Bootstrapping in Peer-to-Peer Systems

Conference Paper · December 2008

DOI: 10.1109/ICPADS.2008.26 · Source: DBLP

4 authors, including:

Mirko Knoll
Telekom Deutschland GmbH
20 PUBLICATIONS **217** CITATIONS
SEE PROFILE

Arno Wacker
Universität der Bundeswehr München
68 PUBLICATIONS **646** CITATIONS
SEE PROFILE

Torben Weis
University of Duisburg-Essen
115 PUBLICATIONS **852** CITATIONS
SEE PROFILE

Bootstrapping in Peer-to-Peer Systems

Mirko Knoll[1], Arno Wacker[1], Gregor Schiele[2], and Torben Weis[1]

[1]University of Duisburg-Essen
47057 Duisburg, Germany
firstname.lastname@uni-due.de

[2]University of Mannheim
68131 Mannheim, Germany
gregor.schiele@uni-mannheim.de

Abstract

Peer-to-Peer systems have become a substantial element in computer networking. Distributing the load and splitting complex tasks are only some reasons why many developers have come to adopt this technology. However, all of them face a severe problem at the very beginning: setting up an overlay network, such that other clients can easily join it. With an empty peer cache common bootstrapping methods require some manually triggered actions for discovering a peer on the overlay. We therefore introduce an approach for an automated bootstrapping based on DDNS. In this paper we give detailed information about our protocol and document its efficiency and scalability.

1 Introduction

In peer-to-peer (P2P) systems, peers form a common overlay network, either unstructured (e.g. [12, 14]) or structured (e.g.[4, 17]). Before being able to use the P2P system, e.g., to search and exchange data, a new peer first must join the overlay network. This operation is known as *bootstrapping* [13, 6, 8]. More specifically, the goal of the bootstrapping operation is to find a peer that is already a member of the overlay network. If no such network exists, the searching peer must form a new overlay network that can be discovered and joined by further peers. Without a working bootstrapping, multiple isolated overlay networks may emerge, reducing the overall system performance for all peers.

In this paper we present our approach to detect an existing overlay network using the Dynamic Domain Name System (DDNS). We describe in detail how we use the DDNS architecture for bootstrapping P2P systems. To legitimate our approach we give an evaluation and discuss the strengths and shortcomings.

The paper is structured as follows: first, we define the terminology and fix the requirements for the bootstrapping protocol. Then we illustrate our approach and give detailed information on the protocol we have developed. In Section 6 we present the evaluation and give a detailed overview of the related work in this area. Finally we conclude our paper with a summary and an outlook on future works.

2 System Model

Before discussing the requirements of our bootstrapping approach, we describe our system model briefly.

Our system consists of a set of computers that are connected by a common communication network, e.g. the Internet. Using this network, the computers can send each others messages reliably. We assume that a subset of these computers runs a P2P-software. We call these computers *peers*. Peers form a connected P2P overlay network on top of the communication network. They may join and leave the overlay at any time without sending any further message. Therefore, the P2P network size varies in size from zero to all nodes of communication network. We do not assume a specific algorithm for forming the overlay. Instead, any such algorithm can be used, e.g. [19, 17].

3 Requirements

In this section we describe briefly the requirements that should be fulfilled by a bootstrapping mechanism. We have presented a more detailed discussion in [13].

1. **Availability:** Availability is one of the most important properties of a bootstrapping process. Operation of the bootstrapping mechanism must be guaranteed at any time. Thus, a probabilistic approach that works only with a given probability is not sufficient. Additionally, the system should be completely decentralized to overcome the problem of a single point of failure.

2. **Automation:** To operate conveniently, the bootstrapping mechanism must work fully automated and without manual user interaction.

3. **Efficiency:** To guarantee the acceptance of a bootstrapping protocol, it has to work efficiently. Thus, a node should be able to join an overlay within a reasonable amount of time, while limiting the network traffic on the communication network to a minimum.

4. **Scalability:** Another important factor is scalability, which has to make sure that the protocol is applicable in large communication networks and in addition is able to handle a large overlay network with many peers.

4 Design Rational

Before describing our approach for bootstrapping in detail we first motivate our major design decisions. There are two classes of approaches for bootstrapping: (1) peer-based and (2) mediator-based approaches. We discuss these classes in more detail in the related work section.

