

# Charme: A distributed social network with end-to-end encryption and contextual information

Manuel Schultheiß  
manuel.schultheiss@tum.de

December 19, 2015

**Abstract.** Decentralized social networks with end-to-end encryption would take back control of data to the end-users. In this paper Charme is presented as such a network. First, some basic cryptographic principles are explained before focusing on the underlying network protocol.

## Contents

<b>1</b>	<b>Prologue</b>	<b>3</b>
<b>2</b>	<b>Introduction</b>	<b>4</b>
<b>3</b>	<b>Encryption and Security</b>	<b>5</b>
3.1	Existing work . . . . .	5
3.2	Formal Notation used in this paper . . . . .	5
3.2.1	Reach . . . . .	5
3.3	Fundamental Concepts . . . . .	5
3.3.1	AES-CBC . . . . .	5
3.3.2	RSA . . . . .	6
3.3.3	Further reading . . . . .	7
3.4	Crypto Implementation . . . . .	7
<b>4</b>	<b>Implementation</b>	<b>8</b>
4.1	Backend Technology Stack . . . . .	8
4.2	Client server seperation . . . . .	8
4.3	Passphrase . . . . .	8
4.4	Verifying private keys . . . . .	9
4.4.1	Generating Fingerprints . . . . .	9

4.5	Preventing List and Message MITM Attacks . . . . .	9
4.6	Post encryption . . . . .	10
4.7	Message encryption . . . . .	10
4.8	Distributing private information . . . . .	11
4.9	Key Updates . . . . .	12
4.10	Primary Attack Vectors . . . . .	12
4.10.1	MITM on Client Software . . . . .	12
4.10.2	Attacks on Integrity Protection . . . . .	12
4.10.3	New Conversations with malicious userId/Name pair . . . . .	13
4.10.4	XSS . . . . .	13
4.10.5	Session Capturing . . . . .	13
4.10.6	DOS . . . . .	14
4.10.7	Replay attacks . . . . .	14
4.10.8	Malformed data . . . . .	14
4.10.9	Logging of Requests . . . . .	14
4.10.10	Spam . . . . .	14
<b>5</b>	<b>Protocol Specification</b>	<b>15</b>
5.1	Login . . . . .	15
5.2	Search . . . . .	15
5.3	Messages . . . . .	15
5.4	Edgekeys . . . . .	15
5.5	Recryption of data . . . . .	15
5.5.1	Signup . . . . .	15
5.5.2	Add public key to keydirectory . . . . .	16
5.5.3	Init Conversation . . . . .	16
5.5.4	Add People to Conversation . . . . .	18
5.5.5	Write message . . . . .	18
<b>6</b>	<b>Future Work</b>	<b>18</b>
6.1	Token Generation and API . . . . .	18
6.2	Use of encrypted Databases . . . . .	19
6.3	Location based Chats . . . . .	19
6.4	Data Sharing for the Internet of Things . . . . .	19
<b>7</b>	<b>Terminology</b>	<b>20</b>
<b>8</b>	<b>Sources</b>	<b>22</b>

# 1 Prologue

It's personal. It's private. And it's no one's business but yours.

These are the first words, of Phil Zimmermann's essay "Why I wrote PGP" from 1991 [3]. 24 year later, electronic communication has gained more and more attention in your everyday life. Today, letters have become instant messages and half of your real world conversations have become social network posts. But while you could be almost sure a letter remained private, today a bunch of stakeholders such as governments, technicians and ad-serving companies has access to data derived from your conversations. Most people I talk to simply give a "I have nothing to hide" as an answer when addressing this point. This may be true, as long as you are not a politician, your are not responsible for a business unit, or do not have anything else to say other people are willing to listen to. Otherwise information about you is always equal to power against you. While your self forgets most of the things you said, it is probably saved in some data center waiting to get parsed by some algorithms. With more and more data you become calculable, your actions can be predicted long before you perform them.

Ads have become the number one revenue model of internet companies. Politicians and trying to expand mass surveillance and ban encryption whenever it is possible [2]. With the decay of classic media, such as newspaper and television, social networks are the primary infrastructure of directing people to information now. The heterogeneous landscape of newspapers and blogs providing different information is slowly fading. Your newsfeed has become the primary navigation source of information. And should not trust a single institution here, because who controls what you read controls who you become.

## 2 Introduction

In contrast to classic social networks, distributed social networks with end-to-end encryption can provide self control of your data and independence from big companies. *Distributed* means, that there is no central server hosting your data, but many small servers all around the world, which everyone can setup. Just like email, every participant has a unique address like *yourname@yourserver.com*. With end-to-end encryption not even the server hosting your data can see your data. This is necessary for a distributed architecture as otherwise the server admin could read your messages for example.

