Department of Software Engineering, Faculty of Mathematics and Physics, Charles University Prague

Doctoral Thesis

The Eternity Service

November 13, 1998

Tonda Beneš

1 Why the Eternity Service?

One of the main streams that could be identified in today's information technology is a massive movement towards more simple and quicker access to informations. Using such techniques as a World wide web, ftp, bbs, etc. are, each piece of information gains a global world wide character. In other words, any specific information could be accessed from almost any place over the world. Many people all over the world can share the information immediately after a creation of it.

This movement corresponds with more general trends in the development of the whole our civilization. The process is especially fast in the more developed part of the world. We assume the integration will lead into the creation of a large multinational units and later into origin of one universal society. The information technology plays a crucial role in this process.

We will not start any discussion, whether the current development of the society and, in turn, whether the information technology is positive or not. Our goal now is to point out some other consequences, that are not discussed usually.

The problem laying in a background of the global character of the information consists in fact, that the physical representation of the information stays to be local. When an arbitrary information is processed, very often only a temporary local copy is created and the process is performed on it. After the processing the local copy is discarded typically. Anyway, the local copy is generally maintained in such a manner, so that there could be no warranty of its integrity any further.

Such arrangement is vulnerable to errors very much. If the primary copy is destroyed, the information is lost, or inaccessible for some time at least. The situation is even worse in the case of an intentional attack. It is not too difficult to track the path of information transfer back to the host that provided it. Today's techniques of information handling incorporates almost no protection against an active attacker attempting to analyse information flow. Then it is relatively simple to destroy the information "as sich", because the information itself is too tightly coupled with its physical representation.

Maybe it sounds paradoxical, but modern techniques of an information handling have in fact a negative impact on the information ability to survive accidents and attacks.

The only solution today available is an appropriate backup of the data. In the case, the primary copy of the data gets lost, the maintainer will try to recover the information using a backup copy. Of course the backup copies must be handled in such a way so that the risk of simultaneous loose of both (or more generally all) copies is decreased as much as possible. It is not a very good solution though, because the backup is meant as protection against hardware failure mainly. It can serve as a protection against theft and malicious employees too, but it is almost all.

Creating of backup copies has two or three main disadvantages. The first, it is off-line very often. It means, that you have to prepare backup copy, store it on a special device and then the device itself deposit somewhere at a safe place. The more distant is the depository, the better. In the case of data loose you must find the backup copy, bring it from the depository and then try to recover necessary information, which process is likely to be time consuming.

Furthermore, everyone who wishes to perform backup correctly has to acquire relatively complex and expensive accessory despite of absolute size of its system. In the case of large systems the need of trained staff even more complicates the situation.

The third big disadvantage of the backup is even more serious. It is vulnerable against intentional attacks against information itself. The court decision or religious leader order will almost surely influence all copies of the data. We must take into account the possibility of extortion of the information owner, or bribery of data maintainer. The traditional backup copies are easy target for professional opponents, such as special services or organized crime gangs are. Even the skilled insider could cause very extensive damage.

In any case destroying of a primary copy of data leads to inaccessibility of the corresponding information for everyone, although in average temporarily only.

The other problem could be named "memory". We would like to remember George Orwell's 1981, or Animals farm. In the era of electronic documents it is very simple to make the history "better". Many countries had experienced such phenomenon.

On the other hand, there is the Internet. The network interconnecting thousands of computers in many countries all over the world. The network, that was originally designed to withstand a heavy damage during a war. The network, on which becomes to be dependent the international business and which starts to play important role in the global information flow. And mainly, the network that servers are conducted by a great number of mutually independent maintainers of different religion, living in different regions. Due to the size of it, it seems to be rather complicated unless even impossible to eavesdrop all communication over the whole Internet, or some substantial part of it, provided the substantial part of the communication will be protected by some appropriate means (encryption for example). Although it is not a case of today's reality, the trends towards better protection of communication within the Internet are becoming obvious and this problem surely belongs among the questions to which is payed a great attention currently.

On the other hand it should be taken into account, that the unprotected communication is object of eavesdropping and analysis done by unauthorized parties. The project ECHELON or planned project SORM are good examples that such threat is a cruel reality rather than a real possibility. A good protection of the information within the whole life-cycle and especially during communication becomes a necessity.

It is clear that to assure a better resistance of a information against a damage or destroying, we have to store it in more than one copies. But the redundance itself is not enough. We have to spread copies as much as possible to increase the probability that at least one will survive. Almost no technical mean can stop a determined and skilled attacker. The only way seems to be a lack of knowledge about the location of the copies of original data. The system must protect not only information itself, but the conductors of involved computers too and before all, the owner of the information.

We also have to assure access to any stored in the system information. An existing, but unaccessible information is useless. Thus we have to make impossible, or a very complicated at least, for attacker to prevent data users and owners from access to their informations.

When an appropriate cryptography will be cleverly used, we can obtain a completely new system of information handling, that will be able to protect the stored information against all mentioned threats. We hope our system can bring completely new measure of the reliability and the data availability.

The basic ideas of the system has described Ross Anderson in the article *The Eternity service* [1]. From this time a dozen of different systems has been created that uses the name "The Eternity service". Most of it are some sort of distributed text storage and retrieval system generally using almost no cryptography. On contrary to it, we propose a system, that strengthens security features of the original Anderson's design.

The goal of our work is to contribute to the solution of the all mentioned problems.

2 What is it the Eternity Service?

It this section we will define some important terms. The purpose of this is as follows. Besides unusual terms related closely to the problems of The Eternity Service (we will use a shorter abbreviation "*Service*' instead in further text) many of the terms defined here are some more generally used data manipulation functions. These functions will be used in further parts without discussion of their internal structure. We will only suppose that some function with required features is available and could be used. Such arrangement allow us to design the whole system independently on any specific cryptographic or similar algorithm. It helps to replace obsoleted algorithms with new ones with the same required features in future without the need of redesigning our proposal of the system (see 5.2.8). In this section we will state required features for the functions used as basic building blocks. An example of every generic function discussed here is in $\boxed{\text{Appendix: samples of used cryptographic functions.}}$.

2.1 *Service* related terminology

Any system consists of logical parts. Each addressable part, i.e. the one that could act as a source or a target of some communication, we will name a *principal*. A principal could be anything within the system - a server, a process or a human doing his job for example. Some of principals are passive, i.e. they does not produce any actions on its own decision. More precisely objects perform only the actions that are direct consequence of some previous actions. As an example of a typical passive principal could serve a data file. Passive principals are sometimes called *objects*. On the other hand a human or a process are typical active principals, often called *subjects*. Subjects are characterized by doing actions on objects, other subjects, and itself.

Subjects, representing humans requesting from a system some service are called *users*. In fact the human accesses system by means of a intermediate process that accepts users actions and translates them into requests to a system. In reverse direction this process receives responses from a system and translates them to the human acceptable form. Just for sake of simplicity we will omit this arrangement everywhere where it will not harm precision of our work. We will not go into details instead and will assume that a user could access a system directly somehow.

A traditional system usually examines identity of a user before any further actions will take place. So when speaking about a user, we normally bear some specific principal with certain identity in mind. On the other hand, the *Service* does not examine identity of subjects accessing it. That is why we will use denotation *client* for them instead of a usually used denotation 'user'.

Maybe you have noticed in the first part of this work, that we are using the two very similar terms: the *information* and the *data*. The term *information* we use in the sence of an *idea* or a *knowledge*. Any information is an abstract fortune and we could only check the existence and availability of it.

A *data*, on the other hand, is a real thing and means some physical representation of the information. We can say, that some arbitrary information *exists* if and only if there is at least one piece of data (datum), that represents this information. We say, that the information *effectively exists (within some system)*, if and only if at least one datum that represent it still exists (within the system) and is available at least to one authorized subject (see further). [1] For sake of simplicity we will skip the clause 'within the system' unless it will cause a loose of a precision of the text. In a computing system such as that we are going to describe in this work the data occurs in the form of a file, or a message mainly.

The information *is not available* whenever it does not exist or is not accessible, i.e. the knowledge where some data representing the information are stored is lost, or the data cannot be obtained due some communication problems or any other reason. Clearly when it is possible within some reasonable amount of time to obtain at least one piece of a data representing the specified information, the information is *available*.

The availability of an information should be observed in respect to the subject requesting it. The subject have to be *authorized* to obtain the information, it means that the subject must possess the right to get the information. The fact whether subject is authorized or not is derived from global security strategy over the system.

In the case the subject is not authorized no availability problems are involved (in respect to the subject). In such a case, on the contrary, the system must assure a *secrecy* to protect information from unauthorized use. In other words, the system protects secrecy of stored information if it prevents unauthorized principals from accessing the information. The questions of secrecy are widely discussed in contemporary literature involving security, so we will not discuss it any more.

An information we take as an idea. So the informa-

[1] In the case the data representing the information is stored in such a way that it can not be understand directly — for example using an encryption or a secret splitting scheme — we want not only the data itself, but also any information needed to transform data into readable form to be available.

tion itself can not be changed. It could be replaced with some other one only. A replacement could potentially (but not necessarily) take place when the data representing the information are changed. The task of any information handling system is to preclude any unauthorized information replacement. However, it is not possible very often, mainly due the fact that some phases of the whole information processing are carried out in environment that is not under our control fully. In such cases we have to arrange sort of 'after act' control mechanisms that enables us to detect unauthorized changes at least.

The fact, that the information does not differ from the original form we usually name the *integrity* of the information. The integrity of an information is usually injured if the data representing the information is changed. We stated 'usually' because the same idea could be expressed in many ways. So there are many instances of the data representing the same information. But especially in the case of intentional unauthorized change of the data the replacement of information is highly probable.

If the system is not able to distinguish between original unchanged information and bogus variant introduced by an attacker, the overall value of all information in the system will be radically decreased.

The ability of a system to detect unauthorized replacement of the information is crucial for *reliability* of the information. The reliability is measured by amount of the trust we can place to the fact that obtained from the system information does not differ from the original form. In a broader sense the reliability of an information concerns too the other claims related to the information, for example the origin of the information, time-stamps etc. The reliability of an information is proportional to its usefulness. Unreliable information tends to be useless, so the reliability is very important feature of any information.

A special term should be mentioned in conjunction with the ability of a system to protect data against attacks. It is *resiliency.* The resiliency could be seen as a probability that information withstand attack without any harm. The more resilient information means the higher probability that information will not be damaged.

2.2 Used techniques and notation

Now let us spend a few words about a technology. In further parts we will use widely some cryptographic techniques, so here is a good place to discuss them briefly. We will show a way of record at the same time.

Let us start with some basic symbols used to describe data manipulations. When the message contains more different parts, the individual parts are separated by the ";" sign. The sign "$|$" is used for concatenation. The difference between these operators is in the fact that the second one does determine the exact order how the message should be assembled from individual parts. When the message is created up from several subparts with the same internal structure, we use the repetition sing "$\bigvee()$". For example $\bigvee_{i=1}^{n}(a_i)$ is exactly the same as $\underbrace{a;a;\ldots;a}_{n\text{-times}}$.
When it is neccessary to split the record of a data structure into two or more lines the sign "$\wr$" is used both at the end of previous line and at the begining of the following.

Principals are denominated with uppercase letters mainly from the beginning of the alphabet, for example $\mathcal{A}$, $\mathcal{B}$, etc. A principal usually communicates with other principals; "$\mathcal{A} \longrightarrow \mathcal{B}$: M" means that principal $\mathcal{A}$ sends a message M to principal $\mathcal{B}$.

The basic cryptographic operation is an *encryption* $\{M\}_{\mathsf{K}}$ of a message M with some key K. An encryption could be taken as a function parameterized by a key assigning to plaintext corresponding ciphertext.

The most important feature of the encryption is the fact, that it hides the information contained in a plaintext into ciphertext that is without the knowledge of corresponding key meaningless. It means that the probability that an attacker could successfully perform the inverse operation to obtain the plaintext is small enough. More commonly this probability is expressed as a middle time necessary to obtain the plaintext (or the key). What is "enough" depends on the situation. The protection should withstand the whole period during which the information have to be protected, despite the progress of cryptanalysis, the grow of computing power etc.

From our point of view, besides a hiding of encrypted information, an encryption has another important feature. It completely changes the processed data in such a way that without a knowledge of the relevant key, there is no correspondence between a plaintext and the resulting ciphertext.

The inverse operation - decryption is written as follows: $\{M\}_{\mathsf{K}}^{-1}$.

When necessary we will distinguish between a long-term key K written in uppercase and a short-term session key k written in lowercase. Each key K belongs to some principal $\mathcal{A}$ which is indicated by $\mathsf{K}_{\mathcal{A}}$, or is shared between two or more principals $\mathsf{K}_{\mathcal{AB}}$.

