of blocks with the highest depth in the tree minus a predefined number of blocks from the top. As it is not clear when a new block arrives and if it belongs to the mainline, transaction should only be applied to the database if there are blocks in front of them. The buffer of blocks makes sure that no other branch overtakes the lead and changes have to be reverted on the database. Transactions that reside in a block that has been determined as orphaned have to be reschedule for integration into a new block.

Listing 4. *Block chain format in* XML

```
<blocks>
  <block>
        ...
  </block>
  <block>
        ...
  </block>
</blocks>
```

Meta Data File

The meta data file is the key to finding peers in a network. Based on the idea of the torrent file from bittorrent, the meta data file contains all information that is necessary to join a network. It is also syntax compatible with the bittorrent meta file, to reuse the existing infrastructure. The file consists of the following fields and is encoded in the bencode format:

- tracker links
- the info_hash for the network
- the hash for the genesis block
- a name for the network

With this file it is possible to initialize the network application and connect with other peers in the network.

The hash for the network is the key to identify the network. It is compatible to the info_hash value from bittorrent which allows us to use the existing infrastructure in form of bittorrent trackers to bootstrap a new machine with peer information.

The hash for the network is build by hashing the values of the the genesis block hash and the name of the network. Since trackers are subject to change during the lifetime of a network, the remaining fields do not. Also the validation rules are subject to change and should not be tight to the meta data file.

Network

This section details the network part of the protocol. How finds a machine other machines and how should it behave to achieve the storage behavior.

Web Application Extension

The application that deals with the archive network, can be seen as an extension to the existing web architecture. By using an update mechanism like, e.g., a trigger in a database, a transaction is build according to the syntax rules of the network. The transaction is then distributed among the other machines. These machines apply the change to their local databases when the block is valid and in the mainline. It is possible to convert initiatives that do now want to support a network archive directly by using other data representations of the history as input. A mechanism that converts the RSS feed into blocks for example could be used to fill the network as long as all relevant data is in the RSS feed. There is no full synchronization between all machines here since RSS is a read only format.

Discovery

A network is distributed by nature. The block chain can be used on a single machine but the true value of exchanging the blocks arises when other machines join a network. When a machine wants to join a new network, it asks a tracker from the meta data file if it posseses information about other peers. The network is identified on the tracker via the hash of the network. The tracker keeps records of which IP address asked for information about the network and relays any other machine that has asked before.

The network application then tries to contact the machines returned from the tracker. If one or more machines are found the block chain is reconstructed by downloading all blocks that exist in the network and validate them. When all blocks are validated the node is a full member of the network.

Tracker

A tracker is HTTP web server that connects peers interested in a special network. The purpose and syntax is identical to the bittorrent specification. Via a HTTP GET request the client announces that it is interested in finding peers from the identifying network. The trackers saves information about the client like

- IP address
- Port
- How many blocks have been downloaded so far
- Info_Hash
- Event

The tracker response is a bencode dictionary.

To have an up-to-date representation who is a member of which network, the client should advertise itself every 30 minutes and when a new block is deemed valid. This makes sure that a new machine finds fresh peers quickly in new network.

Network Behavior

To form the peer-to-peer network that underlies the archiving network, all machines have to react on events. Since a client might leave a network at any time, important information like other peers and the block chain is spread to all machines as soon as they join a network.

node up event

The node up event is a basic building element. When this routine is called the machine should announce itself to all trackers that are in the meta data file or have been collected by other peers. This makes sure the trackers have up-to-date information about all peers in the network.

Other peers should be asked if they might have new blocks that was missed. This is done by asking for a short list of all blocks a client posseses. This block list is checked against the local block chain and missing blocks are requested as necessary from the peers.

This routine should be executed every time the machine experiences network downtime. This ensures that the machine has the latest block chain elements.

node up timer event

This is a regular interval event at which the node up routine should be called. Depending on the number of updates in a network, this routine might be called from every hour to once a day.

tracker timer event

This routine should be called every 30 minutes to announce to the tracker that this machine is still alive.

boot up event

When the application is started, it checks the local block chain and all transactions for validity and thus uses second approach to recovery from a crash detailed by A. P. Sistla et al. [10]. Then the node up event should be run to ensure that no blocks have been missed since the last boot.

receive block event

When a block is received the block itself and all transactions in it are validated. When a block is valid, it is rebroadcasted to all known peers. This kind of flooding algorithm makes sure that every machine in the network receives a newly found block as soon as possible. If something is wrong with the block, it is dropped and not rebroadcasted to avoid further distribution of the block.

asked for block event

The client simply returns the block it was asked for identified by the hash.

asked for block chain event

The client returns the short list of all blocks the client posseses.

Limitations

The protocol using the block chain algorithm at its heart has some limitations that we want to discuss in this chapter.

Real-Time-Updates

Due to the asynchronous nature of allowing contibutions from every machine, the block chain is not a real time data structure. It takes time for a block to travel throughout the network. The data structure is eventual consistent, meaning that at some point in time the whole network converges with the same state of the database. The time it takes to achieve this point is determined by many factors including the network latency, number of machines in the archive network and available bandwidth.

For data that is expected to change very infrequently the limitation of slow updates is not a problem. Most static data does not have the requirements to be available at another machine for quite some time. Since an archiving network is build for sharing longterm data, it might be sufficient when the block chain is updated only once a day, to pack as many transactions in a block as possible.

This makes it possible to push the CPU consuming task of solving the puzzle in a time slot where the application would be idle otherwise, like midnight.

Truncation Attack

As described in [13], the block chain as a log file, is vulnerable to a truncation or tail cut attack. In this attack the last n blocks are omitted when a client asks for the complete block chain.

