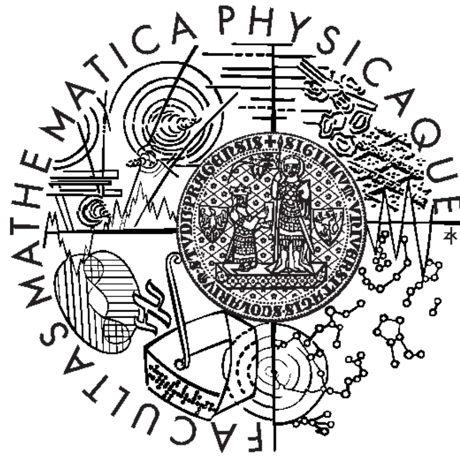
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Maroš Kasinec

Desktop client for open social networks

Department of Applied Mathematics

Supervisor of the bachelor thesis:	RNDr. Tomáš Valla, Ph.D.
Study programme:	Informatika
Specialization:	Programování

Prague 2012

I would like to express appreciation for help and guidance throughout this thesis to my supervisor, RNDr. Tomáš Valla, Ph.D. Least but not last great thanks belongs to Diane and Matúš Kasinec for their prompt assistance and expertise regarding proofreading.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In .... date .... signature of the author

Název práce: Desktop client for open social networks

Autor: Maroš Kasinec

Katedra: Katedra aplikované matematiky

Vedoucí bakalářské práce: RNDr. Tomáš Valla, Ph.D., Informatický ústav Univerzity Karlovy

Abstrakt: Sociálne siete zažívajú v poslednej dekáde obrovský romach a ovplyvnili nielen spôsob on-line komunikácie a sociálnej interakcie ale tiež oblasť obchodu, médií či vládnych inštitúcií. Avšak ich hlavný nedostatok, uzvretý a centralizovaný charakter, ostáva nepovšimnutý medzi širokou verejnosťou. Táto práca pojednáva a hodnotí ich otvorené a decentralizované alternatívy a súčasne sa zameriava na jednu konkrétnu – buddycloud. Vďaka využitiu XMPP protokolu, buddycloud a jeho protokol Channel sa javí ako sľubný prístup pre otvorenie ekosystému sociálnych sietí. Umožňuje im komunikovať federatívnym spôsobom ako funguje dnes e-mailová sieť. Ako príspevok do projektu buddycloud táto práca predstavuje aplikáciu SocialDesktopClient, desktopový klient pre širokú škálu sociálnych sietí. Bližšie sa zaoberá modulárnym návrhom klienta a implementáciou protokolu Channel ako prvej sociálnej sieti.

Klíčová slova: desktop client, open social networks

Title: Desktop client for open social networks

Author: Maroš Kasinec

Department: Department of Applied Mathematics

Supervisor: RNDr. Tomáš Valla, Ph.D., Computer Science Institute of Charles University

Abstract: For the past decade social network sites emerged rapidly and effect not only online communication and social experience but also businesses, media and governments. However, their greatest deficiency, closed and centralized character, remains unnoticed among the general public. This thesis discusses and evaluates open and decentralized alternatives for them and draws attention to one particular – buddycloud. While leveraging the use of XMPP protocol, buddycloud with its Channel protocol appears to be a promising approach for opening ecosystem of social networks. It enables them to work in federated manner like e-mail network does today. As a contribution to the buddycloud project this thesis presents SocialDesktopClient, a desktop client for multiple social network services. It deals with modular client architecture and a Channel protocol implementation as the client's first social network service.

Keywords: desktop client, open social networks

Contents

1 Social Network **2**
- 1.1 Problem definition . . . 2

2 Federated Social Network **5**
- 2.1 Network topology . . . 5
- 2.2 Social network aspects . . . 6
- 2.3 Existing technologies . . . 7
- 2.4 Evaluation . . . 14

3 XMPP – the golden mean **17**
- 3.1 Protocol basics . . . 18
  - 3.1.1 Communication primitives . . . 18
- 3.2 Components . . . 20

4 buddycloud **22**
- 4.1 Channels . . . 22
  - 4.1.1 Privacy settings . . . 23
- 4.2 Channel protocol . . . 23
  - 4.2.1 XEP-0060: Publish-subscribe . . . 24
- 4.3 Content format . . . 25
- 4.4 Operations . . . 27
  - 4.4.1 Channel service discovery . . . 27
  - 4.4.2 Creating and controlling channel . . . 28
  - 4.4.3 (Un)subscribing . . . 30
- 4.5 Current project overview . . . 31

5 SocialDesktopClient **33**
- 5.1 Implementation and preliminaries . . . 34
- 5.2 Architecture . . . 36
  - 5.2.1 Threading model . . . 36
  - 5.2.2 Core . . . 37
  - 5.2.3 Qt GUI . . . 40
- 5.3 buddycloud plugin . . . 41
- 5.4 Overview and future works . . . 41

Conclusion **42**

Bibliography **44**

List of Figures **48**

List of Tables **49**

List of Abbreviations **50**

Attachments **51**

1. Social Network

The term *social network* was introduced in 1954 by J. A. Barnes in his research of class and committees in Norwegian island parish [1]. In theory, it describes a social structure composed of entities, either individuals or organizations, tied together according to the relationship between them. A whole field of theory has been formed to study and analyze social networks.

However, the term itself acquired a new meaning in ordinary vernacular. It becomes more of a phenomenon than a sociological term in the past decade. As the matter of fact, for many people the "social network" of today is just a synonym to the actual largest social network provider - Facebook.

To avoid misconception, we are going use appropriate terminology. Common denominator for all those online social network service is a Web-based architecture. For this reason we are going to use the term *Social Network Sites*, further SNS. Boyd and Ellison defined SNS as *"Web-based services that allow individuals to*

- *construct a public or semi-public profile within a bounded system,*
- *articulate a list of other users with whom they share a connection,*
- *view and traverse their list of connections and those made by others within the system."* [2]

On the other hand, SNS are often referred to as *Social Media*. This term is also correct, nonetheless it has broader meaning, because it exceeds the range of internet-based applications. TV or radio broadcast can also be considered as Social Media.

As the content-sharing phenomena grew, many websites progressively implemented new social features and become SNS. As we can see in Figure 1.1, a spectrum of SNSs is quite colorful. They are focusing on various kinds of data, from simple bookmark sharing to streaming entire online activity.

They also differ in a way they address audience. While many of them reaching the crowds without difference, some are attracting users and communities with shared interests or common language, nationality or religious identity.

They have expanded further beyond the personal utilization. Media uses them for publishing, businesses for advertising, governments for campaigns and the space of opportunities continuously grows. To conclude, the impact of SNSs on our society is great and inevitable.

1.1 Problem definition

While the significance of SNSs at present is apparent, serious deficiency of them remains unnoticed among general public. The environment SNS work and run in is closed and controlled by a single company, naturally driven by profit. Users' data is collected in one big information silo, where they are sold and used for targeted advertising in better-case scenarios. Information from one silo cannot be

Figure 1.1: Variety of SNSs [45]

reused in the other. This pushes users to sign up for every such SNS by accepting service's policies agreement, thus creating profiles and articulating same connections every time.

Moreover, people have only little control over how their data is presented to them or to their connections. Even less control over how they are processed. There are numerous cases, where people suffered from those SNS practices. Deactivating accounts [3], censoring information [4], discretely changing privacy policies [5], gradually leaking private data to advertisers and trackers [6], etc. Having to choose between sharing content at a cost of loosing privacy and keeping data to ourselves is a dilemma aware users have to face.

Of course, people are not obligated to stay with particular SNS provider. But the cost of leaving the party is too high. It means loosing important connections, consequently loosing the way to communicate with them. In the worse case also loosing valuable personal content.

In this project we are going to map and describe current situation and efforts in the field of open social networks. We clarify the paradigm of open or rather federated social network and take a look at existing technologies, protocols, standards and implementations dealing with various aspects of this paradigm. As long as majority of them are built on top of Web we will evaluate this approach in terms of federated social network.

Further on we bring attention to *eXtensible Messaging and Presence Protocol*, XMPP in short or *Jabber* as many are familiar with, and show reasons why is it befitted for building federated social network platform.

Our main focus will be on buddycloud project as the prospective approach for building such platform. In frame of buddycloud we are going to discuss *Channel protocol*, an extension build upon XMPP, which proposes model for federated social network interaction.

As a result of software engineering part of this thesis we present SocialDesktopClient, a desktop application for multiple social network services. We take closer look at its modular design and architecture in general.

In order to contribute to the buddycloud project, first social network service plugin will be an implementation of buddycloud Channel protocol. To this extent we stayed in touch with buddycloud open-source community throughout SocialDeskopClient development.

As long as there is no open-source, cross-platform, desktop client designed especially for social networks, we think this project can bring a decent product for the users. Moreover, by implementing buddycloud plugin we want to manifest the principles of open federated social network.

2. Federated Social Network

The problem menioned above can be solved by giving a user the ability to move between providers without taking away their connections and also retain the ability to interact with those connections in the same or similar fashion. It will not only solve users' isolation problem and breaks information silos, but it may also lead to healthier competitive ecosystem of social network providers.

In such environment independent providers are capable of exchanging rich social content. The way they interact together would be standardized in open and collaborative manner and improvements accepted by consensus, thus no one can dictate the terms. From a user perspective one may freely choose a trustworthy provider, as long as technically capable, install its own private service and join in.

As the problem clearly states, proposed platform requires *decentralized* approach. Such network architecture of a system is not a new concept. We are all familiar with e-mail architecture (Figure 2.1). One does not have to sign up to different providers in order to exchange messages with users of the same provider. A little further behind internet boundaries - telecommunication system shows another example of a world interconnected by different providers.

2.1 Network topology

For the network topology side of architecture we need to distinguish a couple of decentralized models. A *fully* decentralized topology constitutes a peer-to-peer network, where all nodes are able to communicate with each other, therefore are considered to be equal. However, such topology is not suitable for social networking, since there is no guarantee a user always stays online in order to provide his profile and other shared content.

There has to be a compromise made between entirely centralized client-server model and fully decentralized peer-to-peer network. A *federated model* offers

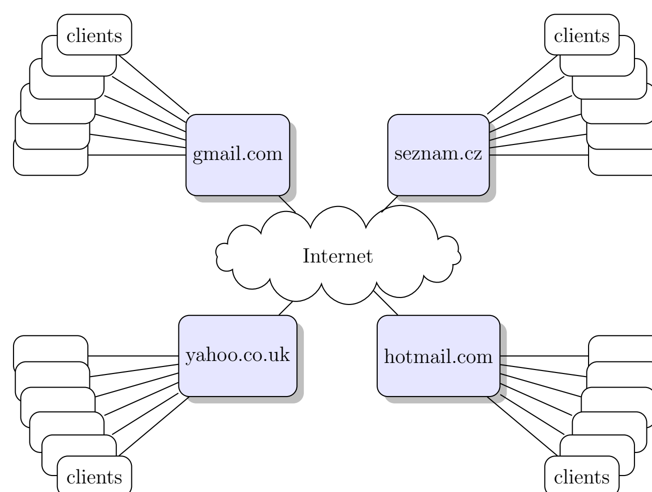


Figure 2.1: Decentralized architecture of e-mail network

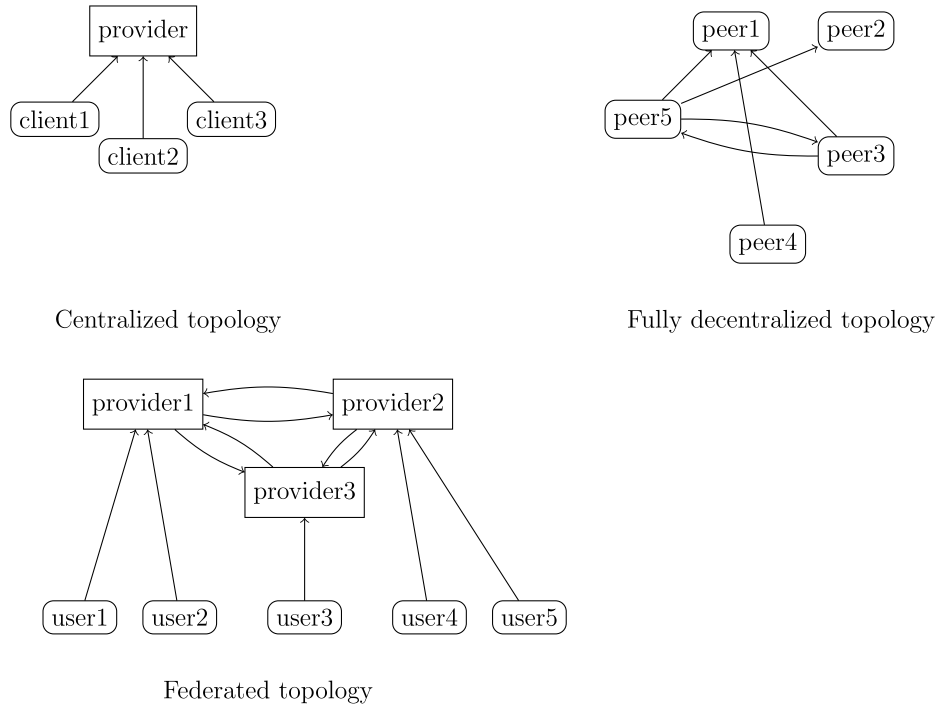


Figure 2.2: Network topologies comparison

such a network. Unlike fully decentralized topology it is always on *home server* which acts on behalf of a user, while still preserving decentralized character. In Figure 2.2 we can see the comparison of three mentioned topologies.

