P2P-RMI: TRANSPARENT DISTRIBUTION OF REMOTE JAVA OBJECTS

Thomas Zink, Oliver Haase, Jürgen Wäsch[1] Marcel Waldvogel[2]

[1]HTWG Konstanz, Konstanz, Germany
[tzinkjhaasejwaesch]@htwg-konstanz.de
2University of Konstanz, Konstanz, Germany
marcel.waldvogel@uni.kn

ABSTRACT

Java Remote Method Invocation (RMI) is a built-in and easy-to-use framework for the distribution of remote Java objects. Its simplicity and seamless inter-virtual machine communication has made it a valuable tool for distributed services. It nevertheless exhibits certain constraints that practically limit RMI applications to the classical client/server distribution model, and make highly distributed and highly dynamic systems very difficult to build atop RMI.

We present an approach that makes Java RMI usable for P2P and similar distribution models. The solution basically consists of three ideas: (1) separate the location of the registry from the remote service object, (2) distribute the registry across a DHT infrastructure, and (3) transparently enhance the built-in communication between RMI servers and clients to allow traversal of NAT and firewall boundaries. Our approach is extremely lightweight, transparent, and requires practically zero configuration.

KEYWORDS

P2P, RMI

1. INTRODUCTION

Java Remote Method Invocation (RMI)[1] allows the distribution of Java objects, enabling clients to invoke methods on remote objects. The basic architecture is shown in Figure 1. A server exposes a remote object by exporting its client stub to the RMI *r*egistry (1). Clients use the registry to lookup exported objects and receive their stub (2). If the client JVM does not find the class definitions of the remote object's interface or any other types that are used within the interface in its local classpath, it queries the *c*odebase whose location can be provided within the stub by the remote object server (3). The client can then invoke methods on the remote server object (4).

The standard Java RMI registry requires the remote server object and the RMI registry to be colocated on the same machine; for clients to be able to contact the registry, this machine must be publicly reachable. The same applies to the codebase, which must reside on a publicly reachable HTTP server.

These constraints have no negative effect on a standard client/server architecture with dedicated, public servers, but become a burden when remote object servers are to be deployed

DOI : 10.5121/ijcnc.2012.4502

behind firewalls or NAT boxes. This is the case in highly distributed and P2Pbased scenarios, where every peer acts as both a remote object server and a client. This shortcoming is a major reason why Java RMI is not widely used for highly distributed Java applications, despite its simplicity and elegance.

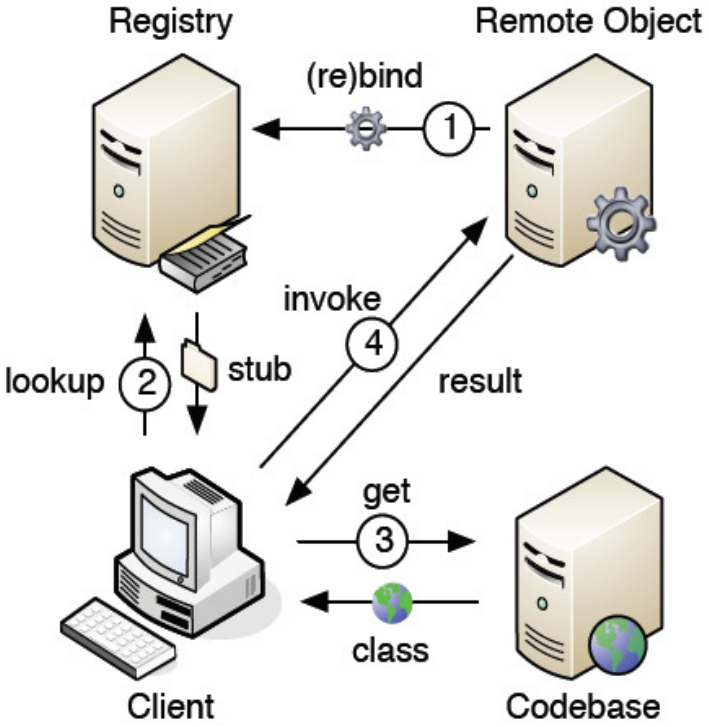

Figure 1: Java RMI architecture. 1. the server registers the remote object. 2. the client looks up the stub in the registry. 3. the client requests the stubs class file. 4. the client invokes methods on the remote object. The remote object and registry have to reside on the same machine

In this paper we present an approach that enables Java RMI to utilize existing P2P infrastructure for the exchange of remote object stubs. This *P2P-RMI* enables peers to function as remote object clients and servers alike, allowing the distribution of Java applications in P2P networks. We make use of Java RMIs feature of exchanging custom socket factories to traverse NAT boxes and firewalls, thus allowing deployment of remote object servers even behind network boundaries.

As a result, the framework is very lightweight, requires no changes to client code and practically zero configuration. In addition, it is extremely flexible and scalable to the extend that it can be deployed in practically any environment and be extended to any desired size.

Currently, there are a number of proposed techniques to distribute Java applications and services. However, they all require significant changes to clients or the use of dedicated application servers or similar web services. To the best of our knowledge, our approach is unique in the sense that it allows true P2P-like distribution of Java remote objects without requiring any changes to client applications.

2. RELATED WORK

Java has always been targeted towards web and distributed applications and naturally both the language itself and the core library offer built-in support for distributed computing. Distributed systems can be built on a multitude of different architectures that also define their granularity and flexibility.

Java RMI has first been introduced in [16] and has been integrated into the Java core library with platform release 1.1. Since then, it matured with new releases and has also been extended to support CORBA through RMI-IIOP. Later, RMI was extended with custom socket factories to allow for customized communication. RMI is an object-oriented approach and as such on a lower layer of abstraction than component-based or service-based approaches (like web services).

Sun Microsystems further leveraged distribution capabilities for Java by introducing Jini, now Apache River, in 1998. Apache River supports the development of secure distributed applications and services. Communication is facilitated by the Jini Extensible Remote Invocation protocol (JERI), which is semantically similar, but not equal to RMI. Resource sharing is embodied through JavaSpaces [13] that are targeted towards grid computing. Apache River also utilizes a centralized registry and class file servers for the distribution of stubs and class files. In addition it features compatibility with RMI by allowing the direct usage of the Java Remote Method Protocol (JRMP).

Cajo is a library for building distributed Java virtual machines. Internally, Cajo uses RMI but it encapsulates the low level details. It also features controller and agent objects that can be exchanged between virtual machines to offload computation and reduce network traffic. Cajo uses multicast for communication, which limits its applicability in highly distributed systems.

