An Overview of Technologies for Peer-to-peer Widget Environments

Alain M. van den Berg

Delft University of Technology

An Overview of Technologies for Peer-to-peer Widget Environments

Research Assignment Computer Science

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

Alain M. van den Berg

3rd December 2008

Abstract

This report considers techniques that can be used to create a peer-to-peer widget environment. It considers the deployment of such a widget system in a peer-to-peer environment and the distribution of the widgets using the same peer-to-peer environment. First, several deployed centralized widget engines are examined and requirements for a decentralized widget engine are gathered. Then, existing techniques to decentralize the distribution are described and possible runtime environments are layed out. Lastly, various security vulnerabilities which may occur in peer-to-peer environments with portable code are discussed.

Preface

This document is the report of my literature study, the preparation for the Master of Science project in the Parallel and Distributed Systems Group of the Delft University of Technology. The report reviews several widget systems which use a central distribution site and discusses techniques on how to create such a widget system in a peer-to-peer environment, where the distribution is also done in a decentralized way using the same peer-to-peer environment.

I would like to thank my supervisors dr. ir. D.H.J. Epema and dr. ir. J.A. Pouwelse for their advice and guidance.

Alain M. van den Berg

Delft, The Netherlands
3rd December 2008

Contents

Preface **v**

1 Introduction **1**

2 Currently deployed widget systems **5**
- 2.1 Eclipse Plugin System 5
- 2.2 Google Gadgets 8
- 2.3 Facebook Apps 11
- 2.4 Firefox Extensions 14

3 Zero-server widget repository **19**
- 3.1 Directory service for widgets 19
  - 3.1.1 Review of peer-to-peer directory service systems 20
  - 3.1.2 Comparison of DHT and gossip based systems 23
- 3.2 Moderation in a distributed environment 24
  - 3.2.1 Distributed rating systems 25
  - 3.2.2 Moderation without a rating system 28
- 3.3 Package management 28
  - 3.3.1 Storing and finding multiple versions 29

4 Widget runtime environment **31**
- 4.1 Runtime environment comparison 31
  - 4.1.1 Running code with an interpreter 31
  - 4.1.2 Compiling the code 32
  - 4.1.3 Other execution environments 32
- 4.2 Code restriction 33
  - 4.2.1 Methods of restriction 33
  - 4.2.2 Controlling resource usage 34
- 4.3 Creating a peer-to-peer API for widgets 35

5 Security issues in a peer-to-peer widget environment **37**
- 5.1 Pollution 37
  - 5.1.1 Problem description 37
  - 5.1.2 Solutions 38

5.2 DDoS attacks 39
5.2.1 Problem description 39
5.2.2 Solutions 40
5.3 Malicious code 42
5.3.1 Problem description 42
5.3.2 Solutions 43

6 Conclusions **47**
6.1 Summary and conclusions 47
6.2 Further research 48

Chapter 1

Introduction

Currently, peer-to-peer applications are quite popular and a vast amount of internet traffic is resulting from peer-to-peer applications. Instead of the common client-server architecture, every user is essentially a client and a server. They connect to each other, managing large amounts of data while maintaining reliability and scalability. Apart from their popularity, they are also very interesting to research because of their properties. Widgets also emerge on the internet, where they can be found on the majority of the social networks (e.g., Facebook and MySpace), blogs or any other website. They are also found on mobile phones and on desktops (e.g., Google Gadgets, Yahoo Widgets or Vista Widgets). This literature study is about peer-to-peer widgets and we would like to start with discussing the definition of a peer-to-peer widget. Before we can do this, we have to find the correct definition of *widget*, since this term is rather ambiguous in the computer environment.

Looking up the term for meanings in the computer world[1], one finds two definitions:

1. an element of a graphical user interface such as a button or a scroll bar.
2. a module of software for a personalized Web page.

The first definition, however, is not the kind of widget we are looking for and the second definition is rather narrow. Wikipedia[2] distinguishes, apart from the Graphical User Interface (GUI) Widget in the above definition, three different types of widgets:

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.

[1] http://dictionary.reference.com/browse/widget
[2] http://en.wikipedia.org/wiki/Widget_engine

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.

All these definitions have in common that they are a portable chunk of code to be installed and executed in a specialized environment. The difference between them is in which environment they are used. Desktop Widgets and Mobile Widgets use a widget engine to run the various widgets. The widgets can be downloaded and plugged into the widget engine. The widget engine provides an API (Application Programming Interface) to facilitate the widget accessing the application data and functionality. The widgets can be created by third parties by using the API and plugging it into the widget engine. Examples are Google Desktop Gadgets and Yahoo Widgets. Web Widgets do not have to be downloaded, but can instead just be added to your page, blog or social profile by embedding a bit of HTML code. An example of some Desktop Widgets can be seen in Figure 1.1.

To distribute created widgets to other users, central widget portals are used. Mostly these are websites where users can easily find and download widgets, find documentation, rate or comment on widgets and a lot more. For example, Google Desktop Gadgets actually provides a portal where you can not only browse through the widget directory, but also view the top contributors of widgets, giving an incentive to create widgets. In short, these portals are becoming a social community around the application.

Peer-to-peer widgets, derived from the definition of a widget, are portable pieces of code that can be added to a peer-to-peer client to extend the functionality, provide information or give a bit of fun. While some peer-to-peer clients already provide a way to extend the application (e.g., Azureus[3]), they still use a central distribution site. Distributing widgets in a decentralized way creates two additional issues. First there is the issue of security: everybody is able to distribute his own (possibly malicous) widget. Second, the distribution and storage of the widgets is a problem in its own. Creating a distributed portal with the same functionality as current centralized widget portals (such as a list of most popular widgets) requires state of the art techniques.

Porting widgets to a peer-to-peer environment also provides the possibility to let widgets interact with other widgets by using an overlay on the peer-to-peer network. While this is an exciting new possibility, it also provides a new way to exploit the application.

Lastly we would like to point out that extending an application with new functionality or services is not new. Most applications nowadays have the possibility to extend their features by installing components called extensions, plugins or components. There are many commonalities with widgets, but also a few differences. Plugins are mostly used for applications that were created with a certain objective in mind. For example, browsers to browse the web, chat clients to chat, music

[3] http://azureus.sourceforge.net

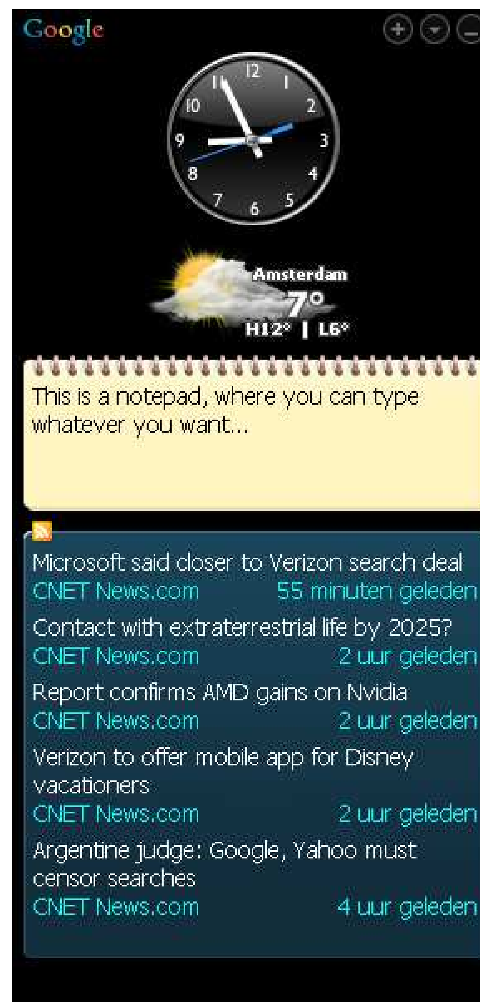

Figure 1.1: Google Desktop Sidebar, including widgets displaying a clock, the weather, a notepad and an RSS reader.

players to play music, etc. Plugins extend the functionality or provide a service that would help reaching the objective of the host application, e.g., a Winamp plugin would typically extend the player to play a different kind of music format or a browser plugin that lets users view PDF file formats from their browser. Widgets, on the other hand, provide a more general service: showing the time, a notebook, a calendar, a small game, etc. Also, widgets generally have a small area on the user interface to display their information, while plugins might be hidden from the user interface or completely integrated. However, the difference is more in meaning than technology: both plugins and widgets extend an application and thus require a container engine to contain the components, both run portable chunks of code. Some plugin systems, like the Eclipse Plugin System, are just a container engine

with some plugins delivered with it, closing the gap between widgets and plugins.

In the peer-to-peer context, differences between widgets and plugins become even smaller, as both are extensions to the functionality of the client. The peer-to-peer environment creates possibilities for other features, which might use the social aspect of peer-to-peer, e.g., one could create an extension to support peer-to-peer radio stations (adding to the functionality of the peer-to-peer client) or one could make a social game which uses an overlay to communicate with friends (social, general). As we see it, peer-to-peer widgets are as different from other types of widgets as from plugins.

The report is continued as follows. In Chapter 2, some widget systems including their central distribution sites are reviewed. In Chapter 3, we will discuss the issues of distributing the widgets in a peer-to-peer manner and in Chapter 4 the widget runtime environment is discussed. Further, we will discuss various security aspects in Chapter 5. Finally, we conclude the report with conclusions in Chapter 6.

Chapter 2

Currently deployed widget systems

Currently, there are many widget systems for various kinds of applications. These widget systems differ in several aspects, such as the environment the widgets run on (the web, a social network or within an application), what functionality is possible and how they are created. However, they all use a central distribution site to distribute widgets. In this chapter some of the currently deployed widget systems will be evaluated. For every widget system we will cover the technical aspects (their runtime environment), their distribution method and security.

In Section 2.1, the Eclipse Plugin System is reviewed. Section 2.2 considers the Google Gadgets and in Sections 2.3 and 2.4 the Facebook apps and Firefox extensions will be discussed, respectively.

2.1 Eclipse Plugin System

Eclipse[1] is an open source community focusing on building an extensible development platform for building, deploying and managing software and is entirely coded in Java. Eclipse has a unique setup that makes its extensibility possibilities enormous. The whole program is based on a Platform Runtime which is basically a plugin engine, while the rest of the program functionality is coded into plugins. A plugin is the smallest unit of the Eclipse Platform which can be coded and delivered separately. A feature that makes Eclipse so unique is that plugins can provide extension points, which other plugins may extend such that there is a hierarchy in the plugin system.

We will first treat the plugins in more detail, then discuss the distribution of Eclipse plugins using the Eclipse Plugin Central and finally we will discuss security issues.

[1] http://www.eclipse.org

Technical Details

An Eclipse plugin is a jar file (which is the Java archive file) with the Java classes in it, its resources and metadata (a manifest file and a plugin.xml). The general information about the plugin (name, author, version, required version of java, etc) is located in the manifest file. There may also be hashes for every executable to be able to verify the integrity of the code, but a separate signature file might also be included. In the plugin.xml, the interconnections with other plugins are defined. It defines extension points and declares which extension points of other plugins it extends. The extension points may declare new XML element types to be able to communicate arbitrary information from the extension plugin to the plugin that defines the extension point. The extension points may come with an API (Application Programming Interface), which the extension plugins have to implement [8].

On startup, Eclipse discovers the plugins (they are located in a special plugin directory) and loads their metadata, thus creating a register of all the plugins and their interconnections. The plugins are not yet loaded until they are activated, but missing interconnections or wrong interconnections can be detected beforehand. The resulting information is then available to every plugin.

The major components of Eclipse are shown in Figure 2.1. The workbench, workspace, help and team are all plugins that define extension points to let other plugins extend them. In this way, there may be extensions that modify Eclipse to be a Java IDE (Integrated Development Environment) or a C IDE or whatever language you want. But it is not required to extend one of the major components: the plugin author can create a plugin that does not extend anything.

Eclipse Plugin Central

The Eclipse Plugin Central (EPIC)[2] is the official community portal to distribute Eclipse plugins. It provides two services: a *plugin directory* and *training and consulting*. The training and consulting part of the portal consists of a list of entities that provide training and consulting services. The plugin directory will be explained below.