Because of federated newtork topology, in this project we will refer to the proposing social network platform as *Federated Social Network*, further FSN.

2.2 Social network aspects

In terms of decentralized social network architecture certain aspects need to be discussed in order to understand the challenge of building FSN.

Social graph representation

A social graph, in the context of online social networks, constitutes a data structure of a graph, where nodes represents entities and edges a specific relationships between them. Such data structure can be very beneficial in terms of searching capabilities or statistics. Facebook's Open Graph protocol [7] is an example of utilizing social graph.

Opposed to centralized system, capturing such graph in FSN can be quite challenging task, however not impossible.

Data model and distribution

Speaking of acting entities in online social network in general, it is important to define a *model of interaction*. In detail, such model will define how entities

interact with each other, what information they can share and in what way are they shared.

For FSN it is essential to develop standardized data format and protocol of communication for the content to be distributed between independent providers. This can be a long-term process, in which a community of developers and testers takes their part.

Security

For an arbitrary online social system it is essential to preserve an identity of the involved individual. Because a user, specifically client's software, cannot guarantee to be constantly active, the identity has to be maintained on an always-online server, which will act on behalf of a user.

On the other hand there is a matter of privacy settings. While a central authority is ruling, present SNSs lacks of fine control over content privacy. Therefore it is desired that privacy setting will be implemented at the very protocol level.

2.3 Existing technologies

There is an ongoing effort to build FSN using Web technologies. As a matter of fact, World Wide Web of today is an ultimate sphere of data presentation. While the field of Web technologies is constantly emerging, a W3C[1] group called *Federated Social Web*, takes part in supervising in this matter. In the following we are going to briefly present those technologies and the use of them. They are all open and designed to work in a decentralized fashion.

OpenID

A simple and widely-used decentralized authentication across the web presents OpenID protocol [8]. It was adopted by big companies like Google, Yahoo!, Facebook and MySpace.

The principle of authentication is straightforward. Basically the user is queried for authentication by some third party system that implements OpenID protocol. Subsequently the user provides his identifier, which is an HTTP(S) URL given by OpenID provider. User's browser is then redirected to the OpenID provider's site for authentication, which in return advises the third party of the result. Means of such communication are shown in Figure 2.3.

OAuth

While OpenID provide just a simple way of authentication, OAuth [9] protocol extends security in terms of authorization. It enables service provider to share only limited resources with a third party on behalf of a user.

Initiative to create OAuth comes from increasing trend of Web data utilization. Web service providers were pushed to open up user's content to the third

[1] World Wide Web Consortium

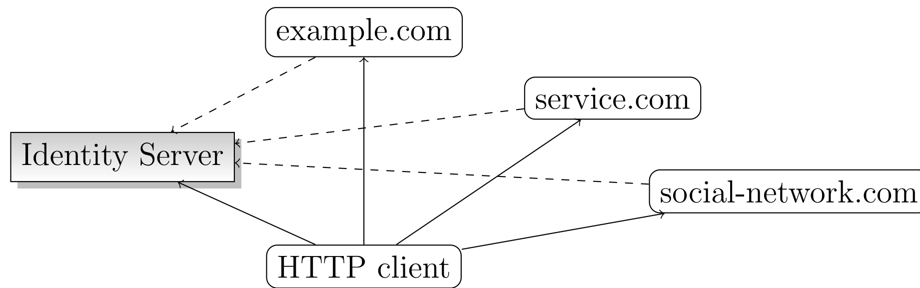


Figure 2.3: Identity server network model

party applications. Each provider was developing his own system of accessing user's data, which ends up with with protocols like Google's AuthSub, AOL's OpenAuth, Yahoo's BBAuth and FlickrAuth and Facebook's FacebookAuth. OAuth main design goal is to unify those existing proprietary protocols, which shows as a difficult challenge as main protocol developer claims [10].

WebID

A slightly different approach for authentication presents WebID protocol [11]. The idea behind it is to use a self-signed certificate in the user's browser to log in to a third-party site.

Initially, on user's request, WebID provider generates X.509[2] certificate. Among other information inside the certificate, there is also WebID URI pointing at user's profile page. In order to be authenticated by a third party, user's browser provides stored certificate. Third-party site will then extract domain part and verifies identity with the server. This process is described closely in Figure 2.4.

The key advantage of WebID is that it utilizes existing infrastructure already implemented in Web browsers.

Webfinger

A question of representing identity as a form of unique human-readable identifier has not yet been dealt with. Assuming identifier shall be addressable there are not many choices, while trying to reuse existing infrastructure. Identifier as a form URL is fairly acceptable, since it is readable and discoverable by design. However, users' mindset recognizes URL to be a representation of a resource as a document or media file rather than identity of a person.

The most suitable would be to use an e-mail address as it reflects identity in the online world the most. It is discoverable and broadly implemented in network application. The problem with e-mail addresses is that it is not readable like URL in the Web space.

Webfinger [12] makes it possible to attach rich metadata to e-mail address, like public profile, pointer to identity provider, services used by that address.

[2]X.509 – A standard for security systems with public key infrastructure described in RFC 3280

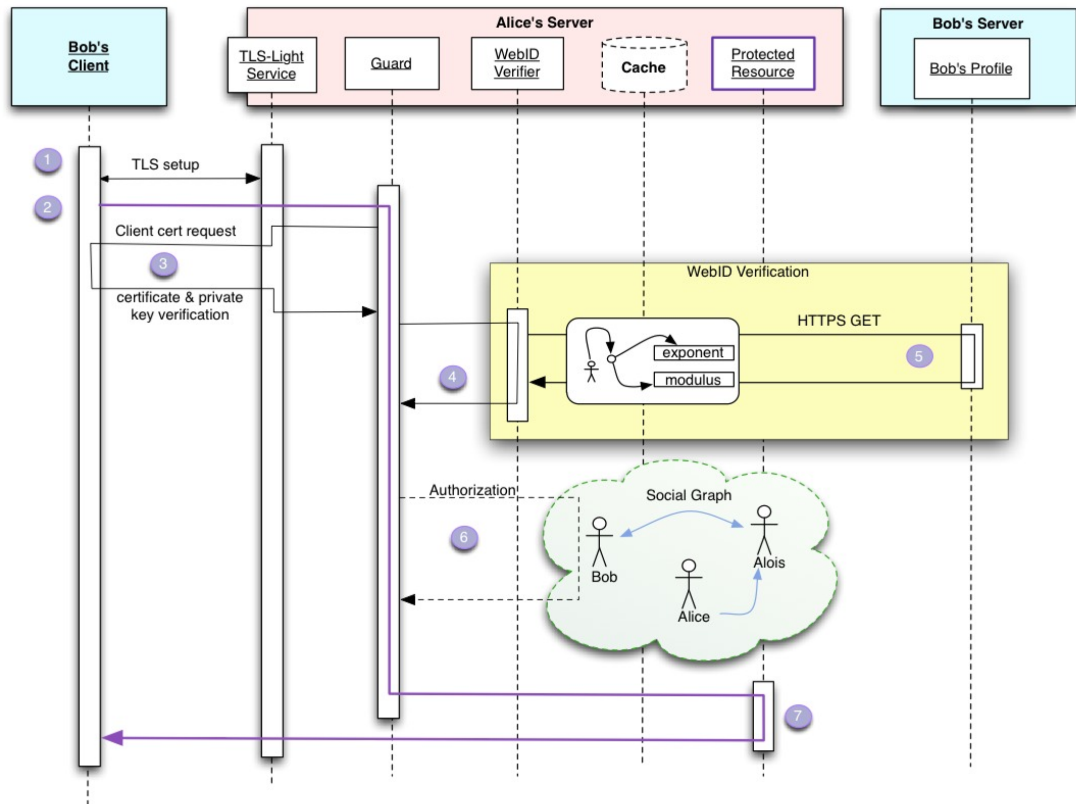


Figure 2.4: WebID authentication process [11]

FOAF

Another aspect that needs to be covered in a social network is a representation of relationships. There are many personal websites, blogs, social network profiles, that articulates relationships with others.

Project FOAF, as an abbreviation of Friend Of A Friend, is designated for such purpose. It introduces FOAF language, that is based on W3C's RDF [13] framework FOAF vocabulary [14] can help machines to understand how people are linked on the web. Listing 2.1 shows an example of a person's profile and his connections using FOAF.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:foaf="http://xmlns.com/foaf/0.1/">
  <foaf:Person>
    <foaf:name>Harry Potter</foaf:name>
    <foaf:gender>Male</foaf:gender>
    <foaf:title>Mr</foaf:title>
    <foaf:givenname>Harry</foaf:givenname>
    <foaf:family_name>Potter</foaf:family_name>
    <foaf:homepage rdf:resource="http://www.hogwarts.edu"/>
    <foaf:weblog rdf:resource="http://www.hogwarts.edu/blog/"/>
    <foaf:knows>
      <foaf:Person>
        <foaf:name>Ron Weasley</foaf:name>
      </foaf:Person>
```

```
    </foaf:knows>
    <foaf:knows>
      <foaf:Person>
        <foaf:name>Hermione Granger</foaf:name>
      </foaf:Person>
    </foaf:knows>
  </foaf:Person>
</rdf:RDF>
```

Listing 2.1: FOAF example

Atom Syndication Format

In order to distribute data over a decentralized system with independent software implementations there has to be some sort of consensus for the structure of carried data. For this purpose standard like The Atom Syndication Format has been developed [15].

Atom is an XML-based format designed to hold related information in so-called *feeds*. Each feed is composed of items known as *entries*. Format describes an extensible set of attached metadata like title, author name, content category, related timestamps, and more. An example of atom feed is shown in Figure 2.2.

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">

  <title>Example Feed</title>
  <link href="http://example.org/"/>
  <updated>2003-12-13T18:30:02Z</updated>
  <author>
    <name>John Doe</name>
  </author>
  <id>urn:uuid:60a76c80-d399-11d9-b93C-0003939e0af6</id>

  <entry>
    <title>Atom-Powered Robots Run Amok</title>
    <link href="http://example.org/2003/12/13/atom03"/>
    <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
    <updated>2003-12-13T18:30:02Z</updated>
    <summary>Some text.</summary>
  </entry>

</feed>
```

Listing 2.2: Atom example

Atom format is currently used Web-wide by all sorts of different applications. Since it is a format it is not bound to any particular communication protocol. Thanks to extendible XML based structure it can used for various content syndication.

ActivityStreams

In comparison with Atom, which defines a format for content syndication in general, ActivityStreams standardize a way of expressing an online activity in a machine-readable format [16]. Motivation comes from SNSs, that opened up to a third-party applications. The standard was adopted by big players in the field like Facebook, MySpace, Opera, Socialcast, Superfeeder, Windows Live, and others.

Standard in the simplest form describes activity as a composition of three types of information, an *actor*, a *verb* and an *object*. For example "Joe posted a video" or "Sara uploaded a photo". In addition to those three, there may be metadata like *time*, when it happens, *target* to which activity was performed, e.g. "Sam saved a movie to wishlist", *ID* of an object, *summary*, a human-readable description of an object, etc.

Activity base schema [17] defines wide spectrum of verbs and object types, thus offering rich expression capabilities for variety of applications also beyond social services.

OExchange

People browsing the Web are all familiar with sharing Web content by providing its URL. With the advent of social network services, this practice becomes instantaneous. SNS providers developed mechanism for sharing URL-based content within their system, so when a user chooses to share particular content all she has to do is to click on SNS provider symbol somewhere near the content on the same site and the URL is shared instantly.

Since each SNS provider developes its own way of sharing URLs, content sites had to integrate with each individually. OExchange protocol addresses this issue by providing a standardized mechanism of sharing URLs. It also describes how a service can be discovered for an OExchange support automatically [18].

Moreover, it can be used for posting URLs to arbitrary services like online translation or printing. Protocol is as of now adopted by practically every popular SNS.

Pubsubhubbub

As the distribution of user's content is essential in a decentralized system, there has to be a way of pushing new content to the interested parties. Such task can be typically addressed by a *publish-subscribe pattern* known in sofware architecutre.

For delivering content within the meaning of this pattern a protocol called *Pubsubhubbub* has been created [19]. It introduces three components in the process. A publisher, subscriber and a hub in the middle. Publisher and subscriber can either be a public service like content aggregator or a private webserver with user's profile. The concept is rather straightforward. When the publisher updates his content, she pings hub for the update. Hub will retrieve the content from the publisher and pushes it to every subscribed party. Illustration of this process can be found in Figure 2.5.

All of this is happening on the basis of HTTP GET and POST requests, therefore the URL endpoint needs to be provided by publishers and subscribers. To that end, it is impossible to push content directly to clients.

