Peer-to-Peer Networking and Applications (2020) 13:137–150
https://doi.org/10.1007/s12083-019-00725-3

P2P matchmaking solution for online games

Michał Boroń[1] · **Jerzy Brzeziński**[1] · **Anna Kobusińska**[1]

Received: 25 September 2017 / Accepted: 30 January 2019 / Published online: 16 February 2019

Abstract

Matchmaking players is an important problem in online multiplayer games. Existing solutions employ client-server architecture, which induces several problems. Those range from additional costs associated with infrastructure maintenance to inability to play the game once servers become unavailabe (due to being under Denial of Service attack or being shut down after earning enough profit). This paper aims to provide a solution for the problem of matchmaking players on the scale of the Internet, without using a central server. In order to achieve this goal, the SelfAid platform for building custom P2P matchmaking strategies is presented. After the developer creates a service algorithm defining the matchmaking behavior specific to his/hers case, the SelfAid platform designates a number of player machines to execute the service. Furthermore, the number of designated machines adapts to the demand. SelfAid uses only spare resources of player machines, following the trend of sharing economy. A distributed algorithm is presented and its correctness is proven.

Keywords P2P · Games · Resource management · Large-scale distributed systems · Matchmaking

1 Introduction

Video games are a popular form of entertainment. In January of 2018, Steam, one of the most successful gaming platforms, hosted as much as 18.5 million concurrent users [1]. Video games are also appealing to business. According to [2] worldwide PC game market was estimated to be worth $36 billion in 2016. The market is composed of many types of games. Some of the most popular and widely recognized categories include: simulation, strategy, action, role-playing, fighting, adventure, puzzle [3–6].

Although game genres significantly differ from one another, many games have one thing in common: they can be played between many players. Games, which additionally can be played over the Internet, as opposed to playing on a local network or one machine, are called online multiplayer games. For example, ”Counter Strike: Global Offensive” is a successful (about 400,000 concurrent players) online multiplayer game, belonging to the action genre.

Developers of online multiplayer games are not usually trained in networking, yet need to make their game work over the Internet. Popular game engines [7–11] make it easy, by providing high-level features such as distributed object management or state synchronization. While these solutions perfectly abstract networking concerns, they rely on a client-server model, which implies additional costs for players.

Let us consider a company which wants to release a game in the client-server architecture. In order to create a global gaming experience, by connecting players all over the world, the company needs to provide publicly accessible servers. They may decide to invest in their own infrastructure. This would mean building a specialized server room, hiring administrators, buying hardware. What is more, they would have to over-provision servers to be able to handle load peaks. Another possible way is to move infrastructure to the cloud. Whether running on a private infrastructure or in the cloud, servers generate maintenance costs. The price is ultimately paid by players in monthly fees or watching advertisements. In addition, this model makes it very difficult to release free multiplayer games. Also, when

✉ Michał Boroń
mboron@cs.put.poznan.pl

Jerzy Brzeziński
jbrzezinski@cs.put.poznan.pl

Anna Kobusińska
akobusinska@cs.put.poznan.pl

[1] Institute of Computing Science, Poznan University of Technology, ul. Piotrowo 2, 61-131 Poznań, Poland

people who released the game shut down servers, players will not be able to play it anymore.

Recently the idea of "sharing economy" emerged, informally defined as "a socio-economic ecosystem built around the sharing of human and physical resources. It includes the shared creation, production, distribution, trade and consumption of goods and services by different people and organisations" [12–14]. A recurring pattern present in several successful "sharing economy" based companies is making use of un- or under- used resources. Airbnb [15] built its business model on renting spare housing space. Uber [16] takes advantage of underutilized cars by proposing a taxi-like service. BlaBlaCar [17, 18] makes use of empty car seats on longer trips.

Multiplayer games may also profit from "sharing economy". User bandwidth and computing power may be thought of as an underused resource. While playing a game at home, CPUs and bandwidth are not utilized completely, except for some extremely resource-hungry games. Sharing bandwidth with others does not impose additional costs, as most Internet Service Providers (ISPs) offer unlimited data plans to domestic users. By making use of these resources, it is possible to develop multiplayer games and charge players only for game content. Thus making games more appealing to the customers by lowering the overall price.

This goal may be achieved using the P2P model [19–21]. P2P is a decentralized communication model, where each participating node has the same rights (there are no "special" nodes). The term peer-to-peer (P2P) refers to direct communication between parties with equal rights. Some people incorrectly assume that fulfilling this criterion is enough for any system to be called P2P. What really brings together systems called P2P are the goals they aim to achieve and benefits associated with them. One of the goals is shifting the balance of computation from central servers to regular, personal computers. Another goal is to use otherwise unused distributed resources such as computing power, storage, network bandwidth. Common benefits include scalability and eliminating the need for expensive infrastructure such as servers and specialized networks.

Expecting that some work in applying P2P model to the area of computer games has been done already, an analysis of literature related to the problems of P2P multiplayer games was performed. The majority of reviewed articles presented solutions for either synchronizing the game state [22, 23] or contributed to other elements directly associated with playing the game [24, 25]. However, there is a research subject other than those two, which becomes apparent after examining the shortcomings of the reviewed articles. [23] did not take issues of trust and safety into account and a central directory service had to be used in order to find other players. [22] assumed a static division of the game world and a coordinator for each game region. [25] put a severe limitation on the size of playing group and did not discuss how players find opponents. Solution described in [24] generates a lot of network traffic, because each action is broadcasted to all players in the region and a byzantine voting is frequently performed.

The crucial observations include the fact that [23] had to use a central directory service in order to find other players and [25] did not discuss how players find opponents. Both works left out the problem of matchmaking players in a P2P environment. Matchmaking in multiplayer video games is the process of connecting players together for online play sessions. The primary task of a matchmaking system is to find another person willing to play. However, existing matchmaking systems, which function in the client-server model [26], may connect players in many ways. The most simple case is random matchmaking, in which players subsequently connecting to the server play with each other. Another way is to allow players to create named game instances. Since they are publicly visible, other players may see all of them and join a game instance of their choice. It may also be the case that a global ranking system exists (such as ELO [27], Trueskill [28]) and that players with similar ranking scores are chosen to play together. Players may be connected based on other criteria such as age or country as well.

Many matchmaking strategies compare multiple possible matches and choose an approximately satisfying result. For example, when searching for one opponent based on ELO ranking and knowing that it's desired for ranking difference between players to not exceed 100. We still would choose to match players with 120 ranking points difference if no better matches are available after, say, 1 minute — as players are unwilling to wait longer. To be able to compare multiple possible matches it's necessary to gather player requests.

In a P2P setting comprised of end-user machines it's necessary to take churn into account (ongoing process of new nodes joining and leaving the network). Thus, we propose to direct player requests to a group of nodes which will process it. Concretely, we propose a failure resistant group structure, which enables efficient workload division among its members. The structure elects one node as a leader. Furthermore, we describe a design of the system composed of multiple groups and show how to use it to implement custom matchmaking strategies.

This paper aims to provide a solution for the problem of matchmaking players in multiplayer games on the scale of the Internet, without using a central server. Implementing all of the aforementioned strategies in a single library would be a daunting task. Instead, in the paper we propose a P2P platform — the SelfAid network, which facilitates the process of setting up those strategies in a P2P environment, and ensures their appropriate instantiation, load balancing and replication.