In the plugin directory, about 1100 plugins can be found at the time of writing. They can be found by searching or browsing through the directory. Browsing can be done by first selecting a category and then browsing a list of plugins in that category. The list can be ordered by alphabet or by rating. EPIC also provides their visitors with a list of *New and Updated Plugins*, *Top Rated Plugins* and *Most Active Plugins*. Each list shows 15 plugin names with some additional information. For example, the *New and Updated Plugins* are shown with their version and when the new plugin was added and the *Top Rated Plugins* are shown with their ratings.

Every plugin has its own page with general information such as author, name, description, version, license. Users can rate the plugin on a scale from 1 to 10 and

[2]http://www.eclipseplugincentral.com

comment on the plugin. The site also makes it possible to let users vote from outside this website (e.g., from the authors own website). The user feedback (ratings

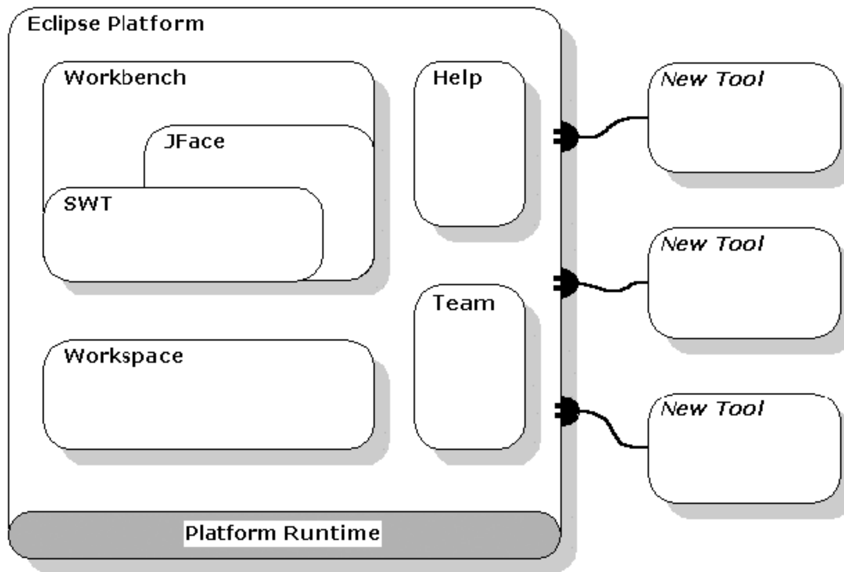

Figure 2.1: The architecture of Eclipse with its major components.

eclipse PLUGIN CENTRAL
LOGIN
HOME | PLUGIN DIRECTORY | TRAINING & CONSULTING | MY EPIC | ADD CONTENT
SEARCH

Plugins (1094)
Application Management (38)
Application Server (13)
Build and Deploy (26)
Code Management (46)
Database (38)
Documentation (17)
Editor (54)
Entertainment (11)
Graphics (8)
IDE (71)
J2EE Development Platform (19)
J2ME (9)
Languages (58)
Modeling (69)
Network (14)
Other (34)
Process (8)
Profiling (9)
Rich Client Applications (66)
SCM (4)
Source Code Analyzer (43)
Systems Development (13)
Team Development (55)
Testing (60)
Tools (147)
UI (49)
UML (32)
Web (40)
Web Services (16)
XML (23)

Training and Consulting (50)

eclipse Summit
Register Now!
November 19th-20th
Ludwigsburg

Eclipse Plugin Central (EPIC) offers the Eclipse community a convenient, information-rich portal that helps software developers find open source and commercial Eclipse plugins, tools, and products that enhance the entire software development life-cycle.

Featured Plugins

IBM Rational Software Modeler
IBM Rational Software Modeler is a Unified Modeling Language (UML) 2.0-based visual modeling and design tool for architects, systems analysts and d ...

eclipse PLUGIN CENTRAL Featured Plugin
Cape Clear Orchestrator
Cape Clear Orchestrator is an intuitive Eclipse-based editor intended to simplify the design, deployment, and management of orchestrated business p ...

New and Updated Plugins

Plugin	
eclox 0.8.0	+20 hours
Remove Tag Tool 1.0.0	+2 days
jQueryUMTP 0.2	+2 days
WolfDBManager RCP 6.1	+2 days
ActiveVOS 6.0	+2 days
tomcat deployer 1.2	+5 days
eclipse4SL - Eclipse Tools for Silverlight 1.0.0.M1	+5 days
YUI WTP 0.2	+5 days
System Variables	+5 days
SvnAdmin 0.9.0	+5 days
HiberObjects Import DB Pro 1.5.0	+6 days
HiberObjects 1.5.0	+6 days
Java Code Generator 1.0.0	+6 days
WindowTester Pro 3.6	+6 days
RCP Developer 3.6	+6 days

New | Updated — More Plugins...

Top Rated (Lifetime | 6 Months | 3 Months)

Plugin	Rating
Regex Util	9.5 (197)
Pulse | Download, manage and maintain Eclipse tools	9.5 (656)
Log4E	9.5 (480)
AnyEdit Tools	9.5 (534)
SWT Win32 Extension	9.4 (121)
eFace free edition - XAML for Java	9.4 (481)
Extended VS Presentation	9.4 (260)
MyEclipse Enterprise Workbench	9.4 (7814)
Spring IDE	9.3 (606)
DBViewer Plugin	9.3 (120)
Mylyn	9.3 (1330)
Hibernate Synchronizer	9.2 (5221)
openArchitectureWare Code Generator Framework	9.2 (502)
Azzurri Clay Database Modeling in Eclipse	9.2 (2230)
JFormDesigner - Swing GUI Designer	9.1 (326)

More Plugins...

Most Active (This Month | 24 Hours)

Plugin	Count
Pulse | Download, manage and maintain Eclipse...	5258
MyEclipse Enterprise Workbench	3665
eUML2 free edition	2532
Subversive - SVN client	2087
eFace free edition - XAML for Java	2017
Pydev - Python Development Environment for Ec...	1924
Jigloo SWT/Swing GUI Builder	1790
UMLet - UML Tool for Fast UML Diagrams	1654
Green UML	1498
Eclipse SQL Explorer	1418
Sysdeo Tomcat Launcher Plugin	1272
Spring IDE	1155
Regex Util	1110
DBViewer Plugin	1009
AmaterasUML	903

More Plugins...

Figure 2.2: Eclipse Plugin Central.

and comments) is a very valuable resource for other users and thus EPIC has measures against spamming and other attacks. Among those measures are attempts to block votes from automated proxies and making sure every IP address may only vote for each plugin once.

To submit your own plugin to the directory, it is required to sign up for an account. When registered, it is necessary to fill in a form with the general information of the plugin. The submissions require approval before they are published. The approval is probably done manually and is an important check to make sure only benevolent plugins are published.

Security

The Eclipse Equinox Security project[3] is a project that is seeking to integrate Java security mechanisms into Eclipse. Java has a very powerful architecture for fine-grained code authorization. A Security Manager checks whether certain code is authorized to execute various code segments or access resources such as network sockets or files. Security Policy files present the access rules for the code. For example, a policy may say that a certain code package is allowed to access a certain file on disk. Using this security architecture, a sandbox can be created, see Section 5.3.2. At the time of writing, policies are not yet used for Eclipse plugins.

Eclipse, however, does provide facilities to digitally sign the plugin package for authentication using standard Java jar signing mechanisms. In this way, the identity of the author of the plugin can be proven. For more information on code signing, see Section 5.3.2. When a plugin is installed, Eclipse checks the digital signatures and will prompt the user when an author is not trusted by the system. The user may decide to trust the author anyway and further install the plugin.

2.2 Google Gadgets

Google Gadgets are, according to Google, miniature objects that can be placed on an iGoogle startpage[4] or on Google Desktop. iGoogle is a start page service of Google where you can add any Google Gadget to personalize it. Google Desktop is a special program (a gadget engine) which runs the gadgets on your desktop instead of on the web. Google Gadgets can be anything, from a countdown gadget to an RSS reader, from a translator to games. They typically take some small amount of space on the user interface (web page or the Google Desktop Canvas). A new development is that of Google Gadgets for social networks using the OpenSocial API[5]. OpenSocial defines a common interface for social networks, being able to retrieve and store information about friends and activities. The OpenSocial API is

[3] http://www.eclipse.org/equinox/incubator/security

[4] Actually, it is also possible to include iGoogle Gadgets on another website than the iGoogle startpage.

[5] http://code.google.com/apis/opensocial

currently already supported by many social networks such as MySpace, Friendster and Hyves. A gadget that is created using the OpenSocial API can run on all the social networks supporting the API.

First we will dive into some technical details, then we will discuss the distribution of Google Gadgets and finally security of the gadgets is discussed.

Technical Details

There is a difference between Google Gadgets and Google Desktop Gadgets. Although they both are coded using XML and Javascript, the XML specification and Javascript API are different. Despite that they are different, some Desktop Gadgets can run on iGoogle and most iGoogle Gadgets can run on Google Desktop. iGoogle Gadgets XML specifies some general information, the content, and the user preferences that are supported in the gadget. The content can either be HTML and Javascript or from another website. To make the gadget dynamic, the Javascript language can be used. The iGoogle Gadget Javascript API adds objects and functions to the Javascript language to make it easier to create gadgets. For example, the API supports registering a callback function to be called when the gadget is loaded or functions to pop up a message. The gadgets XML should be hosted on the web, such that the Gadget server can process it.

The Desktop Gadgets are more extensive: they are typically packed within a .gg package, which is a zip archive with another extension. The archive typically contains a gadget.manifest with metadata about the gadget, a main.xml which specifies how the gadget will appear, string files to support localization and optionally some Javascript files and other resources. It is even possible to provide more functionality to Javascript by creating dynamic link libraries in C, C++ or VB.NET. The main.xml has a specification that supports different GUI widgets, such as a list box or a button. The Javascript API for Desktop Gadgets is more extensive than the iGoogle Gadget API and provides access to information of the host machine and provides event notifications when a the users opens a web browser or visits a certain page, among a lot of other features. An exciting feature is that of GoogleTalk, which provides some functionality to Desktop Gadgets that let them use Google Talk as a medium to communicate with other gadgets of the same sort. Desktop Gadgets are identified by a unique identifier called the Universally Unique Identifier (UUID)[6]. A UUID generator generates a 16-byte identifier which has a great chance of being universally unique.

Distribution of gadgets

Both iGoogle Gadgets and Desktop Gadgets has its own central distribution site. iGoogle can sort the gadgets by popularity, most users and newest. The directory can be further narrowed down by selecting a category or it can be searched by a simple search form. The gadgets are summarized by showing a thumbnail, name,

[6] http://en.wikipedia.org/wiki/UUID

description, author and rating. Every gadget has its own page where users can comment and rate (on a scale from 1 to 5). A nice other feature is the sidebar which tells you which other gadgets you might like. A screenshot of the iGoogle Gadget directory is shown in 2.3. The Desktop Gadgets directory has about the same features but has a tiny difference in layout. It also has a New Gadgets RSS Feed, which is nice way to stay up to date on new gadgets.

To submit your gadget to the iGoogle directory, you have to specify an URL to the XML of your gadget. In this XML, the general information about the gadget should be correct and complete. Google automatically detects updates in the gadget. The Desktop Gadget submission procedure works about the same: the gadget archive should be located somewhere on the web and Google automatically detects updates (whenever the version is increased in the archive). Google says they are approving the gadgets to their directories, thus filtering out malicious gadgets. However, malicious gadgets can still be used whenever the URL to the gadget is found.