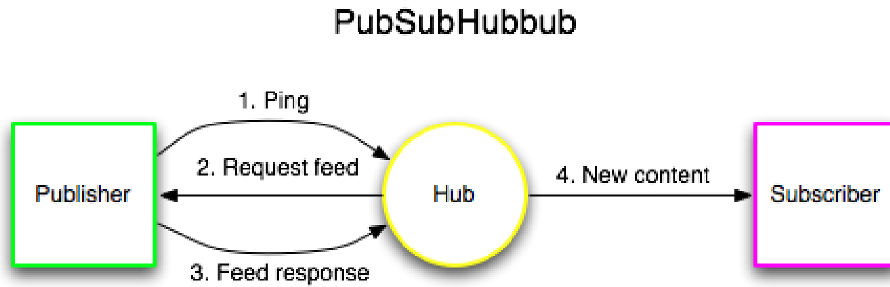


Figure 2.5: Pubsubhubbub architecture [20]

Salmon

On the contrary feed subscribers often requires to interact on the published content in the form of comment, annotations, replies, mentions, etc. Once the content is already out of the source site, it is desired for the content publisher to be notified of such feedback and possibly being able to republish it.

For this reason Salmon protocol has been developed [21]. It provides a kind of counter-part for the Pubsubhubbub protocol.

OStatus

Each protocol or standard we have mentioned so far provides only partial functionality required for FSN. Implementation of an arbitrary subset of them would result in compatibility issues for federation. A compound solution proposes OStatus [22]. It is rather a suite of protocols than protocol itself. It covers the following five of aforementioned protocols.

- Atom
- Activity Streams
- Pubsubhubbub
- Salmon
- Webfinger

It takes and adopts these standards together in a way that enables users to follow each other independently on OStatus provider.

It has found a widely recognition in the field FSN implementors and has been adopted by projects like StatusNet, Friendica, Duuit!, GNU Social, Lorea, Project Danube, buddycloud and others.

Tent

Tent [23] is an HTTP-based protocol for federated social interaction. It enables users to share profiles, statuses, messages, photos and other content types. Tent implements a concept of *apps*, which makes it possible for the developers to build custom applications upon Tent network. Eventually, users can make a use of them by authorizing app for access to certain content selectively.

Control over data is entirely in the hand of users. For this reason, as the documentation states, Tent server can also be employed as a user's personal data vault.

In relation with other standards, Tent leverages JSON [24] format for data syndication and for the apps authorization side of things OAuth 2.0 [9] protocol has been utilized.

DSNP

Distributed Social Networking Protocol is an example of a comparatively different approach to those we have presented so far [26]. Although it is a protocol, we describe it here, because it has the one and only reference implementation consisting of two parts.

The first one is a deamon, DSNPd, which is responsible for server-to-server communication on behalf of its users. It implements core protocol [25]. DSNP address the security as a key factor of social interaction. To do that it leverages RSA-based identity. This type of cryptography is used also for the sharing of secrets and the declaration of relationships. In summary, protocol defines several RSA private keys, each for a different priviledge level. Network communication is not specified in the DSNP, therefore it is independent of network protocols.

The second part is a front-end called ChoiceSocial, written in PHP. It is installed together with DSNPd on a server machine where it provides a Web-based interface. This way the presentation layer is separted and can represented by any front-end in the future.

Diaspora

Very popular for its mediated startup is Diaspora project [27] started by the group of sholars in 2010. Their goal was to make an open decentralized social network platform for everyone. Some people also believed it is the Facebook's first serious challenger. Figure 2.6 shows a preview of Diaspora's Web interface.

Diaspora's approach for creating FSN was to attract as many people as possible. They put the effort for documentation and protocol development beside, which results in the inability for developers to implement compatible software. Despite the fact Diaspora is decentralized, it can communicate only within its own instances.

Independent attempts were made to interconnect Diaspora with existing technologies, but they could not cope with further separated development of the project. As of now this project is in the hands of open-source community.

buddycloud

The buddycloud project has been in the field of FSN for quite some time. It proposes an architecture for building FSN on top of existing XMPP network infrastructure.

The idea of sharing content spins around the concept of *Channels*. A particular channel can be bound to a user, organization, or any topic in general. Access model of buddycloud protocol allows users to create, moderate, publish, or just follow the channel.

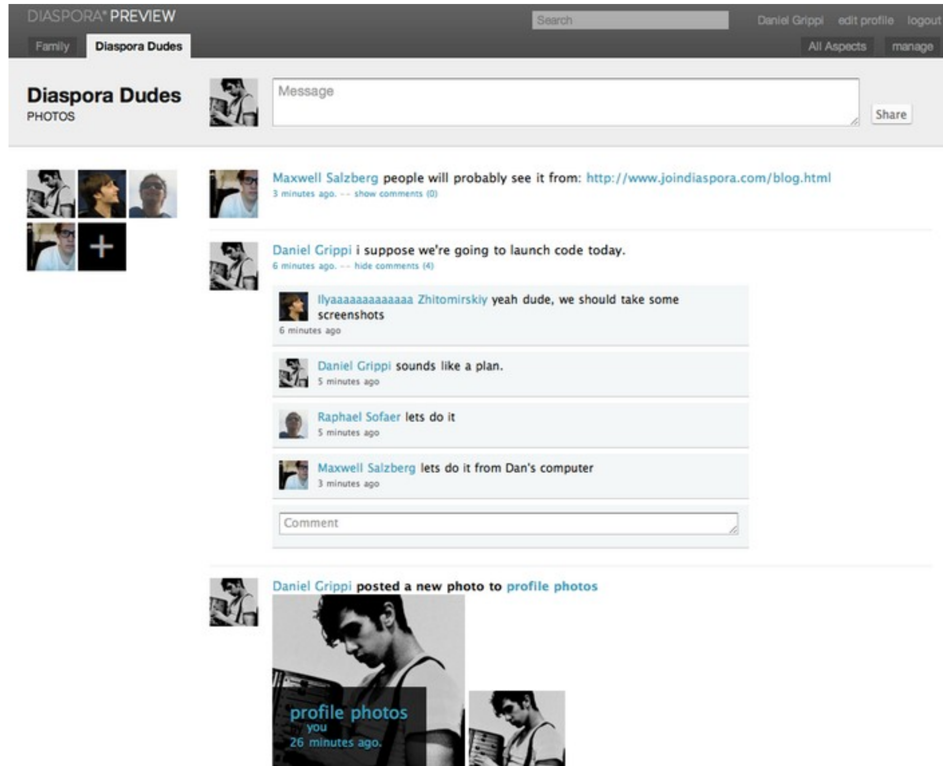


Figure 2.6: Diaspora* preview [28]

FSN buddycloud as a whole will be further discussed in chapter 4.

Other projects

There are also other projects more or less involved in the process of building FSN we did not mention like a french multi-purpose communication tool Salut à Toi [31] or Kune as an Apache Wave implementation [30].

However, many of them are either dead in development, or addresses FSN only partialy. Others does not take any part in standardization process, which makes it impossible for other implementations to be established, thus getting little attention from the side of developers.

2.4 Evaluation

What most of those mentioned protocols, standards, implementations have in common is a Web-based architecture. From a protocol perspective, they are designed to communicate over HTTP[3] protocol. Taking Web phenomenon into consideration, it is only reasonable that aforementioned technologies advanced this way.

However, several shortcomings arises for building FSN on top of Web technologies.

[3]RFC 2616 - http://www.ietf.org/rfc/rfc2616.txt

Non bi-directional communication Network topology of HTTP connections follows simple client-server model. The communication between client and server works on a request-response basis. After satisfying client's request HTTP connection is terminated.

For communication in federated environment it is desired to exchange information between server both ways instantly. In such a way network bandwidth can be conserved and additional state can be held naturally.

For HTTP server-to-server communication is not native and must be circumvented by higher level of application protocol abstraction.

Stateless connections By the means of client-server conversation HTTP provide stateless connection by design. In order to hold a state by HTTP client, e.g. for the purpose of preserving identity, Web browser cookies or URL parameters have to be used.

For the purpose of preserving identity, which is a fundamental element in FSN architecture, it is desired the communication protocol holds the state if required.

Lack of real-time We described the mechanism for event-driven content distribution provided by Pubsubhubbub [19] and Salmon [21] protocols. However they can only communicate between URL-based endpoints. By no means can data be pushed to an HTTP client. Because of this it is impossible to make almost real-time communication at protocol level, which is an important aspect of social network architecture.

Polling character In order to provide almost real-time experience HTTP client is forced to poll the server for the new updates. Whereas in most cases server's answer is simply "nothing new". As a result of this TCP connections has to be established frequently, which impacts network load and server resources.

As the large service like Twitter demonstrates, polling character of HTTP makes it difficult to scale for third party services [32] [33].

URL addressing scheme limitation Furthermore communication over HTTP is limited to the URL-based endpoint. As we pointed out in Section 2.3, when explaining Webfinger protocol, it is not natural to conceptualize identity as URL.

Again abstraction layer in form of standard like Webfinger has to be introduced to solve this.

Fragmentation Moreover, all those protocols we talked about completes different tasks separately. In terms of security, which is crucial in FSN system, it cannot be implemented vertically through all protocols at once, thus each one has to deal with it individually.

There are several standards for doing the same thing, e.g. OpenID and WebID for authentication. Implementing different subset of mentioned technologies may result in compatibility issues.

The point here is that the Web of today is too fragmented. It is not consistent enough to provide fundamental components of FSN.

Whenever you find yourself on the side of the majority, it is time to pause and reflect.

Mark Twain

It seems as a right approach to reuse existing Web infrastructure, but from architectual point of view it is not satisfactory for FSN. To move forward, the barriers of pervasive omnipresent Web phenomenon need to be left behind. FSN requires more integrated network approach.

Some projects like Diaspora are making its own path to establish social network for independent providers. Although it solves centralization problem, it does not follow the principle of federation.

On the other hand projects like DSNP are trying to build a whole new application layer protocol for FSN. It is a reasonable approach, however, inventing a new network standard is quite a challenging task. It is a long and living process, which requires not one but several implementations that put the protocol to the test.

To conclude, there is no need to build a new TCP/IP protocol or stack of protocols from the ground up. It is even undesirable, since there already exists suitable technologies to build upon.

3. XMPP – the golden mean

An ideal foundation for building a federation of social networks is *Extensible Messaging and Presence Protocol* (XMPP). It is also known as *Jabber* and it has been introduced by Jeremie Miller in 1999 [34], who was tired of using multiple clients for closed IM messaging services at that time. Since then a community of developers started to emerge and build Jabber-compatible software. Later on in a process of development protocol has undergone rigorous public review within *Internet Engineering Task Force* (IETF) and published as *Request For Comments* (RFC 6120) [36]. After that it has found its applications in multiple areas. Also various prominent companies got interested about XMPP and used it for their own purposes. Today we know instant messaging services like Google Talk or Facebook Chat are based on XMPP.

There are numerous reasons why to use XMPP for FSN.

Low level XMPP is an application layer protocol within TCP/IP network layered model. Like HTTP it is based directly upon TCP transport protocol. For this reason it is freed from lower application protocol dependency compared to Web technologies.

Federated topology Network topology of XMPP network is federated by its very nature. XMPP specifies two types of network conversation, i.e. client-to-server (c2s) and server-to-server (s2s). Therefore it conforms to the federated network topology we explained in section Section 2.1.

Addressability Addressing scheme relies on generally employed *Domain Name System* (DNS). Anyone can set up its own XMPP server and join the federation.

Identity awareness In order to take a part in XMPP network one has to be registered within its home server. Upon connection a server verifies user's identity based on his credentials and initiates bi-directional communication. Since the XMPP conversation is always mediated by the home server, there is a guarantee of a veritable identity.

Extensible data format XMPP leverages XML data format for network conversation in general. For this reason it is easy to customize payload in order to satisfy the needs of variety of applications.

"Real-time" Due to event-driven charater of XMPP network communication it is possible to create almost real-time social network experience.

Security Integral part of core protocol specification is support for encryption and strong authentication. Significant deployments at some of the most security-conscious financial organizations and government agencies put XMPP security to the test and proved it [35].

Standardization In addition to the XMPP core specification [36] there is a possibility to create *XMPP Extension Protocol* (XEP). *XMPP Standards Foundation* (XSF), a nonprofit organization, has been established to take part in the process of XEPs standardization. As of now there are over two hundred XEPs dealing with diverse XMPP use-cases.

Community & proven XMPP yet exists for over 10 years. It has acquired large community of independent developers, involving companies, wide deployments with dozens of interoperable codebases and millions of end users.

Existing infrastructure XMPP is already broadly employed for instant messaging purposes. Greatest social network service provider like Google and Facebook employed XMPP long ago into Google Talk and Facebook Chat. This might be an important aspect for FSN propaganda.

3.1 Protocol basics

In this section we throw a light on basic principles of XMPP protocol.

XMPP communication is connection-oriented, which means it holds the state between connected endpoints. Because of that the whole communication can be event-driven. There is no need for the client to poll the server for the information like HTTP protocol requires. Server will push it to the client when that particular event arises.

It can significantly minimize server load and preserve network bandwidth, since it is cheaper to keep long-lived TCP connection rather than repeatedly TCP handshake. This is the fundamental difference from HTTP connections.