Another innovation of Charme is the so called *Context*, every post can have optionally. A context contains semantic, computer-readable information about the post. An algorithm can later aggregate all users moving by car from point A to point B tomorrow, with having 3 seats free with a tolerance of 20 kilometers for example. Or you can look up for friends making pizza for lunch and want to share their food today. Figure 7 shows the creation of a context and the creation of a filter to search for existing context of other users. Websites already providing share-economy services can also add their data to the Charme network, using the Charme Schema JSON Syntax, as defined in the file *schema.coffee*. The context system introduces elements of the *Web of Data*, formerly known as the semantic web to social networks.

The figure consists of two side-by-side dialog boxes. The left dialog, titled 'Filters help you to find relevant information in your stream. You can show only Events from your friends in a 10km radius for example.', contains a 'Filter Name' input field, a checkbox for 'Only people from my lists', a list of context types with radio buttons (Move from A to B, Offer, Service, Meal, Activity, Review, Event), a location selector (Munich) with an 'Add Location' button, a radius selector (1km), and fields for 'Destination' and 'Seats'. At the bottom are 'CANCEL' and 'ADD FILTER' buttons. The right dialog shows a grid of context icons: Move (green circle with arrow), Sell (orange circle with dollar sign), Review (red circle with star), Event (purple circle with calendar), Activity (blue circle with speech bubble), Service (blue circle with wrench), and Meal (green circle with plate). A 'CLOSE' button is at the bottom right.

Figure 1: Context dialogues. Finding existing context on the left; Adding context on the right. After a user has selected a context on the right, the user can add parameters like Location or time for the context.

## 3 Encryption and Security

### 3.1 Existing work

With Diaspora one of the first popular distributed social networks was released. The messages are not end-to-end encrypted, however, which makes it possible for the server admin to read its users messages. A more secure concept is *Twister* (<http://twister.net.co>), a Twitter like peer-to-peer micro blogging platform. Using the blockchain makes it possible to encrypt messages and even meta data (including the IP Address!). Charme does not encrypt meta data at the moment for simplicity and performance reasons, but such an encryption scheme could be implemented later on using techniques like already present in onion routing for example.

### 3.2 Formal Notation used in this paper

#### 3.2.1 Reach

We define the *reach* of a server as this: Each server  $S_i, i \in 0, 1, 2, \dots$ , on which at least one user has a public key of another user hosted on server  $T_j, j \in 0, 1, 2, \dots$  is *reached* by this server. The reach is a list of tuples  $(S_i, T_j)$ . See figure 2 as an example.

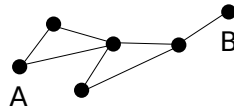


Figure 2:  $|Reach(A)| = deg(A) = 2$  as the users on server A have public keys of two different servers.

### 3.3 Fundamental Concepts

#### 3.3.1 AES-CBC

AES is a fast symmetric encryption algorithm that was standardized in 2001 and is still considered secure, although there exist some attacks on it. The following explanation is based on the lecture slides of Carle et al [1].

A key  $K_{A,B}$  is used for encryption and decryption by the two parties A and B. Cipher Block Chaining Mode (CBC) ensures that blocks that are identical in the plaintext, are not identical anymore in the ciphertext. This is essential, as otherwise images can still be recognized for example, although they are encrypted. First, an initialization vector (IV) is created:

$$c_0 = IV$$

A plaintext block  $p_i$  becomes a cipher block  $c_i$  during encryption.

$$c_i = Enc_K(c_{i-1} \oplus p_i)$$

and the same thing reversed for decryption:

$$p_i = c_{i-1} \oplus Dec_K(K, c_i)$$

It is important that the IV is shipped with integrity protection, otherwise an attacker could modify the first 256 bits in order to set the first decrypted plaintext block to a custom value. The other blocks can not be influenced however. There are several techniques to add it ( [https://en.wikipedia.org/wiki/Authenticated\\_encryption](https://en.wikipedia.org/wiki/Authenticated_encryption)). Integrity protection can be achieved using the following steps using an encrypt-then-mac Scheme:

1. Encrypt Message using AES-CBC
2. Get SHA-256 hash value of encrypted message and salt with key k and crypto algorithm version a.
3. JSON Result is:

$$\{ \text{a: 1, m: } Enc_{k||v}(text), \text{h: sha256HMac}(Enc_{k||v}(text), k||v) \}$$

The algorithm version a is added, as otherwise, an attacker could extract the plaintext and use backwards compatibility to let the decryption and MAC check be performed by an old algorithm, also there is a better algorithm available.

For example: The current implementation supports Non-HMAC AES when the information was encrypted without a HMAC. An attacker could remove the HMAC key and just send the plain AES encrypted content to the client and the client would still decrypt it correctly.

Please note that for more security, the salt for the sha256HMac should also not use k, but a derived version of k.

