A Complete Solution for Peer-to-Peer Widgets

tribler

Alain M. van den Berg

TU Delft

Delft University of Technology

A Complete Solution for Peer-to-Peer Widgets

Master's Thesis in Computer Science

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Alain M. van den Berg

11th September 2009

Author
Alain M. van den Berg

Title
A Complete Solution for Peer-to-Peer Widgets

MSc presentation
31st August 2009

Abstract

Nowadays, the World Wide Web is becoming more and more an interactive and social platform than just a means to find information. This is called the Web 2.0. Quite new in the World Wide Web are widgets; small applications that use a little area of a website and displays something specific, often making use of a Web 2.0 application to get its information. The widgets can be combined to create a personal page.

Compared to the client-server architecture traditionally used in the Internet, the Peer-to-Peer technology can be used to design and create much more scalable and robust systems. Unfortunately, while they are so promising, P2P systems are mostly used for file transfers.

We present a complete P2P Widget System, that introduces social and interactive elements to the P2P paradigm. Using P2P technology, the widget repository (where the widgets are downloaded from) are more scalable and robust than the centralised repositories currently used for widgets. Not only do we present a complete design for this system, we also present a working implementation of this system, which can be deployed in real-life. Furthermore, we show that the system works as it should, by presenting the results of several experiments. From our results, we can conclude that our P2P Widget System is scalable, robust, fast and bandwidth efficient.

Preface

This is the report of my MSc thesis project in Computer Science on combining widgets with P2P technology, using the Tribler P2P client. This research has been carried out in the Section Parallel and Distributed Systems of the Faculty of Electrical Engineering, Mathematics, and Computer Science of Delft University of Technology in the context of the I-Share research project.

I would like to thank my supervisor, Johan Pouwelse, for his guidance during my thesis project. The brainstorm sessions were always fun and inspiring. Further, I would like to thank the Tribler team. Many of you had good comments and suggestions and were always in for a discussion.

Alain M. van den Berg

Delft, The Netherlands
11th September 2009

Contents

Chapter 1

Introduction

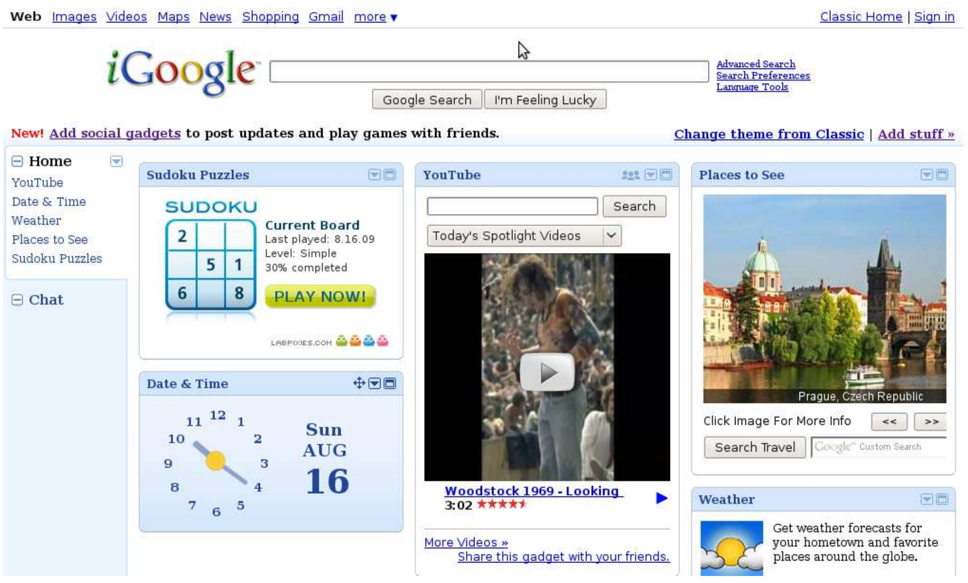
Nowadays, the World Wide Web (WWW) is becoming increasingly more interactive, social and collaborative than before. Instead of the traditional way of providing users with information, the WWW is becoming a platform of web services where users can create, share and distribute content. This trend is also identified as Web 2.0, emphasising the metamorphosis from Web 1.0, the traditional web. Examples of Web 2.0 applications are weblogs (blogs), photo sharing sites such as Flickr, video sharing sites such as Youtube, and social network sites such as Facebook or MySpace.

Mostly because the rise of Web 2.0 applications, widgets (synonyms: gadget, badge, module, etc.) were able to be invented and popularised. Widgets on the web are small portable chunks of code that can be added to any website. They use a small area on the website to provide a single service to the user or visitor. Most widgets use a Web 2.0 service to retrieve their information. Example widgets are widgets that show a random word of the day (serviced from a dictionary site), the latest news (serviced from a news site), your picture of the day (from a photo sharing site). More interactive widgets are also available such as games, translating text widgets, to-do list widgets and more. A very popular widget is the Youtube Gadget, which shows a user selected Youtube video. Widgets enable the user to create their own personalised pages where they can collect all services they are interested in. Widgets can also be added to their blogs or social network profile. The iGoogle start page is such a personalised page where users may add different iGoogle Gadgets which they can select from a repository. An example iGoogle startpage with widgets is shown in Figure 1.1. Another example is the Facebook Apps. Facebook is a social network, where everybody can create a profile, add friends and stay in touch. Facebook Apps can be selected and added to the users profile, such that the user (and visitors) can view the widget and possibly use it.

Widgets for other platforms are also quite popular, such as desktop widgets or mobile widgets. Mobile and desktop widgets are small portable pieces of code that can be added to a widget engine. The widget engine is a program wherein the widgets run; it maintains their state, provides an API for the functionality and provides the widgets the necessary screen space. Examples of desktop widgets are Windows Vista Gadgets and Google Desktop Gadgets. Because every widget engine specifies a different runtime environment and API, the widgets for one engine are typically not compatible with other engines. By providing AJAX functionality (i.e., asynchronously send an HTTP request to a Web 2.0 service), desktop widgets are also able to use Web 2.0 services.

Most Web 2.0 services are built using the client-server architecture, where the services are hosted by a single server or a cluster of servers. Consequently, when the number of clients rises, the servers may not be able to serve all requests. Sites such as Youtube, with a very big company behind it (Google Inc.) solve this by simply adding more servers and tweaking the software, hardware and network [16]. Other companies might not have this extensive amount of required resources. Eventually, even Youtube will have to resort to other, more scalable measures. The scalability drawback becomes even more dramatic because of widgets. Widgets from a certain host are distributed over multiple (possibly thousands) sites and all these widgets generate requests for this host.

The peer-to-peer (P2P) paradigm tries to improve on the formerly stated client-server architecture, especially by providing incremental scalability. In P2P, everybody is equal (they are peers) and contributes to the system. This improves scalability, because added peers do not only use services, but also provide services to other peers. According to several Internet studies [17, 18], P2P generates a vast amount of traffic, a big percentage of the total Internet traffic. This can also be seen in Figure 1.2. From these statistics, we can conclude that P2P is already very popular and probably will stay so for a long time. P2P is used mostly for file sharing,



New! Add social gadgets to post updates and play games with friends.

Current Board
Last played: 8.16.09
Level: Simple
30% completed

Get weather forecasts for your hometown and favorite places around the globe.

Figure 1.1: Personalised iGoogle startpage with multiple widgets.

but other possible applications are instant messaging, telephony, media streaming and grid computing.

The aim of this thesis is to combine the concept of widgets with the P2P concept. In this way, the widgets are distributed via the P2P network and use the P2P network for communication. When the architecture is powerful enough, an interactive, social and collaborative P2P experience is possible. Ultimately, it is possible to create a self-managing and evolving environment, where users may contribute by writing new or extending older widgets.

In the rest of the chapter, we will introduce the concepts for this thesis more broadly. Section 1.1 gives an introduction to widgets, Section 1.2 explains general P2P technology and specifically the BitTorrent protocol and the Tribler P2P program. Further, Section 1.3 lists our contributions and Section 1.4 discusses related work. Finally, Section 1.5 provides the outline of this thesis.

1.1 Widgets

Although widgets are found all over the Internet, many do not know exactly what they are and why they are called widgets. Therefore, we will start with defining the widget as exact as possible in Section 1.1.1. Then we will state the properties of current widget systems in Section 1.1.2. Finally, we will explore technologies related to widgets in Section 1.1.3.

1.1.1 Definition

The word widget has two different definitions shown below:

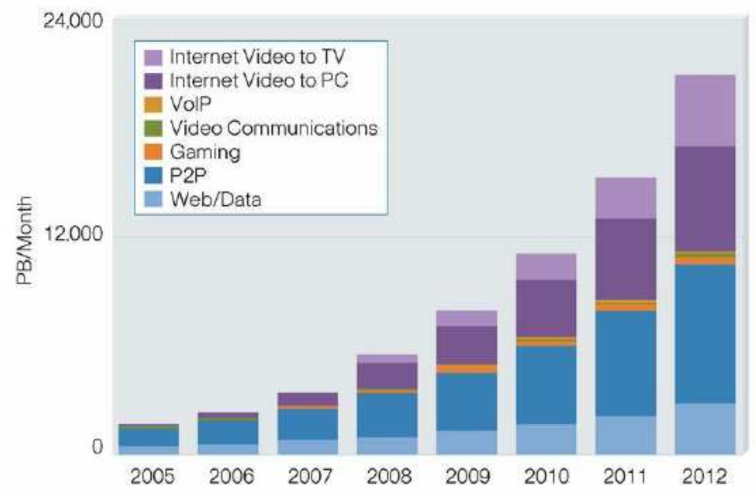


Figure 1.2: Global Consumer Internet Traffic Forecast. Source: Cisco, 2008 [18]

1. an element of a graphical user interface (GUI) such as a button or a scroll bar.
2. a portable chunk of code that runs in a specialised environment (widget engine)

Although the first definition is not the definition we are looking for, the second kind of widget (the one this thesis is about) is probably emerged from the GUI widget in the first definition. The second widget definition is very broad and means that a widget can almost be anything, but it mostly implies that the widget has a small area on the GUI of the widget engine and that they are rather light, as opposed to large software packages.

Widgets are also categorised, reflecting the platform they run on. The platform affects the way the widgets are created and what can be done with them.

Wikipedia distinguishes three types of widgets [9]:

1. *Desktop Widgets* are interactive virtual tools that provide single-purpose services such as showing the user the latest news, the current weather, the time, a calendar, a dictionary, etc.
2. *Mobile Widgets* are like desktop widgets, but for a mobile phone.
3. *Web Widgets* are portable chunks of code that can be installed and executed within any separate HTML-based web page by an end user without requiring additional compilation.

While these definitions might seem all right, they are not totally satisfying. For example, the definition of Desktop Widgets also applies to both the other definitions. This is because it lacks the sentence where it says the desktop widgets run on a desktop program, the widget engine, which first has to be installed.

Further, we argue that there is already another type of widget, which we call the Social Widget:

4. *Social Widgets* are portable chunks of code that run within social networking sites, having access to social information such as friends and activities.

And lastly, this thesis introduces a new class of widgets, called P2P Widgets:

5. *P2P Widgets* are portable chunks of code that run within P2P networks, having access to functions the P2P network provides.

This definition again is rather broad, because P2P networks might have different functions. At the time of writing, they are mostly used for file sharing. However, Tribler [21] is a P2P program which tries to implement social features, such as taste buddies and friends. Within Tribler, P2P Widgets may also become social.

1.1.2 Properties of Current Widget Systems

Here we examine the properties of current widget systems. As explained in the previous subsection, there are multiple types of widget systems, depending on the platform they run on. The most common are Web Widgets, Desktop Widgets and Mobile Widgets. Although they run on different platforms, their properties remain the same to a large extend. We will focus primarily on the network properties of widget systems, both of the runtime and their distribution method (i.e., how the widgets are distributed).

Centralised distribution. Currently, most widgets are distributed by central repositories, usually operating as a website where visitors can find, comment and rate widgets. Only Web Widgets are the exception, because they are primarily distributed by copying a small piece of code and pasting it on your own site.

Use of Web 2.0 services. This is not a prerequisite for a widget, but most widgets use Web 2.0 services to provide the user with information. A Facebook App can actually be seen as a Web 2.0 service, because it is an interactive application which has to be hosted on your own website (which can use the Facebook API). Most Web 2.0 services are also centralised.

Interactive. Again, this is not a prerequisite for a widget, since most widgets can just display some information. However, the power of most widgets is that they are interactive. Think of looking up a word, a game (sudoku for example), social interaction. In social networks, even the little widgets which just allow you to poke a friend or say 'hi' are a huge success, just because it allows you to interact with others.

Widgets are rather small. Web Widgets are, most of the time, just small chunks of code to be copied. Other widgets, such as Desktop Widgets are packaged in a small archive to be able to contain other resources such as images. In Table 1.1, the average size of various widget types are presented to support our statement.

Widget platform	estimated number of widgets	estimated average size per widget (KB)	estimated total size (MB)
Eclipse Plugins	1,200	900	1,054
Facebook Apps	40,000	–	–
Firefox extensions	3,100	500	1,500
Google Desktop Gadgets	1,250	80	100
Windows Vista Gadgets	5,800	140	780
Yahoo Widgets	5,000	600	2,930

Table 1.1: Estimated repository size of various widget platforms.

The following properties are derived from the properties above:

Finding a widget is easy and deterministic. Because all widgets are stored in a central repository, the repository knows all widgets and a search will always find the widget when it is available (i.e., the search is deterministic). All widget repositories support the browsing of the repository by providing a list which can be sorted on name, rating, popularity, et cetera.

Not cost-effective scalable. Because of the first two properties (centralised distribution and use of Web 2.0 services), widgets are not cost-effective scalable. The problem of distribution is however alleviated because of the small widget size: a repository of 100,000 widgets is a lot more scalable than a website with 100,000 small video's. Estimated repository sizes of various widget or extension systems are shown in Table 1.1.