However it is not always possible to preserve long-lived TCP connection. Especially mobile device possess such incapability, while they are hopping from network to network as they move.

Every client has its home server and all of client's traffic is distributed between their home servers.

As we pointed out XMPP relies on DNS addressing scheme. Every user has unique identifier called *Jabber ID* (JID in short). Here is another similarity to the familiar e-mail system. JID in its bare form look very much like an email address(e.g. `user@example.com`).

Another substantial advantage of XMPP is that it permits connection of multiple clients of the same user. One can use multiple devices with different availability states, capabilities, etc. In order to distinguish between those devices JID includes a *resource* identifier that follows the account address. For example user can have `user@example.com/home` JID for his home desktop computer client and `user@example.com/mobile` JID for his smartphone.

3.1.1 Communication primitives

XMPP is essentially a technology for streaming XML [35]. A session consists of XML streams in both directions, that take place upon long-lived TCP connec-

tion. The actual communication happens when client or server send XML chunk of data, which is interpreted by the other end. These chunks are called *stanzas* and they constitutes basic units in XMPP communication. All data that is send through XMPP network is wrapped into stanza.

There are three types, i.e. `<message/>`, `<presence/>` and `<iq/>`. Each of them behaves distinctively and are used for different purposes. We take a brief look at each one in the following text.

Message

The simplest stanza is `<message/>`. It is used for pushing data through the XMPP network. Servers treat messages as some kind of packets that need to be delivered from one end to another. They don't do deep investigation of stanza, they just take a look at a `to` parameter and forward message to its destination. In IP analogy messages are routed as packets and XMPP server acts as a routers for them. Here a simple example of message stanza:

```
<message from="harry@hogwarts.wiz/desktop_client"
        to="ron@hogwarts.wiz"
        type="normal">
  [... any kind of~XML payload ...]
</message>
```

When a target entity is offline, message is held by its home server in *offline message queue* until the recipient comes online again.

Presence

A fundamental element in XMPP world is *presence*. Presence stanzas are used to advertise availability of entity on the network. One can subscribe to other's presence and get notified every time the presence changes.

For example, if an entity comes online it sends presence to its home server. In the shortest form presence can look like this:

```
<presence/>
```

Server is then responsible to broadcast this presence to all subscribed nodes. Here we gets to another aspect of protocol, which is XMPP contact list also called as a *roster*. Any entity has a roster stored on its home server. Every item in a list consists of JID and the state of its subscription. In order to broadcast presence of an entity server simply look up its roster.

IQ

Info/Query stanza provide a request-response mechanism similar to GET, POST, PUT methods from HTTP. Let's illustrate this on an example. A client typically request a roster after it negotiates a connection with its home server. In order to do that it sends an IQ stanza of type `get`.

```
<iq from="harry@hogwarts.wiz/social_client"
    to="hogwarts.wiz"
    type="get"
```

```
    id="w2f3">
  <query xmlns="jabber:iq:roster"/>
</iq>
```

A server will then respond with `<iq/>` of type `result`.

```
<iq from="hogwarts.wiz"
    to="harry@hogwarts.wiz/social_client"
    type="result"
    id="w2f3">
  <query xmlns="jabber:iq:roster">
    <item jid="ron@hogwarts.wiz" subscription="both"/>
    <item jid="hermione@hogwarts.wiz" subscription="both"/>
  </query>
</iq>
```

Notice response's `id` parameter. Server include the same `id` as was in the request so the client can pair request with response. This is important for the asynchronic character of XMPP communication.

In comparison with `<message/>` and `<presence/>` stanza, an `<iq/>` stanza must be acknowledged every time.

3.2 Components

Another feature of many XMPP server is the ability to attach components to itself as defined in *XEP-0114: Jabber Component Protocol*. Components can be deemed as a separate software running within the XMPP server machine and providing additional functionality.

In order for the component to be attached to the server, it needs to authenticate itself to avoid server exploitation. For this purpose a shared key must be set on both ends. Whole communication between component and server happens upon an *XML stream*.

When component successfully connects to the server it obtains configured subdomain. This serves as an identifier for XMPP communication. Let's consider we have an XMPP server running on `example.com` domain and three components connected to server. Figure 3.1 shows such installation with MUC[1] conference, SMS gate service and channel service components. A stanza from a client or a remote server addressed to `sms.example.com` will be delivered right to that component. The server only mediates the XMPP communiacation. Note that subdomain is known only to the server, it does not have to do anything with DNS space.

By design components runs autonomously and can be attached to any XMPP server. This makes them independent from a server codebase. By leveraging component architecture many XMPP exentension protocol can be implemented and proven without touching server codebase.

Disadvatages Higher abstraction of XMPP and protocol core features, although they are reasonable, have taken their toll. XMPP compared to HTTP is

[1] XEP-0045: Multi-User Chat

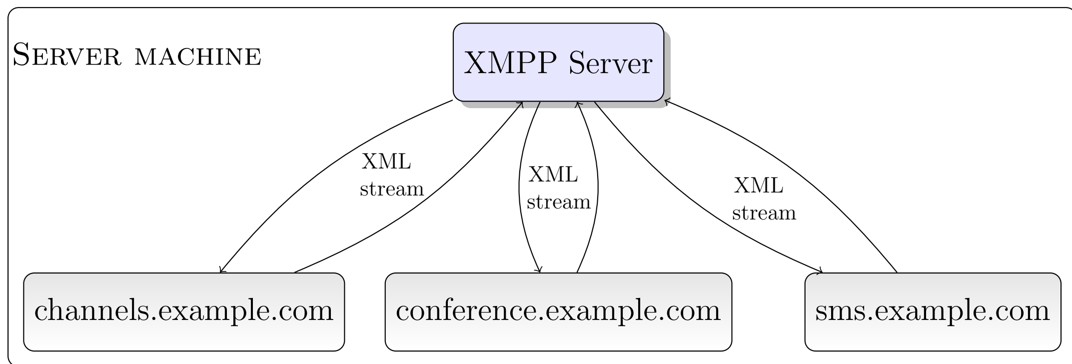


Figure 3.1: XMPP server and components

rather complicated, which has several adverse implications. For its complexity it is unattractive for developers, therefore little software is created. It is usually known only for its instant messaging features, thus it is used just for such intentions, exceptionally as a proprietary solution for company's purposes.

Conclusion For its robustness XMPP provides stepping stone for a wide range of applications. It is able not only to meet social networking requirements of today, but also to extend them by any means. buddycloud protocol employs the use of XMPP in a way that fits the praxis of present social networks.

4. buddycloud

buddycloud originaly started as a pure mobile project founded by entrepreneur and technologist Simon Tennant.

The philosophy of buddycloud project is to build simple and user-friendly product for people to share, discover and communicate whatsoever. Focus is revolving around user needs in the first place. While trying to keep things simple question of "What would Simon's mom do?" became a catch phrase among buddycloud developers [39].

The approach for building FSN here is to design the product first and then come back to the protocol specific stuff. Protocol should not do too many assumption as it can lead to useless complexity.

However the overall architecture is also very important. As a result of designing protocol on top of XMPP standard lot of FSN fundamental questions were solved right away. In order to provide a seed for healthy FSN ecosystem buddycloud is primarily focusing on the sole core protocol and good reference implementations throughout rather than secondary peripheral stuff.

We already outlined the benefits of building FSN on top of XMPP protocol. In this chapter we are going to explain how buddycloud leverages the XMPP core and its extensions.

4.1 Channels

Basic interaction model of buddycloud is elementary as it was intended in the beginning. In the center of content creation stands the concept of *channels*. We can look upon channels as a feed of rich content such as simple mood or status messages, short tweet or long blog posts, additional meta information like geographical location, user's connections or online activity in general. buddycloud presently introduces two types of channels, i.e. a *personal* channel and a *topic* channel.

Personal channel

The purpose of a this channel is to provide a personal online place for an individual to share content with others in general. The user has a full control over what and to whom data is shared. Main content type in buddycloud channel model is called simply a *post*. There are no particular length limits for the post, it can either be long as a blog post or short as a tweet[1].

There are two categories of posts available in a channel, i.e. a *topic* and a *comment*. A simple conversation can be made by binding the two together as illustrated in Figure 4.1.

Besides posts, the channel model currently provides a way to share *location*, simple *status* messages and a list of channel *followers*.

[1] tweet - a Twitter's main content entry, it is limited to 140 characters

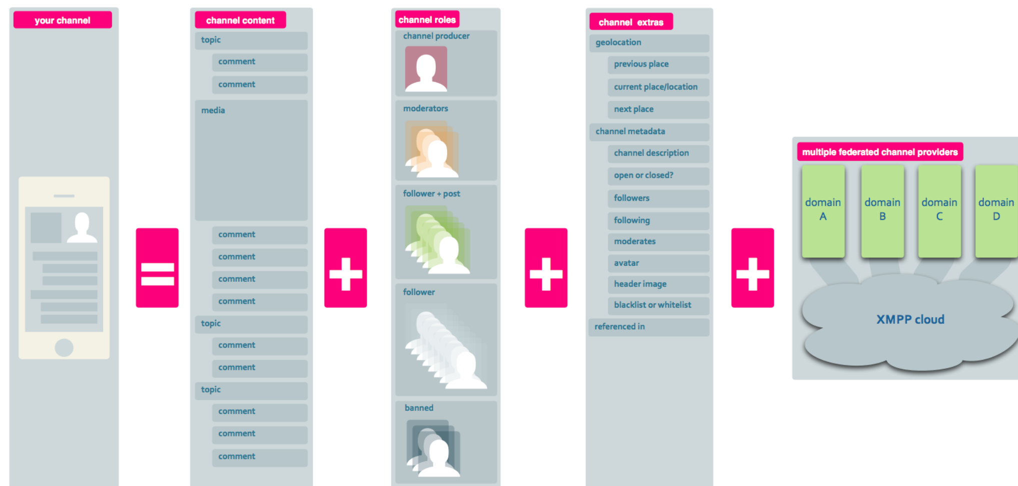


Figure 4.1: Expression of buddycloud channel [42]

Topic channel

On the contrary to the personal channel, topic channel is intended just for a particular subject of discussion. Therefore, it lacks irrelevant location features.

4.1.1 Privacy settings

Because of privacy settings being the stumbling-block of present online social networks, we breach the subject of channel privacy first before we dig into any protocol specifics. buddycloud enables users to set their channels either

- **open** or
- **private**.

An open setting makes the channel visible for anyone and for everyone to subscribe to. This also makes channels discoverable and its content crawled by search engine like the one we describe at the end of this chapter.

On the other hand, it may not be desired for the channel to be publicly accessed. Ergo user has the ability to restrict it to a *whitelist*. To follow such a channel an authorization request has to be approved by the owner or a channel moderator.

In order to get a clear control over user access buddycloud defines five *roles* for a channel described in Table 4.1. It is apparent that the scope of user abilities is satisfactory enough. From the owner of the channel who has the ultimate control over it, down to banned users that are ignored and have no rights at all.

4.2 Channel protocol

As we pointed out channel data model enables users to follow and/or publish. But what it actually means to follow a channel? From the protocol perspective it is more correct to identify follower as a *subscriber*. In order to read content from channel one has to be subscribed to it.

User role	Reported as	Capabilities
producer	owner	ultimate control
moderator	moderator	read, publish, remove posts, approve new followers
follower+post	publisher	read, publish posts
follower	subscriber	read posts only
banned	outcast	no access, new follow requests are ignored

Table 4.1: Channel roles

In Section 2.3 we described Pubsubhubbub protocol implementing publish-subscribe pattern for content distribution on the Web. For channel purposes we need something similar, however, on top of XMPP. There is no need to design XEP from ground up. We have a fairly known extension, called *Publish-Subscribe*, which fits to our needs precisely. In the next section we will briefly introduce this extension while being the cornerstone for content distribution in buddycloud.

4.2.1 XEP-0060: Publish-subscribe

Publish-subscribe, shortly called Pubsub, is a fairly known XMPP extension defined in XEP-0060 [37]. It provides functionality for publish-subscribe pattern over XMPP network. Pubsub has widely used implementations in the field of micro-blogging services.

Although the Pubsub specification is quite long, the basic idea behind it is very simple. Pubsub service resides on a server machine either as an XMPP component or built in right into the server codebase. User's content is stored in so-called *nodes*. There are publishers feeding the node on one side and subscribers receiving content updates on the other. When an entity publishes data to the node the Pubsub service pushes an event notification to all of the node's subscribers.

