An incentive-compatible decentralized file storage

Final Project in Decentralized Systems Engineering

Carolin Beer, SCIPER Nr. 294852

Introduction

Our Peerster protocol allows us to exchange messages and files in a decentralized network. While we have the functionality of uploading files, the topic of *privacy* has not been a big concern in this implementation since all contents in the network are accessible to everyone. With the presented final project, Peerster is extended to allow for the upload of private files while preserving incentive compatibility for nodes in the network and tolerating temporary node failures without affecting file availability. Furthermore, the private messaging functionality is extended by content encryption and digital signatures to prevent eavesdropping.

Related work

Cryptography Providing privacy for the contents of files and messages can be achieved utilizing cryptography. In asymmetric cryptography, peers generate a key-pair (pub, priv) of which only the public key is made available to other peers. Data that is addressed to a peer can then be encrypted using this node's public key. Once encrypted, the data can only be decrypted by the peer possessing the corresponding private key. One asymmetric key encryption algorithm is Rivest–Shamir–Adleman (RSA). On the other hand, in symmetric cryptography, a single key is used to both en- and decrypt data. Data can thereby be shared among peers by distributing the utilized key, e.g. through an asymmetrically encrypted message, or, in bilateral settings, using Diffie-Hellman key exchange to generate a key up-front. One method of symmetric cryptography are algorithms using block ciphers such as the Advanced Encryption Standard (AES) (Smart, 2002).

Availability Availability of data in p2p networks where the nodes themselves cannot be assumed to be reliable may be achieved through *replications* or *erasure coding* (Weatherspoon and Kubiatowicz, 2002). While the approach of replications distributes copies of files or chunks of files among nodes, erasure coding introduces redundancy on the data level and breaks up the data into fragments that are stored across different nodes.

Incentive compatibility in p2p cloud storage When storing Public Files, other peers may have some inherent advantage from storing a file published by another peer, such as having constant access to the data. However, when files are encrypted and made private, this incentive fades.
Feldman and Chuang (2005) examines different incentive mechanisms and identifies *Inherent Generosity*, *Monetary Payment Systems* and *Reciprocity-Based Schemes* as main concepts.
A concept that relies on inherent generosity is the p2p cloud storage system Freenet (Clarke et al., 2001). However, this doesn't seem to provide sufficient incentive compatibility as the system suffers from lacking storage capacity (Kopp et al., 2017).
Monetary incentives are used in a multitude of different systems. Many of them issue tokens that can be traded on exchanges (e.g. Vorick and Champine (2014), Wilkinson et al. (2014) and Lambert et al.

(2015)). Thereby, differences in supply and demand of storage space may result in price changes of the coin and, in turn, changes of supply and demand resulting in a new equilibrium.
Thirdly, in *Reciprocity-Based Schemes* transaction histories are incorporated into decision processes. Schemes like tit-for-that such as BitTorrent (c.f. Cohen (2003)) or classic reputation systems are attributed to this class of incentive mechanisms (Feldman and Chuang, 2005). One approach to act based upon transaction history is for example *stranger discrimination* which discriminates newly joined nodes to avoid Sybil attacks (Lai et al., 2003).

Proving data posession A basic way of ensuring replication if the so-called Proof-of-Retrievability (PoR). The core idea is to let peers compute values based on the data they are supposed to store. By using a hash instead of downloading the entire file, communication overhead is reduced. To further ensure that a peer cannot not precompute hashes, the peer may only know the precise task he has to solve at the time he is asked to provide the proof.
To make collusion among peers for the computation of PoRs infeasible, Protocol Labs (2017) propose to send out pseudo-random permutations of data, whereas Vorick and Champine (2014) sends out data that was encrypted using different symmetric keys. Protocol Labs (2017) additionally introduce a Proof-of-Spacetime, which is an iterative PoR that takes the previous proof as input of a subsequent proof and thereby aims at proving the continued storage of a file while reducing communication overhead.
Kopp et al. (2017) uses publicly verifiable PoRs based on *Homomorphic Linear Authenticators* as introduced by Ateniese et al. (2009). An alternative to this are *Zero Knowledge Proofs* (c.f. Lambert et al. (2015)).
Combining this with blockchain-based *Smart Contracts* allows to automate the monitoring and verification of PoRs (c.f. Vorick and Champine (2014)).