Peer-based approaches try to detect peers in the overlay by contacting other peers directly. A well known example for this class are peer-caches. A peer-cache contains a list of previously known peers. When a peer wants to enter the overlay, it tries to contact a peer in its peer-cache. If it is available, the contacted peer answers and can be used as entry point into the overlay. This approach is simple and very efficient. However, it cannot guarantee that the bootstrapping succeeds, as no peer in the cache may be available.

In contrast to this, mediator-based approaches use a well known entry point (WKEP), the mediator, to help in the discovery process. The mediator can, e.g., be a server provided by the operator of the P2P system. It manages a list of peers that are currently in the overlay and can point newly joining peers to one of them. The main challenge is to keep the mediator's data fresh and to make sure that the mediator is available. As long as this is the case, a mediator-based approach can guarantee bootstrapping. However, maintaining the mediator may consume a considerable amount of resources.

For our bootstrapping approach we combine peer-caches with a mediator acting as WKEP. First, the bootstrapping peer tries to contact any of the peers contained in its peer-cache. If this fails, it contacts the mediator and requests a currently active peer from it. This way we can take advantage of the efficiency of peer-caches and combine it with the guaranteed bootstrapping provided by the mediator. In this paper we concentrate on the mediator-based part of our approach and omit the usage of peer-caches. This simplifies the presentation of our algorithms and makes them easier to understand.

Our approach requires the mediator to be highly available. One approach to achieve this is to realize the mediator as a distributed service. Clearly, the operator of a P2P system can provide this distributed service. However, doing so could result in a high overhead simply for bootstrapping. Therefore, we propose to use an existing distributed Internet service as WKEP, instead of maintaining our own servers. This allows us to share the overhead of providing the distributed service with other applications that may be completely unrelated to the P2P system. To the best of our knowledge we are the first to propose using an existing distributed Internet service for bootstrapping a P2P system. A more detailed discussion of related work is given in Section 7.

The remaining question is, which Internet service to use for the bootstrapping process. In prior work [13] we discussed three promising services, namely (1) search engines, (2) DDNS, and (3) IRC. While the IRC approach is still ongoing research (see Section 8) in this paper we focus on the DDNS approach. We omitted implementing the approach based on search engines. The main reason for this is that search engines – while they are very scalable and highly available – usually update the information they are referring to rather slowly. Thus, a new search result may become available after several hours or even days. This is too slow for an effective bootstrapping.

Note that the design of our approach aims at keeping the additional load for the preexisting service low. This is very important. Otherwise, if the bootstrapping induces a high level of additional load to the preexisting service, its provider will not tolerate its usage for bootstrapping.

5 Dynamic DNS

DDNS is a variant of the domain name system (DNS). Just like DNS it allows to map domain names to IP addresses. However, DDNS allows to change this mapping much more easily than ordinary DNS. To do so, the DDNS provider usually offers a web-based interface to change the IP address that is associated with a given name. In addition, DDNS uses very low time-to-live values to allow changing the domain name mappings dynamically. In this section we describe a bootstrapping approach based on DDNS. We start by giving an overview of this approach and introduce the needed terminology. After that we present the details of the DDNS approach.

5.1 Overview

The main idea of the DDNS-based bootstrapping approach is to associate the IP address of a currently active peer with a predefined domain name. We call this peer the *bootstrapping peer* (BSP). To join the overlay, a new peer contacts the DDNS, resolves the predefined domain name (called *BS-DomainName*), and contacts the BSP. If no BSP can be resolved or the BSP does not answer, the new peer assumes that there is no overlay network present at this time and forms a new one. When the BSP leaves the system, another peer takes over by changing the domain name mapping to its own IP.