We outline a sample data flow for 'Potions class' node owned by JID `harry@hogwarts.wiz`. The publisher sends `<iq type="set"/>` stanza to pubsub service declaring node and filling desired content.

```
<iq from="harry@hogwarts.wiz/social_client"
    to="pubsub.hogwarts.wiz"
    type="set" id="pub1">
  <pubsub xmlns="http://jabber.org/protocol/pubsub">
    <publish node="Potions class">
      <item>
        [ ... DATA ... ]
      </item>
    </publish>
  </pubsub>
</iq>
```

Pubsub service is then responsible to push event notifications to all subscribed entities. In our example these are `ron@hogwarts.wiz` and `hermione@hogwarts.wiz`.

```
<message from="pubsub.hogwarts.wiz"
```

```
        to="ron@hogwarts.wiz" id="foo">
  <event xmlns="http://jabber.org/protocol/pubsub#event">
    <items node="Potions class">
      <item id="ae890ac52d0df67ed7cfdf51b644e901">
        [ ... DATA ... ]
      </item>
    </items>
  </event>
</message>

<message from="pubsub.hogwarts.wiz"
        to="hermione@hogwarts.wiz" id="bar">
  <event xmlns="http://jabber.org/protocol/pubsub#event">
    <items node="Potions class">
      <item id="ae890ac52d0df67ed7cfdf51b644e901">
        [ ... DATA ... ]
      </item>
    </items>
  </event>
</message>
```

In the terms of node access control Pubsub defines a set of affiliations that can be configured on per-JID basis. With the exception of the moderator role, Pubsub affiliations correspond with channel roles as we indicated in the **Reported as** column in Figure 4.1. In order to distribute the burden of maintaining the channel among several users, it is necessary to have kind of moderator role, e.g. capable to approve new subscriptions or remove spam posts. Inasmuch as only owner posses the ability to approve new subscribers in Pubsub, buddycloud Channel protocol introduces a moderator role.

Pubsub specification is rather extensive and covers many use-cases, that are out of scope of buddycloud channels. To preserve simplicity of a channel, buddycloud uses only a subset of features defined in XEP-0060 [37].

4.3 Content format

As well as for the data format buddycloud is trying to reuse as many existing standards as possible. To express channel post in the most conventional way it uses Atom Syndication Format, which we have already described in Section 2.3.

```
<entry xmlns="http://www.w3.org/2005/Atom"
      xmlns:activity="http://activitystrea.ms/spec/1.0/"
      xmlns:buddycloud="http://buddycloud.com/atom-elements-0">
  <published>2010-01-06T21:41:32Z</published>
  <author>
    <name>Harry</name>
    <buddyclud:jid>harry@hogwarts.com</buddycloud:jid>
  </author>
  <content type="text">I am a~Half-blood Prince</content>
```

```
  <geoloc xmlns="http://jabber.org/protocol/geoloc">
    <text>Number 12, Grimmauld Place, England</text>
    <locality>London</locality>
    <country>Great Britain</country>
  </geoloc>
  <id>/user/harry@hogwarts.com/posts:1291048772456</id>
  <activity:verb>post</activity:verb>
  <activity:object>
    <activity:object-type>note</activity:object-type>
  </activity:object>
</entry>
```

Listing 4.1: buddycloud topic post

Content format is rather self-explanatory as Listing 4.1 shows. We can notice four different XML namespaces here.

- `http://www.w3.org/2005/Atom` - Atom Syndication Format
- `http://buddycloud.com/atom-elements-0` - buddycloud
- `http://jabber.org/protocol/geoloc` - XEP-0080: User Location
- `http://activitystrea.ms/spec/1.0/` - Activity Streams

Worth noting is namespace of Activity Streams standard, which we described in Section 2.3. The vocabulary defined in this standard is used to declare actions, that were performed with a channel content. Currently there is only single verb *post* supported by Channel protocol.

In order to distinguish between topic and comment post, Activity Streams object type of *note* and *comment* is used respectively. It is also required that a comment post includes `<in-reply-to>` element with an ID of a specific topic post. Otherwise the comment cannot be tracked, thus will not be visible in the channel. An example of comment post is shown in Listing 4.2.

```
<entry xmlns="http://www.w3.org/2005/Atom"
       xmlns:thr="http://purl.org/syndication/thread/1.0"
       xmlns:buddycloud="http://buddycloud.com/atom-elements-0">
  <author>
    <name>Ron</name>
    <buddycloud:jid>ron@hogwarts.wiz</buddycloud:jid>
  </author>
  <content type="text">
    Whatever!
  </content>
  <published>2010-11-29T16:40:10Z</published>
  <updated>2010-11-29T16:40:10Z</updated>
  <id>/user/harry@hogwarts.com/posts:1291048810046</id>
  <geoloc xmlns="http://jabber.org/protocol/geoloc">
    <text>On the~quidditch pitch</text>
    <locality>Hogwarts</locality>
    <country>England</country>
```

```
  </geoloc>
  <thr:in-reply-to ref="1291048772456"/>
</entry>
```

Listing 4.2: buddycloud comment post

If we look at the **ref** attribute closely, we can see that Ron actually commented on Harry's previous topic post.

4.4 Operations

The first step of using channels is to create one, naturally. The prerequisite for this is that a provider maintains XMPP server with a channel service component. We also assume that a user has an existing XMPP account, but is not acquainted with buddycloud FSN yet. In this section we will describe particular actions, that can be performed on a channel.

4.4.1 Channel service discovery

For number of reasons we described in Section 3.2 buddycloud channel service implementation resides in an XMPP component alongside XMPP server. To avoid naming collisions for component subdomain, e.g. `channels.example.com`, the decision is left to specific server configuration.

For the component to be discoverable it has to advertise itself. *Service Discovery* mechanism defined int XEP-0030 [38] is then used to find channel service.

The algorithm is following. In the first place user's client initiate standard XMPP connection with its home server. Using Service Discovery protocol client queries server for its capabilities.

```
<iq from="harry@wizards.wiz"
    to="wizards.wiz"
    type="get" id="req1">
  <query xmlns="http://jabber.org/protocol/disco#items"/>
</iq>
```

Server answers with the following exemplary output.

```
<iq type="result" id="req1"
    to="harry@wizads.wiz"
    from="wizards.wiz">
  <query xmlns="http://jabber.org/protocol/disco#items">
    <item jid="mediaserver.wizards.wiz"/>
    <item jid="channels.wizards.wiz"/>
    <item jid="search.wizards.wiz"/>
  </query>
</iq>
```

Consequently client has to query each item for its information. Note the different `to` parameter and `info` XML namespace.

```
<iq from="harry@hogwarts.wiz"
    to="channels.hogwarts.wiz"
```

```
    type="get" id="query1">
  <query xmlns="http://jabber.org/protocol/disco#info"/>
</iq>
```

Channel service is then recognized by `<identity/>` element with following attributes.

```
<iq type="result" id="query1"
    to="harry@wizards.wiz"
    from="channels.wizards.wiz">
<query xmlns="http://jabber.org/protocol/disco#info">
  ...
  <identity type="channels" category="pubsub"/>
  ...
</query>
</iq>
```

4.4.2 Creating and controlling channel

To create a channel, one has to register within the channel service. In practice registration can be performed by service custom interface, i.e. Web registration form, or directly between XMPP client and channel component. This is done by *In-Band Registration* extension defined in XEP-0077. For this to work component must advertise this feature by Service Discovery.

```
...
<feature var="jabber:iq:register"/>
...
```

By simply sending following stanza, channel service performs all actions required for channel creation.

```
<iq from="user@example.com/ChannelCompatibleClient"
    to="channels.example.com" type='set'>
  <query xmlns='jabber:iq:register'/>
</iq>
```

This process should be done transparently by the client's software, so the user is not burdened with any registration forms.

Nodes

From the perspective of Pubsub, channel constitutes a set of Pubsub nodes, which serve as an elementary storage for content. The naming convention for channel nodes is `/user/user@example.com/name`. Upon channel registration following Pubsub nodes are created.

- `/users/user@example.com/posts`
- `/users/user@example.com/status`
- `/users/user@example.com/geo/current`

- /users/user@example.com/geo/previous
- /users/user@example.com/geo/next
- /users/user@example.com/subscriptions

From now on we will be discussing in terms of a specific node rather than channel in general. Although we will not discuss *geo* nodes, i.e. `geo/current`, `geo/next`, `geo/previous`, because there is unfinished logic around them.

Posts node

Posts node is basically used to store user's general content. We can look upon it as a kind of a channel representative node. Table 4.2 show properties that can be configured for posts node. Important security attribute is `pubsub#access_model`.

Property name	**Field**	**Options**	**Defaults**
Title	`pubsub#title`	any XML string	[jid] Channel Posts
Description	`pubsub#description`	any XML string	A buddycloud channel
Access model	`pubsub#access_model`	- open - whitelist	open
Publish model	`pubsub#publish_model`	- subscribers - publishers	subscribers
Default affiliation	`buddycloud#default_affiliation`	- publisher - subscriber	publisher (follower+post)
Type	`buddycloud#channel_type`	- personal - topic	-
Creation date	`pubsub#creation_date`	ISO 8601 time format	-

Table 4.2: Channel properties

By setting node access model to *whitelist* no items can be retrieved until subscription request is approved by channel owner. On the other hand, *open* access model subscribe JID on request automatically and also makes it possible to retrieve posts items regardless of JID subscription.

While access model restricts node reading capabilities, publish model is used to limit user ability to write into the node. Channel protocol makes use of the two entered options, i.e. *subscribers* and *publishers*. Apparently, subscribers setting allows all of the subscribed entities to publish content to the node, whereas publishers restricts this ability just to the respective role. Typically channel follower should be permitted to interact with channel content, therefore subscribers publish model is required for posts node.

As we can see protocol logic adds to standard Pubsub XEP properties, i.e. *channel type* and *default affiliation*. For setting and resetting properties standard Pubsub `<configure/>` construct is used.

Status node

Status node provides a place for publishing user's mood status messages. On the contrary to posts node, other connection can subscribe to the node, however only publisher role has the right to write status messages. This is restricted by the publishers publish model setting.

Subscriptions node

Important aspect of social networks is the possibility to traverse the list of user's connections or channel followers in case of buddycloud. There is no way to do this by Pubsub protocol unless user is also the owner of a node. For this reason a *subscriptions* node has been proposed.

It is a channel component's responsibility to fill the node according to user's subscribing activity. Each subscription item includes ID of a node and channel component JID, e.g. Harry's subscriptions would contain items like one in the following listing.

```
...
<item jid="channels.hogwarts.wiz"
      node="/user/harry@hogwarts.wiz/posts"
      name="ron@hogwarts.wiz"/>
...
```

On the contrary to other channel nodes, subscription item payload conforms to XEP-0030: Service Discovery item, therefore the protocol is also used for item listings.

Above all, subscriptions node can also be used by client in order to avoid unnecessary steps to discover remote channel component.

4.4.3 (Un)subscribing

To follow someone's channel means to be subscribed to it. However we are actually subscribing to channel's Pubsub nodes. Again Pubsub mechanism for subscribing is used.

```
<iq type="set"
    from="harry@wizards.wiz"
    to="channels.wizards.wiz"
    id="sub1">
  <pubsub xmlns="http://jabber.org/protocol/pubsub">
    <subscribe node="/user/dumbledore@wizards.wiz/posts"
               jid="harry@wizards.wiz"/>
  </pubsub>
</iq>
```

Based on what sort of access model node is set, user receives notice with subscription state. In case of whitelist access model subscription keeps the state of pending until the channel owner approves it.

Unfollowing the channel is done very much like subscribing with the exception that client sends `<unsubscribe/>` element instead. On the contrary to subscription request there is no reason to approve user's will to be unsubscribed, thus

the action performs outright.

4.5 Current project overview

As of now Channel protocol is in development stage, and is still evolving, thus changes can be made to it until it becomes a XEP standard. All of the project work is tracked at rich wiki Web site, buddycloud.org [41], together with draft Channel protocol specification [40]. Center of protocol and development discussion is located at `buddycloud-dev` mailing list provided by Google Groups service.

There are also other buddycloud sub-projects and implementations on the way, dealing with various FSN aspects or social networking in general.

Channel component codebases

Reference implementation for channel component or as usually called *buddycloud server* is built in Node.js[2]. Work has been commenced also on other channel server implementations in Java and Python programming language.

Bridging with Web

Project is now mainly focusing on integration with Web services in order to address the majority of online users and developers. For this reason buddycloud Web client and HTTP API server are being developed. To that end it is important to point out the contrast between TCP connections of HTTP and XMPP we discussed earlier in this project. An XMPP extension, *Bidirectional-streams Over Synchronous HTTP* protocol defined in XEP-0124, is used for emulating XMPP connection over HTTP.

Channel directory & search component

As in any content system, significance of search capabilities in FSN is obvious. It is always convenient to welcome new user with display full of nearby, similar, or otherwise interesting channels. In contrast with centralized platform providing search capabilities in federated enviroment can be quite a challenging task.

For this purpose buddycloud directory & search component was created. It is able to crawl nearby content and search within channel metadata and channel content. Consequently, it enables users to search for similar channels and make recommendations based on what channels user follows.

Media sharing

Since XMPP is an XML text streaming protocol, it falls short of binary data sharing like multimedia. To deal with such an important aspect of FSN, buddycloud came up with *media server*. It is supposed to provide seamless sharing of

[2]Node.js is a platform, which uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices. (http://nodejs.org/)

media with other channel followers and also the ability to archive them.

In conclusion, buddycloud project is growing tremendously in many directions in pursue to build FSN. It is open and seeking for developers in every area, whether it is protocol stuff, codebase development or UI designs. Among list of alternatives it appears as a highly promising solution for FSN platform for its openness and overall architecture.

5. SocialDesktopClient

Purpose of a software engineering part of this project is to bring contribution to the buddycloud and FSN in general. buddycloud community has already started development on console client, web client, and mobile clients for iPhone and Android. To fill the gap we decided to implement desktop client.