CloudSNAP [15] is a platform for decentralized deployment of Java Enterprise Edition (Java EE) web applications. It uses a structured P2P architecture in which every peer runs a web server and can be host to any kind of web service. Applications can thus be distributed among the peers. The distribution is achieved with an Aspect Oriented Interception Middleware (DAMON) that can intercept and distribute aspects. Although the underlying architecture is a P2P network, CloudSNAP can be better described as a cloud platform service that is specifically targeted towards aspect-oriented Java EE web applications.

P2P-MPI [9] is a project that aims at parallel P2P grid computing. It is a framework that extends the file sharing paradigm to general resource sharing and provides a P2Pbased Message Passing Interface (MPI) infrastructure. The idea is to have peers share their computation and memory resources and effectively act as a computational grid.

JXTA provides a protocol definition and library to build P2P-based applications. The protocol itself is language independent but services and applications must at least implement the JXTA core functionality.

In conclusion, it is safe to say that while there are efforts addressing P2P-based distribution of Java programs, none of them actually provide a simple, object-oriented approach that allows seamless and transparent integration of unmodified clients and applications. P2PRMI aims at filling this gap.

3. JAVA P2P-RMI SERVICES

The RMI registry is the central instance in the RMI infrastructure. Remote object servers export their services to the registry. Clients use the registry to lookup remote object stubs that include information about the object's server and class. Stubs can also include their own custom socket factories and information regarding the codebase. In order to transparently distribute RMI services, the following problems have to be solved.

1. *Lookup*: Both clients and servers must be able to use a remote RMI registry for lookup and export.
2. *Retrieval* : Clients must be able to obtain class files for unknown classes from a codebase.
3. *Method Invocation*: Remote object services must be accessible even behind firewalls and NAT boxes.

Accessibility of remote objects (problem 3) can be ensured by exploiting RMI's feature of transporting custom socket factories within the stub, which has been discussed in-depth in [10][12]. It is reviewed in ?? for the sake of completeness.

To provide a distributed registry and remote object service (problem 1) we utilize a *Distributed Hash Table* (DHT). The basic idea is to store the service stubs in the DHT. The P2P-RMI registry simply provides bindings - or adapters - to the DHT. Remote objects that are exported to the P2P-RMI registry are bound to the DHT using a unique *URI* as key. They can be looked up by any P2P-RMI-aware peer. Please note that we distinguish between two different types of peers: (1) DHT *peers* that provide the DHT service, and (2) P2P-RMI *peers* that provide and consume RMI related services. It is not a requirement for P2P-RMI peers to also be DHT peers. It is also possible to join the P2P-RMI network without participating in the DHT network through the use of public gateways that remotely provide the necessary binding.

There is also the issue of where to store the class files and how to make them accessible for peers (problem 2). When a client deserializes a server stub, it first searches its local classpath, and if unsuccessful, inspects the `java.rmi.server.codebase` property, which is set by the remote object server and included in the stub. Usually the codebase is a HTTP or FTP server that offers the class files for download. However, the mechanism can be exploited to distribute the codebase in P2P networks.

In the following, we briefly review [10] that addresses problem 3, and then elaborate on problem 1. While the focus of this work is the distribution of the RMI registry we also discuss problem 2 to provide a complete solution for a Java P2P-RMI network.

3.1. NAT- TRAVERSAL WITH RMI

In [11] the authors show how custom RMI socket factories can be used to enable NAT-traversal for remote object servers. The mechanism is illustrated in Figure 2.

The remote object server first connects to a publicly available rendezvous server (1). It then exports the remote object by registering its stub that includes the custom *NAT- traversal socket factory* (2). A client receiving the stub (3) and trying to connect to the remote object transparently uses the custom socket factory. This socket factory directs the client to the rendezvous server (4), which helps start (5) the mutual hole punching process between client and server. It should be noted that the stub includes a socket factory *instance*; the client either

needs to have the class definition locally or download it from a remote codebase, see problem 2 in section.

This scheme practically allows connectivity in any environment since any kind of NAT-traversal mechanism can be used and communicated with the clients through the stub. It requires, however, additional well-known rendezvous servers for relaying or negotiating the actual client to server connection. In this paper we will not evaluate NAT-traversal techniques and their performance since this has already been done in [12].

3.2. PRIVATE P2P-RMI SERVICE

To distribute the RMI registry in a P2P network, the most straight-forward approach is to equip all peers with a local P2P-RMI registry that connects to the DHT. This design is depicted in Figure 3. In this architecture, remote object servers simply use their local P2PRMI registry to bind the remote object to the DHT. A server behind NAT takes advantage of the NAT-traversal techniques described in [10] and includes the NAT-traversal socket factory in its exported stub. Clients, respectively, look up the stubs using their own local P2P-RMI registry.

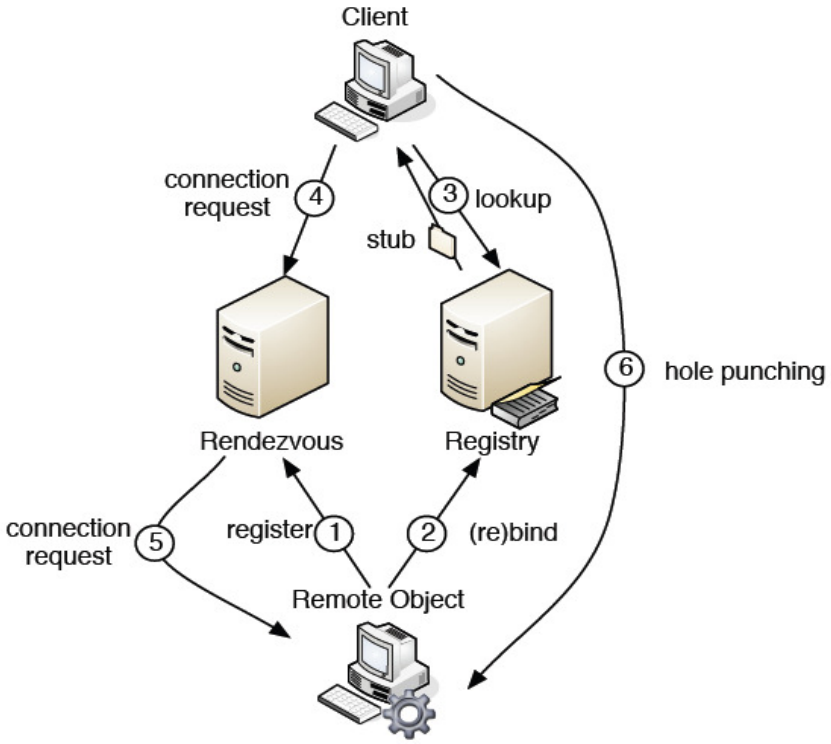

Figure 2: NAT-traversal for remote objects.