### 3.3.2 RSA

In RSA there are two different keys. The public key is visible to everyone, while the private key is only visible to the generator of both keys. Asymmetric encryption is a lot of slower than symmetric encryption. So usually we generate an AES key, encrypt the text with the AES key and encrypt the AES key with the RSA key, rather than encrypting the whole text with the RSA key. Technically we generate large prime numbers  $p$  and  $q$ .

$$n = p * q$$

and define

$$\Phi(n) = (p - 1) * (q - 1)$$

Let e not equal to 1, smaller than  $\Phi(n)$  and not share a common divisor with  $\Phi(n)$ . Now we compute d under the restriction:

$$e * d = 1 \pmod{\Phi(n)}$$

To encrypt a message  $M$  or decrypt cipher  $C$  use:

$$C = M^e \pmod{n}$$

$$M = C^d \pmod{n}$$

To provide *Ciphertext indistinguishability* RSA is used with a padding scheme like PKCS1. Here we generate  $n$  with more than two primes:

$$n = p_1 * p_2 * \dots * p_n$$

with  $p_1$  being  $r$  and  $p_2$  being  $q$ . [1]

### 3.3.3 Further reading

1. PKCS#1 v2.2 RSA Cryptography Standard  
<http://www.emc.com/collateral/white-papers/h11300-pkcs-1v2-2-rsa-cryptography-standard-wp.pdf>
2. Lessons learned and misconceptions regarding encryption and cryptology  
<http://security.stackexchange.com/questions/2202/lessons-learned-and-misconceptions-regarding-encryption-and-cryptology/2206#2206>
3. HMAC versus raw SHA hash functions  
<http://dev.ionious.net/2009/03/hmac-vs-raw-sha-1.html>

## 3.4 Crypto Implementation

There are a Javascript library called Gibberish AES for Javascript and Tom Wu's RSA Library for RSA providing the main functionality. Google wrote with its E2E project a better implementation of cryptographic algorithms which should replace the above ones as soon as possible. The long term goal is to use the W3C web crypto API however, if supported in all major browsers, as this one is more secure regarding key deletion from memory (RAM) and random number generation.

## 4 Implementation

### 4.1 Backend Technology Stack

On server side, PHP is used as an interpreter. The reason for PHP is that it has a wide community supporting it, as it is used by many other big websites like Facebook or Wordpress for example. Furthermore it does not depend on 100 extra packages for basic function with questionable future support like Node.js. *ZeroMQ* is used to send socket messages and communicate between different PHP threads. *Gearman* is necessary to execute background tasks like sending messages in the asynchronously. As a database, *MongoDB* is used, as it is scalable and has the ability to directly store JSON object to the database. Administrations scripts are written for GNU Bash to provide easy ssh access.

### 4.2 Client server seperation

In contrast to classic social networks, we seperate client and server in two seperate projects first. This is necessary, as otherwise a server could return malicious javascript to the user. In a perfect word, the user has to download and install the client locally and verify the file hash, For testing purposes, he can also use a web hosted version of the client. Although they are hosted on different domains, client and server can still connect to each other via cross origin resource sharing (CORS).

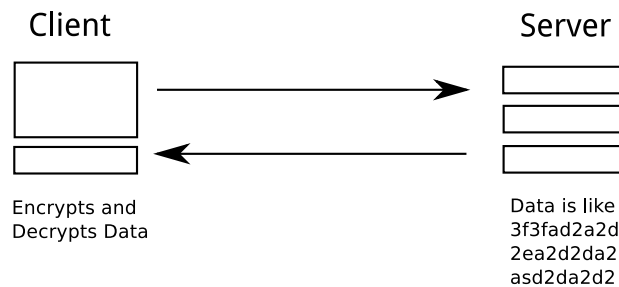


Figure 3: Client and Server code is strictly separated. The client loads the Charmet application from a local webserver, a browser plugin or a trusted server. This application connects to the server hosting the data via JSON.

### 4.3 Passphrase

Charmet uses an *encrypted data storage*. This means critical data is encrypted with a passphrase using AES-CBC (Section 3.3.1) and hosted on the server (Figure 3). Consequently the server can not read the user data. Such data includes

1. **Personal information** like phone number. This information can later be decrypted and encrypted with a public key to distribute it to other people.



2. **The private RSA key** used to decrypt everything from other users.
3. **The key directory**, which contains the public keys of all contacts. Here it is important that the receiver B of a message sent by user A only uses the newest private key to decrypt, as a malicious server of the key directory owner could return an old public key of user B.

for example. The data is signed to prevent forgery of information.

## 4.4 Verifying private keys

When we want to write an encrypted message to user B, we need the public key of user B first. However, as the server or someone else can provide a wrong public key, it is necessary to check the public key for correctness. If done so, we encrypt the public key of user B with our passphrase and store it in the encrypted data storage.

### 4.4.1 Generating Fingerprints

Currently the fingerprint  $f$  is generated by applying a SHA-256 Hash function on  $n$  (modulus) and  $e$  (exponent) concatenated:

$$F = Sha256(n||e)$$

The function in the code is called *buildHash* and is located in *keys.coffee*. For a better user experience, the fingerprint can be truncated and : seperated later on, so the result looks something like:

12 : 3f : 3e : 2f: 12 : 3f : 3e : 2f: 12 : 3f : 3e : 2f (...)

instead of

123f3e3f123f3e2f123f3e2f (...)