We will use both asymmetric (public – private key pair) keys and symmetric (secret) keys. This classification is not fully correct in fact. But we have decided to adhere with the common notation and to omit the sparse class of asymmetric ciphers that are not public-key algorithms. So everywhere the term "asymmetric encryption" or "asymmetric key" is used the public-key algorithm or key respectively is meant. An asymmetric key is everywhere indicated whether it is a public key Kpbl or a private key Kprv respectively. We adhere with names "public" and "private" for asymmetric keys. Names "encipherment key" and "decipherment key" used sometimes are confusing a bit, because both keys are used for encryption or decryption occasionally.

Many times we need to encrypt some data with an asymmetric key. It is not very practical, however, use directly a public-key algorithm for encryption due to its slowness. The solution in such a case is operation called *sealing*. The encryption of data is performed in two steps. In the first step a short-term symmetric key is generated and the data is encrypted with it. Then the symmetric key itself is encrypted with an appropriate asymmetric key and the encrypted symmetric key is concatenated to the encrypted data. When recipient wants to decrypt data, it first decrypts the concatenated symmetric key using its asymmetric key and then the data using obtained symmetric key. Because modern symmetric algorithms are about 1000 times faster than asymmetric ones achieving the same security degree, the sealing is much more efficient than simple asymmetric key encryption. Furthermore we can seal the same data with more than only a one asymmetric key to allow to more principals to read the data without the need of share any key with them all and without significant grow of the data. Unfortunately, an encryption with asymmetric key can not be substituted with sealing everywhere. A sealing could be considered to be a standard way of encryption of data with asymmetric algorithm. A sealing of a message M with a key Kpbl written $[M]_{Kpbl}$ is in fact shortcut for more complicated expression $\{\mathsf{k}\}_{\mathsf{Kpbl}} | \{M\}_{\mathsf{k}}$. The second expression describes the matter exactly. The sealing with more than one key is written $[M]_{\mathsf{Kpbl}_1, \mathsf{Kpbl}_2}$. On contrary in the case when the sealing can not be used and the direct encryption with an asymmetric key is required, we will use standard denotation: $\{M\}_{Kpbl}$.

Another important cryptographic operation is a *one-way hash function* [13]. Although useful itself it is even more important for creation of digital signatures. A one-way hash function maps any data (the size is usually limited, but the limit is used to be large enough for any practical application) to fixed size result that depends tightly on every part of the data. The important feature of one-way hash functions is that it is very complicated to find for given result corresponding value and to find another data different from the fixed original data producing the same result. So it is difficult to substitute different data into a pair data–corresponding hash value. We will use these hash functions only not using secret key during operation. Such functions are named *digest functions* and their result simply *digest*. It is reasonable to use functions that are also *collision resistant*, i.e. it is difficult to find two different data yields the same result. Application of a digest function on a data is written $\|M\|$.

Digital signature of a data is a bit string possessing following features. It is unforgeable and authentic – it means that nobody, but the signer, could create it and thus sign the data. Signature is not reusable – it is tightly coupled with the signed data and can not be used in any way to create signature of any other data. Signed data can not be altered without corruption validity of the signature. For our purposes less important is the last feature – signature can not be repudiated i.e. signer can not deny that he did sign it. We will use only those signatures, that are *publicly verifiable.* A publicly verifiable signature could be verified by anyone not using any secret piece of information. It is important to use such mechanism of creation and verification of digital signatures not involving any central authority. Such signatures are mostly created using public-key enciphering algorithms. In such a case a fast hash (digest) algorithm is used to produce relatively small digest of the data and then the digest is encrypted with secret-key algorithm using signer's secret key. Due to features of a digest algorithm and an appropriate public-key algorithm it is not difficult to realize that this way created signature has all required features. A digital signature of data M with respect to a secret key Kprvs is denoted $\langle M\rangle_{Kprv}$. Again it is a shortcut for $\{\|M\|\}_{\mathsf{Kprvs}}$. We use another shortcut too: $\langle\!\langle M\rangle\!\rangle_{\mathsf{Kprvs}}$ stands for $M|\langle M\rangle_{\mathsf{Kprvs}}$.

A very important building block is a *random (or pseudo-random) number generator.* A problem of constructing and testing of good generators is discussed extensively in $\boxed{\text{Knuth: Art2}}$. For usage together with cryptographic functions we need more strong conditions to be fulfilled. Besides traditional features of standard generators, such as distribution of probabilities in output, length of a period of generator, absence of repeating patterns in output, etc. we need that sequence generated will be unpredictable. We say that pseudo-random generator is *unpredictable to the right* if and only if given sequence produced by generator of any length (shorter than the period of generator), it is difficult to predict following characters that will be produced. Symmetrically should be defined unpredictability of a generator to the right.

We will use random values of different quality. For the most demanding usage truly random values are needed. They have to be produced by a generator based on an appropriate physical random process. A free-running oscillator could serve as an example of such generator. Because generating of large volumes of random values without special hardware devices (which tends to be a typical situation on most of todays computers) is likely to be pretending both to the aspect of a processor load and a time consumption, we will use also algorithmic unpredictable pseudo-random generators. [9, 6] In such a case an algorithmic generator have to be "seeded" with a truly random seed at least. So whenever an abbreviation "random (number) generator" will be used in further text, a physical truly random process based generator is to be used. When only a "pseudo-random (number) generator" is mentioned an algorithmic unpredictable generator could be used instead of a truly random one.

2.3 Practical note

Till now we have discussed used cryptographic techniques generally only. Although it is possible to continue in explanation without knowing which algorithms exactly we are using, lets spend a few words about that problem.

It should be repeated, that the *Service* is not dependent on any specific algorithm. We need an algorithm with specified features only. We even suppose that in future implementations, the algorithms used by us will be replaced with more contemporary ones. But in our opinion it may help you to better imagine the whole system when will be possible think about real algorithms, not on some theoretic ones only.

We use slightly modified RSA algorithm suite – i.e. RSA algorithm [11] for both signing and public-key encryption, MD5 [12] as digest algorithm, D-H algorithm [14] for dynamic key agreement and Blowfish as a symmetric cipher for bulk encryption. Our selection is "matter of taste", for example the combination DSA – SHA – KEA could be used with analogous results. We only think that RSA data security is a better firm than NSA.

3 To put it simply

As has been stated in the previous section the Eternity Service represents one possible solution of the data storage problem. In this section we will describe the main ideas behind the *Service* shortly. Only a basic principles that characterizes the *Service* will be discussed here. The purpose of this section is to enable to the reader to gain an overview of the whole problem more quickly and to better understand to our further reasoning. Any problem mentioned here will be discussed in following parts of this work deeply and more strictly.

The goal of the *Service* is to assure good availability and resiliency of the data. It is proposed to be highly reliable. The Service does not provide any specific means to achieve data privacy.

The basic idea is very simple, but almost ingenious. Using conventional backup brings several disadvantages. First of all, accordingly to backup plans, typically only one or only a few copies of the data is created. The backup copy is then stored at well defined (and thus easy revealable) place. On the top of the problem a data owner have to conduct safe depository for backup copies. It represents either by most manual handling of copies which tends to be slow and complicated either some more sophisticated automatized solution, which probably will be too expensive, especially for individual users. All we need is easy to use on-line solution providing good reliability. The Internet is likely to be a right way.

Our data we randomly spread on servers across the world in an arbitrary number of copies. We will keep no record about which servers were involved. Instead we associate with the data appropriate list of keywords which will enable us to acquire data later. Server stores the list of keywords along with the data. Within the process of storage of the data we agree with the server on the period, for which the server will hold the data. When the agreed period is passed server will remove the data automatically. Server does not know identity of the data issuer.

This way we have stored our data. In the case we want to access it we will just contact any server and ask him to provide data matching our criteria. The exact form of the criteria is not a very important. It could be list of keywords, name of file in which the data is stored etc. The server checks, whether he holds any data satisfying our request. If not, it randomly selects an arbitrary number of other servers and resends our request to them. Of course depth of such search have to be restricted in some appropriate manner. Server than summarizes responses from other server and the result sends back to us.

As you can see, the repertoire of services that *Service* provides to us is very simple. We can just store data for some specified period and ask any server to provide us with specified data. No more, no less. Note that server provides the data to anybody who is able to construct corresponding request.

Let us to point out some important features of the *Service*. It will at once clarify the reasons leading to the proposition of the system. First of all note that the data can be accessed by anybody from anywhere. Access to the data could be done through any server using arbitrary number of intermediate cooperating servers. It brings a basic difficulty to our potential attacker trying to prevent us from using of the data. This way it have to guard too many entry points to the *Service*.

The second important feature is the fact that a server does not associate the data with any user, or owner. On contrary it even provides the data to anybody without inquiring his identity. It prevents our attacker from attempts to disturb communication with some specific principal which is much more feasible than to disturb whole communication.

The third recall that data issuer does not keep any record of the location of stored data. It is important in the case of active opponent tempting to damage or modify the data. Now he has only a little chance to find the data and manipulate it. Realize that if he wants to destroy the data, he have the only possibility. He must find out *all* copies of the data. It seems to be rather complicated.

The last important feature is the fact, that servers used for storage of copies of the data are selected randomly. This arrangement makes position of active attacker even more complicated. It is not the only advantage of such arrangement. Randomly selected servers are likely to be spread across the world. Its will be located in different geographical, political, religional and others regions. It will strengthen the "passive" resistance and the reliability of the *Service*.

The *Service* protects our data against all threads as the traditional backup does. We suppose that reliability of the *Service* is at least as good as in the case of backup, but probably better. In addition, the *Service*, thank to its specific features, can protect data against less commonly considered threads, such as skilled and purposeful opponent, legal force and others mentioned in previous section. Unlike backup the *Service* also protects against insider attacks.

The special attention worth very uncommon set of operations that the *Service* provides to surroundings. The data can not be changed or deleted prematurely by anyone, including their creator. It has two practical consequences. The first is reliability growth. Nobody can masquerade the real data owner and damage the data. It simplifies the whole design significantly. Nobody can also using bribery, extortion, and similar practices to force real data creator or somebody of his authorized employees to destroy the data.

The second consequence is, maybe, even more interesting. The *Service* represents highly reliable "memory" which could be valuable by itself. The reader could easily imagine various utilization of the system with such feature. We will point out some applications to cryptography shortly. For example the *Service* could serve for storage of public keys and certificates. Using the features provided by *Service* a very simple implementation of time-stamps and digital signatures is possible. The protocol for bit commitment could be simplified too. here comes the additional examples

4 At the very beginning

In this section we will rewrite shortly the original proposals of the *Service* by Ross Anderson [1]. We will attempt to summarize all Anderson's ideas as exactly as possible, but using our terminology introduced previously. The goal of this section is to describe the point, from which had started our work on the problem and to better separate the ideas coming from Anderson from our own proposals.

4.1 Specification

The basic idea of system functionality is as follows. Any client prepares data to be stored, and a digital payment order. The amount of money will depend on the size of data and the time, for which the data is to be stored. The data and the payment order will be passed together to some *Service* server. The server will ensure placement of copies of the data on (pseudo)randomly selected *Service* servers. From this moment the information represented by the data is available to anybody requesting it until the pre-payed time period expires. The data can not be deleted, so the information can not be destroyed.

A diversity of individual servers and the fact, that servers will be dispersed in many jurisdictions assure good resistance against external attacks and disasters. A very important feature of the *Service* is a lack of any central part. This way it is free of any single point which damaging could cause that large parts or even a whole system could not continue in a proper work.

4.2 Structure of the system

Anderson supposes two-layered architecture of the system. Each layer is created using servers of a specific type. A lower layer contains so called *file servers*. File servers are intended to be relatively simple devices providing storage capacity and capable to store and provide data files in accordance to commands from upper layer servers. File servers does not contain any internal logic directing manipulation with data. They does not enter into contact with clients.

The upper layer is formed from *security servers*. They incorporates a whole logic driving the system. When a client wants anything from the *Service* it contacts a local security server which either itself, either in cooperation with other servers selected in random, fulfills the request and sends a response to the client. Security server is aware, which data are stored at file servers under its control, responses to requests for shipping of a data and drives storage of a new data to file servers. Each security server controls a number of file servers that again are as much as possible spread in various jurisdictions. In order to improve overall efficiency of the *Service* security server maintains a local cache of most requested data.

Each security server contains tamper-proof hardware trusted computing base (TCB). A TCB performs a substantial part of cryptography involving manipulation wit stored data. Using a TCB we can achieve an important but a bit surprising feature of the *Service*: neither the conductor of a security server neither owner of a file server knows which information represents the stored data.

The *Service* should maintain a global directory of stored files. The system's directory itself should be stored as a regular file. It will contain logical names of a data, and with each of them some further information such as a keyword list, a plaintext name etc. allowing to people not remembering the logical name to find out the relevant data and thus obtain the requested information.

4.3 Communication, data handling

All communication has to be anonymised and encrypted. As a suitable protection for internal communication between *Service* servers was recommended dc-net proposed by Chaum [5]. A Mix-net technology [4] by the same author is suggested as a good solution for communication between client and security server. Encryption of the whole communication is important for reducing of the risk of selective denial–of–service attack. A padding traffic should be used in order to make traffic analysis more difficult.