This design basically eliminates the need for a central, publicly reachable registry service at the expense of excluding unmodified clients from looking up remote objects. The peer-local registries do not provide services but merely allow local utilization of the DHT as storage for RMI. Since there is no public RMI registry service, ordinary clients cannot use this P2P-RMI network.

Peers must also know the keys of remote objects prior to the lookup. This is true for any RMI architecture and not a new problem. Registries implement a list() function that lists all remote

objects which have been exported using the queried registry. Here, list() can only be called locally and thus only returns all objects exported by the local peer. Listing all globally available remote objects requires the DHT to support approximate string searches like Cubit [17]. Whether or not global listing is possible therefore depends on the DHT implementation.

3.3. P2P-RMI WITH DEDICATED REGISTRY SERVERS

The reference Java RMI registry (`rmiregistry`) does not persistently store its bindings and requires servers to run collocated with the registry. [11] proposes a *Remote RMI Registry* that uses persistent storage to store stub information and can be used by remote servers. Instead of local file systems or data bases, the remote RMI registry can also utilize a DHT - or any other distributed storage - as storage backend. This *Remote P2P-RMI Registry* participates in the DHT network and exposes the registry service to remote clients and object servers. It basically serves as a gateway to the P2P-RMI network for legacy clients and servers. They can use this registry to lookup and export remote objects, thus effectively becoming P2P-RMI peers without the need to participate in the DHT network themselves (Figure 4).

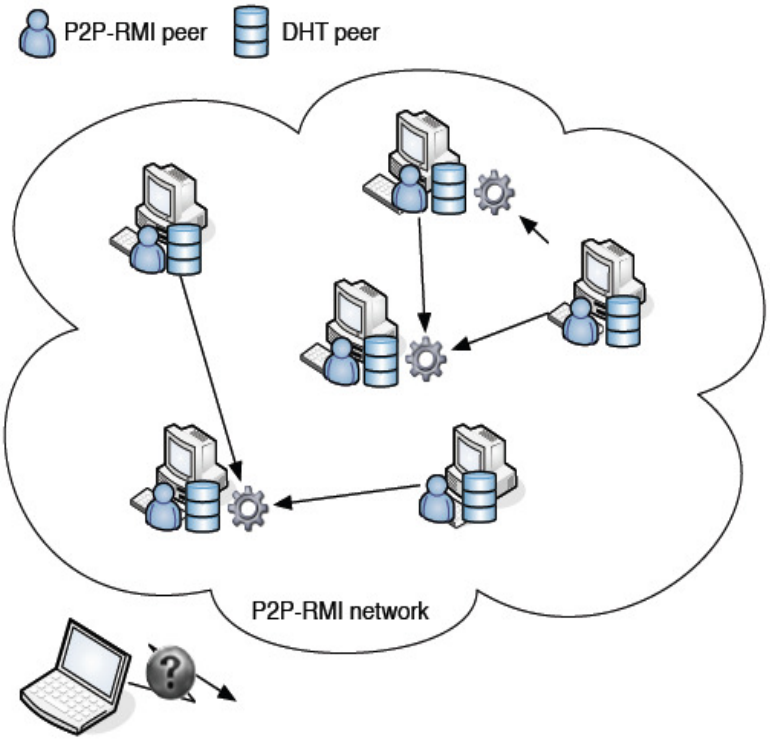

Figure 3: P2P-RMI. All peers can export and lookup remote objects using their local registry to connect to the DHT network. Clients that are not peers in the DHT network cannot use the service.

The use of P2P-RMI gateways solves incompatibility issues with unmodified clients and servers at the expense of re-introducing dedicated registry servers. This solution is useful as an interim step during the roll-out of the architecture presented in the next section, until enough super-peers become available to provide a reliable infrastructure.

3.4. P2P-RMI WITH SUPER-PEERS

To eliminate the need for dedicated P2P-RMI gateways, we suggest an approach that is influenced by well-known P2P networks like Gnutella and Skype in the form of super-peers. Highly available peers can be promoted to super-peers, which provide a number of services that are otherwise difficult to deploy in a distributed manner (Figure 5). In particular, a super-peer can take on the role of a remote P2P-RMI registry, serving as a gateway to the P2P-RMI network. In addition, super-peers can also work as rendezvous or relay servers and thus provide NAT-traversal services for NAT'ed peers. Finally, super-peers can provide a distributed codebase service as described in the next section.

The use of super-peers removes the need for central gateways and allows for purely self organizing, highly robust, and scalable P2P-RMI networks. The cost for decentralization, however, is bootstrapping. To participate in the P2P-RMI network a client or server must either have access to the DHT or know at least one active super-peer to make use of its remote registry service. This bootstrapping problem is present in any P2P network and is not unique to the P2P-RMI network presented here. Therefore, multiple solutions exist, both centralized and decentralized, including hotlists, random address probing, or even utilizing service records in DNS. Since a RMI client needs to know the address of a RMI registry anyway, we assume that a potential client or peer is aware of an existing super-peer or other gateway. This information can be provided as a hotlist, configuration file or even as a system property. For readers interested in decentralized bootstrapping approaches we refer to literature such as [4, 7, 6].

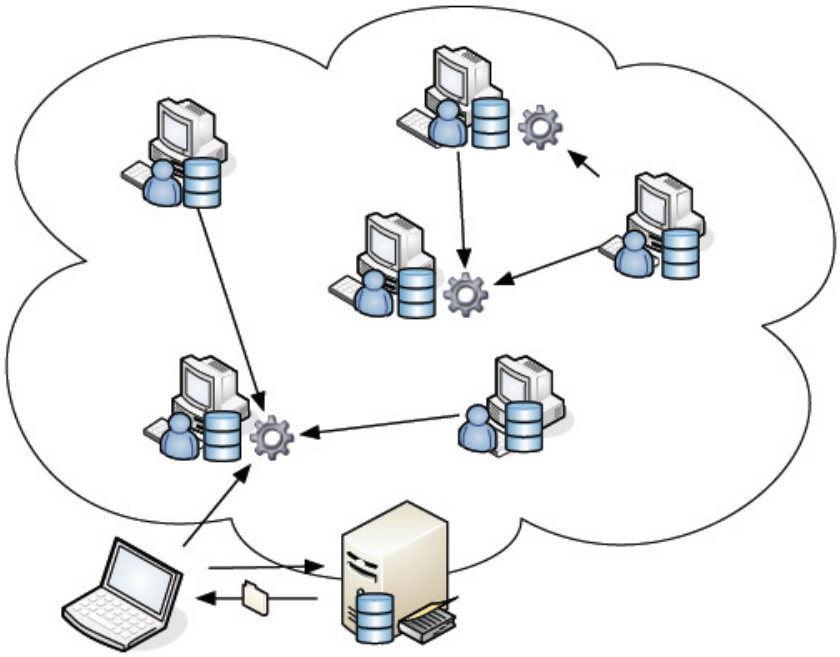

Figure 4: P2P-RMI with dedicated servers. The servers provide the RMI registry service as a gateway to the P2P-RMI network, allowing external clients to lookup and export remote objects, effectively becoming a P2P-RMI peer.