Gadget Security

In the beginning, iGoogle Gadgets could run either in an iframe or inlined on the page. When a gadget was inlined, it had full access to the Document Object Model (DOM) of the web page it was placed on. Wrapped in an iframe, the gadget does not know it has a parent page and does not have access to it, which is more secure but limits the functionality. This helps the user protect against malicious gadgets

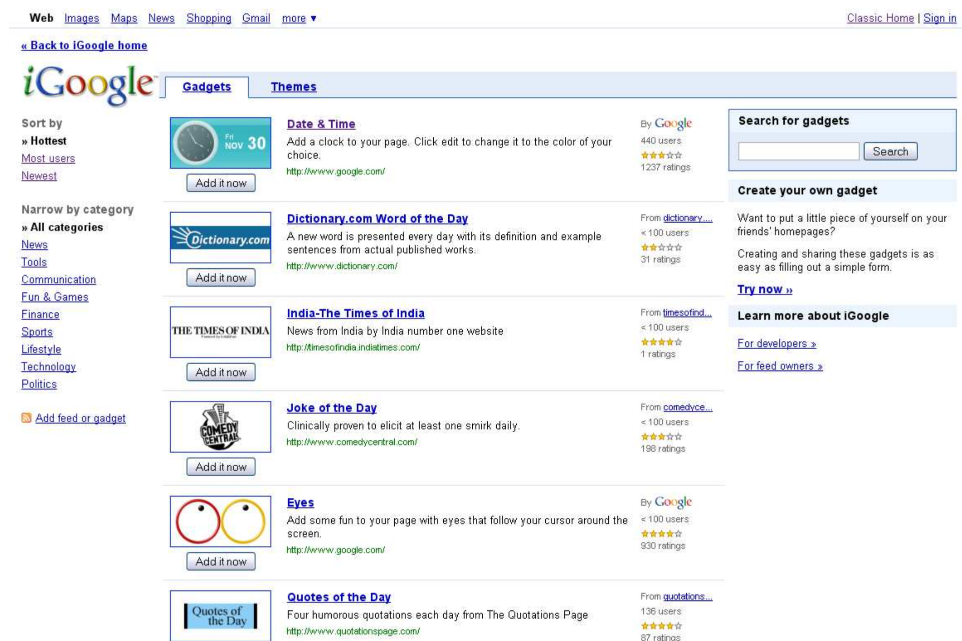

Figure 2.3: iGoogle Gadget directory.

that might want to steal or modify cookies.

Some Cross Site Scripting (XSS) and Cross Site Request Forgery (CSRF) security holes in Google Gadgets have been reported in the past, but Google has fixed them. Cross Site Scripting is a type of security vulnerability where attackers may inject malicious code to websites that are viewed by other users, possibly gathering information about those users, e.g., by bypassing the *same origin policy*. The *same origin policy* is a security measure where scripts from one origin may not access data from other origins. Cross Site Request Forgery exploits the trust a website has in a user. The user is abused by the attacker to perform an action without the user knowing. Websites usually have task URLs, for example http://www.foo.com/buy?articleid=9. When a user is logged in, an attacker may be able to trick the user to go to such a task URL, performing an action the user did not intent.

Desktop Gadgets even pose a more serious security risk, since it allows users to complement the Javascript code with C, C++ or VB.NET, unrestricting the possibilities an attacker has. The API also makes it possible to send XMLHttpRequests (asynchronous HTTP data requests) from your desktop, possibly sending confidential information from the users to a site. Since there are currently no restraints on Desktop Gadgets, users should be very cautious when installing them. Google Desktop gives a warning when installing, but that is about the only security measure it has.

2.3 Facebook Apps

Facebook is a growing social network where people get connected, stay in touch with friends or create new friends and share photos, links, videos. It provides a framework for developers to create their own small applications (Facebook Apps) that integrate into Facebook and have access to the social features of Facebook. Facebook Apps are stored and executed on a third party server. The output of the Facebook App is then post-processed by the Facebook server, creating the output for the client. The intermediate output of the Facebook App is typically in Facebook Markup Language (FBML), which is a subset of HTML and extended with Facebook specific features, and in a modified Javascript language called FBJS. After processing by the Facebook server, it is turned into HTML and Javascript. An illustration is provided in Figure 2.4.

Technical Details

The Facebook API consists of HTTP request calls and thus Facebook Apps can be created in any language that supports HTTP requests. The applications are hosted on a third party server (everyone has to host their own application) supporting a server-side language. An API wrapper for the specified language is used to wrap the HTTP requests in functions. Instead of creating a HTTP request and parsing the

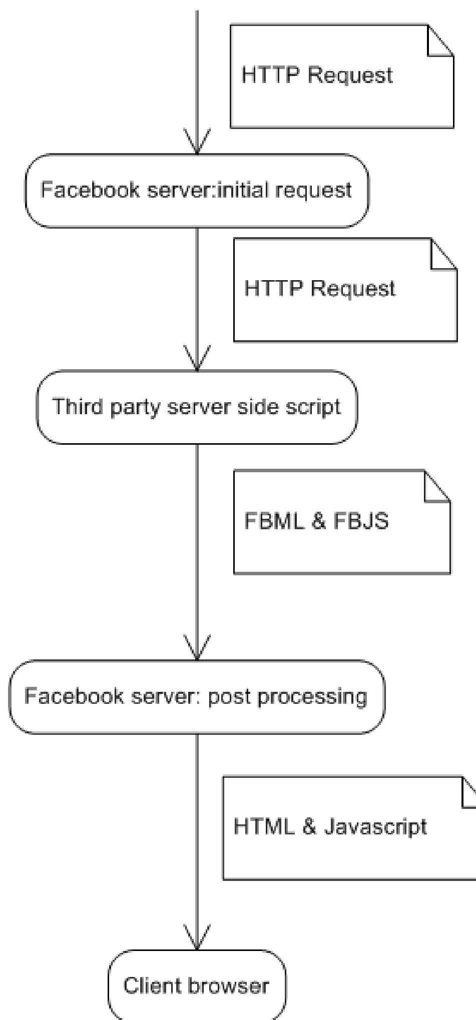

Figure 2.4: The path of a Facebook App request.

response yourself, the wrapper does this for you when you call one of its functions. The function then makes the request and returns the output in an appropriate format for the language used. The application ultimately outputs FBML and FBJS code, which is post-processed by Facebook to generate output that the browser understands (HTML and Javascript). The Facebook API also supports a query language called FQL, which is similar to SQL. An FQL call can be made from the application by issueing a HTTP request with the FQL query and the Facebook server will respond with the result, typically in XML or JSON (Javascript Object Notation), which is a lightweight data-interchange format mostly used in Javascript.

A developer creating a Facebook App should first create the application in Facebook, notifying you are going to create an application and specify some general information. An API key and secret key and application ID are generated by Face-

book for your application. The API key and secret key are used in the application to authenticate the application to Facebook when making API calls. The application can be created in a language you prefer and you should make sure it is hosted on the web. When the developing is done, you can submit the application to the Facebook directory, so that others can find and use your application.

Distribution of Facebook Apps

Facebook has its directory of Facebook Apps, which is the only place to get the Facebook Apps because Facebook is the only platform they run on. The directory shows the applications initially ordered by popularity, but it is also possible to sort them on "Most Active Users" or to show the newest applications. As is possible in other widget directories, the applications can be browsed by category and there are about 22 categories. The applications are shown with a thumbnail, title, author and a small description as well as the monthly active user base for that application and a link to the reviews of that application. Currently, there are about 40,000 Facebook Apps in the directory listing. Each application also has its own page with general information, reviews (which are basically a comment and a rating), a discussion board and a button to add this application to your profile. Only Facebook users can submit reviews (they are the only ones being able to use the applications) thus there are no anonymous reviews, but the question is whether the comments are better as a result. It is also interesting to see that the applications do not have to be useful but can be simple applications, like being able to rate other users for their *hotness* by looking at their pictures or just digitally hug someone.

When submitting an application (which is simply a submit button), the application is first approved by Facebook, before it is added to the directory.

Security

In a social network like Facebook, privacy is an important aspect of security. For example, when Facebook added the *News Feed* feature, which informs all your friends about the actions you take, many Facebook users were quite upset[7]. Facebook replied that all this information was already publicly accessible, but the crowd still did not want the feature. People like to have control over their privacy and thus eventually Facebook made the News Feed optional. Now, with Facebook Apps, anyone can develop applications that use this privacy sensitive data so it is necessary to have additional security measures.

There are two security issues involved with Facebook Apps: benevolent applications that have vulnerabilities, and malicious applications. The former issue might be because applications are not developed with security in mind, or because Facebook itself has security glitches. There already have been reports with proof-of-concepts of security holes in Facebook[8]http://www.theregister.co.uk/2008/05/23/facebook_xss_flaw/ and http://www.theregister.co.uk/2008/05/23/facebook_xss_flaw/, and Facebook has

[7]http://www.danah.org/papers/FacebookAndPrivacy.html

fixed them eventually. More serious are the security flaws in the third party applications. For example, in an article in the 2600 Hackers Quarterly magazine [30] some flaws in running applications are shown. Most of the flaws involve modifying the user id or owner id in a task URL, assuming someone else's id to perform malicious actions. It shows that most Facebook applications developed by third parties do not have proper authentication, which can be exploited easily for phishing attacks or spam.

An example of the latter issue, malicious applications, is the Secret Crush[9], where you have to invite twenty friends to use the application before it shows your secret crush. However, instead of showing your secret crush it points the user to a site which tries to install adware on your computer. While Facebook deleted the application from the directory, it could not prevent users from installing it.

Malicious Facebook applications are also researched in [1], where the authors developed a malicious Facebook application called FaceBot to experiment with social networks. Their application, *Photo of the Day*, displays a photo from National Geographic and also adds a few hidden iframes pointed to the victims host with the goal of a Distributed Denial of Service (DDoS) attack. With a least effort attack they made it to a peak of 300 requests per hour, showing that malicious applications are easily created and deployed in a social network such as Facebook. When they would have made more efforts to advertise their application or amplify the attacks, it could do serious damage. Considering that peer-to-peer networks have the same social aspect, this security issue is as important in peer-to-peer networks which allow third party code to run.

Facebook users seem to be ignorant about the possible security issues when installing third party applications. Although you have to grant the application explicit access to your information, still malicious applications are able to spread virally. Also, we find it weird that Facebook, while having a central directory where submitted applications have to be approved before they are added to the directory, still can not manage to reject malicious applications or applications that are not secure. Finally, we could not really find a lot of information about securing your application on the Facebook developer pages. It would be really useful to make developers aware of the security issues and how to develop secure Facebook applications.

2.4 Firefox Extensions

Mozilla Firefox is currently the second most used browser in the world according to Net Applications[10]. A core feature is the one of extensions, which lets the user customize the browser to their likings by adding extensions you want. They can add more security, toolbars or even add complete applications inside the browser, e.g., an instant messenger. While Google Desktop Gadgets use similar technology, also defining the user interface in XML and interaction in Javascript, Firefox

[9]http://www.sophos.com/pressoffice/news/articles/2008/01/facebook-adware.html

[10]http://marketshare.hitslink.com/report.aspx?qprid=0

extensions can be more integrated in the browser. Google Gadgets have a small amount of space reserved for their user interface, while Firefox extensions can add menu items, toolbars, new buttons, etc.

Technical Details

Mozilla has an extensive amount of documentation on the extension framework which can be found at `http://developer.mozilla.org`. We will first explain the structure of an extension. A Firefox extension consists primarily of *locale* information, *skins* and *content*. Locale information are strings used in the extention in specific languages to support internationalization, skins are Cascading Style Sheets (CSS) and images which make up the appearance of the extension and the content consists of XUL (XML User Interface Language) and Javascript files. In Firefox, the user interface is defined in XUL, and you use XUL to create new windows or dialogs, add GUI widgets to them (buttons, toolbars, textfields) or create overlays to add or modify existing user interface elements. For example, using an overlay, you are able to add content to the existing Firefox statusbar or toolbar. It is also possible to extend windows that are defined by other Firefox extensions. Javascript is used to handle the actions when someone interacts with a user interface element. It is also possible (just like Google Desktop Gadgets) to write components in other languages such as C++ or Python. Using XPCOM, which is a Cross Platform Component Object Model used by Mozilla applications, it is possible to create such components which can be interfaced from Javascript. However, the existing XPCOM API already includes numerous objects and functions to be used by the extension and thus only if you need more performance or more functionality you could create an XPCOM object in another language.

Along with the files discussed above, a Firefox extension includes an install manifest file written in RDF, which describes the application (name, author, version, compatibility with which Firefox version, etc) and a Chrome manifest file. Chrome is the set of user interface elements that are outside of the content area (in Firefox the content area is the area that shows the webpages). The Chrome manifest file explains to Firefox what the content directory is, which skins are added, which locales are supported and which browser overlays it uses, including where the files are found.