Figure 4 show the dialog to validate fingerprints.

## 4.5 Preventing List and Message MITM Attacks

Lists can be used to restrict access to certain information. Each list contains some people defined by the users. When a server behaves evil, we must ensure, that he can not manipulate the list and add new people for example, we will later send our message to.

Furthermore we must ensure, that the autocomplete only returns name/address pairs that have been added by the user. Imagine the following case: Alice adds Bob with his CharmeId bob@somewhere.com to her key directory. Now she wants to send a message to Bob and opens the auto complete. She types Bob, and the server returns a modified CharmeId, probably on the same server (eve@somewhere.com). The Charme client

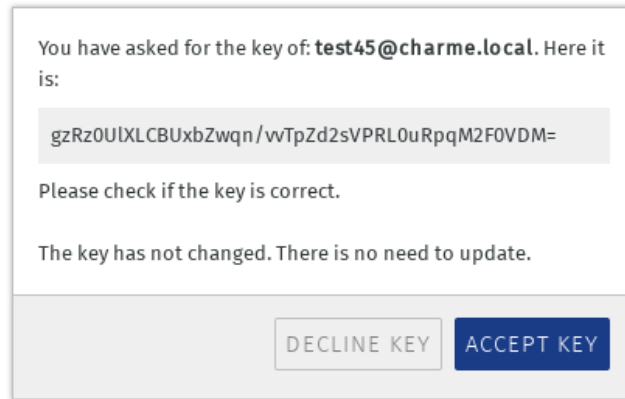


Figure 4: Fingerprint Validation Dialog

would now encrypt the message with the other key and send it to Eve. So Eve could be able to get some sensitive information as Alice thinks she is communicating with Bob.

Now how can we ensure, that this can not happen?

The second case is that a list contains additional people added by an evil server.

#### 4.6 Post encryption

A new post is encrypted, when it is restricted to a list of people. First we generate a postkey  $k$ , which encrypts the content of the post. Next we need to distribute  $k_1$  to other people. We can use the symmetric edgekeys  $e_1, e_2, \dots$  of these people. We encrypt the post key with each edgekey ( $Enc_{e_1}(k)$ ) and distribute them to the receivers. An edgekey is a asymmetric key, that is established after public key verification and send via public key to the other party. This is because asymmetric cryptography is a lot of slower than symmetric cryptography.

#### 4.7 Message encryption

Message encryption is basically working like in PGP. We save public keys encrypted in the cloud, however, here. A simplified concept is illustrated in Figure 5). A conversation  $C$  consists of  $n$  messages. A set of messages  $S \in C$  has a symmetric key each. The symmetric key should change if a user is removed or a public key of a conversation participant is updated or a certain span of time has passed. Each message is encrypted with the newest symmetric key  $s$ . A message consists of the following information:

1. Text encrypted with symmetric key  $s$
2. Conversation ID
3. Key Id of symmetric key  $s$

#### 4. Time

This information is also signed using the senders private key.

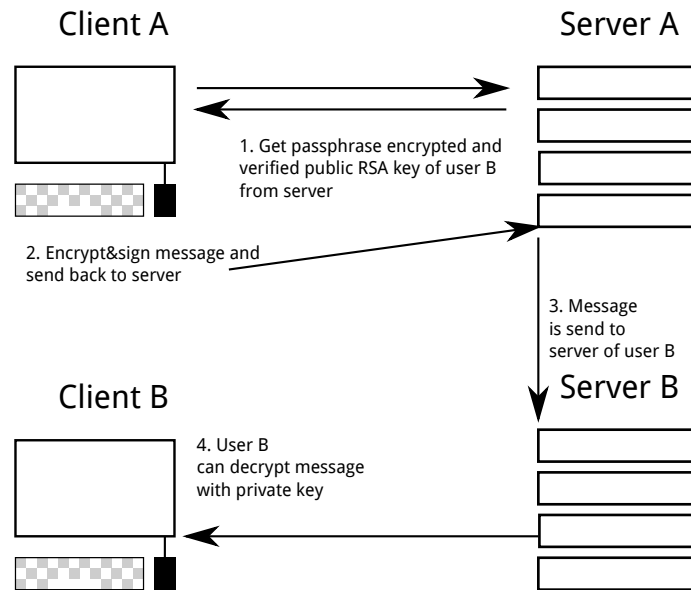


Figure 5: Basic message encryption. Server A can also send the message to more servers than only Server B. The client does not send the message directly to server B, as with increasing number of receivers this would cost to much bandwidth for the client.