Since a malicious attacker would need to control all clients surrounding a victim, the attack has only a small possibility of success in a large network. As long as one clients relays all blocks the attack would be mitigated.

The Sybil Attack

Like all distributed systems the network archives are subject to the Sybil attack described by John R. Douceur [6]. In a system with an open account creation mechanism like generating PGP keys nothing prevents a user to generate a new pair of keys for every transaction and thus gain new identities with every key pair. Depending on the mode the application is running this might become a problem when contributions from all machines are allowed. A malicious user could not get banned by trying to identify him through their key, since he could change the keys for every block.

Block flooding

A malicious user is able to generate fake blocks [11] and occupy CPU time on a machine, since all incoming blocks need to be validated. To prevent this attack, misbehavior of peers regarding bandwidth consumption could be taken into account which might result in dropping the peer.

Related work

The block chain algorithm in combination with a cryptographic puzzle was detailed in [12] to provide a distributed ledger for currency transactions. The block chain variant used in this paper is a simplified version that drops the mini programming language that allows complicated transactions in the block chain.

The bootstrap mechanisms from bittorrent are reused. There is an already working infrastructure with these tools that we can repurpose to bootstrap a network. A meta data file as configuration initialization file for a network can easily be send via email or placed on regular bittorent index sites as the file format is compatible with each other.

Distributing the history of a web application was also researched by several other teams [1, 2, 3, 4, 5]. Their main focus was to distribute a wiki. Since the type of application was known before hand, it was possible for them to include fine grained conflict resolution mechanisms. A network archive simply applies all transactions from a mainline block, regardless of the semantical meaning a change might have for an application. Since the block chain is the agreed time line of events, conflicts are overwritten by the latest change set in the block chain.

Conclusion

In this paper we detailed a peer-to-peer protocol that enables decentralized archiving networks without the need for a central synchronization database. Every participant hosts a complete version of a decentralized transaction log. The transaction log is replicated and synchronized in a peer-to-peer network and contains the history of update operations from every node. The participants together form the main database.

The difference from a distributed archiving network to existing synchronization approaches is the data structure used to hold and distribute the history. An archiving network is synchronized using a temper proof, cryptographically linked chain of blocks. A block groups single change operations to a local database together. To form a time line, the blocks are linked by hash values. To seal a block, a cryptographic puzzle has to be solved and the solution is signed with cryptographic keys.

To further enhance the resilience of archive networks it could follow the footsteps of bittorrent and decentralize the bootstrapping of peers by using a distributed hash table (DHT) implementation for peer finding like *kademila* [8]. *kademila* is a peer-to-peer system to find peers that are interested in the same *info_hash*. This would eliminate the need for trackers in the meta data file.

To eliminate the need for having the meta data file to join a network an instead rely on a link, magnet link can be used. A magnet link points again to a DHT like peer-to-peer network that provides the corresponding meta data file. The file is identified by the *info_hash* and the meta data file itself is downloaded first from the *kademila* network and then the machine is also able to join the archive network. Only the magnet link is needed to join a network instead of the meta data file itself.

Multi master database replication with a distributed log file, does not ease the tasks of preservation in itself but makes the data much more accessible than a download. The access to the data becomes structured and formalized and the infrastructure is opened to the public. Everyone is able to download a copy of the data and actively contribute storage and bandwidth to keep the initiative alive.

References

[1] Sebastian Schaffert, IkeWiki: A Semantic Wiki for Collaborative Knowledge Management, In 1st International Workshop on Semantic Technologies in Collaborative Applications (STICA'06), 2006

[2] Guido Urdaneta et all., A Decentralized Wiki Engine For Collaborative Wikipedia Hosting, (2012).

[3] Weiss, S. and Urso, P. and Molli, P., Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks, Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on, pg. 404 -412 (1998).

[4] Charbel Rahhal and Hala Skaf-molli and Pascal Molli, SWOOKI: A Peer-to-peer Semantic Wiki, UbiMob '09 Proceedings of the 5th French-Speaking Conference on Mobility and Ubiquity Computing, pg. 91-94 (2009).

[5] Gerald Oster, Pascal Molli, Sergiu Dumitriu, Rubn Mondjar, UniWiki: A Reliable and Scalable Peer-to-Peer System for Distributing Wiki Applications, COO - INRIA Lorraine - LORIA , Department of Computer Science and Mathematics - URV, pg. 18. (2009).

[6] John R. Douceur, The Sybil Attack, IPTPS '01 Revised Papers from the First International Workshop on Peer-to-Peer Systems, pg. 251-260 (2001).

[7] Bram Cohen, Incentives Build Robustness in BitTorrent, (2003).

[8] Petar Maymounkov , David Mazires, Kademlia: A Peer-to-peer Information System Based on the XOR Metric (2002).

[9] Bernhard Haslhofer et al., ResourceSync Framework Specification - Alpha Draft (2012).

[10] Sistla, A. P. and Welch, J. L., Efficient distributed recovery using message logging, pg. 223-238 (2012).

[11] Byung-Gon Chun, Mechanisms to Tolerate Misbehavior in Replicated Systems, Technical Report No. UCBEECS-2007-103 (2007).

[12] Satoshi Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System, (2009).

[13] Ma, Di and Tsudik, Gene, A new approach to secure logging, Trans. Storage (2009).

Author Biography

Samuel Goebert is a computer science Ph.D. student at the Plymouth University (UK) and the University of Applied Sciences Darmstadt, Germany. Goebert has over ten years of experience in software development and associated technologies. He currently researches peer-to-peer methodologies to improve preservation of digital cultural heritage.