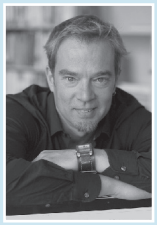
A COMPARATIVE ANALYSIS OF NAT HOLE PUNCHING

Daniel Maier, Oliver Haase, Jürgen Wäsch, Marcel Waldvogel

Daniel Maier

2004 Abitur am Friedrich-Hecker-Gymnasium in Radolfzell, danach Studium der Wirtschaftsinformatik an der HTWG Konstanz. 2009–2011 Masterstudium der Informatik ebenfalls an der HWTG Konstanz. Während und nach dem Mastersstudium wissenschaftlicher Mitarbeiter im Fachbereich Informatik an der HTWG Konstanz. Seit Oktober 2011 Software-Entwickler bei der Bosch Software Innovations GmbH.



Prof. Dr. Oliver Haase

Studium der Informatik an der Universität Karlsruhe, danach Promotion zum Dr.-Ing. an der Universität Siegen. 1998–2005 Industrieforschung, zuerst bei NEC Europe in Heidelberg, dann bei den Bell Labs in Holmdel, New Jersey. Seit Sept. 2005 Professor für Verteilte Systeme und Software Engineering an der Fakultät Informatik der HTWG Konstanz. Zahlreiche Konferenz- und Zeitschriftenpublikationen, Patente, Bücher und Buchkapitel. Seit März 2011 Leiter des Informations- und Medienzentrums der HTWG.

Prof. Dr. Jürgen Wäsch

Studium der Informatik und Wirtschaftswissenschaften an der Universität Kaiserslautern. 1993–1999 GMD – Forschungszentrum Informationstechnik GmbH in Darmstadt. 1997–1998 externer Berater bei der Software AG. 1999 Promotion zum Dr.-Ing. an der TU Darmstadt. 2000–2003 Bereichsleiter bei der e-pro solutions GmbH in Stuttgart. Seit 2004 Professor für E-Business Technologien und Datenbanksysteme an der HTWG Konstanz. 2008 Forschungssemester bei der SAP AG. Seit 2009 Studiendekan und Prodekan der Fakultät Informatik sowie Studiengangsleiter Wirtschaftsinformatik.

Prof. Dr. Marcel Waldvogel

Studium zum Eidg. dipl. Informatik-Ing. an der ETH Zürich (1994). Während des Studiums Gründung eines Startup-Unternehmens im Bereich Objektorientierte Software. 1996–1999 Promotion zum Dr. sc. techn., ebenfalls an der ETH Zürich. Assistenzprofessur an der Washington University in St. Louis 1999–2001, danach bei IBM Research im Zurich Research Laboratory. Seit Ende 2004 Professor für Verteilte Systeme und Leiter des Rechenzentrums an der Universität Konstanz.

IPv4's address space is getting exhausted any day now, as addresses only consist of 32 bits. The success of the Internet made it clear already in 1994 that this address space would not last. However, IPv6 is still only scarcely supported. Instead, the use of Network Address Translation (NAT) boxes to hide entire networks behind a single IPv4 address is the dominant solution. Sometimes, this is even done multiple times, e.g., most mobile network operators use NAT for their entire set of mobile devices, which in turn may offer a bluetooth or wireless local area network for 'tethering' support, hiding again behind the single NATted address of the smartphone.

This works great as long as the machines behind the NAT box only initiate outgoing connections and do not have to accept incoming requests. However, increasingly interactive Internet applications prefer direct contacts, where a central server would only increase latency, limit throughput, or become a single point of failure. Direct connections are essential to such distinct applications ranging from interactive games and general peer-to-peer applications to VoIP and file transfers among instant messaging partners. Other reasons to contact machines behind NAT include the wish to access data on your home machine or providing screen sharing for collaboration or support.

NAT hole punching is one technique to traverse NAT boxes. It has the advantage of not requiring any user configuration, and establishes direct connections between two peers without the need for additional relay servers. Hole punching is suitable for UDP and TCP. For TCP, two main options exist, namely sequential and parallel hole punching. These are the main targets of our analysis. We compare them according to various criteria in different scenarios.

Even when IPv6 should ever become wide-spread, NAT and thus hole punching will not become obsolete. NAT will be one of the main techniques for IPv6<->IPv4 translation [8], despite the declarations of the IETF to the contrary [1]. It can also be assumed that some people and organizations will continue to use IPv6<->IPv6 NAT for the believed security and privacy bene.t, despite the fact that NAT does not replace a real firewall and the availability of special privacy mechanisms in IPv6 [6].

1 NAT TRAVERSAL

Several techniques have been invented to establish connections to machines behind NAT boxes. Many of them require some control data being exchanged with a machine on a globally reachable address. This machine, commonly called *Mediator,* keeps a directory of the machines behind NAT and is the endpoint of a persistently open control connection to these NATted machines.

A simple scenario is when only one machine, peer B, is behind NAT. Then, peer A, having a global address and wishing to set up a connection to B, will contact the mediator M with its own address and port combination. M will forward this connection request message along its persistent control connection to B, who will then set up a direct connection to A. This process is known as *connection reversal* and is possible, as the NATed host B has no restrictions in setting up a connection from inside to A's public address. This technique also works when B is behind multiple layers of NAT.