During any communication with client is not inquired client's identity. Data is stored anonymously with no record who had submitted or requested it. This arrangement strengthen resistance of the *Service* against selective attacks.

Service does not provide any means to assure privacy of the data. This problem can not be overcome by no means, it is a consequence of anonymity of contacts with clients.

4.4 Other arrangements

Two more prerequisitions worth our attention. The first of them is so-called "perjury trap". A log-on procedure of server's administrator will require declaration under oath, that administrator is a free agent. The log-on banner should state, that access is authorized only under the condition, that he is really free. It is probable that in most democratic countries court will not compel anybody to lie committing perjury.

Conducting of *Service* servers is likely to be a long term business. It is necessary to include some in-service upgrade capability into design so that servers could follow progress in cryptology and computer engineering. Also an occurrence of programming bugs and design flaws must be taken into account.

4.5 Problems left opened

An appropriate payment protocol have to be designed. A sort of digital cash should be associated with each data. Such cash should be anonymous both from client's and from server's side so that nobody can attack *Service* by delaying or freezing specific payments. We also need that payments to server's owners will be carried out at the end of the period for which they are issued. There is another problem tightly coupled with this need. We have to design some mechanism allowing to servers to prove that it is still holding the data but stopping them from cheating by claiming payments without keeping the data all the time at once. Especially we can avoid the possibility that server "borrows" data from another one and use it during proof construction.

The *Service* server is highly sensitive to accurate time. The data is deleted automatically after an agreed period is passed, so the simplest attack will be to "push" server's clock forward and let him to delete data prematurely. *Service* have to incorporate secure and reliable time service or gain reliable time information by another means. Some extra rules could be constructed avoiding this peril. For example we can state that a data can not be deleted before all payments associated with it have been received.

5 Design - goals and decisions

First of all we will turn back to Anderson's proposal shortly. We will discuss it from our point of view and then we will show some parts that, in our opinion, should be designed better. In the next part we will introduce goals that should the new design of the *Service* match and design guidelines leading to this. We will try to convince you that our goals are well selected and that they constitute a good basis for the new *Service* design.

5.1 Analysis of Anderson's proposal

Although the basic idea of the *Service* – to distribute data randomly on servers over the world – is very good, the Anderson's proposal contains some drawbacks.

Dedicated security servers represents an obvious objective of attack. Size of the *Service* is this way reduced to the total number of security servers instead of all servers within it. The fact that data are further distributed to file servers is irrelevant from this point of view. An interruption of operation of a security server causes detriment to all file servers under its control. When the security server becomes lost all the data in its jurisdiction becomes unusable. This fact could be omitted in the case that every communication with security servers will be unconditionally secure, i. e. there will be no way to trace data along the transfer path to the server. Although it is theoretically possible it si hard to achieve in real life. To hide security servers using Chaum's dc–net protocol for example, we will need that every (or significant number at least) nodes connected to Internet will take part and receive all communication involved. Using dc–nets only to hide the exact server that stores data among others is possible but far from being effective, when we consider the typical expected size of a data. So we must take into account that tracing of the whole *Service* is probably very difficult, but not impossible. A difficulty of tracing growths rapidly with the size of the system to be analyzed. Thus reducing of the overall size of the *Service* could result into significant weakening of its security.

We presume that a too strong role is adjudicated to security servers. It seems to be reasonable that security servers will be 'hidden' as much as possible to reduce risk of direct attack against them. A placing data into *Service* in such a way that we will contact some (even the local) security server and ask it to create required copies and store its at randomly selected servers seems to be flimsy. Furthermore the digital cash have to be submitted together with the data. In the case of a server malfunction some or all of the data copies will be lost or placed unappropriately and we will have problems with cash refunding. The situation is even worse when the contacted server is driven by our opponent and intentionally behaves incorrectly. It can for example receive payments, acknowledge placing copies of the data at other servers but in fact do nothing but discard the data. As we will see further, such arrangement would bring difficulties to the problem of payments and proofs that data are still placed at the server.

Another difficult–to–realize thing seems to be the index containing list of all currently available information (or data?). We afraid that the index will tends to be a very large affair. It is not a very clear who will initiate a collecting of the knowledge to construct a new index and who assures related computing. It is likely that collecting of necessary knowledge will take considerable effort resulting in a list that will be obsolete just in the moment of its finishing. At more there is no clear purpose of having of such thing.

Using different communication protocols between a server and a client and two servers does not seem to bring any significant advantage. Because anybody could act as an *Service* server there is no need to distinguish between these types of communication. It is even advantageous to 'standardize' all communication. It could help at least to 'hide' *Service* servers among machines of clients. When our opponent intercepts some piece of communication it could not ascertain whether it comes from a server or from an uninteresting client only. A specific value represents the fact that uniform communication requires only one protocol to be developed, implemented, verified and tested. Using dc–net for communication between a security server and its file server brings only a little improvement of a security. It prevents the security server to locate where the data is stored once placed on any file server. An opponent capable to locate a security server managing the data however could probably locate all file servers under its jurisdiction (dc–net does not hide the information, who takes part in communication, it only hides, who exactly is transmitting) and find out the data there.

5.2 Design goals

First of all we will summarize the goal of the *Service* in few sentences:

We want the *Service* to be a system capable to store a data of high importance. It should offer high degree of reliability and data availability. Data have to be protected against any imaginable thread. The degree of security should be variable in accordance to data nature and client's requirements. The system should stay operational even in the case of arbitrary large damage.

Let us subdivide the process of creating of a basic principles of our design into two steps. First we will try to enumerate all the features that we assume to be desirable for resulting system. In the following section we will show, by using of which means we want to achieve these features. Looks logical, doesn't it?

5.2.1 Unlimited availability

Because the main goal and the reason for the very existence of the *Service* is to assure good availability of a stored data, it is reasonable to require a good availability of the *Service* itself. It should have as many entry points as possible. The results provided by *Service* should be equivalent despite which particular entry point was used for submitting of a request. It is necessary that for our potential will be as difficult as possible to block communication with servers by filtering or any other means. The underlaying communication platform used should be selected carefully to be commonly used by other applications. The functionality should not depend on any "magical numbers" for example of communication ports, address etc. The communication involved by *Service* should be, if possible, similar to any other data flow.

5.2.2 High resistance

The *Service* should assure that data once stored will be effectively available until an agreed period expires. Thus it is necessary that it is able to withstand a damage caused by broad variety of reasons. The reasons are often called *threads*. Let us discuss them in more details. The threads we can classify into several groups. Some groups are not sharply separated and we can expect simultaneous impact of influences ranging into different groups.

The first it is a thread to data caused by **blunt influences**. A blunt influences should be characterized as phenomenons that, although often harmful, are not directed against the data itself. They does not posses the ability to interpret our data or to understand to principles of our system. The significant characteristics of it is that they acts independently on the current state of the system and without any feedback from the system behavior or system responses to their actions. The damage of a data or loss of it is often only a side effect of such influence. They represents danger for the data existence. Only occasionally they could endanger data integrity. Among blunt influences we classify technical failures (medium illegibility, controller malfunctions), natural disasters (floods, fires), thieves and various vandals.

Second group consists of **amateur opponents**. Our opponent is always a human with various degree of motivation, understanding to the functionality our system capable to interpret a data. He could try actively influence the system behavior and effectively employ feedback obtained by observation of the system activity. His effort primarily leads to some damage or modification of particular data. Although it could possess significant knowledge his equipment and methods are limited. Besides traditional anonymous external attacker here comes an employee of server owner etc.

The third group – **professional opponents** – is very similar to previous one. The difference is that professional opponent dispose of a significant knowledge, technical equipment, investigational or intelligence capabilities and financial resources. This allows him to penetrate or avoid much more security mechanisms that an amateur is able. Goals of professional opponents are almost the same as amateur's. Into this group comes big companies, intelligence and various special services, terrorist organizations etc.

The bigger danger represents the last group – **authorities**. Their goals are generally the same as in the previous two cases. We should assume that they dispose with the same potential as professional opponents. The difference is that authorities could act in such way that will impact a large parts of the whole *Service* besides only a particular server as it is likely in the case of professional. Their great advantage is that they must not act latently, at least from point of view of a public. An authority could force server owners or conductors to destroy data or stop providing the service by themselves. Among others the authorities are governments, courts, religional communities etc.

Obviously we want that *Service* will be highly resistant to all of above mentioned threads. We require that it will be able to withstand large damage or temporary or permanent loss of its significant part without any serious consequences if possible. The resistance against long–term influence of active opponents is also very important.

5.2.3 Extendibility

We believe that the best defense of data is the size of the whole system. The *Service* has to be large, only a large system can assure information against attacks. The large system is less susceptible to impacts of blunt influences, it is worse subverted by amateurs and professionals because of high cost of attack. Only a truly global system could withstand centralized influence of authorities. So provisions must be created allowing fast and wide spread of the *Service*. On the other hand, it is undesirable that majority of *Service* servers will be owned or conducted by a single individual or a company.

Individual parts of the *Service* should be independent as much as possible. There should not be any certification process or so when a new part of *Service* is created and starts working. Independence of individual parts brigs another advantage – in the case of destroying of any server no consequences will occur on any other part of the system. We must take into account that at least first installations of the *Service* servers will be conducted mostly by enthusiasts.

5.2.4 Fully distributed design

Any centralized part of the *Service* represents point of future problems. Centralized parts are likely to cause bottle–necks due to insufficient power, communication capacity etc. Damage, unavailability or even a loss of such part will cause stopping of the whole *Service* for some time at least. Thus the most efficient way for any attacker will be attack some centralized part of the system. This is why we will avoid such things in our design for almost any price. Instead we will introduce the notion of a *freedom*. Each principal within the *Service* including clients needs acquire some duties from others. Some of such duties will not be produced inside the *Service* (for example accurate time from atomic clock or bank services), but will be taken from outside sources. We do our best so that in any such situation there will be as broad as possible palette of options, how or where required service obtain. This way we will avoid dependence on any service. Allowing to any principal (or server conductor) to obtain needed external services from the source of his choice (and thus randomly selected from a global point of view) will strengthen the *Service*.

5.2.5 Portability and reusability

It is reasonable to require that *Service* servers could be ported and operated on various hardware and operating system platforms. This requirement is somewhat overlapping with our freedom principle discussed in section 5.2.4. Here thought we follow slightly different goal: we want to assure that almost anybody will be able to start *Service* server on his current equipment without need to change it. We assume to be very important (within the initial time period at least) that *Service* servers will be able to coexist with another software on such machines as network routers, gateways, http servers–etc. are. Although using special hardware devices will bring better results, it should be possible to conduct server without such equipment – in fact we are not able anyhow to prohibit anybody from replacing of a hardware based module with a software one.

Some parts of *Service* seems to perform similar tasks that could be effectively employed by other similar services, such as anonymous remailers or web anonymizers are. The relevant parts of the *Service* should be designed in this manner that such coexistence with other systems will be possible.

5.2.6 In service upgrade

Conducting of a *Service* server is likely to be a long–term affair. The mutually overlapping time periods of storage of individual data will cause that server has to be in–service uninterruptly. On contrary there are many reasons to stop server and replace some of its parts. The cryptography that is widely used continuously develops and new algorithms, different key lengths or new standards have to be introduced in order to preserve required level of security. We also can not completely avoid presence of design or implementation bugs. A new more efficient better featured versions of server software will arises. The server conductor will decide to switch to completely different hardware or software platform due to whatever reason. A hardware failure will occur or a replacement with new more powerful hardware will became necessary. All such facts and many more leads us to conclusion that will be desirable to try to incorporate into design appropriate provisions so that servers could be upgraded without need to interrupt working.

5.2.7 Terseness and simplicity

The more code, the bigger possibility of a mistake, a design flaw or a need to replace with newer one. The *Service* servers should incorporate only the functions absolutely necessary for proper functionality. Any other functions that could be reliably acquired from surrounding environment should not be implemented within the servers again. The principle of freedom namely have to be kept. Substantial functionality could be transferred to clients. The advantage of this solution easily visible. Client's software could be simply replaced with newer versions without need of exchanging costly hardware devices. Also it brings better division of computation between client and servers.

We will tempt to restrict used cryptography to the techniques that are well known and widely considered to be reliable whenever it will be possible.

5.2.8 Separation of cryptography

The *Service* will contain substantial amount of various cryptography functions. To avoid problems with various export/import restrictions in many countries, it is necessary that all parts of the proposal and the further implementation will be done in such way that all cryptography is fully separated from other logic. This way with clearly defined interfaces the cryptography libraries could be substituted with other ones created in accordance with legal restrictions applied in relevant country without the need to re-implement any other parts of the *Service*. Such arrangement will also simplify upgrades of obsoleted cryptography.