For this approach to work we have to assure that two properties are fulfilled: First, we must ensure that there is always a BSP present and reachable by new peers. As the BSP is an ordinary peer because we do not assume the existence of super-peers. Thus, it can leave the system at any time without further notice. Thus, we must check the availability of the currently registered BSP regularly and must replace it if it becomes unavailable. The most straight forward way to do so is to let all other peers in the overlay try to contact the BSP periodically. Clearly, in a P2P system with many peers, this approach could easily overwhelm the BSP, as too many peers try to communicate with it. Therefore, we introduce a new peer role, so-called guardians, that take over the responsibility to check the BSP's availability. Only guardians contact the BSP periodically. If a guardian detects a missing BSP, it takes over the BSP role itself. This allows us to drastically reduce the resulting overhead for checking and keeps it at a constant level even for large peer populations. Guardians are elected from the total set of peers dynamically. If a guardian leaves the system, a new one is elected. To do so, guardians not only check the availability of the BSP but also that of other guardians. The resulting architecture of the P2P system contains of three peer groups. The BSP is responsible for letting new peers join the overlay. The guardians make sure that the BSP as well as a sufficient number of guardians is available and reelect new ones if necessary. The ordinary peers do not participate in the bootstrapping system after they joined the overlay. However, they might be contacted by a guardian at any time and turned into a guardian, too.

Secondly, we must ensure that we do not induce too much additional load to the DDNS. Otherwise, the DDNS provider might deny us the further usage of the service. Most importantly we must minimize the number of DDNS update requests, i.e., the number of requests to change the mapping between *BS-DomainName* and a new BSP's IP address. If too many update requests arrive at the DDNS in a short time, a typical DDNS installation will suspect a denial of service attack and will stop performing updates for our domain name. For our approach to be usable, we must avoid this situation. An update request occurs in two cases. First, when a new overlay network is established and its first member becomes BSP. Second, when the current BSP leaves the system and the guardians replace it with another peer. In the first case, a problem might occur if many peers try to join a nonexisting overlay at the same time and try to become BSP at the same time. In practice, we assume this case to be rather uncommon and assume new peers to arrive in the system at random times. The second case is much more likely and we expect it to occur regularly. A problem might occur if many guardians detect the missing BSP at the same time and try to take over its role. To avoid this situation we use a randomized back-off algorithm before replacing a lost BSP.

In the following we describe our DDNS-based bootstrapping algorithm in more detail. We assume the existence of a number of basic procedures, e.g., for sending messages over the P2P overlay. We describe these procedures in Tab. 1.

5.2 Bootstrapping

When a new peer wants to join the overlay, it executes the `Bootstrapping` procedure as described in Alg. 1. The procedure is called with a single parameter, the predefined domain name (*BS-DomainName*) used for bootstrapping.

Algorithm 1 The main DDNS bootstrapping procedure

```
1:  procedure Bootstrapping(BS-DomainName)
2:  // LocalIP is the local IP-Address
3:  IP := DNSResolve(BS-DomainName);
4:  if CheckActive(IP) then
5:      JoinOverlay(IP);
6:      CheckBecomeGuardian();
7:  else
8:      Wait($\Delta t_{IT}$ + Random($\epsilon$));
9:      IP := DNSResolve(BS-DomainName);
10:     if CheckActive(IP) then
11:         JoinOverlay(IP);
12:         CheckBecomeGuardian();
13:     else
14:         DDNSUpdate(BS-DomainName, LocalIP);
15:     end if
16: end if
```

In the procedure the peer first resolves *BS-DomainName* to get the associated IP address. This IP is supposed to be the IP of the BSP. Before using the BSP to join the overlay, the peer first checks that the BSP is alive by calling the `CheckAlive` procedure. If the BSP cannot be contacted, the new peer must detect if (a) it is the first peer and should become BSP itself, or (b) the BSP is currently being replaced by another peer in the overlay. To do so, the peer waits for the maximum time needed to takeover the BSP role (Δt_{IT}) plus a randomized back-off time between 0 and ϵ. This random back-off algorithm reduces the risk that multiple peers try to update the DDNS entry at the same time. After Δt_{IT} + Random(ϵ) it resolves *BS-DomainName* again and repeats the test. If the BSP is

`DNSResolve(`*domainname*`)`	Returns the IP address for a the given name *domainname*.
`DDNSUpdate(`*domainname, address*`)`	Updates the ip address for a given *domainname* to the value of *address*.
`CheckActive(`*address*`)`	Checks if the address is used by an active peer.
`JoinOverlay(`*address*`)`	Enters the P2P overlay by using the given IP address.
`GetGuardianCount()`	Uses the BSP to retrieve the total number of active guardians.
`AnnounceGuard()`	Uses the BSP to announce the calling peer as a guardian.
`Wait(`Δt`)`	Waits for time Δt.
`Random(`x`)`	Returns a random number between 0 and x.
`SendOverlay(`*peer, msg*`)`	Sends the message *msg* to the given peer *peer*.