The second property however, use of Web 2.0 services, might be a bigger problem. Because of the nature of widgets, they are rapidly distributed among thousands of people. This is best illustrated by an example: the YouTube video gadget. This is a Web Widget, and everyone can just copy a small chunk of HTML code to their website to show a video on their site. Now imagine thousands of users putting YouTube Video Gadgets on their site. All these video's are served by YouTube itself, generating bandwidth which can be compared to a Distributed Denial of Service attack. This is of course a bit exaggerated, because most widgets do not use video's (yet), but they do send requests to Web 2.0 services. For example, RSS feed widgets poll every now and then to see if there are new items (and imagine thousands of users polling the same server because they have the same widget installed).

Not very robust. Again because of the centralised distribution and use of Web 2.0 services, widgets are not very robust. Whenever the distribution site is down, it is not possible to get any widget. When a Web 2.0 service is down, the widget will not work either.

Central authority. Distribution is done centralised, which means that when you have created a widget, the authority may either reject or accept the widget. The rules for rejection are defined by the authority itself. Although a trusted authority might only use this power to reject malicious widgets, it might also reject widgets with a certain (political) statement or because it is coerced by governments or other higher authorities (e.g., they may want to reject a widget that is a free alternative to a business solution). Another problem with a central authority is the power they have because of privacy sensitive data. Social widgets for example, might use privacy sensitive information which is stored on the server.

1.1.3 Related Technologies

Widgets belong to a group of technologies that extend a particular system. Below, we discuss other related technologies and their similarities and differences.

Plugins are not the same as widgets, but are close related. Widgets typically use a small area of the applications screen and displays specific information or provides specific features. Plugins, on the other hand, are not bound to a small area of the screen, but may hook into the system more freely. Plugins are mostly more focused on extending the features of the system, while widgets are more focused on serving the user with information and services. Another difference is that widgets are stand-alone, in that they can not be extended by other widgets. Some plugins, such as Eclipse Plugins, do support these features but require a very sophisticated Runtime Environment. The runtime should check for plugin dependencies and provide an API to be able to extend the plugins. Nevertheless, the boundary between the two is vague, especially from a technological view.

Linux packages extend the Linux operating system. Almost all applications for Linux can be found as a package. The package management for Linux packages is quite sophisticated, dealing with a lot of dependencies between applications, while also taking their versions into account. While widgets are very small applications and require no installation, Linux packages can get as large as possible and may take a while to be installed. Package management software is usually included in the Linux distribution, which allows the user to find, install, upgrade and remove packages. There are also package repository sites where the packages can be found and downloaded.

1.2 P2P Technology

The key differences of the P2P architecture with the client-server architecture is that every peer acts as a client and a server and that the peers are equal. Every peer that joins the network, does not only consume resources, but also contributes extra resources to the other peers in the network. This means that more peers can be served when more peers join, which means that it is inherently scalable. When resources are replicated on multiple peers, then it does not matter whether a few peers go offline, because the other peers also have these resources. This makes a P2P system more fault-tolerant than a client-server architecture. However, it is a challenge to design correctly working, fault-tolerant and robust P2P systems. First, it is a difficult task to let all peers cooperate seamlessly, especially in the case of peers that are not altruistic or even malicious. The second is that the design is distributed over all the peers.

We will start this section by presenting an overview of several P2P systems in Section 1.2.1 and define the most important properties of P2P systems in Section 1.2.2. Then we will focus on the most popular P2P protocol today, BitTorrent, in Section 1.2.3. Finally, the P2P program Tribler is discussed in Section 1.2.4. We discuss Tribler because we will use Tribler to implement a fully functional P2P Widget System.

1.2.1 Overview of P2P Systems

Most people were introduced to P2P systems by the introduction of Napster. This file-sharing P2P system allowed users to share their music with others. However, Napster used a server to index the songs of each user. Queries to search for music were handled by the central server and Napster thus had a central point of failure.

Other file sharing P2P systems emerged, which were fully decentralised. They can be classified as either structured or unstructured.

Structured networks control object placement in such way that the nodes can find the appropriate objects easily. They use a mapping of objects and nodes to the same address space, such that objects that are close to nodes in the address space typically reside on those nodes. The nodes maintain a distributed routing table to efficiently forward the query to the right node. The structured networks are more scalable than the unstructured ones, but they also have drawbacks: only exact-match queries are supported and it is hard to maintain the structure in networks with high churn (i.e., high rate of leaving and joining nodes). Structured networks are mostly called Distributed Hash Tables (DHT), as they are a mapping of object IDs (keys) to the object, just like hash tables. Examples include Chord [26], CAN [23] and Pastry [25].

In unstructured networks, the placement of content is completely unrelated to the topology. Unstructured networks differ in the way the index is distributed. Some networks do not keep any indexing information. These networks need to locate objects by querying the network (e.g., by flooding or random walks). Examples are Freenet [13] and Gnutella [12]. While they are fully decentralised, they have some downsides. First, flooding is not scalable and second, both flooding and random walks may fail to find objects because they are curtailed. Other networks spread the index using gossipping. Once in a while, every peer seeks a gossip partner and gossips a bit of the index to the others. Depending on the details, this may eventually result in a complete copy or a part of the global index on every peer. Examples of gossipping P2P networks are PlanetP [15] and NewsCast [19].

1.2.2 Properties of P2P Systems

In this section, we will explore the properties of current P2P systems.

Cost-effective scalability. In contrast to the client-server architecture mostly used in Web 2.0 applications, P2P systems are very scalable in a cost-effective way. Every peer consumes resources, but because of the contribution the total amount of resources also increases. The primary example is of course the video sharing site YouTube, where Google is putting a lot of effort and money in the maintainance of the site. This is logical, because video files are large; they require a lot of storage and quite some bandwidth to stream or download them. But also sites such as Wikipedia, the online encyclopedia, require more and more resources because of their popularity.

Robustness. In an ideal P2P system, there is no central component. This makes the system more robust, as it will be able to stay running when some components fail. However, a big research question is how to create a P2P system without central component in an effective way. Most P2P systems still use central components for non-trivial tasks such as bootstrapping or indexing the resources.

Search is not always deterministic. In fully distributed P2P systems, it is very hard to find all items, because there is no central index. Random walks or flooding, which do not cover the whole network, are undeterministic. Distributed Hash Tables (DHTs) are able to do a deterministic search, but there are still a lot of problems in practice. Gossip based systems which use full replication are not very scalable (such as PlanetP) and gossip based systems which do not use full replication are not deterministic.

Little (social) interactivity. Currently, the most widely used P2P application is file sharing. People start the P2P client, download what they want and either quit the program or stay idle. P2P programs are trying to create more interactive P2P, by providing chat, streaming video, comments, but interactivity in P2P programs is still in its infancy. The P2P program Tribler tries to create a social overlay, which might be the first step to P2P interactivity, but also this is still being researched and developed.

No central authority. In centralised applications, there is always one entity in charge. Most of the time, they safeguard the quality of the services, for example by using moderation. However, the authority may misuse privacy sensitive data of users and may also silence opinions it does not like. In P2P systems, it is the choice of the designers whether there is one authority or none at all. When there is none, other possibilities for moderation should be used, such as rating or reputation systems. In this way, P2P systems give control to the people, instead of one authority.

1.2.3 BitTorrent

The most common Internet P2P protocol today is BitTorrent, created by Bram Cohen [14]. The Ipoque Internet study [17] shows that BitTorrent is by far the most popular P2P protocol and is generating more than 30 percent of the total Internet traffic.

The protocol works as follows. Someone who wants to publish a file creates a static file called a torrent file. The torrent file contains information about the file, such as the name, its length and hashing information, and a tracker URL. Users that want to download the file, must contact the tracker, found in the torrent file, to find other peers that are leeching or seeding the file. The leechers and seeders together form a swarm for that file, which is tracked by the tracker. Seeders are peers that have the file completed on disk and only upload to other peers, while leechers are peers that are still downloading the file. The peer joins the swarm as a leecher when it connects to the various peers in the list it received from the tracker. The file that is being distributed has been split into various pieces of fixed size and the peers exchange information on which pieces they have. They select pieces to download using the rarest first policy. This policy tries to replicate the rarest pieces first, to increase availability. Peers to exchange pieces with are selected using tit-for-tat. Tit-for-tat is a policy that uploads pieces to peers from whom we have received a lot of pieces. This makes sure free-riders have trouble downloading the file, because they contribute little to the swarm.

Of course, trackers are a central component in this protocol, which limits the scalability and robustness. There have been numerous attempts to decentralise this component. First, the torrent file has been changed to support multiple trackers. Second, a Distributed Hash Table, which provides a lookup service similar to a hash table is used. File hashes are then mapped to a set of peers. Third, Peer Exchange (PEX) [1, 20] assumes that every peer knows part of the swarm, and thus instead of contacting the tracker, they exchange sets of other peers in the swarm.

Although the DHT and PEX solutions are scalable, they lack protection against peers that try to misuse the protocol. Therefore, the epidemic protocol Little Bird [24] that disseminates swarm information was developed, which uses BuddyCast (discussed below) instead of a tracker to bootstrap. Other research is also ongoing at the Technical University of Delft to create a better solution for the tracker problem, as Little Bird is too resource intensive.

Another problem is the distribution of torrent files. Currently, there are a few popular websites that have a lot of torrent files. This can also be seen as a central component. Tribler, discussed below, is able to retrieve torrent files without the need for a website, using BuddyCast.

1.2.4 Tribler

After researching P2P networks and in particular BitTorrent for years, the researchers of the Parallel and Distributed Systems group of the Faculty of Electrical Engineering, Mathematics and Computer Science designed a new client called Tribler [21]. While Tribler uses the BitTorrent protocol for file transfer, it adds social features and context. They believe that social groups can be used to solve current P2P research challenges, such as full decentralisation, availability, integrity and providing incentives to cooperate. Normally, P2P systems see their users as anonymous, unrelated entities, but Tribler facilitates the formation and maintenance of social groups and exploits their social phenomena for improved content discovery, recommendation and sharing.

By supporting permanent identifiers called PermIDs, which do not change when the IP address changes or the session is restarted, Tribler enables users to build relationships. The peers are authenticated using a public key challenge response, where the PermIDs are the public keys.

The core of Tribler is the BuddyCast epidemic protocol stack [22]. BuddyCast creates a taste preference profile by looking at your current and old downloads and exchanges the SHA1 hashes of these downloads, a list of peers that have similar taste (taste buddies) and a list of random peers with other Tribler peers. Using this information, a semantic overlay is created where peers are partly connected to taste buddies and partly to random peers. Other services are built on top of BuddyCast. For example, TorrentCollecting collects the full metadata of various SHA1 hashes. Using BuddyCast, Tribler is able to discover new content and new peers without the use of websites to find torrent files. Research is still ongoing to implement a fully functional, scalable, secure and fast distributed tracker algorithm, which would make Tribler a fully decentralised P2P client.

1.3 Contributions

In this thesis, we make the following contributions:

1. We designed and implemented a fully operational, security enabled P2P Widget Runtime Environment.
2. We created a zero-server implementation for discovering and downloading P2P Widgets within a P2P network.
3. We designed and implemented an operational zero-server market system for finding, rating and reviewing P2P Widgets.
4. We extended the widget paradigm with local storage and zero-server intra-widget communication, which collaboratively support global storage.

Together, these contributions are a giant leap towards an interactive, social and collaborative P2P experience.

1.4 Related Work

In this section we will discuss related technologies, while focusing on the differences from our work. Because our P2P Widget System is complete, including a widget runtime environment, a decentralized widget repository and a widget market, we will address all these fields.

1.4.1 Widget Systems

First, there are a lot of widget systems, such as Google Gadgets [3], Facebook Apps [2], Yahoo! Widgets [10], Microsoft Gadgets [4]. They are related to this work, because they all provide a widget runtime environment, with their specific features, and a widget repository. Although these systems were an inspiration to our work, there are two major differences with our work. They all provide a widget repository that is centralized and is managed by a central authority, and their widgets are only using centralized services such as Web 2.0 services. This is discussed in more detail in Section 1.1.

Google Gadgets may use GoogleTalk to communicate with another gadget instance on another pc, provided they are the same gadget. The GoogleTalk API provides features to find the user's Google Talk friends, and to exchange data with these friends. This concept is different from the Intra-Widget Communication our system provides, in that we do not distinguish friends and use a gossipping protocol to synchronize data between peers. Further, communication partners in our system are found using decentralised mechanisms, while Google Gadgets uses the centralized Google services. It is not known whether the communication is done via Google or directly to the users widget.

Azureus Plugins are maybe the closest related to our work. While the differences between plugins and widgets are discussed in Section 1.1.3, they are more closely related to our work because the plugins run in a P2P environment. Azureus is a P2P file sharing application, also based on the BitTorrent technology. Azureus Plugins are able to extend the Azureus Extended Messaging Protocol with their own messages. One plugin that uses this features is the Chat Plugin. It can be used to chat to other users in a BitTorrent swarm, because it creates a channel per BitTorrent swarm. However, this is not used frequently, because the people in the swarm have no social connection, except that they all want to download the same file. Further, the Chat Plugin is the only plugin that uses the communication features Azureus supports. Maybe this is because the plugin developer has to design its own protocol, which is far more difficult than our Intra-Widget Communication API. The last difference with our work is again the centralised distribution of the plugins. Azureus Plugins are downloadable via a website, while we use a decentralised Widget Market with ratings and comments.

1.4.2 Distributed Repositories

There has been done a lot of research to create scalable and robust distributed repositories using P2P technology. We already discussed these systems in Section 1.2.1. The two systems that are mostly related to our decentralised widget repository are PlanetP [15] and BuddyCast [22]. We will discuss these systems now. Because we already discussed BuddyCast and our work even uses BuddyCast, we will only state the differences from BuddyCast and our widget repository.

PlanetP [15] is a P2P content search and retrieval infrastructure targeting communities wishing to share large sets of text documents. It uses gossiping to replicate a global directory that includes a list of peers and their Bloom filters [11]. Bloom filters are used to summarize the text files each peer has, in an efficient way. Our approach differs from PlanetP, because we use gossipping to disseminate torrent files, which allow us to download the widgets. To download the files in PlanetP, the list of peers is disseminated using gossipping. While our approach builds on the scalable BitTorrent protocol for downloading. Apart from only disseminating the index (the torrent files), we also replicate the datafiles, by collecting the widgets. This seems less scalable than PlanetP, but our datafiles and repository are so small that neither bandwidth consumption nor disk space are a problem.