Apart from this, objectives of our software exceed buddycloud project. The application might be designated for other social network services too. In this way we want to address more people and by creating support for buddycloud in the first place we could manifest the principles of FSN in general. For this reason application earns rather general name – *SocialDesktopClient*.

Before we begin it is desired to define a set of requirements or axioms that our software development is going to conform. It is immensely important to define such axioms, as it determines the purpose and the foundation of the software. Having them in mind will also create barriers that should not be overcome during the development. In the following text we are going to describe these axioms, explain why are they significant for FSN and shortly discuss how are they related to the current state of the project.

> Requirements are like water.
> They're easier to build on when
> they're frozen
>
> Anonymous

Openness

Vital aspect of this project is to bring code to the community, so it can be evaluated and extended. This of course implies a free distribution of the software.

SocialDesktopClient is issued under the terms of General Public License, version 3. The project is available online as a Git repository situated in public repository space at the following URL:

`https://github.com/maros-/SocialDesktopClient`

Modularity

To conform to variety of social networking services out there, client application should be easy to extend and add new functionality. Therefore it should provide simple interface for potential developers, so they can leave common client implementation details behind and focus on particular social service from the beginning.

Current version of SocialDesktopClient offers an interface for an implementation of other social network services in the form of shared [1] libraries, in context

[1]shared library – *.so file on Linux platforms, dynamic library - *.dll file on Windows platforms

of our design commonly called plugins. In addition to this the Core of the application is designed independently on UI framework. Main motivation was to take the overhead away from core client components.

Cross-platform

In order to address as many users as possible it is desired for the application to support multiple operating systems.

SocialDesktopClient place emphasis on the use of cross-platform code and libraries. However in the current stage of development it is only tested on Linux platform.

Desktop environment

In terms of desktop environment we mean a non-browser always-on application running withing desktop operating system. Although the motivation for the desktop client comes from the buddycloud project, there are other reasons for this too.

Intention with the SocialDesktopClient is to exploit desktop resources as much as possible and not being limited by browser capabilities. It is also vital for the FSN to have an always-on desktop client.

Among many existing desktop clients, none conforms to all of our requirements. They are either platform-specific or proprietary, thus closed-source, e.g. popular ones Tweetdeck, Seesmic. On the other hand there are also several decent open-source XMPP clients, however, they are designed as an instant messengers, not a social network service clients. As a result we decided to build one from scratch.

5.1 Implementation and preliminaries

Programming language decision Since the purpose of this application is to work in a desktop environment we want the application to run natively on a platform. Our intention is to let programmer have the full control over desktop resources and avoid being dependent on third party runtime. For the SocialDesktopClient implementation we choose to employ C++ programming language.

GUI framework Decision for C++ programming language also relates with the choice of GUI[2] tools for the program. There is not many options in the field of cross-platform GUI frameworks. As a result for many advantages over different existing GUI tools, we choose to use fairly known *Qt framework*. It has a quality documentation, lots of exemplary source code and the community of developers behind it. It not only aims on GUI implementation but it also provides tools for the work with Multimedia, Databases, Networking, Web applications, etc. To that end it may be beneficial for the SocialDesktopClient future development.

[2]Graphical User Interface

Boost Libreries While keeping the application core independent on used GUI framework certain implementation questions like multi-threading, filesystem access, data serialization had to be addressed. It would have been a needless effort to redevelop these common paradigms as long as they are available in many frameworks and libraries. In order to minimize the distance from C++ standard library the decision was made to use *Boost*.

Boost libraries are constantly developed, close to the standards, as well as many parts of present C++ standard actually transfered from Boost. They provide variety of low-level functionality, good documentations and above all are cross-platform, which is a must for our software.

XMPP library For the XMPP protocol side of implementation we choose an appropriate library for C++. Preconditions were a satisfactory documentation, support for wide range XMPP features, continuous development and support. Among several options, some libraries were dead in development, some supported only a little XMPP features and many poorly documented. The decision was finally made to use *Swiften XMPP library* [44].

It has wide range of features, especially a full XMPP core protocol [36] implementation. Except library itself, there is a Swift IM client in development built on top of Swiften. This would not only gives us an exemplary use of library interface but also motivates Swiften developers to keep library supported. Moreover, library source code is rather self-explanatory.

Conventions From the beginning we choose to follow conventions according to *Google C++ Style Guide* [43]. However in the process of development it appeared bothersome to follow some of the conventions strictly. In the end, we had to also conform them to the conventions of Qt framework and Swiften XMPP library, which are extensively used in this project.

In terms of code commenting we employed Doxygen as a widely-used tool for auto-generated documentation. Therefore all source code comments are conformed to Doxygen redundant vocabulary and commenting guidelines.

IDE During the work on SocialDesktopClient it has been shown that it is important to choose appropriate development environment. In the beginning a choice has been made to leverage Eclipse IDE with CDT extension and Qt Integration plugin. There is a great community behind Eclipse, IDE is universal and a lot of features can be added by easily attaching plugins from Eclipse Marketplace. C++ language integration has been provided by CDT extension to Eclipse. For the Qt project manager and GUI elements designer comes also with the easy Eclipse plugin integration. But the cost of universality has taken its toll. After a project increases on its size Eclipse IDE slows down accordingly.

We had to look for alternatives, where there was only one left with Qt integration running on Linux platforms. Qt creator shows as the most suitable IDE. In contrast with Eclipse, which is a Java-based IDE, Qt Creator runs natively in operating system. The increase of IDE speed was apparent. Qt Creator proves to be the most appropriate IDE for SocialDesktopClient development, despite the early prejudice.

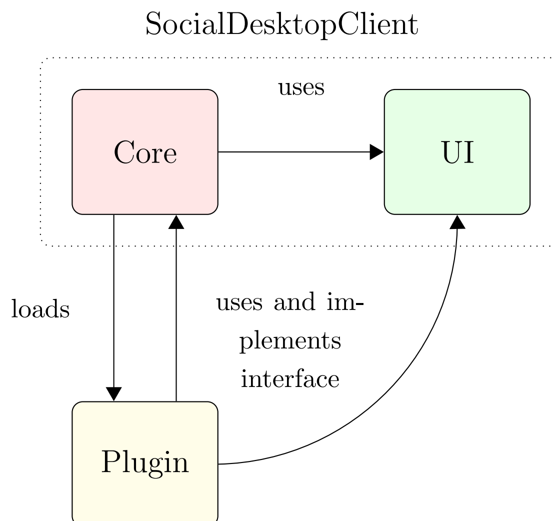


Figure 5.1: SocialDesktopClient top design perspective

5.2 Architecture

From the top architectural point of view we differentiated three general parts of application.

1. **Core**
2. **UI**
3. **Plugin**

Figure 5.1 shows how they relate with each other

5.2.1 Threading model

Design of SocialDesktopClient should allow to run multiple social service simultaneously. This requires to plan a threading model for service network connections. For this, two network handling models can be considered, i.e. *synchronous* or *asynchronous*.

While using synchronous connections for each service there is no need to create additional application threads. All connections would be queued and upon the start of each connection, thread would be blocked until a response came from network. But what if the connection protocol required to be kept for a longer period or even for the whole application run. One example of such protocol is XMPP, which runs upon long-lived TCP connection. In such a case the synchronous network model comes short.

In order to give the service plugin implementer choice for asynchronous network connections each service instance has to reside in its own separate thread. In addition also the Core needs its own thread in order to provide thread safe access to common core shared data. Taking into consideration also the GUI threads, overall threading model presented in Figure 5.2 is used for SocialDesktopClient.

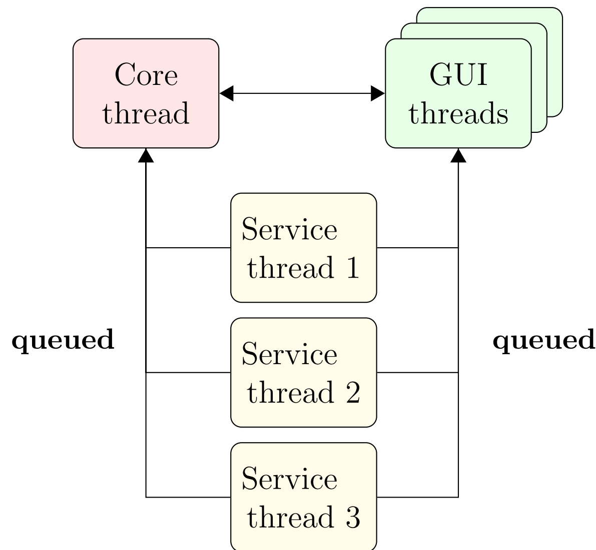


Figure 5.2: Threading model

5.2.2 Core

We can look upon Core as a heart of SocialDesktopClient. It is responsible for initiating and terminating main application components. From the source code perspective, it can be said that Core is everything except for UI and plugins.

According to the specific competences the Core is divided into several components. We call them *managers*. Implementation for each manager resides in separate class inherited from `AbstractManager`.

Each component provides specific functionality as we describe them in next sections together with other significant Core aspects.

DataManager

The role of data manager is to provide a storage for the application. Currently it offers an interface for the work with program configuration and service account data.

The configuration is stored in an XML document, `sdc.xml`, located in root directory together with the application executable. The serialization process for XML is done by Boost Serialization library. For the filesystem, Boost Filesystem library was used.

PluginManager

An entry point for application plugins is coordinated by PluginManager. From the operating system perspective, plugin is nothing but a *shared library* (or dynamic library on Windows platform) file. In order for the plugin to be loaded it has to be situated in `./plugins` directory relative to the executable.

There is no pretty way in standard C++ to attach shared library and work with classes implementations right away in the program runtime. While trying to preserve independence from other frameworks low-level C functions need to be used. At program startup PluginManager tries to load available shared libraries

and through `dlsym()` function interface looks up for `registerLibrary` symbol, which represents respective C function in actual plugin library. Function signature has to match beforehand in order to return a `Registration` object holding required plugin information[3].

PluginManager currently provides interface for the `Service` class plugin implementation, which holds actual social network service implementation. However it is very simple to add new pluginable type for future program needs [4].

EventManger

EventManager job is to provide a thread-safe access for shared core data within whole application. It is the EventManager's event loop that make the Core thread alive.

In order to access shared data an event callback is pushed back to the event loop queue. Events in the loop are processed sequentially, therefore the access from multiple threads is synchronized and shared data are secured against race conditions.

Accounts

Another Core responsibility is to manage specific service accounts. All common necessary data are encapsulated in `Account` type[5]. Most important account information include *UID*, a unique user ID, *password*, *service signature* to determine particular social service and *enabled flag* determining the activated state of an account.

On application exit DataManager stores accounts in `sdc.xml` file.

Core singleton

Essential part of SocialDesktopClient we did not mention yet is the Core class. It mixes business logic of all mentioned SocialDesktopClient components together.

During the application runtime it actually behaves as a *singleton*[6]. Core instance can be access by a call to `sdc::Core::Instance()`.

Within the application `main` function core instance and GUI specific implementation is initialized[7]. This is the point where actual specific GUI implementation and Core object binds together.

In order to start the application simply call to `sdc::Core::Instance()`
`->Start()` is made.

Signals

Important aspect of the Core is its signaling interface. The idea behind signals is to provide a notification mechanism for external components. Signals can be also

[3]Implementation can be found in `plugins/buddycloud/main.cc` file

[4]see `PluginProvider` template class in `sdc/core/plugin_manager.h` file

[5]see `sdc/core/account.h` file

[6]Singleton is a software architectural design pattern restricting the instantiation of a certain class to the only single object

[7]see `main.cpp`

referred to as *publishers* or *event* in similar systems. By using signals object can emit specific signal on a particular event without the need to know the receiver of it. In order for this to work a *slot* callback has to be connected onto the signal. When object emits particular signal all connected slots execute.

This makes objects fully independent from external components, while still providing the means of communication. It is entirely up to external component whether to use the signal or not.

For the SocialDestkopClient, Boost Signals library is used extensively throughout whole application. In such a way business logic in the Core can be completely separated from UI specific implementation and still be able to provide event notification mechanism.

Moreover, this kind of separation makes it possible for the Core to be packed and used as a library in the future.

Services

Integral part of SocialDesktopClient is a social service implementation. Due to modular character of the Core it is possible to implement it in a plugin library.

At the top, there is the `Service` class encapsulating social service metadata and important factory methods for creating `ServiceModel` and `ServicePresenter` instances. The abstraction of latter two is taken from *Model-View-Presenter* design pattern.

Job of ServiceModel is to encapsulate entire social network service business logic such as connecting and disconnecting, synchronization, all sorts of service features, etc. As a model it acts solely upon itself. In other words it is not dependent on ServicePresenter or any of the Views. It uses signaling interface for firing notifications outside.

On the contrary, ServicePresenter is responsible to hide all the service presentation logic. It has direct access to the main UI elements such as main window and central account button. In order to get notified for ServiceModel events it connects to its signals.

Connecting signals between ServiceModel and ServicePresenter can be kind of a tricky thing. As for the application Core, ServiceModel is designed to be independent from any UI logic whatsoever. It means there are no UI classes or any UI framework parts involved at all.