Distribution

Mozillas distribution site has about the same features as the other sites discussed previously, including browse by category, search, comment and rate extensions. More prevalent are the ”We Recommend” sections throughout the site, which highlight certain extensions. Unlike the other distribution sites, Firefox requires users to log in before you can comment and rate an extension. The extension reviews seem pretty decent and popular extensions are reviewed extensively.

When you select a category to browse, first some highly popular extensions are

shown, but from there you can view all extensions in a list ordered by name, date, number of downloads (popularity) or rating. An interesting option is to show experimental extensions, which are still in a production or testing phase. Advanced users may download them for testing purposes when they log in. It actually takes some effort from the visitor to be able to view low quality and unpopular extensions, because the pages always start with highlights of popular extensions. The links to the list of widgets are easily sorted on popularity or on rating. This is illustrated by a screenshot of the distribution site in Figure 2.5.

When submitting an extension, you should first create an account or log in, before you can use the submit wizard. The wizard consists of a few steps. First, the extension should be packed and uploaded. Then some general information about the extension and version information is entered. Lastly, it is also possible to include translations for localizations. After submission, you can manage the extension from the site. The extensions are also approved before added and Mozilla seems to do quite a good job there, as most extensions are rated pretty high.

Figure 2.5: The distribution site for Firefox extensions. It is difficult to find low quality extensions.

Security

A number of warnings and reports about malicious Firefox extensions can be found on the web, for example FFsniFF[11] and FormSpy[12] are both malicious extensions that try to retrieve passwords, credit card numbers and other private information. These malicious extensions are possible because the API is very powerful, giving full access to the file system and network, but also the possibility to write native code giving an extension limitless possibilities [3]. The same paper shows some stealth possibilities, allowing extensions to hide its presence by various techniques. For example, there is a *hidden* option, which prevents the Add-On Manager from displaying the extension. An extension could also create an overlay for the Add-On Manager to modify it to hide the malicious extension. Because there are no restrictions (there are no security policies for extensions), extensions can basically do whatever they want. Below we will discuss the security measures Firefox does have to fight malicious extensions. First, Firefox supports the signing of extensions. Whenever you want to install an extension, it will pop up a warning window with the question if you really want to install the extension. Second, Firefox blocks requests to install extensions from other sites than the official site, but you can choose to install the extension anyway. Other malware can also install extensions without asking permission, but that means the computer has already been infiltrated. When other software has installed an extension, Firefox 2 does not even alert the user, but Firefox 3 does. From these measures, it is clear that Mozilla greatly relies on their central distribution site and the moderation of submitted extensions. They seem to do a pretty good job at it, also because of the structure of the distribution site; very popular and highly rated extensions are highlighted and less popular extensions are harder to get.

[11] http://blog.trendmicro.com/malicious-firefox-extensions/

[12] http://us.mcafee.com/virusInfo/default.asp?id=description&virus_k=140256

Chapter 3

Zero-server widget repository

In the previous chapter we have seen examples of centralized widget repositories and their features. In this chapter we will discuss how to port such a centralized repository to a peer-to-peer architecture. In a peer-to-peer architecture, all peers help to store the widgets. This has advantages like scalability and reliability as there is no central point of failure. Key requirements of a repository include to find widgets, package management and moderation of widgets. In centralized repositories, these requirements are not that hard to implement, but in a distributed environment they are more difficult to accomplish.

First, a directory service to be able to find the widgets is discussed in Section 3.1, where a small overview is given and a comparison between two promising candidates is made. Second, moderation of widgets is discussed in Section 3.2. Third, we will discuss the package management in Section 3.3.

3.1 Directory service for widgets

A directory service is used to find widgets in a distributed environment. All peers work together to deliver content to other peers, creating an environment which is scalable and has high availability. A directory service for widgets is different than other peer-to-peer systems currently in use. Peer-to-peer systems are commonly used for file sharing, where peers share gigabytes of movies, music and applications. Another, but less common use for peer-to-peer systems is to share computation, where peers receive data to analyse.

A directory service for widgets is different than file sharing peer-to-peer because widgets are mostly small packages consisting of source files or binaries, some metadata files and possibly small resources such as thumbnails, pictures, etc. Also, there is a limited amount of widgets in a repository. Every year, thousands of new songs and movies are released, but widgets are created by volunteers that first have to get comfortable with developing in a new runtime environment. To support our statement, we summarized the size of some popular widget repositories

in Table 3.1. Note that we cannot estimate the total size of Facebook Application repository because the applications are executed server-side. We however wanted to add the Facebook apps size, because it shows that widgets can be a huge success in social networks, increasing the size of the repository dramatically.

Now we consider the *metadata*, which consists of general information on the widget (name, author, description, version information, etc.) and locations where to find it (IP addresses). We estimate the size of this metadata about 2 KB per widget. Current metadata files for Google Gadgets or Firefox Extensions are less than 2 KB, but they are in XML which is readable but creates large files. Consider a more compact format and/or compression and the size will drop considerably. Now we can estimate the metadata size. With sizes ranging from 1200 to 5800 widgets, the total amount of metadata will range from 2 to 12 MB. Facebook, however, has a much larger sized repository of 40,000 widgets, which would create a total of 80 MB metadata. Still, the metadata is quite small.

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 3.1: Estimated repository size of various widget platforms.

We obtained the results of Table 3.1 as follows. The repository sizes are estimated by multiplying the number of pages with the number of widgets per page. The estimates for average size per widget are obtained by a random sampling of about 20 widgets per repository. The estimated total size is deduced from the previous measurements.

We will continue this section with a review of some peer-to-peer object location systems in Section 3.1.1. After this review, we will make a comparison in Section 3.1.2.

3.1.1 Review of peer-to-peer directory service systems

In the literature, a rough classification has been made of current peer-to-peer architectures. The first classification is whether it is partially decentralized or fully decentralized. In a partially decentralized architecture, there still is a central index which stores the location of the objects. An example of such a system is Napster.

We will only consider fully decentralized peer-to-peer systems here, i.e., the index is maintained by the peers without a central index or there is no index at all.

Another way to classify peer-to-peer systems is by network structure. In unstructured networks, the placement of content is completely unrelated to the topology. These networks need to locate objects by querying the network (e.g., by flooding or random walks). An example is Gnutella. We will not consider unstructured networks, because first, flooding is not scalable and second, both flooding and random walks may fail to find objects because they are curtailed.

Structured networks control object placement in such way that the nodes can find the appropriate objects easily. They use a mapping of object 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 further 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 (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, Pastry and CAN.

There are a number of problems with these DHTs. First, only exact matches are possible. Research on keyword searching is still ongoing. Second, latencies in these DHTs can still be quite large. These problems are discussed below.

Keyword searching. The classic approach to the keyword searching challenge, used in centralized search engines, is to use inverted indexes. Per keyword a list is maintained of the documents related to that keyword. Using this technique in a peer-to-peer environment, the keyword lists are stored in a distributed manner, for example using the same DHT as was used to store and locate the documents. To process a keyword query, the lists of documents per keyword is retrieved. Operations can then be applied to the lists of documents using intersection for the boolean ”AND” and union for the boolean operator ”OR” [29]. PeerSearch [31] is build on top of CAN and creates a semantic overlay, where documents that are semantically close to eachother are also close in CAN address space. Normally, CAN generates random document IDs which have no meaning at all. Using the semantic vector (a vector which indicates which keywords describe the document best) to create the IDs in the address space, the semantic overlay is built. The semantic overlay can be exploited to support keyword search in an efficient way.

Latency. Because of the structure of DHTs, the search is deterministic (i.e., if it can be found it will be found) and is asymptotically of order O(log n). However, since there is no relation between the location of the object and the requester, the request can still take a lot of time. Tapestry [16] tries to decrease the latency by replication. Documents and peers have a global unique identifier (GUID) and queries have a destination GUID which ultimately resolves to a peer. Peers can publish any document with a GUID by sending a publish message to the root set of the GUID, leaving a pointer to itself at every hop and at every root node. A query

for a particular GUID is also directed towards the root set of the GUID, but when a pointer is found, it is followed. Leaving the pointers will direct a query to the nearest replica (fewest hops) when one exists, and when no pointer is found it will ultimately go all the way to the root set.

The DHTs described above manage a distributed index: no one has a whole view of the index. Because of the structure, the objects can still be located. Another way to locate objects is to let everyone maintain a local copy of the global index, instead of partitioning it as in structured networks. Using epidemic protocols (gossiping), the index can be maintained. Because such an index could grow very fast, the index should be summarized in a compact way.

PlanetP [7] is an example of the infrastructure described, which uses bloom filters to summarize the indices. A Bloom filter is an array of bits which represents a set of strings. Strings are hashed using multiple hashing functions, which hash the string to a bit position and the bits in those positions are set. By checking the bit positions of a string, it can be tested whether the string is part of the set. While providing a very compact summary of a set, the Bloom filter can give false positives when bit positions of a string are set because of other strings. PlanetP uses gossiping to replicate a list of peers, their IP-addresses and their Bloom filter.

Newscast [17] is a general news dissemination framework which uses gossiping. They use the term news to emphasize the concept that fresher contributions are more important than older contributions. A Newscast application consists of a collective of agents and a news agency. The news agency consists of multiple news correspondents which constantly retrieve fresh newsitems from the agents and disseminate the news to other news correspondents. They also update the agents with other news they retrieved from other correspondents. The algorithm is similar to other gossiping algorithms, however the correspondents do not know the complete member list but only a fraction of it. In fact, the member list is disseminated also. Further, peers do not now the complete list of newsitems but only a small random fraction of it. This is different than in PlanetP, as they try to create a local copy on each peer and summarize it to keep it small. Using the Newscast framework, an application can be made where every peer disseminates the files they serve as newsitems. Other peers ultimately retrieve this information and can contact the peer. This is called the push model and is also used in PlanetP. Newscast can however also be used to query the network, by disseminating newsitems which include requests. The peers that have items that conform to the request may contact the requester. This is the pull model and is not often used in gossiping frameworks. It is also possible to combine these two models to create the push-pull model.

Using a gossiping protocol to manage the resource index has a number of advantages and disadvantages compared to DHTs. In a gossiping system the index structure will be maintained when a few nodes disconnect, because every node has a local copy of the index. In DHTs this index is partitioned and thus when some important nodes disconnect (which had data that other nodes did not have), part of the index could get lost. This advantage is also a disadvantage for gossiping sys-

tems, because this data is diffuse everywhere, while it might only be necessary for a subset of the users. Another advantage is the possibility to browse the directory. Other disadvantages are scalability and the time it takes for new data to spread. Scalability is a greater issue when using gossiping, because maintaining the index takes a lot of bandwidth when there are a lot of nodes. This can be alleviated by having only to gossip the changes instead of the whole index. Cuenca-Acuna et al. [7] studied the scalability of PlanetP and reports that is scales very well, maintaining a contant recall and precision for communities of up to a thousand peers. They however did not study the scalability beyond that point. Thousand peers is in our opinion not an extremely scalable solution such as the DHTs discussed previously.

3.1.2 Comparison of DHT and gossip based systems

Which directory service would be the best for a widget repository? We will compare DHTs and gossip based systems based on a number of topics , because we think those are the most promising.

To support keyword searching in a DHT, extra distributed datastructures need to be implemented and maintained by the peers. This costs thus more effort from the developers and the users bandwidth and space. In gossip based systems, one can easily do a local search because all the metadata is already available. Browsing in DHTs is not researched yet, but this would require a peer to receive metadata for every object and thus would require a lot of bandwidth in a short moment, while gossip based systems disseminate the data over long time intervals. Browsing is inherently possible in gossip-based systems, because one could do local browsing. High churn also effects the DHT, but it does not effect gossip based systems. The reason is that DHTs distribute their data and in gossip based systems, data is diffuse everywhere except for when something is just published and it still has to travel to other peers. On the other hand, DHTs are very scalable, supporting thousands of peers collaborating to maintain the structure, while gossip based systems are less scalable, as is studied by Cuenca-Acuna et al. [7]. Another disadvantage in gossip based systems is the time it takes for just published data to be available. In Table 3.2, the comparison is summarized.