Table 1. Basic procedures used in the DDNS approach

still not reachable, the peer can conclude that it is the first peer, since otherwise another peer would have taken over the BSP by now. Thus, the peer becomes BSP itself by calling the `DDNSUpdate`procedure. This procedure maps *BS-DomainName* to the peer's IP address.

Note that this approach cannot guarantee that only exactly one peer assumes to be BSP. However, this does not do any harm, because the DDNS entry unambigiously determines the current BSP.

After successfully joining the P2P overlay network every peer (except the BSP) checks if it should become a guardian peer by calling the `CheckBecomeGuardian` procedure (see Alg. 2).

Algorithm 2 Checking whether to become a guardian

```
 1: procedure CheckBecomeGuardian()
 2: if GetGuardiansCount() < GuardThreshold then
 3:     Wait(Random($\Delta t_G$ + Random($\epsilon$)));
 4:     if GetGuardianCount() < GuardThreshold then
 5:         BSPWatchdog();
 6:         GuardWatchdog();
 7:         AnnounceGuard();
 8:     end if
 9: end if
```

It first checks if there are enough guardian peers active. If there are too few guardian peers active (i.e. fewer than *GuardThreshold*) the peer must wait for some randomized time Δt_G + Random(ϵ). This waiting time is needed to prevent situations where many peers become guardians at the same time. If the number of active guardian peers is still below the threshold after the second check, the peer must become a guardian peer itself.

A guardian peer has two responsibilities. First, it must regularly check if the BSP is still active (`BSPWatchdog`). If not, it tries to become BSP itself. Secondly, a guardian is responsible for regularly checking if there are enough other guardian peers active, in case one of them leaves the overlay (`GuardWatchdog`). Note that this check is done in addition to the check done by new peers. When a peer becomes a guardian it announces this to the BSP by using the `AnnounceGuard` procedure. This allows the BSP to keep track of the current number of active guardians and to realize the `GetGuardianCount` procedure.

5.3 Maintenance

In the following we describe the two watchdog procedures used by guardian peers. We start with `BSPWatchdog` after which we describe the `GuardWatchdog` method.

As already mentioned, a responsibility of a guardian peer is to check whether the BSP is still active. This is done by using the `BSPWatchdog` procedure given in Alg. 3.

Algorithm 3 Checking the bootstrap peer

```
 1: procedure BSPWatchdog()
 2: while true do
 3:     IP := DNSResolve(BS-DomainName);
 4:     if not CheckActive(IP) then
 5:         Wait(Random($\Delta t_T + \epsilon$));
 6:         IP := DNSResolve(BS-DomainName);
 7:         if not CheckActive(IP) then
 8:             DDNSUpdate(BS-DomainName, LocalIP);
 9:             return
10:         end if
11:     end if
12:     Wait($\Delta t_{TI}$);
13: end while
```

It runs an endless loop in which the guardian peer regularly checks if the BSP is still active. If this is the case, the peer waits for the interval time Δt_{TI} before checking again. Otherwise, the guardian peer waits for the randomized time $Random(\Delta t_T + \epsilon)$. This waiting time is needed to prevent multiple takeovers of the domain name *BS-DomainName*, effectively preventing update bursts. After waiting for the back-off time, the peer checks a second time. If no new BSP is detected, it takes over the BSP role itself. If a guardian becomes the BSP, it exits the BSP watchdog – since there is no need to check itself.

In addition to the `BSPWatchdog` procedure, each guardian executes the `GuardWatchdog` procedure (see Alg. 4). Just like `BSPWatchdog`, this procedure runs in an endless loop.