With the use of the proposed solution, a game developer can obtain a matchmaking system for his game by providing only a specific matchmaking strategy (further in the paper called a service algorithm). The SelfAid platform is responsible for creating (at runtime) processes running the defined service algorithm at the machines of some players. Furthermore, it finds (provides an IP address of) a process when a player wants to find an opponent. The traffic to processes is subject to load balancing. What is more, if load on the node running the process proves to be too much, the SelfAid network creates additional processes. This ensures that the nodes of SelfAid network use only their spare bandwidth. On the other hand, if the load on processes is too small (players have to wait for a long time), some instances are deleted.

Besides introducing the SelfAid network concept, the paper also discusses the correctness of the proposed solution — all guarantees exposed by the system described in the paper are formally defined and proven.

The paper is structured as follows. Section 2 elaborates on the literature related to the problems of matchmaking players in P2P multiplayer games. Section 3 describes the assumed system model. The conceptual project including functional and non-functional requirements, as well as detailed description of the Self-aid network is presented in Section 4. Section 5 contains description of a proof of concept implementation of the proposed system and an example of a matchmaking service. Finally, conclusions and further work are presented in Section 6.

2 Related work

This chapter describes several works related to matchmaking in P2P multiplayer games.

In 2009 Microsoft research released an article describing a latency prediction system called Htrae [29]. Before, there existed strategies for minimizing latencies experienced by players based on either geolocation data or direct latency measurements. Htrae excels at combining both of those strategies into one. It works similarly to a network coordinate system called Vivaldi [30], that is, it assigns to each peer coordinates in a virtual spherical space (in Vivaldi the space was non-spherical). The clue of the system is that the initial positions in the virtual space are dictated by geolocation data but they are later adjusted based on latency measurements. This way the distance in virtual space can be used to predict latency between any nodes in the system. The system provides means to predict latencies between players machines, but it is not a complete matchmaking solution ready to use by game developers. Htrae was designed with P2P games in mind, however the system itself is not free of central components such as GeoIP database or a routing table service for AS correction.

Switchboard [31] is a P2P matchmaking system for mobile devices. Switchboard puts emphasis on scalability, as well as ease of use for game developers. It exposes a cloud service with a simple API for game developers, allowing them to define a few basic criteria for matchmaking. The available criteria consist of: tolerance for latency, number of players, the exact same requirements (e.g. play game ”x” on map ”y”). The main contributions include elaborate methods of predicting latency components specific to cellular networks, the architecture of the cloud service for processing matchmaking requests, and the API for game developers. For latency prediction not specific to cellular networks, the authors decided to use previously described Htrae. Similarly to Htrae, Switchboard is a system for P2P games, but is itself a non-P2P system. Additionally the matchmaking criteria available to game developers are not sufficient to connect players based on a ranking scheme.

Both previously described works focused on reducing latency between players, as the most important problem in matchmaking. The authors of the next work present another approach to matchmaking, focusing on providing the best user experience by using additional data about players to prevent hostile situations during gameplay. The article [32] presents an idea for a matchmaking system for the game ”League of Legends” which takes into account the inner mechanics of the game. The game is played as a match between two teams. Each team is created from players chosen (to some degree randomly) by the matchmaking system. Once the team is formed, each player is allowed to choose a character whom they will be playing during the match in order of their ranking (the best player chooses first). Each available character corresponds to a different role and style of play and can be chosen only by one player. The combination of the aforementioned factors results in the dissatisfaction of lower ranking players when they are forced to choose a character they do not like. The idea described in the article is to keep track of the players preference to specific characters and then compose teams of people with non-overlapping preferences. This article shows how case specific matchmaking rules may be required for a good user experience.

LOM [33], presents an attempt to create a general matchmaking system, using an abstract *association* criterion. The goal of LOM is to group players into independently running game sessions comprising of a certain amount of players. In order to achieve this, a leader for each group is chosen (e.g. randomly). The problem is then transformed into a minimum-cost flow problem where the source is connected to a layer of leaders, which in turn is connected to a layer of members (all players who are not leaders), which converges in target node. The weights on the arcs between leaders

and members correspond to the association criterion (which is defined by game developer as e.g. difference in ranking score). It is an efficient design of a server-based flexible matchmaking system.

Each of the mentioned works focuses on a different matchmaking strategy and presents a concrete solution for the considered problem. The aim of this paper is to provide a more flexible solution. Instead of focusing on a single matchmaking strategy, a platform supporting any matchmaking strategy is proposed.

3 System model

The *SelfAid* system described in this paper is a decentralized and structured P2P network. It does not feature a central node (like Napster [34]) or supernodes (like Kazaa [35]). Instead, all nodes in the system have equal roles — they perform the same tasks and have the same rights. As all structured P2P networks, the SelfAid has a precisely defined set of rules on how nodes should choose with whom they establish and maintain connection. This set of rules is called a Distributed Hash Table (DHT) [36] and allows efficient resource location. Following the trend of sharing economy, SelfAid network assigns only as much work to a particular node as it can handle using spare resources.

SelfAid network is a synchronized distributed system, which means that the Upper bound Transmission Time of messages [37] is assumed to be known (further abbreviated UTT). All algorithms presented in this article assume perfect links (messages are always delivered) and FIFO channels. Also, it is assumed that failure detection takes at least UTT. Furthermore, the code is executed in a single thread to avoid issues related to concurrent modification of variables. To be able to predict the arrival time of a message from another node, computation is assumed to be instantaneous (it cannot cause a message to be delayed). The code in all algorithms contains no blocking operations. The considered failure model is crash-stop (halt failure) [38].

SelfAid network operates under the assumption that each node knows the names (unique identifiers) of all service algorithms. Once again, a service algorithm is a specific matchmaking strategy implemented by the game developer. A game developer may define many service algorithms. A service algorithm is required to be stateless (SelfAid does not provide mechanisms to ensure data consistency across processes). Furthermore, each node is able to run any of the defined service algorithms when asked by the SelfAid network. In order to do this, a node has to have the code or executable of the service algorithm. One way to achieve this would be for all nodes to get the code or executables of service algorithms beforehand. For example, the code may be embedded in data downloaded in order to play the game. SelfAid network is responsible only for creating processes and providing contact information to reach them, so clients have to know the appropriate protocol to communicate with the service.

4 Concept

The goal of this paper is to provide a platform for building case-specific matchmaking strategies. The developer implements one or more service algorithms (a server program for a concrete matchmaking strategy). SelfAid automatically creates processes of a given algorithm, each running on a separate machine. SelfAid also makes the processes available to clients. For a given algorithm, a client receives addresses of all processes and picks one at random, then sends a request to it. How this request is handled is determined by the algorithm implemented by the developer.