BuddyCast is primarily used to create a repository for torrents. Torrent files are disseminated, but only until a maximum of 5000 are collected. Thus, only a small part of the global directory is disseminated. Files are found by looking at your own part of the directory and sending queries to the neighbours. Thus, it uses the push-pull model just as is described in NewsCast [19], on which BuddyCast is based. The difference with our approach is that the whole directory (all widget torrents) and all the datafiles are disseminated, while BuddyCast only disseminates part of the global directory. This is logical, because the torrent repository is much larger than the widget directory. Also, our repository uses the push model (i.e., parts of the directory are only pushed and no requests are sent to neighbours.

1.5 Thesis Outline

The remainder of this thesis is as follows. We start with describing the problem of merging widgets with P2P technology in Chapter 2. Then we present the design of our fully operational P2P Widget System in Chapter 3 and discuss the implementation of the system and example widgets in Chapter 4. We present the experiments we conducted to evaluate our implementation, and their results in Chapter 5. Finally, in Chapter 6, we give our conclusions and propose further research that can be conducted in extension of this thesis.

Chapter 2

Problem Description

In this chapter we describe in detail the problem of merging widgets with P2P technology.

First, we will show our vision about P2P widgets and how they combine the best of both worlds in Section 2.1. Then, we will focus the requirements of these P2P widget systems and in particular define the scope of this thesis, respectively in Sections 2.2 and 2.3.

2.1 Properties of P2P Widgets

In Chapter 1, properties of current widget systems and P2P systems have been analysed. A summary of the properties is given in Table 2.1.

P2P systems are a good choice when scalability and robustness are properties a system should have. Also, there will not be any central authority in control, as this is inherent to a P2P system. However, current P2P systems do not have a lot of interactivity among users. This is because current P2P systems are primarily used for file transfers without any social interaction.

Creating a widget environment in a P2P system might increase social interactivity, but only when a social overlay is being provided (you have to be able to find your friends) and the widget runtime provides an API for communication among widgets. Increasing social interactivity will have a good impact on current P2P problems such as the current high churn rates (high rate of peers who join and leave) and leechers (users who only download and then leave the system). Currently, a lot of Internet users are putting time in social interactivity through centralised social networks. When P2P also provides this functionality, they might stay longer to chat with a friend, play a game, et cetera. When they stay online longer, they automatically provide more resources to other users.

However, the deterministic search or browsing through the repository, which centralised widget repositories provide, is hard to accomplish in P2P Systems without losing scalability. Scalable deterministic search accomplished by using DHTs, however their performance is currently still lacking in practise [24]. Browsing the complete repository (which we will call deterministic browsing) can also be done by DHTs or by replicating the index on every peer, which is done by PlanetP [15]. We argue that undeterministic search or browsing can be as successful or even more successful than their deterministic counterparts. For example, undeterministic file sharing P2P systems such as Gnutella and Tribler provide quite a good search mechanism. Also, undeterministic browsing (where not every item can be seen) might work when at least a good percentage is browsable and this percentage contains the most popular items.

All in all, we conclude that there are quite a few good reasons why widgets should also be available in a P2P system. Eventually, P2P might even become as interactive as Web 2.0 or merge with the web. P2P widgets are the first step towards a scalable and robust widget system and towards social interactive P2P.

	Current Widgets	Current P2P systems	P2P Widgets
Scalability		x	x
Robustness		x	x
No central authority		x	x
Social interactivity	x		x
Deterministic search/browsing	x		

Table 2.1: Summary of properties of current widgets, current P2P systems and P2P systems with widgets.

2.2 Requirements

As can be seen in Table 2.1, merging of widgets with P2P technology adds functionality of both paradigms. We should note that it is easy to create a P2P Widget System that does not have those properties, and it remains a challenge to merge the two paradigms such that all the properties defined in Table 2.1 remain. For example, the P2P Widget System might note be scalable when it is designed poorly. Also, social interaction can easily be left out when the P2P Widget System does not provide a way to find other widgets to communicate with, or when the Communication API is poorly designed. We take the first step to a P2P Widget System conform the properties of Table 2.1 by defining the following requirements for our system:

1. Discovery and download of widgets must be done in a scalable, decentralised manner.

2. The widget repository must be browsable, i.e., provide a sorted list of widgets.

   The list does not have to be complete all the time, but should eventually contain most of the widgets (undeterministic browsing)

3. The widget repository must support rating and reviewing of widgets in a scalable, decentralised manner.

4. The widget system must support local storage per widget.

5. The widget system must support scalable intra-widget communication.

   The widget system must provide a useful Communication API for intra-widget communication.

   The widget system must find and maintain a set of intra-widget communication partners.

In this thesis project, we conduct experimental research, which means we want to create a system to be deployed and obtain actual usage data. To be able to obtain actual usage data, a key issue is to convince many users to use it. To be able to make this research a success, the following non-functional requirements are necessary:

1. It should be easy to use.

2. It should be secure.

3. It should appeal to many users.

4. It should be easy to obtain performance data without much user interaction.

2.3 Focus and Scope of This Thesis

Because the subject of creating a P2P widget system is rather broad, we will define several aspects on which we will focus. First of all, since we want to be able to deploy the P2P widget system, at least every aspect should receive attention. The runtime environment is an essential part of the P2P widget system, but not the focus of this thesis. Therefore, we will focus on the essential features of the runtime environment. There should be enough security measures to make the P2P widget system secure enough to deploy, but this should also not be the focus.

On the other hand, creating a decentralised widget repository using the P2P network is one of the key focuses of this thesis. There has been done a lot of research on file sharing, but creating a decentralised widget repository which can be sorted on popularity, for example, is still rather new. Another focus point is to create a communication API for widgets, to be able to use the P2P network to communicate. Without this feature, social interactivity is not really introduced into P2P networks, because every widget would be an island. When intra-widget communication is introduced, the widgets are really becoming a P2P widget.

Chapter 3

Design

In the previous chapter, we have defined what we want to have in our P2P Widget System. In this chapter we will present the design of our system and why we chose this design. By now, four subsystems can already be distinguished, namely the Runtime Environment, Discovery and Download, the Widget Market, and Widget Communication and Storage.

In Section 3.1 we will show what each different subsystem encompasses and argue the general design direction we will take for them. In Section 3.2, we will then dive into the technical design.

3.1 General Design Directions and Motivations

In this section, we explain per subsystem what each subsystem entails, give several alternative design directions and motivate our chosen design direction.

3.1.1 Runtime Environment

A Widget Runtime Environment or Widget Engine is the part of the system that takes care of executing and managing the widgets. For a successful Runtime Environment, it is necessary to determine the widget language and file format, the features that are available to the widget (API), and any security measures for execution of malicious widgets We have already said that our focus is not on the Runtime Environment and thus we will focus only on the essential features.

Because we will implement our system into the Tribler P2P Program, which is written in Python, the Runtime Environment will have to be written in either Python or a language that is easily integrated with Python, such as C. When we choose a language and file format for the widget, it is important that the Runtime is able to parse the files and execute the language. There are several interpreters and compilers written in Python, for example Spidermonkey [6] is able to execute Javascript. None of them, however, are far enough developed to be used without any problems. Therefore, we chose to use the Python interpreter itself to execute the widgets. This means the widget must at least contain one or more Python files. Other files could be metadata files (which contain for example the author, a Widget ID, version number) and resource files such as images. We chose to not support metadata and resource files and only support widgets that consist of one Python file. This is more than sufficient to create great widgets and it can easily be extended later on.

The widgets, which are essentially Python modules, have access to all the Python libraries and all the code of Tribler. This means the Widget API is already abundant and the developer can implement a lot of features, when he knows about them, but can also misuse a lot of features. Thus, on the one hand there should be enough developers resources to be able to create interesting widgets, but we should also take care of security issues. RestrictedPython [7] or the Pythons Restricted Mode [8] could be used to restrict the API of the widgets, to make them more secure. However, RestrictedPython is inactive for a few years already and Python's Restricted Mode has several security vulnerabilities. We decided to implement security measures in other ways, namely code signing, whitelisting and a rating system.

Code signing means that the author of the widget and the widget integrity can be verified. Thus, when a widget is malicious, we know for sure that the author is really the author and that he or she really intended this.

Whitelisting can be done by certificates. Certain peers are designated to be initially trusted and they may create whitelist certificates to whitelist a widget author. We can then distinguish between trusted widget authors and untrusted widget authors.

A rating system will be implemented in the Widget Market, where users can rate and review widgets. These reviews and ratings can be used by other users to check if they really want a widget. However, a rating system alone would not suffice. The reasons are as follows. First, there may be false ratings and comments. Second, users have to try the widget to rate or review it, which means they must first install the malicious widget before they notice it is malicious. But then it may be already too late. However, the three security measures combined should take care of most of the abuse.

3.1.2 Discovery and Download

The Widget Discovery and Download subsystem is a key element of the Widget System and collaborates closely with the Widget Market, which is used to find, browse, rate and review the widgets. It is only needed for widgets in a P2P environment, because the discovery and download of widgets in a centralised repository is trivial. The Discovery and Download subsystem should take care of the discovery and download of widgets using the P2P System in a scalable and decentralised manner. With the design of the subsystem, the browsing requirement of the widget repository should be taken into account.

The discovery and download strategies are the biggest design choices to be made. We will first discuss the discovery strategies. There are three alternative scalable and decentralised strategies. First, it is possible to use a DHT to index the repository. For example, it is possible to use the first letter of the widget as a key to a list of widgets. This way, it is possible to browse the repository alphabetically, by querying the network for widgets per letter. This strategy is quickly disposed, because a DHT is still lacking performance in practise [24]. Second, it is possible to disseminate the whole widget index, replicating it on every peer. This can be done quite scalable by gossipping the right data, as PlanetP [15] shows. We know that the widget repository will probably not be that large, thus this could be a valid strategy which also allows us to browse the repository easily. Third, it is possible to use the existing features of Tribler's BuddyCast, which disseminates infohashes and torrents. When we create a torrent for each widget and start seeding, BuddyCast automatically disseminates the torrent files in the network. We only need to distinguish widget torrents from others and on receipt of a widget torrent, notify the Widget Market. However, BuddyCast only maintains 5000 torrents in the local database. But with a few modifications in BuddyCast and TorrentCollecting, we could make a bias towards widget torrents, such that they are disseminated faster. Because we now use torrents, the BitTorrent download protocol can be used easily for downloading. We chose to use the third discovery strategy, because it seems the most beneficial. It reuses code, which is a good thing in software development, and BuddyCast and BitTorrent already proved their usefulness.

We distinguish two extremes in download strategies, namely the Download-On-Install and the Download-On-Discovery strategies. The Download-On-Install strategy takes the least space on disk, but might take a lot of patience from the user when he wants to install a widget. The Download-On-Discovery strategy takes the most disk space and the most bandwidth, because it will eventually have most of the widgets on disk, but instalment will take almost no time. We chose for a strategy that is most similar to the Download-On-Discovery. However, we will only download one widget at the time, to be able to control the bandwidth usage. We select the most popular widgets to collect first, as this will decrease download time on installation in most of the cases. It does not make sense to collect uninteresting or dead widgets.

3.1.3 Widget Communication and Storage

The Widget Communication and Storage subsystem takes care of the local storage per widget, finding intra-widget communication peers in the P2P network and handling the intra-widget communication messages. The local storage and communication are combined in one subsystem, because they are so closely related. One can create a Communication API that has an abundance of features, but these will not be used when there is not a simple and effective way to store this information. Both the local storage and Communication API can be designed such that there is a lot of freedom for the widget or they can be very strict. For example, we might design the local storage to support 5 key-value pairs per widget, or to support a full database where tables can be created and queried as one would like.

The Communication API might support functions for retrieving communication partners, direct communication and broadcast. This way, there is little control over the messages and bandwidth to be communicated. It could also support gossipping in a way that the system controls the bandwidth. This can be done by letting the widget implement functions for handling received messages and for the creation of a gossip message. The system then chooses the communication partners and is able to control the bandwidth by checking the message size and choosing the message interval.

We have chosen to support the gossip system, to have more control over the widget communication. Also, in a P2P system without social overlay, direct communication does not really make sense and makes implementing widget communication too difficult. The Widget Communication and Storage subsystem takes care of selecting the gossip partners and controls the bandwidth. The message structure is left to the widget itself. Because widgets are essentially small programs with one theme, we think it is enough to support one database table per widget. The table structure can be defined in the widget such that its data is completely customised. With an easy API for inserting, deleting, selecting and updating the storage table, a widget that communicates should be easily created.

Finding communication partners for a widget can be supported in two obvious ways. First, BuddyCast data contains information on who is seeding which torrent. Since widgets also have a torrent which it seeds when the widget is installed, this information is automatically propagated by BuddyCast. Another way is by hooking into the swarm: retrieving peers from the tracker, DHT or PEX messages. The peers that are seeding are likely to have installed the widget. Of course, the swarm seems the most active and up to date, thus this will be the primary source for widget communication partners.

3.1.4 Widget Market

The Widget Market is the part of the system that allows users to browse, install, rate and review widgets. It collaborates closely with the Discovery and Download subsystem. After the widget to be installed is downloaded, the Widget Runtime Environment is called to install the widget. Therefore, the Widget Market's primary focus is on the user interface and the dissemination of ratings and reviews. For the design of the user interface of the Widget Market, we refer to Section 3.2.4.

The dissemination of ratings and reviews can be done in numerous ways. A few possibilities are: using a DHT to look up comments per torrent infohash, extend PEX messages to exchange ratings and reviews per swarm, extending the BuddyCast message with ratings and reviews. We did not choose any of these methods because of the following reasons. First, we have been showing the bad performance of DHTs already throughout this chapter. Second, extending PEX would require the user to join the swarm of the torrent it wants the ratings and comments of, and then wait until we got most of the ratings and comments. We would like to know the comments and ratings fast, thus this solution does not seem valuable. Third, extending BuddyCast with ratings and reviews would work for every torrent, not only widget torrents. This is a good thing, but would probably not exchange the ratings and reviews fast enough because there would be so many. Currently, to find the torrent files you want, it is still necessary to query your neighbours for information too. Such a remote query would probably also be necessary for ratings and reviews.