It gets hairy, though, when both A and B are behind NAT but wish to establish a direct connection. Then, a widely used solution is hole punching. Before going into the details of hole punching, let us revisit the basic functionality of NAT:

Mapping. The most basic functionality is the mapping of the internal address/port pair to the external pair and vice versa. The

source address and port combination of any packet leaving NAT is mapped using this forward translation, while any destination address and port from the outside will be mapped in reverse. Most of today's NAT boxes *employ endpoint independent mapping,* i.e., an internal host/port pair used for concurrent connections to multiple external hosts will use only a single external address/port pair. This endpoint independence is the only real requirement for hole punching but is standard in almost all NAT boxes (for exceptions to this rule, see Section 4.4).

Filtering. Even though the mapping may be endpoint independent, many NATs do restrict sending data back through a mapping to a list of peers the inside machine has already had contact with. Many NAT boxes do employ filtering; hole punching, however, is able to deal with it. If all NAT boxes were to suddenly cease filtering, hole punching could be slightly optimized.

Additional properties of NAT boxes include whether the port mapping is predictable. NAT boxes vary widely in their port allocation policies, so hole punching does not rely on predicting the mapping. If the NAT box did not provide endpoint independent mapping, port prediction would be required, however.

A nice property of hole punching is that it is independent of the number of layers of NAT boxes that shield our endpoints from the global address pool [4].

An in-depth description of the two most relevant hole punching techniques for TCP connection follows. We consider TCP because it is the protocol of choice for the exchange of non-real time data; in addition, TCP seems to become popular even for streaming data, as it is the transport protocol used by HTTP streaming. UDP, on the other hand, is not covered here. Its connectionless behavior, however, makes hole punching essentially trivial.

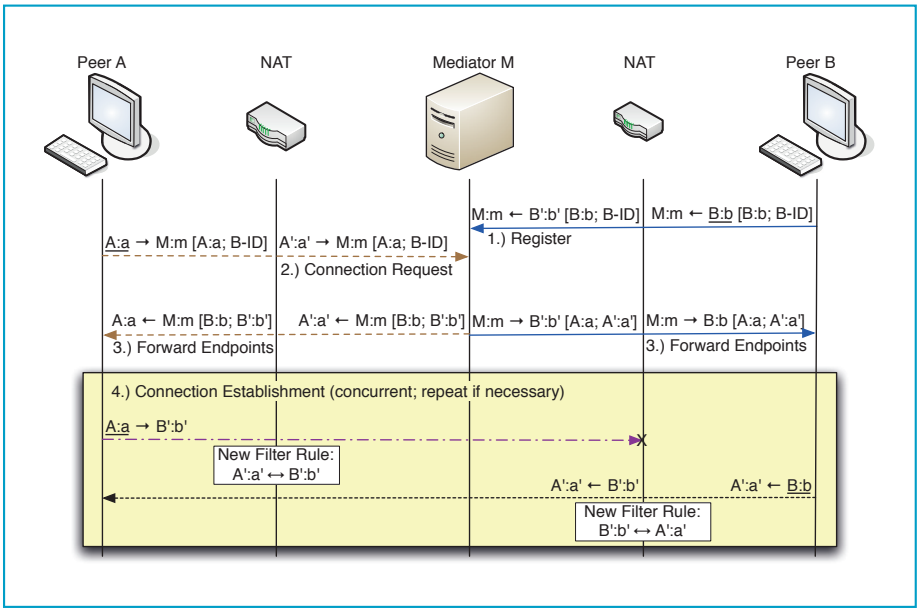


FIG. 1: Message sequence diagram of parallel TCP hole punching (adapted from [4]). The notation S:s -> R:r [P] denotes that a sender with address S and port s sends the payload p to a receiver with address R and port r.

1.1 Parallel TCP hole punching

Parallel TCP hole punching was first described in [4]. Figure 1 shows the message exchanges.

1) Peer B registers through TCP with mediator M, sending its private endpoint B:b and a unique identifier, B-ID. The mediator also obtains the public endpoint B':b' from the apparent source of the connection. B and M keep this connection open. This establishes the mapping and allows control information to flow later.

2) At some later point in time, peer A requests a TCP connection to peer B, knowing B-ID. It sends M a message containing this request and its own private endpoint, A:a. M also obtains A':a' from the packet headers. A and M keep their connection open.

3) M introduces A and B to each other by sending each the endpoint information of the other.

4) Now both peers have all the information they need to set up the mapping. If their NATs did not filter, they could immediately set up a connection. However, we assume filters to be in place. Nevertheless, both endpoints try to set up direct connections to each other. The first one will be refused as its TCP SYN packet reaches the peer's NAT without a proper filter in place. However, this packet has established a filter rule in its own NAT, making sure that the peer's reverse connection setup attempt will succeed. This all happens in parallel, hence the name. When a connection fails or times out, each of the peers will simply retry. Care only needs to be taken that incoming and outgoing connections use the same local and remote ports.

To make sure the connection is actually the one that was expected, the endpoints should perform mutual authentication.