**The current implementation misses some fundamental security stuff. An attacker could initiate a conversation with the identity of client A and start a MITM attack. Therefore the next step of the implementation is to sign the initial message and each following message.**

#### 4.8 Distributing private information

When giving private information like telephone number or email to other people, this is working slightly different to message distribution for performance reasons. See figure 6 for a illustration of the process. This may look kind of complicated first and you may wonder why the encryption is not done like for messages. The answer if the access to a certain person is revoked, it would be inefficient to send the newly encrypted information to all users. Instead we creates called *buckets*, each containing a maximum of  $k$  users. When a new piece of information is encrypted, we generate a random key  $K$ , and encrypt it with each bucket key  $B_i$ . As every user who has access to the information has the bucket key, the user can decrypt it with

$$plaintext = Dec_{B_i} Dec_K(chyphertext)$$

When access is revoked for a user, we simply generate a new bucket key for the bucket, the user was in and do not distribute the bucket key to the user. With this system we are able to revoke access to information and release new information to a user with doing  $\mathcal{O}(k) = \mathcal{O}(1)$  requests. For Charme we set  $k = 10$ . The code is located in *settings\_privateinfo.js*.

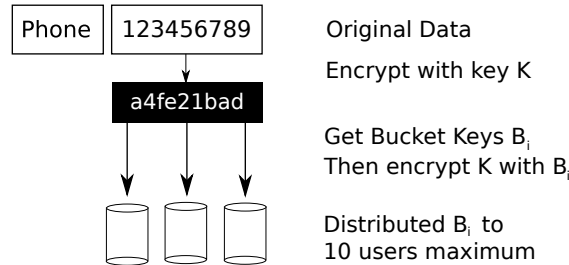


Figure 6: Encryption of personal information

## 4.9 Key Updates

When a private key or passphrase is compromised it is necessary to reencrypt the encrypted data storage. In this case, all information are transmitted to the client, decrypted with the current passphrase and encrypted with the new passphrase. The revision number for the public key increases afterwards.

## 4.10 Primary Attack Vectors

### 4.10.1 MITM on Client Software

The client software must be distributed with integrity protection in the final release. For example as a Cordova package. To use the client software on a third party server is very dangerous and only recommended for testing purposes! The host of the client can theoretically read all your data by modifying the client software.

There is also a new technology called sub resource integrity which can be used to validate Javascript and CSS resources (See <http://www.w3.org/TR/SRI/> and <http://githubengineering.com/subresource-integrity/>). Maybe they introduce some mechanism to validate a whole web page with a hash recommended by the majority of browser users and additionally located on a (Mozilla or Google) server. So let's keep an eye on that.

### 4.10.2 Attacks on Integrity Protection

Certain information, such as the Name/UserID pairs of a conversation must be distributed with integrity protection using either the private key of each receiver or the

conversation key. Asymmetric private keys used for signing the message data have the following advantages and disadvantages in contrast to the symmetric conversation key.

#### Using the private key

1. (Advantage) Slower (Factor ca. 1000)
2. (Disadvantage) No attackers inside a conversation manipulating username/userId pairs

#### Additional Information:

OWASP Checklist: OTG-BUSLOGIC-003

#### 4.10.3 New Conversations with malicious userId/Name pair

An attacker can initiate a new conversation with the name of a friend of you, but use another userId instead. Example:

1. Alice (alice@server.com) and Bob Alice (bob@server.com) are friends
2. Eve initiates a new conversation with Bob with Name "Alice" and userId eve@server.com.
3. Bob thinks he is talking to Alice, but he is not.

A solution for this problem would be to allow only names from the key directory.

#### 4.10.4 XSS

One of the most critical attack vectors in Charmé is Cross Site Scripting as with this technique private keys can be stolen. Therefore it is important to sanitize all displayed data from other people and servers. Never trust the server. For example imaging data is a JSON Object returned by the server:

```
\$("html").append("<a href='"+data.href+"'>TEST</a>");
```

If the server does not provide a hyperlink for data.href, but rather something like:

```
#' onload='$.post("someotherserver.com",{key: theprivatekey})
```

this is extremely dangerous. Therefore always sanitize data from the server!!!

#### Additional Information:

OWASP Checklist: OTG-CLIENT-002

#### 4.10.5 Session Capturing

Send JSON request from another Website to host server. Make sure you can not access unencrypted user data such as lists for example.

#### **4.10.6 DOS**

Attackers can flood small servers with a lot of requests.

#### **4.10.7 Replay attacks**

Server can provide old public/private keys which have been compromised earlier for the user .

#### **4.10.8 Malformed data**

Attackers can send malformed data to clients which results in the client to producing an exception and therefore make Charmé unusable for the user. Therefore malformed requests should be filtered by the client by checking for missing and correct typed values.

#### **4.10.9 Logging of Requests**

In the current implementation, the user id is sent to other servers when requesting a profile or performing a search on this server for example. The server host could log who is visiting which profiles. Possible counter measures are

1. Performing a set of user ids, in which only one is the actual user id. This would affect the performance however and will only work if enough users are provided.
- 2.

#### **4.10.10 Spam**

All posts from non-friends should be blocked. An exception are posts containing semantic information which should be searchable all over the web. Here we can either expand the search to friends server (efficient but may contain more spam) or even check friend of friend posts only (more computational power, but less spam).