We, however, chose a different direction: the Widget Market should be a widget itself, and use the Widget Communication and Storage features to disseminate and store ratings and reviews. The reasons are as follows. First, the Widget Communication will only exchange information with peers that have the same widget. This implies that the bandwidth of other peers is not wasted and every exchange of information is useful for the receiver. Communication is thus a lot more effective than for example using BuddyCast, which would exchange ratings and reviews for millions of torrents, most of which the receivers of those ratings and comments will not look at. The second reason is that using this setup we can easily show the effectiveness of the Widget Communication and Storage subsystem.

3.2 Technical Design

In this section, we give the details of our design. We start by presenting the architecture of the system and then discuss each subsystem in detail. In Figure 3.1, the class diagram of the major components are visualised with their connections to the associated Tribler components. BuddyCast, MetadataHandler, TorrentCollecting and the OverlayBridge are original components of Tribler, which are used by or modified for our system.

3.2.1 Runtime Environment

The WidgetPanel, WidgetCore and WidgetDBHandler together form the Widget Runtime Environment. The WidgetPanel is the panel where the widgets are shown on. The WidgetCore is used for monitoring widget downloads for installation, reading the widget files and installation of the widgets. The WidgetDBHandler is the database handler for both the Widget and the WidgetInstance tables. Apart from the usual insert, update and delete functions, it has more advanced functions such as calculating a suitable free position on the WidgetPanel for a widget using other widget locations and sizes, and selecting a popular widget for downloading.

When a torrent is received from the MetadataHandler, various torrent information is stored in the database. Widget torrents are enriched with specific widget data and this is stored in the Widget database table. The fields it includes can be viewed in Figure 3.2. This information is primarily used by the Widget Market.

Upon installation of a widget, the widget filename is retrieved from the database and the file is read. The widget Python module is imported and a widget class instance is made. A suitable position on the WidgetPanel is calculated and when

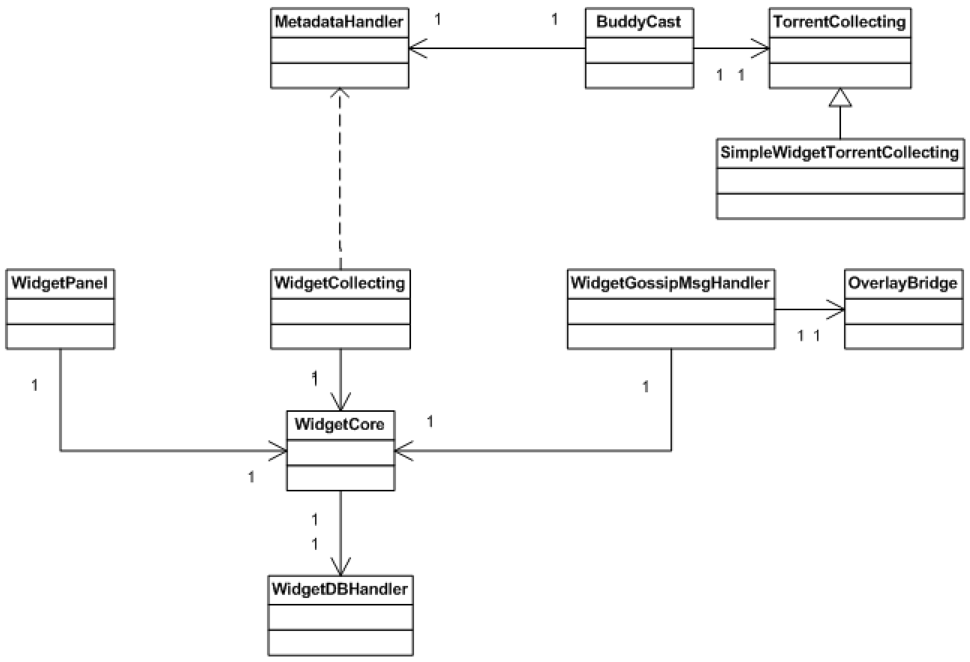


Figure 3.1: Class Diagram of the Widget System in relation to Tribler

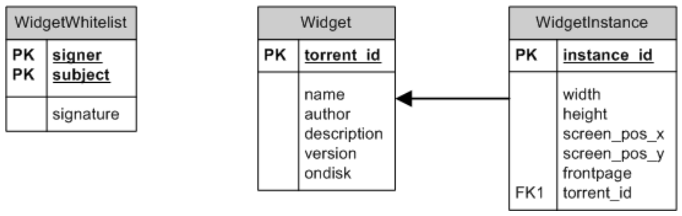
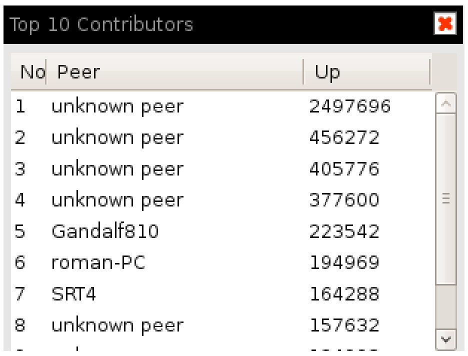


Figure 3.2: Database design of the Widget System.

the widget instance, including size and position information, is inserted into the database, it receives its unique instance ID. The widget is then encapsulated inside a widget wrapper, adding a title bar with close button to the widget and for security reasons: it is harder to get to the WidgetPanel from the widget wrapper, because the WidgetPanel is not the parent of the widget. A screenshot of a widgets interface including its title bar is shown in Figure 3.3.

Now that the widget is installed, the WidgetCore initialises the local storage and gossip features of the widget, by creating a database table and local storage database handler, and notifying the WidgetGossipMsgHandler of this widget. It is now possible to use the whole Tribler API and Python libraries, without intervention of the Runtime Environment. Also, the WidgetCore starts seeding the widget file, for finding gossip communication partners and to allow fast downloads for new users of the widget.


Top 10 Contributors

No	Peer	Up
1	unknown peer	2497696
2	unknown peer	456272
3	unknown peer	405776
4	unknown peer	377600
5	Gandalf810	223542
6	roman-PC	194969
7	SRT4	164288
8	unknown peer	157632

Figure 3.3: Interface of a widget, including title bar.

Upon removal of the widget instance, its instance is removed from the Widget-Panel and WidgetGossipMsgHandler, and removed from the WidgetInstance table. The seeding of the widget file is stopped and the user is removed from that swarm, but the Widget file itself is kept on disk. This enables fast re-installation without the need for downloading the widget again. Upon re-installation, the local storage is lost because that table is dropped when the widget instance is removed.

After realising it would be useful to have widgets that do not necessarily have to be shown on the WidgetPanel (we will call them frontpage widgets), we made it possible to create widgets that have a menuitem in the Extras-menubar in Tribler. For example, the Widget Market widget is not a frontpage widget, but has an ”Add/Remove widget” menuitem, which displays the Widget Market is a separate dialog. However, it is also possible to use both features (a frontpage widget with a menuitem), or none, which could be a silent addition to functionality of Tribler.

Widget Format

Widgets consist of one Python file, thus one module. However, the widgets Python file should conform to a specific format. First, it is necessary put the following metadata in the widget module: name, author, version, description. Second, the frontpage and menuitem options can be set. They default to a frontpage widget without menuitem. For frontpage widgets, the width and height must also be set. Further, there must be a *widget* class in the file, that extends the *tribler_widget* class, which is basically a panel with several function stubs, such as OnClose, OnPostInit, OnCreateGossipMessage, OnHandleGossipMessage, and OnMenu. In the init function, the user interface must be created. In this widget class, the gossip option can be enabled, and the local storage structure must be set. Further, the widget developer is left free to put whatever he wants in the Python file.

Security

As discussed, we choose to implement three security measures, namely code signing, whitelisting and a rating system. Code signing uses the facilities provided by Tribler to sign a torrent with the users PermID. When the widget is inserted into Tribler, a torrent is created and the torrent is automatically signed with the publishers PermID. Using this signature it is possible to verify the publisher of the widget. Further, the infohash in the torrent is used to check the integrity of the widget code.

Whitelisting is done much like public key infrastructures. At first, every widget is untrusted. There are a few initially trusted peers, that can create certificates to whitelist widget authors. In a whitelist certificate, the public key (PermID) of the widget author is put and it is then signed by the initially trusted peer. Signing is done by first adding the signers PermID and the date to the certificate, creating a signature of this certificate and adding the signature to the certificate. Validating a certificate can be done by first removing the signature from the certificate and then verifying the certificate, signer PermID and signature. Initially, only whitelisted widgets are shown in the Widget Market, but there is an option to show every widget, including the untrusted ones.

The rating system is one of the primary security measures for the widget system, because it makes use of the wisdom of the crowd. We will discuss the technical details of the rating system in Section 3.2.4.

3.2.2 Discovery and Download

Widget discovery is done by first creating torrents for the widgets and seeding them in Tribler. BuddyCast then automatically disseminates the widget torrents just like the other torrents. A BuddyCast message includes infohashes of the torrents it is seeding and the infohashes it has collected most recently. Each receipt of a BuddyCast message triggers the TorrentCollecting module to select one random torrent to download from the other peer. By extending the lists of infohashes with the type of the torrent (video, music, document, widget, et cetera), we can differentiate the widgets from other file types. Therefore, TorrentCollecting can be more effective and specialised. For example, when the user is primarily interested in movies, it can create a bias by selecting more movie torrents to collect than other torrents. Thus, this BuddyCast extension is not only useful for this Widget System, but for Tribler as a whole. We, however, created the SimpleWidgetTorrentCollecting, an extension of the TorrentCollecting module that selects either a widget torrent or another type of torrent. The widget torrent that will be selected is the most popular widget locally known; the other types of torrents are still randomly selected. The calculation of the popularity of torrents is discussed at the end of this subsection.

The MetadataHandler is the Tribler component that handles the downloading of torrent files. When a torrent is downloaded, the extra widget information that is stored in the widget torrent (name, author, version, description) is added to the database and will be available to the Widget Market from then on. The MetadataHandler then informs the WidgetCollecting module that there is a new widget torrent available.

WidgetCollecting is the component that is dedicated to downloading the most popular widgets on discovery, one at a time. Widgets can also be downloaded when the user selects a widget to be installed which is not yet collected. When WidgetCollecting is initiated, it first checks whether there are any widgets being downloaded. These downloads are checked whether they are being downloaded for installation or not. When there is a widget that is not being downloaded for installation, WidgetCollecting continues to monitor that widget. If there is no widget currently being collected, it queries the database for the most popular widget locally known that is not yet collected, and starts to download and monitor that widget. When there are no widgets locally known that are not yet collected, it waits for the MetadataHandler to send a notification of a new widget torrent and starts to download that widget.

When the widget is collected within the time limit, a value that can be adjusted but is set at a 5 minutes default, the procedure to find a new widget to collect is restarted. But when the widget download takes too long, it tries to find another widget to collect. If another widget is found, the current widget is marked as being a slow download, and starts downloading the other widget instead. If not, it will reset the time limit and proceed with the slow download. Bandwidth limitations for widget collecting are currently not implemented, as widgets are so small they can be downloaded without notice with current Internet connection speeds. However, because WidgetCollecting uses the standard BitTorrent Download API of Tribler, WidgetCollecting could easily be extended with bandwidth limitations, when necessary. The widget collecting process is illustrated by the state machine in Figure 3.4

Widget Popularity

The Widget TorrentCollecting and WidgetCollecting both make use of the popularity of widgets. A natural value for the popularity would be the swarm size of the torrent, or a derivative of this. Since some trackers and peers lie about the number of seeders and leechers, giving false popularity's, it is necessary to verify this information. This is the full-time job of another researcher in the Tribler team. He extended BuddyCast such that peers exchange their latest information about the swarm. This swarm size information contains the number of seeders, leechers and the number of locally met Tribler peers who are seeding this torrent. The swarmsize information from all encountered peers is stored with timestamps in the Popularity table.

For ranking torrents by popularity, we average the latest number of peers, the number of Tribler seeds seen by this peer, and the number of locally known Tribler seeds. The number of locally known Tribler seeds is added to reduce the effect of lying peers. The value for each torrent is saved per torrent in Tribler's Torrent table, and updated when new popularity information is received.

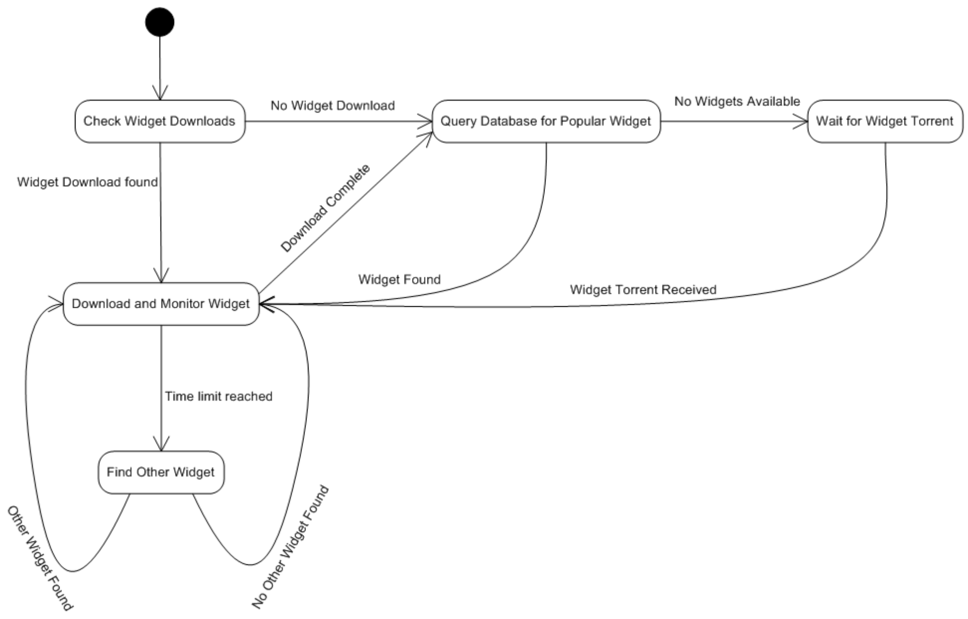


Figure 3.4: State machine of the WidgetCollecting module.

3.2.3 Widget Communication and Storage

The Widget Communication and Storage subsystem encompasses two things. First, it must take care of the initialisation of the communication and local storage. Second, it must allow the widgets to communicate with each other by finding the communication candidates and dispatching messages.