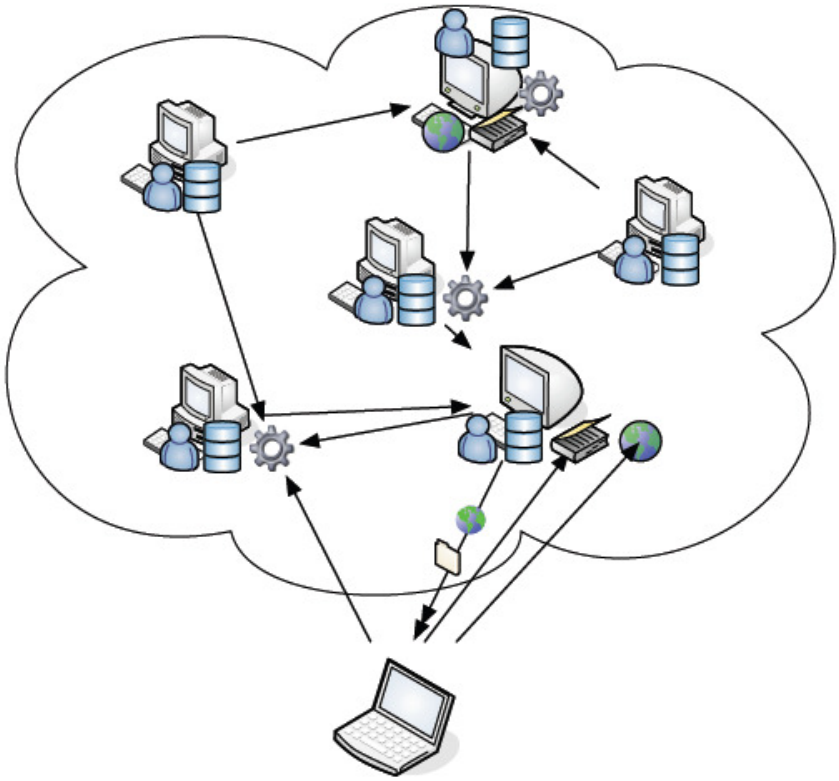

Figure 5: P2P-RMI with super-peers. Highly available peers can be promoted to super-peers, which offer RMI and connectivity services.

3.5. DISTRIBUTING CLASS FILES

As described earlier in this section, one challenge for highly distributed RMI systems is for clients to retrieve all class definitions that are necessary to deserialize the server stubs they looked up at the registry. If a client cannot find such a class definition in its local classpath - which will mostly be the case in a highly distributed system - the client retrieves it from the codebase that is included in the remote object's stub. For the implementation of this codebase, several options can be thought of (Figure 6):

1. Dedicated codebase server
2. Local DHT-based codebase
3. Peer-based class file server
4. Super-peer-based codebase

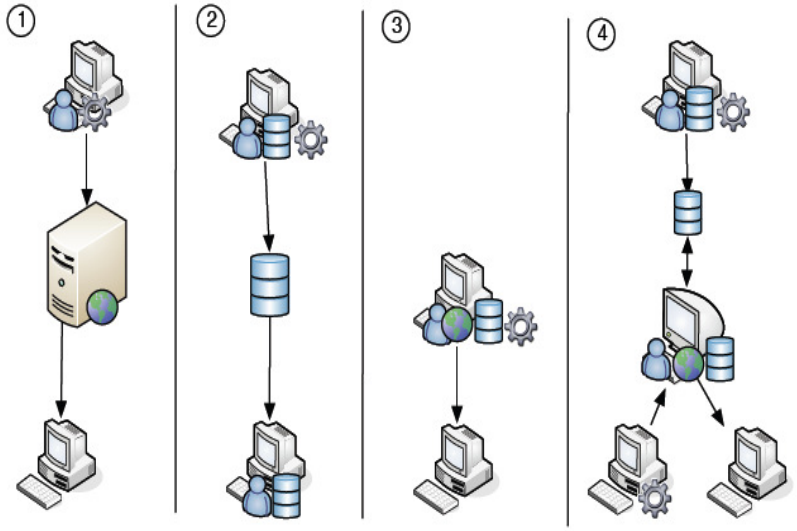

Figure 6: Distributing class files using (1) a dedicated server (2) the DHT to store code (3) the remote object's server (4) a super-peer-based codebase

Solution 1 is to use a globally reachable, dedicated codebase server. Peers that export objects upload the class definitions to the dedicated server that clients can download from. While this is the easiest solution, it is also the most centralized one; it should be noted, however, that a client needs to contact a codebase server *per class definition*, as compared to registries that are contacted *per object instance*. Hence, scalability is generally a lesser issue for codebase servers than it is for registries.

A more decentralized solution is to use the DHT as storage for class files (solution 2). The exporting peer stores the class file in the DHT and sets the codebase property to a custom URL format, such as `dht://package.classname`. Any client that is also a DHT peer can then lookup the class files directly in the DHT and load them from there. Thus POST and GET of code are both performed locally. This approach requires a custom *ClassLoader* with DHT bindings on the client side and only works for clients that are both P2P-RMI and DHT peers.

As with the RMI registry, the codebase service can be directly provided by remote object servers (solution 3). That is, the peers exporting remote services function as a codebase server for their own exported objects. This requires serving peers to provide at least two services: the actual remote object(s), and the class file server. This can lead to a notable overhead, especially in the presence of NAT boxes. However, the overhead is constant since only one class file service is needed for all exported remote objects. The class file server basically is a stripped down HTTP server that only accepts GET requests for class files. These requests are satisfied by serializing the requested classes directly from the peer's local classpath, requiring no additional storage space.