The parallel box of Figure 1 shows only one possible message exchange sequence. It could also happen that both connections traverse their NATs in outbound direction before they reach the opposite NATs. Then, a simultaneous connection setup happens, which the NAT boxes and especially the hosts' OSes need to handle.

One complication with the use of TCP hole punching is related to the fact that

sockets can either be passive (accepting incoming connections) or active (initiating an outgoing connection). Thus each peer needs to have four sockets open: One to the mediator, one for incoming connections, and one each for connecting to the public and private endpoint of the peer.[1] (If connections to other peers are open as well, this further adds to the socket count.) All four sockets need to be mapped to the same source port for the system to work. Operating systems do not allow this by default, however, this can be overridden on most of them by setting the `SO_REUSEADDR` socket option. On top of that, FreeBSD systems allow even more reuse of ports using `SO_REUSEADDR`. This problem, its implications and workarounds are discussed in detail in Section 2.

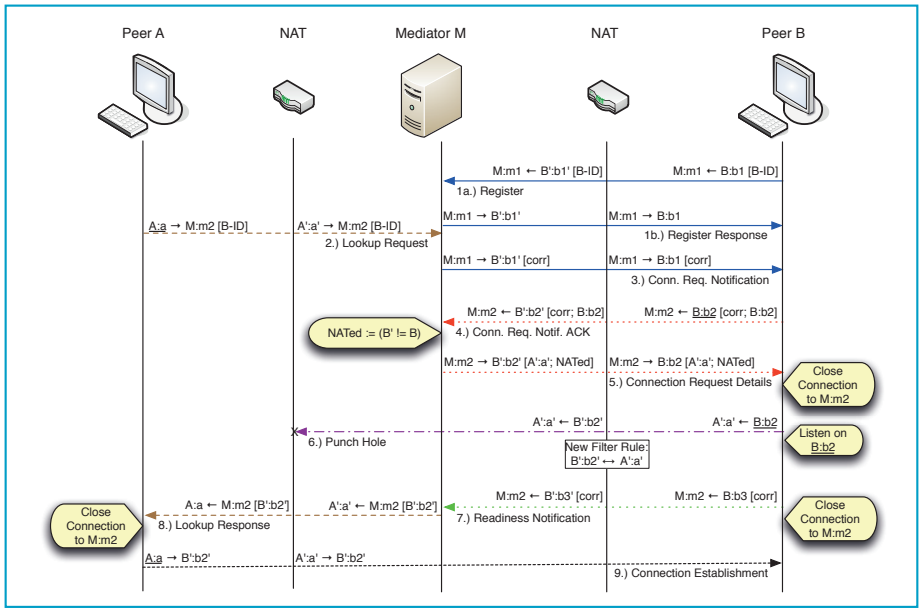


FIG. 2: Message sequence diagram of sequential TCP hole punching

1.2 Sequential TCP hole punching

In contrast to parallel TCP hole punching, the message order in sequential hole punching is deterministic as coordinated by the mediator. Eppinger et al. [3] have first published this approach under the name *NatTrav*. The original NatTrav also describes the use of multiple, redundant mediators. This aspect is not included here, as it is not essential to the technique, and is orthogonal to the topic of this paper.

Figure 2 shows the process as described by Eppinger et al. Arrows in the same color/pattern belong to the same connection. The steps are as follows:

1) Peer B registers with mediator M, passing a unique identifier, B-ID. The registration, which can be transmitted in UDP or TCP, is acknowledged.

2) When later peer A would like to connect to B, it sends a connection request message including B-ID to M. This message always uses TCP.

3) M sends a connection request notification to B along the connection established in step 1. This notification includes a correlator, corr, which will be used to associate steps 4 and 7 with this request.

4) Peer B acknowledges this message by opening a new TCP connection to M and including its private endpoint, B:b2.

5) M checks if B sits behind a NAT box by comparing B's private and public endpoints, and sets the NATed flag accordingly. M sends B the public endpoint of A and the NATted flag. This connection then has to be closed to enable B to reuse the same port subsequently.

6) If B sits behind a NAT box (as indicated by the NATed flag), it sends a TCP connection request to A which will fail due to one of the following reasons:

- A's NAT box silently discards the unsolicited SYN message which eventually results in B's timer to expire. We use a default timeout of 2 s as recommended in [3], which might not be long enough for slow or congested links, but significantly contributes to the overall performance of sequential hole punching. Silently discarding unsolicited SYN messages is by far the most widely implemented NAT behavior.
- A's NAT box actively rejects the unsolicited SYN message. In this case, B can proceed right away with step 7.
- A does not sit behind a NAT or A's NAT box lets the unsolicited SYN message through. Even in this case B's connection request will fail, because A still has an open connection with M and therefore does not listen for incoming connection requests yet.

Please note that it is imperative for B to use the same local port as in step 4. If B does not sit behind a NAT box, this step is omitted.

7) B now listens on port B:b2, from which the previous request originated. It then indicates its readiness for connection establishment to M.

8) M sends B's public endpoint to A using the connection from step 2.

9) A closes the connection to M, then connects to B using the same local port as the previous connection to M. B's NAT device already has an appropriate mapping and filter, so the connection will succeed.