Algorithm 4 Checking the number of active guardians

```
 1: procedure GuardWatchdog()
 2: while true do
 3:   if GetGuardiansCount() < GuardThreshold then
 4:     Wait(Random($\Delta t_G$ + Random($\epsilon$)));
 5:     if GetGuardianCount() < GuardThreshold then
 6:       SendOverlay(SomeActivePeer,
          "BecomeGuard");
 7:     end if
 8:   end if
 9:   Wait($\Delta t_{GI}$);
10: end while
```

First it checks if there are enough active guardians, i.e., more than *GuardThreshold*. If this is the case, the guardian waits for a time Δt_{GI} and starts over. Otherwise, one or more guardians have left the overlay since the last check and must be replaced by new guardians. To achieve this, the guardian chooses an arbitrary peer in the overlay and sends an invitation message to it. Clearly, this peer must not be the BSP or a guardian already. After receiving the invitation, the peer becomes a new guardian. Again, this approach can lead to multiple guardians inviting new peers at the same time, resulting in too many guardians. To reduce this effect, we use another randomized back-off time $\Delta t_G +$ Random(ϵ). Before inviting a peer, the guardian waits for the back-off time and checks again. The invitation is then only sent if there are still too few guardians.

6 Evaluation

To evaluate our approach, we developed a prototypical implementation using C#. Our prototype is independent of a specific P2P overlay protocol and focuses on the bootstrapping process. Thus, we only consider messages that are necessary for bootstrapping. We omit other messages exchanged in the P2P system, e.g., to maintain an already created overlay network. Clearly, these messages introduce additional load into the system. However, this load is not related to bootstrapping and therefore independent of our approach. In addition, our approach is independent of the total number of peers present in the system. Only the BSP, the guardian peers, and the peers that are currently trying to join the network participate in the bootstrapping. Therefore, we can omit other peers that have already joined the overlay. This results in scenarios with much lower numbers of peers, allowing us to use real world experiments instead of simulations.

6.1 Settings

In our evaluation we examined the traffic necessary to provide a successful bootstrapping for the DDNS approach. Therefore, we created a set of scenarios with different join and leave parameters as follows.

Scenario 1: To model a realistic behavior of peer arrival and leaving we chose to use Markov birth and death processes [21]. As the birth and death of a process are independent from previous events, we set the birth rates $\lambda_0 = \lambda_1 = ... = \lambda_n = 0.8$ respectively the death rates $\mu_0 = \mu_1 = ... = \mu_n = 0.81$ equal for all nodes of the Markov chain. The total runtime for the simulation was one hour with an event (birth or death) taking place every 10 sec.

Scenario 2: On the other hand, we want to make sure that our approach copes with a steady rise of nodes as well. Therefore, we randomly generated between 1 and 5 nodes every 5 seconds until we reach the desired amount. According to the observations of [7] we set the lifetime of the nodes between 4 and 6 minutes after which it will leave without sending further messages. Each node leaving the network is replaced within the next 5 seconds the latest, thus keeping the overall amount on the desired value. We increased the node amount from 10 to 50 by 10 in 30 minute steps, repeating each measurement 5 times leading to an overall simulation time of 12,5 hours.

6.2 Messages

For our approach we use a set of messages depending on the underlying infrastructure. We therefore consider the following message types:

- `UpdateDomain`: messages that are used to tell the DDNS server that a new guardian has become bootstrap peer
- `GetAddress`: resolves the domain and thus informs the requesting peer about the current bootstrap peer
- `Ping/Pong`: these messages are used by the guardians to check the liveliness of the bootstrap peer

The `UpdateDomain` messages have to stay low. That is at most one message per minute. Otherwise most DDNS providers (e.g. dyndns.com) will disable the update function and lock the domain for at least some time. Each peer willing to enter the overlay will contact the DDNS server using a `GetAddress` message. Thus, the amount of messages will increase with growing networks. However, as DDNS is scalable due to its complex hierarchy, these messages will be compensated and not effect neither the DDNS provider nor the bootstrap peer. Yet, the bootstrap peer has to cope with the ping messages sent by the guardians. As the amount of guardians varies only slightly regardless of the amount of peers in the overlay, this amount stays constantly low.

Results: Using Scenario 1 we can see in Figure 1 that the load on the DDNS server is minimal. Only very few updates occur due to a high lifetime of the bootstrapping peers. To verify that availability of the BSP, guardians exchange `Ping/Pong` messages with it. As the number of guardians stays constant, the load on the BSP stays constant as well (see Figure 2). The amount of `GetAddress` messages is dependent on the amount of nodes requesting a name resolution. However, this name resolution does not put any load on the DDNS server, but uses the DNS protocol. The DNS protocol relies on a hierarchical architecture with caching servers and thus isolates this traffic from the DDNS provider. In Figure 3 we can see the performance using Scenario 2. Measurements with different increasing numbers of participants have minimal effect on the amount of messages being sent. In this scenario there are more domain updates as the lifetime of a bootstrap peer is limited to a maximum of 6 minutes. Nevertheless, the guardians absorb the messages resulting in the change of bootstrap peers and thereby keep the overall message count for the DDNS constantly low.

