STUNT Enhanced Java RMI

Oliver Haase, Wolfgang Reiser, and Jürgen Wäsch

Computer Science Department,
Konstanz University of Applied Sciences,
Constance, Germany

Abstract. Java RMI uses HTTP tunneling for NAT traversal. While HTTP tunneling is a valid technique for traditional client–server–architectures, it is too heavy-weight for highly distributed systems such as peer-to-peer applications. In this paper, we propose a STUNT enhanced RMI mechanism that takes advantage of the hole punching NAT traversal technique that many successful peer-to-peer applications use. Because the modified communication behavior is made part of the RMI server stub, our approach is fully transparent to the RMI client.

Keywords: Java RMI, NAT traversal, STUNT, TCP hole punching.

1 Introduction

Java's platform independence and built-in networking support have made it an interesting language for distributed computing. One key feature for the development of distributed Java applications is the Java Remote Method Invocation (RMI) technology [6]. The main idea behind RMI is to provide a communication mechanism that allows developers to use the method invocation paradigm to communicate between remote objects. This simplifies the development process because programmers do not need to deal with communication protocols, on-the-wire data representation, and connection management. The RMI communication protocol, Java Remote Method Protocol (JRMP), which is used for all communication between RMI clients and servers, builds on top of the TCP/IP protocol stack. In order to seamlessly connect RMI clients and servers throughout the Internet, the ubiquitous reachability of all involved entities via TCP/IP is essential.

Since the early days of the Internet—when every node had its own globally unique IP address and could be addressed directly—things have changed as a result of security considerations and a shortage of IP addresses. The old addressing model has been replaced with a new addressing model consisting of one global address realm and innumerable private address realms linked by Network Address Translators (NAT) [10].

Although this model is suitable for traditional client–server communication where at least the server is in the global address realm and can be reached directly, it makes it complicated for scenarios where the server, or both server and client, reside in different private networks. This kind of scenario is clearly on

R. Meier and S. Terzis (Eds.): DAIS 2008, LNCS 5053, pp. 269–274, 2008.

the rise with the spread of peer-to-peer technologies beyond use of file-sharing. Therefore, a solution to make RMI communication work even in the presence of NAT is essential for RMI to become a viable technology for modern peer-to-peer and other highly distributed applications.

The RMI built-in solution for NAT traversal, HTTP tunneling, requires significant administrative overhead and in many cases is in conflict with corporate security policies. In addition, the approach is not feasible for home users whose entire network sits behind the Internet service provider's NAT box. Another drawback is the bandwidth inefficiency due to the tunneling overhead.

Our primary focus is to develop a light-weight RMI NAT traversal technique with minimimal overhead that works with no changes to RMI clients and minimal changes to RMI servers. Therefore, we propose an enhanced RMI mechanism for NAT traversal that is based on a technique known as *hole punching* (first mentioned in [8]). Even though the name suggests otherwise, hole punching does not compromise the security of private networks, but rather empowers applications to communicate within the security policies of different types of NATs (e.g., cone NAT, restricted cone NAT, port restricted cone NAT, symmetric NAT [5]). There are a number of commerical applications which use hole punching techniques — Skype [12,11] is certainly one of the best known of them.

In order to use hole punching, nodes must have the capability to identify the presence and type of NAT they are behind, as well as their public IP address/port combination. One way to gather all this information is to use a public *STUNT* (*Simple Traversal of UDP Through NATs and TCP too*) server. STUNT is a protocol presented by Guha et al. in [3,4] which extends the STUN protocol [9] with TCP capabilities.

This paper shows the use of the hole punching technique to establish an RMI communication between two NATed RMI parties. In order to make this hole punching technique work, both parties have to go through several steps in their communication process. They first have to determine whether they are behind a NAT and which their public IP address/port combination is. After they have collected this information with the help of a public *STUNT server*, both parties have to publish the results through a public *Rendezvous Server*. To set up a communication, party *A* polls the data of the public communication endpoint of the other party from the Rendezvous Server. This polling automatically triggers a mechanism which pushes party *A*'s public IP address/port combination to party *B*. After this step, both parties have each other's address data which enables them to perform the actual hole punching process [8] to set up the communication between *A* and *B*.

In the following, we describe how to integrate STUNT-based hole punching into the RMI communication concept. Our solution makes use of a custom `RMISocketFactory` to modify the RMI connection behavior: after a failed direct RMI connection attempt, both RMI client and RMI server go through the STUNT-enhanced RMI communication process to set up a connection between the NATed RMI parties using hole punching.

2 STUNT Enhanced Java RMI Solution

Integrating TCP hole punching into Java RMI evidently changes the way Java RMI communication usually takes place. One of our top goals, however, is to change the RMI mechanism *without the need to modify any RMI client*, i.e., whether it uses a regular or a STUNT enhanced RMI server object should be transparent to the client. For the server object it is acceptable to implement behavior specific to STUNT enhanced RMI; the changes to regular RMI should nevertheless be minimal.