The other thing is multi-threading. As we presented in Figure 5.2, service instances and GUI framework operates in different application threads. Single service thread in the latter Figure is an actual running instance of particular ServiceModel, while ServicePresenter instance works within a GUI thread. Regarding Qt GUI threads, there is a restriction, that class members can be accessed only on objects of same parenting thread. In order to access members on object from different thread, *Qt signal/slot queued connection* must be used, whereas it ensures a thread-safe access.

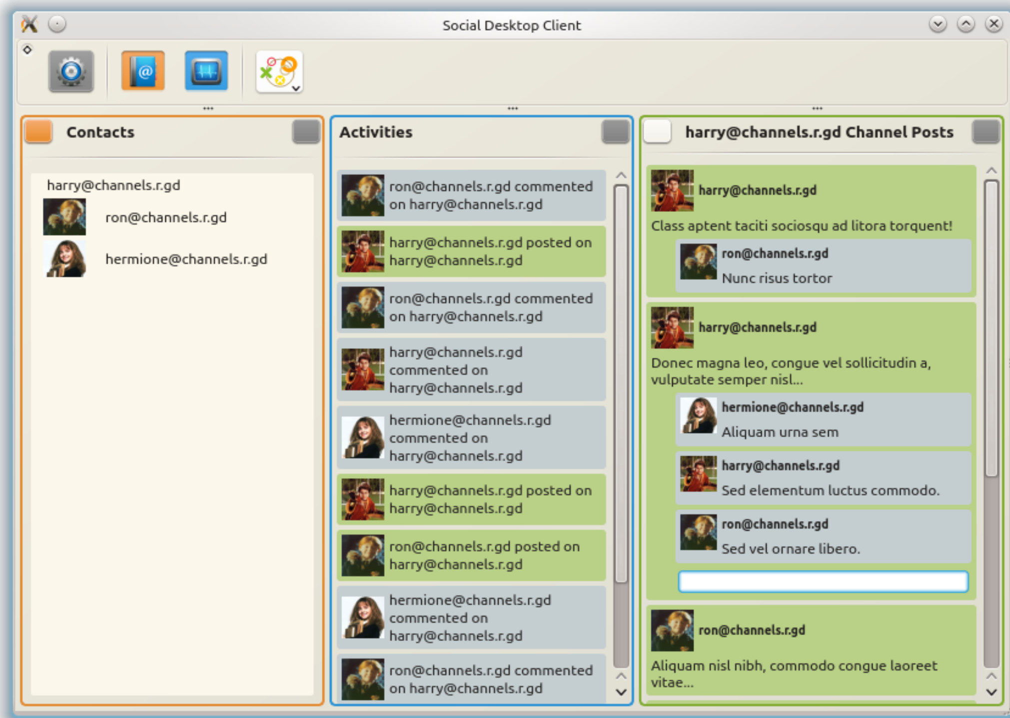


Figure 5.3: SocialDesktopClient preview

This leaves us with Boost Signals interface on the side of ServiceModel and Qt signal/slot mechanism on the side of ServicePresenter. There is no standard way to make a queued connection between them as Qt thread requires. For such a reason `bind()` function was introduced[8].

5.2.3 Qt GUI

In order to separate UI implementation from the Core an abstraction of `UI` class is introduced. SocialDesktopClient currently provides GUI implementation using Qt framework. It resides in `QtGui` class inherited from `UI` abstract type[9]. QtGui responsibility is to handle and process Core signals.

Now we look closer on the actual user interface. Taking it from the top there is a main window managed by a `MainWindow` class[10]. Figure 5.3 shows a preview of SocialDesktopClient main window.

There are two main elements in the view, i.e. top *control panel* and *content panels*. Job of control panel is evident. It provides main application control along with activated accounts buttons for the control of particular account.

More interesting is the concept of content panels[11]. There are two common always available panels in the view, i.e. *contacts panel* and *activities panel*[12].

[8]see `sdc/qtgui/bind.h` file
[9]see `sdc/qtgui/qtgui.h` file
[10]see `sdc/qtgui/main_window.h` file
[11]see `ContentPanel` class
[12]see `ContactsPanel` and `ActivitiesPanel` classes

Contacts panel is a common place for social network contacts from all accounts, while activities provide a room for displaying activities throughout all running social network services.

In the preview (Figure 5.3) we can see another panel on the right, which is actual service implementation of content panel. This way GUI enables plugin service developers to implement specific content display using the standard SocialDesktopClient content panel.

5.3 buddycloud plugin

Channel protocol plugin uses previously described service interface implemented in respective classes, i.e. `BcService`, `BcModel`, `BcPresenter`.

For the work with XMPP protocol, Swiften XMPP Library has been employed extensively. In order to work with new types of XMPP payloads, Swiften provides interface for *Parsers* and *Serializers*. Implementation for Channel protocol parsers and serializers resides in files `plugins/buddycloud/pubsub_parser.h` and `plugins/buddycloud/pubsub_serializer.h`.

An issue or rather inconveniece worth noting, which is related with Swiften library it the incapability to register XMPP account upon XML stream initiation as In-Band registration proposes.

Since XMPP communication is asynchronous by design, lot of handler function is required further on to process each type of XMPP stanza. Therefore it was decided to use new C++ standard *lambda functions* in order to make cleaner code.

5.4 Overview and future works

Detailed look at source code and particular implementation details can be found in auto-generated Doxygen documentation, which is included with attached CD (see Attachment C). In order to compile the application from the source, there is a procedure described in Attachment A. Also user manual for SocialDesktopClient can be found in Attachment B.

At the current stage of development SocialDesktopClient is not a complete product due to constantly developing specification of Channel protocol. It is also dependent on other social services' implementations.

As long as it is an open-source project, it is publicly available to the community of developers and users. Therefore it is open for developers to implement new plugins and also to contribute to the SocialDesktopClient codebase.

Whereas buddycloud project and channel protocol is constantly developing, future works for SocialDesktopClient will be focused on buddycloud plugin, while still introducing new Core features on the way.

Conclusion

The idea of Federated Social Network running the way e-mail network runs today is quite distant. It requires great contribution of the community to develop and test various implementations, while making a standard protocol or protocols' specifications for others to get involved in.

On the other hand it is also the willingness of the very users to quit those closed social silos and support the federation. This might be the toughest challenge FSN might face, however it is necessary for the healthy ecosystem of distinct providers.

We described existing technologies and software focusing on social network decentralization and federating social content. It turns out the majority of them are built and developed on the Web under the W3C's Federated Social Web group. Considering the fragmentation of the Web mentioned technologies, which could hardly satisfy all FSN requirements as a whole. Each can address only certain FSN aspect at a time, e.g. OpenID for authentication, Pubsubhubbub for pushing Web content. This can result in compatibility and security issues.

Moreover Web's HTTP protocol is by design not suitable for content syndication and entities interaction in the federated manner. As we pointed out there are serious shortcomings like stateless connections, non-bidirectional communication, polling character, URL addressing limitation, etc.

On the other hand it turns out there is no need to create completely new application layer protocol for FSN like some of the mentioned projects were trying to do. There is an XMPP protocol, which fits many requirements of FSN. For various characteristics and features implanted right into the core protocol, aspects like federated topology, identity preservation, entity addressability, extensible data format, etc. were solved right away.

In relation to XMPP, presented buddycloud project and its proposed Channel protocol, show as a prospective approach for building FSN on top of XMPP. To the extent of Channel protocol, project involves various extensions like directory & search component, media server, HTTP API server and others. As a result of this, the overall architecture of buddycloud appears to be the most promising one among many other approaches in this field.

By applying Publish-subscribe XMPP extension and reusing existing standards like Atom and Activity Streams, Channel protocol proposes a simple way of sharing content between affiliated users.

As a contribution to the buddycloud project this thesis dealt with architecture and development of SocialDesktopClient, a desktop client for multiple social network services. As a result of no open-source, cross-platform, social-network-oriented desktop clients decision was made to build one from scratch. It turns out as a challenging task to design modular architecture and implement it in a relatively low-level programming language like C++. Dealing with problems like simultaneous debugging of executable and plugin library or interconnecting Boost and Qt signal/slot mechanism in a thread-safe manner prolonged overall

development time significantly.

In regard to buddycloud project, service plugin for Channel protocol has been implemented and is still in development due to changing character of protocol specification and actual server reference implementation. Protocol implementation also required a certain involvement in the events of active buddycloud community - developers mailing list, chat-rooms, video-conferences.

Despite the inconsistencies in “living” Channel protocol specification, the codebase of SocialDesktopClient is ready to be extended and reused for other social network service plugins. Since the project is open-source, contribution by the community is desired and welcome. It has already attracted attention from buddycloud open-source community. This is a good sign for SocialDesktopClient as it may become a desktop platform for social networking in general.

Bibliography

[1] Barnes, J. A. *Class and committees in a Norwegian island parish* Hum. Relat. 7:39-58, 1954 London School of Economics. University of London. England

[2] Boyd, Danah M. – Ellison, Nicole B. *Social Network Sites: Definition, History, and Scholarship* Journal of Computer-Mediated Communication, issue 13 (2008), pages 210-230

[3] O'Brien, Dany *Michael Anti's exile from Facebook over 'real-name policy'* [online] 9.3.2011 [cit. 12.11.2012] Available at: http://www.cpj.org/internet/2011/03/michael-antis-exile-from-facebook-over-real-name-p.php

[4] Gerstein, Josh *Activists upset with Facebook* [online] 18.9.2010 [cit. 12.11.2012] Available at: http://www.politico.com/news/stories/0910/42364.html

[5] Opsahl, Kurt *Facebook's Eroding Privacy Policy: A Timeline* [online] 28.4.2010 [cit. 8.11.2012] Available at: https://www.eff.org/deeplinks/2010/04/facebook-timeline

[6] Esguerra, Richard *Facebook's Broken Promises: Facebook Apps Leaking Private Data to Advertisers and Trackers* [online] 18.10.2010 [cit. 8.11.2012] Available at: https://www.eff.org/deeplinks/2010/10/facebooks-broken-promises-facebook-apps-leaking

[7] *The Open Graph protocol* [online] [cit. 18.11.2012] Available at: http://ogp.me

[8] Fitzpatrick, Brad – Hardt, Dick – Recordon, David and company *Final: OpenID Authentication 2.0 - Final* [online] 5.12.2007 [cit. 29.11.2012] Available at: http://openid.net/specs/openid-authentication-2_0.html

[9] Hardt, D. *draft-ietf-oauth-v2-31 - The OAuth 2.0 Authorization Framework* [online] 31.7.2012 [cit. 29.11.2012] Available at: http://tools.ietf.org/html/draft-ietf-oauth-v2-31

[10] Hammer, Erran *OAuth 2.0 and the Road to Hell* [online] 26.7.2012 [cit. 29.11.2012] Available at: http://hueniverse.com/2012/07/oauth-2-0-and-the-road-to-hell

[11] Sporny, Manu – Inkster, Toby – Story, Henry *WebID 1.0* [online] December 2011 [cit. 29.11.2012] Available at: http://www.w3.org/2005/Incubator/webid/spec/

[12] Jones, Paul E. – Salgueiro, Gonzalo *draft-jones-appsawg-webfinger-00 - WebFinger* [online] 23.10.2011 [cit. 30.11.2012] Available at: http://tools.ietf.org/html/draft-jones-appsawg-webfinger-00

[13] Beckett, Dave – McBride, Brian *RDF/XML Syntax Specification (Revised)* [online] 10.2.2004 [cit. 14.11.2012] Available at: http://www.w3.org/TR/rdf-syntax-grammar

[14] Brickley, Dan – Miller, Libby *FOAF Vocabulary Specification* [online] 9.8.2010 [cit. 29.11.2012] Available at: http://xmlns.com/foaf/spec/

[15] *RFC 4287 - The Atom Syndication Format* [online] [cit. 21.11.2012] Available at: http://tools.ietf.org/html/rfc4287

[16] Atkins, M., et al. *Activity Streams Working Group: Atom Activity Streams 1.0* [online] 13.2.2011 [cit. 22.11.2012] Available at: http://activitystrea.ms/specs/atom/1.0/

[17] Snell, J., et al. *Activity Streams Working Group: Activity Base Schema (Draf)* [online] 30.8.2012 [cit. 22.11.2012] Available at: http://activitystrea.ms/specs/json/schema/activity-schema.html

[18] *OExchange Technical Specification* [online] 14.5.2010 [cit. 14.11.2012] Available at: http://www.oexchange.org/spec

[19] Fitzpatrick, B. – Slatkin, B. – Atkins, M. *Draft: PubSubHubbub Core 0.3 – Working Draft* [online] 8.2.2010 [cit. 14.11.2012] Available at: http://pubsubhubbub.googlecode.com/svn/trunk/pubsubhubbub-core-0.3.html

[20] Slatkin, Brett *Comparison of PubSubHubbub to light-pinging protocols* [online] 24.2.2010 [cit. 6.12.2012] Available at: http://code.google.com/p/pubsubhubbub/wiki/ComparingProtocols

[21] Panzer, J. *Draft: The Salmon Protocol* [online] 7.1.2011 [cit. 14.11.2012] Available at: http://salmon-protocol.googlecode.com/svn/trunk/draft-panzer-salmon-00.html

[22] Prodromou, Evan *Understanding OStatus — StatusNet* [online] 7.3.2010 [cit. 29.11.2012] Available at: http://status.net/2010/03/07/understanding-ostatus

[23] *Protocol Introduction · Tent - the decentralized social web* [online] [cit. 29.11.2012] Available at: https://tent.io/docs

[24] Crockford, D. *RFC 4627 - JavaScript Object Notation (JSON)* [online] July 2006 [cit. 29.11.2012] Available at: http://www.ietf.org/rfc/rfc4627.txt

[25] Dr. Thurston, Adrian D. *DSNP: Distributed Social Networking Protocol, Protocol Specification* [online] 19.5.2011 [cit. 6.11.2012]. Available at: http://www.complang.org/dsnp/spec/dsnp-spec.pdf

[26] Dr. Thurston, Adrian D. *DSNP: A Protocol for Personal Identity and Communication on the Web* [online] Vancouver, British Columbia, Canada Available at: http://www.complang.org/dsnp/dsnp-overview.pdf

[27] *The Diaspora Project* [online] [cit. 29.11.2012] Available at: http://diasporaproject.org/

[28] Bilton, Nick *Diaspora, the Open Facebook Alternative, Releases Its Code - NYTimes.com* [online] 16.9.2010 [cit. 6.12.2012] Available at: http://bits.blogs.nytimes.com/2010/09/16/diaspora-open-facebook-alternative-releases-code

[29] Maka, Stephan *Design and Implementation of a Federated Social Network* 1.6.2010 Hochschule für Technik und Wirtschaft Dresden. Supervised by Prof. Dr. rer. nat. Ralph Großmann.