An enhanced class file service can also be outsourced to super-peers (solution 4). As in solution 2, the class files are stored in the DHT but the codebase service is provided by super-peers, that allow both the download and the upload of class files. Remote object servers that have access to the DHT can store class files directly in the DHT. Other remote servers use the super-peer's class file service to upload the code that is then stored in the DHT by the super-peer. In all cases, remote object servers set the codebase property to any known and available super-peer.

Clients then can use the super-peer as if it were a dedicated codebase. This relieves ordinary peers from the need to provide an additional service at the cost of putting more load on the super-peers. But it allows any super-peer to locate and provide every class file stored in the DHT and as such provides a codebase service that is distributed over the whole P2P-RMI network. In addition it is transparent to both clients and remote object servers, since they only need to know the address of any available super-peer to distribute and retrieve class files.

3.6. P2P-RMI ARCHITECTURE

Figure 7 shows an overview of the P2P-RMI architecture. The framework is lightweight, modular and extremely flexible. Depending on the desired levels of transparency and scalability nearly all components can be individually implemented, deployed, extended, and even replaced on demand. A P2P-RMI network can be very heterogenous with services being distributed among peers, super-peers, gateways and agnostic servers in any desired combination. Non-transparent P2P-RMI networks are possible through utilizing DHT based, local-only registries and codebases.

Distribution of the Registry service can be achieved through a Local RMI Registry, used by P2P-RMI peers to bind and lookup remote objects to and from the DHT. The Remote RMI Registry exposes the registry service to P2P-RMI agnostic clients and object servers. It can be deployed on super-peers or dedicated gateways alike.

Distributing the codebase is slightly more complex. P2P-RMI peers could directly store class files to the DHT and load them using a *Custom ClassLoader* at the cost of losing transparency. Alternatively, peers store class files in the DHT but provide a codebase property set to known a *ClassFile GetServer*. This service can be provided by the remote object server itself, any super-peer or other dedicated gateway. In addition, the class file service can offer both down- and upload functionality (*ClassFile Server*) to also allow P2P-RMI agnostic remote object servers to upload class files. Again, this additional functionality can be provided by super-peers or dedicate servers.

Finally, reachability of exported remote objects is guaranteed by using *Custom Socket Factories*, which, depending on the environment and application demands, provide NAT-traversal methods. Since both the codebase as well as the custom socket factories are metadata of the remote object's stub, any client can transparently make use of them.

4. EVALUATION

We implemented the Local RMI Registry with a DHT adapter to evaluate the performance of a distributed RMI registry with respect to

- bootstrapping a DHT peer
- rebinding a remote service with the registry
- looking up a remote service at the registry

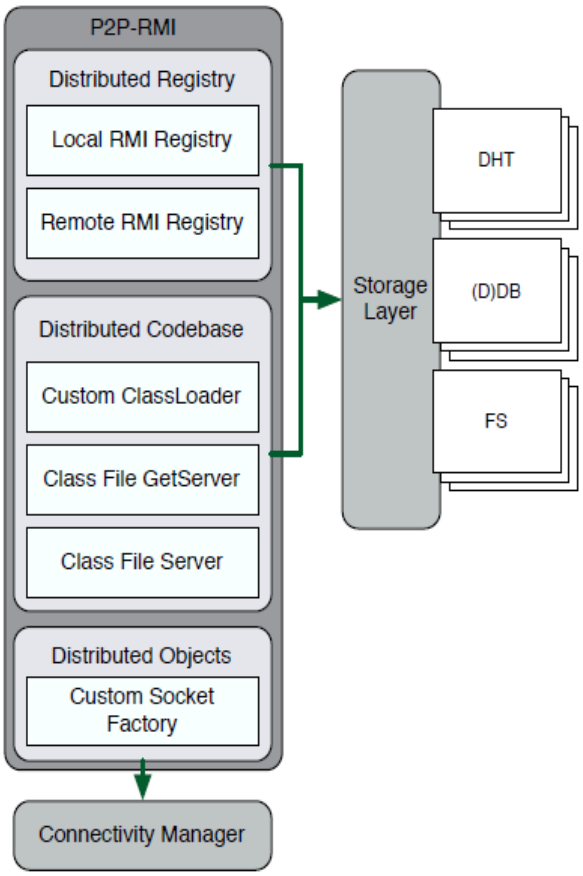

Figure 7: P2P-RMI architecture. Distributing RMI services can be divided into three challenges that can be individually addressed and implemented

Our analysis consists of two parts; we first examine the behavior and performance of P2PRMI servers and then independently examine P2P-RMI clients. Each server evaluation runs the following sequence:

1. create and bootstrap a new peer
2. create and register a new remote object instance
3. rebind the remote object with the registry
4. unbind the remote object from the registry
5. destroy DHT instance and peer

To evaluate client performance, we run the following sequence of client operations:

1. create and bootstrap a new peer
2. look up a remote object through the local P2P-RMI registry
3. invoke a method through the received stub
4. destroy DHT instance and peer

Our implementation uses the DHT of the BitTorrent client Vuze[2] as a storage backend. The Vuze DHT is a modified Kademlia [14] implementation and even though it is mainly intended for distributed trackers and tracker backup, its API allows writing arbitrary key/value pairs. Vuze is a very popular BitTorrent client and the Vuze DHT is one of the mostly used DHTs

world-wide. The high amount of Vuze users and DHT nodes allow testing on a world-wide distributed platform. We also take this as an opportunity to provide an evaluation of the Vuze DHT itself. In the following, we shortly review Vuze's Kademlia implementation and how it affects the P2P-RMI platform, and then proceed to present the performance results.

4.1. VUZE DHT

Apart from an incomplete wiki page[3] and open source code there is no documentation about Vuze's DHT implementation. Being an extended Kademlia DHT, the Vuze DHT supports Kademlia's four essential message types:

- PING: Check nodes for availability and keep routing tables up-to-date.
- STORE: Store a (key, value) or a (key, array of values) pair.
- FIND_NODE: Find the nearest node for a given key.
- FIND_VALUE: Find the nearest node for a given key and retrieve the associated
- value(s).

In addition to these standard Kademlia message types Vuze also features some extensions like anti-spoofing, verification and NAT-traversal. Please note that there is no message type for the removal of an entry. Since entries are stored on many nodes which can freely join and leave the network, each node has to periodically republish its key mappings. An entry automatically expires after a certain amount of time, and is effectively removed if it is not republished periodically. For that reason, we do not include unbind performance in our evaluation, since it does not bear significance. Standard Kademlia uses a republish period of one hour that can even be increased under certain circumstances [14]. In [8, 5] the authors claim that entries in the Vuze DHT remain valid up to 8 hours after removal and that the peer's local data store is republished every 30 minutes. However, our empirical analysis has shown that Vuze nodes republish their mappings once every minute and that the entries remain valid up to one hour. It is unclear why Vuze uses such short intervals between publications. We assume that the intention is to prevent entries from being manipulated by malicious peers and that this falls into Vuze's anti-spoofing mechanisms.