Based on the estimated size of the repository, we think gossip based systems might be quite a good solution for a widget directory service, as it provides search and browsing features. All centralized repositories include browsing per category as a core feature and we think decentralized repositories should support this as well. Further, in widget environments, some widgets are very popular but most of them are not, which kind of resembles the gossip paradigm. Popular widgets are gossiped more about than less popular widgets thus amplifying the popularity. On the other hand, while most widgets are not popular, their metadata are still diffuse on all peers.

All in all, we think that gossip based systems are quite a good solution for a widget repository in relatively small peer-to-peer environments. Large peer-to-peer environments might suffer the consequences of the scalability limitations of

Topic	DHT	Gossip
Keyword searching	hard	easy
Browsing	hard	easy
High churn affects data availability?	yes	no
Scalability	very scalable	quite scalable
Published data is available right away?	yes	no

Table 3.2: Comparison between DHT and gossip based systems

gossiping: data takes too long to spread or every peer has to store too much data. Of course, a hybrid between something similar to a DHT and to a gossip based system is also possible. This option is supported by PlanetP and could create a possible environment which supports the advantages of gossiping and of DHTs. For example, using a DHT for support might make the system a lot more scalable and it also increases the availability of just published widgets.

3.2 Moderation in a distributed environment

Due to the ubiquity of spam and low quality content, most online sites where users are allowed to provide input (e.g., social networks, message boards, torrent sites) are moderated. Most prevalent are the moderators that manually check whether some input is spam or not, possibly allowing users to report input as spam to help the moderators. But there are also examples of self-regulating moderation schemes. For example, Nabble[1] is an online message board system that uses a scheme where users may rate comments as spam. When the rating drops below some value, the comment is no longer displayed. In the centralized distribution sites of widgets discussed in Chapter 2, the injection of new widgets or comments is moderated by a few pre-trusted moderators. They are mostly the founders or creators of the system. A second layer of moderation comes from the users of the widgets. They can comment on the widgets and rate them. This information can be very useful to other users who have not tried the widget yet. A lot of research has been done on peer-to-peer rating or reputation systems where peers vote for each others services, because this supports the peer-to-peer environment to be self-regulated without a central authority. The first line of defense of manual moderation is now gone, which was very effective but not very scalable because all the moderation tasks fell to a small group of trusted moderators.

We will now first discuss distributed rating systems and then show a peer-to-peer moderation technique without a rating system.

[1] http://www.nabble.com/help/Answer.jtp?id=28

3.2.1 Distributed rating systems

Rating systems are currently deployed all over the web, for example every widget repository discussed in Chapter 2 has one. Most of these rating systems consist of everyone being able to rate a widget and the site then aggregates the ratings to come up with an average. Such rating systems have been deployed in peer-to-peer file sharing systems also. For example, KaZaa uses a rating system, where users can rate the files they download either as *excellent*, *average*, *poor* or *delete file*. Due to low participation and a high amount of false ratings (i.e. ratings which rate the file excellent when in fact it is poor, or the other way around), the rating system does not work as well as we would like [22].

We will first explain possible rating system designs and discuss some rating system proposals. Most of the rating systems described below use ratings to rate peers instead of the objects they provide. The goals of these systems are to weed out malicious peers, while rating objects provides a means to weed out bad or malicious objects. The architecture of the systems is however still appropriate for rating objects. Which rating system to use for widgets is hard to say. Rating creators of widgets is more effective against malicious authors, but rating widgets can be useful to show the differences in quality and usefulness.

Rating design

There are several possible rating strategies possible. Dutta et al. [10] discusses several options, which we will explain below.

First it is possible to use positive or negative voting, or both. Positive voting means users may either give credit for the service or not. The accumulated value of credits describes the services a user has provided the community. With only positive voting, users are discouraged to do whitewashing attacks, because they would start at zero again. Negative votes can be used to punish users who provide bad services. It can be used to isolate misbehaving users, but negative voting alone is vulnerable for whitewashing attacks. Second, it is possible to use a raw value for rating or a rating with multiple levels. Third, it is possible to consider the whole history of the user to build its rating, or to use only the more recent ratings. When considering the whole history, a user which has built a solid reputation has less incentives to keep providing quality services. But when only the recent history is used, the user has continuing incentives [34]. It is also necessary to consider the context of ratings [33]. For example, consider a peer-to-peer file sharing system which also supports widgets and where each peer can build a reputation by providing high-quality files and by creating benevolent and useful widgets. A peer could now build up a reputation by providing quality files, but abuse this reputation to create and distribute malicious widgets. Now, because his reputation is good, users might also have more trust in the widgets he created.

Managing and distributing ratings

Tian et al. [32] explains two different rating systems in a peer-to-peer system: the unstructured self-managing rating system (UMR) and the structured supervising rating system (SSR). In a UMR, each peer keeps its ratings of other peers locally. When a peer wants to know the rating for a particular peer A, it sends a request to the network, either flooding or using a random walk. Peers who have a rating on peer A can then reply. The SSR design needs a structured overlay, such as discussed in Section 3.1.1. When a peer issues a rating on another peer, it is stored on another peer which is called its supervising peer. To retrieve ratings for a particular peer, it is enough to query the supervising peer. Examples of SSRs are Eigentrust [18] and PeerTrust [34]. Both propose two security measures for peers providing false ratings. First, encryption can be used to store and distribute the ratings. Second, multiple supervising peers can store the same rating. Majority voting is then used to come up with the most likely value. Another way to manage and distributed ratings without using an overlay, is by using gossiping. GossipTrust [35] is such a system, which uses several aggregation cycles consisting of exchanging rating information with random peers to eventually come to consensus on the global trust value.

Aggregation of ratings

We will now discuss the aggregation algorithm basics of Eigentrust [18] and GossipTrust [35]. For more details we refer to the corresponding papers. In **Eigentrust**, the global reputation for a peer i is given by the local ratings for i by other peers, weighted by the global reputation of the other peers. A rating value is given by the number of satisfying transactions minus the number of unsatisfying transactions. First, a peer normalizes its issued ratings.The symbol c_{ij} is the normalized local rating peer i gave j. Then, peer i asks its acquaintances for their trust in peer j, weighting it with the trust it has for them:

$$t_{ik} = \sum_j c_{ij} c_{jk},$$

where t_{ik} is the trust peer i has in peer k, when asking its friends. This can be written in matrix notation with C the matrix of $[c_{ij}]$ and $\vec{t_i}$ the vector of all the values t_{ij} as $\vec{t_i} = C^T \vec{c_i}$. Asking his friend's friends and their friends and so forth, eventually the peer can have a whole view of the network. In matrix notation, this is $\vec{t_i} = \left(C^T\right)^n \vec{c_i}$ for sufficiently large n.

In the distributed version, each peer approximates its own global trust value t_i and this approximation is done iteratively. Each peer first queries all peers which have used its services to retrieve their global trust value at iteration 0. Then it calculates its own global trust value for the next iteration using their global trust value and the trust they have in him. This is displayed in the following simplified formula. The original formula also contains measures to resist collusions which are left out here.

$$t_i^{(k+1)} = \sum_j c_{ji}^{(k)} t_j^{(k)}$$

GossipTrust tries to approximate the outcome of the above formula, but gossiping is used to reach consensus about the global scores. We will now discuss how gossiping is used to reach consensus about the global score of one node. This process can be extended to reach consensus about all global scores.

Associated with a node i is a gossip pair $\{x_i(k), w_i(k)\}$. In this pair, the symbol $x_i(k)$ is the gossiped global score of node i at step k and $w_i(k)$ is the gossip weight applied by node i at step k. At each gossip step, the node sends half of its gossip pair $\{0.5x_i(k), 0.5w_i(k)\}$ to itself and to another random node. It also receives the halved gossip pairs of other peers and sums them up to calculate the gossip pair at step $k+1$, so we get

$$x_i(k+1) = \sum_j x_j(k),$$

$$w_i(k+1) = \sum_j w_j(k).$$

At the final step g, when consensus is reached, the global score of the particular node can be calculated for node i by $x_i(g)/w_i(g)$.

Attacks on rating systems

Tien et al. [32] distinguishes several attacks which a rating system should be able to resist. First, there is the whitewashing attack where a peer, that has a bad reputation takes a new identity to start over with a neutral reputation. Whitewashing attacks can however not be used to obtain a good reputation. Second, there are the false rating attacks, where peers issue false ratings to subvert the rating system, rendering it useless. They can either try to give themselves or their colluding partners a good reputation, or give honest users bad ratings. The former is called *ballot stuffing* and the latter *bad mouthing*. False rating attacks can either be performed by a Sybil [9] peer or by colluding peers. A Sybil peer is a peer that generates multiple identities to issue false ratings to one of the identities, while using that identity to act maliciously. Douceur et al. [9] states that it is practically impossible in a distributed environment to enforce a unique distinct identity for each entity, without a central authority.

To discourage whitewashing and creating multiple identities, some cost should be introduced to newly starting peers. For example, VoteCast [28] lets newly introduced peers only vote when they become experienced.

False rating attacks can be alleviated by using a personalized trust system, where the ratings which are issued by others are weighted by the trust the peer has in them. It is also possible to filter out some ratings, based on the rating given to them.

3.2.2 Moderation without a rating system

It is possible to create a system with moderators, just like in centralized repositories. The moderators have the power to filter out low quality or malicious widgets from the repository. It can easily be seen, that this approach is not scalable. ModerationCast [14] takes another approach, which creates a self-regulating environment, where everyone can become a moderator but popular moderators get more power than unpopular moderators. We will now describe ModerationCast.

ModerationCast is a metadata dissemination service for peer-to-peer video files. Peers may add metadata to video files, and they become moderators. They gossip their metadata to other peers. Other peers may either decide that the metadata is wrong or bad or decide that they like the metadata. When they like the metadata, the moderator is added to a local list and all moderations that the moderator sends will also be forwarded by this peer. In this way, moderators which add good metadata spread their metadata faster than malicious peers (when the forwarders are honest users). It also uses digital signatures to prove the identity of the moderator, when forwarding the metadata.

When wrong metadata is found, the peer may either block the moderator or update the wrong metadata. In their simulation, it is shown that with the help of forwarders, the metadata of good moderators is disseminated a lot faster than polluted metadata. There is also research going on to aggregate the local ratings of moderators and use this to build up reputations. In this way, it would closely resemble a rating system.

Of course, the principle in ModerationCast is not limited to metadata and could be used for dissemination of other objects. When it is used for widgets, malicious widgets would have more trouble propagating through the network because of less forwarders. The problem however is that metadata can be shown easily to the user without doing any harm, but widgets have to be tried out or closely examined before a user might have an opinion. This makes it a less attractive solution for dissemination of widgets.

3.3 Package management

Widgets are basically small software packages, just like the packages in Linux. Thus they also need to be installed, upgraded, configured and removed. Package management systems help to automate this process. Package managers typically verify the packages using checksums, verify the signatures to authenticate the author of the package, upgrade packages to the latest version and manage dependencies between packages. The first two functionalities are out of the scope of this chapter. Also, when using widgets which can neither extend each other nor use each others functionality, there are no dependencies to manage. Otherwise, these dependencies should be stored in the register and checked whenever widgets are updated, installed or removed. Typically, a dependency like *widget x depends on version 1.x of widget y* is formulated in the metadata of widget x and stored in the

register after installing widget x (if all dependencies were correct). The package manager could automatically find the appropriate version of widget y when it is not already installed.

Because the widget repository is now distributed, we have to provide storage for multiple versions. The package management system should be able to find the latest version, so that it can upgrade the widgets. We will now discuss several techniques for storage of multiple versions of a widget.

3.3.1 Storing and finding multiple versions

A version management system for widgets will need a way to group different versions of the same widget. In distributed systems, this is usually done by a unique identifier for a widget, such as the UUID used in various widget systems discussed in Chapter 2. Using a unique identifier, it is possible to find different versions of the same widget in the system. In a gossip based system, this can be done easily. A DHT however, would need a different mapping from a key to multiple values. The key would then be the unique identifier and the values the different versions. Project Cassandra[2], an open source peer-to-peer storage system for managing structured data, has an expanded data model which can be used for version management. The Cassandra data model is derived from Google Bigtable [6], with some additional features. Bigtable is a sparse multidimensional sorted map, where rows are identified by an arbitrary string. Each row may have multiple columns. The columns are grouped by Column Families. A column key is named using the syntax: *family:columnname*. Each row may have different column families, making the rows dynamic. Further, each cell (a data entry under *family:columnname* for a particular row) may have different versions, indexed by timestamps. In Cassandra, a column family can contain either columns or supercolumns, but the number of columns a family can contain is very large and are created dynamically. Supercolumns are constructs that have a name and can contain an infinite amount of columns. Further, each column may be indexed by either name or timestamp. How Cassandra provides high availability, consistency and persistence is out of the scope of this report. Using a data model like the one of Project Cassandra, it is possible to include versioning for widget files, however a simpler data model which supports multiple values for a key would suffice too.