The initial registration can use either TCP or UDP. While UDP has the advantage

1 Also trying to connect to the private endpoint ensures that peers behind the same NAT box will use the most efficient connection possible.

of requiring less kernel state at M (a single UDP socket is enough, TCP would require one per client), the clients would need to regularly and actively refresh NAT state using keep-alive messages over UDP.

The advantage of sequential TCP hole punching over its parallel cousin is that only one single socket ever needs to be bound to a given local port and the operating system does not need to correctly handle simultaneous TCP connection setups. The NAT boxes, however, still need to cope with an outgoing TCP SYN packet being answered by an incoming SYN packet instead of the normal SYN-ACK, as illustrated on B's NAT in Figure 2.

NatTrav [3] described here is far from the only sequential TCP hole punching proposal. Others, such as NUTSS [5] or NATBLASTER [2], require manipulating the packets' Time-to-Live (TTL) field, reading TCP sequence numbers or injecting handcrafted network packets by the application. These mechanisms frequently require superuser privileges and are definitely impossible to implement in a platform-independent way in Java. Therefore, such approaches will not be further discussed in this paper.

2 BINDING MULTIPLE SOCKETS TO THE SAME PORT

Parallel hole punching requires binding multiple sockets to the same local endpoint, which is not permitted by default. Socket options can help, however.

2.1 OS capabilities

Different operating systems differ in their TCP implementations and how they support corner cases of the protocol; moreover, different operating systems provide slightly different socket APIs.

One difference in the protocol implementation concerns simultaneous connection establishment, where two endpoints try to establish connections to each other at the same time. As tests of a set of relevant operating systems – Windows 7, MacOS X 10.6.5, Linux (Ubuntu 10.04 LTS), Solaris (Open-Solaris 2009.06), and FreeBSD 8.1 as a representative of the *BSD family – have shown, all tested OSes support simultaneous connection establishment, see Table 1.

Operating System				
Windows	Linux	FreeBSD	MacOS X	Solaris
✓	✓	✓	✓	✓

TABLE 1: Simultaneous connection establishment support.

Another difference concerns the ability to bind two or more sockets to the same port. This corner case is rarely well documented or tested. Our second experiment therefore tested whether a C program could bind two sockets to the same port, examining all possible combinations of server ('listen') and client socket creation. For each operating system, the most reuse-friendly socket option was chosen. For MacOS X and FreeBSD, this was their special `SO_REUSEPORT`, the other three used the common `SO_REUSEADDR`. All sockets were bound to the wildcard IP address `IPADDR_ANY`. Table 2 shows whether the particular combination worked ('✓'), did not work ('–'), or only worked when the client socket was already connected or at least in connection setup before the second socket was bound ('C'), i.e., the remote endpoint was already specified.

The results indicate that on Linux and Solaris, the server socket must be created after the client socket for the two to coexist. Thus, a portable implementation should never rely on the other order. This sequence can be achieved in parallel TCP hole punching, but requires some care.

2.2 Support within Java

For a portable Java implementation, OS support by itself is not sufficient; in addition, the OS's capabilities must also be accessible within Java.

Java provides the classes `java.net.Socket` and `java.net.ServerSocket`. To achieve platform independence, both support only a limited set of socket op-

Socket creation		Operating System				
First	Second	Windows	Linux	BSD	MacOS X	Solaris
Client	Client	✓	✓	✓	✓	C
Client	Server	✓	✓	✓	✓	C
Server	Client	✓	–	✓	✓	–
Server	Server	✓	–	✓	✓	–

TABLE 2: Operating system support for socket combinations. See text for explanation.

Socket creation		Operating System				
First	Second	Windows	Linux	BSD	MacOS X	Solaris
Client	Client	✓	✓	C	C	C
Client	Server	✓	✓	C	C	C
Server	Client	✓	–	–	–	–
Server	Server	✓	–	–	–	–

TABLE 3: Java support for socket combinations. See text for explanation.

tions. SO_REUSEADDR can be enabled by calling the setReuseAddress() method since Java 1.4, but there is no option to set SO_REUSEPORT, due to its limitation to FreeBSD derived platforms. This restriction leads to the results in Table 3, when testing the multiple-bind capabilities of the different operating systems in Java.

There is one notable difference to Table 2: MacOS X and FreeBSD implementations now share the Solaris limitation of a socket requiring at least a pending connection setup (i.e., defined remote endpoint) before another socket can be bound to the same port, because their SO_REUSEPORT option cannot be taken advantage of in Java. Even though the limitations for Java are more strict than for native applications as described in Section 2.1, the most stringent case is not further curtailed. Therefore, the consequences for portable, OS agnostic applications remain the same.

For Java implementations, care needs to be taken that server sockets for the Unix relatives cannot be reused due to the limitations outlined in Table 3. While this is not a problem under Windows, portable programs are required to close the old server socket and create a new one instead of reusing the existing socket, as a listening socket will prevent more client sockets from being opened. This is especially important because the first connection setup for parallel hole punching generally fails.