## 5 Protocol Specification

### 5.1 Login

**1st step:** Get password salt value via *reg\_salt\_get*

**2nd step:** Login with *user\_login* and parameters u (username) and p (salt hashed password). Status PASS is returned if successful.

### 5.2 Search

When it comes to search, we face some problems. Of course, it is not efficient to query hundreds or even more server from one client in the charme network. Filesharing applications typically use Kademlia or Chord to index data within peer-to-peer networks. Such concepts may also be useful for search, as YaCy for example has shown. Thorsten Schütt et al. have proposed a technique which can be used for multidimensional parameter search in their paper *Range queries on structured overlay networks*. A search in social networks could also use the social graph to rank results of near people higher than from people more far away.

### 5.3 Messages

To load the conversation preview list, *messages\_get* is called. Each message has a message key. If a new message Key is generated, it is encrypted with the edgekey of the receiving users (Section 5.4).

### 5.4 Edgekeys

For every kind of communication between two users we generate a directed edgekey, to increase performance as asymmetric encryption for every kind of message would require to much computational power. The edgekeys should change after a certain timespan. (1 week?). Currently the edgekey is generated after a key was added to a key directory in *settings\_keymanager.js*. The edgekeys are requested to decrypt from user B. Later they will be cached in local storage of user B to even gain the performance we have planned to gain.

### 5.5 Recryption of data

When the private key is updated, it is required to recrypt the data in the encrypted data storage. This is done in the function *updateDataOK()* in *settings\_keymanager.js*.

#### 5.5.1 Signup

Client:

1. Get salt s from server
2. Generate password  $p = \text{Sha256}(s||p')$  from plaintext password  $p'$

3. Generate passphrase PP and asymmetric keypair KP. Encrypt KP with PP.
4. Generate sub keys fastkey1 and fastkey2, encrypt them with passphrase.
5. Hash username with FK1 to prove integrity later
6. transmit  $\text{Enc}(\text{KP})$ ,  $\text{Enc}(\text{FK1})$ ,  $\text{Enc}(\text{FK2})$ ,  $\text{Sha256}(\text{username}||\text{FK1})$  and other user-data to server.

### 5.5.2 Add public key to keydirectory

A new public key is requested, verified and added to the key directory.

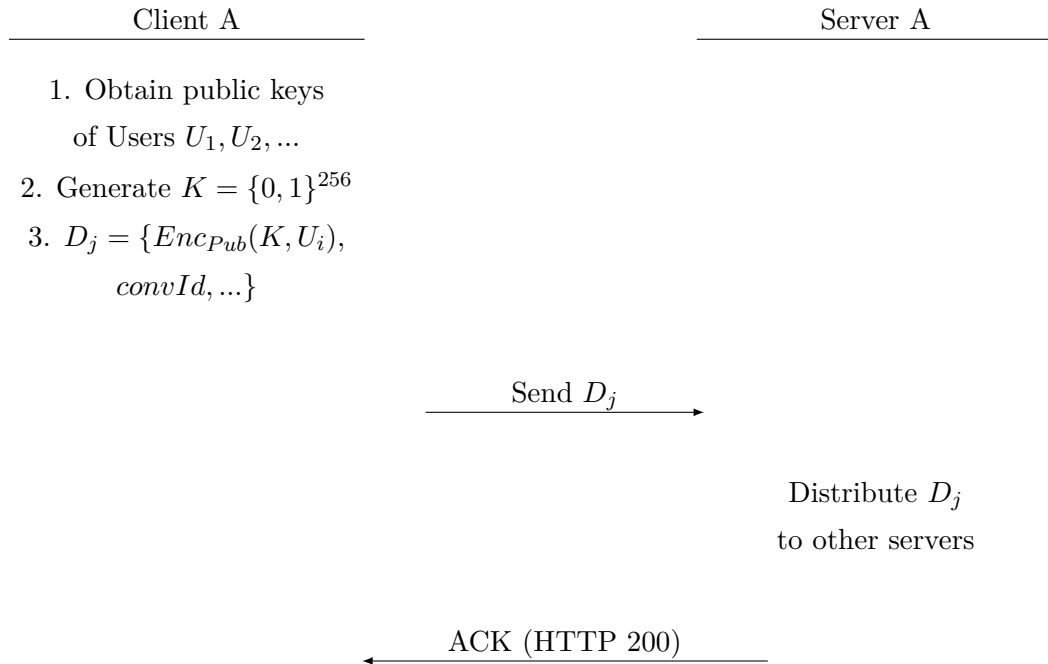
<u>Client A</u>	<u>Server A</u>	<u>Server B</u>	<u>Client B</u>
-----------------	-----------------	-----------------	-----------------

1. Client A requests public key of client B
2. TODO

### 5.5.3 Init Conversation

Let convId the id of the conversation. Let's call  $D_j$  the payload. It contains information relevant to the conversation such as time, last preview text, members or identifier.  $D_j$  must be integrity protected of course. K is a 256 bit symmetric key used for AES encryption. The communication between the client initiating a new conversation and its server is defined as this:





Server A distributed the payload to other servers afterwards. Technically this is done by a Gearman daemon running in the background. The receiver clients can afterwards decrypt the key  $K$  with their private key.

