Presented at: IEEE LCN 2011, The 36th IEEE Conference on Local Computer Networks; 4-7 October 2011, Bonn, Germany

NAT Hole Punching Revisited

Daniel Maier Oliver Haase Jürgen Wäsch
Konstanz University of Applied Sciences
Konstanz, Germany
[dmaier|haase|waesch]@htwg-konstanz.de

Marcel Waldvogel
University of Konstanz
Konstanz, Germany
marcel.waldvogel@uni-konstanz.de

***Abstract*—Setting up connections to hosts behind Network Address Translation (NAT) equipment has last been the subject of research debates half a decade ago when NAT technology was still immature. This paper fills this gap and provides a solid comparison of two essential TCP hole punching approaches: sequential and parallel TCP hole punching. The comparison features current conditions and thoroughly compares setup delay, implementation complexity, resource usage, and effectuality of the two approaches. The result is a list of recommendations and a portable, effectual, and open-source Java implementation.**

I. Introduction

Currently the use of Network Address Translation (NAT) is the dominant solution to lessen the exhaustion of IPv4 addresses. This works great as long as the machines behind the NAT box only initiate outgoing connections and do not have to accept incoming requests.

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. 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. The explanations of the two approaches can be found in [2] and [3]. We compare them according to various criteria in different scenarios.

II. 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.

A. 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. As tests of a set of relevant operating systems—Windows 7, MacOS X 10.6.5, Linux (Ubuntu 10.04 LTS), Solaris (OpenSolaris 2009.06), and FreeBSD 8.1—have shown, all tested OSes support simultaneous connection establishment.

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`. Table I 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.

TABLE I: OS support for socket combinations.

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

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.

B. 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.

To achieve platform independence, Java supports only the use of `SO_REUSEADDR`. This restriction leads to the results in table II, when testing the multiple-bind capabilities of the different operating systems in Java.

TABLE II: Java support for socket combinations.

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	✓	—	—	—	—

There is one notable difference to table I: MacOS X and FreeBSD implementations now share the Solaris limitation of a socket requiring at least a pending connection setup 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 II-A, 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 II. 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.

III. 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 III-A.

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 III-B.

Three different scenarios were evaluated in both environments:

Concurrent connection requests. Peer A launches multiple concurrent requests for connection establishment. 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. 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. 'Virtual Internet' Environment

Each participant is realized as a virtual machine. Even the internet is simulated by a virtual switch with delay and bandwidth limitation. More concretely, the delay between peers A and B is 30ms, and the delay between any one peer and the mediator is 25ms. The download bandwidth of the peers is limited to 2048 Kb/s, the upload bandwidth to 192 Kb/s.

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.

B. 'Real Internet' Environment

In this environment the communication takes place across 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, 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 Mac OS. All tested NAT boxes employed endpoint independent mapping, as well as address and port dependent filtering.

IV. Evaluation

In this section we evaluate and compare the two hole punching techniques with respect to the four criteria mentioned in section section III.

A. Performance

Figure 1 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 IV-D.

As can be seen, for peer B running MacOS X the connection times are around 1sec in most cases. These times stem from the fact that simultaneous TCP connection establishment on MacOS X takes about 1sec, 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 250ms and 690ms.

Figure 2 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 2sec higher setup time, which is due to the 2sec timeout specified in [2]. This timeout should cover most situations without packet loss today, although slow or lossy connections might become a problem. Even in developed countries, these 2sec may not be enough, especially for mobile Internet access.

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.