To show how we achieved the above mentioned transparency, a few words about Java RMI are helpful: An RMI client stub, i.e., the local proxy of the remote server object, consists of two parts, the `RMISocketFactory` and the actual `RemoteReference`. The `RMISocketFactory` controls the instantiation of the sockets used for RMI communication, and hence controls the communication behavior itself. This technique makes the `RMISocketFactory` the ideal hook point to alter the RMI communication while still complying to standard RMI on the API level. We thus replace the standard `RMISocketFactory` with a custom `RMISocketFactory` to transparently change the RMI communication behavior on the client side. The second component of the server stub, the `RemoteReference`, defines the IP address or the hostname of the RMI server object. When the server object is exported, i.e. when the client stub is created, the value of the Java property *java.rmi.server.hostname* is copied into the `RemoteReference`. Setting this property to a publicly reachable IP address pushes the desired address into the `RemoteReference`.

An RMI client obtains a server stub in one of two ways: It either (1) gets it as a return value or return parameter from another remote server object, or (2) it uses the *RMI registry*, the RMI specific naming service. Because the communication process in case (1) is a mere subset of the process in case (2), we focus on case (2) in the following.

The RMI registry is both an API specification and a reference implementation which is part of Sun Microsystem's Java SE. For security reasons (or a lack of proper authentication and authorization mechanisms), the reference implementation needs to run on the same machine as the RMI server object, a restriction which is not acceptable for our solution exactly because the server machine can sit behind a NAT box. Part of our solution is therefore a custom RMI registry that can register server objects from other machines. Again, for an RMI client, this change is transparent because the client uses the standard API to locate and query the custom RMI registry.

2.1 Communication Process

The sequence diagram in figure 1 shows the STUNT enhanced RMI communication process. This process is divided into server (ⓐ to ⓒ) and client (① to ⑥) behavior.

In step ⓐ, the server interrogates a STUNT server to learn its public IP adress and NAT type. It then sets the *java.rmi.server.hostname* property to its

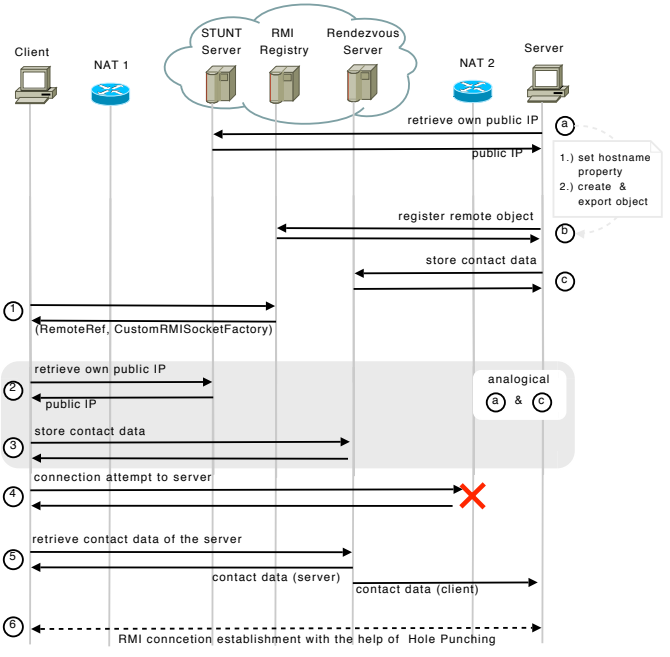

Fig. 1. STUNT RMI communication process

public IP address, creates and exports the server object and registers it with the naming service, i.e. the custom RMI registry (step ⓑ). Please note that when exporting the server object, the `RMISocketFactory` part of the stub is set to our custom `RMISocketFactory`. The server is now ready for incoming remote method invocations.

The client uses the standard RMI API to locate the naming service and look up the stub for the remote server object ①. After this step, the custom `RMIClientSocketFactory` controls the further communication process, *while the client still executes regular remote method invocations*. Steps ② and ③ are analog to steps ⓐ and ⓒ on the server side; the client interrogates its public IP address and NAT type, and has this data stored in the rendezvous server. In step ④, the client tries to directly connect to the remote object. This connection attempt succeeds only if the remote object is either in the same private network or belongs to the public Internet. If the direct connection attempt fails, the client contacts the rendezvous server (step ⑤) to retrieve the remote server object's contact address. The rendezvous server not only returns the remote object's public IP address, but also pushes the RMI client's public IP address to the RMI server object. After both hosts have received the other party's contact address they hole-punch their NAT boxes to establish a connection between each other. More specifically, each party's attempt to contact the other side establishes a temporary mapping in their NAT box that allows the other party to traverse the far side's NAT box.