Another question is how to find updates for the widgets that are installed. For systems which are similar to a DHT, the pull model would obviously be appropriate. Peers would probe the system with regular intervals to check whether a new version is available. Gossip based systems would not need to probe the system as the information is ultimately pushed to the peer. When such a message arrives at the peer, the system can simply alert the user. Newscast would be highly appropriate for the job and it could even use push and pull as described in Section 3.1.1.

[2]http://code.google.com/p/the-cassandra-project/

Chapter 4

Widget runtime environment

The runtime environment for widgets is of great importance because it defines what widgets are able to do or what widgets can not do and how they can do it. We have seen several runtime environments in Chapter 2, using different languages. In this chapter we will discuss the different possible runtime environments, their advantages and disadvantages. In particular, we will discuss how every runtime environment is able to restrict widgets such that they run in a sandbox. For more information on the sandbox model, see Section 5.3.2. Further, we will discuss what the difference is between a *peer-to-peer* widget runtime environment and other widget runtime environments.

First, we will compare several different runtime environments in Section 4.1. Second, we will talk about restricting the code in Section 4.2. Third, and finally, creating a peer-to-peer API is discussed in Section 4.3.

4.1 Runtime environment comparison

In this section, different runtime environments are evaluated. First we will discuss interpreting, then compiling and finally other possibilities are layed out.

4.1.1 Running code with an interpreter

When using an interpreter to run code, instead of compiling the source to binary files and then running, the source is taken as the input for the interpreter. The interpreter then parses the source, creates an internal representation, performs some checks and starts running it, instead of generating machine code (which is the next step in compiling). Scripting languages such as Javascript, Lua or Python are commonly interpreted. Interpreting has some disadvantages which we will explain now. First, because the source must be parsed, type checked and run, interpreting is usually quite slow in comparison with compilation. Second, interpreting uses

more memory because apart from the code that is interpreted, the interpreter is also in memory including the necessary data structures for the program.

There are also advantages when interpreting, namely interpreted programs are typically smaller, tend to be more portable (assuming machine-independent representation) and they have access to runtime information [2].

When interpreting widget code, there should be an interpreter delivered as part of the widget runtime environment. While this makes the widget engine larger, it is considerably easier to write an interpreter than a compiler [13].

4.1.2 Compiling the code

When compiling the code to run, the code will run faster than when using an interpreter. However, distributing binaries might not be a very smart in a peer-to-peer environment, because it will decrease collaboration, transparency and portability. When distributing source files, all peers can modify other widgets, which creates a collaboration environment, as seen by other open source projects. With transparency, we mean that distribution of source files generates the possibility that the code is checked by other developers, which can be seen as some kind of extra security measure which is lost when distributing binaries. Portability is decreased because in machine code is usually machine specific. This can however be solved by using intermediate bytecode, as Java does.

It is also possible to distribute the source files, and let the runtime environment first compile the code to machine code (or an intermediate bytecode, which would then have to be interpreted) and then run it. This creates a larger startup time, but execution is still fast. Also, the next time the widget is loaded it does not have to be compiled anymore. Now, the runtime environment should be supplied with a compiler to do the *just-in-time compilation*. Just-in-time compilation can combine advantages of both interpreting and static compilation [2]. The Javascript engine of Google, called V8[1] uses just-in-time compilation and is known to be a fast Javascript engine.

The two disadvantages of compilation, being the decreased collaboration and transparency are both targeted by just-in-time compilation, at the expense of execution speed.

4.1.3 Other execution environments

Other execution environments are also possible. For example, Yahoo Pipes[2] is an online service which can be used to aggregate data from different sources and create data mashups. This can be done in a visual editor, predefined modules can be dragged and coupled to perform. Modules include data source modules, which fetch data from other website feeds, and modules which manipulate the fetched data. Most of the time, a pipe requires some user input. For example lets a People

[1] http://code.google.com/p/v8/

[2] http://pipes.yahoo.com/pipes/

Finder pipe requires a name. The name is used to search multiple search engines, parse the data and aggregate them and finally the pipe outputs the results in a format which you specified when the pipe was created. See Figure 4.1 for the Yahoo Pipes visual editor.

Such runtime environments can be more secure and more people might want to create widgets, because a visual editor is more appealing to learn than a complete scripting language. It also has some disadvantages, for example the number of modules and configurability limits the possibilities of creating interesting widgets.

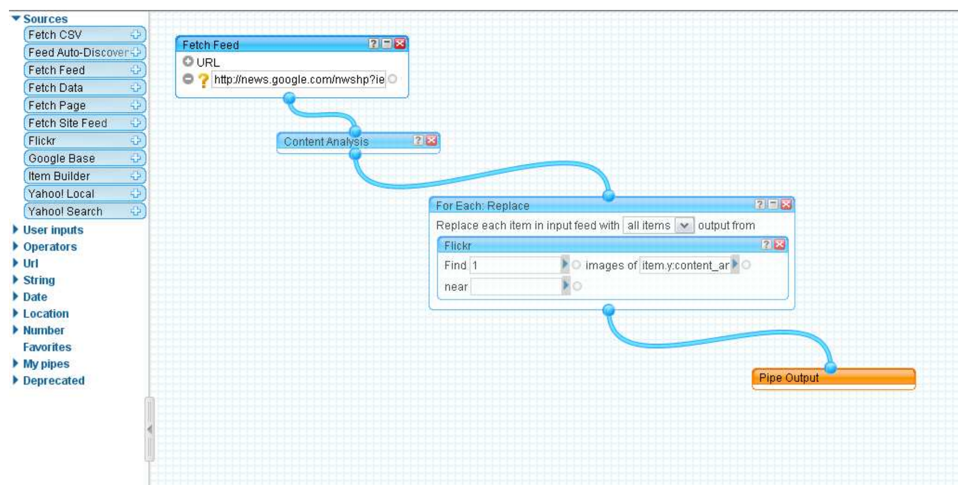

Figure 4.1: Yahoo Pipes visual editor

4.2 Code restriction

In this section, we will discuss the methods to restrict the code. Both functionality and resources need to be controlled, to create an effective sandbox as discussed in Section 5.3.2. We will first discuss how to restrict the functionality and then discuss how we can control the resource usage.

4.2.1 Methods of restriction

There are a multitude of ways to create a sandbox by restricting the code. Some depend on which kind of runtime environment is used. First, it is possible to instrument the source code with policy functions before it is executed. The policy functions require policies which tells whether the real function may be executed or not and prevents the function from running. Instrumenting source files is ofcourse not limited to be used by interpreters. An example of this technique is RestrictedPython[3], which can restrict Python programs. How well the sandbox works is

[3]http://pypi.python.org/pypi/RestrictedPython/3.4.2

defined by the policies.

Second, the source code (or intermediate data structure) may be verified before execution. When dangerous code is found, it the runtime may throw an error.

Using an interpreter, another way to create a sandbox is to modify the interpreter to disallow potential dangerous built-in functions by using run-time tests. For Python this is proven to be hard to do, as was proven by the original restricted and bastion mode, which is now deprecated because of too many vulnerabilities. Javascript however, might be easier because it does not have any system functions in its specification. All these potential dangerous functions will have to be added by extending Javascript. The reason why Google Gadgets and Firefox extensions are still dangerous is because they extend Javascript with potential dangerous functions and they let third parties extend Javascript by adding components written in other languages, such as C++.

When binaries are distributed, it is possible to use bytecode instrumentation. This technique is similar to source code instrumentation, but is less transparent. It adds additional bytes to the bytecode, which perform the necessary runtime tests. In this way, both restriction and resource usage controlling can be handled [5].

4.2.2 Controlling resource usage

Resources on a computer are typically shared by a multitude of programs. The operating system distributes the resources between the different programs. It makes sure that the programs are runned concurrently by using a scheduler and allocates some memory to the programs. The UNIX operating system also supports chroots, which change the current root directory of a program to a subdirectory. This way, the program can not read privacy sensitive data and can not overwrite or delete important files. However, it does not control the amount of disk space which can be written.

Using a widget system, the program should make sure that the resources are not consumed only by one widget. Using functionality restriction discussed above, we can either give widgets full access to a resource or no access. For example, all widgets may access the harddisk, but no restrictions on how much they may write or where they may write or read are deployed. Typical resources that widgets may have access to are CPU resources, memory, harddisk, network and other I/O devices. Herzog et al. [15] extends the current Java security mechanisms with resource control. It does so by creating policies which state to what extend the program may use the resources. For example, in the policy it is states how many bytes a program may write in a certain directory or how many network sockets may be opened. Because of the limitations of the Java virtual machine, policies for CPU or memory can not be created: It is impossible to ask the virtual machine for a certain amount of CPU or memory. After the policies are created, they should be enforced. Herzog et al. [15] extends the Java access controller for this purpose. The resources are monitored and when a policy is broken, the access to the resource fails. Binder et al. [4] uses Java bytecode rewriting to limit memory and CPU

access. With every memory allocation, resource limitation checks are inserted. To ensure normal CPU consumption, the number of executed bytecode instructions are counted per thread. Every once in a while, the number executed bytecode instructions are checked. When a thread exceeds its maximum number of executed bytecode instructions, the priority of the thread is lowered.

4.3 Creating a peer-to-peer API for widgets

Usually, widgets have access to the built-in library functions of the language they are coded in. These access rights may be restricted, as we saw in the previous section. The host program or widget engine may also extend the features that are available for their widgets. The widget engine then exposes an API to the widgets, such that widgets can perform various tasks.

These tasks may involve general tasks such as retrieving information about the system memory or processor usage, or more specific features for the widget engine. For example, Firefox extensions have an API which lets them access the Document Object Model (DOM) of the webpages the browser displays. If only general features are available, the platform would be just another widget platform.

To create widgets that run in a peer-to-peer environment, the widget engine should also have some specific peer-to-peer functions. Azureus[4], a BitTorrent client written in Java that supports plugins, exposes certain functionality which is very interesting:

1. Use the DDB (Distributed Database, actually a DHT) to make data available to other peers
2. Add additional messages to the peer-to-peer protocol to communicate with other users using the same plugin

These features can really make the difference between peer-to-peer and normal widgets. Imagine a peer-to-peer chat widget, a peer-to-peer RSS feed mechanisms or a peer-to-peer wiki. When the API is quite extensive, it is also possible to create the whole peer-to-peer functionality in widgets, just like the Eclipse Platform. This way, the program is easily extended with other features and new features are easily prototyped.

[4] http://azureus.sourceforge.net/

Chapter 5

Security issues in a peer-to-peer widget environment

Peer-to-peer widgets are a dangerous tool as both threats of portable code and threats to peer-to-peer systems are involved. Portable code security is a great issue because everyone can create and distribute their code using the internet. General peer-to-peer security is also a serious issue for peer-to-peer widget systems, because these threats could become worse since everyone can inject code in the community. Peer-to-peer systems present some serious security vulnerabilities. For example, when a vulnerability in the software is found, it is easily propagated since most client software cache IP addresses of recently discovered or accessed peers. These peers are likely to have the same vulnerability [23]. In this chapter, we will discuss several attacks on peer-to-peer systems currently known and further discuss the issue of portable code threats.

The outline of this chapter is as follows. First, peer-to-peer security threats are explained, including their current solutions. In Section 5.1, the threat of pollution is discussed and in Section 5.2 the problem of Distributed Denial of Service (DDoS) attacks is evaluated. Lastly, the problem of portable code security (malicious code) is discussed in Section 5.3.

5.1 Pollution

In this section, the problem of pollution in peer-to-peer systems is discussed. We begin by describing the problem and then discuss several solutions to the problem.