Since there is no active removal of entries, the Vuze implementation simply removes the entries from the local data store, thus effectively stopping the republication of the entry. As a result, if a remote service becomes unavailable, replications of its stub will still reside in the DHT for a period of about an hour. Clients that lookup the remote service would retrieve a stale entry and try to connect to an unavailable service. The remote registry could check service availability prior to returning the stub and only satisfy the lookup request if the service is still available. An alternative is to push responsibility to the client side. This approach seems reasonable because in a distributed system, clients need to be equipped with mechanisms to deals with stale remote references anyway.

4.2. P2P-RMI VUZE DHT ADAPTER

Due to Vuze's extended implementation of Kademlia there are certain pitfalls that need to be addressed when implementing a storage adapter for the Vuze DHT. For one there is a maximum packet size of 512 bytes. A stub or class file that exceeds the maximum byte limit has to be split into multiple parts for storage in the DHT. Fortunately, the Vuze DHT implementation supports the storage of (key, array of values) pairs. We exploit this feature to split longer stubs into multiple parts and save them under the same key (To be specific, the key

is also split to indicate the presence of a split stub and the need for reconstruction. For the sake of simplicity we abstract from this detail).

Multiple values can also be stored under a common key by different nodes. As mentioned earlier, the Vuze DHT is used as a distributed tracker. That is, the stored keys correspond to a file's infohash value (an identifier for a shared file), while the stored values represent file sources. Any peer that is a file source adds itself to the value array. While this is a very nice feature that allows storing collections of stubs or classes under single keys, it raises the question of how to deal with multiple return values. Currently, our implementation neglects this fact and simply considers the first returned value only. A more sophisticated approach could make better use of value arrays. Examples include a service directory for P2P-RMI registries. This directory could list all services that are present in the network (allow implementation of list()). In addition, each listed service can be associated with a list of service providers. In this case the P2P-RMI registry could also provide load balancing functionality, by only returning the service closest to the requesting client or that has the least load. Regarding the codebase, instead of storing each class file individually, affiliated classes could be stored under a common key. The class file server would then collect all affiliated classes with one lookup and cache them locally to reduce pressure on the network by multiple lookups. This would also allow independent codebases that are addressed by unique identifiers.

Vuze also implements some security related features like spoofing protection and storage verification, although, it remains unclear what techniques are applied. We believe that in this context Vuze also limits the amount of store requests issued by a node within a specific time frame. Our evaluation has shown that multiple fast subsequent stores lead to a state in which publications are no longer possible for a period of time. In this case we had to force-close the affected node and start a fresh one. This made rebind evaluation rather difficult and unfortunately limits Vuze's applicability for P2P-RMI. With theses restrictions a server can only offer a limited amount of remote services before its store quota is exhausted.

4.3. RESULTS

Table 1 summarizes the times we measured for all evaluated P2P-RMI registry operations. The server evaluation ran 100 iterations of bootstrap and rebind operations, while the client evaluation ran 400 iterations of bootstrap and lookup operations. This explains why we measured a total of 500 bootstrap operations.

Table 1: Measured minimum, maximum, average and standard deviation for all operations

	N	min	max	avg	stdev
bootstrap	500	4.54 s	87.43 s	32.48 s	11.50 s
rebind	100	39.95 s	134.43 s	94.28 s	17.46 s
lookup	400	0.35 s	44.23 s	3.54 s	8.42 s

It can be seen that the measurements have high variance. This is especially notable for rebinds, which vary between 40 and 134 seconds. Bootstraps and lookups show a similar, but less pronounced, behavior. Curiously, lookup operations perform much faster than rebinds. Intuitively, one would assume both to perform similarly, especially because the Vuze DHT documentation[3] states that "... store is performed on [...] 20 nodes [...] lookup does request [...] data from 20 nodes". However, store messages are significantly larger and must be propagated to every node that is a storage candidate or that is close to one. When a requesting

node receives a response, it will update its routing table to increase performance. Lookups can then be satisfied much faster, because the path leading to the storing nodes is more efficient.

Table 2 shows the correlation matrix (using Pearson's correlation coefficient) of the three measured operations. As expected, there is no significant correlation neither between bootstraps and rebinds nor between bootstraps and lookups.

Table 2: Correlation matrix of operation performances

	bootstrap	rebind	lookup
bootstrap	1.000	−0.231	−0.047
rebind	−0.231	1.000	NA
lookup	−0.047	NA	1.000

Figure 8 shows the histogram, normal distribution as well as the estimated kernel density of the bootstrap operation. The measured and the estimated bootstrap performance closely resemble a

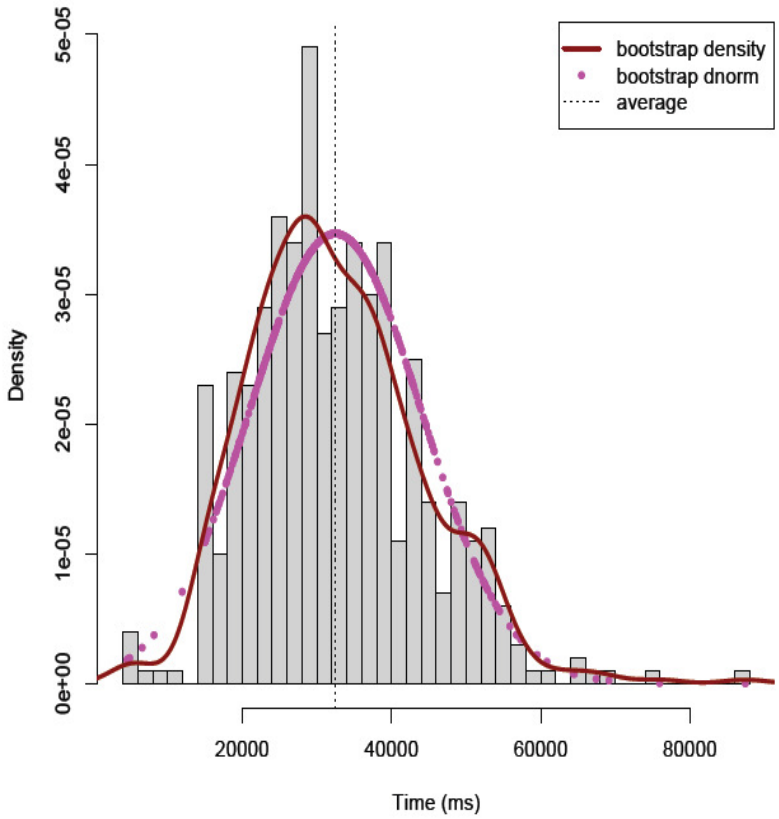

Figure 8: Histogram, normal distribution and estimated kernel density of the bootstrap operation

normal distribution with its peak at 32.5sec. The similarity between the two curves indicate the high statistical significance of our measurement results. Because bootstrapping is a necessary step in P2P systems, improvement of its performance would certainly be beneficial; however,

because bootstrapping is performed exactly once per peer - in our context: once per RMI server and once per RMI client -, the measured performance seems acceptable for many applications.

