Tribler-G: A Decentralized Social Network for Playing Chess Online

Egbert Bouman

Delft University of Technology

Tribler-G: A Decentralized Social Network for Playing Chess Online

Master's Thesis in Computer Science

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

Egbert Bouman

19th February 2012

Author
Egbert Bouman

Title
Tribler-G: A Decentralized Social Network for Playing Chess Online

MSc presentation
1st March 2012

Graduation Committee

prof.dr.ir. D. H. J. Epema (chair)	Delft University of Technology
dr.ir. A. Iosup	Delft University of Technology
dr.ir. A. R. Bidarra	Delft University of Technology

Abstract

There are currently many systems that offer online board games, for instance, social networking sites, where board games enjoy enormous popularity, but also systems such as the Internet Chess Club, which has been around for over a decade. However, these systems are all centralized and typically have drawbacks for the user, such as subscription fees or advertisements. As an alternative, we have designed a decentralized protocol, called GameCast, that enables users to play turn-based multi-player board games over a peer-to-peer network.

The GameCast protocol supports three processes, dissemination of peer and game information within the network, game agreement, which allows one peer to invite another by sending invites, and game-play, which enables peers to play a game over the network. Our current GameCast implementation, called Tribler-G, is built as an extension to the Tribler file-sharing application, and focuses on enabling users to play online chess through the Tribler-G GUI.

To evaluate the performance of the GameCast protocol, we have created GameTest, a system capable of emulating a peer-to-peer network. Using GameTest, we have conducted a large-scale emulation of hundreds of peers on the DAS-4 distributed supercomputer. The results acquired during the emulation show that GameCast scales well and uses little bandwidth. Additionally, we have performed user tests, the results of which show that users are generally positive about Tribler-G in terms of usability.

Preface

The document before you is my Master of Science thesis, and represents my final work as an MSc student. The presented research was performed at the Parallel and Distributed Systems group of the Faculty of EEMCS of Delft University of Technology. This thesis describes my research on the creation of a decentralized system that allows users to play turn-based board games over a peer-to-peer network.

There are a number of people that I would like to thank for their support and help in making this thesis. First, I would like to thank Alexandru Iosup and Dick Epema for giving me invaluable help and advice during my work. I would also like to thank my parents for their support, both morally and financially. Furthermore, I would like to thank the members of the graduation committee for taking the time to read this document and for providing many helpful suggestions. I would further like to thank Boudewijn Schoon and the other members of the Tribler team for their comments and suggestions. And finally, I would like to thank the volunteers who participated in the user testing.

Egbert Bouman

Delft, The Netherlands
19th February 2012

Contents

Chapter 1

Introduction

Despite the availability of many modern online games which use complex 3D graphics, traditional board games are still quite popular, as evidenced by their popularity on online social network systems. Think, for instance, of games available on Facebook such as Chess, Checkers and Go. Further strengthened by the enormous popularity of board games on smart phones and tablet devices, it is likely that board games will continue to flourish for the foreseeable future.

Currently, users who wish to play board games online have a number of options. For instance, they can choose to use social networking sites, such as Facebook, or websites that offer similar services, such as Yahoo! Games. Additionally, there are also several services in existence that focus on one particular game, such as the Internet Chess Club (ICC), the Free Internet Chess Server (FICS), and the KGS Go Server. However, what most of these services have in common is that they need to generate revenue, which typically means users have to endure advertisements or pay subscription fees. Furthermore, the centralized architecture used by these systems introduces a single point of failure, high game hosting costs, and poor scalability characteristics.

To provide users with a more attractive alternative to the current systems, we have created a decentralized system that allows users to play turn-based board games over a peer-to-peer network. The protocol that governs the interactions between peers in our decentralized gaming system is called GameCast. In order for the system to function properly, we need GameCast to address several problems. First, we need an overlay network which we can use for the distribution of peer and game information. GameCast realizes this through the use of an epidemic mechanism, similar to the one used by the Tribler peer-to-peer file-sharing application [33], which was also developed in Delft. Secondly, with the overlay network in place, we need a mechanism that allows people to contact each other and agree to play a game. And finally, once a game agreement is reached, we require mechanisms to allow for the actual game-play. The current implementation of our decentralized gaming system, called Tribler-G, is built as an extension of Tribler, and focuses on enabling users to play online chess.

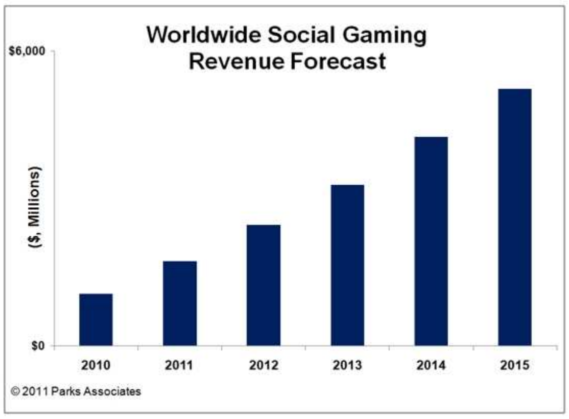


Figure 1.1: Predicted social games market growth [14].

In this chapter we provide an introduction to online gaming systems, social network systems and peer-to-peer networks; it is organized as follows. Section 1.1 gives an introduction to online gaming systems, and goes into detail about several aspects of current systems. Sections 1.2 and 1.3 discuss online social network systems and peer-to-peer networks, followed by Section 1.4 which discusses epidemic protocols. Section 1.5 states the contributions of this thesis. Finally, Section 1.6 provides an overview of the remaining chapters.

1.1 Online gaming systems

In this section we will discuss several aspects of online gaming systems, beginning with the groups of games that are currently popular: social games, massively multi-player online games or MMOGs, and casual games. First, social games are games that integrate with social networking sites (see Section 1.2) in a way that enhances the game-play. The people who play these games are generally not hardcore gamers, and the games themselves are usually multi-player and turn based. The social gaming industry has undergone, and will continue to undergo, explosive growth (see Figure 1.1). For example, Facebook, one of the largest online social networks, is currently the biggest player in the social gaming market, and 66% of its traffic is related to social games [30]. Second, MMOGs are games that are capable of supporting a large number of players. Examples include World of Warcraft, Call of Duty Modern Warfare 3, and Lord of the Rings Online. Finally, casual games are games that have (almost) no learning curve and usually have a sort duration. Examples of services that offer such games include Pogo, Cafe, Doof and Kongregate. It should be pointed out that these groups of games are not separate, and games can belong to multiple groups.

The way online gaming systems can be accessed varies. Some games run on online social networking systems, while others run on regular web-sites. Games that offer more complex graphics typically require the installation of a stand-alone application.

Most of the online gaming systems run on a traditional client/server architecture, which has significant drawbacks. First, centralized systems introduce high costs, especially if companies need to over-provision a large number of servers in order to cope with a sudden increase in the number of users. Second, relying on a centralized architecture introduces a single point of failure. This means that due to, for instance, a DDoS attack or because of a hardware issue, the system could face significant downtime. Finally, the scalability of central systems is limited by the capacity of a single server or cluster of servers. The centralized architecture has also various advantages, such as less software complexity and more control over the system since it relies on a central authority.

Since most online gaming system are run by companies, they often attempt to generate revenue, which is often done by showing advertisements while the system is being used, or by direct payment. Direct payment is usually achieved by requiring that the user pays for the application that is used to access the system, by requiring a periodically subscription free, or by selling virtual goods within the game. Finally, there are several systems that do not attempt to generate revenue, but still require donations to pay maintenance and hosting costs.

1.2 Online social network systems

Especially in the last several years, the Internet has become a popular place to interact with one another. There are many different ways in which people can collaborate, maintain social relationships, and share content, all supported by online social network systems.

Before continuing further, let us provide definitions for a number of entities related to this subject. First, a social network represents a map of all links between the people as they interact and create relationships, whereas an online social network refers to a social network that is formed in an online setting. Second, online social network systems are systems that support and manage online social networks. A type of online social network system that is particularly often mentioned in the media are social networking sites, which are defined as “web-based services that allow individuals to (1) construct a public or semi-public profile within a bounded system, (2) articulate a list of other users with whom they share a connection, and (3) view and traverse their list of connections and those made by others within the system” [20].

Social networking sites currently enjoy tremendous popularity. Think, for instance, of Facebook and LinkedIn. But other popular online social systems also exist. For instance, Youtube (a video sharing service), and Fickr (photo sharing service). Currently in-use systems exist mostly on centralized architectures, and while there do exist decentralized systems such as Diaspora [4], NoseRub [11], GNU Social [7], they have yet to again a reasonably sized user base. For more technical details regarding social networks, please refer to our literature survey [19]

Since we will be constructing an overlay network based on how frequent gamers play games against each other, the resulting network can be considered a social network, according to the definition above. Unlike in many of the existing systems, in our social network system, the creation of relationships does not happen explicitly, but implicitly by picking opponents against who you wish to play.

1.3 Peer-to-peer networks

Peer-to-peer networks are self-organizing networks consisting of interconnected nodes which participate in the sharing of resources. Peer-to-peer networks have no notion of clients and servers, and in their purest form, the network only exists of equal nodes. However, equality of nodes is not a requirement for peer-to-peer systems, and there also exist hybrid forms in which the network commonly has two types of nodes (namely, regular and 'super' nodes). Since peer-to-peer networks have no servers that manage the operations within the network, nodes are required to independently perform tasks such as searching, managing connections with other nodes, and message forwarding. Additionally, in order to maintain the overall performance of the peer-to-peer network, nodes are required to adapt to failures that can arise in network connections with other nodes or at the nodes themselves. This behaviour leads to a highly dynamic network topology that is always changing as nodes enter/leave the network and connections are created/dropped. One of the most widely used peer-to-peer networks today is the BitTorrent file-sharing platform [23]. In fact, BitTorrent is so popular, that it is responsible for a considerable portion of all internet traffic (see Figure 1.2).

Peer-to-peer networks often create an overlay network on top of the physical network topology. Overlay networks, which decide the network organization and location and routing algorithms, can be classified into two categories: structured and unstructured networks. When dealing with structured peer-to-peer networks, nodes have a fixed limit on the number of other nodes that they are allowed to connect to. Furthermore, in structured networks there are rules that nodes are required to follow when deciding to which nodes they should connect. Examples of structured peer-to-peer networks are Chord [35], Overnet [24], and Tribler [33]. In unstructured peer-to-peer networks, nodes have no fixed limit on the number of connections that they establish to other nodes, nor do nodes have rules that regulate to which nodes connections can be made. Example of unstructured peer-to-peer networks include Gnutella [8] and Freenet [22].

Tribler is a file-sharing application based on the BitTorrent protocol, and developed at the Delft University of Technology. Tribler attempts to improve the BitTorrent protocol with features that were not available initially, while remaining backwards compatible with the original protocol. Most of these features tend to make the network more social. An example of such a feature is Tribler's recom-

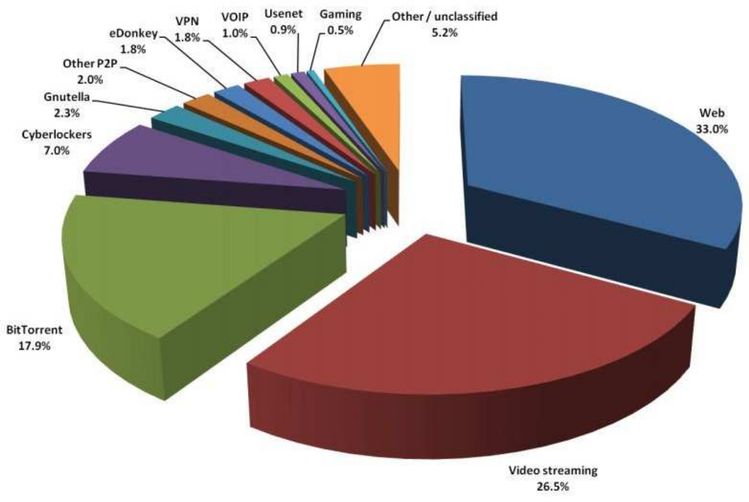


Figure 1.2: Estimate of overall internet usage for 2009 [27].

mendation system, which presents users with files he/she may want to download. Another example is the remote search feature, which enables users to search for files through querying remote peers. However, most of these improvements require peers to be able to contact each other, even if they do not reside in a common swarm. The original BitTorrent protocol does not allow this. In order to enable peers to contact each other, even if they do not reside in a common swarm (i.e, a group of peers sharing a file or group of files), Tribler introduces the notion of the overlay swarm. The overlay swarm is basically a virtual swarm which encompasses all peers that are using the Tribler software. Tribler constructs the overlay swarm through the use of an epidemic protocol, called BuddyCast (see Section 2.2.1). When constructing an overlay network for our own system, we utilize a mechanism based on BuddyCast.

1.4 Epidemic protocols

In epidemic protocols, or gossip protocols, information is spread within a computer network much like disease or infection spreads within a population, hence the name epidemic protocols. Using the terminology from epidemics, a certain node in the network can either be: infected, if it has an update (i.e., piece of information) and is willing to spread it; susceptible, if has yet to receive a certain update; and removed if it has a certain update, but is not willing to share it. Epidemic protocols are simple, scalable, easy to deploy, and are very reliable even dealing with of a lot of link failures.

Various propagation models exist, such as that of anti-entropy, a model originally proposed for replicated database maintenance. In the anti-entropy model, an infected node randomly chooses another node and exchanges updates. The exchange of updates can take place in a number of ways. In a push approach, the infected node sends its own updates. When using a pull approach, the infected node receives updates. And the final approach, push-pull, is a combination of the former two (i.e., both nodes send updates to each other). In the anti-entropy model, nodes never become removed.

Information dissemination in our work, which is based on BuddyCast, is achieved using a model very similar to the anti-entropy model with push-pull update exchanges. However, since we are dealing with a peer-to-peer network, a network in which lack a complete list of all nodes it is comprised of, we cannot choose a random node from the entire population. Therefore, we choose a node from a sample of the population, where the sample is comprised of random nodes and nodes that are frequently played against (see also Section 3.3.2).

1.5 Contributions

To address the issues introduced by gaming systems with a centralized architecture (see Section 1.1), we design and implement a decentralized system that allows users to play turn-based board games over a peer-to-peer network. We also design and implement a separate overlay network in which gamers that frequently play against each other cluster, promoting the creation of communities of gamers. By focusing on the social aspect of the overlay network's structure, we enable gameplay in a way that is similar to the games played on social networking sites. We implement our decentralized system as an extension to the Tribler peer-to-peer file-sharing application. As an added benefit, we hope that these new functionalities could further expand the user-base of Tribler. In this thesis we provide answers to the following research questions:

1. How to design and implement a protocol that enables playing turn-based board games over a peer-to-peer network?
2. How does the protocol that we design for playing turn-based board games perform?

In order to provide answers to these questions, we face the following technical challenges (challenges 1-3 are related to the first research question, whereas challenges 4-5 are related to the second research question):

1. How to create a suitable overlay network on the physical topology?
2. How to provide users with the ability to agree on playing a game?
3. How to provide users with the ability to play a game?
4. How to emulate a large network of game playing peers?
5. How to assess the performance of a network of game playing peers?

1.6 Thesis layout

The remainder of this document exists of five chapters and is organized as follows. Chapter 2 discusses related work in the area of online gaming and also gives an overview of existing mechanisms that could help to solve our problem. Chapter 3 defines the functional and non-functional requirements of Tribler-G, along with its design and implementation. Next, Chapter 4 provides an evaluation of the GameCast protocol through several experiments. Chapter 5 provides the results of the user testing. Finally, in Chapter 6, conclusions are drawn from previous chapters and future work is discussed.

Chapter 2

Background

In a traditional online gaming system, clients in the network send the game information to a single server (or cluster of servers). It is not difficult to imagine that using such a client/server architecture, the server will eventually become a bottleneck as the number of clients grows. Despite this potential concern, many, if not all, of the truly popular online gaming systems today rely on a centralized architecture. These systems serve millions of users each day, and require over-provisioning in order to guarantee that the system remains working properly, even if the number of users suddenly increases.

Decentralized solutions are believed to offer better scalability, while keeping game hosting costs down. Unfortunately, it is also considerably more challenging. An example of a challenge that we face is the construction of an overlay network, which we can utilize to achieve large-scale distribution of peer and game information. Tribler, a peer-to-peer file-sharing client based on the BitTorrent protocol, already has such a overlay network in place. However, this overlay is structured in a way that users with similar downloading tastes are more likely to be connected to each other than users with different tastes. For the game network, it is unlikely that this same structure is of use in disseminating game events, and therefore we require a separate overlay network.

Additionally, we require a protocol that allows players to invite another, play games and discuss games. Internet Chess Servers (ICSs) use a simple text-based protocol that already suits most of our needs. While this protocol is really only meant to be used for playing chess, its syntax can easily be adjusted to work on a variety of other games as well.

The remainder of this chapter is organized as follows. Section 2.1 provides an overview of various existing centralized and decentralized online gaming systems, while Section 2.2 introduces existing mechanisms that we used in our work.

2.1 Related work

In this section we will discuss current online gaming systems. First, we will elaborate on various popular centralized systems. Second, we will briefly discuss several decentralized gaming systems and explain how they differ from our work.

2.1.1 Centralized gaming systems

Currently popular gaming systems are mostly using central servers. For example, World of Warcraft [18], on of the most popular Massively Multi-player Online Game (MMOG) today has more than 10 million subscribers [17]. Another example is Call of Duty Modern Warfare 3 [1], a First-Person Shooter (FPS) which has been reported to have around 20 million unique online players each month [13].

However, games do not necessarily need to have complex 3D graphics in order to attract a large audience. Think, for instance, of Farmville, a social networking game that has almost 30 million daily users [21]. But also more traditional board and card games such as Chess, Checkers and Poker, have a substantial user-base on websites such as Facebook and Yahoo! Games. Additionally, with the enormous popularity of smart phones and tablet device, Android/iOS games such as Wordfeud [15] (a word game based on Scrabble) have enjoyed tremendous popularity.

Additionally, besides the relatively recent services, there are also several central gaming systems that have been around for more than a decade. Think, for instance, of the Internet Chess Club (ICC) [9], the Free Internet Chess Server (FICS) [5] and the KGS Go Server [10].

What most of these central systems have in common is that they need to generate revenue, which typically means users have to endure advertisements or pay subscription fees. Furthermore, the central services introduce a single point of failure, large game hosting costs and generally do not scale well.

2.1.2 Decentralized gaming systems

Most of the current research into peer-to-peer gaming systems focusses on providing players with a virtual world, in which the player can move around and interact with his/her environment. Examples include games such as Donnybrook [16], a first-person shooter, and OpenTTD [31], an open source reimplementation of the Real Time Strategy (RTS) game Transport Tycoon Deluxe. Contrary to these systems, Tribler-G aims to provide games that are much more similar to those of social networking sites in terms of complexity. Additionally, we focus on structuring the overlay network such that gamers that frequently play together cluster, thereby implicitly creating virtual communities of gamers.

Our work provides online gaming services over the Tribler file-sharing network. Building a gaming system on top of a file-sharing network is not new, and various other systems exist that utilize file-sharing networks to provide gaming capabilities. Think, for example, of PastryMMOG [26], which uses PAST, a distributed file system on top of Pastry, and P2P Second Life [37], a MMOG that is build on top of the Kad network. However, Tribler-G is, to the best of our knowledge, the first system that provides gaming functionalities over the Tribler network.

In GameCast, the peer-to-peer protocol that Tribler-G utilizes, the creator/owner of a game is responsible for administrative tasks such as notifying all players that the game has started, distributing game information and ensuring that comments are posted. To some extent, this is comparable to systems in which one or more peers are given the task of managing a certain region of the virtual world. Examples of such systems are PastryMMOG and MOPAR [38].

2.2 Building blocks

In this section we provide an overview of existing mechanisms that we used as building blocks while creating our decentralized online gaming system.

2.2.1 BuddyCast

In the introductory chapter of this thesis we already briefly mentioned that Tribler introduces the notion of an overlay swarm in order to enable peers to contact each other, even if they do not reside in a common swarm. The overlay swarm is basically a virtual swarm which encompasses all peers that are using the Tribler software. The most notable property of the overlay swarm is that is does not use a central tracker, unlike traditional swarms. The overlay swarm is secured through the use public-key cryptography, in which each peer is given a public/private key-pair. Peers on the overlay swarm are identified using their public key, which Tribler calls a PermID (or Permanent IDentifier). Each PermID maps to a single IP address and port number. When peers on the overlay swarm contact each other they need to exchange and validate their PermIDs, after which communication can start.