The first task is done by the WidgetCore upon installation of the widget. The widget specifies the local storage structure within the Python file as a list of tuples. The first entry of the tuple is the column name and the second entry the column type. The primary key of the table can be specified by placing the keyword ”pkey” behind the column type of the primary key columns. Valid column types are ”integer”, ”text” and ”numeric”. The WidgetCore validates the local storage structure and dispatches the creation of the database table to the WidgetDBHandler. When the local storage table is initialised, the widget gets a database handler variable, which can be used to manipulate the table. The widget must also specify whether it would like to use the gossip features. When this is the case, the WidgetCore notifies the WidgetGossipMsgHandler of this new widget.

The WidgetGossipMsgHandler takes care of all the intra-widget communication. It keeps a list of all the widget instances and a queue of their connection candidates. Every gossip round, it calls the function of the widgets that create and return their gossip message, validates the message, selects a gossip candidate per widget and tries to send the message. Before the message is sent, the message is prepended with the WIDGET_GOSSIP type and the widget infohash. Similar to BuddyCast, it keeps a Send Block List and Receive Block List per widget.

When a message is sent to/received from a particular peer, the peer is kept in the Send/Receive Block List for the block interval, respectively. When the peer is removed from the list, it is added to the end of the connection candidate list. After selecting a gossip communication partner from the connection candidate list, we shuffle the list to randomise peer selection.

Upon receipt of a WIDGET_GOSSIP message, the widget infohash is taken from the message and it is checked whether the widget is installed. If not, a WIDGET_NOT_INSTALLED message will be replied. If the widget is installed, the appropriate widget message handler is called. After handling the message, the peer is added to the Receive Block List. When a WIDGET_NOT_INSTALLED is received, the peer is removed from that widgets connection candidate list and the Tribler Preferences table is updated. When a message could not be sent, the peers number of tries is increased and the peer is added to the end of the list. If the number of tries is greater than three, the peer is removed from the connection candidates.

Connection candidates are added from BuddyCast messages and Swarm connections. Upon receipt of a BuddyCast message, it is checked whether the other peer has installed widgets we have installed too. If this is the case, we add the peer as connection candidate for the widget. When a swarm connection is made, we also check whether this is a widget swarm or not. If it is, the connection partner is added. Because the swarm connection is already established, we do not have to create an additional connection and thus keep the number of connections to a minimum. Further, we know from the peer that he is online because we just communicated with him. When Tribler is started, initial connection candidates are retrieved from the Preference table of Tribler. This table stores information on who is seeding which torrents.

3.2.4 Widget Market

To accomplish the goals of creating a Widget Market, we created a management widget. A widget is ideal for this, because it requires a lot of user interface code and the Widget Communication and Storage features for the widgets are ideal to disseminate reviews and ratings.

Dissemination of Widget Reviews and Ratings

We must define the local storage structure to store the reviews per widget, and choose a gossip message format. We define a review as a comment and a rating on a widget. The local storage structure can be seen in Table 3.1. Because we limit each user to rate and review each widget only one time, we must use the PermIDs of the users and the infohashes of the widgets as primary key. Further, we added a human readable name, because not every Tribler user knows the name of all PermIDs. The rating, comment and date fields are self-explanatory. The clock integer is used to create an order in the reviews, such that a review with a reply to another user is shown beneath that user. When a rating is inserted, its clock value must be higher than the maximum clock value in the table for the corresponding infohash.

The gossip message consists of the 5 most recent reviews and 5 random reviews. Each review has the same format as the local storage structure.

Field	Description	Type
infohash	infohash of the widget	text
permid	permid of rating/comment author	text
name	name of the rating/comment author	text
comment	the comment	text
rating	the rating	integer
date	local date when this rating/comment was added	numeric

Table 3.1: Local Storage Table structure for storing the Widget Market's widget reviews.

Graphical User Interface

The Widget Market widget is not shown on the frontpage of Tribler, but has a menuitem called ”Add/Remove widgets”. This saves space for the widgets the user really wants to see on the frontpage. The widget has a dialog with three tabs, where the user can perform several actions.

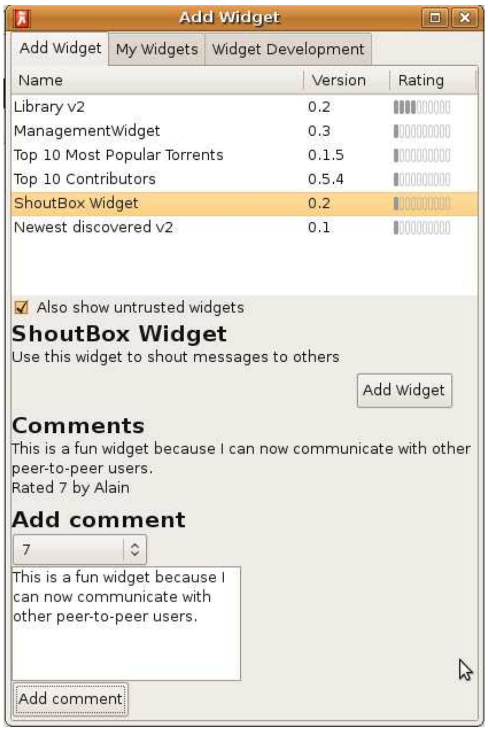
On the first tab, the widget market can be browsed, by viewing a sorted list of widgets. They can be sorted alphabetically or by average rating. There is an option to show every widget, because initially only whitelisted (trusted) widgets are shown in the list. Widgets can be selected to show more detailed information about the widget and the individual reviews and ratings of other users. The user can also add one review and rating per widget.

On the second tab, the installed widgets are listed and they can be removed easily by selecting a widget from the list and pressing the remove button.

The third tab enables widget developers to add their own widget to the Widget Market. They can select a widget file, and this file is then validated. The user can then choose to either test the widget or add it to the Widget Market. A overview of the graphical user interface is shown in Figure 3.5.

3.3 Review of the Widget Discovery Method

For a successful Widget Market browse function, most of the popular widgets should be discovered within a reasonable time. This bootstrapping is of utmost importance, because otherwise users will have to wait too long and loose interest in the widgets. After testing our software using multiple computers, we concluded


Add Widget

Add Widget | My Widgets | Widget Development

Name	Version	Rating
Library v2	0.2	▮▮▮▮▯▯▯▯▯▯
ManagementWidget	0.3	▮▯▯▯▯▯▯▯▯▯
Top 10 Most Popular Torrents	0.1.5	▮▯▯▯▯▯▯▯▯▯
Top 10 Contributors	0.5.4	▮▯▯▯▯▯▯▯▯▯
ShoutBox Widget	0.2	▮▯▯▯▯▯▯▯▯▯
Newest discovered v2	0.1	▮▯▯▯▯▯▯▯▯▯

✓ Also show untrusted widgets

ShoutBox Widget

Use this widget to shout messages to others

Add Widget

Comments

This is a fun widget because I can now communicate with other peer-to-peer users.
Rated 7 by Alain

Add comment

7

This is a fun widget because I can now communicate with other peer-to-peer users.

Add comment

Figure 3.5: Graphical User Interface of the Widget Market

that the widget discovery method did not meet our bootstrap requirement: it took way too long to discover one widget. We decided to perform a calculation to check whether this is right.

BuddyCast downloads the information on 600 torrents in the first hour of Tribler start up and stores a maximum of 5,000 torrents on each peers disk [22]. Now we assume that there are about 2,000,000 torrents already being disseminated by BuddyCast. The largest torrent tracker, The Pirate Bay, tracks around the same amount of torrents [5]. As seen in Table 1.1, most widget repositories do not exceed 5,000 widgets, only the very popular Facebook does. Let us take 5,000 for the amount of widgets in our repository. There is a difference in BuddyCast exchange with a peer that uses an old Tribler version and a peer that uses the up-to-date version.

An exchange with a peer with an old Tribler version will result in randomly selecting one of the torrents it has to download, because its BuddyCast message does not have type information on the infohashes it sends. This results in a chance of 0.0025 for a random torrent to be a widget torrent and this is the same chance of selecting a widget torrent to collect.

An exchange with a peer that does exchange type information, the chance that a widget torrent will be collected is 0.5. At least, if there are widgets infohashes in this BuddyCast message. In the message are the infohashes of the peers seeds, with a maximum of 50, and 50 infohashes of most recently collected torrents. When the user has installed one or more widgets and is thus seeding them, the message automatically contains widget torrent infohashes. When he has not, it depends on the collected torrents. It is not possible to say how many users will have installed a widget, nor is it feasible to calculate the chance that there are widget torrents in the collected torrents infohashes. This calculation will result in a recursive chance calculation, because the chance of finding one or more widgets in the collected torrents infohashes depends on what the chance is to collect a widget torrent (result of this calculation). Therefore, we assume there always is a widget torrent in the BuddyCast message.

By running Tribler multiple times for an hour and logging the versions of the Tribler peers encountered, we noticed that the average chance of encountering a peer with the newest version is 0.4. Thus, in the first bootstrap hour we should receive:

$$600 * (0.4 * 0.5 + 0.6 * 0.0025) = 120.9$$

widget torrents on average. This calculation is telling us that there is no need to worry about the bootstrap. The problem with our test, however, is that we only had a few computers running this version and only a few widgets: all encounters with BuddyCast peers were peers with old versions.

With 5000 widgets, our assumption that each BuddyCast message of an up-to-date peer contains at least 1 widget torrent might be right, but with only 50 widgets this is probably not the case. We want to deploy the system with only 10 initial widgets created by us. Further, as can be seen in Figure 3.6, there are about 2000 active Tribler users with the newest version, when it is deployed. Many users do not even consider creating widgets, thus 5000 widgets in the repository is way too much. We decided to extend the Widget Discovery for fast bootstrapping, even when there are just a few users with the newest Tribler version and even when there are only a few widgets in the repository.

3.3.1 A Widget Discovery Extension

The problem with the original widget discovery method was that when there are not many widgets or not many peers that use widgets, most BuddyCast encounters are with peers that either do not send the type of the torrent in the message or do not have any widget torrents. To solve this problem, we need an overlay with peers that use widgets. This is in fact accomplished by our own intra-widget communication!

By extending the Widget Market's gossip message to include infohashes of five of the widgets it is seeding and thus either has the widget installed or it is published by himself. When a gossip message is received, the TorrentCollecting is then triggered with these infohashes. The Widget Market widget is installed by all users with the widget version of Tribler, thus every gossip encounter is successful when the user has widgets installed. Further, because only infohashes of seeded widget torrents are sent, WidgetCollecting will almost certainly succeed to collect the widget.

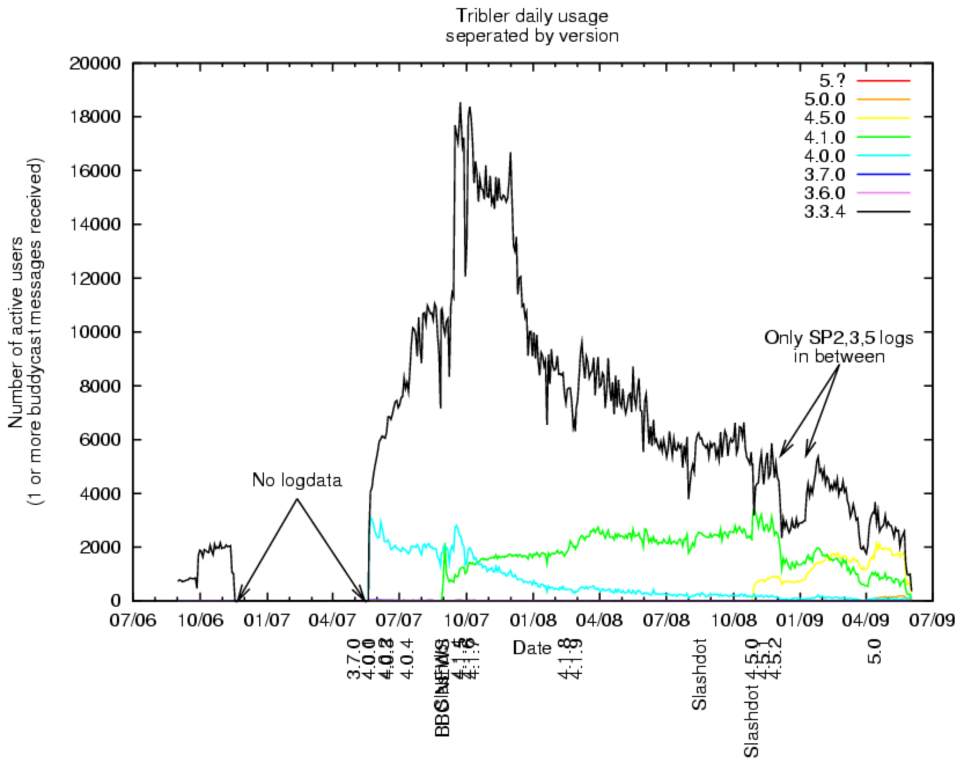


Figure 3.6: Tribler daily usage statistics

In the next chapter, we will discuss the implemenation of our complete P2P Widget System, including example widgets and a Widget Developing Center.

Chapter 4

Implementation

In this Chapter, the implementation of the widget system is reviewed in Section 4.1 and the example widgets are discussed in Section 4.2. Further, we discuss our Widget Developing Center, a resource for widget developers, in Section 4.3.

4.1 Widget System Implementation

In Table 4.1, the different parts of the implementation are shown, including the number of lines of code it has cost. As shown, most of the code is written for the Widget Runtime Environment. Especially the methods for installation of widgets cost quite some lines of code. However, most of the code has been written for the User Interface of the Widget Market. This is logical, because the user interface is quite extensive, with all kind of hooks that make it interactive and user friendly.

It is also very interesting to see that the Comments and Ratings Dissemination only takes 22 lines of code, because it uses the Local Storage and Intra-Widget Communication subsystem. Apparently, the Intra-Widget Communication interface is written in such a way, that it does not take a lot of effort to write something worthwhile. Also, the Widget Market Discovery Extension only takes 9 lines of code! This extension, discussed in Section 3.3, also uses the Intra-Widget Communication and triggers the TorrentCollecting to collect the discovered widget torrents.