Processes executing the same algorithm A form a group G_A. Each group G has one leader, which we name the coordinator C_{GA}. A coordinator is responsible for (I) publishing the contact information of the processes comprising the group, and (II) for recruiting new members to lessen the load on all processes. Matchmaking requires accumulating user requests in one place. To ensure that there is only one group for a matchmaking algorithm, a rule was established. If there are concurrent groups for the same algorithm, the one which runs longer will prevail. In case of ties, the conflicts are resolved deterministically based on the id of the node which was the first member of the group. This node is called the original node

The rest of this section is divided into subsections. First subsection contains a description of the ring structure, consisting of nodes running an instance of a particular service algorithm. Variables stored at the nodes are listed as well. Second subsection discusses how a process running a particular service algorithm can be found and defines the concept of an announcement. Third subsection presents the process of adding new members to the ring structure. Fourth subsection describes when and how nodes are removed from the ring structure. Fifth subsection explains how failures in the ring structure are handled.

4.1 Ring structure

In order to be able to detect failures, instances are organized in a ring structure, as in the Fig. 1. Each node monitors the state (up or down) of one other node which is called the parent. Since the coordinator may fail, its responsibilities have to be transferred to another node in case of failure. The responsibilities include publishing announcements, checking whether they are reachable by clients, adjusting the number of nodes in the ring. In the beginning this role

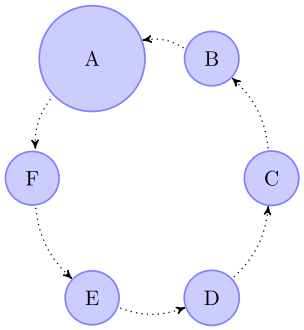

Fig. 1 Processes from a group G_A form a ring structure. Arrow points at parent

is fulfilled by the original node. Later, the node which detected the failure of a ring coordinator becomes the new ring coordinator. In figures, the node fulfilling the role of ring coordinator is distinguished by a bigger size of the node circle. In Fig. 1, node A is the ring coordinator.

This solution makes the ring more robust in case of failures because every node knows about all other nodes in the ring and their order. It is reasonable, assuming that the number of nodes comprising a particular ring grows linearly with a small constant (as it should be in case of matchmaking for games).

4.2 Service algorithm lookup

Service algorithm lookup is the task of finding a process P_A given the identifier of algorithm A. Processes are located by retrieving from the DHT an announcement stored under the key corresponding to the A identifier. The announcement contains, among other things, a list of nodes hosting processes running A. In order to ensure proper load balancing, an instance is chosen from the list randomly, with uniform distribution of probability. If no announcement is found, a process is created at the node which issued the request.

As shown in Table 1, an announcement contains the name of the service algorithm, the list of contact information to the nodes hosting a process running that algorithm, running time of the ring, hash of the original node identifier, hash of the service algorithm name.

It is possible that many nodes simultaneously tried to lookup a service algorithm which was not run by anybody

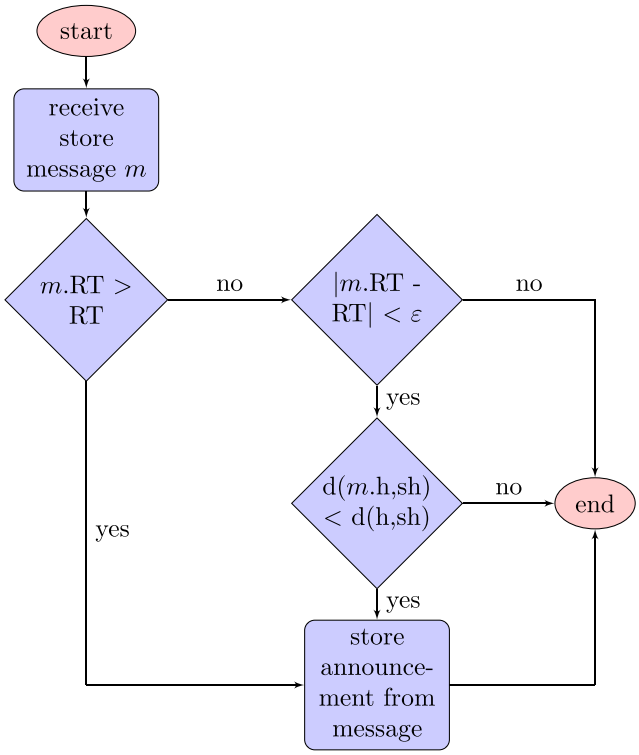

Fig. 2 Processing a request to store an announcement. d(x,y) — function of distance between two hash values

and ended up creating processes unaware about each other. In order to deal with this situation, the announcement contains additional information. Announcement contains hash of the original node's identifier, hash of service algorithm name and running time — the time which elapsed since the moment in which the original node started its process. A node which received request to store an announcement stores it if it contains a higher value of running time than the one currently stored. If it is impossible to tell which process is running for a longer time, due to networking delays (the difference in running times is smaller than parameter ε), the announcement with the hash value of the original node identifier closer to the service name hash value is stored. The nodes publishing announcements must periodically check if their announcements are stored. If not, they should shut down because no clients will contact them. The process of storing an announcement is visualized in Fig. 2 and its

Table 1 Structure of an announcement

Field	Type	Comment
Service algorithm name	String	The name of provided service
instances	List of contact information (IP)	Contacts to running instances
RT	TimeSpan	Running time
h	Hash	Hash of the original node identifier
sh	Hash	Hash of the service algorithm name

parameters are shown in Table 1. To prevent spreading outdated information, an announcement is stored for a limited amount of time (given as parameter RA) and then deleted. To keep them available, announcements must be stored periodically in the DHT by the ring coordinator.

Coordinator periodically checks if multiple rings are operating. In order to do so, it retrieves an announcement associated with its service algorithm name from the DHT. Then, it checks if the retrieved announcement comes from a different ring by checking the hash of original node. It decides to shut down the ring if the running time stored in the announcement is longer than the running time stored by coordinator. If a ring is to be shut down, a `SNOTIF` is broadcasted. Then, `SHUTDOWN` is broadcasted, but in reverse order. Finally, the coordinator shuts down. When a node receives `SNOTIF`, it remembers to rebroadcast `SNOTIF` if it becomes the coordinator. When a node receives `SHUTDOWN`, it shuts down. Shutting down the ring is illustrated by Fig. 3.

4.3 Ring construction

Before the functioning of the system is explained, the variables representing state of each node are described here. The `local ring view` is the representation of the ring — which nodes are in it and how they are connected. Each node knows the address of `parent` — the node whose state it is supposed to monitor (watch). The ring coordinator watches the youngest node in the ring or a node trying to become the youngest node in the ring. Since any node may become ring coordinator, all nodes have to remember the nodes which they may have to put in the announcement when they become the coordinator. The `sname` (service algorithm name) is a string identifying the service algorithm which all nodes in the ring provide. Other values needed for the announcement are `orig` (the address of original node) and `rt` (running time) — the time which elapsed since the original node created the process. `accRt` is the snapshot of running time value stored in the message received when the node joins the ring (or 0, for original node). Since nodes may fail, sometimes it is necessary to resend messages. `DS` (dead nodes), `detected` (detected failures) and `newNode` are used for this purpose.