While peers download files from other peers, they build up a preference list. A preference list contains all files that a peer has downloaded in the past. Utilizing the overlay swarm, peers exchange these lists. The collection of all lists that are gathered by a certain peer is called the preference cache of the peer. Using the preference cache, a peer is able to calculate its similarity to other peers. Peers with high similarity are called taste buddies. Beside exchanging preference lists,

Preference list	**10 Taste buddies:** permid, ip, port, last_seen, preference list	**10 Random peers:** permid, ip, port, last_seen

Figure 2.1: The BuddyCast message format.

peers also exchange lists of taste buddies and random peers, in order to distribute the peer information throughout the network. BuddyCast refers to the algorithm that manages the exchange of preference and peer lists [32]. Peers spread their preference and peer list by sending BuddyCast messages (see Figure 2.1) either periodically, or in response to a received BuddyCast message. When a peer decides to send a BuddyCast message, it needs to select a target peer. The type of the target peer alternates between a taste buddy and a random peer. The reason for alternating between these types of peers is to enable the discovery of peers that are even more similar. Once a BuddyCast message has been send to a target peer, that specific peer will not be send another message for a predefined period of time. Also, once a peer receives a BuddyCast message from another peer, it will ignore all subsequent messages until a predefined period of time has passed.

When a peer first joins the network, it needs an initial list of peers in order to start participating in the exchange of BuddyCast messages. This problem is solved using a number of special super-peers which provide the necessary information to newly arrived peers. These super-peers can be seen as hubs within the network, keeping the network connected and providing a low network diameter.

Several of the features of Tribler rely on maintaining connections between taste buddies. For instance, the recommendation feature uses the similarity between the peers to recommend other downloads. Its easy to see what the idea behind this is: if peers a and b are taste buddies, then a may also be interested in downloads of b that a has not downloaded yet. Another feature that utilizes taste buddies is the remote search, in which a peer may query the databases of taste buddies in order to find needed content.

In order to realize Tribler-G, we decided to create a separate overlay network, which we call the game network. The game network is created using a algorithm very similar to BuddyCast. However, we will not be distributing preference lists, but game information. Additionally, the network will not be structured based on similar downloading tastes of users, but based on how frequently they play games against each other. Finally, we will use the structure of the overlay network in order to find suitable opponents

2.2.2 Internet Chess Servers

A popular way to play chess online is through the use of an Internet Chess Server (ICS). Communication between an ICS and a player is achieved through the use of a text-based telnet protocol. Because of this an ICS can be contacted using a standard telnet client (see Listing 2.1). However, most of the users use some sort of graphical interface to contact the ICS. Examples of popular interfaces are e.g., xboard/Winboard, PyChess and BabasChess.

The ICS protocol enables users to invite another and subsequently play a game of chess, and while this protocol is really only meant to be used for playing chess, its syntax can easily be adjusted to work on a variety of other games as well. Therefore, we use the ICS protocol as a basis for GameCast. The advantage of using

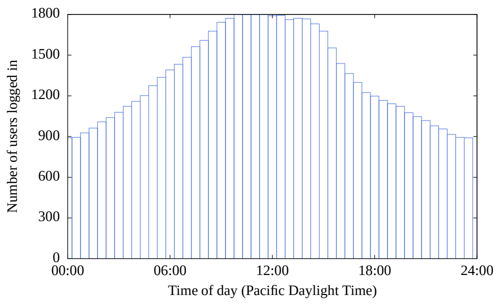


Figure 2.2: Average number of users logged in on the Free Internet Chess Server (FICS) during the day.

the ICS protocol syntax is that we can more easily involve the ICS community in our peer-to-peer gaming application. The ICSs have a substantial user-base. For example, the Free Internet Chess Server (FICS) has over 300,000 registered users and at least 900 users are logged in at any time (see Figure 2.2).

The telnet output shown in Listing 2.1 is called ”style 1” on most ICSes. This style is mostly used by users who play chess using a telnet client. The current default style is ”style 12”, which displays the entire game state (the board, how much time the players have left, etc.) in a single line of text. The reason for this change of style is that the output from ”style 12” is much easier to parse by a software application then the ”style 1” output. Note that changing output style merely ensures that the board is displayed differently, everything else remains the same.

Frequently used ICSes

By far the largest part of the community plays on only a handful of servers (some of which date as for back as 1995). Among the most popular are:

FICS: Free Internet Chess Server (freechess.org) Users can register for a free account on the FICS website, or login as guest. Game statistics and ratings are only for registered users. The description of the commands used by FICS can be found at: http://www.freechess.org/Help/AllFiles.html

ICC: Internet Chess Club (chessclub.com) The Internet Chess Club is a paid service and users can't fully access it without becoming a paying member. Information regarding the commands that ICC supports is available at: http://www.chessclub.com/help/help-list

Chess.Net (chess.net) Chess.Net is another commercial service, and can be seen as one of the main competitors of ICC. Information about available commands can be found at: http://www.chess.net/askoweb/tutorial/index.htm

Listing 2.1: Playing a game of chess on FICS using a telnet client

```
fics% seek 2 12
fics% Your seek has been posted with index 89.
(100 player(s) saw the seek.)
fics%

Melegin accepts your seek.

Creating: Melegin ( 993) GuestBWSK (++++) unrated blitz 2 12
{Game 521 (Melegin vs. GuestBWSK) Creating unrated blitz match.}

Game 521 (Melegin vs. GuestBWSK)

       ---------------------------------
    1  | R | N | B | K | Q | B | N | R |     Move # : 1 (White)
       |---+---+---+---+---+---+---+---|
    2  | P | P | P | P | P | P | P | P |
       |---+---+---+---+---+---+---+---|
    3  |   |   |   |   |   |   |   |   |
       |---+---+---+---+---+---+---+---|
    4  |   |   |   |   |   |   |   |   |     Black Clock : 2:00
       |---+---+---+---+---+---+---+---|
    5  |   |   |   |   |   |   |   |   |     White Clock : 2:00
       |---+---+---+---+---+---+---+---|
    6  |   |   |   |   |   |   |   |   |     Black Strength : 39
       |---+---+---+---+---+---+---+---|
    7  | *P| *P| *P| *P| *P| *P| *P| *P|     White Strength : 39
       |---+---+---+---+---+---+---+---|
    8  | *R| *N| *B| *K| *Q| *B| *N| *R|
       ---------------------------------
         h   g   f   e   d   c   b   a
fics%
```

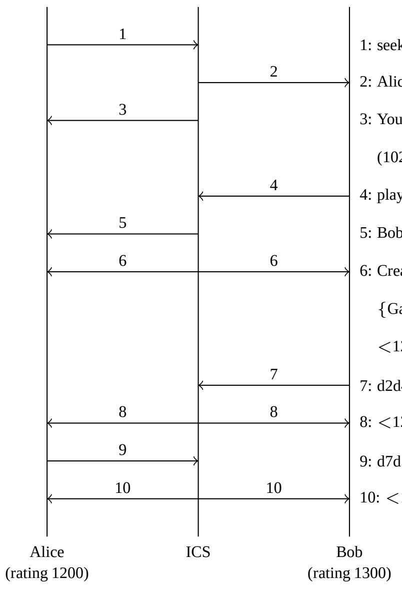
Basic ICS commands

The ICS protocol is not standardized. However, looking at the commands available at the different ICSes, there appears to be (almost) no difference. In case that commands do differ, we follow the syntax described in the FICS help-files, since FICS seems to be the largest free ICS. Below is a list of basic ICS commands, which can be used to find an opponent and play a game (for brevity, some optional parameters have been left out):

- *seek time inc type colour start rating1-rating2*
  Using the seek command, users are able to post a request for a chess game of a specific type. The seek command takes a number of arguments: *inc time* means that each player initially gets *inc* minutes to play the game, and each time they make a move they get *time* additional seconds. The *type* parameter, which can be set to either ’rated’ or ’unrated’, denotes whether or not the result of the game should affect the ratings of the players. The *colour* argument can be set to either ’black’ or ’white’, denoting the colour with which the player executing the seek will play. The value of the *start* argument is set to either ’manual’ or ’auto’, where ’auto’ ensures that the first player that responds will be accepted automatically, and ’manual’ allows the player executing the seek to explicitly accept or decline a responding user. Finally, *rating1-rating2* means that only players who have a rating within this range are allowed to respond (e.g. 1200-1300).
- match *user type time inc colour*

Using the match command, users can challenge a specific player to a game of chess. The match command has several parameters similar to the seek command, with the additional requirement that a user-name must be specified with the *user* argument.

- play *seekid* / *user*
  The play command enables users to respond to a request posted using the seek or match command. The play command takes either a request number (denoted with *seekid*), or a user-name (denoted with *user*) as an argument.
- accept / decline *responseid*
  When a user executes the seek command with *type* set to 'manual', and one or more players respond to it using the play command, the accept command allows the user to accept the response identified by *responseid*. The decline command works the same, except that it will decline the response.
- unseek *seekid*
  The unseek command will cancel an outstanding request. In order to cancel a specific request, unseek should be followed by a number identifying the seek (*seekid*). If no argument is given, all requests are cancelled.
- MOVE
  With this command, users can make moves when a game has started. The command exists out of a string in coordinate notation (e.g. d2d4). Other notations are also available, but coordinate notation seems to be the simplest.



```
1: seek 10 5 rated black auto 1100-1400
2: Alice (1200) seeking 10 5 rated standard [white] ("play 50" to respond)
3: Your seek has been posted with index 50.
   (102 player(s) saw the seek.)
4: play 50
5: Bob accepts your seek.
6: Creating: Bob (1300) Alice (1200) rated standard 10 5
   {Game 85 (Bob vs. Alice) Creating rated standard match.}
   <12> rnbqkbnr pppppppp ——— ——— ——— ——— PPPPPPPP RNBQKBNR …
7: d2d4
8: <12> rnbqkbnr pppppppp ——— ——— —P—- ——— PPP-PPPP RNBQKBNR …
9: d7d5
10: <12> rnbqkbnr ppp-pppp ——— —p—- —P—- ——— PPP-PPPP RNBQKBNR …
```

Figure 2.3: Example of the message flow when Alice and Bob play a game of chess on FICS (with style 12 enabled).

To illustrate the use of these commands, Figure 2.3 shows an example of how communication between two players and an ICS could go. In the example, Alice executes the seek command on the ICS, the ICS notifies Bob, and Bob responds to the seek. The commands used in this example will work on FICS and ICC (and possibly others). The output from the ICS to the player can differ slightly depending on which server you are using. For example, the second message in Figure 2.3 can have arguments in different order on ICC, and ICC also mentions the rating range from the initial seek command.

Chapter 3

Design and implementation of GameCast

In the previous chapter we discussed the ICS protocol, a text-based protocol used for interacting with ICS servers, and the BuddyCast protocol, a peer-to-peer gossiping protocol. In this chapter we introduce GameCast, a system that incorporates ideas from both protocols in order to achieve multi-player gaming functionalities over the Tribler peer-to-peer network. GameCast allows players to explicitly invite one another or invite any player within a certain rating. Additionally, besides the functionalities required for playing a game, GameCast ensures that finished games are distributed throughout the network. Players receiving the game will be able to review the game and attach comments. For now, GameCast will only support turn-based board games, by which we mean that the players of the games will have a pre-determined order in which they make their moves and both the number of players and the frequency at which they move will be relatively small (as opposed to Massively Multi-player Online Role-Playing Games or MMORPGs). To realize the GameCast system, we use a mechanism similar to BuddyCast to allow peer discovery and game distribution within the network. Furthermore, the mechanisms used for game agreement and for playing a game are based on the idea that for each game, its creator/owner is responsible for administrative tasks such as notifying all players that the game has started, distributing game information and ensuring that comments are posted.

The remainder of this chapter is organized as follows. Section 3.1 defines the functional requirements that need to be satisfied by GameCast. In Section 3.2, we define a number of design requirements that need to be met. Section 3.3 elaborates on the design of GameCast. To some extent, Section 3.3 will also go into implementation details, explaining why particular design decisions are made. Finally, Section 3.4 provides a brief introduction to the graphical user interface of Tribler-G.

3.1 Functional requirements

The Tribler peer-to-peer client will be extended with multi-player gaming functionalities. Although, in the future, Tribler-G will have built-in support for a variety of games, for the timing being, we will focus our efforts on online chess. The features of the chess game should be on par with the main features of current online chess games [2] [5] [12]. The following basic functionalities should be available:

Maintain a separate game network At this point we need to make a distinction between the overlay network, that is comprised of all the users of Tribler, and the game network, which is defined as a subset of the overlay network. For the time being, we will let users who are running the modified Tribler client automatically join the game network, i.e., the game network encompasses all the users who are running GameCast. Therefore, no functionality is required for joining or leaving the game network.

Create a new game Any player can create a new game at any time. There are two possible ways to create a new game: (i) invite a friend, (ii) invite a random player of a certain rating. In the case of inviting a specific friend, this friend should be identified using his/her unique identifier or by selecting a player from the high score list. Invites for players of certain ratings are to be distributed within the game network within a distance of h hops. Online peers within this distance will be able to view the invite (see next point).

Besides what (type of) user to play, the player creating the game can also set the color to play with as well as the different timing options. Our chess game will be using Fischer-after clocks, where the timing options are the start time for each of the players clocks and the time with which the clock of a player is incremented when he/she makes a move.

Join a game The player has access to a list of outstanding invites (from other players) that he/she can respond to. This includes both specific invites from friends and from players who request to play against a player of a certain rating. If no open invites exists, the player should create a new game.

Play a game against another user Once the chess game has two players, it will start. If not enough players can be found to play the game, it is discarded.

Players are only allowed to enter valid chess moves into the program. Once the move is entered, the program sends a message to the opponent notifying him/her of the move. The opponent also checks the validity of the move to prevent players from cheating.

Play a game against the computer Events from games played against the computer are completely localized (and are not distributed). This also means that games played against the computer are not taken into consideration in the statistics, and the outcome of the game does not affect the ratings of players.

For most games, there is some kind of open source AI software available, which can play that particular game. These programs, also called engines, can therefore be used to replace a player in the game. Examples of open-source chess engines are Crafty [28], and GNU Chess [6].

Distribute finished games Only once a game is finished, it will be distributed throughout the game network. Peers receiving these games should store them in their databases, along with games that they have played themselves.

Review finished games Users should be able to access the game information collected during the distribution process. This information includes at least: the names of the players, the winner of the game, and the date on which the game was played. Furthermore, users should have the ability to step through all the moves of the chess game.

Users of the game network should also be able to attach text messages to a game. Newly created messages will spread through the network using the same protocol that spreads the games themselves. To limit traffic, constraints need to be put in place on both the total number of messages associated with a game and the maximum size of the messages.

View statistics Users should have access to statistics related to all games that were gathered during the distribution process. Each player should have a rating, indicating their level of skill with the game. Ratings are to be calculated using information that is available in the database.

Furthermore, users should have access to a high score list showing the ratings of known players within the game network. To meet this requirement, information about previously discovered games (stored in the database) is to be used to extrapolate this list.

In order for ratings to be more meaningful to a player (i.e. what the ratings of the other players are), the system should display some type of distribution graph of all known players and their ratings.

Import games from FICS Since there does not yet exist a community of users who frequently play chess on the Tribler network, it is entirely possible that potential players find themselves unable to find a suitable opponent on the game network. To prevent these users from being disappointed and leaving the game network, our application should be able to import additional outstanding invites from the Free Internet Chess Server (FICS) [28]. Since playing rated games on FICS requires players to be registered members, for now Tribler will only support games that are not rated.

3.2 Non-functional requirements

In addition to the functional requirements from the previous section, a solution that enables online gaming over the Tribler peer-to-peer network should also meet a number of design requirements:

Flexible design The software should be designed in such a way that extending the range of possible turn-based board games will be easy. While playing online chess involves only two players, many other games, for example monopoly, can involve a different number of players. The solution should be designed in such a way that no (or at least minimal) changes are necessary to implement a game with more than two players.

Scalability Since the number of peers on the Tribler network has the potential the grow very large, it is important that the online gaming solution scales well. When talking about the scalability of the solution we are most concerned with the bandwidth that the gaming functionalities use. In other words, the amount of traffic that the online gaming solution uses should grow only moderately with the number of peers.

Responsiveness Playing social games over a network requires game events to be propagated timely, which assures that the gaming experience is as fluent as possible. Of course delays are unavoidable since sending a message of the network takes time, even under the best of circumstances. Furthermore, users that utilize the gaming facilities of Tribler may also have a number of downloads running. Running these downloads and maintaining the numerous connections that are related to them, will further impede the responsiveness.

Furthermore, the responsiveness should also be scalable. This means that playing a game over a larger network and/or with a larger amount of participants should have as little effect on the responsiveness as possible. However, for the time being, we assume that the number of participants will be limited.

Note that playing a simple game of chess, will not suffer too much if the responsiveness is lacking, since making a move will usually take some time. However, the design should consider that future games could generate game events more frequently.

Security Generally speaking, peers within peer-to-peer networks implement certain protocols and use these protocols towards some common goal. In such systems, however, untrustworthy peers could try to affect the system in a way that is undesirable. For instance, a peer in the game network could decide to start distributing fake game information in order to influence its own rating or the rating of someone else. In this example, a partial solution is to attach the signatures of all involved players to the message, thereby preventing anyone else from tampering with its content. Another example of undesired behaviour is the possibility of a peer to start attaching spam messages to a game (using the discussion feature), which calls for some sort of spam detection mechanism. Unfortunately, often, security features cannot be added later. However, we consider these problems to be outside the scope of this research assignment, and focus more on delivering a working protocol and the necessary features that enable to start building an online gaming community.

User friendliness The user interface needs to be intuitive—users should be able to use the software without any help or training. When the system gives a user an error, the generated error-messages should be as clear as possible.

3.3 The GameCast protocol

GameCast enables its users to play online multi-player games in a decentralized setting. This is achieved by exploiting a peer-to-peer network, in which peers manage their own local game information. To ensure that each peer has the most recent information, GameCast peers periodically exchange messages containing recent peer and game information. In addition, GameCast provides a way for peers to invite each other and subsequently play a game.

3.3.1 Design overview

GameCast, named similar to BuddyCast and BarterCast, consists of three processes:

Information dissemination The information dissemination process spreads the peer and game information throughout the network.

Game agreement The game agreement mechanism enables one peer to invite another by sending an invitation message.

Game-play The process of multiple peers actually playing a game across the network.

In essence, the former process is a variation of the existing BuddyCast protocol, whereas the latter processes are new additions. Messages related to GameCast are passed through Triblers secure overlay, which enables high-level communication between peers. To easily differentiate between Triblers standard messages and those of GameCast, GameCast messages are prefixed by GC_ (see Table 3.1). The remainder of this section will further elaborate on each of these three processes.

3.3.2 Information dissemination

GameCast uses a gossip protocol to disseminate the game-information across the network. Analogous to BuddyCast, GameCast discovers peers within the network and forms a overlay network. The resulting overlay network, which we call the

Message type	Primary function
GC_GOSSIP	Used in the information dissemination process for the exchange of peer and game information.
GC_ALIVE	Used in the information dissemination process for keeping connections with game buddies and random peers alive.
GC_CMD	Used for sending commands related to game agreement and playing games.

Table 3.1: GameCast message types and their primary functions.

game network, is formed in such a way that peers that frequently play games against each other tend to cluster. The reason for us to promote such a network structure is based on the idea that users who frequently play against each other will likely also be interested in each others games. The game network addresses the first technical challenge from Section 1.5.

This network structure is achieved through the introduction of so-called *game buddies*. Game buddies are basically peers that frequently play games against each other. In order to quantify this frequency, we define the interaction factor of peer i and j as:

$$\min(f_{ij}, c)/c,$$

where f_{ij} denotes the number of games that peer i and j have played against each other, and c is a constant representing the maximum number of games that should be taken into consideration. Peers that have a high interaction factor are called game buddies.