To test the implementation, we have written several unit tests that check whether the code does what it needs to. However, it is hard to test distributed systems and user interfaces with unit tests. The user interface is therefore tested manually and the distributed aspects of our system are analysed in Chapter 5.

4.1.1 Widget Statistics Gathering

Making our system ready for actual deployment, means that code must be written to gather statistics about the usage of the Widget System. Using these statistics,

Subsystem	LOC
Widget Runtime	**1305**
Database	276
WidgetPanel	310
Widget Installation	495
Widget Whitelist Certificates	141
Tribler User Interface Extensions	83
Widget Market	**916**
User Interface	894
Comments and Ratings Dissemination	22
Widget Discovery and Download	**481**
WidgetCollecting	269
TorrentCollecting	77
BuddyCast-extensions	92
Popularity	34
Widget Market Discovery Extension	9
Local Storage and Intra-Widget Communication	**487**
Local Storage API	74
Intra-Widget Communication	413
Unit Tests	**559**
Statistics Gathering	**130**
Total	3878

Table 4.1: Implementation Statistics of the Widget System

we can evaluate several aspects of the system. Which aspects can be evaluated, are determined by the statistics that are gathered.

For the gathering of statistics, we created a WidgetCrawler. It is based upon the Crawler framework that is already in Tribler, and is therefore quite small. The WidgetCrawler is ran by every peer to respond to crawler query messages. These query messages are sent by the actual crawler, and the responses are logged into a file. We chose to gather the information that is shown in Table 4.2

Using this information, we can already answer tons of questions about the system. A few among them are shown below. By gathering the information over a long period of time, we can create a function of the answers against the time, showing exactly how well the system is working.

- How many widgets does each peer know about?
- How many widgets are collected per peer?
- How many users know about a particular widget?
- How many users have collected a particular widget?

Field	Description
infohash	The unique identifier for the widget
installed	A boolean telling whether the widget is installed or not
ondisk	A boolean telling whether the peer has collected the widget
nrcomments	An integer counting the number of comments it has about the widget

Table 4.2: The fields with information that is gathered by the WidgetCrawler

- How many users have installed a particular widget?
- How many comments are there for a particular widget?
- What is the total amount of widgets in the system?
- What is the total amount of comments?
- What is the average amount of comments per widget?
- How many widgets are installed on average? How much percent of the total amount of widgets is this?
- How many widgets are collected on average? How much percent of the total amount of widgets is this?

4.2 Example Widgets

To demonstrate the systems capabilities, we have created several widgets. Of course, the Widget Market is also created as a widget, but this widget is already extensively discussed in Chapter 3. The other example widgets are discussed in this section. Furthermore, we will show some implementation statistics of the widgdes.

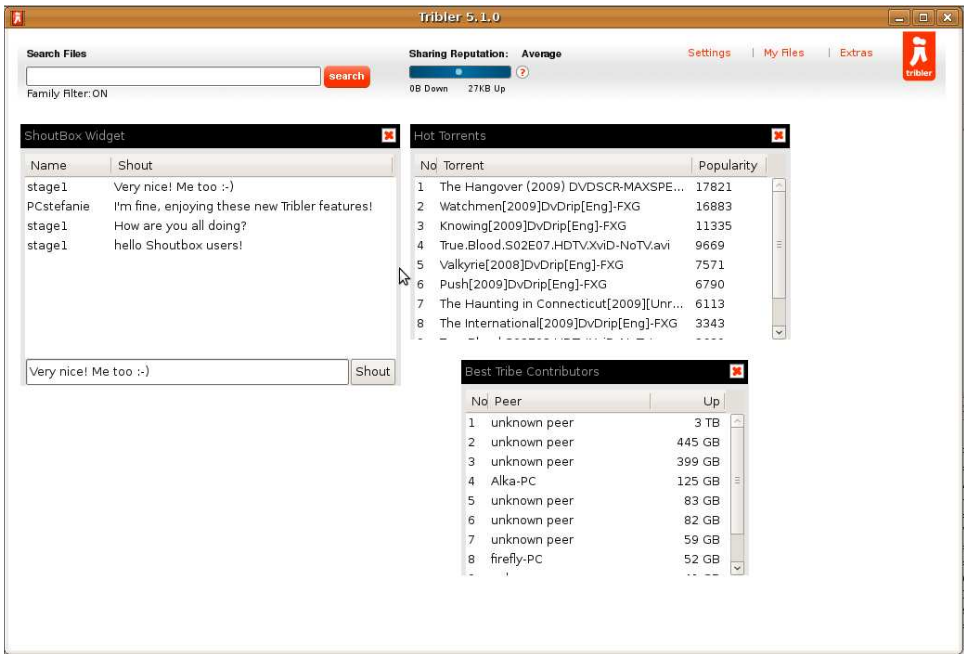
4.2.1 Implementation of the Example Widgets

For each widget, we will discuss what the widget does, the user interface, and the implementation of the widget. In Figure 4.1, the user interfaces of several widgets are shown.

ShoutBox Widget

The ShoutBox Widget is a widget that uses both the local storage and intra-widget communication to provide social interaction between P2P users. Users can leave a shout and the last ten shouts are displayed to everyone that uses the widget.

The widgets user interface is quite simple, as most widgets interfaces are. It consists of a listbox with two columns, one for the name and one for the shout.



Name	Shout
stage1	Very nice! Me too :-)
PCstefanie	I'm fine, enjoying these new Tribler features!
stage1	How are you all doing?
stage1	hello Shoutbox users!

Very nice! Me too :-)

No	Torrent	Popularity
1	The Hangover (2009) DVDSCR-MAXSPE...	17821
2	Watchmen[2009]DvDrip[Eng]-FXG	16883
3	Knowing[2009]DvDrip[Eng]-FXG	11335
4	True.Blood.S02E07.HDTV.XviD-NoTV.avi	9669
5	Valkyrie[2008]DvDrip[Eng]-FXG	7571
6	Push[2009]DvDrip[Eng]-FXG	6790
7	The Haunting in Connecticut[2009][Unr...	6113
8	The International[2009]DvDrip[Eng]-FXG	3343

No	Peer	Up
1	unknown peer	3 TB
2	unknown peer	445 GB
3	unknown peer	399 GB
4	Alka-PC	125 GB
5	unknown peer	83 GB
6	unknown peer	82 GB
7	unknown peer	59 GB
8	firefly-PC	52 GB

Figure 4.1: A screenshot of Tribler, including the ShoutBox, Hot Torrents and Best Tribe Contributors widgets.

Beneath the listbox, there is a textfield to type the shout and a button to actually send the message.

The local storage structure consists of three columns; the name, the shout and a clock value. The name and shout are self-explanatory. The clock is used to create a message order. When a new shout is inserted, the clock value is one higher than the current maximum clock value in the database. This way, messages that are replies to other messages get a higher clock value and are positioned higher than the others. Messages with the same clock value may be ordered differently on different computers. The gossip message consists of the ten most recent shouts. When a gossip message from someone else is received, its shouts are either inserted or ignored when they already exist.

The ShoutBox widget is one example of a simple, yet powerful widget. The widget consists of 63 lines of code, including 14 lines of whitespace.

New Torrent Notifier Widget

The New Torrent Notifier widget shows the most recently locally discovered torrents by BuddyCast. It does not use any local storage or intra-widget communication, but instead hooks into the notifying system of Tribler.

The user interface is again very simple. The listbox, where the torrents are shown, is the most prominent GUI element and takes up most of the widgets space.

It has one column, which shows the torrent name. A download button is placed beneath the listbox.

By registering an observer to be notified on widget insertion, the widget receives its torrents. The function that is called on insertion of a widget, receives the info-hash of the torrent. We can then retrieve the torrent name and filename from the Tribler Torrent database. The name is inserted in the list and a mapping from torrent name to filename is stored, to be able to download the torrent. The list is then purged, such that at most 10 torrents are shown in the widget. When an item is selected to download, we retrieve the torrent filename and call the session to start the download. This widget implements the OnClose event handler, which removes the observer before the widget is removed.

Since the widget is hooking into Tribler, this widget is somewhat more advanced than the ShoutBox Widget. It consists of 87 lines of code, with 13 lines whitespace.

Hottest Torrents Widget

The Hottest Torrents Widget displays the hottest torrents known at that time. It uses the information known locally, collected by BuddyCast, and combines this information with the information of other Hottest Torrents users using intra-widget communication for fast convergence.

Again, the widget displays the hottest torrents in a listbox, with three columns. First column shows the place of the torrent, the second the name of the torrent and the third the popularity. This is the same popularity value as is discussed in Section 3.2.2.

Upon initialisation, the user interface is created. Upon OnPostInit, which is called by the runtime when the local storage is initialised, the top ten of most popular torrents is retrieved from the Tribler database and stored in the local storage of the widget. Also, a timer is started to update the information of the widget with the Tribler database every few minutes. Updating the database means that the top ten of torrents from Tribler is merged with the top of the widget: new torrents are added, old torrents are updated and finally the list is purged.

The local storage structure consists of the torrent infohash, torrent name, popularity and checktime, which is the local time of the peer when it retrieved the information from the Tribler database. The torrent infohash is the primary key. Intra-widget communication is used to exchange the peers hot torrents and synchronise with with the other users. Upon creation of a gossip message, the ten hottest torrents are added to the message. Upon receipt of a message, this top ten from the other peer is merged, taking into account the checktime values.

The Hot Torrents Widget is built in such a way that the convergence is much faster than when only BuddyCast is used. In just a few minutes, it is possible to show the top torrents of the whole Tribler overlay. The more users there are, the faster the convergence. By using the checktime value, torrents that are becoming less popular, will eventually disappear from the list, because the torrents swarmsize is occasionally checked by Tribler peers. The widget consists of 150 lines of code, of which 48 lines are either whitespace or comments.

Best Tribe Contributors Widget

This widget has basically the same architecture as the Hot Torrents widget. It takes a top ten, of BarterCast peers in this case, and synchronises it with other peers and occasionally with the local Tribler database, converging to the global top ten of BarterCast's most altruistic peers.

This time, the interface shows the place of the user, his name and how much he has uploaded, according to Tribler. Because not every peers nickname is known, a lot of names are filled with "unknown peers", but by synchronising with other peers, these names will eventually be filled with their real nicknames.

The local storage also consists of three columns: a permid, name and up. Up is the number of bytes uploaded according to Tribler. The primary key consists solely of the permid. Upon creation of a message, the current top ten is added to the message. When a message is received, the information in the message is merged with the currently known top. Either an entry is updated, meaning that up and name are taken from the message, or the entry is added. The highest value of up is always taken, since the value can only increase. A timer is used to merge the Tribler BarterCast information into the widgets top every minute. The widget is implemented in 143 lines of code, with 39 lines of whitespace or comments.

4.2.2 Statistical Information on the Widgets

The widgets are all implemented in less than 150 lines of code, with a minimum of 63 for the ShoutBox Widget. Also, 6 lines are used to specify the metadata, such as author, description, et cetera. Creation of the user interface takes about 10 lines of code and the intra-widget communication functions (OnCreateGossipMessage and OnHandleGossipMessage) are all implemented within 8 lines of code! A summary is shown in Table 4.3.

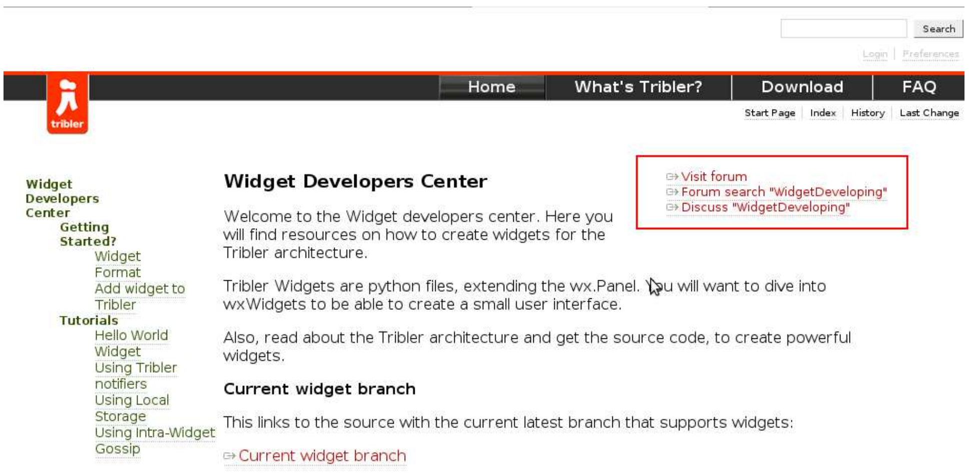
Most of the widgets use the local storage and intra-widget communication features, but only the ShoutBox creates direct social interaction between users. The Tribler API is used in all widgets but the ShoutBox Widget, but only the New Torrents Notifier uses the Notification API.

Widget	LOC	Whitespace	LOC for Communication	Size (KB)
ShoutBox	63	13	5	2.3
New Torrents Notifier	87	13	–	3.8
Hot Torrents	150	49	8	6.3
Best Tribe Contributors	143	39	8	5.6

Table 4.3: Implementation statistics of the example widgets.

4.3 Widget Developing Center

Although creating a widget is quite easy, we have also created the Widget Developing Center with information on how to develop P2P Widgets. We have created several tutorials which range from a HelloWorld widget to using the Local Storage and Intra-Widget Communication APIs. All tutorials follow a foolproof step-by-step explanation of what is happening, including example code. Further, the Widget Format is explained in detail and it is explained how the widgets can be inserted and tested within Tribler. A screenshot is shown in Figure 4.2. The Widget Developing Center consists of several online wiki pages and thus can easily be accessed by anyone using a browser.



Widget Developers Center

Welcome to the Widget developers center. Here you will find resources on how to create widgets for the Tribler architecture.

Tribler Widgets are python files, extending the wx.Panel. You will want to dive into wxWidgets to be able to create a small user interface.

Also, read about the Tribler architecture and get the source code, to create powerful widgets.

Current widget branch

This links to the source with the current latest branch that supports widgets:

Current widget branch

Figure 4.2: Screenshot of the Widget Developing Center.