3 HOLE PUNCHING EXPERIMENTS

Four criteria are key for the evaluation of NAT traversal techniques, namely effectuality, performance, implementation complexity, and resource usage. Obviously, the technique should be effectual, i.e., work even under adverse circumstances, and the connection setup should be fast and efficient. Moreover, the implementation should be easy to understand (debug, test, and maintain), and avoid any resource wastes.

To verify the first two criteria, multiple tests were run in two different environments:

Virtual Internet. All nodes and boxes were simulated in our lab using virtual machines. The concrete setups are described in full detail in Section 3.1.

Real Internet. Peers A and B were behind real NAT boxes, behind DSL connections of different providers. The mediator has a public Internet address. The concrete setups for this environment are described in Section 3.2.

Three different scenarios were evaluated in both environments:

Concurrent connection requests. Peer A launches multiple concurrent requests for connection establishment. This scenario tests how well both hole punching implementations can deal with multiple concurrent incoming connection requests. This is particularly interesting for parallel hole punching that has to cope with many sockets bound to the same local port and with many concurrent threads that have to synchronize with each other. Sequential hole punching, on the other hand, is expected to be affected much less due to its sequential nature. The number of concurrent requests was set to 5.

Successive connection requests. Peer A initiates connection requests one after the other, with increasing waiting times. One of the goals is to verify the long-term state retention behavior of the NAT. For this experiment, the waiting times are 1, 5, 10, 20, 30, 60, 120, and 240 s.

Random connection requests. 5 threads on peer A initiate connections to peer B. Each thread uses a repeatable uniformly distributed pseudorandom waiting time between subsequent connections in the range of 0 to 60 s. This setup attempts to model real-world behavior.

A few bytes of data were exchanged after each successful connection establishment in order to verify the connection.

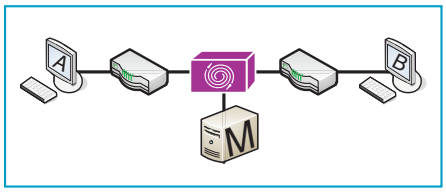


FIG. 3: 'Virtual Internet' Environment

3.1 'Virtual Internet' Environment

Figure 3 shows this environment. Each box is implemented as a virtual machine. The center box acts as a switch with delay and bandwidth limitation for a crude Internet simulation. More concretely, the delay between peers A and B is 30 ms, and the delay between any one peer and the mediator is 25 ms. The download bandwidth of the peers is limited to 2048 Kb/s, the upload bandwidth to 192 Kb/s, corresponding, e.g., to a typical 2 Mb/s DSL connection.

The NAT is implemented using standard iptables masquerading on Linux. This provides endpoint independent mapping, allowing only connection setup from the inside. Any other packet is silently discarded.

The combination of one of the five considered operating systems for peers A and B yields 25 different concrete setups. On each of these setups, we performed tests for each of the three scenarios (1) Concurrent connection requests, (2) Successive connection requests, and (3) Random connection requests. In all setups, the mediator was run on a virtual Linux machine.

3.2 'Real Internet' Environment

This environment matches Figure 3, with the center box being the real Internet with two DSL connections to the NAT boxes, and the mediator placed on campus. Because the main focus of this environment is on testing real NAT boxes (as opposed to the iptables simulation in the 'virtual Internet' environment), only two different setups, i.e., combinations of OSes for peers A and B, were used, namely peer

A always running Windows 7 and peer B running either MacOS X or Windows 7. The mediator was run on MacOS X. All tested NAT boxes employed endpoint independent mapping, as well as address and port dependent filtering.

4 EVALUATION

In this section we evaluate and compare the two techniques, parallel vs. sequential hole punching, with respect to the four criteria mentioned in section Section 3, namely performance, implementation complexity, resource usage, and effectuality.

4.1 Performance

Figure 4 contains the plots for the 'virtual Internet' environment. Please note that only results for parallel hole punching are shown because sequential hole punching does not work in this environment, as will be discussed in section Section 4.4.

As can be seen, for peer B running MacOS X the connection times are around 1 s in most cases. These times stem from the fact that simultaneous TCP connection establishment on MacOS X takes about 1 s, as we could observe in isolated tests. Whenever neither peer A nor peer B run MacOS X, then the mean connection setup times vary between 250 ms and 690 ms.

Figure 5 shows the results for the 'real Internet' environment. Please note that they were not taken in a controlled environment, so individual packet delay or losses do affect comparability. Sequential hole punching clearly shows an about 2 s higher setup time, which is due to the 2 s timeout specified in [3]. This timeout should cover most situations without packet loss today, although slow or lossy connections might become a problem. Even in developed countries, these 2 s may not be enough, especially for mobile Internet access, where the mobile provider operates the NAT for