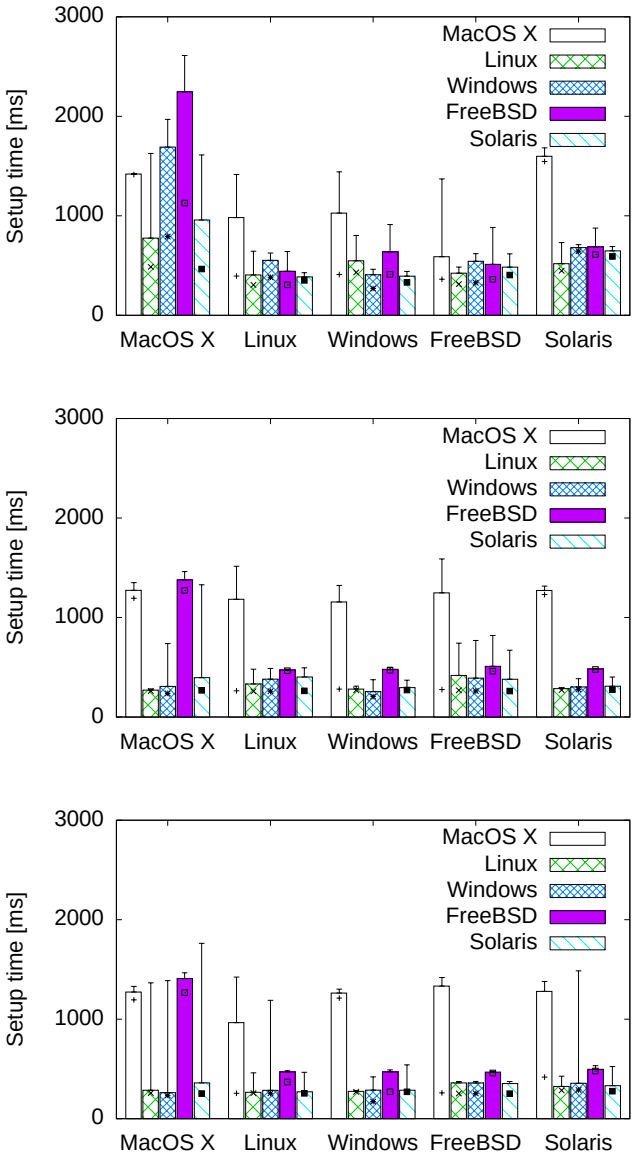

Fig. 1: Mean connection setup times for parallel hole punching in the 'virtual Internet' environment. The upper plot shows the results for the *concurrent connection requests* scenario, the middle plot for *successive connection requests*, and the lower plot for *random connection requests*. Error bars 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.

B. Implementation Complexity

Sequential hole punching as described in [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 keep-alive messages, which can be integrated into the application main loop. However, [2] assumes that there is no packet loss in the hole creation step and the timeout was chosen generous enough. When these assumptions fail, 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.

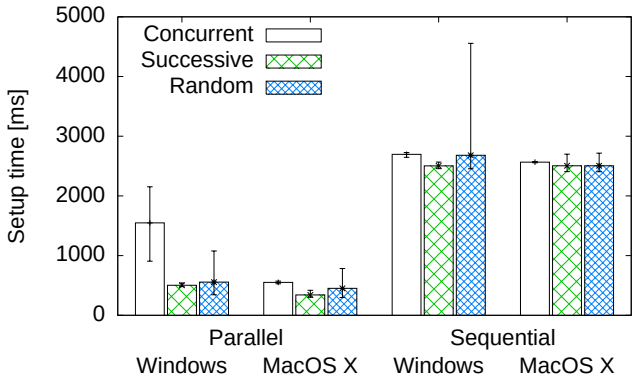

Fig. 2: Mean connection setup times in the 'real Internet' environment. Error bars indicate minimum/maximum times. The labels on the X-axis denote peer B's operating system, the color code of the bars indicate the scenario under test.

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

C. 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`), 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.

D. 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 and end-system behavior. A summary can be found in table III.

TABLE III: Hole punching effectuality components

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

1) Mapping: Usually endpoint independent mapping is mentioned as a precondition for 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.

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 returns a TCP RST message to the hole creation step, then the reverse connection in the last step 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.

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) 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, however, succeeds only if the NAT supports *hair pinning*.

5) AVG anti-virus software: In our experiments, sequential hole punching failed when peer B was running Windows with anti-virus software by AVG (version 10.0.1191). Close examination revealed the following behavior introduced by AVG: When `connect()` is called on a socket and then aborted after the 2sec timeout, the application behavior is as expected. However, a packet analyzer reveals that retransmits of that initial SYN packet continue after 3 and 9sec, 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 (last step) 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.

As AVG claims [1] 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.

E. Summary

Table IV summarizes the comparison between parallel and sequential hole punching. As we have seen in section IV-D, 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 punching is implementation complexity. With respect to resource consumption, we do not see a clear winner on either side. Our open source parallel hole punching implementation is available at `http://ice.in.htwg-konstanz.de/`. For further reading, [4] contains a more detailed version of this article.

TABLE IV: Hole punching metric summary.

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

References

[1] AVG Technologies, "AVG Technologies - Unternehmensprofil," http://free.avg.com/de-de/company-profile.

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

[3] B. Ford, P. Srisuresh, and D. Kegel, "Peer-to-peer communication across network address translators," in *In USENIX Annual Technical Conference*, 2005, pp. 179–192.

[4] D. Maier, O. Haase, J. Wäsch, and M. Waldvogel, "NAT Hole Punching Revisited," University of Konstanz, Tech. Rep. KN-2011-DiSy-02, 2011.