Information is spread within the network through the exchange of GC_GOSSIP messages. When a peer decides to send a GC_GOSSIP message, it needs to select a target peer. The type of the target peer alternates between a game buddy and a random peer within game network. The reason for introducing random peers is to explore new peers (and games) in the network. When choosing a game buddy target, the peer with the highest known interaction factor is chosen, and when choosing a random target, the peer is chosen at random. In either case the target peer is chosen such that it has not been send a message for at least time interval t. In our current implementation we have set t to 5 minutes, since such a short interval allows for a timely information distribution with little bandwidth usage (see also section 4.3.3 for bandwidth usage). The peer receiving the GC_GOSSIP message will first check whether it has already received a GC_GOSSIP message within time interval t. If this is not the case, the receiving peer will update its database with the information found in the message, and subsequently send a GC_GOSSIP message back (unless it has already send a message within time interval t).

Looking at the contents of the GC_GOSSIP message as depicted in Figure 3.1, we see that every message contains information related to its own identification.

Peer info:	**50 Recent games:**	**10 Game buddies:**	**10 Random peers:**
ip, port, name, connectable	game_id, owner_id, winner_permid, moves, players, time_per_move, messages	ip, interaction factor, permid, oversion, connect_time	ip, interaction factor, permid, oversion, connect_time

Figure 3.1: The format of the GC_GOSSIP messages.

Furthermore, the message includes the 50 most recent games the player has created and finished, which should be enough for most players to encompass a one year history (assuming players create and finish about 1 game a week). Finally, the message contains 10 game buddies and 10 random peers, which corresponds to the lists of connected game buddies and random peers further discussed below. Once a GC_GOSSIP message is received, the receiving peer updates its database to reflect the newly discovered information. Among other things, this database provides the necessary information used when determining game buddies.

Since a considerable portion of almost any peer-to-peer network exists of unconnectable peers (i.e., peers that can only be connected to if the peer itself initiates the connection), we keep open connections with a number of game buddies and random peers, and use only those peers to include within a GC_GOSSIP message. This ensures that we only distribute peers that are currently online. Connections are kept open by periodically sending a GC_ALIVE message. To keep track of all these connections, each peer maintains several lists in its memory:

- Unverified connections list: The secure overlay of Tribler allows for protocols such as GameCast to listen to incoming connections, which initially are all added to the unverified connections list. However, by design the secure overlay does not make the distinction between connections coming from e.g. BuddyCast-only peers and peers that have GameCast running. Therefore, connections in this list are not necessarily GameCast peers. Peers are allowed to stay in this list for up to 5 minutes before they are automatically removed. This list is essential in preventing non-GameCast peers from being distributed within the game network.
- Connections list: Once a peer that is in the unverified connections list has sent a GameCast message, we know for sure that GameCast is indeed running on the peer in question. To reflect this information, the peer is moved from the unverified connections list to the connections list. The connections list is further divided into two sub-lists:
  - Unconnectable peers list: A list of connected peers that do not accept incoming connections. These peers are not distributed, but this list it still required to keep connections to connectable peers alive.
  - Connectable peers list: A list of connected peers that accept incoming

connections. Currently this list can contain no more than 20 peers, and can be further divided into connected game buddies, which are the top-10 peers with the highest interaction factor, and connected random peers, which are the remaining peers. For the size limits of the lists we have used the same as those of the original BuddyCast protocol, since they have proven to be effective. Furthermore, while calculating the interaction factor at most $c = 100$ peers are taken into consideration, meaning that we consider peers that have played more than 100 games together to be equally close friends. The peers from these lists are the ones that are included in the GC_GOSSIP messages.

In the current GameCast implementation GC_GOSSIP messages are sent periodically every 10 seconds, which ensures that game and peer information gets discovered quickly after start up. After the peer has been online for over an hour, this is increased to 30 seconds, since at that point a most of the information discovered will already be known.

Bootstrapping

When a Tribler client is first installed, it does not yet have an initial list of GameCast-peers to start sending GC_GOSSIP messages to. In order to provide new clients with such a list, we first need to run a process called bootstrapping. Bootstrapping works through the use of special peers in the network, called superpeers. Superpeers (usually) do not participate in the network actively, i.e., they do not take part in the game agreement process or play games. The contact information required to connect to the superpeers is located in a dedicated file that is included with a Tribler installation. The process works as follows:

- A peer that needs bootstrapping reads the addresses of the superpeers from the hard drive, and randomly picks one. The reason for the peer to run the bootstrapping process can either be because the peer is new to the network or because the peer has not enough contact information to participate in the network successfully.
- The peer sets up a connection (using the secure overlay) and sends a GC_GOSSIP message to the superpeer. The contents of the message is the same as with a normal message. However, the ’recent games’ field may be left empty, because the superpeer has no need for them.
- The superpeer responds with a GC_GOSSIP message of its own. The contents of the message is the same as with a normal message, with the exception that usually both the ’recent games’ and ’game buddies’ field will be empty, because a superpeer does not actually play games on the network.

Discussion feature

In Section 3.1 of this chapter we mentioned that we wanted peers to have a game discussion facility. To meet this requirement, we introduce the discuss command, the first in a range of GameCast commands shown in Table 3.2. When a peer wants to attach a discussion message to game, it sends a discuss command to the owner of the game. The owner of the game is defined as the peer that created the game and send out the initial invites (see also Section 3.3.3). Next, the owner will import the received discuss command into its database and distribute the newly received information through the exchange of future GC_GOSSIP messages.

Command	Function
discuss	When a peers wants to attach a discussion message to game, it sends a discuss to the owner of the game. The owner is responsible for spreading this information through the exchange of future GC_GOSSIP messages.
seek	Used for notifying peers of a random invite.
match	Used for notifying a particular peer of a personal invite.
play	Sent by a peer wanting to accept a random invite.
accept	Sent as positive response to either a play or a match command.
decline	Sent as negative response to either a play or a match command.
unseek	Used for notifying peers that a random invite has been closed.
start	Receipt of this command tells a player that the game has begun.
abort	Used to offer the opponent to abort by agreement.
draw	Used to offer the opponent a draw by agreement.
resign	Used to resign the game. The opponent is declared the winner.
move	Used by players in order to notify each other of their moves.

Table 3.2: The GameCast commands and their functions.

The GameCast commands shown in Table 3.2 are simple text-based strings, existing of the command itself, followed by a number of arguments. Table 3.3 lists the command syntax, which is based on the ICS protocol. For the discuss command, the number of arguments is limited to two arguments identifying the game and the message itself and a third argument containing the contents of the message that is to be attached to the game. When a peers sends a command to another peer, it sends a GC_CMD message with the command-string located in the 'command' field of the message. The complete format of the GC_CMD messages is shown in Figure 3.2. In this figure, the owner field contains information of the owner/creator of the related game. The hops field tells receiving peers how many hops the message needs to be forwarded. The signature field (created using the private key of the sender) is used for commands that need forwarding, and prevents users that pass the message along from tempering with the command.

Command	Arguments
seek	*time, inc, type, colour, start, min_rating, max_rating,* ***gamename***, ***inviteid***, ***gameid***
match	*user, type, time, inc, colour,* ***gamename***, ***inviteid***, ***gameid***
play	*inviteid*
accept	*inviteid*
decline	*inviteid*
unseek	*inviteid*
start	***gameid***, ***players***
abort	***moveno***, ***gameid***
draw	***moveno***, ***gameid***
resign	***moveno***, ***gameid***
move	***moveno***, ***gameid***, ***time_taken***
discuss	***gameid***, ***messageid***, ***content***

Table 3.3: The GameCast command syntax (boldface indicates that the command or argument is not available in ICS and italics indicates a variable name).

Owner info: ip, port, permid	**Hops**	**Signature**	**Command**

Figure 3.2: The format of the GC_CMD messages.

3.3.3 Game agreement

For players to be able to play a game against each other, they need some way to contact each other and agree to play a game. This is done through the process of game agreement, which addresses the second technical challenge from Section 1.5.

Each game has an owner associated with it, which identifies the peer that created the game. The owner is responsible for finding the appropriate number of opponents that are required for the game. For instance, since a game of chess is played by two players, the owner is obligated to create and send one invite. In order to comply with the requirements defined in Section 3.1, we differentiate between two different kinds of invites:

Personal invites Invites that are meant for a specific peer in the network.

Random peer invites Invites that are meant for a peer that has a rating within a certain range.

First, lets start with the process of sending a personal invite, where one peer (the inviter) wants to invite a specific second peer (the invitee) for a game. First, the inviter needs to ensure that there exists a connection between the two peers, and create one if needed. Next, the inviter sends a match command, which notifies the

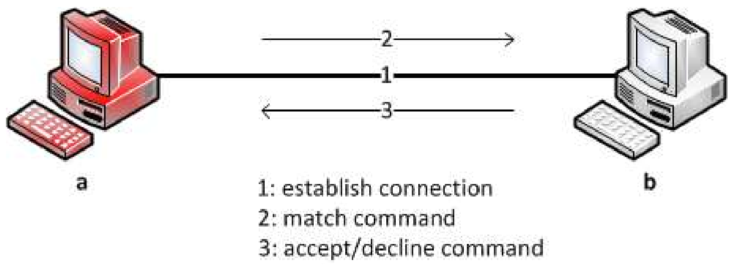


Figure 3.3: Example in which peer a sends a personal invite to peer b.

invitee of the invite. As mentioned in Table 3.3, the match command takes several parameters, some of which also exist in the ICS protocol (see Section 2.2.2 for an explanation of ICS arguments). Regarding the ICS arguments, one thing that is worth pointing out is that the *type* argument should always be set to 'rated', since GameCast does currently not support unrated games. Besides the ICS-supported arguments, GameCast also requires a number of additional arguments needed to identify the type of game (*gamename*), the game itself (*gameid*), and invite (*inviteid*) in question. After the invitee receives the match command, it will respond with a accept/decline command, depending on whether or not it decides to accept the invitation. An example of the process is depicted in Figure 3.3.

Because the invitee can be off-line when the inviter tries to send the match command, but could come back online before the invitation has expired, we should have some kind of mechanism that deals with the temporary unavailability of peers. The solution is simple: when the invitee is off-line (meaning that we cannot set-up a secure overlay connection to it), the inviter will retry every 5 minutes, until either the message has been successfully sent or a certain time constraint has expired (15 minutes, in the current GameCast implementation). This same method is applied to any command that is sent.

Second, we discuss the process of sending random peer invites. Initially, we thought about spreading random peer invites by appending them to all outgoing GC_GOSSIP messages. However, remember that when we previously discussed information dissemination, it was mentioned that sending consecutive GC_GOSSIP messages to the same peer should be at least 4 hours apart (to prevent peers from sending the same message over and over again). Especially for smaller networks, this can cause periods of time in which are will not be any outgoing GC_GOSSIP messages because everybody already received one recently. This would mean that the invite will not be distributed in a timely fashion.

To avoid such issues, a different mechanism has been created for sending random peer invites. Consider a situation where a peer wants to invite a random peer of a certain rating for a game of chess. First, the inviter sends a seek command to all connected game buddies and random peers. The seek command includes many of the same parameters as the match command that we previously discussed (see Table 3.3). Arguments that are worth mentioning are the *user* argument which

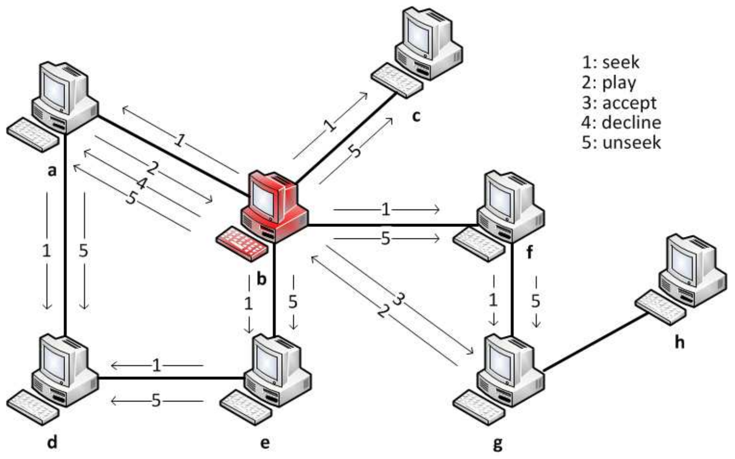


Figure 3.4: Example of the commands sent when peer b creates a random invite. Peers a and g respond to the invite, and the response from g is accepted.

denotes the permID of the invitee, and the *start* argument which should always be set to ’manual’. The peers receiving the seek command will forward the message to their connected game buddies and random peers. How many times the command is forwarded depends on the hops field of the message, which is decremented each time the message is forwarded. The forwarding process stops when the counter runs out. The default start value is currently $hops = 2$, which ensures that several hundred peers can be reached. Next, the peers that decide to accept the invitation will respond to the invite by sending a play command back to the inviter. When the inviter receives the first play command, it accepts the invite, and notifies the peer by sending back an accept command. At this point the inviter declines all further play commands related to the particular invite by sending back decline commands. Now that the invite is no longer valid, the peers in the network need to be notified to avoid inconveniencing users with trying to respond to invites that are no longer valid. To this end the inviter sends a unseek command in the same way that the seek command was send. In case that no peer responds to the invite within a certain time constraint (currently set to 15 minutes), the invitation is automatically marked invalid. An example of the process is depicted in Figure 3.4.

3.3.4 Game-play

Once the game agreement process has been successfully completed, the owner of the game will have collected a list of players that will be playing the game. Next, we need to address the third technical challenge from Section 1.5, which is providing users with the ability to play a game. To this end, the owner first sends a start command to all other players. The start command comes with a complete list of players involved in the game, and only after a peer receives this command will it be aware against which players it will be playing. Each player that receives a start command will now assume that the game has officially started and marks the time.

At this point, the players establish connections between each other. We assume that all players will be constantly online for the duration of the game. Since the game has started, it is time for the players to start making moves. At the start of this chapter it was mentioned that GameCast supports only games that have a predetermined order in which players make their moves. For example, in the case that we are dealing with a game of chess, white will make the first move, black the second, etc. This means that, since the colour of the players has already been established during the game agreement process, each player is now independently able to determine when it is time to make its own move.

When a player makes a move, it sends a move command to all other players of the game. When a new move command arrives, the receiving peer checks whether processing the command should be postponed (see the section below on message ordering). Furthermore, the receiver also checks the validity of the move and if the sender of the command is indeed next to move. Finally, the receiver checks whether the clock of the sender has run out (see the section below on game clocks). If this is the case, the move command will be ignored and the receiver stops participating in the game. If all verifications are successful the peer imports the received move command into its database.

A game is finished once a closing move (i.e., a move after which the game will be finished) has been detected. To this end, each player has knowledge of the rules of the game. Once a new move is received, it uses these rules to determine whether or not the game is finished. If this is indeed the case, it stops participating in the game, notifies the user, and in case the receiving peer is also the owner, it will start distributing the game using the GC_GOSSIP messages discussed previously.

As mentioned in Table 3.3, the first argument of the start command is used to identify the game that the command relates to. Furthermore, the second argument contains a list of all players within the game. Turning our attention to the parameters of the move command, we notice that contrary to other GameCast commands, the syntax of the move does not start with the name of the command, but with the move itself in a notation specific to the kind game that is being played (e.g., for chess a move command could be 'd2d4 20 1'). The second entry of the command contains a number used to identify the game, followed by the move number (see also the section below on message ordering).

Message ordering

So far, we have not been concerned about the order in which messages arrive. This is due to the fact that the secure overlay of Tribler utilizes TCP connections, which ensure that messages between each pair of peers arrive in the same order in which they were send.

However, in these case that we are dealing with a game with more than two players, it is possible that move commands are received in a different order. In order cope with this possibility, each move command has a move number attached to it (*moveno* in Table 3.3). This move number denotes how many moves have already been made, including the current move (so $moveno = 1$ for the first move, $moveno = 2$ for the second, etc.). Since it is also possible that a peer receives a start command out of order, we implicitly assign a move number to the start command with a value of 0.

Now that, for each game, we have numbered the commands in which we are expecting to receive them, we can simply buffer unexpected move numbers until the time that they are ready to be processed. The size of this buffer will be limited to the number of players within the game, since eventually the game will be blocked by the peer that owns the buffer.

Regarding TCP connections of the secure overlay, we should also note that in the event that a connection gets dropped after a message is sent, but before it is received by the target peer (depending on the network latency), the message will be lost. However, we assume that these events will be rare, and for now simply make the message handling functions deal with them (e.g., a game could time-out due to a lost move command).

Game clocks

Many competitive games such as chess, Go, and Scrabble utilize game clocks in order to keep track of the total time that each player has taken so far. Each player has its own clock, and the clock of a player is only running when it is his/her turn to make a move. As soon as a move has been made, that clock will be paused and the clock of the opponent will resume. Several different types of games clocks exists, for instance Fischer-after clocks, Fischer-before clocks, hourglass clocks, and simple delay clocks.

Like the Free Internet Chess Server (FICS), GameCast uses Fischer-after clocks in order to keep track of time. Fischer-after clocks count down from a specified number of minutes, and after each move a player makes a specified amount of time is added to his/her clock. Furthermore, like on FICS, the game clocks of each of the players will not get incremented after the first move and the clocks will only start running after both players have made their first moves.

In GameCast, the Fischer-after clock takes two parameters, namely the start time in minutes to which the clock of each player gets set, and the time in seconds with which the clock of a player is incremented when a move is made (*time* and *inc* in Table 3.3). By recording the time at which the various move commands related to a game were received, each peer independently keeps track of the game clocks. Move commands received after the clock of a player has run out are ignored.

The difference between the time at which a player makes a move and the moment the other player(s) receive a move (i.e., latency) can result in the game clock not being properly synchronized. To prevent this, the GameCast protocol includes the time taken for the move in question. Next, players receiving the move command will calculate the difference between the time taken according to the sender, and compare it to the time taken from the receivers perspective. Next, receiving peers will correct the opponents clock with the value that was previously calculated. To limit malicious use, to maximum allowed correction is set to 1 second. We have added this mechanism after the first round of user testing (see Chapter 5, in particular Section 5.2.2)

Special commands

Until now, we only discussed making normal moves, but there are also a number of special game-play related commands: abort, draw, and resign (see also Tables 3.2 and 3.3). Using these commands player may choose to end the game without making a closing move.

The resign command allows a player to resign the game, after which the opponent will be declared the winner. The resign command works much like a regular move command, with the exception the a player does not have to wait his/her turn before executing the command. The draw command, however, requires that all players agree that the game is to be drawn. In order to reach an agreement, GameCast requires that all players send an draw command with the same *moveno*, after which the game is considered drawn. The abort command works just like the draw command, with the exception that if any player has yet to make a single move, the game can be aborted by any of the players without reaching an agreement.

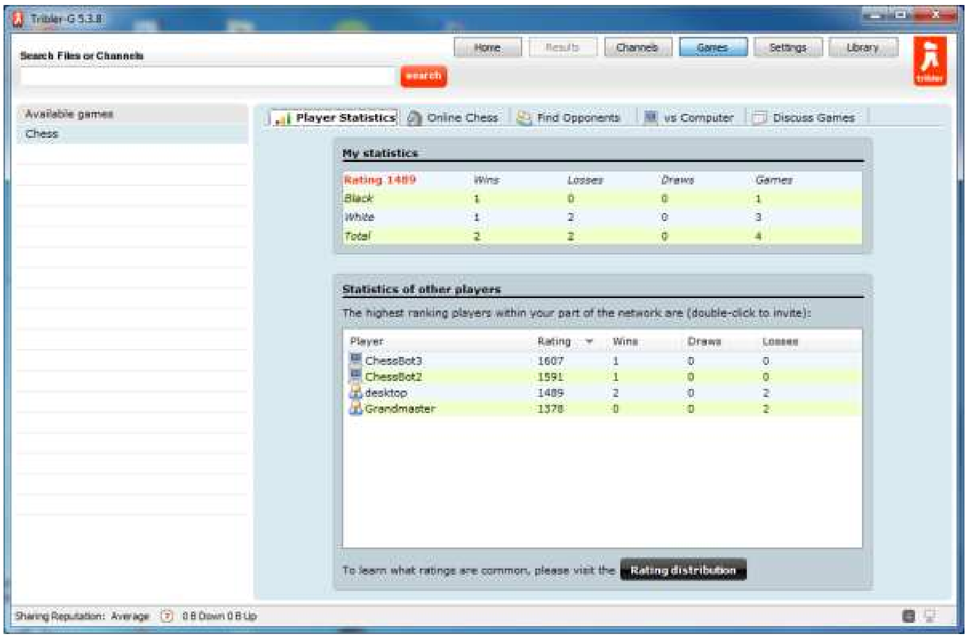
Game ratings

Currently GameCast supports the Glicko [25] rating system, which is a system for rating players of a two-person game. The reason for adopting the Glicko system is partly due to that fact that the FICS uses the same system, and therefore gamers that also play on FICS will be more comfortable with understanding the rating system.