System Goals & Functionalities & Architecture

Goals The goal of this final project implementation is to enable Private File storage through Peerster. As storing private files of others does not inherently benefit any node that cannot access its contents, other measures have to be taken to ensure the distribution of the file across the network. This way, availability of a file can be preserved in case of temporary node failures.
As payment systems additionally complicate the implementation and, if traded on exchanges, are often subject to high volatility, the presented approach aims at providing incentive compatibility without introducing tokens or coins.

Private file upload To describe private files, we introduce a new struct `PrivateFile`. Private files are similar to the `File` struct that we have previously used. However, the result of the file indexing process will not be published for `PrivateFiles`, i.e. it is not included in the blockchain and not returned as result of search requests. Besides the `File` struct that we used in previous homeworks, `Replica` are used to represent the storage of the file at another peer. Each `Replica` instance represents an AES encrypted version of the original file, each using a different `EncryptionKey`. Right after the indexing of the file, the variables `NodeID` and `ExchangeMFH` are empty as the replications are only stored locally at this point.

Distribute replications To distribute the file among the network, a process resembling a three-way handshake using `FileExchangeRequests` is performed for each replica:

1. The node broadcasts a `FileExchangeRequest` with the status `OFFER` to the network. Further

```go
type PrivateFile struct {
        File
        Replications []Replica
}

type Replica struct {
        NodeID          string // Name of the remote peer storing the file
        EncryptionKey   []byte // AES Encryption key
        ExchangeMFH     string // Exchange Metafilehash
        Metafilehash    string // Metafilehash of the encrypted file
}
```

Listing 1: Structs modeling private files and their replications

```go
type FileExchangeRequest struct {
        Origin               string
        Destination          string // Empty for OFFER, i.e. the initial broadcast
        Status               string // OFFER, ACCEPT, FIX
        HopLimit             uint32
        MetaFileHash         string // hex-representation of the Metafilehash of the Replica
        ExchangeMetaFileHash string // Empty for OFFER, i.e. the initial broadcast
}
```

Listing 2: Struct used to initiate the exchange of file replicas

variables contain information about the `MetaFileHash` of the `Replica` and the name of the node sending the request.

2. Interested peers respond to that message with another `FileExchangeMessage` including the metafilehash of the file they would be interested to send in exchange as variable `ExchangeMetaFileHash`. This message has a status of `ACCEPT`. The peer then starts a timeout counter to avoid waiting infinitely long for a response.

3. The initiating node checks on every incoming `FileExchangeRequest` whether the replica they want to distribute, as specified in `MetaFileHash`, was already assigned to some other peer or whether the peer is already storing another `Replica` instance of the same `PrivateFile`. If this is not the case, it sends an acknowledgement `FileExchangeMessage` with status `FIX` and starts downloading the file specified by `ExchangeMetaFileHash` from the peer.

4. On reception of the `FileExchangeMessage` with status `FIX`, the peer, in turn, starts to download the file specified by `MetaFileHash`.

At this point, the file exchange has been established and the variables `NodeID` and `ExchangeMFH` of the exchanged `Replica` are assigned to the corresponding peers. Both nodes then periodically request a *Proof of Retrievability*.

Proof of Retrievability For the implemented PoR, a peer needs to compute a hash based on the data he has stored. To avoid precomputation, we use the `Challenge` struct that specifies the precise

```go
type Challenge struct {
        Origin       string
        Destination  string
        MetaFileHash string
        ChunkHash    string // Targeted chunk
        Postpend     []byte // Data to be used for PoR
        Solution     []byte // Empty for request
        HopLimit     uint32
}
```