Analogously, Figure 9 shows the histogram, normal distribution as well as the estimated kernel density of the rebind operation. Rebinds, like bootstraps, closely follow a normal distribution, with the peak at around 95sec. Rebinds have, however, a much greater variance, and can take quite a long time. This can pose a problem for servers that want to serve many remote objects. In such a case it would be best to aggregate store requests and reduce the number of messages. Naturally, store messages can also be sent in parallel, the server does not have to wait for a reply before registering the next remote object.

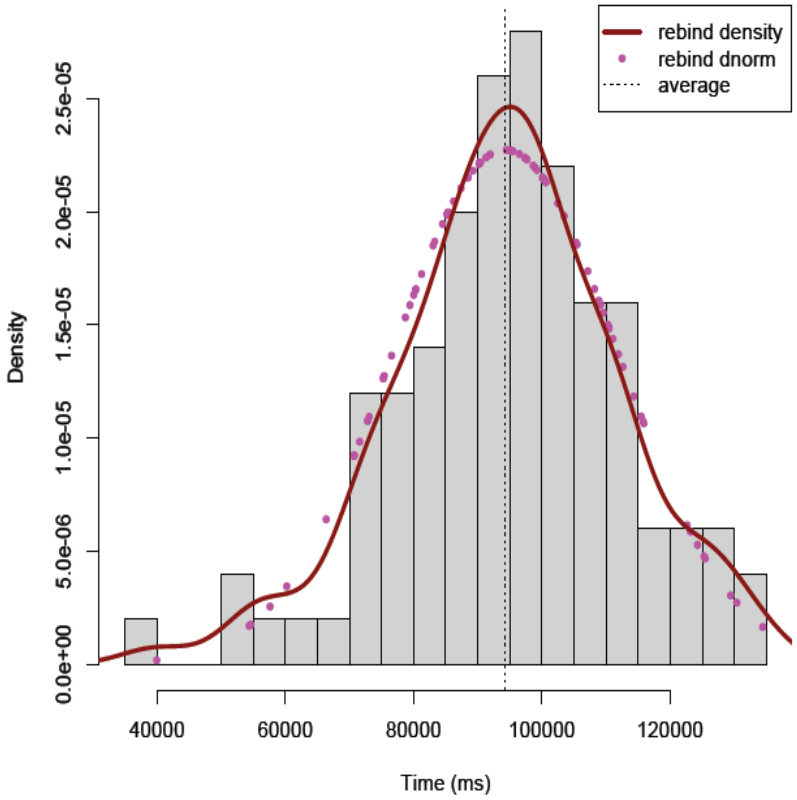

Figure 9: Histogram, normal distribution and estimated kernel density of the rebind operation

A combined average bootstrap and rebind duration of approximately 2 minutes might be acceptable for the server side of many applications. However, faster DHT store operations would clearly be desirable, to make the rebind times acceptable for an even broader array of applications.

Finally, Figure 10 shows the histogram, distribution and estimated kernel density of the lookup operation. Compared to bootstraps and rebinds, lookups show quite a different behavior. While the great majority of about 80% of lookup operations succeed within 1sec, the remaining values seems almost randomly scattered over a range of approximately 44sec, which is the longest measured lookup time. There seems to be a second, small peak at about 32secs whose statistical significance is not clear.

From an application and user perspective the lookup performance is most crucial. As can be seen, for a great lot of lookup requests, response times are very short. The few spikes, however, may pose serious problems to client applications, especially when many stubs are requested sequentially. While the reason remains unclear, we assume the long response times to be related to churn and the respective reorganization of the stored data. Vuze stores the data on the 20 nodes that are closest to the key space. Each time one of those nodes leaves the network, or another one that is closer to the key space arrives, the data has to be reorganized and routing tables have to be updated. Since Vuze requests the values from all those 20 nodes for verification reasons, lookup performance will suffer at times when data and routing tables are currently being restructured.

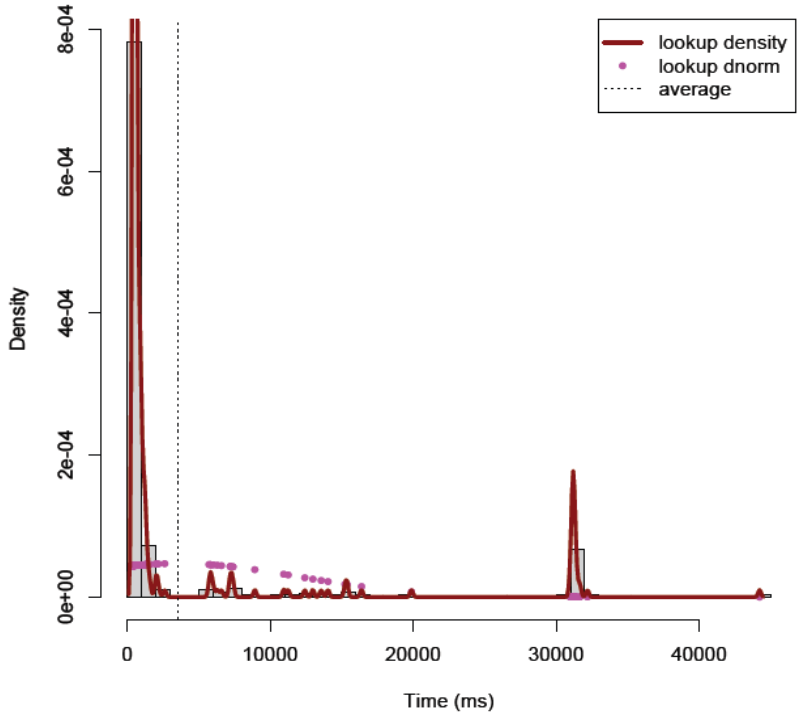

Figure 10: Histogram, normal distribution and estimated kernel density of the lookup operation

4.4. DISCUSSION

Performance of the Vuze DHT leaves room for improvement, both in terms of average response times and in terms of predictability. Especially store requests can take quite long, but also lookups take a long time under certain, unpredictable circumstances. Vuze is a classic file sharing infrastructure, and thus it does not come as a surprise that its DHT is optimized towards distributed tracker lists. Considering this kind of application, performance of the Vuze DHT is certainly acceptable. For distributing applications, however, it might not be the best choice. While rebind duration is acceptable for most applications, a long lookup time may not be since this requires special care on the client side.