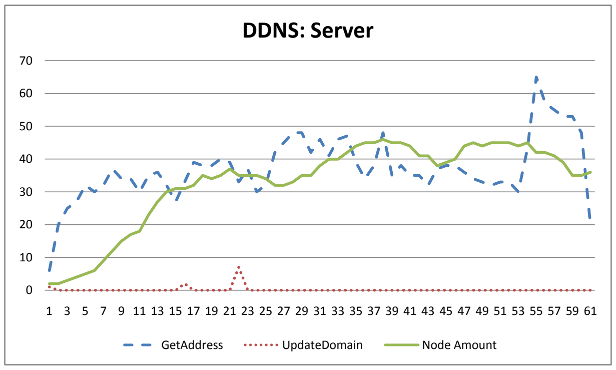

Figure 1. DDNS Server Load

6.3 Open Issues

In this paper we focus more on the technical feasibility than on security aspects. One of the major weaknesses of the DDNS approach is the fact that each domain is accessible via a password only. A malicious node could change this password and thereby hinder all nodes to update their DDNS entries with their most recent IP addresses. Nodes will then either receive node caches with addresses from malicious respectively fake nodes or not receive a node cache at all. However, this approach does not differ from a major authority's effort to cut off file-sharing by providing their own supernodes in KaZaa. Furthermore, in case the bootstrap peer changes frequently due to log-offs, additional pauses have to be introduced. Otherwise, the DDNS server would receive too many updates and lock the account for this subdomain.

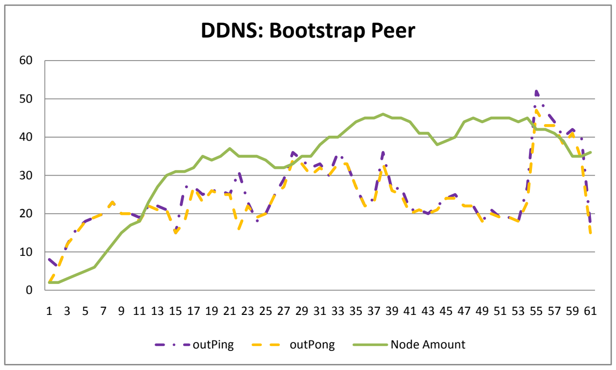

Figure 2. Load on BSP using DDNS

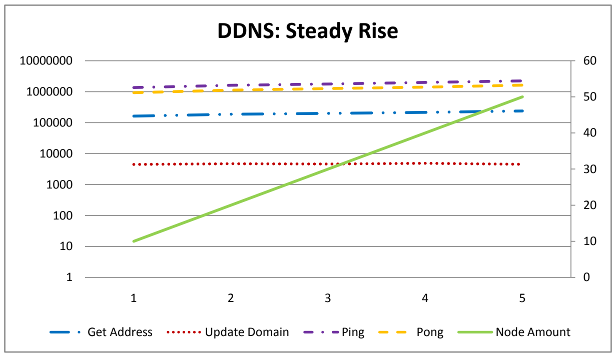

Figure 3. DDNS Behavior under a steady rise

7 Related Work

There are several different approaches on how to discover a peer in an overlay network. In the following we distinguish between two classes, namely the peer-based approaches and the mediator-based approaches.

7.1 Peer-based approaches

Nodes applying the peer-based approach are trying to directly detect and contact peers of the overlay network. After exchanging communication with regular peers, they gain information for joining the network themselves.

Using *peer caches* is one of the simplest approaches, yet it proves to be effective. After signing off from the overlay network, a peer stores the addresses of all other peers that is has been connected to in its cache, which is then written to stable storage. On the next attempt to join the overlay, the peer loads its cache and tries to connect to the enlisted nodes directly. However, in case none of the peers from the peer cache is active and currently present in the overlay, the bootstrapping process will fail.