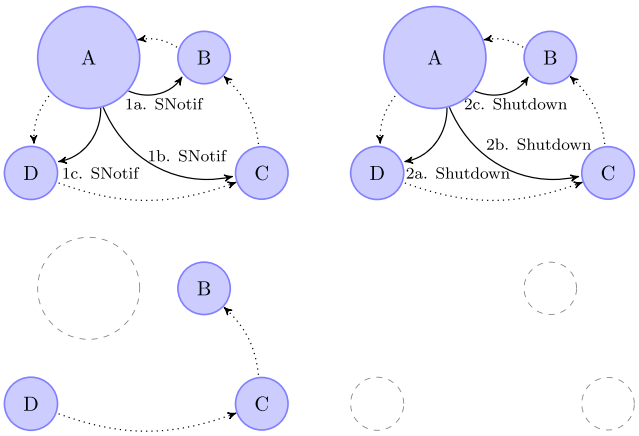

Fig. 3 An example of shutting down the ring

Algorithm 1 Key events in the SelfAid platform.

```
1:  recruiting a new node at coordNode
2:      newNode ← get a free node
3:      send PARENT to newNode
4:      parentNode ← newNode

5:  upon receiving PARENT at newNode
6:      launchServiceProcess()
7:      send LAUNCH_CONF to PARENT.sender

8:  upon receiving LAUNCH_CONF at coordNode
9:      //notify the whole ring about new
10:     bcast(NNOTIF(new))

11: upon receiving NNOTIF at newNode
12:     parentNode ← coordNode
13:     become established

14: when parentNode failure detected at newNode
15:     if next coord candidate = NULL then
16:         shutdown
17:     end if
18:     parentNode ← next coord candidate
19:     send NEW_QUESTION to next coord candidate

20: upon receiving NEW_QUESTION at coordNode
21:     if new is unknown then
22:         send Shutdown to m.sender
23:     end if
```

The number of processes depends on the demand for the service algorithm (bigger demand means more instances). When load becomes too big, the coordinator recruits a new node from the nodes present in the system (l. 1-4). The coordinator is able to estimate the load on ring nodes based on local load measurements, since a client picks a ring node from the list received in the announcement with uniform distribution of probability. A `PARENT` message is sent to the new node, containing the addresses of all nodes in the ring, the current value of running time and the name of service algorithm. After sending the message, coordinator starts to watch the new node (by making it the parent node). When the new node receives `PARENT` message (l.5-7), it launches a process and copies values embedded in the

message. The list of nodes from the message is copied to local ring view. Once the process of joining the ring is over, the newly added node watches the previously added node (the ring structure is sorted according to the joining order). However, at this moment, the new node starts to temporarily watch the coordinator. This ensures that all other nodes in the ring know about the new node before it can become the coordinator. If it would immediately start to observe the last node of the ring, it is possible that a node which does not know about the new node, would become new coordinator and would create an additional new node. This situation would be indistinguishable from two nodes being added concurrently and carries the same risk of both of them becoming coordinator at the same time. Finally the new node sends `LAUNCH_CONF` message to the coordinator.

When coordinator receives `LAUNCH_CONF` message (l.8-10), it broadcasts a `NNOTIF` with the address of the new node.

The broadcast reaches also the new node, to inform it that it is acknowledged by all nodes in the ring. When last `NNOTIF` was sent, coordinator knows that all nodes will receive it (perfect links) in maximum UTT time (known upper bound transmission time), so it recognizes the new node as a normal node in the ring. Then it broadcasts `NCONF`, so that nodes do not resend notification (because all nodes already got it). When a node receives `NNOTIF`, it remembers the address of new node (in case it has to be resent later). If the node which received the message is the new node (l.11-13), it starts to watch the last node in the ring and considers itself a recognized member of the ring. All other nodes recognize the new node as a part of the ring. When a node receives `NCONF`, it will not broadcast `NNOTIF` notifying other nodes about the new node when it becomes the coordinator. Adding a new node to the ring is illustrated by Fig. 4.

There are two reasons for the new node being added to the ring as the parent of ring coordinator. First, detecting node can take the place (watch next node or become coordinator) faster because some time is needed for the free node to be recognized by other nodes in the ring. The speed of assuming the role of the failed node is especially important if the failed node was the ring coordinator. The longer the time without the ring coordinator, the bigger the possibility that some clients requesting contact information to processes will not be able to receive it. Because the ring coordinator is responsible for publishing announcements which are necessary to contact the processes. Second, by doing it this way, the node publishing announcements is always the oldest node in the ring (the one who is running the process for the longest period of time). It is a beneficial property, assuming that the nodes operating longer exhibit smaller probability of failure.

4.4 Node removal

If coordinator detects that load became too small, a node is removed from the ring. First, a `RNOTIF` containing the address of the youngest node is broadcasted. Then, the same node is removed from the ring view of the coordinator. Finally `RCONF` with address of removed node is broadcasted.

When a node receives `RNOTIF`, it first checks whether it is the removed node or not. If it is, it shuts down. If not, it remembers the address of the removed node (in case it has to be resent later). Then, the removed node is deleted from the local ring view. When a node receives `RCONF`, it will not resend information about the last removed node. Removing a new node from the ring is illustrated by Fig. 5.

4.5 Failure handling

When a failure occurs, the node which detected it (called the **detecting node**) starts watching the node which was previously watched by the failed node. For example, in

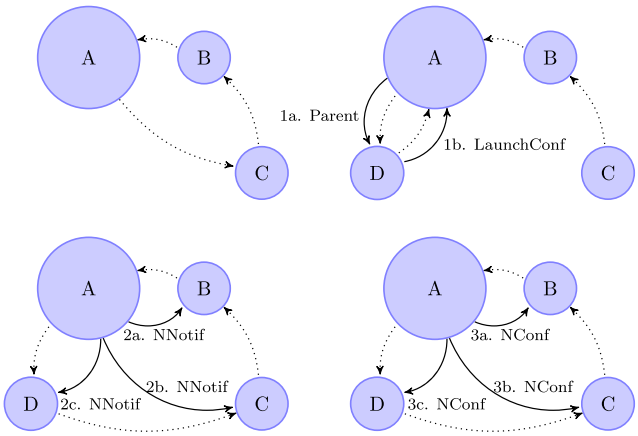

Fig. 4 An example of adding a new node to the ring

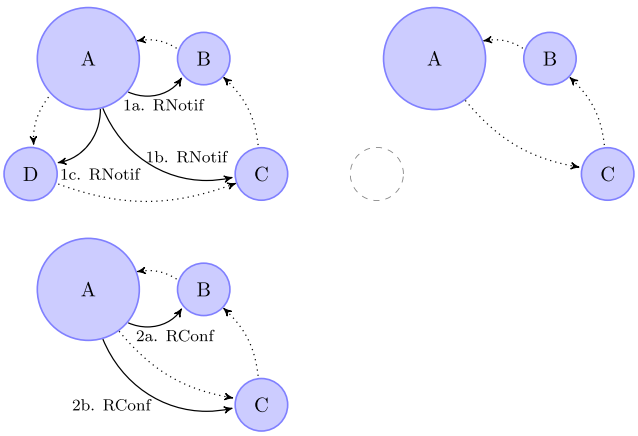

Fig. 5 An example of removing a node from the ring

Fig. 6 node D fails and E starts watching C. Additionally, if the failed node was the ring coordinator node (A in this example), the detecting node (B) would become new ring coordinator. The detecting node notifies coordinator of failure of the failed node. Then coordinator notifies all nodes in the ring.