5.1.1 Problem description

Pollution is a sabotage technique where someone deposits content in the network which is useless or tampered with. Currently this technique is used very much in file sharing peer-to-peer systems, where a lot of copyrighted material can be

downloaded. The publishers of the copyrighted content (e.g., games, songs) might pay a pollution company to spread useless copies.

Pollution can be classified as **Content Pollution** and **Metadata Pollution**. Content Pollution means that the content is modified, e.g., replacing part of the song with white noise. The latter pollution strategy is not to tamper with the content, but inserting a different file with the metadata of this target file. Now, users who search for the targeted file will instead download something else [22].

Pollution might also be a problem in a widget repository, but then it would probably be polluted with malicious widgets. When widgets are open source, there will not be any copyrighted material which was a great incentive for publishers of copyrighted material to pollute. Nevertheless, malicious widgets can be seen as pollution and the solutions discussed below might work to prevent their spreading.

5.1.2 Solutions

Liang et al. [22] provide several anti-pollution mechanisms and classify them as either *detection with downloading* or *detection without downloading*. Matching and user filtering are methods classified under the former, while the latter comprises methods which depend on experience of other users: rigid trust, web of trust and reputation systems. In the following paragraphs, the methods are explained and their relevance to a widget repository is discussed.

Matching

This method uses a trusted database of signatures of authentic material. When the file is downloaded, it is checked whether a signature exists in the database. If no match is found, it is concluded that the file is polluted. For file sharing peer-to-peer systems, this method might not work as well as one wants it to, because in those systems there are a multitude of versions file most of the time, resulting in deletion of proper files. Especially songs might have radio versions, album versions, other bitrates and polluted files. For widgets, this system might be feasible since, as we have seen in Chapter 2, the magnitude of widget repositories is of another order. Note that it is also possible to let the database consist of signatures of polluted files instead of authentic files, which resembles the method of signatures used to combat malicious code, see Section 5.3.2.

The signature database can either be managed by a central authority or by the users itself. A central authority however kills the peer-to-peer paradigm, and on the other hand users can not always be trusted. But Kamvar et al. [18] notes that there are always a few pre-trusted peers within a community, e.g., the creators of the peer-to-peer community. These pre-trusted peers could create the signatures, while others only manage them.

User filtering

Most peer-to-peer file sharing systems copy downloaded files directory to the shared folder, distributing them further. Liang et al. [22] reckons that when users filter out the polluted versions from their shared folder, pollution would greatly reduce. The challenge is how to encourage users to filter out the polluted files.

We think this solution might not be appropriate for a widget repository because widgets would probably be distributed differently: there would not be any shared folder. Also, widgets would have to be executed before knowing whether they are bogus or malicious and by that time, it might already be too late.

Trust and reputation

In the **Rigid Trust** scheme, users only download from friends who the user trusts. Friends watch eachothers back and try to keep pollution out. This scheme might work fine for file sharing peer-to-peer applications, but for a widget repository it is more important who created the widget, as the widget creator is the one a user should trust or not.

Web of Trust is a trust scheme where a user retrieves updated lists of friends of friends and only downloads from this subset of peers. When he finds a polluted file, he notifies his direct friend and stops downloading from this user. This solution might not work for a widget repository for the exact same reasons as explained for the rigid trust scheme.

Reputation systems‘ are a good way to prevent pollution, when users engage actively in issuing honest ratings. Reputation systems are described in Section 3.2.1.

5.2 DDoS attacks

5.2.1 Problem description

Distributed Denial of Service (DDoS) attacks are attacks where a multitude of infected hosts attack a certain victim host. The infected hosts can either overwhelm the victim's connection resources by creating a lot of connections (*Connection attack*), or try to generate a lot of bandwidth such that the up- or downstream links of the victim congest (*Bandwidth attack*).

Current peer-to-peer systems can easily serve as a DDoS engine. Naoumov et al. [26] specifies two ways to exploit a peer-to-peer system for DDoS attacks, which will be discussed next. The first is *Index Poisoning* and the second is *Routing Poisoning*. Peer-to-peer systems with a widget engine could also get infected by malicious widgets which perform DDoS attacks, when the widgets are granted enough access. Malicious widgets are discussed in Section 5.3.

Index Poisoning

An attacker doing an index poisoning attack tries let the system believe there is a popular file located at the victims node. Note that the victim does not have to be part of the peer-to-peer system. The attacker sends publish messages into the system for this particular file to every node it finds. The nodes will add the file hash along with the victims location in their index. When someone queries for this file, the victims location is returned and a connection will be made to the victim. The victim will not understand the protocol specific message and may either ignore it, send an error message or terminate the connection. The hosts looking for the file may however retry to download the file from the victim. This attack, when the file is popular, ultimately results in a connection attack, where there victims connection resources are depleted. Moreover, because the index entry may exist for a while, the attack continues long after the attacker has stopped poisoning the index.

Routing Table Poisoning

A routing table poisoning attack is started by sending bogus node announcement messages to nodes. In these node anouncement messages, the victims IP address is included. Using knowledge of how the system manages its routing table, the attacker can make it more likely that nodes add the victim as a neighbour. After the routing tables of nodes are infected, they may route messages to the victim and thus receive a flood of messages. Since the victim will probably reply with an error message, both the up- and downlink will be flooded. However since most peer-to-peer systems will remove the entry from their routing table when the host is unreachable, the attack may fade away when the attacker stopped poisoning the routing table, unlike the index poisoning attack.

5.2.2 Solutions

In the following subsections, two solutions are discussed, namely Identity Based Cryptography as a solution for Index Poisoning in Section 5.2.2 and a solution for Routing Table Poisoning in Section 5.2.2.

Reliable Index Exchange Protocol

This solution to Index Poisoning in peer-to-peer systems was proposed by Lou et al. [23] and uses the concept of *peer accountability*. Normally, when a peer receives a publishing message from peer Y which says that a file with file identifier D can be found on peer P_u, it can not verify that the source of that message was also the one that published the file (the publisher of the file is denoted by P_b). For all it knows, peer Y might have forged the message and made D an identifier of a popular file and P_u an endpoint (IP address) of the victim. The possibility of such an Index Poisoning attack lies in the fact that X can not verify that P_u is in fact P_b. To eliminate this vulnerability, a peer should be able to verify this property. A peer is

accountable when there exists a function with parameter D such that it returns true when $P_b = P_u$.

The protocol, the Reliable Index Exchange Protocol (RIEP), by Lou et al. [23] is based on digital signatures to enforce peer accountability. When a peer P_b wishes to publish a file F, it must first request a private and public key. A public/private key pair is generated by a Private Key Generator (PKG), which is similar to a Certification Authority (see Section 5.3.2) in a Public Key Infrastructure (PKI) but only needs to generate the public/private key pairs. Then, P_b should publish the index using a secure file index format:

$$D(F, P_b), S_b(D), G, S_g(P_b)$$

where $S_b(D)$ is the original file index, digitally signed by P_b, G the identity of a PKG, and $S_g(P_b)$ the signature of G. The publisher signature is there to enfore peer accountability and G and S_g are to accomodate multiple PKGs and to provide security for the identity of the PKG. When a peer sends a query to another peer, it includes a signature. The peer replying with the file index (in the secure format specified above) also signs its message. To verify a file index, one needs to check whether $S_p(D)$ was signed with the key of P_b, which is equivalent to verifying $P_b = P_u$.

While this solution greatly reduces the number of targets an attacker can use (it can only use itself or previously authorized IP addresses), it also comes with extra computational overhead and the need for certification authorities to issue public/private key pairs [19]. It is however an elegant solution without compromising the anonimity requirements in peer-to-peer systems.

A solution for Routing Table Poisoning

The above solution for Index Poisoning using peer accountability (RIEP) could be used to avoid Routing Table Poisoning also, in the following way. When a peer P_u sends a peer announcement message (which consists of a peer identifier and an endpoint (usually IP address and port), it signes the message and it again includes the Private Key Generator identifier G and its signature $S_g(P_u)$. A receiver of this message can verify the authenticity of this message and add it to their routing table when it is verified or discard the message when it is not. Drawbacks of this method are again the computational burden on the peers and the need for certification authorities [19].

Another, more elegant solution to Routing Table Poisoning is proposed by Kohli et al. [19]. First, it is assumed that every peer has a secret value S. A peer A sends a node announcement message (NAM) to peer B, presenting the existence of peer V. Peer B then sends a message to V, consisting of a timestamp T and a hash of T, the NAM and its secret value S_b, but does not yet add V to its routing table until V's reply is valid. V either replies with NAM, T, and a hash of T, NAM and S_b. Peer B can now validate the reply by hashing NAM and T received from V and S_b and verifying it is the same as the hash returned by V. Since it is computational

infeasible to create the hash without knowing S_b, B can be sure that V is indeed on that location.

When an attacker tries to perform a routing poisoning attack, the peers it is trying to use will send one message to the victim and not add the victims address to their routing table. Thus the network now acts as a reflector instead of an amplifier: the attacker will have to send as much fake node announcement messages to other peers as the victim will receive, flooding its own link.

This solution is does not use a PKI and has lays less computational burden on the peers than the solution derived from RIEP.

5.3 Malicious code

The problem of malicious code is discussed in this section. First malicious code is described extensively and then several solutions are discussed, ranging from prevention mechanisms, accountability and detection.

5.3.1 Problem description

Malicious code (also known as malware, which comes from malicious software) is software that fulfills the deliberately harmful intent of an attacker when run. Malware thus includes viruses, worms, trojan horses, spyware, rootkits and adware, among others [20]. A computer virus is a program that attach itself to a host program and when run, infects other executables and propagates that way. A worm is a program that propagates itself through networks to jump from machine to machine. Trojan horses hide malicious intent inside a host program that seem to be useful and spyware is computer software used to send data about the infected user to third parties. Adware automatically plays, displays or downloads advertisements to a computer and lastly rootkits are collections of tools used by attackers after gaining administrative privileges. Rootkits are mostly used to hide the attackers presence, gather information about the system or to ensure that the attacker can regain access at a later time.

In the past decades, malware has become a fast growing problem for a number of reasons. McGraw et al. [25] argues that the following trends have a large influence on the recent wide spread propagation of malicious code. First, there is the growing connectivity of computers through the Internet, which makes it a lot easier to make an attack. Second, systems are becoming so large and complex that bugs are inevitable. These bugs can be exploited by attackers to install malicious code, but also improper configuration opens the door for attackers. It becomes easy to hide malicious code within a complex system such as an operating system. Third, most applications support some form of extensibility, which introduces mobile code that can be installed and runned on the fly. Extensibility makes it hard to prevent unwanted extensions.

5.3.2 Solutions

We now present several defenses against malicious code and discuss their merits and drawbacks. First, in Section 5.3.2, syntactic signatures and semantic signatures are discussed, which are used to detect malicious code. Then Section 5.3.2 discusses code signing, which is a technique to authenticate the code producer and to verify code integrity. Proof Carrying Code (PCC) is discussed in Section 5.3.2 and restricting the execution environment (the Sandbox model or using more fine-grained security policies) is treated in Section 5.3.2.

Detection using syntactic or semantic signatures

Most of todays malware detection systems (e.g., virus scanners) detect malicious code by scanning the code looking for syntactic signatures. Syntactic signatures specify byte sequences that are particular for a specific malicious program. Detection using syntactic signatures does have several drawbacks however. The first drawback is that every time a new malware program is found, analysts first have to find a signature, then add it to the database and then distribute it to everyone using their malware detector. Thus, there is a timewindow where unknown malware can act freely. Attacks exploiting this knowledge are called zero-day attacks. The second drawback, which is a reaction to the detection software, is that current malware is trying to avoid detection by obfuscation and stealth. Metamorphic malware, which is malware that mutates, renders the detection software using syntactic signatures useless.

For the reasons enunciated above, a new malware detection trend has come along, namely detection using semantic or behavioral properties. Because the behavior of the program is analyzed, this method can detect mutating software and identify unknown malware, which detection based on syntax could not. To create behavioral signatures, the malicious programs have to be examined to extract behavior that is specific for their class. For example, Kreugel et al. [20] notes that regular kernel modules have different goals than rootkit kernel modules, and thus completely different functionality. They use symbolic execution to statically analyze an executable. Symbolic execution is a technique in which programs are simulated using symbols instead of actual input data, and thus expresses the output and program state as mathematical or logical expressions. Symbolic execution basically runs the program for every input value simultaneously, which makes it ideal to analyze behavior.