5.2.9 Good habits

The requirement of a good habits does not address any particular part of a *Service*. Much more it concerns the way by which the running *Service* will behave and interact with surroundings. Thus this requirement have to do with general programming style used and with a reasonable design level. We request that *Service* will be non-obtrusive, so our goal will be to minimize communication load produced, to reduce processor load and to keep disc and memory space needed for running servers as low as possible. These claims looks self–evident, but at a first glance only. To say it in other words we will not use such techniques that are not efficient although correct from theoretical point of view. We will try to avoid techniques causing inappropriate load of any sort, very complex computation, required any global or hardly to achieve information about the system etc.

The *Service* will not be widely used if a simple and easy to understand user interface will not be available. Although it is not a live necessary part of the system this problem have to be discussed and an appropriate means have to be incorporated allowing easy access. The problem should be solved within a basic design because later addition could cause security flaws.

6 Proposal overview

In this section we will describe the overall structure of our design of the *Service*. Then we will direct readers attention to the most important design features. The last part of this section is dedicated to elucidation of the reasons of every design decisions in the terms of goals listed in section 5.2. A detailed description of main communication protocols and a minute discussion of each part of the *Service* will be presented in following sections.

6.1 Basic structure

We will stay with the basic principle of the *Service*. It is made up from many and many servers that, we hope, will be spread along the world. To assure all needed functionality we will use servers of different types. Servers of one type are all equivalent providing exactly the same services for clients.

The core of the *Service* represents *Eternity servers (ES)*. Eternity servers provides storage capacity for clients data, handles requests from clients to store data or to find out specified information. The *ES* is designed so that for anybody including the server operator will be very difficult to ascertain which data server holds. It is clear that we have no opportunity to prevent operator from finding out whether its server holds some particular data, but we can assure that even the operator will not be able to locate the data at server. Thus he can delete the data only at high risk that many other data will be injured. The *ES* is designed in such manner that selected set of functions could be implemented using special hardware device – the *Trusted Computing Base (TCB)* achieving better resistance and protection of server operator. Besides the *ES* there are some types of auxiliary servers.

The live necessary are *Mix servers (MX)* or shortly Mixes. They support all communication within the *Service*. The reason of introducing of Mixes is that they offers a possibility of a cheap extension of the system size. They play important role in generating of a padding traffic and keeping communication secret and untraceable.

Another important servers are the *Access certificate servers (ACS)*. As we will see further we tempt to hide all parts of the *Service* as good as possible. In such situation we have to create appropriate entry point into system allowing to clients to acquire informations needed to contact concealed servers. As we will see further there is a number of various certificates that are necessary to obtain in order to be able communicate with most servers within the *Service*. The *access certificates* issued by Eternity servers are of biggest importance.

To the category of 'public relations' belongs the *Eternity proxy servers (EPX)*. Although not live necessary they could easily turn into most significant attribute assuring a wide usage of the *Service*. We start from assumption that majority of data stored within the system will not be critical in the sence of being a probable target of any massive attack. Such data will be stored by amateurs wishing simply to secure its information against technical failure or theft. Client of this type will not wish to cope with relatively complicated procedure of data submitting directly to selected *ES*. Instead it could transfer this task to *EPX* at the cost of somewhat (but in such case probably little) decrease of a security of its information. The benefit is a simple easy–to–use interface without a need of any custom software. Apparently if anybody requires a 'top level security' it could using special software contact directly some *ES* and avoid any possible risk.

Besides these main types of servers the *Service* uses some servers that could be viewed as a abstraction of services acquired from outer providers. As an example of such servers could serve a *bank*. At least some of this still existing external providers will have to incorporate a new type of service or a new communication protocol at least to be capable to cooperate with the *Service*. It means a new software module have to be created so the word "server" in this case seems to be appropriate. Another example from this category is a *time server* – a provider of an exact real time information.

An important part of the *Service* is a *Client module (CM)*. A *CM* contains necessary functionality allowing to client contact properly servers and employ provided services. Typically it have to implement protocol for message transfer within the *Service*, working with informations needed to contact servers etc. Optionally it could contain protocol for digital payments that allows to a client to simply carry out all necessary actions leading to storage of data at server. The *CM* could also incorporate some functions allowing to client better manipulate with its data and to achieve some important features that are not provided by *Service*. For example it could be a compression routine, appropriate encryption algorithm for assuring secrecy of the data or a threshold scheme for better protection of individual copies of data. In the case that client wishes to use *EPX* to contact *Service* a Client module could be very simple – for example general http browser will be sufficient.

The operation of the *Service* is very simple. Anybody could in any time arrange selected type of server and launch its operation. A simultaneous operation of various types of servers is also possible and in many cases brings better results. The most reasonable looks following variants:

1. A dedicated Mix server – a simple 'opportunity for everybody', *MX* could be conducted at various machines driving the communication flow within the Internet, such as routers, gateways, mail, http and ftp servers are.

2. A Mix server in combination with Eternity server – a variant for 'eternity professionals'. Simultaneously operating Mix will help to hide much more important *ES* behind its own communication flow.

3. A Mix server together with Access certificate server – why don't operate Mix, *ACS* have to be capable of communicate using *Service* specific protocols anyway. The *ACS* itself could be conducted together with any http or ftp server for example with almost no more expenses.

Although other combinations are also possible, we render them as less reasonable. It is important that anybody could select which server it will operate independently, no certification or registration is required. Server operator will select his own set of external services providers. This selection should be based on the personal knowledge and experience of the operator and could be subject of change in further time.

This is the way the *Service* will come into existence. Now we will discuss the principles of operation of it.

6.2 Operation

The repertoire of functions is the same as in the Anderson's proposal – *Service* offers possibility to store data and to retrieve requested information, if available. A client could store its data at any *ES*. A period of data storage is agreed during this process and corresponding money are deposited in the bank in accordance with server's requirement. Payments to server (i.e. server conductor) will be carried out conformably to agreed installment plan that should cover the whole period of data deposition. Before receiving a payment server has to prove that it is still storing the data. All parameters of a transaction, i.e. length of storage period, total price, installment plan and bank selection are fully dependent on the agreement between a server and a client.

When a data is stored to a server a list of keywords should be associated with it. Server will append a logical name to every data. Both this structures could be used to form a request when client wishes to obtain some information. It can specify some keywords characterizing requested information or logical name under which the information is supposed to be stored. Formed request client passes to arbitrary *ES* or *EPX*. In the case the request was submitted via *EPX* this server transforms it into internal form and resends it to some *ES*. An *ES* first looks–up whether it holds corresponding data by itself. If so, it responses back with this data. If not, it randomly selects a number of other *ES*–s and resends request to them. The depth of such search process is limited somehow. If the size of a response exceeds a limit due to too high number of corresponding data, the server responses only with a list of data matching the request and asks client to specify more strict search conditions. A general rule is used that response to any request is sent back to immediate requester (i.e. a server at n-th level of a recursive search sends its response to a server at level $n-1$ instead directly to client).

6.3 Communication

We assume that effective attack that brings a serious harm to *Service* could lead either directly against the *ES*–es or against communication between *Service* servers. Other parts of the system are nearly useless, causing large enough damage to threaten system functionality (and thus any particular data) is unlikely. This is why the good overall design of a communication is crucial.

A whole communication within the service is encrypted and anonymised. As was noticed in [10] keeping even the non–secret communication anonymous is very important because it avoids or at least significantly reduces the possibility to mount selective attack against the system. This is very important, especially in the case of a very large system such as the *Service* is going to be since mounting an attack against whole system will be probably much more unfeasible. Although the *Service* does not introduce a notion of data ownership or user identity the problem is still here. Our opponent could direct its actions against some type of data (containing selected words, coming from particular server or so). Thus we will keep all communication end–to–end encrypted.

We tempt to hide Eternity servers as good as possible. An obvious way how to locate the server holding particular data is to submit corresponding request and attempt to track communication to the server. To prevent such attack, we have introduced a special routing mechanism – *Eternity Routing Mechanism (erm)*. It is similar to protocols proposed in [4, 7, 2]. *Erm* assures that the 'look' of a message is completely changed at every point of transfer path. The size of a messages is rounded before transmission in order to achieve sort of uniformity of message sizes. The message is at every point delayed randomly selected amount of time, changing the order of incoming and corresponding outgoing messages.

Nor sender, nor recipient of a message knows any information about other party except the digital pseudonym. The *erm* allows to communicate to both parties without exposing them to thread of revealing their identity, location of the server or the place from where the client is currently working etc.

The last, but not least significant, provision is a padding traffic. Each server within *Service* generates dumb messages from time to time and sends them to randomly selected other machines. Recipient of such message simply discards it without any following action. This way the whole outgoing and incoming traffic at any server has almost stochastic character. The total amount of padding traffic depends on the number of useful data to be transmitted. Real data should remain hidden among dumb messages. Padding traffic avoids possibility of an another serious thread – traffic analysis.

6.4 General design features

Here we will highlight the most important features of our proposal. We will attempt to relate them to requirements stated in section 5.2 and show how the *Service* with these features meets the mentioned goals.

6.4.1 Flat structure

The main characteristic of our proposal is non-hierarchical flat structure of whole *Service*. It means that any server is independent on any specific part of the rest of the system. There is no server supervising any others in any way. (further we can see that internal structure of some servers allow somewhat hierarchy, but this has nothing to do with security problematic). The principle of a *freedom* is of great importance.

Flat design makes the *Service* easily expandable – an intending server operator may start operation without having to cooperate with anybody. It also avoids any reduction of the number of points which damage could cause data availability loss or operation interruption. In the case of successful attack against some part of the system flat structure of independent servers minimizes negative consequences for other parts. Provided that all servers of one type are equivalent and that they are contacted directly a flat structure also offers a maximum possible number of system entry points which is very important feature.

6.4.2 High *ES* protection

Protocols for unconditional communication security [5] seems to be impractical and ineffective, especially in the case of large distributed system when big data are to be transmitted. We therefore will pay our attention to protocols that makes to an attacker very difficult to trace communication. We hope, that high difficulty of data transfer tracing in conjunction with sufficient size of the system and the fact that no logs about communication are kept will offer fairly good degree of protection. If the *Service* become popular and widely used, it will be probably out of range of anybody including the best intelligence agencies to screen it.

It is clear that the objective of our protection are the Eternity servers. For this purpose a special communication protocol (see 7.1) have been developed, that is used for almost all data transfer within the *Service*. This little hyper–protection (not all transfers are so critical) is meant to increase the total number of messages that an attacker have to analyze and thus help to strengthen *ES* protection. The *ES* itself are designed so that it will be difficult to find out what data the server holds and almost impossible to ascertain where exactly the data is stored. This provision protects servers and a data against authorities.

6.4.3 Auxiliary servers

An usage of auxiliary servers should assure even better protection of the *ES*–es and thus a stored data. Mixes are cheap and easy way how to growth the system size. We suppose that number of *MX* will be several times higher than number of *ES*. It makes spying of the system more difficult and more expensive. The most communication leads to machines that are almost useless for potential attacker. Any *MX* knows almost nothing about target of transferred data and even successful gaining of control over the machine will be of little practical meaning. Realize that attacker has little opportunity to investigate in advance whether the assaulted server is *MX* or *ES*.

6.4.4 Passive behavior

Each part of the *Service* (except the Client module) is passive, i.e. it performs any activity only on demand. The server will not change

6.4.5 Randomness, concealing

"The best way to protect information is to have no information" is often said. This a bit funky sentence summarizes the matter of the problem. For our opponent will be very unpleasant to attack system where 'nobody knows anything'. To achieve this a good random or pseudo-random numbers generators and encryption is widely used. All informations concerning a data (data files itself, keyword lists, clients requests, etc.) are encrypted every time and handled in such way that the possibility of disclosure is kept at minimum that could be achieved. If possible, custom hardware devices should be used. Whenever possible the decision which principal or system resource will be used is done at random. This provisions offers a protection especially against intentional attacks. We assume that when the information will be stored using a reasonable number of copies, the random selection of a server will provide a good tolerance also against blunt influences – it is likely that data will be spread across the world.

No records about performed actions are kept, so an attacker either have to be very fast and analyze whole system 'on the flight' either must posses resources allowing him to 'buffer' the system activity within relevant period and consequently analyze gathered data.

The lack of information about system state we assume to be crucial for achieving of a good resistance. The price that we must pay for achieving of such features is very high and, in fact, it constitutes the most discutable part of our proposal (see Discus).

6.4.6 Two-layered implementation

The implementation of the *Service* is horizontally divided into two parts. The lower part represents mission independent routing mechanism. We denominate it "mission independent" because, although developed as a part of *Service* it is capable to support various other application. It provides special sort of very secure information transfer resistant to traffic analysis, data tracing etc. The upper part contains implementation of *Service* specific servers.

This solution allows us to reuse the work expended on design, implementation and testing of the routing mechanism when implementing another similar services. The other advantage comes from situation when more (Internet) services uses the same communication protocol. As we have written in 5.2.3 on of the best defense mechanism is the overall size of the system, especially when selective attacks are difficult to be mounted. In the situation when more services uses our routing mechanism a new virtual system of collective protection arises which size is equal to the sum of overall sizes of all participating systems. It si obvious that achieved protection is far better than in the case of individual systems.