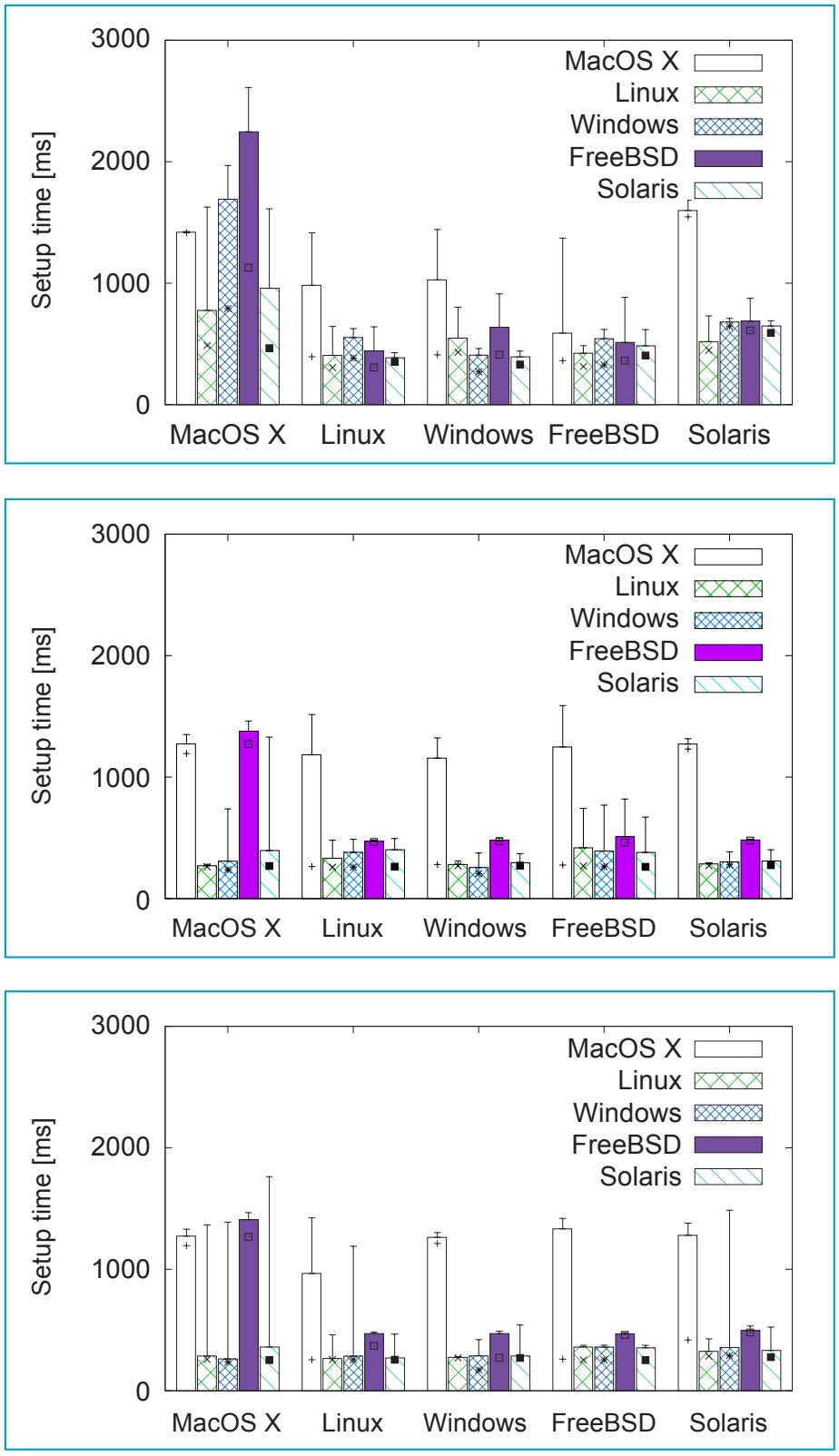


FIG. 4: Mean connection setup times for parallel hole punching in the 'virtual Internet' environment. The upper plot shows the results for the concurrent connection requests scenario, the middle plot for successive connection requests, and the lower plot for random connection requests. Error bars indicate minimum/maximum times. The labels on the X-axis denote peer A's operating system, the color code of the bars indicate peer B's operating system.

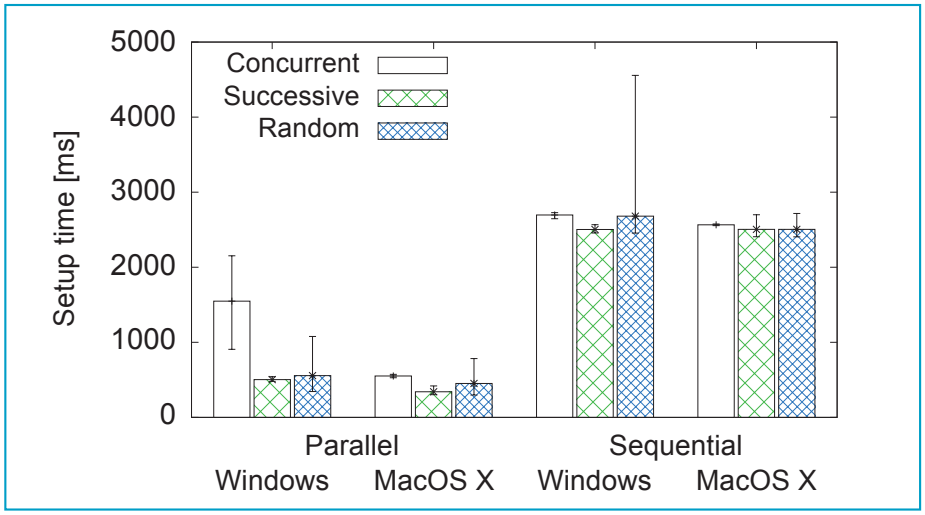


FIG. 5: Mean connection setup times for parallel hole punching in the 'virtual Internet' environment. The upper plot shows the results for the concurrent connection requests scenario, the middle plot for successive connection requests, and the lower plot for random connection requests. Error bars indicate minimum/maximum times. The labels on the X-axis denote peer A's operating system, the color code of the bars indicate peer B's operating system.

its customers and the wireless link may be unpredictable due to signal quality or high usage.