In the Glicko system, each player has a rating and a rating deviation (RD). The RD of a player is used to quantify to which extend the rating of the player should be trusted. In case a player has a high RD, the player may not have competed in many games, while a low RD indicates that the player competes frequently.

The rating of a player can only change as a result of a completed game. How much a rating changes depend on both the rating and RD of the player itself, as well as that of the opponent. The rating of a player with a high RD will change more than a player with a low RD. Furthermore, over time, when a player builds up a more established rating (and thus a higher RD), additional win/losses will have a less substantial affect on his/her rating. Besides the RD of the player itself, the RD of the opponent is also taken into consideration, although to a smaller extent: when a opponent has a high RD, the change of the rating of the player will be smaller than it would be if the opponent had a low RD.

The RD of a player can change both as a result of a completed game and also the amount of time that passes when the player is not playing. Competing in games will always increase the RD of a player, while not competing will always decrease the RD of a player.



My statistics

Rating 1489	Wins	Losses	Draws	Games
Black	1	0	0	1
White	1	2	0	3
Total	2	2	0	4

Statistics of other players

The highest ranking players within your part of the network are (double-click to invite):

Player	Rating	Wins	Draws	Losses
ChessBot3	1607	1	0	0
ChessBot2	1591	1	0	0
desktop	1489	2	0	2
Grandmaster	1378	0	0	2

To learn what ratings are common, please visit the

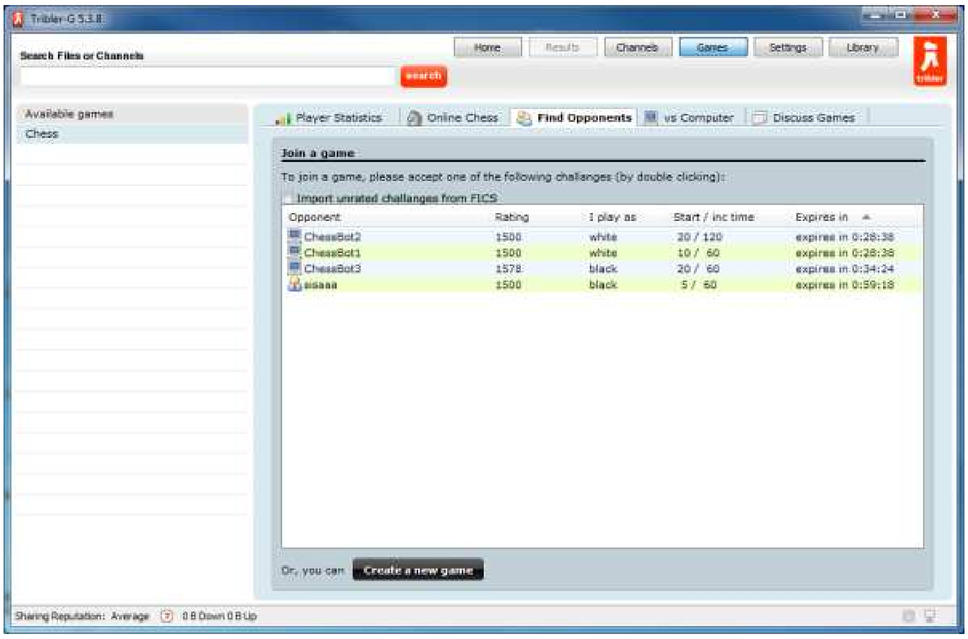
Figure 3.5: Screenshot of the Tribler-G main window while viewing GameCast player statistics.

When calculating ratings using Glicko, a collection of games that were played within a ”rating period” are treated as if they have occurred simultaneously. This length of rating period can be set to anything, but we use a rating period of just one minute. This is due to the fact that the FICS uses the same value, and we would like our rating system to be as close to theirs as possible. Note that in order to know which games have occurred in which rating period, we need to know the time at which games are played. Therefore, we included the age of a game when sending GC_GOSSIP messages.

There is currently one problem with our rating system that is worth mentioning. The problem occurs when a player joins the game network for the first time, or has been off-line for a long period of time. When this happens, the player in question will lack recent game information, and may therefore not have a full history of each gamer (remember that our gossip algorithm only distributes the 50 most recent games of a player). This leads to different ratings for certain players.

3.4 The Tribler-G graphical user interface

In the previous section we elaborated on the GameCast protocol, which is essentially the part of the program that works in the background. Additionally, we also need a graphical user interface (GUI) that enables the user to respond to invites, send invites, make game moves, etc. To meet this demand, we have extended the current Tribler GUI with features required to control GameCast. This section will



Join a game

To join a game, please accept one of the following challanges (by double clicking):

Import unrated challanges from FICS

Opponent	Rating	I play as	Start / inc time	Expires in
ChessBot2	1500	white	20 / 120	expires in 0:28:38
ChessBot1	1500	white	10 / 60	expires in 0:28:38
ChessBot3	1578	black	20 / 60	expires in 0:34:24
aaaaa	1500	black	5 / 60	expires in 0:59:18

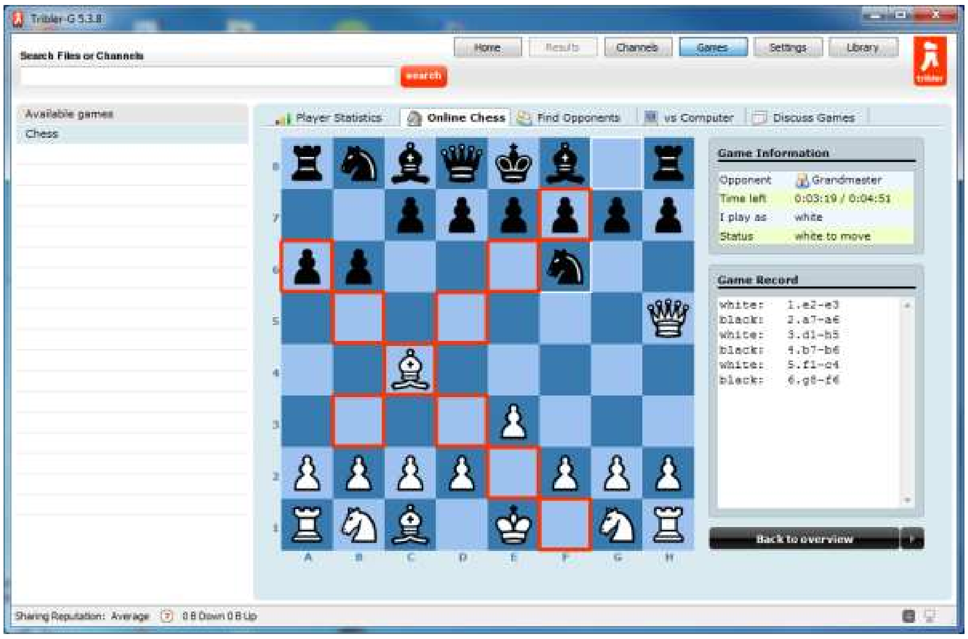
Figure 3.6: Screenshot of Tribler-G while displaying available invites.

briefly discuss the GameCast GUI, and give an overview of the features it offers. How users asses this GUI is discussed in Chapter 5.

Figure 3.5 shows a screenshot of the extended Tribler GUI. The top bar of the screen allows users to go through the different panels within Tribler. This bar is included in any standard Tribler installation, and the GameCast extension merely adds an additional *Games* button to the bar. When a user clicks the *Games* button, the GameCast GUI is displayed, which encompasses the bottom two panels displayed in Figure 3.5. The left panel presents an overview of the currently implemented games (to date, the only game available is chess). The right panel allows users to play the game that is selected in the left panel.

In the bottom right panel, there are the following tabs for a chess game: *Player Statistics*, *Online Chess*, *Find Opponents*, *vs Computer*, and *Discuss Games*. The *Player Statistics* tab, selected in Figure 3.5, enables the user to access statistics related to past games. These statistics do not only show the rating of the current user and how many games were won/lost, but also give a more global picture of ratings of other known users in the form of a histogram. In addition, the user can use the *Highscores* button to switch between the histogram and a list of the 25 best players.

The *Find Opponents* tab allows the user to create and accept invites. The list in Figure 3.6 shows the invites that a user can respond to. By default, invites in this list are collected from GameCast only. However, by checking the ”Import unrated challenges from FICS” option, available invites from the FICS are shown as well. Note that currently only unrated invites are supported. This is due to the fact that



Game Information

Opponent	Grandmaster
Time left	0:03:19 / 0:04:51
I play as	white
Status	white to move

Game Record

```
white:   1.e2-e3
black:   2.a7-a6
white:   3.d1-h5
black:   4.b7-b6
white:   5.f1-c4
black:   6.g8-f6
```

Figure 3.7: Screenshot of Tribler-G while playing an online game of chess.

FICS requires users to be a registered member of FICS in order to play rated games. In the current implementation, however, users are automatically logged in as guest. New GameCast invites can be created by clicking the ”Create a new game” button shown below the list of invites. Creating new FICS invites is not supported at this time.

Once an opponent has been found, the user should select the *Online Chess* tab, where he/she is presented with a list of all current active games. The games in this list can be resumed by double-clicking, at which point the user arrives at a screen similar to Figure 3.7.

The *vs Computer* tab allows the user to play a game against the computer using the Crafty chess-engine. Games played against the computer are not saved and have no time limit. Furthermore, since our GameCast-enabled Tribler client is more geared towards online gameplay, options like setting which colour to play with and which level of difficulty the chess-engine should play with are also not available.

Finally, the *Discuss Games* tab allows users to discuss finished games. When opening the tab, the user is presented with a list of all known games (that have been gathered during the information dissemination process). Since the list can grow rather large, there is also a search option which allows the user to display only games related to a particular player. Double-clicking an entry from the list of games, allows access to the review panel, in which a user can visually browse through all the moves that have been made during the game. Furthermore, the user is provided the opportunity to share comments/insights pertaining to the game, as well as read messages from fellow players. Currently, the GUI does not allow the user to attach a message to another message (i.e., nesting), but such functionality could be added relatively easily.

Chapter 4

Evaluation of GameCast

After having discussed the design and implementation of GameCast in the previous chapter, in this chapter we will evaluate its performance. To this end we have created GameTest, an emulation environment that allows a large network of GameCast peers to be emulated by starting and stopping Tribler-G instances. During the emulation of the GameCast network, GameTest will gather information related to a number of important statistics.

Section 4.1 describes the hardware configuration of the DAS-4 supercomputer on which we performed our emulation. We elaborate on the GameTest emulation environment in Section 4.2. Next, in Section 4.3, we emulate a network of GameCast peers and evaluate the performance of the system.

4.1 The DAS-4

The experimental results that are presented in this chapter were acquired using the fourth generation Distributed ASCI Supercomputer or DAS-4 [3]. DAS-4 is a six-cluster distributed system, of which the clusters are located at the following institutes/organizations: VU University (74 nodes), Leiden University (16 nodes), University of Amsterdam (16 nodes), Delft University of Technology (32 nodes), the MultimediaN Consortium (36 nodes), and the Netherlands Institute for Radio Astronomy (23 nodes). Each cluster has one head node, which is used as a file-server, and the remaining nodes are computation nodes.

The DAS-4 clusters communicate with each other through the use of dedicated 10 Gbps light-paths. Furthermore, each cluster has a 1 Gbps connection to the Internet through its local university. Within each cluster, the nodes are connected locally through 1 Gbps Ethernet for the normal nodes, and 10 Gbps Ethernet for the head node.

DAS-4 runs the CentOS Linux operating system. The nodes of each of the clusters have the following or better configuration: a dual quad-core 2.4 GHz processor, 24 GB memory, 1 TB of local storage, and 18 TB of storage made available by the head node.

4.2 Emulation environment

In this section we present GameTest, a system used to emulate a network of game-playing peers on the DAS-4 supercomputer, thereby addressing the fourth technical challenge from Section 1.5. The emulation is made possible by running Tribler-G instances on nodes of the DAS-4. In order to control what is happening on the emulated network, GameTest requires an input scenario describing which peers join/leave the network. GameTest processes the input scenario one line at a time, and subsequently creates or stops Tribler-G instances. How peers behave during the time that they are online is decided based on uniform random probabilities.

GameTest should be started using the Sun Grid Engine (SGE), the current node reservation system for DAS-4. SGE enables the reservation of a specific number of nodes for the duration of a program run. While SGE comes with all the necessary commands needed to perform node reservation, GameTest currently relies on Prun, an alternative user interface for SGE. We decided in favour of Prun because it is generally much easier to use.

Crowded [34] is an emulation environment very similar in architecture to GameTest. Crowded has been developed to evaluate a swarm discovery protocol on DAS-2. Much like GameTest, the system works by starting and stopping Tribler instances on nodes of the DAS-2. Crowded differentiates itself from GameTest by the input data it requires and the statistics it collects during the emulation.

Despite the design similarities between GameTest and Crowded, we decided to create GameTest from scratch, because we estimate that the overlap between the two systems is only about 250 lines of code. Based on this estimate, we believe that adapting Crowded to fit our needs would have taken as much time as creating the system from scratch. Additionally, creating GameTest from scratch prevents any issues that might have occurred because of unavailable or outdated software dependencies.

4.2.1 Architecture of GameTest

Figure 4.1 shows the architecture of the GameTest emulation environment, which consists out of a single *manager* process and one or more *worker* processes. The manager reads an input scenario file from disk, and subsequently forwards the commands listed in the input scenario to the appropriate worker using XML-RPC (XML Remote Procedure Calls). The workers take these commands and send them to the targeted Tribler-G instance, which will execute the command. The workers are able to communicate with the Tribler-G instances on the same node using pipes. When starting the system, one of the nodes is chosen as the master node, which runs both the manager and the worker, and the remaining nodes are slave nodes, which only run the worker.

On which node a particular Tribler-G instance should run is decided by the manager process and is completely transparent to the user. When the GameTest manager process encounters a command to start a new peer, it forwards the command

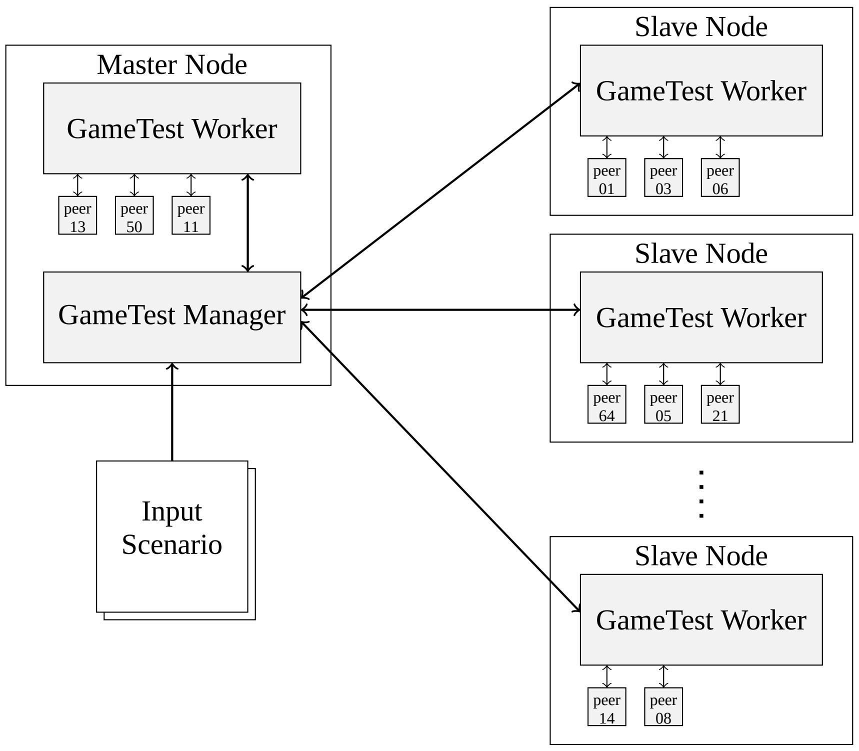


Figure 4.1: The architecture of the GameTest testing environment used on the DAS-4 nodes, with a number of Tribler-G instances running.

to the worker process that has the fewest Tribler-G instances running. The worker process receiving the command will create a new Tribler-G instance, and subsequently signal the manager that the instance has started. Once the manager process has received this signal it will remember on which node that particular instance is running. This ensures that the load is distributed equally amongst the available slave nodes.

4.2.2 Input scenario

Which peers join or leave the emulated network at what time is described in the input scenario. The input scenario is stored as a text file with a number of commands that will be executed (for wait commands) / forwarded (for other commands) sequentially and in the order in which they are specified in the file by the GameTest manager. Listing 4.1 shows an example of what an input scenario could look like.

Currently, the following three input scenario commands are available:

- `shell` c, p: This command is used to execute shell command c on the slave node that holds Tribler-G instance p. We only use this command for monitoring system statistics (e.g., CPU load monitoring).

- `exec` c, p: This command tells the Tribler-G instance p to execute the command c (see below for a list of available commands).
- `wait` s: Issuing the wait command will cause the GameTest manager to pause execution of the input scenario for the duration of s seconds.

The exec command explained above currently implements the following commands:

- `start` i: This command will ensure that a new Tribler-G instance with name p is started on one of the nodes. The name should be a unique string, since it is used in future commands to identify the instance. The i parameter denotes the maximum number of simultaneous games that the peer is allowed to participate in (see the next section). Once the command is executed, a working directory is created named after the instance it belongs to. The working directory is used to store various state information, such as settings and database files. If a working directory already exists, the state information that it contains will be used to create the Tribler-G instance.
- `starts`: This command is the same as the previous command, with the exception that the new Tribler-G instance will be started in superpeer mode (see the previous chapter for more information on superpeers). The superpeer mode does not support playing games, and therefore this command lacks additional arguments.
- `stop`: Once an instance with name p is started using the start command, it can be stopped at any time using the stop command. When a Tribler-G instance is stopped, the GameTest manager process will cache the particular slot (the hostname of the node and TCP port number) that the instance held. In case of a future re-start of the same Tribler-G instance, the GameTest manager will attempt to assign the instance the same slot that it held before. However, there are no hard guarantees that the instance will receive the same slot.

```
exec starts peer1
exec start 2 peer2
wait 1
exec start 2 peer3
wait 1
exec pause peer2
exec start 1 peer4
wait 298
exec stop peer4
exec resume peer2
wait 25
exec stop peer2
exec stop peer3
exec stop peer1
```

Listing 4.1: Example input scenario

4.2.3 Peer behaviour

The input scenario discussed in the previous section tells the emulation environment when and for how long a peer should be online, but it does not describe how a peer should behave during that time. Instead, the behaviour of peers is determined by the peers themselves during runtime, based on random probabilities. Currently, the only behaviour related argument that can be passed to a peer is the maximum number of games a peer is allowed to play simultaneously (see the start command from the previous section).

Peer behaviour is modelled as follows. We assume that actions are executed in bursts. The actions within each burst consist of a wait period that represents the time that a user needs to take a certain action, followed by the execution of the command. Between bursts, there are larger wait periods, representing the user leaving the application and doing some other work. In order to avoid too many games from timing out because of this longer wait period, games are played within a single burst.

The code that governs the behaviour of a peer consists of a single continuous loop that does the following. First, it selects what type of task is to be executed. Next, based on the type of task selected earlier, the code decides on the time to wait, and waits for the selected amount of time. Finally, the peer decides on the actual task and executes it (see Figure 4.2).

```
function EMULATEPEER
    while True do
        $type \leftarrow$ selectTaskType()
        $time \leftarrow$ selectTaskTime($type$)
        sleep($time$)
        $task \leftarrow$ selectTask($type$)
        task()
    end while
end function
```

Figure 4.2: The function of defining peer behaviour.