A DHT targeted towards P2P-RMI applications needs to specifically optimize lookup time. Unfortunately, this competes with storage verification. However, there might be other solutions than retrieving the value from all storage nodes. A subset of nodes might be enough, the stored values could be signed, or verification could be pushed to a higher layer where churn and network delay is not present. It is also worth mentioning that churn might not be as high in a P2P-RMI network as in a file sharing network since the semantics of distributed computing demands a higher degree of availability for both servers and clients. To be of any use in a distributed application, servers must stay available at least long enough to compute invoked methods while clients require the results for further processing. In this regard, a dedicated P2P-RMI DHT that is optimized for distributed applications might be a better solution. On the downside, an exclusive DHT requires a critical mass of storage nodes to be effective and efficient. A solution would be to have alternative backup storage available, either in the form of dedicated database servers or other DHTs like Vuze.

Even though the results suggest that the distribution of object-oriented applications on P2P-RMI networks is not yet generally applicable, we strongly believe that the proposed methodology is sound, and that an optimized distributed storage can satisfy performance demands.

5. CONCLUSION AND FUTURE WORK

We presented a novel solution to build P2P-RMI networks. All RMI related services and remote objects can be transparently distributed and served as well as consumed by peers and unmodified clients and servers alike. Our approach is lightweight, flexible, and scalable since it can be used in practically any environment of any desired size. The level of transparency of each RMI service can also be individually configured. In addition we contributed a performance evaluation of a very popular and widely used DHT implementation and discussed its applicability for distributed applications.

Still, there are some open issues. A major concern is security. Currently, there is no possibility of validating the origin of stubs and remote objects except those offered by the specific DHT implementation. An attacker could try to replace the stubs and class files in the DHT, inject malicious code, and effectively prevent legit services. While the evaluated DHT can and does prevent falsifying data, this is done at the cost of decreased performance. Malicious super-peers could be used to compromise RMI related services, return altered stubs and classes, hijack connections, and perform other man-in-the-middle attacks. Whether these security issues can be addressed regarding the P2P-RMI layer or whether they have to be considered in the storage, application, or even an intermediate layer, and how they affect overall performance, must still be evaluated.

Apart from solving security issues a fully functional P2P-RMI platform needs to be implemented and thoroughly evaluated. Currently, the local and remote RMI registries, DHT adapters, and NAT-traversal are available, leaving the distributed codebase missing. Considering the retrieval of class files, lookup performance is equally crucial than it is for stubs. Further evaluation of available DHTs - like OpenDHT, FreePastry, or OpenKad - is due to improve performance, address security issues, and identify requirements a P2P-RMI exclusive DHT must satisfy.

Future work also includes building a distributed social network that focusses on peer-based applications and services. Other possibilities that still have to be explored include a middleware

that can automatically distribute applications on the P2P-RMI platform. This could lead to self-balancing, object-oriented cloud services based on a P2P platform.

6. REFERENCES

[1] Remote method invocation. online. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp, retrieved 23.01.2012.

[2] Vuze. online. http://www.vuze.com/, retrieved 04.04.12.

[3] Vuze dht. online. http://wiki.vuze.com/w/Distributed_hash_table, retrieved 16.04.2012.

[4] R. Bless, O. Waldhorst, C. Mayer, and H. Wippel. Decentralized and Autonomous Bootstrapping for IPv6-based Peer-to-Peer Networks. In *Winning Entry of the IPv6 Contest 2009 by IPv6 Council*, May 2009.

[5] Scott A. Crosby and Dan S. Wallach. An Analysis of BitTorrent's Two Kademlia-Based DHTs, 2007.

[6] Jochen Dinger and Oliver Waldhorst. Decentralized Bootstrapping of P2P Systems: A Practical View. In Luigi Fratta, Henning Schulzrinne, Yutaka Takahashi, and Otto Spaniol, editors, *NETWORKING 2009*, volume 5550 of *Lecture Notes in Computer Science*, pages 703-715. Springer Berlin / Heidelberg, 2009.

[7] Chis GauthierDickey and Christian Grothoff. Bootstrapping of Peer-to-Peer Networks. In *Proceedings of DAS-P2P*, Turku, Finland, August 2008. IEEE, IEEE.

[8] Roxana Geambasu, Tadayoshi Kohno, Amit Levy, and Henry M. Levy. Vanish: Increasing Data Privacy with Self-Destructing Data. In *Proc. of the 18th USENIX Security Symposium*, 2009.

[9] Stéphane Genaud and Choopan Rattanapoka. *A Peer-to-Peer Framework for Message Passing Parallel Programs*, volume 17 of *Advances in Parallel Computing*, pages 118-147. IOS Press, June 2009.

[10] Oliver Haase, Daniel Maier, and Jürgen Wäsch. Punching Holes with Java RMI. Dr.Dobb's, May 2009.

[11] Oliver Haase, Jürgen Wäsch, and Bo Zhao. A Remote Java RMI Registry. Dr.Dobb's, November 2008.

[12] Daniel Maier, Oliver Haase, Jürgen Wäsch, and Marcel Waldvogel. NAT hole punching revisited. In LCN, pages 147-150, 2011.

[13] Qusay H. Mamoud. Getting Started With JavaSpaces Technology: Beyond Conventional Distributed Programming Paradigms. July 2005.

[14] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53-65, London, UK, UK, 2002. Springer-Verlag.

[15] Rubén Mondéjar, Pedro García-López, Carles Pairot, and Lluís Pamies-Juarez. CloudSNAP: A transparent infrastructure for decentralized web deployment using distributed interception. *Future Generation Computer Systems*, (0), 2011.

[16] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. *USENIX Computing Systems*, 9, 1996.

[17] Bernard Wong and Emingün Sirer. Approximate Matching for Peer-to-Peer Overlays with Cubit, 2008.