7 Used protocols

In this section we will discuss deeply all communication protocols used within the *Service*. Namely it will be *Eternity Routing Mechanism (erm)* used for general communication between any servers and clients, *Payments Protocol (pp)* allowing fees for data storage to be transferred correctly and *Data Care Proof Protocol (dcp)* that is used proofs that server is still holding relevant data.

Every data transfer between some *Service* server and any other node is driven by some of this protocols or by some combinations of its. The main objective of design and usage of this protocols is protection of communicated information against intentional attack, i.e. attacks mounted by both amateur or professional opponents or authorities (see 5.2.2). We assume that protection of a communication against blunt influences provides underlaying transporting mechanism. In the case of the Internet and its TCP/IP protocol suite it looks like reasonable assumption.

The description of each protocol is divided into three parts: first part defines a scope of utilization of the protocol, the second one specifies the protocol overall characteristics and the last part contains full protocol specification.

7.1 Eternity Routing Mechanism

7.1.1 Scope of *erm* utilization

The *erm* is the main communication platform of the *Service*. It is used whenever from the character of the communication could be revealed that at least one of the communicating parties is some *Service* server. Namely the protocol is used for transfer of any communication concerning stored data (i.e. requests to store or retrieve data, responses to those requests, data itself), for communication between banks and servers induced in context of carrying payments and for submitting access certificates to *ACS*–es. On contrary there is no benefit from using *erm* for sessions requesting general services from exterior source, for example for acquiring exact time from a NTP–server.

7.1.2 Characteristics of *erm*

The purpose of *erm* is to provide very good concealing to both communication parties. This way it is one of the most important protection mechanisms. The protocol allows to communicate to two parties that both does not know the identity of its counterpart. Using digital pseudonyms it is available to repeatedly establish connection with the same principal, without any risk of revealing its identity. The basic idea of the protocol is similar to principle used in the Onion routing protocol [7]. Analogous trick uses the Mixmaster anonymous remailer [2]. Chaum had also proposed something similar to achieve anonymous mail system [4]. All mentioned protocols are suitable for applications that are sort of client—server environments. In such case they offer assuring of clients anonymity. Our protocol is suitable to balanced peer–to–peer situations where both parties needs equally strong protection.

As in the case of previous protocols, the communication is lead through a number of intermediate nodes. A special format of messages is used that allows us to conceal recipient's identity. A specific data structure called *Access certificate (AC)* is introduced that is necessary to be created for every principal wishing to communicate via *erm*.

7.1.3 *Erm* description

The routing mechanism should avoid the possibility of locating the exact place where an information is stored via tracing message transfers. Unfortunately any message have to contain complete information about its recipient in order to be deliverable. As we mentioned above, the widely used trick is to subdivide the address of a recipient of the message into several parts. These parts describes path of the message across intermediate nodes. Each part is encrypted so that it si readable only for relevant node (i.e. the node that will send the message to the node addressed by this part of description). When the message is transferred, only the very first part of the path description is transmitted as a plaintext (the rest parts are either still encrypted, either stripped from message by previous nodes). Each node first strips the first part of path description (which points to it) and then decrypts the following part. Using standard message passing mechanism the message is transferred to next node and finally to the recipient.

The would be attacker have to trace message across the nodes and at every node find where the message was routed next. To accomplish such task is not very easy and we can arrange the behavior of the nodes such that it will be even more complicated.

At first, we can completely change the message at every node. It could be done using encryption – the node encrypts the message with respective key before it is sent further. With modern fast symmetric encryption algorithms it is pretty feasible. The next possibility is to "standardize" sizes of the messages. It is similar to the method by which a files are stored on a disc. Size of a message is to be rounded up so that it is multiple of some basic block. The unused space at the end of message is filled with a random padding. Both two previous arrangements follow the same goal. Its complicate to attacker any passive analysis of communication and coupling corresponding incoming and outgoing messages. This way we make an attacker to penetrate directly all involved intermediate nodes, because external observation will bring only a little profit.

Some others provisions are suitable to make opponent's situation even more complicated, although they are concerned with the implementation of the protocol and the whole system rather than with design of the protocol. We will mind them here just for the completeness and we will return to that problems at relevant part of the work Mix. The attacker can attack directly the intermediate nodes and attempt to acquire the needed information from its internal data structures. For this reason the node should not keep any execution log of performed transactions. The node should not have any persistent data structures. All necessary information should be constructed just before the moment it is to be used and destroyed immediately after finishing of the transaction.

The mentioned mechanisms assuring message uniformity and changing its shape could be considered as sort of 'collective defense'. They are effective only if there is enough of other messages. In such situation we can every message 'hide' among other ones. So we need the system is large enough and the traffic is reasonable dense. Our opponent could refine some interesting information from monitoring of the whole network. For example he could invoke heavy communication with some node. Then by comparison of the load of the lines he could identify the path of messages.

To overcome this problem we will use a padding traffic. When there is a lack of the real communication the system itself generates an inner communication. Thanks to encryption, this traffic is indistinguishable from a real data. We must assure that the messages invoked by client requests or any other legal reasons are transmitted without any noticeable increase of traffic load. There are two ways it could be achieved. We could maintain a stable level of communication load using continuous issuing of padding messages. When the real data transfer is necessary, we only the padding messages replace with real messages. The second way is to randomly generate bunches of padding messages. In the case the data are to be transmitted we just stop the padding traffic production and send the data in several randomly delayed bunches of messages.

The trick is to achieve such arrangement, that no noticeable changes in statistical characteristic of message flow will owing to real data transmission. Note that the real data will represent a bigger part of a communication. An occasionally communication peaks we will spread into longer time period and padding will be used only to fill moments, where no reasonable data are available. The concrete setting depends on a local situation and is a question of fine-tuning. It is only necessary that changes in the setting will not follow changes in communication load too closely.

There left the last problem to solve. We have to construct message path description without knowing the exact location of the recipient, which looks uncontrollable. We can divide the problem into two parts that both are pretty manageable. Each node, that wants to receive any message releases so called *Access Certificate (AC)*. The certificate contains description of the second part of the path that ends at the recipient. It also contains recipient's digital pseudonym.

When anyone wants to send a message to an arbitrary recipient it first gains its *AC*. Then it completes the path description adding the first part of the path to the description provided in certificate. To be recipient able to response to the message, sender appends to the message its own access certificate. Then it prepares data and sends the message.

Message is passed through an arbitrary number of intermediate nodes, as was mentioned above. The nodes we name *Mixes*. Each part of the path description is to be encrypted in such a way so that only the relevant *Mix* could decrypt it. It could be easily done using public key encryption. For this purpose each *Mix* issues *Mix certificate (MC)* containing public key of the *Mix* and some other necessary information.

The both part of path description should be created using randomly selected *Mix*es.

We also need to change message at every *Mix*. It is done by enciphering the message with some conventional (symmetric) encryption algorithm. The problem is from where the keys will be taken. The answer is very simple, the keys are part of the path description. Besides a next Mix address, each part of the path description contains a symmetric key.

A *Mix* then works uniformly nevertheless it belongs to the first or second part of the path. The message consists of two parts. The first part is a path description, the second one is a clients data. *Mix* first of all strips its own address from the first part. Then it decrypts the first part of the message. Together with the address of the next *Mix* it decrypts a symmetric key. The key *Mix* uses to encrypt the second part of the message. It also pads the first part to the original length. After this work is done, *Mix* sends the message to a next node – unless the next address is its own too. In such case the *Mix* passes the message to an upper layer server. As a last part of its task *Mix* destroys all the data associated with the transaction.

Because the recipient had created its own *AC*, it knows all the symmetric keys used to encrypt message during the second part of the path. So it is able to take steps to decrypt the message finally. What it lacks to succeed in this mission are the keys used in the first part of a message journey. The sender must help here. It have to decrypt (in reverse order) the message before sending with all the symmetric keys it used to construct the first part of the path.

The data itself is encrypted with recipient's public key, so they are protected even it the "middle" of the path. During the first part of message transfer are removed changes involved by decryption with client specified symmetric keys. Thanks to features of encryption algorithm the changes of message between *Mix*es are totally unpredictable for everybody including the recipient. *Mix*es along the second part of a transfer repeatedly encrypts the data with its symmetric key. It makes the message untraceable for everybody including the sender. In both cases only the constructor of the corresponding part of the path could track the message. The first part of the message keeps the same length along the path. Thus no *Mix* can guess its order within the path.

There is still one problem left to be solved. The description of the second part of the path given in *Access Certificate* stays to be static along all lifetime of the certificate. In the case when any *Mix* involved will interrupt its proper function due any reason, the path described by certificate becomes broken and the transferred messages will be undeliverable. To reduce such risk, we can in every point specify more than a one potential following node only. Each *Mix* then could check, which of the following along the path *Mix*es is reachable and then sends the message to it.

Let's put it all in more formal expression.

The *Access Certificate* of the server $\mathcal{A}$ is of the form:

$$[\mathsf{S}_{n/2}|\mathrm{A}_{n/2-1}|\ldots|[\mathsf{S}_2|\mathrm{A}_1|[\mathsf{S}_1|\mathrm{A}_0]_{\mathsf{Kpbl}_{\mathcal{M}_1}}]_{\mathsf{Kpbl}_{\mathcal{M}_2}}\ldots]_{\mathsf{Kpbl}_{\mathcal{M}_{n/2}}}\wr$$

$$\wr;\mathsf{Kpbl}_{\mathcal{A}};\mathit{Texp};\mathit{Tmax};\mathit{Misc};\mathit{Sign}$$

Here Kpbl_i denotes public key of *Mix i*, A_i its address and S_i is the symmetric key which the *Mix i* will use to encrypt transferred data before sending it to a next node. $\mathsf{Kpbl}_{\mathcal{A}}$ is a public key of the server $\mathcal{A}$ used for encryption, *Texp* is an expiration time of the certificate, *Tmax* a maximal time for which a data can be stored at the server and *Misc* represents a space for additional information. The whole certificate is digitally signed and the resulting sign *Sign* is appended to its end. The *Sign* is obviously formed from the previous pars of the *AC*: $\langle\{\mathsf{S}_{n/2}|\ldots\}_{\mathsf{Kpbl}_{\mathcal{M}_{n/2}}}|\ldots|\mathit{Misc}\rangle_{\mathsf{Kprvs}_A}$, where Kprvs_A is the private key of the server $\mathcal{A}$ used for signing (we recommend to use different asymmetric keys for encryption and for signing).

Here is a message just prepared by sender:

$$\mathrm{A}_n|[\mathsf{S}_n|\mathrm{A}_{n-1}|[\mathsf{S}_{n-1}|\ldots|\mathrm{A}_1|[\mathsf{S}_1|\mathrm{A}_0]_{\mathsf{Kpbl}_{\mathcal{M}_1}}\ldots]_{\mathsf{Kpbl}_{\mathcal{M}_{n-1}}}]_{\mathsf{Kpbl}_{\mathcal{M}_n}}\wr$$

$$\wr|\{\{\ldots\{\{AC|data\}_{\mathsf{Kpbl}_{\mathcal{B}}}\}^{-1}_{\mathsf{S}_{n/2+1}}\ldots\}^{-1}_{\mathsf{S}_{n-1}}\}^{-1}_{\mathsf{S}_n}$$

Recall that $\{m\}^{-1}_K$ means decipherment of a message m with key K.

Bellow the paragraph you can see the same message after processing at first along the path Mix just before sending to a next *Mix*. The first part was decrypted with private key of the *Mix* and after stripping the symmetric key and address of a next Mix padded with *padd* to original length. The second part was encrypted with key S_n i.e. the transformation caused by decryption with this key was removed.

$$\mathrm{A}_{n-1}|[\mathsf{S}_{n-1}|\ldots|\mathrm{A}_1|[\mathsf{S}_1|\mathrm{A}_0]_{\mathsf{Kpbl}_{\mathcal{M}_1}}\ldots]_{\mathsf{Kpbl}_{\mathcal{M}_{n-1}}}|padd\wr$$

$$\wr|\{\ldots\{\{AC|data\}_{\mathsf{Kpbl}_{\mathcal{B}}}\}^{-1}_{\mathsf{S}_{n/2+1}}\ldots\}^{-1}_{\mathsf{S}_{n-1}}$$

The second part of the message "in the middle" of the path is protected by recipient's public key. The more correct record should contain changes of the padding because each Mix appends new padding of the size of stripped information. The padding will never be interpreted, it serves only to keep the same size of the first part. So we omit this detail.

$$\mathrm{A}_{n/2}|[\mathsf{S}_{n/2-1}|\ldots|\mathrm{A}_1|[\mathsf{S}_1|\mathrm{A}_0]_{\mathsf{Kpbl}_{\mathcal{M}_1}}\ldots]_{\mathsf{Kpbl}_{\mathcal{M}_{n/2}}}|padd\wr$$

$$\wr|\{AC|data\}_{\mathsf{Kpbl}_{\mathcal{B}}}$$

The more precise record should contain changes of the padding because each Mix appends new padding of the size of stripped information. The padding will never be interpreted, it serves only to keep the same size of the first part. That is why we omit this detail.