The next obvious question is how do we select what type of task that should be executed. This decision is made based on pre-determined probabilities. The pseudo-code listed in Figure 4.3 demonstrates how the exact process works. In the pseudo-code the variables G_O, G_C, and I_O represent the list of open games (i.e., unfinished games that the peer is currently involved in), closed games (i.e., finished games), and open invites (i.e., invites from other peers that the peer is eligible to respond to) respectively. These lists are not modified in the listed code, and are maintained by code outside this function. The constants P_D, P_N, P_M, P_P, and P_S represent the probabilities that are used to decide on the next task type. Note that, depending on the values of G_O, G_C, and I_O, we are mostly dealing with relative probabilities. Furthermore, G_{max} denotes the maximum number of simultaneous

```
 1: function SELECTTASKTYPE
 2:     $type \leftarrow$ noops                    ▷ the noops task acts as a short no-operation
 3:     if $|G_C| > 0$ and random() $< P_D$ then
 4:         $type \leftarrow$ discuss
 5:     else if $|G_O| = 0$ and random() $< P_N$ then
 6:         $type \leftarrow$ noopb                ▷ the noopb task acts as a long no-operation
 7:     else if $|G_O| > 0$ and random() $< P_M$ then
 8:         $type \leftarrow$ move
 9:     else if $|G_O| < G_{max}$ then
10:         if $|I_O| > 0$ and random() $< P_P$ then
11:             $type \leftarrow$ play/accept
12:         else
13:             if  random() $< P_S$ then
14:                 $type \leftarrow$ seek
15:             else
16:                 $type \leftarrow$ match
17:             end if
18:         end if
19:     end if
20:     return $type$
21: end function
```

Figure 4.3: The function of selecting which type of task is to be executed next.

games that can be played by the peer in question (this value is set by the peer's start command listed in the input scenario). Table 4.1 shows miscellaneous functions used in the pseudo-code listings. Also, line 11 mentions the task type 'play/accept', which denotes a task that either responds to a seek with a play, or to a match with an accept (see also the previous chapter). Finally, notice that the code in Figure 4.3 does not cover the entire range of all GameCast commands. This is due to the fact we reckon the resign, draw, and abort commands among possible moves. Also, the decline command is missing, because this command can currently not be explicitly sent using the Tribler-G user-interface.

After having determined the type of task that is to be executed, we need to decide on the actual task and the interval during which to wait prior to execution. As shown in Figure 4.2, the `selectTaskTime` is used to determine the wait time. The `selectTaskTime` works by returning a uniform randomly selected integer in a range depending on the type of task that is to be executed next. Once we have waited for the proper amount of time, we need to decide on the actual task. In the pseudo-code listed in Figure 4.4, the task is determined by randomly selecting an invite/game from the lists G_O, G_C, and I_O.

In the experiments presented in Section 4.3, we have used the values for the parameters listed in Table 4.2. The first two parameters listed for each task type represent the minimum and maximum time that we estimate the task should take.

```
function SELECTTASK($type$)
    if $type$ == move then
        $game \leftarrow$ randomChoice($G_O$)
        $move \leftarrow$ decideNextMove($game$)          ▷ based on pre-defined games
        $task \leftarrow$ createTask($type$, $game$, $move$)
    else if $type$ == play/accept then
        $invite \leftarrow$ randomChoice($I_O$)
        $task \leftarrow$ createTask($type$, $invite$)
    else if $type$ == seek then
        $invite \leftarrow$ randomly generate an invite
        $task \leftarrow$ createTask($type$, $invite$)
    else if $type$ == match then
        $invite \leftarrow$ randomly generate an invite
        $task \leftarrow$ createTask($type$, $invite$)
    else if $type$ == discuss then
        $game \leftarrow$ randomChoice($G_C$)
        $message \leftarrow$ randomly generate a message
        $task \leftarrow$ createTask($type$, $message$)
    else if $type$ == noop then
        $task \leftarrow$ createTask($type$)
    end if
    return $task$
end function
```

Figure 4.4: The function for selecting the next task.

These values correspond to the intervals from which the `selectTaskTime` function selects its return-values. For instance, we estimate that attaching a discussion command to a game should take between $T_{D,min} = 60$ and $T_{D,max} = 180$ seconds. This time denotes the time that it takes for a user to select a specific game, open up the review panel, type in a message, and press send. This idea also applies to the remaining tasks. The `noops` and `noopb` are different from the other tasks in that they do not actually execute anything. Instead, `noopb` represents the time between bursts, while `noops` represent a short period of inactivity within a bursts. Finally, unless otherwise specified, we have set $G_{max} = 1$, meaning that emulated peers can only play one game simultaneously (like on the Free Internet Chess Server).

Unfortunately, we lack actual data to base these values on, but we believe to have chosen reasonable values, based on previous works in human-computer interaction ([29], [36]). The same problem arises when determining the probabilities for each of the task types. Therefore, further research into this problem is required.

Function	**Description**
random()	Return a uniform randomly selected floating point number in the range $[0.0, 1.0)$.
randomChoice(L)	Return a uniform randomly element from the non-empty list L.
validMoves(G)	Return a lists of a valid chess games that can be made for game G.
createTask($c, \dots$)	Return a task based on the command type c and any additional parameters.

Table 4.1: Functions used while determining peer behaviour.

Task type	**Parameters**		
noops	$T_{O,min} = 20$	$T_{O,max} = 60$	
noopb	$T_{N,min} = 300$	$T_{N,max} = 600$	$P_N = 0.20$
discuss	$T_{D,min} = 60$	$T_{D,max} = 180$	$P_D = 0.01$
move	$T_{M,min} = 1$	$T_{M,max} = 10$	$P_M = 0.99$
play/accept	$T_{P,min} = 5$	$T_{P,max} = 30$	$P_P = 0.75$
seek	$T_{S,min} = 20$	$T_{S,max} = 60$	$P_S = 0.75$
match	$T_{A,min} = 20$	$T_{A,max} = 60$	

Table 4.2: The parameters and the values used during the experiments.

4.2.4 Logging features

In order to determine what is happening on the emulated network, the GameTest emulation environment produces a number of log files. After the emulation ends, these files are analysed by a post-processing script, which provides the results presented in Section 4.3.

There are currently two log files created for each Tribler-G instance. First, the GameCast log, which is primarily used to log at what times GameCast command messages are sent or received. The GameCast log is stored is a file called 'gamecast.log', which can be found in the working directory of the Tribler-G instance. Listing 4.2 shows an example of (part of) a GameCast log. The first column represents the time in seconds since epoch (on Linux systems epoch is January 1st 1970 at 0:00). We choose this time representation because it will be easier for the post-processing script to process. The second column denotes what type of event we are dealing with. All possible types of events and their description can be found in Table 4.3. The third column shows the remote peer that is involved in the event (if any). And finally, the fourth column lists any values that are important for debugging and analysis. Note that the final column has a particular format, namely a variable name followed by an equality sign and a value, and delimited by a semicolon. This format provides expressability and extensibility, but makes parsing more complex. However, we believe that the increase in ease of post-processing is worth the additional complexity.

Listing 4.2: Example GameCast log-file.

```
...
1315330672.47  DB_STATS                                   gc_members = 1 ; uptime = 0 ; ...
1315330707.84  RECV_MSG  Rfrx76KL0z (10.141.0.14:6000)  msg_type = GC_CMD (seek) ; payload = ...
1315330707.86  SEND_MSG  br6zkoxA7P (10.141.0.10:6002)  msg_type = GC_CMD (seek) ; payload = ...
1315330707.86  SEND_MSG  ZUSCsuPXtr (10.141.0.10:6000)  msg_type = GC_CMD (seek) ; payload = ...
1315330707.87  RECV_MSG  br6zkoxA7P (10.141.0.10:6002)  msg_type = GC_CMD (seek) ; payload = ...
1315330707.87  RECV_MSG  ZUSCsuPXtr (10.141.0.10:6000)  msg_type = GC_CMD (seek) ; payload = ...
1315330715.48  SEND_MSG  Rfrx76KL0z (10.141.0.14:6000)  msg_type = GC_CMD (seek) ; payload = ...
1315330715.49  DB_STATS                                   gc_members = 3 ; uptime = 43 ; ...
1315330715.49  SEND_MSG  br6zkoxA7P (10.141.0.10:6002)  msg_type = GC_CMD (seek) ; payload = ...
1315330715.50  SEND_MSG  ZUSCsuPXtr (10.141.0.10:6000)  msg_type = GC_CMD (seek) ; payload = ...
1315330732.84  RECV_MSG  br6zkoxA7P (10.141.0.10:6002)  msg_type = GC_CMD (seek) ; payload = ...
1315330732.85  SEND_MSG  Rfrx76KL0z (10.141.0.14:6000)  msg_type = GC_CMD (seek) ; payload = ...
...
```

Event type	**Description**
SEND_MSG	Either a GameCast command or a GameCast gossip message has been sent.
RECV_MSG	Either a GameCast command or a GameCast gossip message has been received.
POSTPONE[1]	A GameCast move command has been received, but is not yet ready to be processed.
GAMEDONE[1]	A GameCast game has just been marked as finished due to an abort, resign, draw or closing move.
CLOCK_COR[1]	An opponent's clock has just been corrected by the specified amount of time.
DB_STATS[1]	Periodic executed event that lists several database statistics.
CONN_TRY[2]	A connection is being established.
CONN_VER[2]	A remote peer has been verified to be a GameCast peer.
CONN_ADD[2]	A connection to a remote peer has just been opened.
CONN_DEL[2]	A connection to a remote peer has just been closed.
GC_STATE[2]	Periodic executed event that lists several gossip statistics.

Table 4.3: Possible event types shown by the GameCast log-files.

The second log file that GameCast produces is related to the gossiping protocol, and gives an overview of the peers that are connected and when messages are sent or received. The GameCast gossip log is called 'gamecastgossip.log', and can also be found in the working directory. An example of a part of a GameCast gossip log is shown in Listing 4.3.

[1]Only shown in the GameCast log.
[2]Only shown in the GameCast gossip log.

Listing 4.3: Example GameCast gossip log-file.

```
...
1315330685.43  GC_STATE                                     Round = 2 ; nBr = 1 ; nBs = 1 ; nCc = 1 ...
1315330685.44  CONN_TRY  Rfrx76KL0z (10.141.0.14:6000)
1315330685.49  CONN_ADD  Rfrx76KL0z (10.141.0.14:6000)
1315330685.50  SEND_MSG  Rfrx76KL0z (10.141.0.14:6000)  msg_type = GC_GOSSIP (active) ; payload ...
1315330685.56  CONN_VER  Rfrx76KL0z (10.141.0.14:6000)
1315330685.57  RECV_MSG  Rfrx76KL0z (10.141.0.14:6000)  msg_type = GC_GOSSIP (active) ; payload ...
1315330695.43  GC_STATE                                     Round = 3 ; nBr = 2 ; nBs = 2 ; nCc = 0 ...
1315330700.83  CONN_ADD  br6zkoxA7P (10.141.0.10:6002)
1315330700.87  CONN_VER  br6zkoxA7P (10.141.0.10:6002)
1315330700.87  RECV_MSG  br6zkoxA7P (10.141.0.10:6002)  msg_type = GC_GOSSIP (passive) ; payload ...
1315330700.89  SEND_MSG  br6zkoxA7P (10.141.0.10:6002)  msg_type = GC_GOSSIP (passive) ; payload ...
...
```

4.3 GameCast evaluation

Using the GameTest emulation environment discussed earlier, we have performed an experiment designed to evaluate the GameCast protocol. During the experiment, 7,618 unique peers have connected to the emulated network over time, while the maximum size of the network was 446 concurrent peers. The experiment was conducted using 20 DAS-4 nodes. Using less than 20 nodes resulted in higher message transmission times, due to increased system load.

4.3.1 Performance metrics

In order to asses how GameCast performs (the fifth challenge from Section 1.5), we monitor twelve metrics, which can be categorized as follows: information dissemination metrics, game agreement metrics, game-play metrics, and bandwidth usage metrics.

Information dissemination metrics

Metrics 1A and 1B: The knowledge of peers about each peer (A) or game (B) that was added to the network since ***the start of the emulation****.*
These two metrics quantify the performance of the information dissemination process in terms of spreading all past game/peer information. To obtain metric 1A, for each online peer, we divide the number of gaming peers that are known at a particular time by the number of gaming peers that are in existence (both off-line and online). To obtain metric 1B, for each online peer, we divide the number of games that are known at a particular time by the total number of games in existence. Finally, the results from all peers are averaged.

Metrics 2A and 2B: The knowledge of peers about each peer (A) or game (B) that was added to the network since ***joining the system****.*
These metrics quantify the performance of the information dissemination process in terms of spreading recent game/peer information. To obtain metric 2A, for each online peer, we divide the number of gaming peers in its database that are no older

than the peer itself by the number of gaming peers that were added to the network since the peer joined the network. To obtain metric 2B, for each online peer, we divide the number of finished games in its database that are no older than the peer itself by the total number of games that were finished since the peer joined the network. Finally, the results from all peers are averaged.

Metrics 3A and 3B: The number of peers that are aware, over short time intervals, of the finished games or active peers during each interval.
We would like to know how fast game information is spread through the network. To capture this characteristic, we will keep track of the number of peers (off-line and on-line) that are aware of a particular game over time (metric 3A). Additionally, we will do the same for peer information, by keeping track of the number of peers that are aware of a certain other peer (metric 3B).

Game agreement metrics

Metrics 4 and 5: The time it takes for a random peer invite to spread throughout the network and how many peers are covered.
We would like to know how long it takes for a random peer invite to travel the distance of two hops, measured from the time of sending, until the time of reaching all nodes within 2 hops (metric 4). Furthermore, we would like to know how many peers the invite reaches (metric 5).

Game-play metrics

Metric 6: The time that it takes to set up a game.
This metric measures the time it takes to set up a game, starting from the point that the final invitee responds to a invite, until the point that all players of a game have received a start command for the game.

Metric 7: The time that peers need to add to their clocks in order to remain synchronized.
This metric measures how much each peer needs to correct the clocks of its opponents. If a large number clock corrections reach the maximum correction parameter, the game clocks will often not be properly synchronized.

Bandwidth usage metrics

Metric 8: The bandwidth usage of GameCast without any gaming activities.
To obtain this metric, we measure the bandwidth used by the Tribler-G instances without taking the gaming activities into account. In order to achieve this, we log the bandwidth usage of the information dissemination process separately.

Metric 9: The bandwidth usage of GameCast when peers are playing games over the network.

To obtain this metric, the bandwidth that peers use when they are involved in gaming activities (e.g., sending invites, or making moves) is measured.

4.3.2 Scenario generation

In Section 4.2.2 we discussed the notion of the input scenario and what types of commands they can be built out of. However, until now we have not concerned ourselves with how exactly we can create an input scenario. In this section we will elaborate on how to deal with the issue of scenario generation.

In order to create an input scenario, we need to decide which peer is started at what time and for how long it remains running. To solve this problem, we have chosen to base our approach on a basic queueing model, in which peers arrive in the network one by one, according to a Poisson process. Since the arrival times are modelled with a Poisson process, the inter-arrival times will follow an exponential distribution, for which the cumulative distribution function (CDF) is:

$$F(x) = 1 - e^{-\lambda x}.$$

Furthermore, we assume that the intervals during which the peers are running, are uniformly distributed and independent of the inter-arrival times.

An appropriate value for λ in the CDF can be determined by applying Little's law. Using the same terminology as earlier, Little's law states that, given an average number of online peers (N), the average running time for each peer (t), and an average peer arrival rate (λ), the following relation holds when the system is in equilibrium:

$$N = \lambda t.$$

We have created a script capable of generating an input scenario based on user-specified parameters such as the maximum number G_{max} of simultaneous games peers aim to play, and the values of the variables N, and t. The script works by running in a continuous loop that does the following. First, it generates a random number r in the range $(0, 1]$. Using the value of r, the next inter-arrival time x can be calculated by applying the CDF mentioned earlier. Next, the script generates a wait command with x specified as its argument. Following that, a start command is created for a new peer (selected from an infinite source population). Once a start command has been created for a certain peer, the generator will ensure that that peer will remain running for a randomly selected number of seconds within a certain range. After a peer has stopped, it will not return to the source population.

We should point out that starting and stopping peers are blocking commands, meaning that the execution of the input scenario will halt until the commands have been completed. If left unchecked, this will affect the arrival times of all subsequent peers, resulting in a lower arrival rate. In order to alleviate these effects, the GameTest system will measure the time spent during the execution of these blocking commands, and will attempt to correct subsequent wait commands by subtracting the time that was spent waiting for the blocking commands to complete.

While generating an input scenario for the emulation, we have set the targeted network size (N) to 400 peers. Additionally, the time that a peer will remain running is uniform randomly chosen from the range between 10 and 40 minutes (amounting to 25 minutes on average for t).

Figure 4.5 shows the number of online peers within our emulated network as a function of time. The experiment is performed in real-time and lasts 8 hours, during which 7,618 unique non-recurring peers are part of the network, while the maximum network size is 446 peers. Additionally, the peers on the network play a total of 7,736 games of chess. Two of the peers in the network are superpeers, which remain online for the entire duration of the experiment. Additionally, the first half hour of the experiment is the start-up time of the emulation. While presenting the results of the emulation in the next section, we will sometimes exclude this time period from our results, since the network is still in the process of starting.

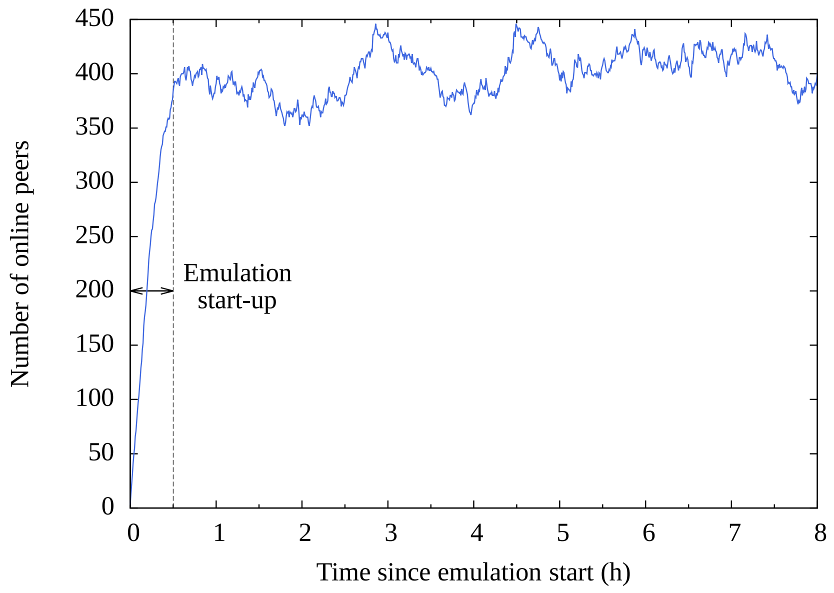


Figure 4.5: The size of the emulated network as a function of time.

4.3.3 Experimental results

In the previous section we introduced twelve metrics for evaluating our system, which we categorized as follows: information dissemination metrics, game agreement metrics, game-play metrics, and bandwidth usage metrics. In the following sections, we will discuss the performance of GameCast in terms of metrics for each of these categories separately.

Information dissemination

In order to asses how well the information dissemination process performs, we monitor six metrics while running our experiment. First, we have monitored the knowledge of online peers about all peers and games that have been added to the network since the start of the emulation (metrics 1A and 1B). The behaviour of these metrics as a function of time is displayed in Figure 4.6. Both metrics are much higher during the first half hour of the experiment because then there are only a few peers leaving the network (recall that peers remain online for about half an hour on average), ensuring that the average fraction keeps rising. However, after the first half hour, more and more peers start to leave the network, leading to a near exponential decay. Note that metric 1B is much lower than metric 1A since only the owner of a game is involved in its distribution, whereas peer information is distributed by all peers that are connected to it. At the end of our experiment, the fraction of known gaming peers has reached 0.04, while the fraction of known peers has reached 0.01, meaning that every peers knows on average 4% of all peers (both off-line and online), and 1% of all games. If we continued the experiment, both metrics would have continued to decrease, because as peers leave the network, the distribution of their peer and game information also stops. Additionally, new games are continuously being created, leading to a continued decrease in the metrics. The results of metrics 1A and 1B tell us that information from inactive peers quickly disappears from the network, which prevents peers from receiving irrelevant information.