When a failure is detected, taken actions depend on the type of failed node and the role of detecting node. The first case is the new node detecting failure of its parent (l.14-19). In this case the failed node had to be the coordinator. Since detecting node was still not established, the coordinator had to fail before it sent `NNOTIF` to the new node. Detecting a failure takes at least UTT and any message sent by coordinator had to be sent before failure so it would arrive before the failure detection procedure was executed. If the new coordinator did not receive `NNOTIF` from previous coordinator, it will create a new node to compensate for the failed coordinator. In this case, the new node will shut down to avoid scenario with two concurrent coordinators. If the new coordinator did receive `NNOTIF` from previous coordinator, it will resend the notification again and if it does not fail in the process, the new node will become part of the ring. Upon detection of parent failure, the new node removes it from its local ring view and starts to watch next coordinator. If the coordinator and all nodes between it and the detecting node have failed (ring may still be functioning with nodes unknown to this new node) then new node shuts down. If the coordinator or any of the nodes between it and the new node is alive, the new node sends `NEW_QUESTION` message, containing its contact information, to the coordinator. When coordinator receives `NEW_QUESTION` (l.20-23), it checks if the new node is present in its local ring view (it would be if coordinator received `NNOTIF` from previous coordinator) and if it is not there, replies with `SHUTDOWN` message. When new node receives `SHUTDOWN` message, it shuts down.

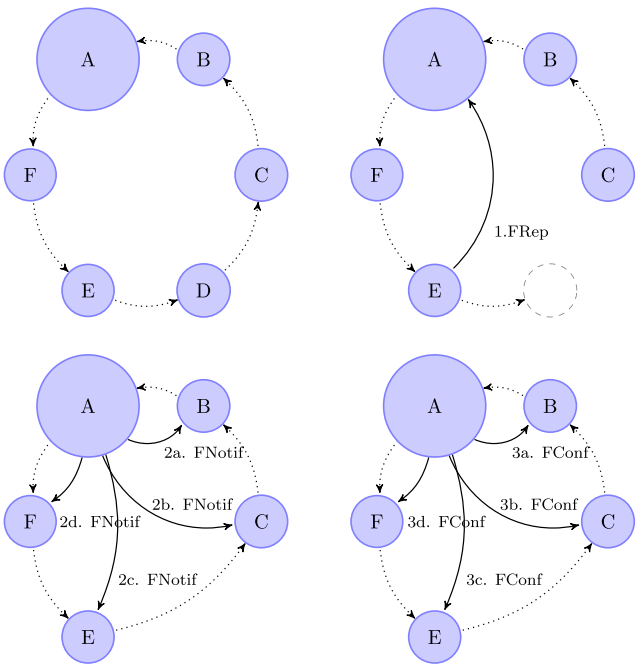

Fig. 6 An example of failure

The second case is failure of any node that is neither coordinator nor the newest node in the ring. The failed node is added to the set of detected failures, called ”detected”. Then the failed node is removed from the local ring view. Next the detecting node sends `FREP` (fail report) message to the node it thinks is the current coordinator and starts to watch the next node on the ring. When the coordinator receives `FREP`, it removes all nodes mentioned in the report from its local ring view. Then the coordinator broadcasts `FNOTIF` and `FCONF` messages to all other nodes in the ring. The first message is meant to inform other nodes of failures, the second prevents rebroadcasting contents of the first when other nodes take over the coordinator role. Rebroadcasting is needed because based on receiving `FNOTIF` it is impossible to tell if other nodes received that message too — coordinator may have failed during the broadcast procedure. When `FCONF` is sent, it is guaranteed that `FNOTIF` was sent to all nodes. Since the assumed model features perfect links and known Upperbound Transmission Time, the sent messages will arrive to destination in at most UTT (even if sending node fails). Therefore, after a node receives `FCONF`, it will not broadcast information from preceding `FNOTIF` when it becomes coordinator. What is more, a node may become coordinator only after (minimum) UTT time passed from previous coordinator failure. So, it is guaranteed that any `FNOTIF` messages sent by previous coordinator already reached all nodes in the ring. When a node receives `FNOTIF`, it removes nodes mentioned in the message from detected. Then it adds the same nodes to DS (dead nodes) set. Nodes in DS are sent to all nodes when a node becomes coordinator. Nodes in detected are sent to the coordinator in `FREP` when parent failed or coordinator changed (it contains all failed nodes whose failures were detected directly by this node because they may be not known to coordinator). Next, if notification contains the node which was until now considered coordinator the node sends unconfirmed nodes in detected to new coordinator. Finally failed nodes are removed from the local ring view. When a node receives `FCONF`, it removes mentioned nodes from DS (dead nodes) list.

The third case is the failure of the current coordinator. The node taking over starts to fulfill the duties of coordinator by starting to execute periodic actions, specific to coordinator. First, the new coordinator checks if there is an unconfirmed new node and rebroadcasts information

about it if needed. If the previous coordinator issued the request to shutdown the ring, the new coordinator broadcasts first SNOTIF, then SHUTDOWN (the second one in reverse order) and finally it shuts down. If SHUTDOWN would be broadcasted in the "normal" order and coordinator would fail during the process, the node closest to becoming coordinator from the remaining nodes would have to detect failures of all nodes which received the confirmation message. For example consider a ring with coordinator A watched by B, followed by C,D,E,F If A fails before sending SHUTDOWN message to E, then B,C,D will shut down and E will have to spend at least 4*UTT time before overtaking coordinator role and rebroadcasting. If the confirmation message is sent in reverse order, even when coordinator fails during the process, the nodes which received SHUTDOWN do not delay the process of overtaking the coordinator. The condition to rebroadcast shutdown notification is checked before condition for removed but after eventual resending of new node. This way, new node does not have to check if every other node has failed. Then, it checks if information about a removed node has to be rebroadcasted. If yes, the new coordinator broadcasts RNOTIF with the address of the removed node. Next, RCONF is broadcasted to avoid needless rebroadcasting in the future. Next, it broadcasts FNOTIF and FCONF with nodes from both sets of detected and DS. Since the node became coordinator by detecting failure of previous coordinator, the detected set would contain at least the address of the old coordinator. Handling a failure of a node is illustrated by Fig. 6.

The fourth case is failure of the new (not yet established) node (only coordinator watches the new node). The failed node is removed from the local ring view of the coordinator and the failure is broadcasted in FNOTIF and FCONF messages. Finally, the coordinator starts to watch the last node in the ring.

The most important thing is for the coordinator to remove dead nodes from its local ring view as fast as possible because it puts nodes from its local view on the announcement (putting a failed node on the announcements risks that a client will not be able to access the service algorithm). Another important issue is that nodes closest to becoming next coordinator have to be notified about the failures first so that they can overtake the coordinator responsibilities faster. To explain the issue, let's consider an example with coordinator A watched by B, followed by C,D,E. If B and C fail, D is not notified and then A fails, D will have to wait at least UTT to detect failure of C then the same time to detect failure of B and again for A. In worst case D detects failure of A and becomes coordinator after 3*UTT. If it would be notified of B and C failures, it would have to wait only 1*UTT.

5 Analysis

5.1 Adjusting time parameters

In order to properly adjust time parameters, some upper bound times have to be known. Let dhtPUT be the upper bound on time needed for a dht.put(...) request to reach destination. Let dhtGETreq be the upper bound on time needed for a dht.get(...) request to reach destination. Upper bound on time needed for response to dht.get(...) to reach the requester will be denoted by dhtGETresp.