After first Mix within the second part of the path the message looks as follows:

$$\mathrm{A}_{n/2-1}|[\mathsf{S}_{n/2-1}|\ldots|\mathrm{A}_1|[\mathsf{S}_1|\mathrm{A}_0]_{\mathsf{Kpbl}_{\mathcal{M}_1}}\ldots]_{\mathsf{Kpbl}_{\mathcal{M}_{n/2-1}}}|padd\wr$$

$$\wr|\{\{AC|data\}_{\mathsf{Kpbl}_{\mathcal{B}}}\}_{\mathsf{S}_{n/2}}$$

In fact the whole thing is little more complicated. Each address block A_i contains not only the address of a next Mix but a number of additional useful information too. Besides others there is a protocol number, used encryption algorithms and relevant parameters, identification of used public key (a Mix could issue several *MC*-s) and so on. Thus each address block A_i has its internal structure that will look as follows:

$Addr|Prot\_Num|Alg\_Num|Params|Key\_Id|Misc.$

The exact shape and meaning of individual parameters is not necessary for understanding of the protocol and will be discussed in $\boxed{\text{Implement}}$.

Complications are not over however. The *AC* prescribes through which mixes the message will be transferred. When any of these nodes become unavailable, the *ES* becomes unreachable using this certificate. Unfortunately there is no way how to realize this fact ahead of sending the data. To make the *erm* more robust we use multiple possible following mix at each point of the path. So the each part of the path description is sealed using multiple keys instead of only a one key. Also the address of next mix specifies multiple possibilities. The mix selects from these possibilities at random, where to send the message next. When the selected mix is unreachable the other one is used. When none from the possible mixes can not be reached, the message is discarded. There is the address part of the massage which the mix exactly sees:

$$\bigvee_{j=1}^{k}(\mathrm{A}_{i+1}{}^{(j)})|[\mathsf{S}_{i+1}|\ldots]_{(\mathsf{Kpbl}_{\mathcal{M}_{i+1}}{}^{(1)},\ldots\mathsf{Kpbl}_{\mathcal{M}_{i+1}}{}^{(k)})}$$

The *MC* contains a public key of the *Mix* and some additional information. For example there is a list of supported protocols, expiration time of the certificate, max size allowed size of the message etc. Again the whole certificate is signed with private key of the *Mix*.

7.2 Payments Protocol

The payment protocol had been designed as standalone general purpose protocol. Within the *Service* it is used mainly in conjunction with the Data Care Proof Protocol. So we will discuss a general characteristics and the protocol structure here. The cooperation of the protocol with *dcp*, which is very tight on account of effectiveness, will be described later in separate section.

7.2.1 Scope of *pp* utilization

The Payments Protocol allows money to be transferred from clients to server owners through selected bank. The protocol will support all the payments concerned with storage of data at Eternity servers within the *Service*. Another point of utilization of *pp* is transfer of charges payed by *ES* operators to *ACS* for publicizing of their Access certificates.

There is some new mechanisms that have to be incorporated in software systems of involved banks. First of all the bank have to be able to accept order with unspecified consignee. The second, bank have to be able to test that some specific conditions are fulfilled before the payment is carried out. And, of course, banks will have to be able to perform necessary new protocols.

7.2.2 Characteristics of *pp*

The problem of money transfers within the system not incorporating any verifiable identity of principals proved to be rather complicated. Almost all today's digital cash systems falls into one of two categories. The first one is made up of systems that performs less or more on–line transactions between payer and taker. Such systems performs only a direct money transfers from one account to another one. Into the second category falls systems utilizing some sort of digital money — some part of actual cash is turned into digital analogy of real money and the actual payments are done using such 'digital coins'.

Unfortunately, we need something between both these possibilities as will become obvious from following description.

The main objective of the Payment Protocol is assuring of better protection of both a payer and a taker. This is done through a separation of the whole transfer of the money from payer to the taker into two parts. The idea is not anything new, it even follows very closely the way that the banks virtually acts like.

Any bank that decided to support *pp* will create a special account, say *Purse Account*, dedicated to payments using *pp*. Anybody wishing to make payment using *pp* saves money to this account. Along with this he can submits set of strings called *Authorizations*. This set defines conditions that have to be full-filed before the whole payment will be carried out, i.e. before the money could be transferred to taker.

A special *Main Authorization* associated with each payment specifies three main parameters. The *Value* defines total amount of money to be transferred. The *Bound time* is a time, before which the payment could not be carried. The last parameter is *Expiration time* which determines the instant, after which the payment could be carried out using simplified procedure. The *Expiration time* has been introduced to allow to a payer to gain back its money if the service was not provided. While the *Value* parameter is mandatory, the both *Bound time* and *Expiration time* are optional. If not specified, they are considered to be passed, i.e. the corresponding condition is always satisfied.

Besides the *Main Authorization*, and arbitrary number of another *Authorizations* containing bit strings could be submitted along with the order of money transfer. Payer have to specify, which subset of these *Authorizations* have to be provided by taker to obtain payment. Generally another subset of *Authorizations* will be sufficient to obtain payment after the *Expiration time* will have been passed.

An important type of Authorizations are *Request seed* that are stored by bank.

Nor the payment order itself, nor the *Authorizations* contains any information about the taker. Anybody wishing to demand any payment could ask bank to provide him with *Request seeds* associated with the payment. If the *Bound time* when the payment had passed, the bank satisfies the request. The requester then could use Request seeds to compute related *Authorizations*. If he succeeds, bank transfers the money in accordance to directions supplied along with the request.

This way we can assure payments that have a very important feature: the money are deposited at one instant and transferred to taker only after specified delay and only in the case that taker is at some defined state.

Note that *pp* does not specify the way, how the *Authorizations* are computed, nor says, how to check the integrity and consistency of any set of *Authorizations*. It only assumes, that such mechanism, based on available data, does exist. The checking, whether a taker had produced correct *Authorizations* is simple — bank verifies that provided *Authorizations* are exactly the same with the copies specified by payer and submitted along with payment order.

7.2.3 *pp* description

As in some other places over the *Service* design, the *Payments Protocol* is more likely a framework than a detailed description of which exactly data bits are to be send ordered by which scheme. The protocol does not rely on any specific format of the data that are to be exchanged between the bank and the client in order to agree appropriate action (i.e. money transfer or so). The protocol only states, which information have to be exchanged at least, without any claims about its contingent representation.

As we stated above, with each payment could be associated an arbitrary number of *Authorizations* of various types that specifies, under which conditions should be performed some defined actions. The *Authorizations* are used to control flow of money. The basic principle is very simple: both taker and payer constructs arbitrary number of *Authorizations*. The exact number depends on situation. They setups a subset of these *Authorizations* that constitutes the proof that requester is an authorized taker of the relevant payment. When anybody wishes to obtain payment, it have to construct all the *Authorizations* belonging to the subset just discussed. If succeeds, the payment is carried out, if fails, nothing happen.

Let us describe the *Authorizations* before we will continue with description of *pp*.

The exact form of the *Authorization* is as follows:

$$[Type|Ret\_tp|Prot\_id|Auth\_data]_{\mathsf{Kpbl}_{\mathcal{B}}}$$

Here *Type* specifies type of *Authorization*. Following types are defined yet:

- `t_seed` — *Request seed* — taker uses it for computation of the proof that it is a proper recipient of the associated payment.
- `t_prf` — resulting proof — necessary to prove that the principal that had submitted this proof is a proper taker of the payment. To receive payment, taker have to construct complete set of this type *Authorizations* associated with the payment.
- `t_main` — *Main Authorization* specifying important parameters of the payment - the *Bound time*, the *Expiration time* and total amount of money.
- `t_eseed` — seed used after *Expiration time* for simplified procedure, similarly to *Request seed*
- `t_eprf` — proof of simplified procedure after *Expiration time*, similarly to resulting proof.
- `t_bstr` — byte string - not interpreted by *pp*

Each individual *Authorization*, except the *Main Authorization* (i.e. `t_main` type), could be of several types simultaneously.

The *Ret_tp* field describes to who will be send corresponding *Acknowledged Authorization* (see bellow). concatenatedIt could be the payer (`rt_pyr`) or the taker (`rt_tkr`) or both of them.

The field *Prot_id* contains identification of subsidiary procedure that will be used for checking of the consistency of the set of *Authorizations*.

The last part — *Auth_data* contains arbitrary byte string. This string is not interpreted by *pp*. The only exception from this rule is *Main Authorization*, where tis field contains following information:

$Bound|Exp|Value$

where the individual parts are the *Bound time*, the *Expiration time* and the amount of money respectively. The only operation that *pp* performs with *Auth_data* part of other *Authorizations* is the comparing of stored `t_prf` type authorization with the received on during the request from taker to transfer money.

The Acknowledged Authorization contains digital signature of original *Auth_data* string with the identification assigned to the transaction by the taker, payer and bank respectively:

$\langle Auth\_data|T\_id|P\_id|B\_id\rangle_{\mathsf{Kprvs}_{\mathcal{B}}}$

The independent processing of individual Authorizations is very important because it allows that some of its are prepared by potential taker of the payment and passed to payer encrypted with public key of the bank, so that the payer can not intercept it anyway. After receiving of such Authorization acknowledged the payer could pass it along with the acknowledged *Main Authorization* to the would-be taker (or more conservatively provider of the requested service)and this way prove that he actually had payed the correct amount of money.

We are at the point of starting the detailed explanation of *pp*.

At the very first step we have to determine the price of requested service and the conditions upon which the payment will be carried. The conditions could contains for example identification of the bank that have to be used to perform money transfer, maturity time, etc. The situation becomes a bit more complicated when the payment is not done at once, but in accordance to specified installment plan.

In such case the payer have to inquire the total number of installments and detailed description of conditions of each installment. These informations specifies taker (usually the service provider) in some appropriate form.

When the installment plan is agreed a set of *Authorizations* have to be associated with each installment. These *Authorization* could specify both the payer and the taker. All *Authorizations* are transferred encrypted (or more precisely sealed) with the public key of the relevant bank, so that the other party can not see them.

Taker sends *Authorizations* produced by himself to payer. Payer adds its own *Authorizations* and all ones sends to bank. Along with it the payer sends appropriate payment order(s) to bank, or by some other means assures that necessary finance is available.

The bank first of all decrypts all received authorizations and determines the iacknowledgmentsdentification of auxiliary protocol specified in them. This number identified procedure that will be used for consistency checking of the set of *Authorizations.* As we stated above, specification of this procedure is not a part of the *pp*.

If everything is O.K., the bank transfers money from payer's account to *Purse Account*, stores all the necessary Authorizations (`t_seed` and `t_dgst` type) and returns to the payer appropriate *Acknowledged Authorizations* — i.e. those which corresponding *Authorization* had *rt_type*value at least one of `rt_pyr` or `rt_tkr`.

From this instant the money have to be controlled by bank and the bank itself is committed to pay them to anybody who will satisfy the conditions specified by the set of relevant *Authorizations.* Payer can check the bank response and check whether the bank had taken all requested actions correctly.

If his request to the bank succeeds, the payer sends to taker all the of *Authorizations* having the *rt_type* value `rt_tkr`. The taker will check received *Acknowledged Authorizations* and if they are correct, he starts to provide requested service. This way the taker is convinced that he will receive his fees for provided service without any complication despite the actual state of the payer.

When the taker wishes to receive some payment first of all it checks whether it is able to construct necessary Authorizations directly or if it needs some Request seed(s). In the case the seeds are necessary it requests bank to provide it. The bank checks whether the Bound time had passed. If so, the bank provides relevant seeds.

The taker then will try to construct all type `t_prf` *Authorizations* and using them to form the final payment request that it sends back to the bank. The bank checks whether all received *Authorizations* are correct. If so, it carries out the payment.

If the payer realizes that the taker probably had failed in receiving the payment or even did not requested it, it will can, after the *Expiration time* will have been passed, request the bank to return the money back. If necessary, it will request type `t_eseed` *Authorizations* from the bank. In the case he will construct all type `t_eprf` *Authorizations* successfully, bank returns the money. As in the case of regular carrying of the payment to the taker, anybody can request returning of the money and bank returns it to whoever who will correctly construct the proof.

Here is the overall structure of the protocol. Let $\mathcal{C}$ (client) be a payer, $\mathcal{E}$ (*ES*) be a taker and let $\mathcal{B}$ be an involved bank. Suppose, that the installment plan consists of k installments. Let $\mathbf{c}$ be a set of *Authorizations* constructed by payer $\mathcal{C}$, and $\mathbf{e}$ be *Authorizations* prepared by taker $\mathcal{E}$.