Ellis [11] defines several behavioral signatures for detection of worms, which could identify classes of worms even previously unknown. Worms are especially good at zero-day attacks, spreading to millions of computers when it is still unknown, and thus behavioral signatures might work much better at battling them. One of the signatures is that a server changes into a client. A worm has to propagate itself and when a server is infected, will try to make connections to other computers to propagate, thus changing into a client. Another signature is the one

of alpha-in and alpha-out, which states that a worm typically sends similar data across nodes. These signatures might define behavior of a worm very well, but they are rather useless in a peer-to-peer environment.

Code Signing

Code signing is a means of authenticating the code producer and verifying the integrity of the code. The code producer should have generated a private and a public key pair, where everyone may know the public key, but noone except the code producer should know its private key. The code is digitally signed using the private key of the code producer, which outputs a signature. Using the public key of the code producer, the code and the signature, the host that is about to execute the code can now verify that the code is from the code producer and that it has not been tampered with.

However, while this accomplishes authentication and integrity, it does not make the code safe to execute because the code producer might not be trustworthy. One might use a public key infrastructure (PKI) to bind respective code producers to public keys. This requires a Certificate Authority or a Trusted Third Party to verify that the code producer is trustworthy. Since a PKI is based on central authorities, it can not be used in a peer-to-peer system without compromising a bit of the paradigm. Reputation systems (see Chapter 3, Section 3.2.1) can however be used to build a reputation in a peer-to-peer system, which could be used to trust peers.

Another drawback of code signing is that either the code is trusted and is given full permission or the code is rejected and not executed at all, based on a signature [24]. Code signing could however be combined with other defenses, such as taking policing action which is discussed in Section 5.3.2.

Proof Carrying Code

Proof Carrying Code (PCC) is a technique that enables the host system to test whether the supplied program is safe. For this to work, the code supplier has to supply a safety proof. The host can then validate the proof without using cryptography or consulting external agents [27]. An important property of this mechanism is that whenever the code or proof is tampered with, the validation will either fail or, when it does not fail, the program still is safe. It is said to be intrinsically safe.

The original paper uses this mechanism to proof type safety of a program, and the proofs are relatively simple. For mobile code however, there are a lot of properties of the code which would have to be proven, e.g., that the code will not do harm to the host, or that the code does not use too many resources [21]. The same paper reports a list of advantages when using PCC:

1. Effort for the proof is at the code producer

2. The code consumer does not care how the proof is constructed

3. PCC programs are “tamperproof”

4. No cryptography or third party has to be used

While PCC seems very promising, it is however questionable if it will ever be used in a broad sense. Generating proofs for complex programs and complex properties might be too hazardous. For now, PCC is still in its embryonic stages of development and is not yet a practical solution for complex programs [12].

Sandbox Model and security policies

The idea behind the Sandbox model is to let possibly untrusted code execute in a restricted environment so it can not do any harm to the host running the code. A sandbox can be characterized by two different mechanisms [24]:

- it confines code, either through type checking, language properties, or the use of protection domains to prevent the subversion of trusted code and,
- it enforces a fixed policy for the execution of code.

Of course, the sandbox is only safe when the policy is not flawed or inconsistent, something which is not so trivial. Python used to have a restricted execution and bastion mode, which the code deployer could customize. The execution mode was however easy to circumvent and is now deprecated [15]. A better example might be Java, where the early implementations used the sandbox security model and have come a long way since, refining their model and making it more secure. While the Java Security Manager is one of the more advanced security models, Herzog et al. [15] indicates two shortcomings of the Java Security Manager, namely the limited possibilities for resource control and the difficulty of creating policies. Also, new vulnerabilities might be found and exploited, which requires the developers to keep the security system up to date.

All in all, using policies to restrict the execution environment is quite a neat solution to malicious code, but it also introduces a new dilemma because it also restricts the usefulness of code. When the execution environment is restricted too much, no one will be interested in coding because there is nothing interesting to create. However, when the restrictions are lowered, it creates possibilities for malicious code.

Chapter 6

Conclusions

In this chapter, we will give a summary of the material treated, and conclusions. Also, we will lay out the further research that we will be conducting on peer-to-peer widget environments.

6.1 Summary and conclusions

In this report, we have reviewed several centralized widget systems and discussed a number techniques for the creation of a decentralized widget system. Decentralizing the distribution of widgets is a challenge, which uses state of the art peer-to-peer techniques to store, manage and moderate the widgets. A central authority is no longer needed, but decentralization also amplifies the possibility of injection of malicious and low quality widgets. We have also discussed the possibilities for runtime environments, where there is a tension between speed and ease of distribution, choosing between either compiling or interpreting, respectively. We discussed several techniques to restrict mobile code for these runtime environments to alleviate security threats and showed some new possible features when using a peer-to-peer environment for widgets. Finally, we discussed several existing security threats to peer-to-peer systems and to mobile code environments.

Although several systems are already very close to a peer-to-peer widget system, they lack either the decentralized distribution of the widgets (Azureus) or a peer-to-peer environment to run the widgets. A number of techniques are available to create a peer-to-peer widget environment, but the widget system should be carefully designed to make sure the system will not collapse (e.g., because of scalability problems) or be subverted by malevolent users.

Theoretically, the techniques described here could be put together to create a peer-to-peer widget environment that has the same functionality as the widget systems described in Chapter 2. However, whether the peer-to-peer widget environment will be robust, scalable, reliable and secure is not obvious.

The techniques discussed to decentralize the widget repository (DHTs and gossip-

based systems) have been researched extensibly using simulations, but whether they work as good in practice should still be pointed out. This is because there have not been many large-scale deployments of these techniques yet. In particular, the effects of high churn and unconnectability in DHTs and the scalability in gossip-based systems could be possible problems.

To resist against pollution and malicious widgets, both security and moderation techniques could be used, but they are almost always one step behind. Whenever there is one infection, the peer-to-peer widget environment supports a very fast dissemination, because other peers are likely to have the same vulnerability. Current widget systems do not use a lot of security measures, but we think that a peer-to-peer widget environment gives need for more security techniques as infections could be more disastrous.

6.2 Further research

In the subsequent Master of Science project, we will make the first step to a robust and secure widget environment for the social peer-to-peer program Tribler[1]. First, we will create a few built-in widgets, which the user can activate to use. Second, we will design and implement WidgetCast, a protocol to exchange information on widgets (e.g., who has which widgets installed and general information on the widgets). Using WidgetCast, widget swarms can be created, which are similar to BitTorrent swarms. Users of a specific widget are grouped together in a widget swarm, creating the possibility to distribute the widget code and to let the widget exchange information between different widget instances. After these first two milestones, widgets could be made available for everyone, which means that security measures should be taken and a distributed widget repository should be made. While we have discussed several techniques for both security and a distributed repository, they should be investigated further to make sure they will work in practice.

[1] http://www.tribler.org

Bibliography

[1] Elias Athanasopoulos, Andreas Makridakis, Spyros Antonatos, Demetres Antoniades, Sotiris Ioannidis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Antisocial networks: Turning a social network into a botnet. In *Proceedings of the 11th Information Security Conference (ISC 2008)*, 2008.

[2] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35:97–113, 2003.

[3] Philippe Beaucamps and Daniel Reynaud. Malicious firefox extensions, June 2008.

[4] Walter Binder, Jane G. Hulaas, and Alex Villazón. Portable resource control in java. *SIGPLAN Not.*, 36(11):139–155, 2001.

[5] Ajay Chander, John C. Mitchell, and Insik Shin. Mobile code security by java bytecode instrumentation. In *2001 DARPA Information Survivability Conference & Exposition (DISCEX*, pages 1027–1040, 2001.

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Ch, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of the Conference on Operating System Design and Implementation (OSDI)*, pages 205–218, 2006.

[7] 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.

[8] Jim des Rivieres (IBM). Eclipse platform: Technical overview. Technical report, Object Technology International, Inc., 2003.

[9] John R. Douceur. The sybil attack. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 251–260, London, UK, 2002. Springer-Verlag.

[10] Debojyoti Dutta, Ashish Goel, Ramesh Govindan, and Hui Zhang. The design of a distributed rating scheme for peer-to-peer systems. In *Workshop on Economics of Peer-to-Peer Systems*, 2003.

[11] Daniel R. Ellis, John G. Aiken, Kira S. Attwood, and Scott D. Tenaglia. A behavioral approach to worm detection. In *WORM '04: Proceedings of the 2004 ACM workshop on Rapid malcode*, pages 43–53, New York, NY, USA, 2004. ACM.

[12] Joan Feigenbaum and Peter Lee. Trust management and proofcarrying code in secure mobile-code applications. In *DARPA Workshop on Foundations for Secure Mobile Code Workshop. DARPA*, 1997.

[13] Dick Grune, C. Jacobs, Koen Langendoen, and Henri Bal. *Modern Compiler Design*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[14] Vincent Heinink. Metadata infrastructure for peer-to-peer video. Master's thesis, Delft University of Technology, June 2008.

[15] Almut Herzog. *Usable Security Policies for Runtime Environments*. PhD thesis, Electronic Press, Linköping, 2007.

[16] Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed object location in a dynamic network. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 41–52, New York, NY, USA, 2002. ACM.

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

[18] S.D. Kamvar, M.T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *12th WWW conference*, New York, 2003.

[19] Pankaj Kohli and Umadevi Ganugula. Ddos attacks using p2p networks. 2007.

[20] Christopher Kreugel. *Malware detection*, chapter Characterizing the Behavior and Structure of Malicious Executables, pages 63–83. Advances in Information Security. Springer US, 2005.

[21] Peter Lee and George Necula. Research on proof-carrying code on mobile-code security. In *Proceedings of the Workshop on Foundations of Mobile Code Security*, 1997.

[22] J. Liang, R. Kumar, Y. Xi, and K. Ross. Pollution in p2p file sharing systems. In *IEEE Infocom*, Miami, FL, USA, March 2005.

[23] Xiaosong Lou, Student Member Ieee, Kai Hwang, and Fellow Ieee. Prevention of index-poisoning ddos attacks in peer-to-peer file-sharing networks. In *Multimedia, Special Issue on Content Storage and Delivery in P2P Networks*, 2006.

[24] Sergio Loureiro, Refik Molva, and Yves Roudier. Mobile code security. In *Proceedings of ISYPAR 2000 (4me Ecole d'Informatique des Systmes Parallles et Rpartis), Code*, 2000.

[25] G. McGraw and G. Morrisett. Attacking malicious code: a report to the infosec research council. *IEEE Software*, 17(5):33–41, Sept.–Oct. 2000.

[26] Naoum Naoumov and Keith Ross. Exploiting p2p systems for ddos attacks. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, page 47, New York, NY, USA, 2006. ACM.

[27] George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[28] R. Rahman, D. Hales, M. Meulpolder, M. Clements, V. Heinink, J. Pouwelse, and H. Sips. Robust vote sampling in a p2p media distribution system. Technical Report PDS-2008-004, Technical University of Delft, 2008.

[29] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of International Middleware Conference*, pages 21–40, 2003.

[30] stderr. Facebook applications revealed. *2600 Hackers Quarterly*, 24:32–33, 2007-2008.

[31] Chunqiang Tang, Zhichen Xu, and Mallik Mahalingam. Peersearch: Efficient information retrieval in peer-to-peer networks. Technical Report HPL-2002-198, Hewlett-Packard Labs, 2002.

[32] Ye Tian, Di Wu, and Kam-Wing Ng. On distributed rating systems for peer-to-peer networks. *Computer Journal 2008, vol 51; number 2, pages 162-180*, 2008.

[33] Y. Wang and J. Vassileva. Trust and reputation model in peer-to-peer networks. In *Proc. Third International Conference on Peer-to-Peer Computing (P2P 2003)*, pages 150–157, 1–3 Sept. 2003.

[34] Li Xiong and Ling Liu. Peertrust: supporting reputation-based trust for peer-to-peer electronic communities. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):843–857, July 2004.

[35] Runfang Zhou, Kai Hwang, and Min Cai. Gossiptrust for fast reputation aggregation in peer-to-peer networks. *IEEE Transactions on Knowledge and Data Engineering*, 20(9):1282–1295, Sept. 2008.