There are multiple time parameters associated with the algorithm. ε is used to check if the values of running time of two announcements are so close to each other that it is impossible to tell which is running longer. Let's call the time intervals in which coordinator checks consistency CA. The time interval in which an announcement is published by coordinator will be denoted as PA. RA is the time after which a stored announcement is removed from DHT.

ε should be bigger than DHTput + DHTgetReq + DHTgetResp + CA to compensate for the time between the moment one ring publishes an announcement and another retrieves it. CA influences how fast two concurrent rings notice each other. There are no special considerations for this parameter. PA determines how fast changes in the list of functioning instances are propagated to DHT, which has direct impact on user experience. However, picking a short PA may cause the coordinator to use substantial amount of upload bandwidth due to big size of published announcement. RA determines how fast an announcement is removed. If it is too short, nodes will not be able to access instances of a running ring. If it is too long, nodes will have to wait longer to detect that a ring is not running.

5.2 Proving correctness

This section proves that system functions correctly. Correct functioning of the system is composed of the following claims:

- announcements published by a ring include all properly functioning ring nodes.
- dead nodes eventually are not included in published announcements.
- as long as there are nodes in the ring, an announcement will eventually be published.
- if there are multiple competing rings, eventually only one of them remains.
- if load on processes is too big, it is eventually reduced.
- if load on processes is too small, it is eventually increased.

To make proofs more concise, a new term is introduced. Ring nodes are defined as nodes which received PARENT message and did not crash.

Lemma 1 *When a node becomes coordinator, all messages sent by previous coordinator to any node are already received.*

Proof For the statement to be false, a message sent by previous coordinator (before it failed) would have to be received after a node became new coordinator. A node may become coordinator only when it detected failure of previous coordinator, which takes more than UTT time. That implies that the moment of receipt would have to be more than UTT later then moment of sending. It is impossible because according to assumptions each message is delivered before UTT time passes from sending. □

Lemma 2 *If a node became coordinator and did not receive a broadcasted message m, assuming that new nodes are added to the ring atomically (either all other add the new node to their local view of the ring or none of them do) and in linear order (in order in which coordinators send PARENT messages) , no other ring node received (or will receive) m.*

Proof Messages are broadcasted in the order of node appearance in the ring (message is first sent to the node watching coordinator). If another node in the ring would receive m, it would mean that previous coordinator sent it also to the node which became new coordinator. It is impossible, because it would have to be received by new coordinator (lemma 1), which is contradictory to the statement. □

Lemma 3 *Assuming that new nodes are added to the ring atomically (either all other add the new node to their local view of the ring or none of them do) and in linear order (in order in which coordinators sent PARENT messages), the new node knows about all other nodes in the ring and their order.*

Proof From the assumptions, every node in the ring has the same view of which nodes are in the ring and in what order they are connected. Hence, coordinator also has this knowledge and can properly relay this information to the new node. □

Lemma 4 *Assuming lemma 3, failure of any node in the ring is eventually detected.*

Proof When new node receives NNOTIF it starts to watch the node it considers the youngest after itself (the previous new node). Local ring view contains addresses of all nodes sorted by age which proves that all nodes except for the youngest one either are watched or will be watched after a finite amount of time. Additionally the youngest node (the last recruited new node) is watched by the oldest node (coordinator). When a node dies, its failure is detected by the node which was watching it and the detecting node starts watching next node on the ring or the youngest node if it becomes the coordinator. Since each node in the ring is eventually watched by another node, its failure will be detected. □

Lemma 5 *Assuming lemma 3, if coordinator fails, it will be eventually replaced.*

Proof Since failure of any node is eventually detected (lemma 4), failure of coordinator will be eventually noticed by some node. When a node detects coordinator failure it will become the coordinator. □

Lemma 6 *Adding a new node to the ring is an atomic operation. When it terminates, either all nodes recognized the new node (success) or no correct ring node recognized or will recognize it (abort). Operations of adding a new node are applied in linear order (order in which coordinators sent PARENT messages to new nodes).*

Proof **Initial State**. In the beginning there is only one node in the ring. In this case, adding a new node is trivial. Original node adds the newly recruited node to its local ring view and, since it is the only node in the ring, operation ended in success.

Induction step. If coordinator fails before starting to broadcast NNOTIF, the operation aborts. NNOTIF can be rebroadcasted only when next coordinator knows about the new node. To know about the new node it would have to receive notification sent by previous coordinator (which was not sent).

If the coordinator finishes the broadcast, all nodes will recognize the new node (the coordinator knows the addresses of all other nodes in the ring, channels are perfect). Moreover, it will be accepted before another new node will be accepted. If another new node is added to the ring by the current coordinator, the notifications will travel through the same channels as previous notifications and channels satisfy FIFO property. If another new node is added to the ring by a different coordinator, it has to happen after more than UTT from the moment of sending notification by dead coordinator, which is enough for the message to arrive to destination.

If coordinator fails during the broadcast of notification, operation continues. Based on lemma 5, coordinator will be replaced. Furthermore, from lemma 2, the new coordinator will immediately know if there is a pending operation of

adding new node (if it received a notification). If operation is pending, new coordinator will broadcast the notification again. If it fails during broadcast, the situation looks exactly the same as in the case of previously discussed failure. If broadcasts succeeds (messages are sent), the coordinator may be sure that all nodes will receive this notification (assumption of perfect links) before any other `NNOTIF` messages. If the same coordinator sends next notification the last statement is justified by FIFO channels, else if next notification is sent by another coordinator lemma 1 holds. □

Theorem 1 *The announcements published by ring coordinator include all properly functioning ring nodes (at the moment when they were issued).*

Proof The announcement is published by coordinator, who adds to it all nodes in local ring view. The action of publishing happens only when no adding new node operation is pending. Since a node is added to the ring only when all nodes in the ring are aware of its presence (lemma 6), the local ring view contains all properly functioning and established ring nodes. □

Lemma 7 *Failure of ring coordinator is eventually known by all nodes.*

Proof When a coordinator failure is detected, the detecting node becomes new coordinator and broadcasts `FNOTIF`. If it fails during broadcast, next coordinator will broadcast notification again either because it received notification or because it detected the failure of previous coordinator. If it does not fail during broadcast, all nodes will know about coordinator failure after UTT (all messages are received before UTT). □

Theorem 2 *Dead nodes eventually are not included in published announcements.*

Proof When a failure is detected by node D, eventually a ring coordinator receives `FREP` or D becomes ring coordinator. When coordinator receives the report, it sends `FNOTIF` to all nodes in the ring. If a coordinator is overtaken by a node other than D, D will be notified (lemma 7). When it is notified, it sends `FREP` containing all unconfirmed failures. If D becomes coordinator (and did not receive notification with address of detected failure), it will broadcast notification. If node fails before broadcasting, next nodes will detect failures by watching failed nodes and will broadcast notification. When a notification is received, nodes in the message are erased from the local view of the ring. □

Theorem 3 *As long as there are nodes in the ring, an announcement will eventually be published.*

Proof A coordinator will eventually be replaced (lemma 5), when a node becomes coordinator it will start publishing announcements. □