Reducing the timeout would thus make the protocol less robust. A significant reduction of the timeout would require the introduction of retransmits to achieve a reasonable connection chance. This would in effect make the protocol very similar to parallel hole punching, both in performance and complexity.

4.2 Implementation Complexity

Sequential hole punching as described in Figure 2 is rather straightforward to implement, as there are only few parallel operations needed: None on peer A, and for peer B only to regularly send out keepalive messages, which can be integrated into the application main loop. However, Figure 2 assumes that there is no packet loss in step 6 and the timeout was chosen generous enough. When these assumptions fail (e.g., for wireless/mobile clients as described above), the protocol will fail and recovery mechanisms need to be designed in, which add further setup delays and complexity to the main application having to deal with failures.

A parallel hole punching implementation requires more thought, as it needs to deal with simultaneous use of sockets as described in Section 2 above. It also requires substantial thread operations and synchronization, which adds to the higher complexity of a parallel hole punching implementation.

4.3 Resource Usage

Parallel hole punching requires more resources on the peers than sequential hole punching, as multiple threads need to be running, more sockets are created and destroyed, and more than one connection is open at the same time. Sequential hole punching, on the other hand, requires more messages and more actual connection setups and teardowns to the mediator.

Kernel resources for the mediator are higher for parallel hole punching, as the mediator has to keep an open TCP connection with each registered peer. For sequential hole punching, the mediator can use a single UDP port to register all peers. On the other hand, a mediator for sequential hole punching needs to store some session information (correlator, corr, see Fig. 2), whereas a mediator for parallel hole punching can be completely stateless and is thus better scalable.

In summary, there is little difference between the two approaches from the perspective of the peers. This is especially true because today even mobile end devices, such as smartphones, have enough storage and CPU resources to support slightly more demanding applications. As far as the mediator is concerned, whether fewer open connections or stateless operation is preferable cannot be decided independent of the concrete environment and usage.

4.4 Effectuality

During our experiments, it became clear that actual NAT is harder to deal with than pure theory and message sequence diagrams would implicate. Some of these effects are discussed below, structured into NAT box problems: mapping and filtering behavior, mapping loss, and SYN-ACK checks) and endsystem behavior (direct private connection, anti-virus tools, and OS/Java limitations (discussed in Section 2). A summary can be found in Table 4.

4.4.1 Mapping

In Section 1, endpoint independent mapping was listed as a precondition for

Effectuality		Parallel	Sequential
NAT	Mapping	+	–
	Mapping loss	+	–
	SYN-ACK checks	+	–
Host	Direct connection	+	(–)
	Anti-virus	+	–
	OS support	(–)	+

TABLE 4: Hole punching effectuality components.

hole punching. However, even if one side, say peer A, employs address dependent mapping, hole punching can succeed under the following conditions:

- peer A uses the same external IP address for all mappings;
- peer B uses endpoint independent mapping in combination with address dependent filtering or a less restrictive filter policy.

In this situation, sequential hole punching works only when peer A's NAT is address dependent, but not when B's NAT is. Parallel hole punching, on the other hand, will succeed in both cases due to its symmetric behavior.

4.4.2 Mapping drop

A NAT box could immediately destroy a connection context when the connection is reset or closed. This is disastrous for the sequential approach, because if the remote NAT on step 6 of Figure 2 returns a TCP RST message, then the reverse connection in step 9 will fail.

For parallel hole punching, the SYN packets are likely to cross outside the NATs eventually, and thus create a successful simultaneous connection setup. Also, as long as not both NATs reject packets with RST and drop mappings, parallel hole punching will still succeed.

4.4.3 Linux iptables SYN-ACK check

Linux `iptables` is very strict at checking the validity of packets: In a correct simultaneous setup, the replayed SYN packet must contain the same sequence number as the original SYN. Most if not all NAT boxes, however, do not check this condition, whereas `iptables` does. `Iptables` therefore uses some form of connection dependent filtering. This behavior prevents all sequential hole punching attempts in the 'virtual Internet' scenario from succeeding. There was no problem, however, for parallel hole punching, as both a server and client socket are active, therefore resulting in a simultaneous connection setup, supported `iptables` behavior.

We did not observe this kind of filtering with our tested NAT boxes. However, as `iptables` is frequently used in semi-professional and SME contexts, a hole punching technique should be able to deal with such behavior.

4.4.4 Direct private connection

Parallel hole punching natively supports direct connections to private addresses. This is done in an attempt to connect more efficiently to machines behind the same NAT and can be done with minimal additional overhead. While the same behavior could be implemented for sequential hole punching, the sequential nature would require waiting for an additional timeout (likely) or error (unlikely).