2.2 Components

Our STUNT enhanced RMI mechanism comprises the following entities, some of which are standard off-the-shelf components, while others require custom implementations.

STUNT Server: The STUNT Server implements an extended version of the STUN protocol which includes TCP capabilities. The STUNT server determines the global IP addresses and ports which are assigned by the outermost NAT, assert its type and transmits them back to the host. Public STUNT servers are readily available in the Internet, and can be employed without changes.

Naming Service: Part of our solution is a RMI registry compliant custom naming service, which is enhanced with security features like access control to ensure that remote objects can be managed in a secure way. Keeping the naming service compliant to the standard registry ensures that clients can still use the *LocateRegistry* interface to connect to the service. This naming service must be publicly reachable and can, e.g., be a part of the rendezvous server.

Rendezvous Server: The rendezvous server is a publicly reachable server, providing a mapping service which maps global unique identifiers onto communication endpoint information. As shown in figure 1, both STUNT-RMI client and STUNT-RMI server register their contact address with this server. Appropriate unique identifiers are URIs with the form `<user>@<host>` because these URIs are unique and valid despite NAT boundaries. In order to keep the mapping table clean and to prevent entries from becoming stale, we propose to add a lease time for each entry and to set appropriate intervals to clear out old data.

Custom `RMISocketFactory`*:* As mentioned before, the `RMISocketFactory` is responsible for the RMI client/server communication behavior. In our solution the RMISocketFactory implements the STUNT enhanced communication behavior as shown in figure 1.

3 Conclusion and Future Work

All of the necessary components—except for the STUNT server several instances of which are publicly available in the Internet—are currently under development. As soon as the implementation work is completed, we will evaluate the solution in terms of operability, performance, and scalability. The results will be made public to the research community.

In a previous project, we have developed a neighbor-centric peer-to-peer infrastructure based on Java RMI communication [7]. That project has been the main driver for the light-weight NAT traversal solution presented in this paper. Consequently, the implementation of the STUNT enhanced RMI approach will be integrated into our peer-to-peer infrastructure. We believe, however, that our

solution has the potential to be useful for other Java RMI based projects, infrastructures, and middlewares. Or to put it differently, we believe that the lack of a light-weight, zero-configuration NAT traversal solution is a major obstacles for a more widespread use of Java RMI in large-scale, industry grade distributed applications. We therefore plan to make the resulting software and servers available for public use.

References

1. Biggadike, A., Ferullo, D., Wilson, G., Perrig, A.: NATBLASTER: Establishing TCP connections between hosts behind NATs. In: Proceedings of ACM SIGCOMM ASIA Workshop (2005)
2. Ford, B., Srisuresh, P., Kegel, D.: Peer-to-peer communication across network address translators. In: Proceedings of the 2005 USENIX Annual Technical Conference (2005)
3. Francis, P., Guha, S.: Simple traversal of UDP through NATs and TCP too (STUNT), `http://nutss.gforge.cis.cornell.edu/`
4. Francis, P., Guha, S., Takeda, Y.: NUTSS: A SIPbased approach to UDP and TCP network connectivity. In: SIGCOMM 2004 Workshops (2004)
5. Francis, P., Guha, S.: Characterization and Measurement of TCP Traversal through NATs and Firewalls. In: Proceedings of Interet Measurement Conference (IMC) (2005)
6. Grosso, W.: Java RMI - Designing & Building Distributed Applications. O'Reilly & Associates (2002)
7. Haase, O., Todt, A., Wäsch, J.: A Peer-To-Peer Ring Infrastrucure for Neighbor-Centric Applications. In: Enokido, T., Barolli, L., Takizawa, M. (eds.) NBiS 2007. LNCS, vol. 4658, Springer, Heidelberg (2007)
8. Holdrege, M., Srisuresh, P.: RFC3027 - Protocol Complications with the IP Network Address Translator (2001), `http://tools.ietf.org/html/rfc3027`
9. Huitema, C., Mahy, R., Rosenberg, J., Weinberger, J.: RFC3489 - STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs) (2003), `http://tools.ietf.org/html/rfc3489`
10. Holdrege, M., Srisuresh, P.: RFC2663 - IP Network Address Translator (NAT) Terminology and Considerations (1999), `http://tools.ietf.org/html/rfc2663`
11. Schmidt, J.: The hole trick – How Skype & Co. get round firewalls. Heise Security (2006) [online 2007-11-21], `http://www.heise-security.co.uk/articles/82481`
12. Skype Limited: Guide for Network Administrators (2005) [online, 2007-11-21], `http://www.skype.com/security/guide-for-network-admins.pdf`
13. Sun Microsystems, Inc.: JXTA Java Standard Edition v2.5: Programmers Guide (2007) [online 2007-12-03], `https://jxta-guide.dev.java.net`