Chapter 5

Experiments and Results

Because the widget system is a distributed system, testing it to see whether the requirements in Section 2.2 are met, is a difficult task. Although we conduct experimental research and thus want to deploy the system to obtain data usage data and derive properties from these data, we first have to make sure the system works well enough to be deployed. We already tested the runtime environment, but to analyse the distributed properties of the system, we devised several experiments.

The experiments and their setup are explained in Section 5.1 and the results are shown in Section 5.2.

5.1 Experiments and Setup

We have devised five experiments, which will give us several properties about widget discovery and download and intra-widget communication. They require two to three peers with the widget system installed in the Tribler overlay. Before doing the experiments, bandwidth usage logging is added to the WidgetCollecting and Intra-Widget Communication, such that bandwidth can be measured throughout the tests. The experiments are to be executed multiple times to get an average, because P2P systems can be unpredictable.

Experiment 1: How fast are widgets disseminated from one peer to another?

This experiment tests both the widget discovery and download functionality of the widget system. The experiment can be used to find out how fast the discovery and download of widgets is and how efficient they both are. It requires two peers, A and B. Peer A adds three widgets to the repository. Time and bandwidth usage are measured until B has all three widgets discovered and downloaded.

Experiment 2: How fast are widgets disseminated from two peers to another?

This experiment tests the same aspects as Experiment 1, but uses three peers. Peer A and B both have three widgets discovered and downloaded. Peer C comes online and the time and bandwidth usage are measured until he has downloaded all three widgets.

Experiment 3: What is the normal bandwidth usage without extra widgets?

We want to know what the bandwidth usage is, when a peer does nothing and thus has only the Widget Market widget installed. We use three peers which all have three widgets already disseminated. Further, several fake reviews are added to one of the peers local storage, making sure that the bandwidth is used to its fullest extend. The bandwidth usage is then measured for 30 minutes.

Experiment 4: How fast are intra-widget communication partners for a widget found?

We would like to know how long it takes to provide a new peer with widget data. Two peers are participating in this test. Peer A has the ShoutBox widget installed and adds a shout. This information is to be disseminated to the other peer. Time is measured from the moment that peer B installs the ShoutBox widget, until it has the shout from peer A.

Experiment 5: How fast and efficient is the intra-gossip communication?

This experiment uses a modified version of the Hot Torrents widget. The Hot Torrents widget retrieves from the Tribler database the top 10 most popular torrents locally known at that time. Each Hot Torrents widget instance exchanges its local top 10 with other instances, and merges them until it shows the global top 10 from all users that use this widget. Normally, also updates from the local database would again be merged with its own top 10, but for this experiment we will not do this. The Hot Torrents widget is further modified such that every change in the top 10 is logged.

The experiment uses three peers, which all install the modified Hot Torrents widget at the same time. Thus, they all show their locally known top 10 to the user. Time is started when they installed the widget and stopped when all three peers have the same top 10, that is the result of the merge of their local top 10. Bandwidth usage is also measured.

5.2 Results

The results of the performed experiments are discussed here per topic.

5.2.1 Widget Dissemination

To measure the widget dissemination speed and the bandwidth usage, we have performed the first two experiments four times each. We will first discuss the results of Experiment 1 and then the results of Experiment 2, since this is an extension of the first experiment.

From the results of Experiment 1, we can derive multiple statistics. In Figure 5.1, we can see the time it takes before the other peer has been found and the bandwidth the Widget Market widget uses until all three widgets are collected. Half of the time, the communication partner is found within the minute, but in the worst case it took almost seven minutes. It is interesting to see that the maximum bandwidth usage stays below 1 KB at all times. This has two reasons. First, the Widget Market was not filled with any comments and ratings, thus the messages were very light. Second, the bandwidth for downloading the widgets is not added. We used three widgets, namely the ShoutBox, Hot Torrents and Best Tribe Contributors. Together, these widgets are 14.3 KB in size, which is also downloaded.

To get more insight into the time it takes to collect the widgets, we created Figure 5.2. This graph shows how much time it took to first find the communication partner, and then collect the widget. Time to collect the widgets varied from one to two minutes, but the time to find the communication partners has a lot more variation. Further, the average time to download a widget was 6 seconds. It takes a minute to download them all, because the peer did not know about the other torrents. This is because the TorrentCollecting module is only triggered once after we receive a message. The gossip interval is set at one minute, thus we have to wait for the next message to know about the next widget.

The results of Experiment 2 seem remarkable. Using two peers, both already having the widgets collected, seems to stabilise the collecting of widgets. We did this experiment four times, and at all times it took about one minute to find communication partners and the widgets were collected in about 100 seconds, give or take 10 seconds.

5.2.2 Normal Bandwidth Usage

By performing Experiment 3, we measured the normal bandwidth usage when peers do not use any features of the Widget System. This still means, that widgets are collected and widget reviews are disseminated. We tried to measure the bandwidth usage for the system when it is fully loaded, meaning that the size of each message sent by the Widget Market is fully used. We simulated the fully used Widget Market by manually entering as much fake widget reviews as needed in peer 1. Further, we disabled the Send/Receive Block Lists, such that every round, a communication partner is chosen.

In Figure 5.3, the bandwidth is shown for 2200 seconds for all three peers. As can be seen, Peer 1 has reached a maximum of about 60KB in those 2200 seconds. This is only 0.027 KB per second! Of course, collecting widget torrents and widgets also costs bandwidth. Suppose the average widget is 5KB. This is quite reasonable since all example widgets, we have created range from 2 to 6 KB, as is shown in Table 4.3. If a widget is collected after every gossip communication, thus every minute, we use 5KB per minute, resulting in 0.083 KB per second. Thus, our

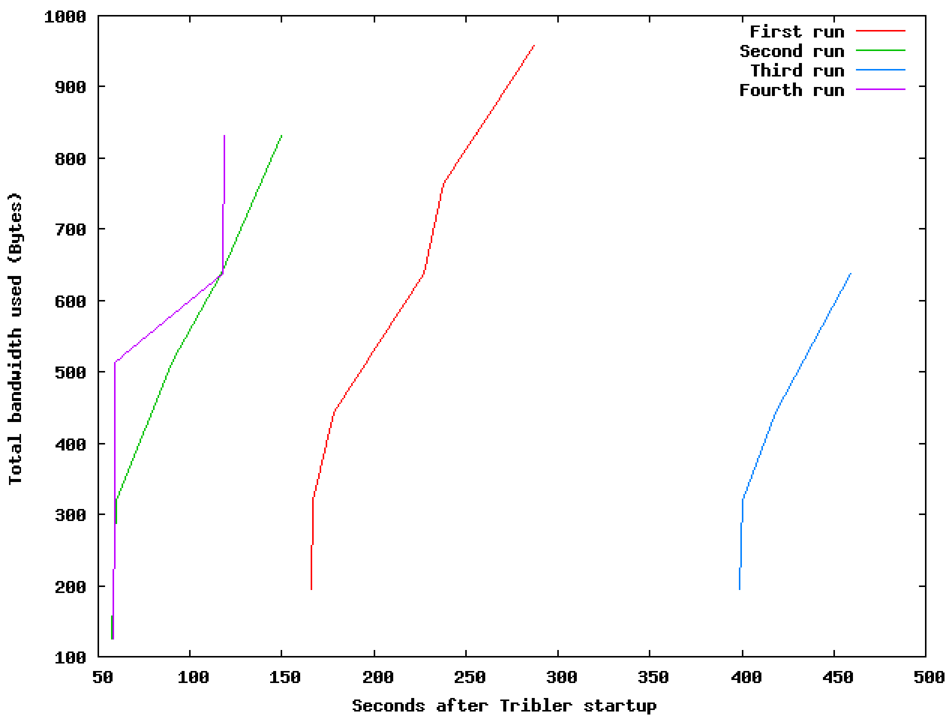


Figure 5.1: Total time and bandwidth usage for collecting three widgets

extensions to the system only use 0.11 KB per second, when the system is at full power and uses maximum bandwidth.

One anomaly can be detected from the graph in Figure 5.3, namely that Peer 2 does not communicate at all for about 15 minutes! This is probably because of the connection candidate list shuffling, explained in Section 3.2.3. With only two peers in the list, shuffling the list might not be such a good idea, especially without using the block lists. However, with more communication partners it would make sense.

5.2.3 Finding Intra-Widget Communication Partners

To measure the speed of finding the Intra-Widget Communication partners, we performed Experiment 4 four times. This resulted in an average of 67 seconds until the first message was sent/received. This means that it takes about one minute to join the swarm after installing a widget, contacting the tracker to find peers and to connect to them. Of course, it also depends on when the widget is installed within the gossip interval.

The variation in the results was also not very high. This is differs from the results of Experiment 1, where the variation was quite high and time it took was normally longer. This might be because Tribler still needs to bootstrap when the Widget Market widget is installed, while the ShoutBox widget is installed later.

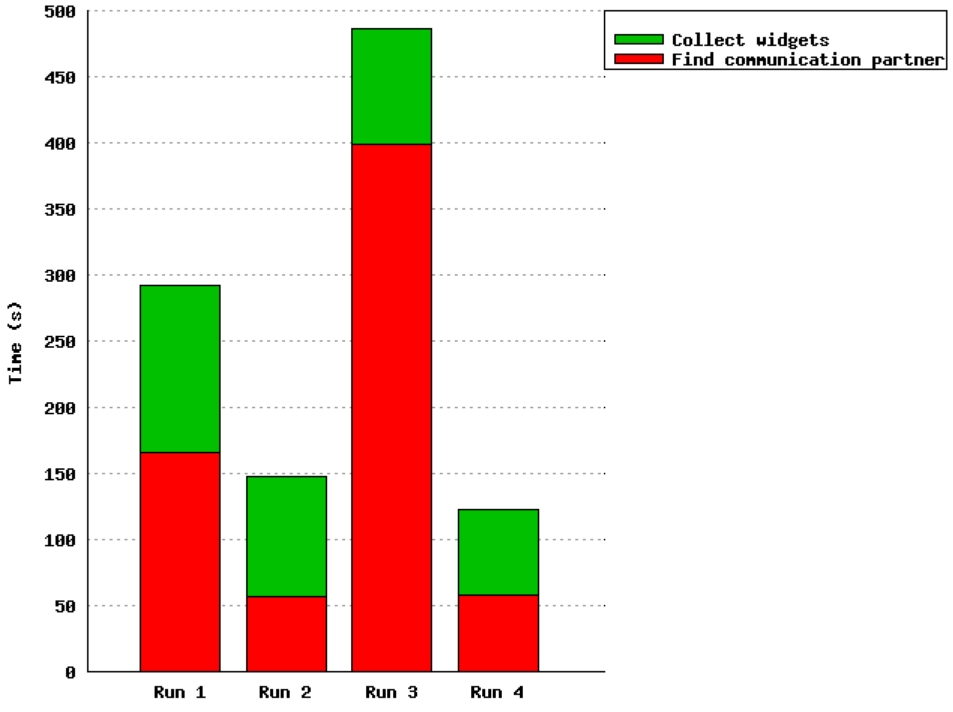


Figure 5.2: Total time to collect three widgets divided into different stages

5.2.4 Intra-Widget Communication

The question of the speed and efficiency of this section is answered by performing Experiment 5, which we performed thrice. Let us first address an issue found by performing the experiment, and then lay out and discuss the results. Normally, when a Top 10 is being formed by merging the results of 3 independent entities, it would take at least 3 exchanges. First, merging user A and user B's Top 10. Then merging the result of A and B with C, and then sending the result to the other peer. However, the experiment was not set up to be sure that all users have information that would be merged into the final Top 10. Therefore, two of the three times we tested it, 2 exchanges were enough. First, merging user A and B's Top 10 and then sending the result to user C, which did not have any information that survived the merging. However, this can also be the case in a real deployment and is only an advantage, because less communication is necessary. Therefore, we did not try to modify the experiment setup later on.

In Figure 5.5, the results are shown. The average time it took to get the merged top 10 is 53.9 seconds. The average number of exchanges necessary to get the final result was 1.2. The size of all messages were around 1 KB. Measuring the bandwidth, we came to an average of 0.047 KB per second.

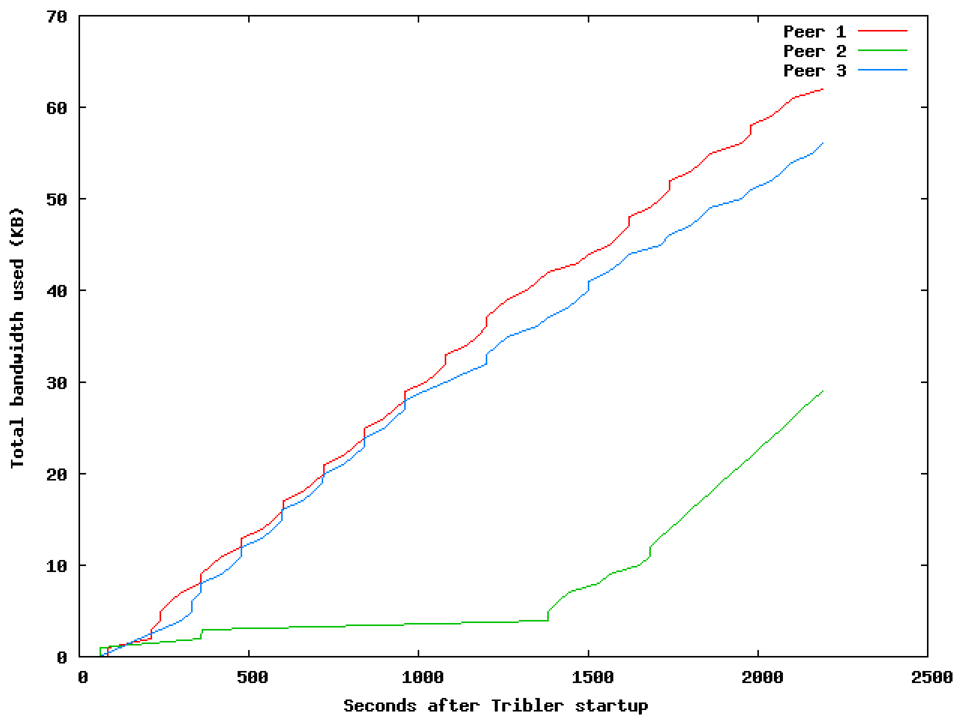


Figure 5.3: Normal bandwidth usage for three peers

5.2.5 Discussion of the Results

When analysing the results of the experiments, we think our system behaves quite well. It is fast and uses just a little bit bandwidth, compared to the Internet speeds nowadays. Of course, the more widgets are installed, the more bandwidth is used. But when one of the heaviest widgets (the Hot Torrents widget from Experiment 5) only uses 0.047 KB per second. This is certainly not a big deal. Also, the user that does not want to use any widgets is only affected by losing 0.11 KB per second.