Listing 3: Struct used for Proofs of Retrievability

parameters that peer should use for the proof. The PoR that the peer is supposed to solve is a SHA1 hash of chunk data (the specific chunk is defined by its hash `ChunkHash`) to which some data provided in `Postpend` is appended. Both the specified chunk and the postpend data is selected (pseudo)-randomly by the requesting node. The value that the peer computes will then be stored in `Solution` and returned to the requesting node.
The PoR request, i.e. the challenge is sent out periodically. If a peer fails to provide a correct PoR for a defined number of times, the node terminates the file exchange and thus, drops the file that it downloaded in exchange to free up the occupied storage space. To decrease communication overhead, the intervals in which the PoR are performed in increasing by a (pseudo-)randomized factor after every successful PoR, and reset to the initial value if the PoR is wrong or left unanswered.
Since every `Replica` is encrypted using a different key, peers cannot collude to solve challenges.
Retrieving a file and state management The state of private files stored by a node can be exported by the client and uploaded to any other node. This state contains all the required data to fetch one of the replications and therefore allows clients to retrieve the file through other nodes in case the initial uploading node is temporarily unavailable. A node can download the remote file by sending a standardized `DataRequest`, followed by an AES decryption with the `EncryptionKey` corresponding to the `Replica`.

Private Encrypted Messaging In addition to private, replicated files, the final project introduces asymmetric encryption to disguise the content of private messages — specifically, the RSA algorithm is used. For this, the nodeID that was previously used is replaced by a public key that is generated during the bootstrapping process of a node. Private messages can then be encrypted using this public key and decrypted by the receiving peer using its private key.
However, third party nodes may still be eavesdropping on messages by altering the `Origin` when routing them. Therefore, a field `Signature` is added to the struct `PrivateFile`. It contains a digital signature which can be computed using the private key and the message as input. Importantly, the signature must computed on the unencrypted message content to prevent malicious nodes from computing a new valid signature after changing the `Origin` field to their own public key.
Through these encrypted private messages, the previously implemented private files can be shared by sending a private file state to another client. Through the combination of asymmetric encryption and digital signatures this sensitive information is secured against access by other nodes.

References

Ateniese, G., Kamara, S., and Katz, J. (2009). Proofs of Storage from Homomorphic Identification Protocols. In Matsui, M., editor, *Advances in Cryptology – ASIACRYPT 2009*, pages 319–333, Berlin, Heidelberg. Springer Berlin Heidelberg.

Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W. (2001). Freenet: A Distributed Anonymous Information Storage and Retrieval System. In Federrath, H., editor, *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*, pages 46–66. Springer Berlin Heidelberg, Berlin, Heidelberg.

Cohen, B. (2003). Incentives build robustness in BitTorrent.

Feldman, M. and Chuang, J. (2005). Overcoming free-riding behavior in peer-to-peer systems. *ACM SIGecom Exchanges*, 5(4):41–50.

Kopp, H., Modinger, D., Hauck, F., Kargl, F., and Bosch, C. (2017). Design of a Privacy-Preserving Decentralized File Storage with Financial Incentives. In *2nd IEEE European Symposium on Security and Privacy Workshops*, pages 14–22, Los Alamitos, California and Washington and Tokyo. Conference Publishing Services, IEEE Computer Society.

Lai, K., Feldman, M., Stoica, I., and Chuang, J. (2003). Incentives for Cooperation in Peer-to-Peer Networks.

Lambert, N., Ma, Q., and Irvine, D. (2015). Safecoin: The Decentralized Network Token.

Protocol Labs (2017). Filecoin: A Decentralized Storage Network.

Smart, N. P. (2002). *Cryptography: An introduction / Nigel Smart*. McGraw-Hill, London.

Vorick, D. and Champine, L. (2014). Sia: Simple decentralized storage. *Retrieved May*, 8:2018.

Weatherspoon, H. and Kubiatowicz, J. D. (2002). Erasure Coding Vs. Replication: A Quantitative Comparison. In Druschel, P., Kaashoek, F., and Rowstron, A., editors, *Peer-to-Peer Systems*, pages 328–337, Berlin, Heidelberg. Springer Berlin Heidelberg.

Wilkinson, S., Boshevski, T., Brandoff, J., and Buterin, V. (2014). Storj a peer-to-peer cloud storage network.