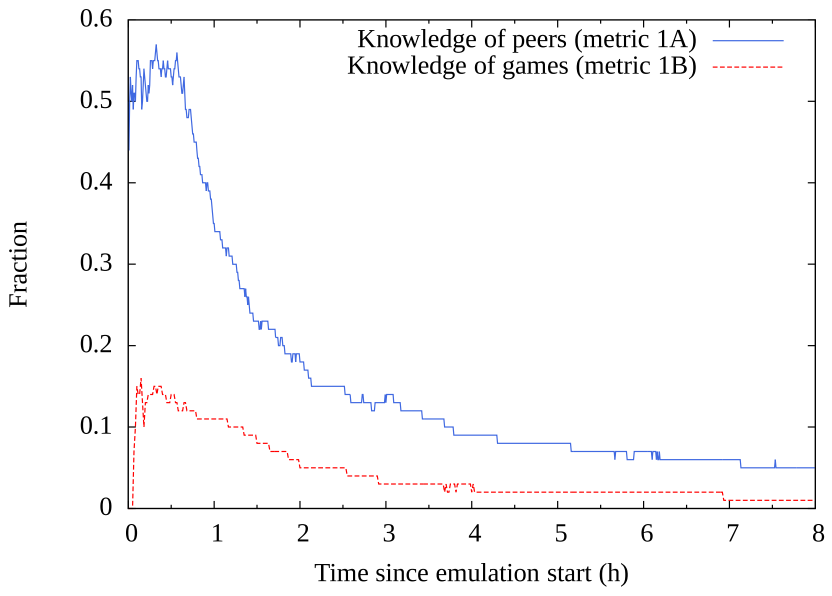


Figure 4.6: The knowledge of peers about the peers and games that were added to the network since the start of the emulation as a function of time.

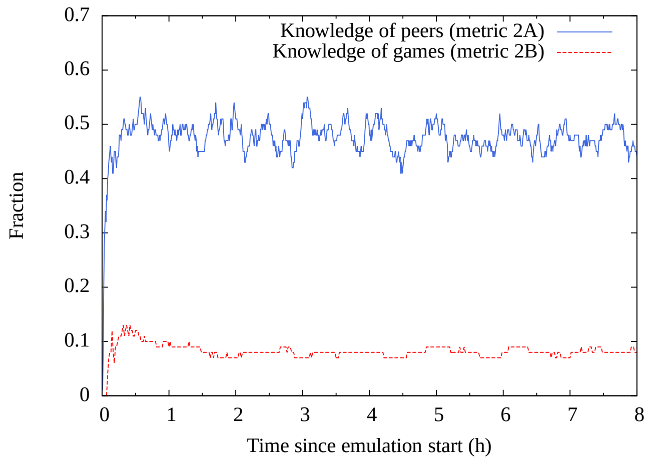


Figure 4.7: The knowledge of peers about the peers and games that were added to the network since joining the system as a function of time.

Figure 4.7 shows the knowledge of peers about peers and games that have been added to the network since joining the system (metrics 2A and 2B). The fractions of knowledge that peers posses are remarkably stable: on average, peers know about 50% of all peers that have connected to network since they joined the system, and almost 10% of all games that finished since they joined the system.

In order to gain insight into the number of peers that are aware of certain information after its insertion into the network, we have monitored the number of peers that are aware, over short time intervals, of the finished games or active peers during each interval (metrics 3A and 3B). The results, displayed in Figure 4.8, show

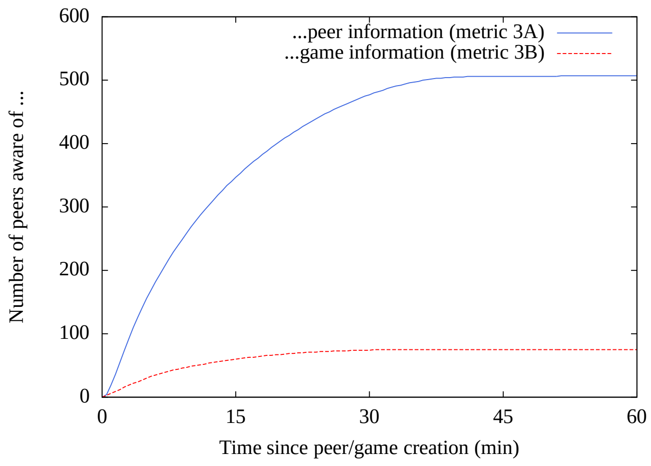


Figure 4.8: The number of peers that are aware of a certain peer or a certain finished game since its insertion into the network as a function of the time.

that on average, 5 minutes after a peer arrives in the network, over 100 peers are aware the existence of the newly arrived peer. It is because of this behaviour, users will be able to invite other players within the network just moments after starting Tribler-G. For games, the behaviour is similar, but less pronounced due to that fact that games are only distributed by their owners. Nonetheless, around 10 minutes after a game has finished and starts to be distributed, 50 peers are aware of the game in question. This is more then we initially expected since each peer will only periodically send the games it owns to a maximum of 10 connected game buddies and 10 connected random peers (see Section 3.3.2). However, since many peers join/leave the network over time, the connected game buddies and random peers also change frequently, leading to a much larger number of peers that are aware of a certain game. Since peers are only online for a relatively small period of time (ranging from 10 to 40 minutes), the number of peers that receive certain information quickly stops increasing. Note that since games are created after a peer is already online for some time, it will be distributed for a smaller period of time, leading to game information converging much faster.

Game agreement

When peers in the network wish to play a game against each other, they must first agree on the parameters of the game. Inviting a specific peer on the network is achieved by sending messages in a point-to-point fashion. Inviting a random peer, however, is achieved by spreading a message within a distance of two hops. How fast this message is spread and how many peers are reached is very important for finding a suitable opponent.

Figure 4.9 (left) shows the distribution times of the random peer invites that were sent during the experiment after the emulation start-up phase (metric 4). In most

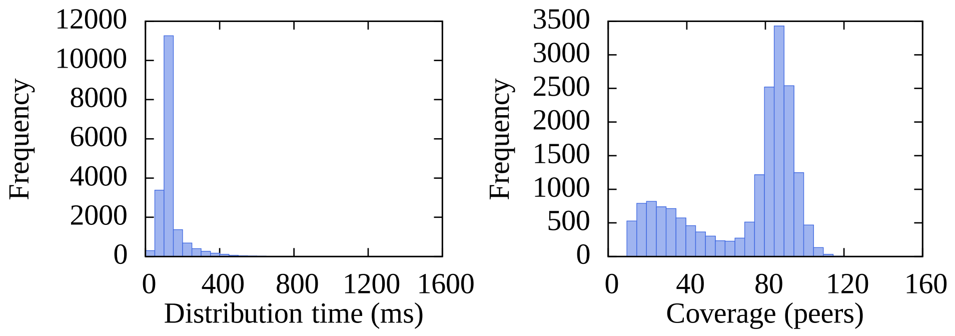


Figure 4.9: The frequencies of the distribution times and network coverages for random peer invites.

cases, spreading the invite over a two hop radius is achieved within just 500 ms. Of course, due to the ideal network transmission times of the DAS4, we can expect that real-world distribution times are higher.

Additionally, Figure 4.9 (right) shows the number of peers that a single random peer invite was able to reach after the emulation start-up phase (metric 5). The minimum number of peers that were reached were about 10 peers. On average, however, the number of peers that were reached was much closer to 90 peers. This is well below the theoretical limit: each peer maintains a maximum number of 10 connections with game buddies and an additional 10 connections with random peers. Random peer invites are spread over a two hop distance giving us a maximum of 420 peers that can theoretically be reached. However, considering that not all peers will maintain the maximum number of allowed connections, and the fact that many of the neighbours of a peer tend to have connections between each other (i.e., clustering), this number is considerably lower.

Game-play

Figure 4.10 shows the time it takes to set up a game (metric 6). In most cases, the game set-up time stays within 100 ms. All measured game set-up times were below 350 ms.

Because in competitive games the game clock will often determine whether a player wins or loses, keeping the game clocks of different players synchronized is of vital importance. We already have a simple mechanism in place to help us achieve this (see Section 3.3.4), but we would like to know more about how large a typical clock correction is. Figure 4.11 shows how much peers need to correct

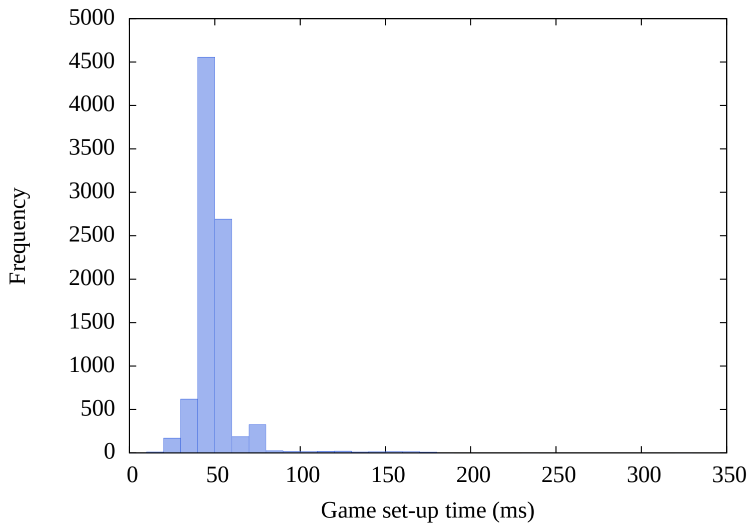


Figure 4.10: The frequencies of the game set-up times during the GameCast evaluation.

the clocks of their opponents (metric 7). Much like the game set-up times, the maximum clock correction stayed within 450 ms on all occasions. Without any corrections, this would lead to the game clocks differ several tenths/hundreds of seconds each time a player makes a single move. This would lead to a difference in game clocks at the end of the game.

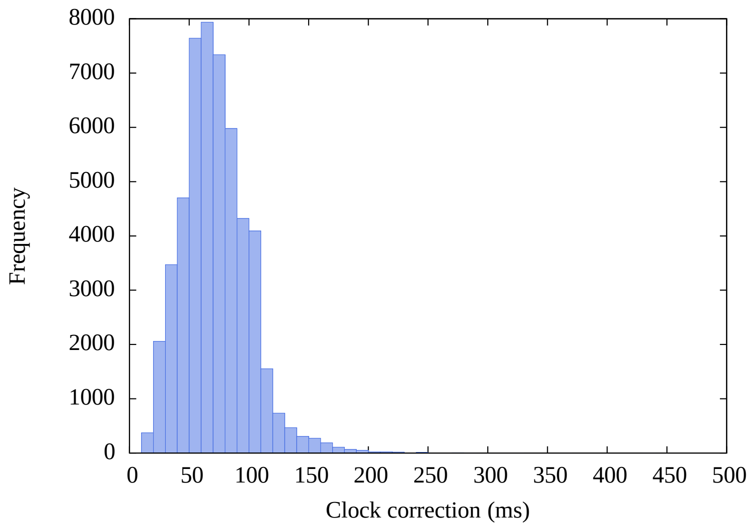


Figure 4.11: The frequencies of the clock corrections needed to synchronize the game clocks of peers on the emulated network.

Bandwidth usage

In order to test the scalability of the GameCast protocol, we have measured the average bandwidth usage of all peers in the network. We have measured the information dissemination protocol separately, in order to get a more clear picture of how the bandwidth is used.

Figure 4.12 shows the average bandwidth used during the experiment (metrics 8 and 9). When looking at the bandwidth usage of the GameCast commands (i.e., the messages related to setting up and playing games), we notice that on average the bandwidth usage remains within about 200 bytes/second. The bandwidth used by the process of information dissemination, also showed in Figure 4.12, is only about 550 bytes/second for each player on average. Even during the emulation start-up, the bandwidth usage rises quickly to about the same level.

We should point out the we have measured that bandwidth at the application level, meaning that the lower layers of the TCP/IP model are not taken into account (e.g. TCP headers, IP headers, etc.). Despite the fact that the actual bandwidth usage will be higher, we believe that the GameCast protocol is efficient enough for real world use, were the bandwidth used by GameCast is unlikely to be noticeable by the user.

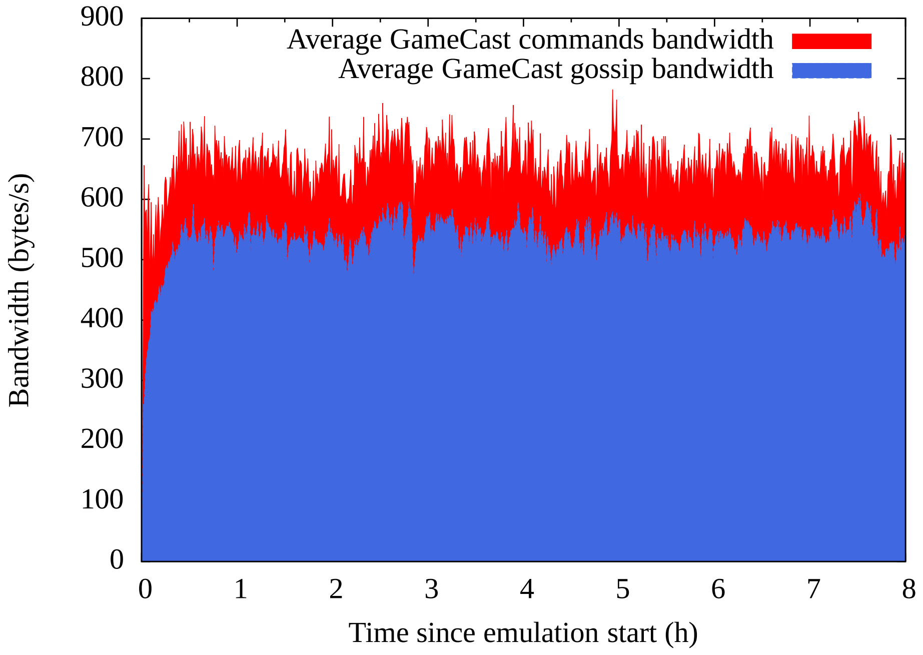


Figure 4.12: The average bandwidth usage of peers on the emulated network as a function of time.

Chapter 5

User testing

In this chapter we elaborate on the results acquired during our two rounds of user testing. The goal of user testing is to validate that Tribler-G meets the functional requirements specified in Section 3.1. In addition, in particular related to the GUI, we wanted to gain insight into the user experience. User testing was performed in two rounds. During the first round, we encountered several technical issues with the software and protocol itself. After fixing these issues, we performed a second round of user testing, during which the performance of Tribler-G was on par with existing centralized solutions. Having studied the results of the questionnaire from the second (and final) round of user testing, we noticed that the participants were overall positive about Tribler-G in terms of software quality and usability.

In Section 5.1 we specify the process that we used while performing user testing. In Section 5.2 we provide the results acquired from both rounds of user testing.

5.1 Testing procedure

For each round of the user test, we found a group of 6 six Master and PhD students in computer science willing to participate. While both rounds of the user test were made using 6 volunteers, not all volunteers participated in both tests. The ages of our volunteers were between 25 and 35 years. None of the volunteers were involved in the design or implementation of Tribler-G, and none had prior knowledge of the system, with the exception of a 20-minute briefing. While we had a lab room with standard computer equipment at our disposal, volunteers were permitted to use an alternate location. During user testing, the volunteers did not interact directly.

User testing was done in two iterative rounds, which both lasted around two hours. The first round was carried out in the early stages of the software development process, which gave us time to listen to opinions of the users and adjust the software accordingly. During the first round, the Tribler-G software showed performance and time synchronization issues (see Section 5.2.2). The second round of user testing was carried out several months later, after the software development was completed.

The process for both rounds of user testing was as follows. First, prior to using the software, the volunteers were briefed about the main features of Tribler-G and the purpose of the user test. After the briefing, the volunteers were given a questionnaire and a software manual. The manual (see Appendix A) provides an overview of six main features of Tribler-G (e.g., creating a new game, accepting a challenge from a fellow user), and lists the tasks that should be performed to use each of them (e.g., clicking on a button or selecting a certain item from a list). The users were asked to use this manual while testing the software, and to fill out the questionnaire (see Appendix B) when testing was completed. During the user test, an assistant was present to conduct informal interviews, provide assistance, and monitor the software.

5.2 Test results

This section describes the results acquired from both rounds of user testing. First, Section 5.2.1 will provide the results from the questionnaire. Second, Section 5.2.2 will elaborate on some of the software issues that we discovered while monitoring the behaviour of the software.

5.2.1 Questionnaire

Upon completion of the user test, our volunteers were asked to fill out a questionnaire. The questionnaire starts with ten multiple-choice questions, followed by four non-multiple-choice questions. The first seven questions of the questionnaire are used to verify if the users are satisfied with the implementation of the functional requirements listed in Section 3.1. The remainder of the questionnaire is meant to gain insight into the user experience.

The results of the multiple-choice component of the questionnaire for rounds one and two are shown in Tables 5.1 and 5.2, respectively. Since the software has not significantly changed between rounds one and two feature-wise, we will discuss the results of the multiple-choice questions from both rounds in one go. The full questionnaire is included in this thesis as Appendix B. For the multiple-choice questions, users are asked to give their opinion about various statements regarding Tribler-G using a 5-point Likert scale, where the negative answers are presented first.

Considering the results from the first seven multiple-choice questions, we can state that our volunteers are generally positive in regards to the implementation of our software. However, they do seem more satisfied with the implementation of Tribler-G's core functionalities (e.g., accepting challenges, sending challenges), and less satisfied with the other functionalities (e.g., discussing games, importing games from other players). This comes as no surprise since we devoted a significant portion of our development efforts to ensuring that the very core functionalities are working perfectly. Furthermore, the answers to the remaining three multiple-choice questions suggest that our participants were satisfied with the Tribler-G user interface and its ease of use, but there still is room for improvement.

Statement	**No. of responses**				
	++	+	+/-	-	–
Creating a new game works as I expected.	1	4	0	1	0
Accepting challenges from other users works without troubles.	1	3	1	1	0
Having the ability to import games from the Free Internet Chess Server (FICS) is usefull.	3	2	1	0	0
Having the ability to play a game against the computer is useful.	1	4	1	0	0
Importing game information from fellow users on the network works well.	1	3	2	0	0
The player statistics give a clear idea of what the ratings of fellow users are.	0	3	2	1	0
The ability to attach comments to games played by fellow users works well.	1	2	3	0	0
Tribler-G is easy to use.	0	3	1	2	0
The user interface is designed well.	0	3	2	1	0
I am good at playing chess.	0	0	1	4	1

Table 5.1: The opinions of the participants of round one about various statements regarding Tribler-G (++ represents strong agreement, – represents strong disagreement).

Statement	**No. of responses**				
	++	+	+/-	-	–
Creating a new game works as I expected.	1	4	0	1	0
Accepting challenges from other users works without troubles.	1	5	0	0	0
Having the ability to import games from the Free Internet Chess Server (FICS) is usefull.	3	1	2	0	0
Having the ability to play a game against the computer is useful.	3	2	0	1	0
Importing game information from fellow users on the network works well.	2	1	3	0	0
The player statistics give a clear idea of what the ratings of fellow users are.	2	2	2	0	0
The ability to attach comments to games played by fellow users works well.	0	3	2	1	0
Tribler-G is easy to use.	0	5	1	0	0
The user interface is designed well.	1	2	3	0	0
I am good at playing chess.	0	0	3	3	0

Table 5.2: The opinions of the participants of round two about various statements regarding Tribler-G.

Next, we will provide the results from the non-multiple-choice questions. We will not discuss the answers that each participant gave individually, but give an impression of what the overall consensus is.

Question 1: What are the positive features of Tribler-G?
Round one: Participants generally found the user interface to be good-looking, responsive, and easy to work with. Additionally, two of the participants were particularly fond of the statistics information that Tribler-G provided.
Round two: Again, almost all participants were happy with aesthetics of the user interface, and found it easy to work with. Some users, especially those who also participated in round one of the user test, noted that the software was stable and performed well.