[30] *Kune: a web tool to encourage collaboration, content sharing and free culture* [online] [cit. 17.11.2012] Available at: http://kune.ourproject.org/

[31] *Salut à Toi: the multi-frontends, multi-purposes communication tool* [online] [cit. 17.11.2012] Available at: http://sat.goffi.org/overview.html

[32] Stone, Biz *Twitter and XMPP: Drinking from The Fire Hose* [online] Twitter Blog 7.7.2008 Available at: http://blog.twitter.com/2008/07/twitter-and-xmpp-drinking-from-fire.html

[33] Fitzsimmons, Seth *Fire Eagle Location Streams* [online] Blog on Fire 19.2.2009 Available at: http://feblog.yahoo.net/2009/02/19/fire-eagle-location-streams/

[34] *Open Real Time Messaging System - Slashdot* [online] 4.1.1999 [cit. 23.12.2012] Available at: http://slashdot.org/story/99/01/04/1621211/open-real-time-messaging-system

[35] Saint-Andre, Peter – Smith, Kevin – Tronçon, Remko. *XMPP: The Definitive Guide* O'Reilly Media, Inc., 2009. ISBN 978-0-596-52126-4

[36] Saint-Andre, Peter *RFC 6120 - Extensible Messaging and Presence Protocol (XMPP): Core* [online] March 2011 Available at: http://tools.ietf.org/html/rfc6120

[37] Millard, Peter – Saint-Andre, Peter – Meijer, Ralph. *XEP-0060: Publish-Subscribe* [online] 12.7.2010 Available at: http://xmpp.org/extensions/xep-0060.html

[38] Hildebrand, Joe – Millard, Peter – Eatmon, and company *XEP-0030: Service Discovery* [online] 6.6.2008 [cit. 3.12.2012] Available at: http://xmpp.org/extensions/xep-0030.html

[39] Tennant, Simon *buddycloud - A Quick Update* [online] buddycloud blog 15.11.2011 [cit. 3.12.2012] Available at: http://blog.buddycloud.com/post/19625443663/a-quick-update

[40] *XMPP XEP - buddycloud wiki* [online] [cit. 3.12.2012] Available at: https://buddycloud.org/wiki/XMPP_XEP

[41] *buddycloud wiki* [online] [cit. 3.12.2012] Available at: https://buddycloud.org/wiki/Main_Page

[42] Tennant, Simon *buddycloud-diagrams/Omnigraffle/what is a channel.png at master · buddycloud/buddycloud-diagrams · GitHub* [online] [cit. 6.12.2012] Available at: https://github.com/buddycloud/buddycloud-diagrams/blob/master/Omnigraffle/what%20is%20a%20channel.png

[43] *Google C++ Style Guide* [online] [cit. 23.11.2012] Available at: http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

[44] *Swiften* [online] [cit. 23.11.2012] Available at: http://swift.im/swiften/

[45] *The Conversation Prism: Květina 2.0* [online] 5.11.2010 [cit. 6.12.2012] Available at: http://captaview.wordpress.com/2010/11/05/the-conversation-prism-kvetina-2-0/

List of Figures

1.1 Variety of SNSs 3

2.1 Decentralized architecture of e-mail network 5
2.2 Network topologies comparison 6
2.3 Identity server network model 8
2.4 WebID authentication process 9
2.5 Pubsubhubbub architecture 12
2.6 Diaspora* preview 14

3.1 XMPP server and components 21

4.1 Expression of buddycloud channel 23

5.1 SocialDesktopClient top design perspective 36
5.2 Threading model 37
5.3 SocialDesktopClient preview 40
5.4 Pure main window 52
5.5 Adding SocialDesktopClient account 53
5.6 Posting topic 53
5.7 Posting comment 54
5.8 Subscribe to user 54
5.9 CD contents 55

List of Tables

4.1 Channel roles 24
4.2 Channel properties 29

List of Abbreviations

FSN Federated Social Network

IETF Internet Engineering Task Force

JID Jabber ID

SNS Social Network Site

Web World Wide Web

XEP XMPP Extension Protocol

XMPP eXtensible Messaging and Presence Protocol

XSF XMPP Standards Foundation

Attachments

Attachment A: Program compilation

This is a procedure for building SocialDesktopClient application and buddycloud plugin extension. Note that this procedure can be functional only for the time being of writing this thesis. Due to the application development it is most likely that it changes. Actual information can be found on project hosting at

https://github.com/maros-/SocialDesktopClient

Requirements

- OS Linux (tested on Kubuntu 12.04.1 LTS 64 bit)
- Qt Libraries (tested with version 4.8.1)
- GCC 4.5 or later
- Boost 1.46.0 or later

Build steps

Install following developer packages.

```
apt-get install build-essential libboost-all-dev qt4-dev-tools
    openssl git
```

Clone project repository from project hosting.

```
git clone git://github.com/maros-/SocialDesktopClient.git
```

Build SocialDesktopClient

```
cd SocialDesktopClient
./build.sh
```

Build buddycloud plugin

```
cd plugins/buddycloud
./build.sh
```

Run program

```
cd ../..
./social_desktop_client
```

Attachment B: User manual

SocialDesktopClient is a desktop client for multiple social networks services. At this stage it is being developed to support buddycloud Channel protocol. In order to use it, user must have an existing XMPP account on the server with the support for Channel service.

User interface

Upon application start main window is displayed as shown in Figure 5.4. There are seven elements described below.

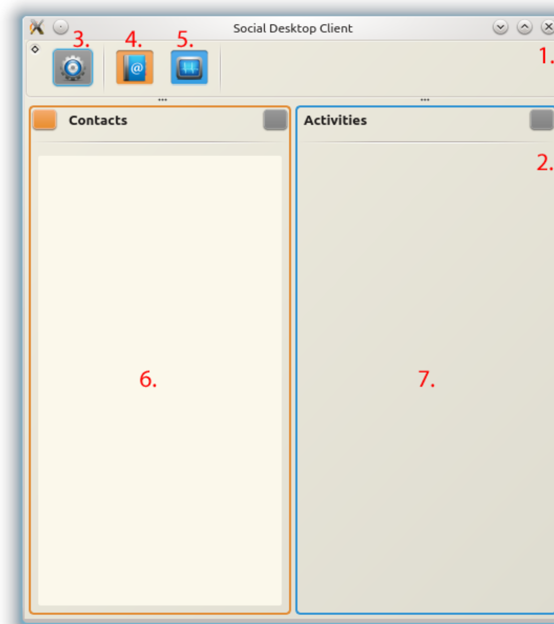


Figure 5.4: Pure main window

1. **Main control panel** Contains main application buttons for settings, contacts, activities and activated SocialDesktopClient accounts.

2. **Content area** Place for all application content. It uses the concept of *content panels* which are behaving equaly within the content area. Basically, all content are displayed in content panels situated side by side within the content area. Each content panel can be closed by clicking the *grey button* in the panel's upper right corner.

3. **Settings button** Entry point for application settings.

4. **Contacts button** Shows/hides contacts panel.

5. **Activities button** Shows/hides activities panel.

6. **Contacts panel** Common place for social service's contacts.

7. **Activities panel** Common place for social service's activities.

Adding account

For adding new account we step into the settings (3. Settings button) and by clicking ADD button we invoke New Account dialog. Since there can be multiple social services we choose the right one from the combobox and fill the form (see Figure 5.5).

Figure 5.5: Adding SocialDesktopClient account

After creating new account, it is activated automatically. We can deactivate by checking the box in the accounts table. By clicking account in the table, we can also edit it or remove it completely. As a result of activating account, the account button shows in the application control panel.

Channel account operations

Upon Channel account activation, an actual user's channel shows in the content area. We can post topic message by clicking *white button* on the left of channel's title (see Figure 5.6).

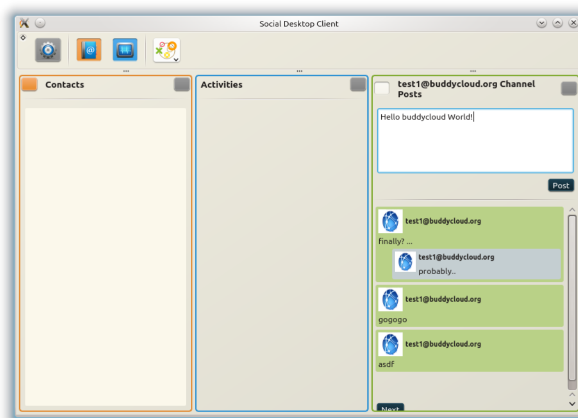


Figure 5.6: Posting topic

If we want to comment on topic post, by clicking it commenting field shows up (see Figure 5.7).

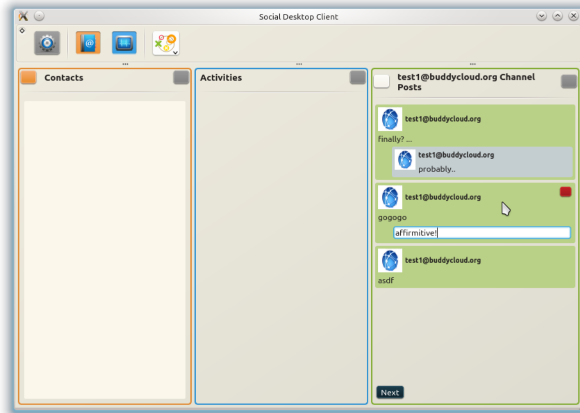


Figure 5.7: Posting comment

In order to follow a new channel, we access New Contact dialog by clicking contacts panel upper left *orange button* (see Figure 5.8). Then we choose actual Channel account and enter channel's user name. Unfollowing channel is easily

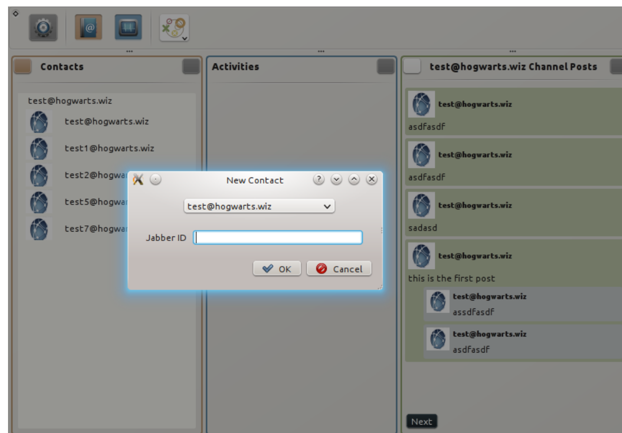


Figure 5.8: Subscribe to user

done by right-clicking on choosen user in contacts panel.

Attachment C: CD contents

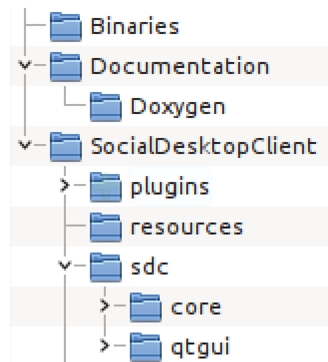


Figure 5.9: CD contents

- `Binaries` Location for SocialDesktopClient and buddycloud plugin binaries.
- `Documentation` Location for thesis, user manual and auto-generated doxygen documentation.
  - `Doxygen` Location for auto-generated doxygen documentation.
- `SocialDesktopClient` Source code location.
  - `plugins` Location for plugin implementations.
  - `resources` Location for application runtime resources.
  - `sdc` Location for SocialDesktopClient source code.
    - `core` Location for Core source code.
    - `qtgui` Location for Qt GUI implementation.