More in detail, the JSON message schema to init a conversation from the senders client to the senders server looks like this:

- **id** = message\_distribute
- **action** = initConversation
- **messageData** (signed with  $K$ )
  - **hmac**: The HMAC to check integrity
  - **revision**: Revision of  $K$
  - **objType** = object
  - **obj**
    - \* **receivers**: List of userIds
    - \* **usernames**: List of (userId,name) tuples.
- **messageKeys**: List of public key encrypted public keys  $K$

The server redirects the message to the receivers servers afterwards, the message JSON has the following properties:

- **id** = message\_receive

- **action** = initConversation
- **messageData** (signed with  $K$ )
  - (See above)
- **key**: Public key encrypted public key  $K$  for the receiver

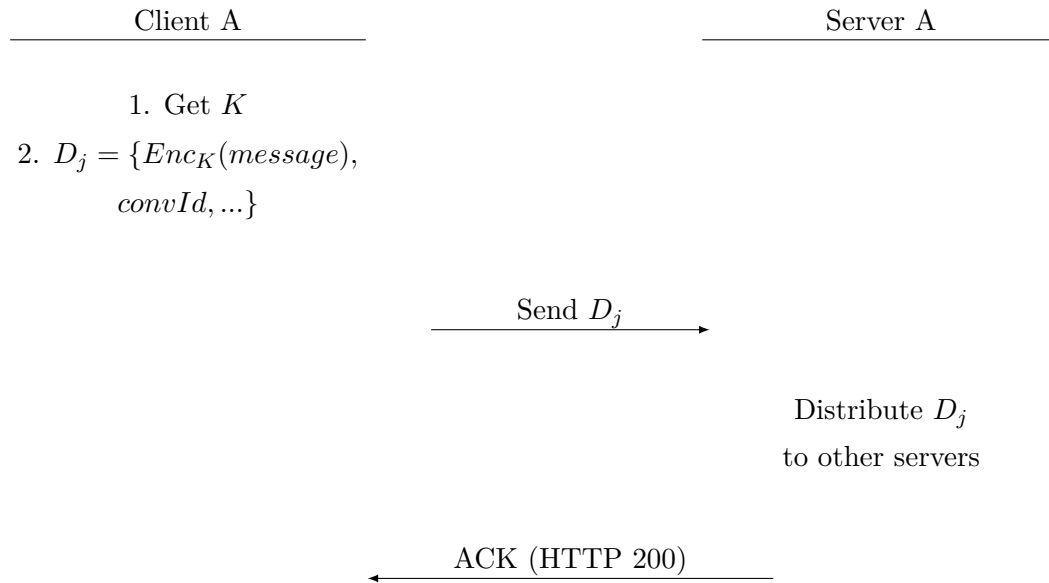
Please note that here we could be more efficient in the future by grouping the message\_receive requests for same people on same servers.

#### 5.5.4 Add People to Conversation

People present in the key directory can be added to an existing conversation. As they should not be able to see previous messages, a new message key  $K_{(revision+1)}$  is generated in this case by the person who adds the new people. This key is distributed to the other people afterwards. The people in the conversation have now to be signed again.

#### 5.5.5 Write message

Let  $K$  be the message key, which has been generated at conversation init.



## 6 Future Work

### 6.1 Token Generation and API

In the future Charme will be able to be used by other applications as data storage. Therefore, we create a key  $A_i$  and a token  $T_i$  for each application  $i$ . All keys and tokens

are generated randomly and encrypted with a passphrase derived key  $A'$ . When the passphrase is updated, we need to reencrypt  $A'$ . An application can now save data to the Charme server, once the triplet  $A_i$ ,  $T_i$  and Charme `userId` was added to the application. To perform an action  $A$ , the charme server sends a random value  $R$  to the application. The application generates the hash value  $H(A|R|T_i)$  and sends it back to the server, where the hash is validated to maintain authenticity. Actions can be `STORE(key, value)` and `GET(key)`. Possible applications could be a note taking app or a calendar, saving data encrypted on the server for example.

## 6.2 Use of encrypted Databases

For searching contextual posts, it will be necessary to hide the meta information. Here we could use encrypted databases as CryptDB developed at the MIT (<http://css.csail.mit.edu/cryptdb/>). The developers also responded to the paper *Inference attacks on property preserving encrypted databases* from Naveed et al., 2015, which questioned the security of such databases.