1. $\mathcal{C} \longrightarrow \mathcal{E}: \quad service\_request; T\_id$
2. $\mathcal{E} \longrightarrow \mathcal{C}:$
$$\bigvee_{j=1}^{k} (Value;Bound;Exp; \bigvee_{i \in \mathbf{e}} ([Auth_i]_{\mathsf{Kpbl}_{\mathcal{B}}}))$$
3. $\mathcal{C} \longrightarrow \mathcal{B}: \quad \bigvee_{j=1}^{k} (\bigvee_{i \in \mathbf{e} \cup \mathbf{c}} ([Auth_i]_{\mathsf{Kpbl}_{\mathcal{B}}}))$
4. $\mathcal{B} \longrightarrow \mathcal{C}:$
$$\bigvee_{j=1}^{k} (\bigvee_{\substack{i \in \mathbf{e} \cup \mathbf{c} \\ type \in \mathbf{r}}} ((\langle Auth_i | T\_id | S\_id | B\_id \rangle_{\mathsf{Kprvs}_{\mathcal{B}}}); B\_id)$$
5. $\mathcal{B} \longrightarrow \mathcal{C}:$
$$\bigvee_{j=1}^{k} (\bigvee_{\substack{i \in \mathbf{e} \cup \mathbf{c} \\ type \in \mathbf{s}}} ((\langle Auth_i | T\_id | S\_id | B\_id \rangle_{\mathsf{Kprvs}_{\mathcal{B}}}); B\_id)$$
6. $\mathcal{E} \longrightarrow \mathcal{B}: \quad B\_id$
7. $\mathcal{B} \longrightarrow \mathcal{E}: \quad \bigvee_{\substack{i \in \mathbf{e} \cup \mathbf{c} \\ type \in \mathbf{q}}} (Auth_i)$
8. $\mathcal{B} \longrightarrow \mathcal{E}: \quad \bigvee_{\substack{i \in \mathbf{e} \cup \mathbf{c} \\ type \in \mathbf{w}}} (Auth_i)$

Note that $\mathbf{r} = \{\mathtt{rt\_pyr}, \mathtt{rt\_tkr}\}$, $\mathbf{s} = \{\mathtt{rt\_tkr}\}$,$\mathbf{q} = \{\mathtt{t\_seed}\}$ and finally $\mathbf{w} = \{\mathtt{t\_dgst}\}$. In all steps following the step 3. of the protocol only a digital signatures are signed to minimize communication. We assume that all parties are able to store necessary data to be able perform checks needed to be sure that nothing goes wrong. We omitted indexing of entries of messages with outermost index j which identifies individual installments. We assume that any confusion is unlikely, but simplification of record worth this lack of strictness.

In the step 3. of the protocol bank performs external procedure to check consistency of received set of *Authorizations* if necessary. Only if no problems occur the bank continues with protocol. The money transfer order to *Pursue Account* or any similar action should be performed by payer within this step too. The actual arrangement of this action will probably depend on the bank and will vary from case to case so we will not write it as a part of protocol.

When payer (or anybody else) wish to request money refund, it will perform steps $6' - 8'$ that are similar to steps $6 - 8$, only the type of used *Authorizations* changes from `t_seed` and `t_prf` to `t_eseed` and `t_eprf` respectively.

Bank will check whether the *Bound time* (*Expiration time*) will have been passed before it will perform step 7 ($7'$) of the protocol.

7.3 Data Care Proof Protocol

7.3.1 Scope of *dcp* utilization

The Data Care Proof Protocol is designed solely as a solution of a very specific problem within the *Service*. It is used to construct Authorizations necessary by *pp* for carrying of the payments from clients to Eternity servers for data storage.

We will start with description of overall protocol idea and functionality. Later we will show the interoperation of both protocols in the last part of this section.

7.3.2 Characteristics of *dcp*

There are some interesting problems to be overcome by protocol. We have to make an *ES* operator to keep data the whole agreed period. It clearly means that checks, whether the server really holds our data, have to be performed after this period is passed. The next problem arises from the fact that it is likely that the same data will be stored also at some others servers, thus we must prevent server operator from performing the proof based on the data acquired from any other server. Another possibility for server operator to cheat is to try to reduce the size of the data in such way that it will be still able using the reduced data, and some other copy of the data eventually, to construct legal Authorization. At the some time we have to avoid the possibility that anybody, even the client who had submitted the data will masquerade the server and successfully constructs the Authorization and this way will steal fees belonging to server.

While the last mentioned problem could be easily addressed by *pp* the other ones have to be solved within the *dcp*.

7.3.3 Description of *dcp*

It is clear that individual copies of the data have to be different. This prevents the server from "borrowing" the data elsewhere and use it for a proof of data ownership. There should not be any feasible way how to transform one copy of the data into any other using a small amount of additional information only. On the other hand it should be simple to recover original data from any copy.

The solution is surprisingly simple. At the beginning of each copy of the data we place a (different) random string. A copy is then encrypted with private key of public-key algorithm. Corresponding public key is concatenated to resulting data and the private key is discarded. Thus anybody can remove the private-key encryption, but nobody will be able to construct later the copy of data from any other. The exact form of copy of data – the *Unique copy (UQC)* – is as follows:

$$\mathsf{Kpbl}|\{nonce|data\}_{Kprv}$$

Note that it is crucial to use public key encryption directly, not the sealing. The public key operations will be done solely at the client's side. So the computation overhead caused by the need of use of a public key cryptography is allowable. We should consider that the private key operations will be performed exactly once in opposite to public key operations that will be performed within each later access of the data. This is why a used public key algorithm should be selected carefully so that a favorable distribution of a computation difficulty could be achieved. It is advisable to make public key computations as simple as possible to the detriment of difficulty of client's operations. In other words it is highly desirable that public key operations will be as fast as possible.

Obviously the proof that server still holds the stored data have to be based on the precise knowledge of the whole data. The simplest way how to arrange things is to construct the proof as a digest of the data. In order to prevent server from pre-computing the digest a randomly selected nonce *seed* have to be connected before the *UQC* itself before the digest is computed. The *seed* provides the involved bank on demand. [2]

When the server wishes to obtain payment for provided service, it requests bank to provide the *seed*. Using the *seed* and some arbitrary digest algorithm it computes a *digest* which will serve as a proof that it still holds the data. The bank checks, whether this *digest* matches with the one associated with requested payment. If so, the bank carries out the payment in accordance to specified propositions.

The arrangement just described has all necessary functionality and is sufficient in the simplest case when we suppose that payments can not be refunded to a payer. This assumption however does not seems to be very realistic. It is likely that servers will accidentally lose stored data or will become unreachable, damaged etc. In such cases we assume to be fair to allow to a client to gain back its money. If we allow that payments are refunded, we have to defend servers against clients which will try to construct *digest* and *seed* pair so that server will not be able to prove that it holds the data.

To reduce the risk that client will cheat and will construct the *digest* so that the server will not be able to reconstruct it later, the following procedure is performed. We will use n different digests simultaneously. Increasing the number of used digests allows us to achieve desired level of certainty that client does not cheat. This number should be selected carefully because increasing the number will cause growth of the amount of the corresponding computation and communication.

The $dgst_i$ is computed as a result of the digest function applied to unique copy of a data to which the $seed_i$ is concatenated as a prefix. The $dgst_i$ together with corresponding $seed_i$ is encrypted with some symmetric key ks_i selected at random. The resulting structure we will denote c_auth_i. Why such strange arrangement?

Server receives all the structures c_auth_i during agreement process. It responses with a list of indexes of the structures that it wishes to decrypt. In the later stage of the protocol client will provide corresponding keys ks_j to the server. In fact the server will request all keys except only a one key. The missing key should be selected (pseudo)randomly. Because the client does not know which keys will the server request, it has to prepare all structures correctly. The client has only a little opportunity (exactly $1/n$) that it can successfully foist a bogus structure and thus make impossible for the server to obtain fees.

After receiving the keys and the corresponding UQC the server will be able to verify that decrypted pairs $seed_i$ and $digest_i$ are correct. In the case of any doubts it can stop performing the protocol and refuse to accept and store data.

The exact form of the structure c_auth_i looks as follows:

$$\{T\_id|seed_i|\overbrace{\|seed_i|UQC\|}^{dgst_i}\}_{\mathsf{ks}_i}$$

Obviously i is any value between 1 and n.

The Data Care Proof Protocol works in two phases. The first phase incorporates construction of all c_auth-s, signing of its by a bank and verifying by a server. This part is conducted during the process of agreement between a client and a server on data deposition and an actual transfer of data to server. During the second part server reconstructs both *digest*-s. This part is performed within the payment protocol between a bank and a server when a server wishes to obtain payment.

Here are the main steps of the protocol. Let $\mathcal{E}$ be an *Eternity server*, $\mathcal{C}$ be a client and $\mathcal{B}$ be an involved bank. The protocol is, for better compatibility with the description of the *pp* in section 7.2.3, explained

[2] As we will see later the *seed* will not be provided before the *bound time* has been passed.

for the case, that payment will be divided into t installments. The server requires n digests for each installment. *List* contains list (probably of $n-1$ randomly selected) keys ks_i that server wishes to obtain for checking the *c_auth*-s.

$$1.\quad \mathcal{C}\longrightarrow\mathcal{E}: \quad \bigvee_{l=1}^{t}(T\_id;\ \bigvee_{i=1}^{n}(\overbrace{\{T\_id|seed_i|dgst_i\}_{\mathsf{ks}_i}}^{c\_auth_i}))$$

$$2.\quad \mathcal{E}\longrightarrow\mathcal{C}: \quad \bigvee_{l=1}^{s}(list) \text{ where } s \leq t$$

$$3.\quad \mathcal{C}\longrightarrow\mathcal{B}: \quad \bigvee_{l=1}^{s}(\bigvee_{i=1}^{n}(c\_auth_i);\mathsf{ks}_i;seed_i;dgst_i)$$

$$4.\quad \mathcal{B}\longrightarrow\mathcal{C}: \quad \bigvee_{l=1}^{s}(\bigvee_{i=1}^{n}(\langle c\_auth_i\rangle_{\mathsf{Kprvs}_{\mathcal{B}}}))$$

$$5.\quad \mathcal{C}\longrightarrow\mathcal{E}: \quad \bigvee_{l=1}^{s}(\bigvee_{i=1}^{n}(\langle c\_auth_i\rangle_{\mathsf{Kprvs}_{\mathcal{B}}});\ \bigvee_{j\in list}(\mathsf{ks}_j))$$

Note that the server could specify different bank to be used for each installment. The steps 3. and 4. of the protocol will be performed several times in such case. Adapting protocol and behavior of the client for this arrangement is straightforward, so we have omitted it just for sake of simplicity.

The server could also in step 2. reduce the predetermined by client number of installments.

We assume that the number of installments is either determined by some other means either it is a well known parameter of the protocol. So the protocol does not take any steps to determine it. The protocol even does not use this number, actions taken over different installments are totally independent.

Bank in the step 3. of the protocol have to check the consistency of received data. Due to this purpose it decrypts all c_auth_i and verifies that $seed_i$ and $dgst_i$ received encrypted within the c_a-s matches to those received in plaintext. Only if everything is O.K. the bank continues carrying the protocol.

In step 5. of the protocol server will check validity of received digital signatures of every c_auth_i. It is necessary to verify, that received set of keys ks_j corresponds with the *list* as had been specified in the step 2.

After receiving the *UQC* a server will decrypt all the c_auth_i to which it posses the key ks_j and verifies $dgst_i$. If and only if all digests matches the data and the relevant *seed*, server stores the data. In the other case it discards all the data associated with the transaction and quits the protocol.

When a server detects that the *Bound time* had passed it continues processing the protocol:

$$6.\quad \mathcal{E}\longleftarrow\mathcal{B}: \quad seed_1|seed_2$$

$$7.\quad \mathcal{E}\longrightarrow\mathcal{B}: \quad dgst_1|dgst_2$$

In the step 6. the bank provides the *seed*-s if and only if the *bound time* had actually passed. The second part of the protocol (i.e. steps 6. and 7.) are mentioned just for the sake of complexness and better understanding of the protocol. In fact these steps will be done within performing of the *Payments Protocol*. The matter of *dcp* consists in the producing and checking of *c_auth*-s in steps 1. to 5.

In the case that it is inconvenient or impossible to send c_auth_i in within the first request to server a similar features could be achieved using a slightly more complicated protocol (see **??**).

7.3.4 Cooperation of *pp* and *dcp*

The message transport platform of the *Service* — the *erm* — is relatively demanding both from the point of view of a computation and communication overhead. This is why we tempt to keep the lowest possible overall amount of communication involved by any task to be performed. The other our goal is to minimize the total number of messages that have to be exchanged between the communication parties. The reason of this is easy to understand. The higher number of messages represents a better possibility for an attacker to investigate who is communicating with who. So it worth our attention to map all steps of simultaneously used protocols to as few messages, as possible.