Sequential hole punching to a peer behind the same NAT succeeds only if the NAT supports *hair pinning,* and otherwise fails completely. With hair pinning, however, the connection will unnecessarily traverse the NAT, leading to additional resource utilization and poorer performance.

4.4.5 AVG anti-virus software

During our experiments, sequential hole punching failed when peer B was running Windows with anti-virus software by AVG (http://www.avg.de). Close examination with purpose-built test applications and Wireshark (http://www.wireshark.org) revealed the following behavior introduced by AVG.

When `connect()` is called on a socket and then aborted after the 2 s timeout, the application behavior is as expected. However, Wireshark reveals that retransmits of that initial SYN packet continue after 3 and 9 s, despite the `connect()` having been aborted and the socket being closed in the application at that time. However, the OS kernel still believes the socket to be active. This discrepancy leads to the wrong behavior, when the SYN packet from the final connection establishment (step 9 in Figure 2) finally arrives: It connects with the client socket, resulting in a simultaneous connection setup. The application, however, has already abandoned that socket, so no data transfer will be possible.

This problem could be reproduced on clean installations of Windows 7 (32bit and 64bit variants) with the current version of AVG Anti-Virus Free Edition 2011 (version 10.0.1191). As AVG claims [7] their anti-virus products to be installed on more than 110 million machines, this is a severe problem for sequential hole punching. Parallel hole punching, once more, is not affected by this problem.

4.5 Summary of the evaluation

Table 5 summarizes the comparison between parallel and sequential hole punching.

As we have seen in Section 4.4, parallel hole punching is by far the more effectual technique, as it can deal with a number of non-standard and even adverse conditions. Sequential hole punching, on the other hand, is more vulnerable under the same circumstances. In terms of performance, parallel hole punching is also superior to sequential hole punching. This is mainly due to the timeout that is inherent to sequential hole punching. The only criterion in favor of sequential hole punch-

Metric	Parallel	Sequential
Effectuality	+++	+
Performance	++	+
Implementation	– –	–
Resources	–	–

TABLE 5: Hole punching metric summary.

ing is implementation complexity. As we have discussed in Section 4.2, parallel hole punching requires significantly more complex code because of its high degree of multithreading. With respect to resource consumption, we do not see a clear winner on either side.

5 CONCLUSION AND FUTURE WORK

In this paper, we have presented and discussed Java implementations of parallel and sequential hole punching. As our evaluation has shown, parallel hole punching is, in most respects, superior to sequential hole punching. Our open source parallel hole punching Java implementation is available at http://ice.in.htwg-konstanz.de.

Plans on evolving this software are fourfold. First, to evaluate the integration of port prediction for the rare case of NAT boxes with address dependent mapping. Second, we are working on an integrated approach for NAT traversal. The goal is to minimize setup times by allowing the first few data packets to go through the mediator and then be seamlessly mapped to the direct connection. This is especially useful for Java RMI between NATted hosts, where we plan to integrate this feature as our third plan. Fourth, we are working on ports onto mobile platforms. First results on Android are available and back our hypothesis that modern mobile devices are powerful enough to support the multithreading and advanced socket options requirements stemming from parallel hole punching.

ACKNOWLEDGEMENTS

This work was done within the scope of the FHprofUnt project 'Transparente Integration von NAT Traversierungstechniken in Java' funded by BMBF and Seitenbau GmbH, Konstanz.

REFERENCES

[1]: C. Aoun and E. Davies: „Reasons to Move the Network Address Translator – Protocol Translator (NAT-PT) to Historic Status". RFC 4966 (Informational), Internet Engineering Task Force, Jul 2007. Online: http://tools.ietf.org/html/rfc4966

[2]: A. Biggadike, D. Ferullo, G. Wilson, and A. Perrig: „NATBLASTER - Establishing TCP Connections between Hosts behind NATs". In: ACM SIGCOMM Asia Workshop, 2005.

[3]: J.L. Eppinger: „TCP Connections for P2P Apps – A Software Approach to Solving the NAT Problem". Carnegie Mellon University, Tech. Rep., Jan 2005.

[4]: B. Ford, P. Srisuresh, and D. Kegel: „Peer-to-peer Communication across Network Address Translators". In: USENIX Annual Technical Conference, 2005, pp. 179-192.

[5]: S. Guha: „NUTSS – a SIP-based Approach to UDP and TCP Network Connectivity". In: ACM SIGCOMM 2004 Workshops, 2004, pp. 43–48.

[6]: T. Narten, R. Draves, and S. Krishnan: „Privacy Extensions for Stateless Address Autoconfiguration in IPv6". RFC 4941, Internet Engineering Task Force, Sep 2007. Online: http://www.ietf.org/rfc/rfc4941

[7]: AVG Technologies: „AVG Technologies – Unternehmensprofil". Online: http://free.avg.com/de-de/company-profile

[8]: G. Tsirtsis and P. Srisuresh: „Network Address Translation – Protocol Translation (NAT-PT)". RFC 2766 (Historic), Internet Engineering Task Force, Feb 2000. Obsoleted by RFC 4966, updated by RFC 3152. Online: http://tools.ietf.org/html/rfc2766