## 6.3 Location based Chats

Idea: We save the GPS position  $p_i = (x_i, y_i)$  every  $x$  seconds. After some time we generate a vector  $v = (x_0 + x_1 + x_2 + \dots, y_0 + y_1 + y_2 + \dots)$ . Now we use a hash function  $H$  to map the vector with a location accuracy defined as  $a$ .

$$H(v) = (\lceil v^{(0)}/a \rceil + \lceil v^{(1)}/a \rceil)$$

This will generate a unique key for each location. We save a trace of locations in the first place to detect a moving object such as a train, ship or bus. However, the location sensors of the mobile device must be very accurate!

## 6.4 Data Sharing for the Internet of Things

For Internet of Things (IoT) devices, it may be helpful to allow services or institutions to access your devices. Charme could handle the key management with using the already existing symmetric edgekeys  $E$  (See section 5.4 for details) derived from the public key for the communication between service and device. (Figure 7) Of course, we can not use the edgekey  $e$  directly, as a evil device could otherwise intercept all communication between the service and the user. Therefore we derive the key  $K\_A$  in the following manner:

$$R = \text{Random256bitkey}$$

$$K\_A = \text{Sha256}(R||e)$$

Afterwards we send  $R$  to the service provider and  $K\_A$  to the device.

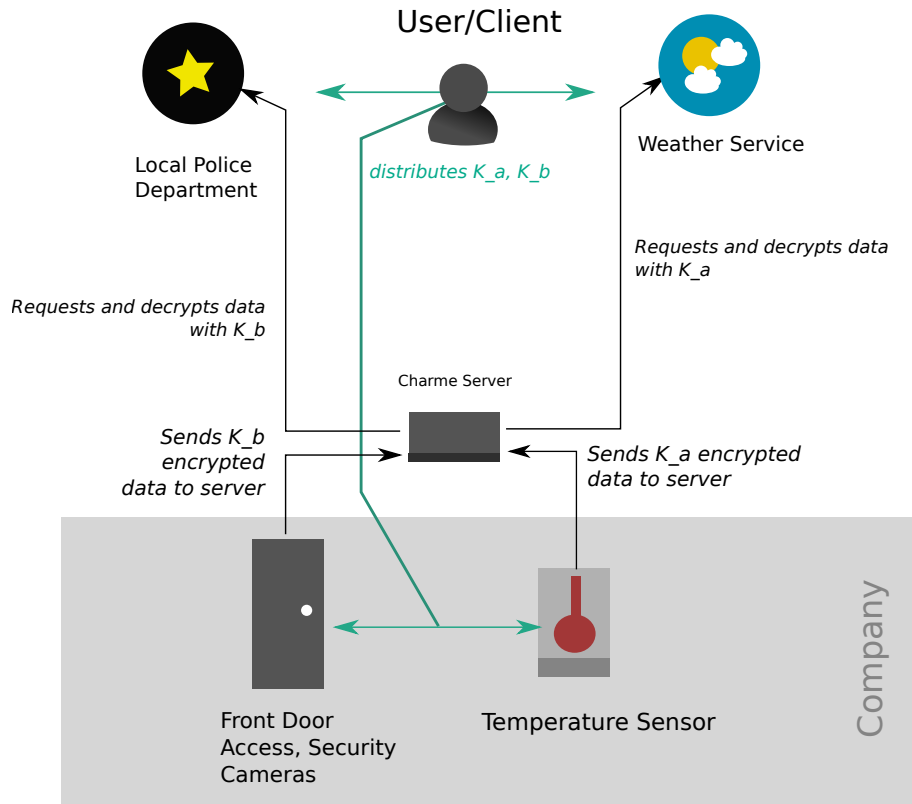


Figure 7:  $K_a, K_b$  are keys shared between some service and the device. The data is encrypted so that only the user, the service and the device can see the data, but not the Charme server. Please note that for  $K_b$  there must be some security mechanism like timestamps to avoid replay attacks when the door is opened for example.

## 7 Terminology

||

Concatenation Operator. For example " $a$ "||" $b$ "  $\Leftrightarrow$  " $ab$ "

### Conversation

A sequence of messages of different users

### Fastkey

A symmetric key dependent on the passphrase. It is only known by the user itself, no other persons or servers.

### Revision

After a key is updated it gets a new revision. A key update can happen after a new

keyring is generated or someone has been removed of a conversation for example. both cases we generate an updated key.

**userId**

An unique identifier for each person in the Charme network, similar to email. Example: ms@myserver.com

## 8 Sources

### References

- [1] Carle et al. *Network Security Lecture @ Munich Technical University*. 2015. URL: <http://www.net.in.tum.de/de/lehre/ws1415/vorlesungen/network-security/>.
- [2] EFF. *Baseless Calls to Expand Surveillance Fit Familiar, Cynical Pattern*. 2015. URL: <https://www.eff.org/deeplinks/2015/11/baseless-calls-expand-surveillance-fit-familiar-cynical-pattern>.
- [3] Phil Zimmermann. *Why I wrote PGP*. 1991. URL: <https://www.philzimmermann.com/EN/essays/WhyIWrotePGP.html>.