Question 2: What are the negative features of Tribler-G?
Round one: Three out of six participants noticed a high delay while undertaking certain actions, such as responding to an invitation and making a move (see also Section 5.2.2). Additionally, two of the participants suggested that the user interface should provide more feedback. Therefore, after the first round of user testing, additional status information and pop-up notifications were added to the software.
Round two: One of the participants pointed out that Tribler-G currently aborts a game, when one of the players fails to make a move within the allotted time, but prefers that this player loses the game instead. We plan to make this change in software behaviour in the next version. Another participant suggested that the messages posted using Tribler-G's discussion feature should be listed in the same order for all users.

Question 3: How would you improve Tribler-G?
Round one: Besides fixing responsiveness problems that occurred when participants issued commands through the user interface (see Section 5.2.2), most of the participants found that the user interface could be more informative and suggested using pop-ups. Additionally, users suggested that we work on expending the Tribler-G user-base, which would make it easier to find opponents on the peer-to-peer network.
Round two: Two participants suggested including a chat feature while a game is being played. Another, suggested that inviting a specific gamer on the network should be done using the gamer's user-name rather then the PermID. Also, most participants would like to see more available games, and a user interface that shows more clearly the status of current games and available invites. Finally, two out of the six user suggested that the chessboard should display immediately after the game has started.

Question 4: Would you use Tribler-G again? If not, please explain why.
Round one: Two out of the six participants said that they would use Tribler-G is there were more users on the network. One participant said that he would use

Tribler-G again, if certain actions did not have such high delays. The remaining participant suggested incrementing the number of games, since they did not like to play chess.
Round two: Two out of the six user said that they would like to use Tribler-G in the future. All four remaining users, said that they did not like chess, but they would consider using the software if there were other game available as well.

Overall, our participants were happy with Tribler-G in terms of software quality and usability. However, there are still many areas in which our software can be improved (see also Section 6.2 for a list of future work). Considering the answers to the question about using the software again, is it important to extend the array of available games in the future.

5.2.2 Software issues

When we did our first round of user testing, we noticed two major issues: the game clocks were often not properly synchronized during a game, and Tribler-G suffered from a lack of responsiveness when using functions such as making a game move or accepting an invitation from another player.

Clock synchronization

The clock synchronization issue was caused by the difference between the time at which a player makes a move and the moment the other player(s) receive a move (i.e., latency). We have addressed the problem by introducing a clock synchronization mechanism that ensures that when a player makes a move, the time taken for the move in question is also included. The subject of game clocks is discussed at greater length in Section 3.3.4.

Software responsiveness

Concerning the issue of lack of responsiveness, we found that running the BuddyCast protocol in the background resulted in significant delays in the application code. We did not notice the severity of these issues until the user testing because, during the development of GameCast, BuddyCast was deactivated for most of the time. Disabling BuddyCast enabled us to start and test Tribler-G instances more quickly on our development computer. This section will further discuss this responsiveness problem and its solution.

Before we go any further, it is useful to explain how Tribler uses various threads of execution while passing messages over the secure overlay. As we already mentioned in the introductory chapter, Tribler's secure overlay enables high-level communication between peers. Tribler protocols, such as BuddyCast, ChannelCast, and of course GameCast, use the secure overlay to send their messages. When Tribler sends or receives a message over the secure overlay, it uses multiple threads of execution to complete the operation. There are two threads relevant to the passing of messages over the secure overlay. First, Tribler has a separate thread, called the *network thread*, that handles all network related tasks, such as receiving and sending messages. Second, Tribler has another thread, called the *overlay thread*, to execute the protocols that run on the secure overlay. The network thread and overlay thread are in constant contact with each other. For instance, when a BuddyCast message arrives, the network threads reads the message from the network socket, and passes the message to the overlay thread, which processes the message further. Similarly, when Tribler decides to send a BuddyCast message, the overlay thread will create the message and subsequently pass it to the network thread, which will ensure that the message is written to the network socket. Activities such as writing to a database or file also frequently occur on the overlay thread. The overlay thread is implemented as a simple queue of tasks which get executed one at a time. Other threads schedule these tasks and the overlay thread executes these tasks in FIFO order.

As we suspected that the responsiveness issues are related to activities that mostly utilize the overlay thread, we measured the delays of the tasks executed by the overlay thread. We found that the overlay thread was severely overloaded, to the point that the application was becoming non-responsive. Figure 5.1 shows that delays of the tasks that were executed within the first 60 minutes after start-up. In the most severe case, a task was delayed for 4.5 minutes. Clearly, if a user needs to wait over 4 minutes to send a message, that user will mostly be displeased with our software. However, when running Tribler-G with BuddyCast disabled, the delays of the tasks stay below 3 seconds. We also measured the delay of tasks with GameCast disabled (and BuddyCast enabled) and still noticed very high delays. Since BuddyCast does not require the timely sending or receiving of messages, these delays have never been reported as a problem by users, in the past.

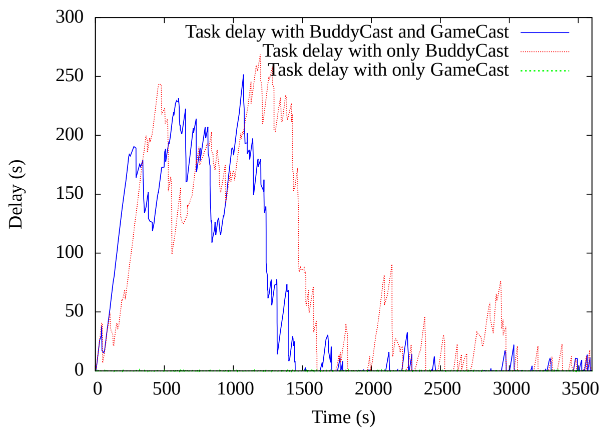


Figure 5.1: Delays of scheduled tasks on the overlay thread while running Tribler-G based on Tribler 5.2.1.

Since the problem seems to be caused by Tribler-G components other than GameCast, and the Tribler version that we used as a basis for Tribler-G was almost a year old, we decided that moving to a recent version was appropriate. Much to our surprise, the delay problems of the overlay thread seemed to have disappeared completely. When running Tribler-G based on Tribler 5.3.8, the delays of tasks peak stayed below 4.5 seconds, on all occasions (see Figure 5.2). Note that the actual delays will likely be lower, since taking these measurements require additional debugging code to be executed. After looking further into this matter, we found that the ChannelCast protocol used in Tribler 5.2.1 used very inefficient means to access the database. This issue has been fixed in more recent versions, significantly reducing the load on the overlay thread and therefore also the delays of the tasks.

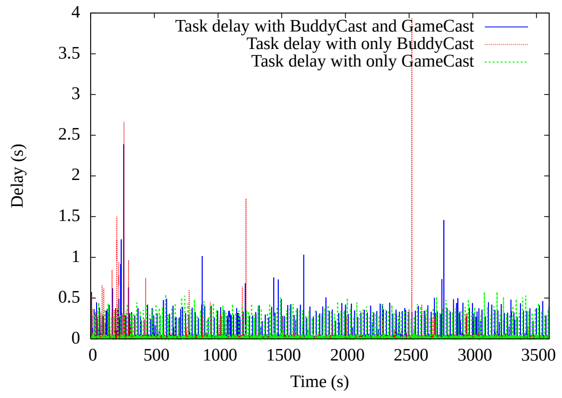


Figure 5.2: Delays of scheduled tasks on the overlay thread while running Tribler-G based on Tribler 5.3.8.

To further suppress task delays, we have also created a separate GameCast thread. The GameCast thread is only used for tasks on which the user is waiting (i.e., high-priority tasks), for instance, making a move or replying to an invitation. Tasks of which the user is unaware (i.e., low priority tasks), such as sending messages related to the information dissemination process, are scheduled normally on the overlay thread.

In conclusion, the major problems of Tribler-G have been resolved, as indicated by the responses of our test users, after round two of testing.

Chapter 6

Conclusion

In Section 6.1 we give a summary of our work and provide our conclusions. Next, in Section 6.2, we propose future work that would further improve our current system.

6.1 Summary and conclusions

In this thesis we have presented a decentralized system that allows users to play turn-based board games over a peer-to-peer network. Our current implementation, called Tribler-G, is build as an extension to the Tribler file-sharing application, and focuses on enabling users to play online chess. Tribler-G supports all main gaming features that traditionally only exist on systems with a central authority, such as the Free Internet Chess Server (FICS), and realizes them in a decentralized setting. The result is a scalable and easy to use application that offers online chess players an attractive alternative to the current centralized services, which typically generate revenue using advertisements and subscription fees.

In order to solve the problem of playing chess over a peer-to-peer network, we have designed and implemented GameCast, a protocol which allow players to explicitly invite one another or invite any player within a certain rating. Additionally, besides the functionalities required for playing a game, GameCast will ensure that finished games are distributed throughout the network. Players receiving the game will be able to review the game and attach comments. Despite the fact that Tribler-G currently only implements online chess, GameCast has been designed to support a variety of turn-based board games, including games that require more than two players.

To realize the GameCast protocol, we use an epidemic mechanism to allow peer discovery and game distribution within the network. Additionally, we have have based the GameCast command syntax on a protocol commonly used by online chess servers, known as the ICS protocol. Furthermore, the mechanisms used for player invitation and for playing a game are based on the idea that for each game, its creator/owner is responsible for administrative tasks such as notifying all players that the game has started, distributing game information and ensuring that comments are distributed.

In order to evaluate the performance of the GameCast protocol, we have created GameTest, a system capable of emulating a peer-to-peer network by starting or stopping a Tribler-G instance for each joining or leaving peer. Using GameTest, we conducted a large-scale emulation on the DAS-4 distributed supercomputer. Additionally, using small groups of six people, we performed two initial rounds of user testing. Based on the results acquired during the emulation as well the user testing, we can draw the following conclusions:

GameCast is effective. Based on the observations made during the emulation, an opponent on the network can typically be found in a matter of seconds. When sending random peer invites, the invites often reached about 75 peers, which is sufficient to find a suitable opponent. When an opponent is found and a game is being played, our clock synchronization mechanism consistently ensured that the game clocks of the players remain synchronized. Further strengthened by the positive outcome of the user testing, we conclude that the GameCast protocol is an effective means to providing simple gaming functionalities over a peer-to-peer network.

GameCast is scalable. Excluding overhead caused by TCP/IP message headers, GameCast uses only 750 bytes/second on average for a peer that is actively playing chess. Of this average bandwidth, about 550 bytes/second can be attributed to gossiping messages. Since each peer sends its gossiping messages at a constant rate, an increase in the network size of the peer-to-peer network will not result in an increased bandwidth usage for each individual peer. Furthermore, the bandwidth caused by the execution of GameCast commands is limited by the number of actions a chess player can make within a certain time period. However, the bandwidth may increase in a less interconnected network, due to the spreading of invites across a two-hop radius. However, since the number of simultaneous connections that a peer can maintain with other peers is bounded, the number of peers that a single invite can reach also has an upper bound.

6.2 Future work

While Tribler-G is a considerable step towards creating an online distributed social network on which users can play board games, there are still quite a few areas in which Tribler-G can be improved. We have categorized these improvements into two main groups: improvements that fix existing technical issues, and improvements that somehow extend the system. For the first group, we feel that the following are among the most important:

Improving the synchronization mechanism When playing a game, each player keeps track of how much time each of the players has used so far. However, the time that passes between the moment at which a player makes a move and the moment the other player(s) receive a move, will result in an offset between the clocks of the players. The FICS deals with these network latency issues by introducing Timeseal, an application that runs on the player's machine and notifies the FICS of the time that the player has taken to make a move. The GameCast time synchronization mechanism closely resembles Timeseal, in that both mechanisms measure that time a move has taken from the player's perspective. Unfortunately, since a player is on a trusted entity, this mechanism also introduces a vulnerability which is likely to be exploited by malicious users. Therefore, we would like to alter our current mechanism so that it would be more tamper proof.

Improving GameCast security An issue with GameCast, and peer-to-peer networks in general, is what happens when one or more malicious peers are introduced into the network. For instance, malicious peers could start distributing fake game information in order to affect the rating of a player (i.e, Sybil attacks).

Dealing with NAT firewalls Many computers connected to the internet today are behind NAT firewalls. NAT creates a private IP address realm separate from the Internet, which often results in difficulties accepting incoming connections from external hosts. This results in a number of issues when unconnectable peers are using GameCast. For instance, a peer behind a NAT firewall receives a random peer invite from game buddy, but is unable to respond because that peer is behind a NAT firewall and cannot accept a connection. A possible way of increasing the usability for users that can not accept incoming connections, could be to increase the role of the super-peers (e.g., messages between peers could be passed trough a connectable superpeer).

Further research into protocol parameters Further research is required in order to determine what the effects of different GameCast parameters are. For instance, peers involved in information distribution are currently only allowed to send/receive messages every 5 minutes. How would changing this parameter affect the bandwidth usage?

Concerning the second group of improvements, which would extend Tribler-G, we feel that the following are among the most important:

Extending the number of available games Many people do not know how to play chess or simply do not like it. Therefore, we would like to extend the number of available games to include, for instance, Scrabble, Monopoly, Checkers, and Go.

Support for random number agreement We would like to include a mechanism that allows peers to agree on a random number, which would help generate random content for games that require this. Think for instance of board games such as Scrabble and Rummikub, or card games such as Texas Holdem Poker.

Penalize leaving the game before it is over Currently, when a game expires (i.e., one of the players fails to move in time) it is discarded. This effectively means that a player who is loosing a game can simply stop playing in order to prevent a negative impact on his/her rating. Therefore, in future versions of Tribler-G, the player that fails to move in time should be considered the loser of the game.

Extending the number of GameCast features Currently GameCast supports the most basic functionalities in order to allow players to play games against each other. However, there are also a number of additional functionalities that we feel would improve the gaming experience. These functionalities include allowing users to observe a game in progress, allowing opponents to chat while playing a game, support for unrated games (i.e., games of which the outcome does not affect the ratings of the players), allowing players to suspend/resume games, support for game tournaments, support for anti-spam functionalities for the discussion board, allowing for the messages in the discussion board to be nested and ordered, offering chess instruction videos through the Tribler download feature, and introducing some kind of trust rating for players.

Unique user names Like the standard Tribler application, Tribler-G identifies peers on the network using quasi-unique permanent identifiers. While these identifiers can be considered unique, the user names are not. Therefore, it stands to reason that at some point the user will be confronted with multiple players on the network using the same user name. This can make identifying a certain player difficult. Therefore, we would like future Tribler-G releases to support unique user names, if possible.

Bibliography

[1] Call of duty modern warfare 3. `http://www.callofduty.com/mw3`.

[2] Chess on facebook. `http://apps.facebook.com/chessfb`.

[3] Das-4: Distributed asci supercomputer 4. `http://www.cs.vu.nl/das4`.

[4] Diaspora. `https://joindiaspora.com`.

[5] Free internet chess server. `http://www.freechess.org`.

[6] Gnu chess. `http://www.gnu.org/software/chess`.

[7] Gnu social. `http://www.gnu.org/software/social`.

[8] Gnutella. `http://www.gnutella.com`.

[9] Internet chess club. `http://www.chessclub.com`.

[10] Kgs go server. `http://www.gokgs.com`.

[11] Noserub. `http://noserub.com`.

[12] Yahoo! chess. `http://games.yahoo.com/ch`.

[13] 20 Million Unique Players Log Into Call Of Duty Every Month. `http://www.xboxdailynews.com/2011/09/05/20-million-unique-players-log-into-call-of-duty-every-month`, 2011.

[14] Social Gaming on Track to Become 5 Billion Industry by 2015. `http://www.parksassociates.com/blog/article/parks-pr2011-socialgaming`, 2011.

[15] Haakon Bertheussen. Wordfeud. `http://www.wordfeud.com`.

[16] Ashwin Bharambe, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. Donnybrook: enabling large-scale, high-speed, peer-to-peer games. In *SIGCOMM*, pages 389–400, 2008.

[17] Blizzard Inc. World of Warcraft subscriber base reaches 11.5 million worldwide. `http://us.blizzard.com/en-us/company/press/pressreleases.html?081121`, 2008.

[18] Blizzard Inc. World of Warcraft. `http://us.battle.net/wow/en/`, 2011.

[19] Egbert Bouman. A survey of developments in online social networks. Technical report, Delft University of Technology, 2010.

[20] Danah M Boyd and Nicole B Ellison. Social network sites: Definition, history, and scholarship. *Journal of Computer-Mediated Communication*, 13(1):210–230, 2008.

[21] Pete Cashmore. FarmVille Surpasses 80 Million Users. `http://mashable.com/2010/02/20/farmville-80-million-users`, 2010.

[22] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: a distributed anonymous information storage and retrieval system. In *International workshop on Designing privacy enhancing technologies: design issues in anonymity and unobservability*, pages 46–66, 2001.

[23] B. Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, May 2003.

[24] Carlton R. Davis, Stephen Neville, José M. Fernandez, Jean-Marc Robert, and John Mchugh. Structured peer-to-peer overlay networks: Ideal botnets command and control infrastructures? In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 461–480, 2008.

[25] Mark E. Glickman. Parameter estimation in large dynamic paired comparison experiments. *Applied Statistics*, 48:377–394, 1999.

[26] Thorsten Hampel, Thomas Bopp, and Robert Hinn. A peer-to-peer architecture for massive multiplayer online games. In *NetGames*, pages 48–52, 2006.

[27] Betjeman House. Technical report : An estimate of infringing use of the internet. *Analysis*, (January):1–56, 2011.

[28] Robert M. Hyatt. Crafty. `http://www.craftychess.com`, 1994.

[29] Bonnie E. John and David E. Kieras. The goms family of user interface analysis techniques: comparison and contrast. *ACM Trans. Comput.-Hum. Interact.*, 3:320–351, December 1996.

[30] Nick O'Neill. 66 percent of facebook traffic is to games. `http://www.socialtimes.com/2010/04/66-percent-of-facebook-traffic-is-to-games`, April 2010.

[31] OpenTTD team. OpenTTD, 2010. `http://www.openttd.org`.

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

[33] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. van Steen, and H. J. Sips. Tribler: a social-based peer-to-peer system: Research articles. *Concurr. Comput. : Pract. Exper.*, 20:127–138, February 2008.

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

[35] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 149–160, 2001.

[36] David Kieras University and David Kieras. Using the keystroke-level model to estimate execution times, 1993.

[37] M. Varvello, C. Diot, and E. W. Biersack. P2P Second Life: Experimental Validation Using Kad. In *INFOCOM*, pages 1161–1169, Apr. 2009.

[38] Anthony Peiqun Yu and Son T Vuong. MOPAR : A Mobile Peer-to-Peer Overlay Architecture for Interest Management of Massively Multiplayer Online Games. In *NetGames*, pages 99–104, 2005.

Appendix A

Guided examples of using Tribler-G

This is a getting started guide for Tribler-G, and was used during user testing (see Chapter 5).

Example 1: Installing and configuring Tribler-G
Before using Tribler-G for the first time, you need to install it to your machine. Additionally, after Tribler-G is installed, you will need to set a number of settings in order to use the software properly. The entire process should only take a few minutes.

1. First, go to the Tribler-G website: `http://tribler-g.org`.

2. If you are using a Windows PC, download and execute the Windows installer and follow the instructions. When entering the installation path, be sure to enter a location that you have write access to. After the installation is complete, launch Tribler-G from the Windows start menu (its listed as ’Tribler’).

   If you are using a Linux PC, download the Tribler-G archive for Linux and open it by double-clicking the file. After opening the archive, extract the folder ’Tribler-G-linux’ to a location you have write access to (e.g., your home folder). Next, open the folder you just extracted in the file manager, and double-click on the file ’run.sh’. At this point, you will be presented with a message box asking you what action to take. Click on ’run’, after which the Tribler-G application should execute.

3. Now that you have started Tribler-G, select the ’Settings’ button from the top bar of the screen. You will now see a window, which, among other things, allows you the set-up a profile and a port number.

4. Next, you need to set-up a nickname, so that fellow users on the network are able to recognize you. Make sure that the ’General’ item is selected in the

listing on the left side of the settings window. You should now be presented with a form which allows you to set your nickname. Next, fill in a nickname of your choosing.