Theorem 4 *If there are multiple competing rings, eventually only one of them remains.*

Proof Ring coordinator polls announcement DHT to retrieve announcement of the ring matching the name of service algorithm. If it retrieves announcement of another ring, it checks the running times differ by less than ε. If $\varepsilon >$ `DHTput` + `DHTgetReq` + `DHTgetResp` + `CA`, it is sure to compensate for all delays. If all delays are compensated for then the ring which decides to shut down will always be the one which either for sure is running for a shorter amount of time or its hash is further away from hash of service algorithm name. □

Theorem 5 *If load on processes is too big, it is eventually reduced if nodes stop to fail.*

Proof Eventually, a node becomes ring coordinator. Since load is equally distributed among instances, it can measure the load locally. Then, it creates a new instance if it detects that load is too much and measures load again. If instances are created faster than ring nodes fail, the number of instances in the ring will increase. Since load is shared equally between instances, the load on each particular service algorithm will be reduced. □

Theorem 6 *If load on processes is too small, it is eventually increased.*

Proof Eventually, a node becomes ring coordinator. Since load is equally distributed among instances, it can measure the load locally. Then, it removes a node if it detects that load is too small. If failures occur, load has to be shared between smaller number of instances which also means that load on each particular service algorithm would increase. □

6 Performance evaluation

Detailed performance evaluation along with metrics was described in conference paper [39]. The work focused on developing metrics to measure the resource cost incurred by being a part of SelfAid network, user satisfaction and resistance to failures. Developed metrics were applied to a simulation of SelfAid network, including the lower DHT

Table 2 Total number of messages in function of ring size and churn (for 1000 rounds)

	size = 10	size = 20	size = 50	size = 70	size = 100
churn = 0	19756	38858	91480	123598	166522
churn = 2	19809	38937	91868	123908	166843
churn = 4	19856	39098	92692	124683	166422
churn = 8	19921	39357	92334	124494	168383
churn = 16	20129	39647	95049	126116	169007
churn = 32	20274	39738	95381	127137	164598
churn = 64	20574	40969	96643	129659	167740

layer as well as the higher application layer. The simulation was implemented using Peerfact.Sim simulator [40].

Several metrics were developed. Overload Time Ratio measures how much time a node spent with load above average (while ring size was adjusted to demand). System Responsiveness is the average time span between issuing a request to find an opponent and getting the response. In the meantime it's necessary to lookup the Announcement and contact a member of the ring. Service Node Response Faults measures how many requests a service node received and how many were left unanswered. The behavior of metrics was checked with various load distributions of application requests and various degrees of crash-stop failures. The simulation showed that SelfAid behaved well with respect to user satisfaction (System Responsiveness metric). Resulting latency was well within acceptable levels (several seconds). Concerning bandwidth, even with modest application-level traffic, most of the bandwidth was consumed by that traffic and not due to maintenance associated with being part of the ring.

Additionally, in this article, a study on message complexity was performed and confirmed by a simulation. The simulation was implemented in Python. The messages in the simulation were delivered synchronously, in rounds. One run of the simulation was limited to 1000 rounds. The simulation focused exclusively on the properties of the ring. Concretely, total number of messages sent during the simulation was measured. Simulation was parameterized by the size of ring and level of churn - how many nodes failed. The value measured is the total number of messages sent by all nodes. The results were averaged over multiple repetitions of simulation with different random seed. The ring tried to maintain a constant, set amount of members.

Results of the simulation are summarized in Table 2, which shows the number of messages in function of ring size and churn. The value of churn is equivalent to the number of nodes that crashed and were later substituted by new nodes. Size refers to number of nodes in the ring. The number of messages that can be sent during one round is bound by a linear function (of ring size). The coordinator may send no more than a fixed amount of broadcasts in one round. The other members can send no more than a fixed amount of messages, either to parent, child or the coordinator. In the table, it is possible to see that number of messages increases with the size of the ring. E.g. for churn=4,size=10 it's 19856 and for the same value of churn but size=50, it's 92692. Message complexity growth is linear with respect to ring size. Furthermore, crashes tend to increase the amount of messages. E.g. for size=50,churn=4 it's 92692 and for the same size but churn=64, it's 96643.

7 Conclusion

The goal of this paper was to provide a tool for automatic locating and managing of processes running matchmaking algorithms in a P2P environment, along with a proof of its correctness.

The presented solution allows a player to quickly connect to others, provided that no failures occur. In this case, accessing a service algorithm is only a matter of issuing one request to announcement DHT and then one request to the process. When crash-stop failures occur, multiple things may get slower or unresponsive for some time. For example, if the nodes storing announcement fail, clients will have to wait for the ring coordinator to publish the announcement again. However, assuming that crash-stop failures happen only once in a while, the system exhibits reasonable performance. The coordinator of a ring is notified about failure in constant (constant worst case) time, which means that propagation of information about failure to the client requesting list of processes for a given service algorithm will not take long. All nodes in the system may perform any role (e.g. storing announcements, being ring coordinator) depending on circumstances. The system is able to handle (connect) as much nodes as the underlying DHT, which was designed for massive scale.

A simplified version of the proposed solution that does not take failures into account was described in [41]. The paper [41] focuses on providing implementation details for both the platform and a particular matchmaking strategy. In

contrast to theoretical analysis and proofs presented in this article.

The SelfAid could be combined with [23] to create a completely distributed online gaming framework.

Authors plan to enhance the proposed solution. The concept presented in Section 4 may be subject to improvements and extensions. Currently, the process address returned to the user by lookup procedure is picked at random, from the list of nodes contained in the retrieved announcement, with equal distribution of probability. This could be changed so that nodes which are able to handle more load (they have more spare resources) would be more likely to be picked. It would result in a more efficient system, since less nodes would have to be added to the ring of instances. Another idea is to introduce cache for announcements stored in the DHT, which would prevent overloading the node storing announcement. The delays (e.g. time to detect failure) introduced by assuming a large value of Upperbound Transmission Time are substantial. It would be interesting to try to create the ring with nodes which are close to each other in terms of latency. Finally, an algorithm with weaker assumptions on synchrony or failure detector could be designed.