By *random probing*, a node tries to randomly connect to another node in the communication network by sending messages over a distinct port [7]. In case the contacted node is a peer, it replies, otherwise the node generates a new IP address and tries to contact this one. Random probing therefore works well in dense networks with a high peer to node ration, however, it is not suited for starting a new overlay network and bootstrapping the first nodes.

Foner's Yenta approach [9] uses *broadcasts* for bootstrapping. A node sends out a UDP broadcast, querying for peers of the overlay. As gateway routers do not forward broadcast messages, the node will only get a reply if there is an active peer in the local area network. Thus, this approach only works in very dense networks or in combination with other bootstrapping approaches. Cramer et al. [7] propose the *multicast* protocol to group all peer of the same local network. When a node wants to join the overlay network, it sends a query to the multicast group and receives the IP addresses of the other participants in return. As for now, this approach cannot be used in a world-spanning communication network as multicast packets are discarded at most routers. *Anycast* addresses the necessity to contact exactly one out of all peers of the overlay. The new node sends a message to the P2P group, not addressing a specific peer. The recipient of the message is determined through the routing, more specific, the topologically closest node to the sender will receive the message and answers with information about further nodes. However, anycast will not scale nowadays [2] [1] as it is very wasteful with IP addresses and requires core router modifications.

In [3] Castro et al. propose a universal ring, which is only used to bootstrap other services. Peers of all existing overlay networks participate in this ring, forming a very large common overlay. If there are lots of peers in the universal ring (e.g. through an operating system integration of the protocol), a certified node may try to find the universal ring through random probing or a form of controlled flooding, such as expanding ring IP multicast.

7.2 Mediator-based approaches

Unlike the peer-based approach, peers using this approach need a special mediator to find other peers of the overlay. The role of the mediator can be played by peers, which are participating in the overlay or by external nodes. Its task comprises of the allocation of a directory service and a *well known entry point* (WKEP), which is hard-coded into the bootstrapping protocol. To join the overlay network, a node then contacts the mediator and requests the address of one or more active peers. One of the major challenges for this kind of approach is to keep the mediator information up to date, such that the directory always contains a list of currently active peers. Furthermore, the accessibility of the mediator is vital to bootstrapping process, as without it, new nodes cannot join the overlay. Some mediator-based approaches have evolved into various products, however only few of them are still in use.

Napster utilizes a directory service to find other peers of the overlay [18]. The index database stored at Napster contains information about all files that participating peers are willing to share. A peer queries for a certain file and the server answers with an address of a node that shares this file. However, this approach suffers from a single point of failure, as the overlay network is no longer operational if the server farm becomes unavailable.

The first Gnutella client (namely version 0.4) used a hard-coded URL to discover a bootstrapping server, the so-called *host cache* [11]. A new node connects to the host cache and receives a list of recently active peers, which it then tries to contact. A similar approach was first presented in YOID [10], and later used in CAN [16] and most other recent P2P networks. Whenever the host cache was unreachable, users had to exchange peer addresses manually in chat rooms (e.g. in the IRC), violating the requirement for an automated bootstrapping.

To distribute information about active servers in the overlay the eDonkey software utilizes a web-based approach. Users search for a recent *server list*, which they download and integrate in the client. Further updates may then be received over the eDonkey network eliminating the need to download the lists before use of the software. However, companies such as the Recording Industry Association of America (RIAA) try to prevent illegal downloads and thus emit fradulent information over the update channel. This often renders the service unusable.

Bittorrent [5] has become one of the most popular file sharing tools.To start the download of a file, the user has to get a special *.torrent* file, which includes various file information and the URL of the so-called *tracker*. However, the user still has to search for the torrent file manually and once the tracker is down, new nodes will not be able to download the file.

8 Conclusion and Future Work

Solving the bootstrapping problem is a challenging task. In this paper we presented an approach offering a solution for this challenge. We use the existing DDNS service as a fall-back whenever the peer-based approach fails. This service is distributed and therefore has a high availability adding robustness to our approach. Through the evaluation we have shown that our protocol can be used for the bootstrapping while putting a constantly low load on the underlying infrastructure only.