As for the undeterministic browsing requirement: most of the times, it takes just a few minutes before the first widgets start to appear in the list. When the widget discovery is initialised, the widget discovery works very fast. This is mostly because of the Widget Discovery Extension. This is shown in the logs, since the WidgetCollecting is triggered just a few seconds after a new incoming message was found. The widgets are so light that they are easily downloaded, unnoticeable for the user.

One anomaly seems to have been discovered when using three peers. The graphs show that sometimes a peer is not communicating with the others, probably because of the shuffling of the connection candidate list and the disabling of the block lists in Experiment 3. This problem is alleviated when there are more connection candidates, but can be solved by not selecting the last connected peer(s), as implemented by the block lists.

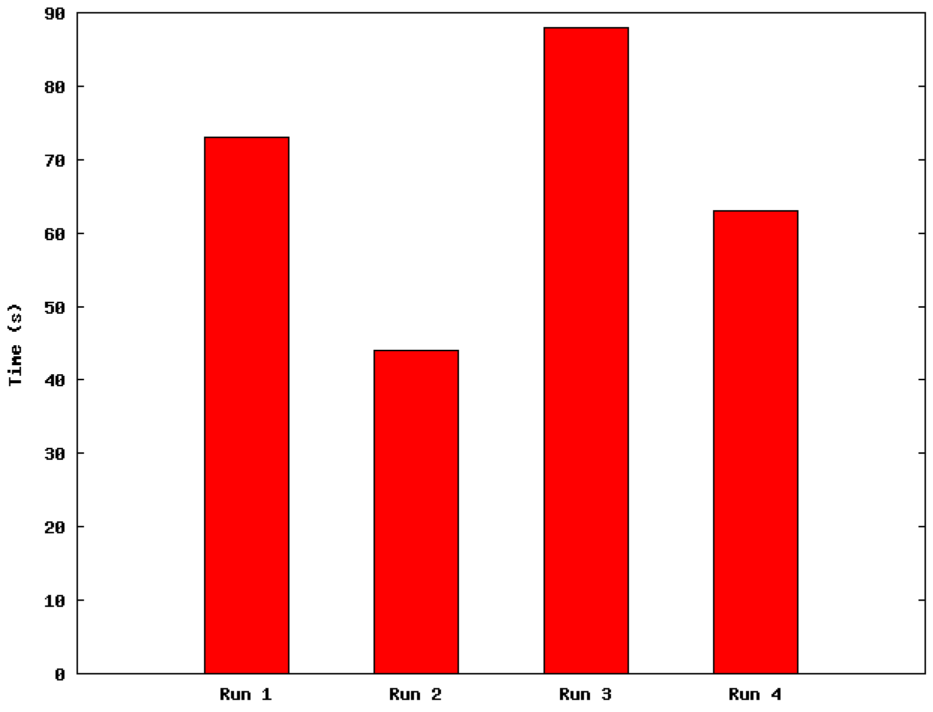


Figure 5.4: Time to find intra-widget communication partners.

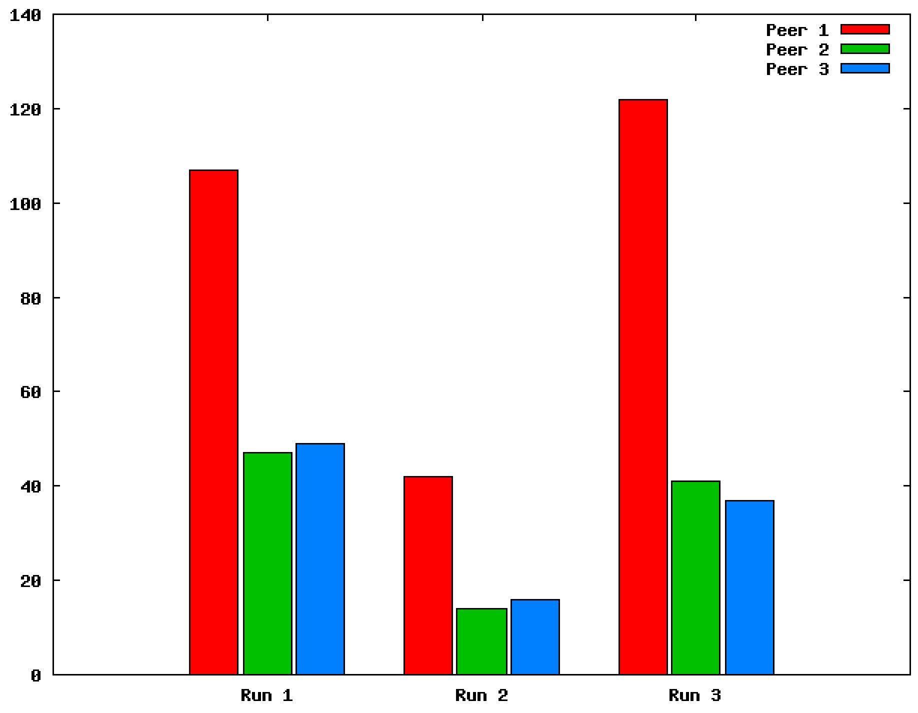


Figure 5.5: Time taken to merge the three different top 10's

Chapter 6

Conclusions and Future Work

6.1 Summary and Conclusions

In this thesis, we presented our complete solution for P2P widgets. Our solution closes the gap between the World Wide Web and P2P. On the one hand, P2P technology can be used to create scalable and robust Internet applications, without any central authority. On the other hand, the World Wide Web is becoming more and more social and interactive. Our Widget System combines all these features, because it is designed for scalability and robustness.

We have shown a few of the millions possible widgets that can be made in Chapter 4, which create interaction between the users of the system. By using the Tribler API, the Local Storage API, and the Intra-Widget Communication API, it is possible to create even more interesting widgets than was possible with other types of widgets!

Our P2P Widget System is designed to be deployed and to gather statistics about the system. The implementation is also ready to be deployed and even has an online Widget Developers Center to aid widget developers in creating widgets. Because of time constraints however, the system is not actually deployed in a large user environment. Though this is very unfortunate, the system can be deployed by the Tribler team to perform further research on this subject.

While we did not deploy the system, we can still conclude properties from our system, because of the experiments that were performed in Chapter 5. In this chapter, we have shown that our system is fast and bandwidth efficient. Moreover, the user controls how much features it wants and thus also controls how much bandwidth the system uses. Browsing the Widget Market is easily done and will show a multitude of widgets in a matter of minutes after bootstrapping Tribler, because of the fast Widget Discovery and Download module. Comments and Ratings are disseminated fast and without using too much bandwidth, because it uses our own Intra-Widget Communication API.

With the results of our evaluation, we can conclude that our P2P Widget System is:

- **Scalable and fast**. We have shown that within the first few minutes, widgets are already discovered and downloaded. It is scalable, because it uses low-bandwidth gossipping to discover the widgets and the scalable BitTorrent download engine to download the widgets. With a repository of about 2000 widgets with on average a size of 5KB, the total size of the repository is only 10 MB.

- **Bandwidth efficient**. Without any installation of widgets, the user will need about 0.1 KB/s bandwidth to be able to use the system. Further, most widgets also do not need more than 0.1 KB/s to be able to work and the user is able to control the number of widgets it has installed and can thus control its bandwidth usage.

- **Innovative**. Because of the unique design of the Local Storage and Intra-Widget Communication API, which supports the automatic formation widget swarms, new *scalable social and interactive* features are introduced without the need to overload Web 2.0 applications, as current types of widgets do.

- **Easy to use**. Not only is the user interface for browsing and installing the widgets easy, creating a widget is also easy. As shown in Section 4.2, simple widgets are created within 50 lines of code. Communication and Local Storage is introduced to a widget with less than 10 lines of code!

Further, we think that this P2P Widget System is able to provide P2P researchers a more powerful tool to design and test P2P applications.

6.2 Future Work

Although the P2P Widget System presented in this thesis is a giant step towards a social and interactive P2P experience, there are also many things that can be improved. Because people with a critical view can make the list of possible improvements infinite, we will discuss the most important ones according to us.

- **Intra-Widget Communication can be more secure**. Currently, no validation of the gossip messages is implemented, although the architecture supports it easily. Also, the size of a gossip message may vary freely. These two things can be implemented, but will also greatly reduce the possibilities of widgets. Questions on how to implement these features arise. For example, what happens when a gossip message is rejected because of its size or invalidation? How to validate a message without creating some sort of standard for messages? These questions are not answered easily and greatly increase the complexity of the Intra-Widget Communication API.

- **Intra-Widget Communication's bandwidth can be more controlled**. The bandwidth for this could be controlled for all widgets together or per widget. A User Interface could be made where the user may prioritise the widgets. Important widgets could get more bandwidth, or the bandwidth could be set manually per widget. A more loose solution could be to control the total bandwidth rate for all widgets. The system can then use a queue for messages and wait to send when it already met its quota for this time interval. Using Round Robin algorithm, the queue could stay fair for each widget. More sophisticated methods are also possible.

  Another option is to create multiple bandwidth modes for widgets, just as is done in BuddyCast. BuddyCast starts with a bootstrapping period, communication a lot in a small time window, but decreases the bandwidth requirements when the system is bootstrapped.

- **The Widget Runtime Environment can be better secured**. This is addressed several times in this thesis, where we already presented that code restrictions or other forms of security is hard to achieve, without greatly reducing the freedom or leaving security gaps to be exploited.

- **Version Management could be introduced**. Finding the latest version of a widget you are currently using, automatic updating of widgets, showing a changelog for each widget are just a few features that are possible with Version Management. Currently, a new version of a widget is just treated as another widget. Using a universally unique identifier (UUID), Version Management is possible. The problem is however, when is a widget a new version of an existing widget and when is it another new widget? Can you update automatically to a new widget version, even if this widget is created by someone else than the widget author? Even if this might be someone you do not trust. Multiple users could create different versions for a widget: which one is the real one?

- **Widgets that extend other widgets**. When widgets can be extended by other widgets, or when widgets can use features of other widgets, a real self-managing and evolving environment could be made. However, this requires a careful design and a lot of Runtime Environment programming. Widgets should become more like plugins, small applications that may extend whichever part of the system. It requires a far more sophisticated widget structure and metadata.

- **Direct Peer-to-Peer Communication could be introduced**. This would innovate the system even more and an obvious application is chatting. Currently however, peers are anonymous entities to each other, which does in fact make direct communication illogical. Who would we choose to communicate with, if peers do not have any identity? When social networks are introduced to P2P networks, which is already being researched by the Tribler researchers, this does in fact make a lot of sense. If you have friends, you could communicate with them. The widgets could get access to a Social API, just like the Social Widgets in social network sites such as Facebook, MySpace and Hyves. Social networking is already very popular, and Social P2P Widgets might get more users to use P2P technology!

We believe that the first steps can be taken by implementing Version Management and bandwidth control for Intra-Widget Communication. These features are more easily designed and implemented than the other features, which would take more than a year to research and implement.

Bibliography

[1] Azureuswiki: Peer exchange. `http://www.azureuswiki.com/index.php/Peer_Exchange`.

[2] Facebook apps. `http://www.facebook.com/apps/directory.php`.

[3] Google desktop gadgets. `http://desktop.google.com/plugins/`.

[4] Microsoft gadgets. `http://gallery.microsoft.com/sidebar/vista.aspx`.

[5] The pirate bay. `http://www.piratebay.org`.

[6] Python-spidermonkey. `http://code.google.com/p/python-spidermonkey/`.

[7] Python's restrictedpython. `http://pypi.python.org/pypi/RestrictedPython/3.4.2`.

[8] Python's rexec module. `http://docs.python.org/library/rexec.html`.

[9] Wikipedia: Widget engine. `http://en.wikipedia.org/wiki/Widget_engine`.

[10] Yahoo! widgets. `http://widgets.yahoo.com/`.

[11] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[12] F.R.A. Bordignon and G.H. Tolosa. Gnutella: Distributed System for Information Storage and Searching Model Description. *Journal of Internet Technology, Taipei (Taiwan)*, 2(5), 2002.

[13] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, pages 46–66, 2001.

[14] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, volume 6, 2003.

[15] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. Planetp: using gossiping to build content addressable peer-to-peer information sharing communities. In *Proc. 12th IEEE International Symposium on High Performance Distributed Computing*, pages 236–246, 22–24 June 2003.

[16] Cuong Do Cuong. Seattle conference on scalability: Youtube scalability. Video, June 2007.

[17] Ipoque GmbH. Internet study 2008-2009. Internet Study, 2008-2009.

[18] Cisco Systems Inc. Approaching the zettabyte era. White Paper, Cisco Systems Inc., June 2008.

[19] M. Jelasity and M. van Steen. Large-scale newscast computing on the Internet. Technical Report IR-503, VU, 2002.

[20] A. Norberg and L. Strigeus. libtorrent: extension protocol. `http://www.rasterbar.com/products/libtorrent/extension_protocol.html`.

[21] J.A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D.H.J. Epema, M. Reinders, M.R. Van Steen, and H.J. Sips. TRIBLER: a social-based peer-to-peer system. *Concurrency and Computation: Practice and Experience*, 20(2), 2008.

[22] J.A. Pouwelse, J. Yang, M. Meulpolder, D.H.J. Epema, H.J. Sips, G.J.M. Smit, and M.S. Lew. Buddycast: an operational peer-to-peer epidemic protocol stack. In *Proc. of the 14th Annual Conf. of the Advanced School for Computing and Imaging*, pages 200–205. ASCI, 2008.

[23] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[24] Jelle Roozenburg. Secure decentralized swarm discovery in tribler. Master's thesis, Delft University of Technology, November 2006.

[25] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *In Middleware*, pages 329–350, 2001.

[26] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM New York, NY, USA, 2001.