References

1. Statista: number of peak concurrent steam users from november 2012 to january 2018. https://www.statista.com/statistics/308330/number-stream-users/. Accessed 15 June 2018
2. Pcgamer: Pc gaming market worth $36 billion in 2016. https://www.pcgamer.com/pc-gaming-market-worth-36-billion-in-2016/. Accessed 15 June 2018
3. Apperley TH (2006) Genre and game studies: toward a critical approach to video game genres. Simul Gaming 37:6–23
4. Heintz S, Law ELC (2015) The game genre map: a revised game classification. In: Proceedings of the 2015 annual symposium on computer-human interaction in play, CHI PLAY '15, pp 175–184. ACM
5. Hrabec O, Chrz V (2015) Flow genres: the varieties of video game experience. Int J Gaming Comput Mediat Simul 7:1–19
6. Kline S, Dyer-Witheford N, Peuter GD (2003) Digital play: the interaction of technology culture, and marketing. McGill-Queen's University Press, Montreal
7. Google Cloud Platform dedicated server gaming solution. https://cloud.google.com/solutions/gaming/dedicated-server-gaming-solution/ Accessed 12 July 2017
8. Anderson EF, Engel S, Comninos P, McLoughlin L (2008) The case for research in game engine architecture. In: Proceedings of the 2008 conference on future play: research, play, share, future play '08, pp 228–231. ACM
9. Petridis P, Dunwell I, De Freitas S, Panzoli D (2010) An engine selection methodology for high fidelity serious games. In: Games and virtual worlds for serious applications (VS-GAMES), 2010 second international conference on, pp 27–34. IEEE
10. Wang S, Mao Z, Zeng C, Gong H, Li S, Chen B (2010) A new method of virtual reality based on unity3d. In: 2010 18th international conference on Geoinformatics, pp 1–5. IEEE
11. Xie J (2012) Research on key technologies base unity3d game engine. In: 2012 7th international conference on computer science & education (ICCSE), pp 695–699. IEEE
12. Heinrichs H (2013) Sharing economy: a potential new pathway to sustainability. GAIA-Ecol Perspect Sci Soc 22:228–231
13. Katz V (2015) Regulating the sharing economy. Berkeley Technol Law J 30:1067
14. Schor J (2016) Debating the sharing economy. J Self-Gov Manag Econ 4:11–22
15. Edelman BG, Luca M (2014) Digital discrimination: the case of airbnb. com. Harvard Business School NOM Unit Working Paper
16. Giuli M, Maselli I (2015) Uber: innovation or déja vu? ceps commentary, 25 february 2015
17. Casprini E, Paraboschi A, Di Minin A (2015) Web 2.0 enabled business models: an empirical investigation on the blablacar. it case. In: Academy of management proceedings, vol 2015, p 18509. Academy of Management
18. Jacque L (2014) Brand community building in peer-to-peer sharing–the case of blablacar
19. Barkai D (2001) Peer-to-peer computing: technologies for sharing and collaborating on the net, Intel Press, USA
20. Park JS, An G, Chandra D (2007) Trusted p2p computing environments with role-based access control. Inf Secur IET 1:27–35
21. Stoica I, Morris R, Karger D, Kaashoek MF, Balakrishnan H (2001) Chord: a scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM Comput Commun Rev 31:149–160
22. Lu H, Knutsson B, Delap M, Fiore J, Wu B (2004) The design of synchronization mechanisms for peer-to-peer massively multiplayer games. Department of Computer and Information Science The University of Pennsylvania Technical Report
23. Ørbekk K (2012) Distributed shared objects for mobile multiplayer games and applications
24. Wierzbicki A (2006) Trust enforcement in peer-to-peer massive multi-player online games. In: On the move to meaningful internet systems 2006: CoopIS, DOA, GADA, and ODBASE, pp 1163–1180. Springer
25. Wierzbicki A, Kucharski T (2005) Fair and scalable peer-to-peer games of turns. In: 2005 Proceedingsof the 11th international conference on parallel and distributed systems, vol 1, pp 250–256. IEEE
26. Oluwatosin HS (2014) Client-server model. IOSR J Comput Eng (IOSR-JCE) 16:67
27. Elo AE (1978) The rating of chessplayers, past and present. Arco Pub., New York. http://www.amazon.com/Rating-Chess-Players-Past-Present/dp/0668047216
28. Herbrich R, Minka T, Graepel T (2007) Trueskill^{tm}: a bayesian skill rating system. In: Schölkopf B, Platt JC, Hoffman T (eds) Advances in neural information processing systems 19 (NIPS-06), pp 569–576. MIT Press
29. Agarwal S, Lorch JR (2009) Matchmaking for online games and other latency-sensitive p2p systems. In: Rodriguez P, Biersack EW, Papagiannaki K, Rizzo L (eds) SIGCOMM, pp 315–326. ACM

30. Dabek F, Cox R, Kaashoek F, Morris R (2004) Vivaldi: a decentralized network coordinate system. SIGCOMM Comput Commun Rev 34:15–26. https://doi.org/10.1145/1030194.1015471
31. Manweiler J, Agarwal S, Zhang M, Choudhury RR, Bahl P (2011) Switchboard: a matchmaking system for multiplayer mobile games. In: Agrawala AK, Corner MD, Wetherall D (eds) MobiSys, pp 71–84. ACM
32. Myslak M, Deja D (2014) Developing game-structure sensitive matchmaking system for massive-multiplayer online games. In: Aiello LM, McFarland DA (eds) SocInfo workshops, lecture notes in computer science, vol 8852, pp 200–208. Springer
33. Jiang JR, Sung GY, Wu JW (2015) Lom: a leader oriented matchmaking algorithm for multiplayer online games
34. Saroiu S, Gummadi KP, Gribble SD (2003) Measuring and analyzing the characteristics of napster and gnutella hosts. Multimedia Syst 9:170–184
35. Good NS, Krekelberg A (2003) Usability and privacy: a study of kazaa p2p file-sharing. In: Proceedings of the SIGCHI conference on Human factors in computing systems, pp 137–144. ACM
36. Galuba W, Girdzijauskas S (2009) Distributed hash table. In: Liu L, Özsu MT (eds) Encyclopedia of database systems, pp 903–904. Springer US
37. Ferrari D (1990) Client requirements for real time communication systems. Network Working Group
38. Avizienis A, Laprie JC, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secure Comput 1:11–33. https://doi.org/10.1109/TDSC.2004.2
39. Boroń M, Kobusińska A, Brzeziński J (2018) Effectiveness metrics for the selfaid network, a p2p platform for game matchmaking systems. In: International conference on information systems architecture and technology, pp 111–122. Springer
40. Stingl D, Gross C, Rückert J, Nobach L, Kovacevic A, Steinmetz R (2011) Peerfactsim. kom: a simulation framework for peer-to-peer systems. In: 2011 international conference on high performance computing and simulation (HPCS), pp 577–584. IEEE
41. Boroń M, Brzeziński J, Kobusińska A (2017) Selfaid networka p2p matchmaking service. In: Multimedia and network information systems, pp 183–191. Springer

Michał Boroń received the B.Sc. and M.Sc. degrees in Computer Science from Poznań University of Technology, Poland in 2014 and 2015 respectively. His research interests include distributed algorithms, P2P systems, and cloud computing. He is currently a PhD student at Poznań University of Technology.

Jerzy Brzeziński received M.Sc. in electrical engineering, and Ph.D. and Dr. Habil. in computer science, all from Poznań University of Technology, where he is currently a Full Professor of Computer Science. His research interests include distributed algorithms and fault-tolerant distributed systems. He is the author and coauthor of two books, and over 100 research papers published in journal and proceeding of many international conferences. He has been involved in many international and national research projects. Prof. Brzeziński is a member of the IEEE CS, ACM, Polish Information Processing Society, Computer Science Committee of the Polish Academy of Sciences, among others.

Anna Kobusińska received her M.Sc. and PhD degrees in computer science from Poznań University of Technology, in 1999 and 2006, respectively. She currently works as an Assistant Professor at the Laboratory of Computing Systems, Institute of Computing Science, Poznań University of Technology, Poland. Her research interests include large scale distributed systems and algorithms, big data, SOA, fault-tolerance, replication and consistency models. She has served and is currently serving as a PC member of several international conferences and workshops. She is also author and coauthor of many publications in high quality peer reviewed international conferences and journals.