Now we will discuss the cooperation of the *pp* and the *dcp* in more details. We will start with a precise description. For better understanding we will write different types *Authorizations* separately. Let $\mathcal{E}$ be an *Eternity server*, $\mathcal{C}$ be a client and $\mathcal{B}$ be a bank. Again we will simplify he protocol slightly assuming that only one bank will be involved. We also will omit explicit writing of *Type*, *Ret-tp* and *Prot* fields of individual *Authorizations* to avoid lengthly record. The type and return type of individual *Authorizations* is evident from the protocol. The *Prot* is always the same and corresponds with the identification of the *dcp*.

Client determines the number of installments from *Access certificate* of the server (see 7.1). Then it creates sets of *c_auth*-s for each installment. Client sends these sets along with specification of the service required (i.e. size of the data, period of storage) to the server.

Server decides whether it satisfies the request or no. If so, it computes the price of storage, creates installment plan, selects the keys used for encryption of individual *c_auth*-s that it wishes to decrypt and all this information sends back to the client. Server also sends to client for each installment its *Authorization SA*. The *SA* prevents the client from stealing the fees later, because this *Authorization* will not be able to construct and thus will no be able to submit complete set of *Authorizations* as required by *pp*. Obviously *SA* is of type `t_prf` and *Ret_tp* is set to `rt_tkr`.

If client complies with the installment plan, it constructs for each installment the *Main Authorization* using the received information and creates sets of own *Authorizations*. The *Authorizations* prepared by client are created simply from the *c_auth*-s (that had been passed to server in step 1.). Client takes steps to transfer moneys as needed to *Pursue Account* and sends to bank all *Authorizations*.

Bank first of all checks integrity of each set of *Authorizations* corresponding to individual installments. This check is done easily using following procedure. Bank decrypts every c_auth_i with the key ks_i and checks, whether the digest and the seed inside the c_auth_i matches with the pair ($seed_i$;$dgst_i$) received at the same time. The c_auth_i and the keys ks_i are of type `t_bstr`, the seeds ($seed_i$ are of type `t_seed` and the digests $dgst_i$) are of the type `t_prf`. All the c_auth_i have *Ret_tp* set to `rt_tkr`⊕`rt_pyr`.

Client then re-sends to server all acknowledged c_auth_i, the acknowledged *Main Authorization*, according to *list* requested keys ks_i and, of course, the *Unique Copy* of the data belonging to that server.

Server checks all acknowledgment, decrypts all the c_auth_i that it can and checks the validity of digests. If no problems are encountered, ti takes steps necessary to properly store data.

When the maturity time (*Bound time*) of a installment will have been passed, the server will ask bank to provide him with corresponding seeds. Then server will compute digests and using them it will obtain the fees.

1. $\mathcal{C} \longrightarrow \mathcal{E}$: $\mathit{Time};\mathit{Size};\bigvee_{l=1}^{t}(T\_id;\bigvee_{i=1}^{n}(a\_c_i))$

2. $\mathcal{E} \longrightarrow \mathcal{C}$: $\begin{cases} \mathit{Refuse} \\ \bigvee_{l=1}^{t}(T\_id;S\_id;list;\overbrace{(Bank_l;Price_l;Bound_l;Exp_l)}^{\text{installment plan}};\overbrace{\{S\_id|SA\}_{\mathsf{Kpbl}_{\mathcal{B}}}}^{\text{sealed } SA}) \end{cases}$

3. $\mathcal{C} \longrightarrow \mathcal{B}$: $\bigvee_{l=1}^{s}([Value|Bound|Exp]_{\mathsf{Kpbl}_{\mathcal{B}}};\bigvee_{i=1}^{n}([c\_auth_i]_{\mathsf{Kpbl}_{\mathcal{B}}});[SA]_{\mathsf{Kpbl}_{\mathcal{B}}};\bigvee_{i=1}^{n}([dgst_i]_{\mathsf{Kpbl}_{\mathcal{B}}};[seed_i]_{\mathsf{Kpbl}_{\mathcal{B}}});S\_id;T\_id)$

4. $\mathcal{B} \longrightarrow \mathcal{C}$:
$\bigvee_{l=1}^{s}(\langle Value|Bound|Exp|T\_id|S\_id|B\_id\rangle_{\mathsf{Kprvs}_{\mathcal{B}}};\bigvee_{i=1}^{n}(\langle c\_auth_i|T\_id|S\_id|B\_id\rangle_{\mathsf{Kprvs}_{\mathcal{B}}});\langle SA|T\_id|S\_id|B\_id\rangle_{\mathsf{Kprvs}_{\mathcal{B}}})$

5. $\mathcal{C} \longrightarrow \mathcal{E}$:
$\bigvee_{l=1}^{s}(\langle Value|Bound|Exp|T\_id|S\_id|B\_id\rangle_{\mathsf{Kprvs}_{\mathcal{B}}};\bigvee_{i=1}^{n}(\langle c\_auth_i|T\_id|S\_id|B\_id\rangle_{\mathsf{Kprvs}_{\mathcal{B}}});\langle SA|T\_id|S\_id|B\_id\rangle_{\mathsf{Kprvs}_{\mathcal{B}}});\wr$
$\wr\bigvee_{l=1}^{s}(\bigvee_{i\in list}(\mathsf{ks}_i));UQC$

6. $\mathcal{E} \longrightarrow \mathcal{B}$: B_id

7. $\mathcal{B} \longrightarrow \mathcal{E}$: $B\_id;\bigvee_{i=1}^{n}(seed_i)$

8. $\mathcal{E} \longrightarrow \mathcal{B}$: $B\_id;\bigvee_{i=1}^{n}(dgst_i)$

9. $\mathcal{B} \longrightarrow \mathcal{E}$: $B\_id;\begin{cases} \mathit{Success} \\ \mathit{Fault} \end{cases}$

This protocol the client performs for each *Unique copy* of the data.

7.4 Auxiliary protocols

Here we will discuss two additional protocols shortly. Both of them are not as complex protocols as the previous are. The reason of a placement of this remark here is that this way the discussions of every data transfer protocols within the *Service* will be more comprehensive. We will summarize the features only that we are necessary to be fulfilled by mechanism actually used to accomplish relevant task.

7.4.1 Obtaining certificates from *ACS*

Anybody wishing to obtain anything from the *Service* will need many various certificates.

It is necessary that in any case the whole communication is encrypted. The certificates have to be received from more than one source. The higher the number is, the better. We recommend to use four sources at least. From each source should be taken several times more certificates than will be actually used.

From acquired certificates the needed amount should be selected at random and the superfluous certificates will be discarded.

The reason of this arrangement is evident. An ordinary key used by contemporary enciphering algorithm ranges from several tens (for conventional symmetric algorithms) to several hundreds (mostly public key algorithms) bits. It means that any key is almost surely unique within the whole system and probably within the whole world. Together with knowledge who had received which certificates (and thus which keys in the case of *MX* certificate) it could simplify tracing of the communication. We have to preclude *ACS* and their conductors from such behavior.

7.4.2 Time synchronization

As has been mentioned in 5.2.4 we assume that exact time necessary for proper function of majority of servers will be acquired from outer sources. We assume that servers operators will use reliable sources of their choice. There are not any specific features that the actually used protocol will have to fulfill except the following one. It seems reasonable to request that used protocol will assure data integrity and authentication of communicating parties, or the server providing time information at least.

A Related cryptography

A.1 Crisis of the trust

The lack of identity brings a very serious problem that have to be taken into account. As had been explained in section 5 and 6, the hiding of *ES*-s is very important for the overall resistance of the whole *Service*. On the other hand sometimes we need that servers will sign some message or so. The problem lays in the fact, that it will be very difficult to implement any method, how the servers will obtain some reliable certificates of their public keys.

The server operator can not use regular way, i.e. to visit personally some Certification Authority (CA) and obtain certificate there. Unlike banks, the CA-s are relatively small organizations that are poorly defended by legal means today. So the fact, that the CA which had created the certificate of some server obviously knows identity of its administrator represents obvious exposition of the system.

This problem, unfortunately, arises despite which existing model of trust we will use — either the X.509 recommendation certificates, either the chains of trust introduced by PGP.

Probably a new model of trust have to be introduced for such purpose. The trust will be dependent on some digital pseudonym rather than on any physical identity.

We will try to sketch one possible solution very roughly.

Each server will create its digital pseudonym. This pseudonym should be long enough so that it will be highly probable that created pseudonym is unique within the whole system. Server also creates needed pair of public and private keys. Then it contacts several CA-s and request them to provide certificate of the public key. Along with the request serve submits its digital pseudonym. The CA checks, whether it had created certificate for such digital pseudonym in some previous period. If not the CA creates certificate and records it. It also creates a ticket (arbitrary random string) that sends to requester along with the certificate. When the CA realizes that it had created the certificate for such pseudonym yet, she will not fulfill the request if the appropriate ticker was not submitted along with it.

This procedure should be performed with several CA-s. When the new certificate is created, the owner of the pseudonym (and the certificate) have to be certified successfully by specified number of previously used CA-s to achieve a valid certificate. When some CA is unavailable, requester has to compensate it with another one.

A.2 Extension of the Oblivious Transfer Protocol

Here we will briefly discuss a simple extension of original Oblivious Transfer Protocol [3]. The protocol allows to communicating parties to obtain the contents of exactly one message from the two (or more generally any arbitrary number) ones sended. The selection of a message is done so that the sender can not influence it neither it knows which particularly message the receiver had obtained.

Our extension allows to the receiver to obtain all but one message from all messages than the sender issues. Similarly to original version of the protocol, the sender can not influence the selection neither knows, which message the receiver lacks.

Let n is the total number of messages. The sender creates n packets. The i-th packet contains the same n-tuple of (decryption) keys. In each packet the i-th key is replaced with random nonce and the whole n-tuple is encrypted with i-th key taken from the first n keys. The structure is labeled with corresponding number which must be the same as the index of the missing key. All n structures are then transferred to the receiver. Using the original Oblivious Transfer Protocol the receiver obtains exactly one key from the first n-tuple and thus is able to decrypt one of the received structures. On the decrypted structure the receiver verifies that the missing key index matches with the label of the packet. This way it gains all but one key from the second n-tuple.

Due to features of Oblivious transfer the sender does not know which key from the second n-tuple the receiver had obtained and thus does not know which key from the first n-tuple it lacks.

Then the extended protocol looks as follows:

1. $\mathcal{A} \longrightarrow \mathcal{B}$: kpbl_i
2. $\mathcal{B} \longrightarrow \mathcal{A}$: $\{\mathsf{ks}\}_{\mathsf{kpbl}_x}$
3. $\mathcal{A} \longrightarrow \mathcal{B}$: $i|\{\mathsf{k}_1| \ldots |\mathsf{k}_{i-1}|nonce|\mathsf{k}_{i+1}| \ldots \mathsf{k}_n\}_{\mathsf{ks}_i}$
   where $\mathsf{ks}_i = \{\mathsf{ks}\}^{-1}_{\mathsf{kprv}_i}$
4. $\mathcal{A} \longrightarrow \mathcal{B}$: $\{msg_i\}_{\mathsf{k}_i}$

Here $i \in \{1, \ldots, n\}$ and x is chosen at random from the same range.

If we change slightly the process of construction structures in step 3. of the protocol, we can transfer any subset of all messages chosen at random. In such case we have to introduce appropriate enumeration of subsets. The i-th structure then contains only the keys belonging to i-th subset. The rest of the protocol stays unchanged.

♡

References

[1] Ross J. Anderson. The eternity service. In *Pragocrypt 1996*, 1996. ' http://www.cl.cam.ac.uk/rja/eternity.html'.

[2] Andre Bacard. Anonymous remailer faq, 1998. http://www.well.com/user/abacard/remail.html.

[3] M. Blum. Coin flipping by telephone:a protocol for solving impossible problems. In *Proceedings of the 24th IEEE Computer Conference (CompCon)*, pages 133–137, 1982.

[4] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, Feb 1981.

[5] D. Chaum. The dinning cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(11):65–75, 1988.

[6] D. Eastlake. Randomness recommendations for security. RFC 1750, Dec 1994.

[7] David M. Goldschlag et. al. Hiding routing informatio. In *Workshop on Information Hiding*, Cambridge, UK, May 1996.

[8] V. Klíma. Naslouchá přítel, nebo snad nepřítel. *Decros news*, (7):10–11, Jul 1998.

[9] M. Shub L. Blum, M.Blum. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, 15(2):364–383, 1986.

[10] R. M. Needham. Denial of service: an example. *Communications of the ACM*, 37(11):42–46, Nov 94.

[11] L. Adleman R. Rivest, A. Shamir. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb 1978.

[12] R. Rivest. The md5 message digest algorithm. RFC 1321, Apr 1992.

[13] B. Schneier. One-way hash functions. *Dr. Dobbs Journal*, 16(9):148–151, Sep 1991.

[14] M. E. Hellman W. Diffie. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, Nov 1976.