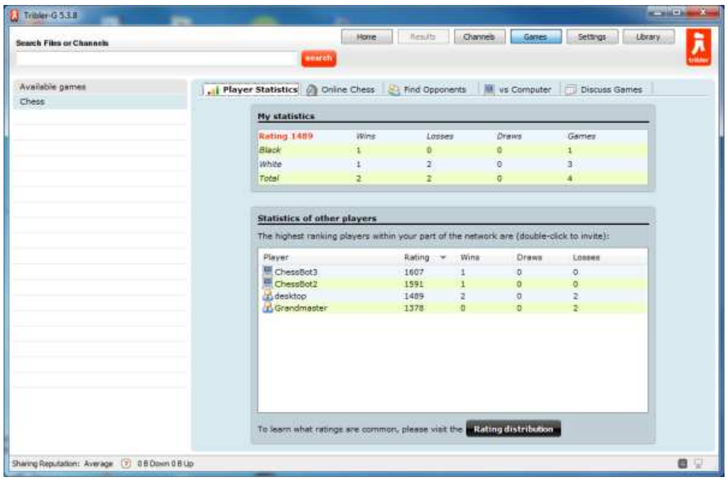
5. We need to make sure that external peers are able to connect to Tribler-G by setting up the correct port number. To do this, make sure that the 'Connection' item is selected in the listing on the left side of the settings window. Ensure that the port number is set to a port that is accessible from the internet, and click the 'Save' button.

6. Restart the application. Tribler-G is now ready to use.

Example 2: Accepting peer-to-peer challenges

Before you can play a game, you need to find a suitable opponent on the network. This is done by either accepting a challenge from another player, or by creating a challenge yourself. In this example we will accept an existing challenge from another player.

1. The top bar of the Triler-G user interface allows you to go through the different panels within Tribler. This bar is included in any standard Tribler installation, and the Tribler-G merely adds an additional 'Games' button to the bar. Please click the 'Games' button. After doing so, the gaming user interface is displayed, which encompasses the bottom two panels displayed in Figure A.1. The left panel presents an overview of the currently implemented games (to date, the only game available is chess). The right panel allows you to play the game that is selected in the left panel.



My statistics

Rating 1489	Wins	Losses	Draws	Games
Black	1	0	0	1
White	1	2	0	3
Total	2	2	0	4

Statistics of other players

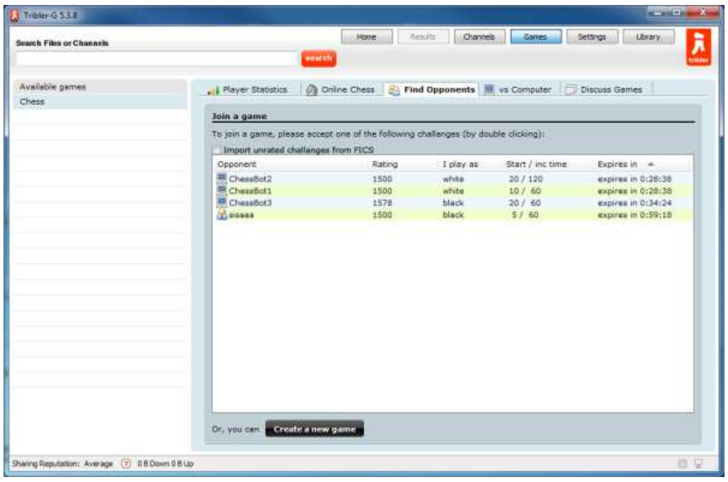
The highest ranking players within your part of the network are (double-click to invite):

Player	Rating	Wins	Draws	Losses
ChessBot3	1607	1	0	0
ChessBot2	1591	1	0	0
desktop	1489	2	0	2
Grandmaster	1378	0	0	2

To learn what ratings are common, please visit the

Figure A.1: Viewing player statistics.

2. Please click on the 'Find Opponents' tab. This will bring you to a screen that shows all challenges that you are eligible to accept (see Figure A.2). At any time there can be different challenges available.



Join a game

To join a game, please accept one of the following challanges (by double clicking):

Import unrated challanges from FICS

Opponent	Rating	I play as	Start / inc time	Expires in
ChessBot2	1500	white	20 / 120	expires in 0:28:38
ChessBot1	1500	white	10 / 60	expires in 0:28:38
ChessBot3	1578	black	20 / 60	expires in 0:34:24
sisaaa	1500	black	5 / 60	expires in 0:59:18

Or, you can Create a new game

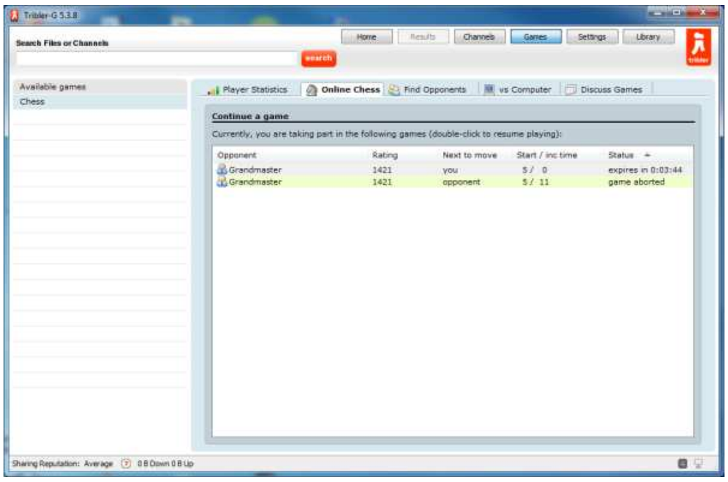
Sharing Reputation: Average 0 B Down 0 B Up

Figure A.2: Viewing outstanding challenges.

Tribler-G differentiates between a number of different types of challenges. First, challenges issued by chess players on the peer-to-peer network. Second, challenges issued by automated chess players, which we call chessbots. Chessbots allow you to play a game of chess on the peer-to-peer network in case that no human players are available (these players can be recognized by their name, which starts with 'chessbot'). Finally, challenges issued by chess players on the FICS network. These challenges are only shown when the option 'Import unrated challenges from FICS' is enabled.

Challenges have several parameters. The 'opponent' and 'rating' parameters state your opponent's name and rating. The 'I play as' parameter states the colour that you will be playing with if you accept the challenge. Next, parameter 'time / inc' denotes the timing settings for the chess clock. The 'time' value denotes the start time in minutes to which the clock of each player gets set, and the 'inc' value refers to the time in seconds with which the clock of a player is incremented when a move is made.

3. We will start with playing a game of chess over the peer-to-peer network by accepting a challenge. If available, pick a challenge issued by a human player on the peer-to-peer network (i.e., an opponent who's name does not start with 'chessbot'), otherwise pick a chessbot challenge. Double-click on the challenge that you picked. This will cause Tribler-G to contact the challenger. If the challenger accepts your response, the challenge will disappear from the list and reappear on list of current games that you are playing, which can be found on the 'Online Chess' tab (see Figure A.3).



Continue a game

Currently, you are taking part in the following games (double-click to resume playing):

Opponent	Rating	Next to move	Start / inc time	Status
Grandmaster	1421	you	5 / 0	expires in 0:03:44
Grandmaster	1421	opponent	5 / 11	game aborted

Figure A.3: Viewing active games.

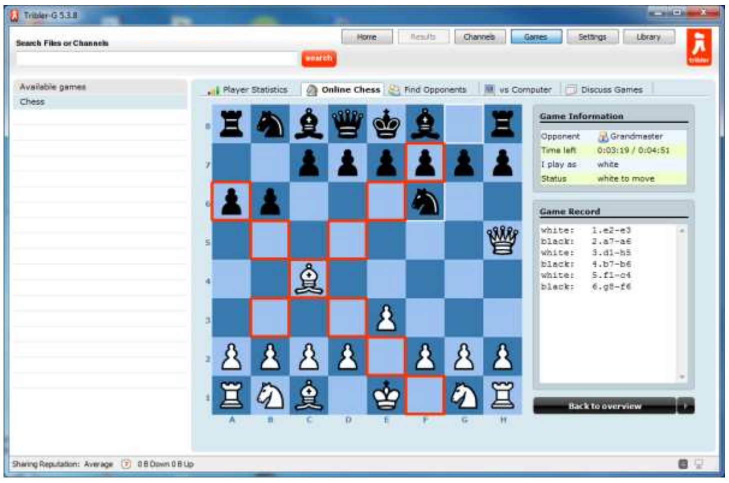
4. At this point you should have a single game entry listed on the ’Online Chess’ tab. Double-click on the game entry in order to go to the chessboard screen. You should now be presented with a screen similar to the one shown in Figure A.4. The screen shows the game information: opponent name, your clock/opponent’s clock, the colour you play with, and which colour is next to move. Below that, you’ll find the game record, which displays all the game moves that have been made. Also, the button ’Back to overview’ will take you back to the previous screen, and the little button next to it will allow you to issue an abort, draw, or resign request. For the abort and draw request the other player will need to agree for the request to be executed.

5. Now for the actual playing of a game. When it is your turn to move, you can click one of your chess pieces, and all the valid moves that you can take using this piece will be marked on the board. If you are unfamiliar with the rules of chess, you could take a look at http://www.chess.com/learn-how-to-play-chess.html.

6. Finish your game of chess. Once you have finished, the outcome of the game will be determined. Unless the game has ended due to a time-out or an abort by agreement, the game will now be displayed in the ’Discuss games’ tab, which we will discuss later.

Example 3: Accepting FICS challenges

Since there does not yet exist a community of users that frequently play chess on the Tribler network, it is entirely possible that you will find yourself unable to find a suitable opponent on the game network. To prevent you from being unable to play a game, Tribler-G is able to import additional invites from the Free Internet Chess



Game Information

Opponent	Grandmaster
Time left	0:03:19 / 0:04:51
I play as	white
Status	white to move

Game Record

```
white:   1.e2-e3
black:   2.a7-a6
white:   3.d1-h5
black:   4.b7-b6
white:   5.f1-c4
black:   6.g8-f6
```

Figure A.4: Playing a game of chess against an online opponent.

Server or FICS. Playing these games will not affect your rating. In this example we will accept a challenge from a FICS user.

1. Go to the challenge list on the 'Find Opponents' tab.
2. Check the 'Import unrated challenges from FICS' option. After waiting for several seconds while Tribler-G is querying the FICS server, you will find additional challenges in this list. These challenges can be recognized by the absence of an expiration time.
3. Accept a challenge from a FICS user by double-clicking it. If nothings happens, it will be because the challenge was already taken by another user. In this case try double-clicking a different FICS challenge.
4. The game itself works exactly the same as with games played over the peer-to-peer network, except that you can only play one FICS game at a time. Once you have finished the game, it will no longer be displayed in the list of current games. Also, games that has been played on FICS will not be displayed in the 'Discuss games' tab, and the game will be removed after is has been played.

Example 4: Creating a new game
Next, we are going to create a new game. A new game should normally be created when you can not find any acceptable challenges, or when you wish to invite a specific player.

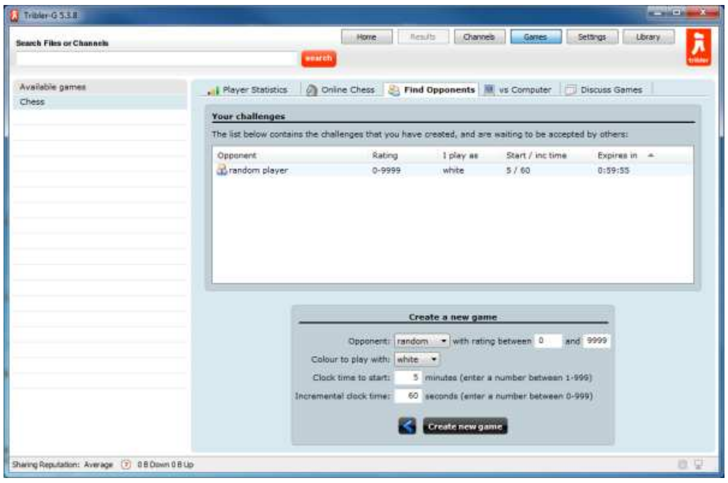
1. In order to create a new game, you will need to go to the challenge list on the 'Find Opponents' tab.

2. Click on the ’Or, create a new game’ button. This will result in the screen shown in Figure A.5.

3. As the creator of the game, you can specify several parameters. First, the type of opponent that you want to play. You can choose to play a game against a random opponent of a certain rating, or play a game against a specific user, which should be identified using his/her permid (i.e., the string that Tribler-G uses to identify the peers on the network). Next, you can choose which colour you want the play with. Finally, we need to choose the timing parameters for the chess clock. The chess clock takes two parameters, namely the start time in minutes to which the clock of each player gets set, and the time in seconds with which the clock of a player is incremented when a move is made. Set the opponent to ’random’, set the other parameters to whatever you prefer, and click ’Create new game’. At this point Tribler-G will attempt to spread a challenge throughout the peer-to-peer network. If you receive an error message stating that there are not enough players connected, please try again after a few minutes.

4. If the game was created successfully, there will now be a new entry in the list of your currently outstanding challenges. Once another user accepts your challenge, the entry will disappear and will re-appear in the list of current games on the ’Chess Online’ tab. At this point you may have to help a fellow user out by accepting his/her challenge, because otherwise none of the challenges will get excepted.

5. Play the game like you would normally.



Your challenges

The list below contains the challenges that you have created, and are waiting to be accepted by others:

Opponent	Rating	I play as	Start / inc time	Expires in
random player	0-9999	white	5 / 60	0:59:55

Create a new game

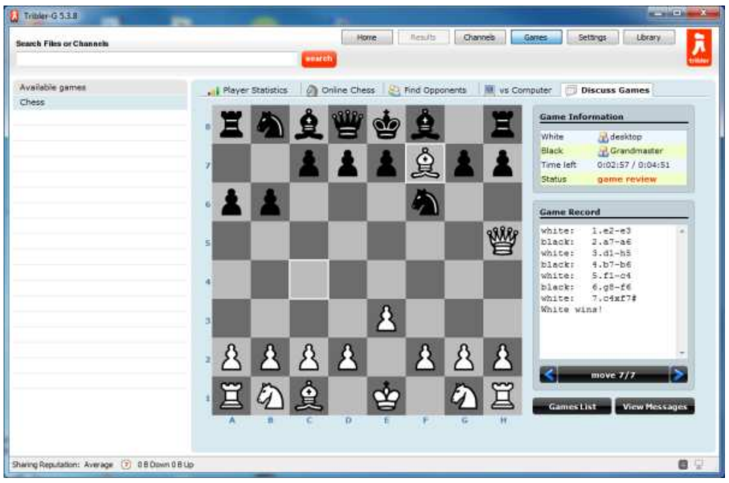
Opponent: random with rating between 0 and 9999
Colour to play with: white
Clock time to start: 5 minutes (enter a number between 1-999)
Incremental clock time: 60 seconds (enter a number between 0-999)

Figure A.5: Creating a new game.

Example 5: Viewing statistics from fellow users

1. Please click on the 'Player statistics' tab. You should not be presented with statistics related to all games that were gathered during the distribution process, as well as your own games. The top panel shows you your rating (when players first join the network, they receive a rating of 1500) and how many games were won/lost, while the bottom panel shows the same, but of the top 25 chess players within the network.

2. Click on the 'Rating distribution' button. You will now see a histogram of the rating distribution of known players in the network. This diagram is meant to give you a more global picture of ratings of other known users.



White	desktop
Black	Grandmaster
Time left	0:02:57 / 0:04:51
Status	game review

```
white:   1.e2-e3
black:   2.a7-a6
white:   3.d1-h5
black:   4.b7-b6
white:   5.f1-c4
black:   6.g8-f6
white:   7.c4xf7#
White wins!
```

Figure A.6: Reviewing a game.

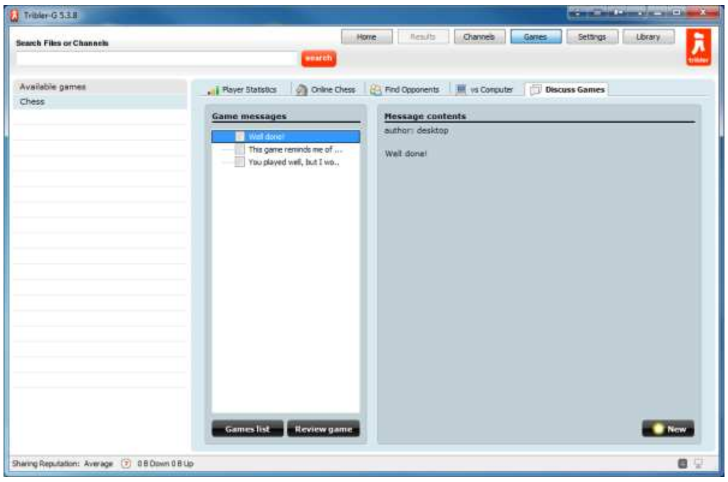
Example 6: Using Tribler-G's review feature

1. Please click on the 'Discuss games' tab. At this point you will see a list of all known games (including games that have been gathered during the information dissemination process). Since the list can grow rather large, there is also a search option which allows you to display only games related to a particular player.

2. Double-click a random entry, and you will have access to the review panel (see Figure A.6). In the review panel you can visually browse through all the moves that have been made during the game using the previous and next button.

3. Next, click on 'View Messages', which will bring you to the screen were all known messages pertaining to the game in question are listed.

4. Now, create a new message by clicking ’New’, at which point you can fill in the message you would like to send. The message can be send by clicking ’Post’. Once you have sent your message, the user that created the game will first receive the message. After that, the creator will start distributing the newly received message in future gossip messages (which may take some time).



Well done!
This game reminds me of ...
You played well, but I wo...

author: desktop

Well done!

Sharing Reputation: Average 0 B Down 0 B Up

Figure A.7: Viewing the list of messages related to a game.

This completes the guided examples of using Tribler-G.

Appendix B

Tribler-G questionnaire

This questionnaire was used for collecting user experiences during user testing (see Chapter 5).

1. **Creating a new game works as I expected.**
   - ☐ Strongly disagree
   - ☐ Disagree
   - ☐ Neutral
   - ☐ Agree
   - ☐ Strongly agree
2. **Accepting challenges from other users works without troubles.**
   - ☐ Strongly disagree
   - ☐ Disagree
   - ☐ Neutral
   - ☐ Agree
   - ☐ Strongly agree
3. **Having the ability to play a game against the computer is useful.**
   - ☐ Strongly disagree
   - ☐ Disagree
   - ☐ Neutral
   - ☐ Agree
   - ☐ Strongly agree

4. **Having the ability to import games from the Free Internet Chess Server (FICS) is usefull.**
   - ◻ Strongly disagree
   - ◻ Disagree
   - ◻ Neutral
   - ◻ Agree
   - ◻ Strongly agree
5. **Importing game information from fellow users on the network works well.**
   - ◻ Strongly disagree
   - ◻ Disagree
   - ◻ Neutral
   - ◻ Agree
   - ◻ Strongly agree
6. **The player statistics give a clear idea of what the ratings of fellow users are.**
   - ◻ Strongly disagree
   - ◻ Disagree
   - ◻ Neutral
   - ◻ Agree
   - ◻ Strongly agree
7. **The ability to attach comments to games played by fellow users works well.**
   - ◻ Strongly disagree
   - ◻ Disagree
   - ◻ Neutral
   - ◻ Agree
   - ◻ Strongly agree
8. **Tribler-G is easy to use.**
   - ◻ Strongly disagree
   - ◻ Disagree
   - ◻ Neutral
   - ◻ Agree
   - ◻ Strongly agree

9. **The user interface is designed well.**
   - ☐ Strongly disagree
   - ☐ Disagree
   - ☐ Neutral
   - ☐ Agree
   - ☐ Strongly agree
10. **Do you know how to play chess?**
   - ☐ No
   - ☐ Yes, but barely
   - ☐ Yes, but I could use some practice
   - ☐ Yes, I am quite good at chess
   - ☐ Yes, I am a grandmaster
11. **What are the positive features of Tribler-G?**

   ____

   ____

   ____

12. **What are the negative features of Tribler-G?**

   ____

   ____

   ____

13. **How would you improve Tribler-G?**

   ____

   ____

   ____

14. **Would you use Tribler-G again? If not, please explain why.**

   ____

   ____

   ____