We showed that the protocol is very *efficient*, since a node normally only needs to perform a single DNS request to detect an existing peer. Furthermore, it is *robust against failure*, as leaving peers are detected and others take their place. In addition, the approach is highly *scalable* as the number of guardians is independent from the participating nodes. The only data the user has to enter manually is the domain, which is used for the then *automated* bootstrapping. Unfortunately, users have to be aware of a common password to update the DDNS entry. If this password is changed, future updates will not be possible.

At the same time we are working on a bootstrapping protocol using the internet relay chat (IRC). As the IRC consists of several thousand distributed servers, it seems almost impossible to shut this service down. Therefore, we use so-called channels to pass out information about active users. As the channels broadcast messages to all participants in the channel, we are confident that this approach scales even with high churn rates. Furthermore, IRC channels do not necessarily require a password, thus eliminating one of the drawbacks of the current DDNS approach. The ongoing results looks promising, so that we plan to publish these in our next paper.

However, special care has to be taken with such protocols as they can easily be abused for controlling large amounts of hacked computers, e.g. bot-nets [15, 20]. Currently bot-nets rely on fast flux networks, which require a very cooperative DDNS server. In contrast, our approach is able to work within standard parameters of the service providers.

References

[1] J. Abley, A. Canada, and K. Lindqvist. Operation of anycast services. Request for Comments: 4786 / Best Current Practice: 126, December 2006.

[2] H. Ballani and P. Francis. Towards a Deployable IP Anycast Service. In *Proc. of First Workshop on Real, Large Distributed Systems (WORLDS '04)*, December 2004.

[3] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. One ring to rule them all: service discovery and binding in structured peer-to-peer overlay networks. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC*, pages 140–145, New York, NY, USA, 2002. ACM Press.

[4] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46ff., 2001.

[5] B. Cohen. Incentives build robustness in bittorrent. In *In Proceedings of the First Workshop on the Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, June 2003.

[6] C. Cramer and T. Fuhrmann. Bootstrapping chord in ad hoc networks: Not going anywhere for a while. In *Proceedings of the 3rd IEEE International Workshop on Mobile Peer-to-Peer Computing*, Pisa, Italy, March 2006.

[7] C. Cramer, K. Kutzner, and T. Fuhrmann. Bootstrapping locality-aware p2p networks. In *Proceedings of the IEEE International Conference on Networks (ICON)*, pages 357–361, Singapore, November 2004.

[8] W. Ding. Bootstrapping chord over manets - all roads lead to rome. In *IEEE Wireless Communications and Networking Conference (WCNC)*, pages 3501–3506, Kowloon, China, March 11-15 2007.

[9] L. N. Foner. *Political Artifacts and Personal Privacy: The Yenta Multi-Agent Distributed Matchmaking System*. PhD thesis, Massachusetts Institure of Technology, April 1999.

[10] P. Francis. Yoid: Extending the internet multicast architecture. 2000.

[11] G. Kan. Gnutella. In A. Oram, editor, *Peer-to-Peer. Harnessing the Power of Disruptive Technologies*, pages 94–122, Sebastopol, CA, USA, 2001. O'Reilly.

[12] P. Kirk. The annotated gnutella protocol specification v0.4, 2003.

[13] M. Knoll, A. Wacker, G. Schiele, and T. Weis. Decentralized bootstrapping in pervasive applications. *Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PerComW'07)*, 0:589–592, 2007.

[14] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 84–95, New York, NY, USA, 2002. ACM.

[15] T. Moore and R. Clayton. An empirical analysis of the current state of phishing attack and defence. In *Sixth Workshop on the Economics of Information Security*, June 7-8 2007.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 161–172, San Diego, CA, USA, 2001. ACM Press.

[17] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object loaction for routing for large-scale peer-to-peer systems. In *Proceedings IFIP/ACM Middleware 2001*, November 2001. Heidelberg, Germany.

[18] C. Shirky. Listening to napster. In A. Oram, editor, *Peer-to-Peer. Harnessing the Power of Disruptive Technologies*, pages 21–37, Sebastopol, CA, USA, 2001. O'Reilly.

[19] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, San Diego, CA, USA, 2001. ACM Press.

[20] The Honeynet Project & Research Alliance. Fast-flux service networks: An ever changing enemy, July 13th 2007.

[21] Z. Wang and X. Yang. *Birth and death processes and Markov chains*. Science Press, Beijing, China, 1992.