

# Peer NAT Proxies for Peer-to-Peer Games

Daryl Seah, Wai Kay Leong, Qingwei Yang, Ben Leong, and Ali Razeen  
Department of Computer Science  
National University of Singapore

**Abstract**—Network Address Translators (NAT) are ubiquitous on the Internet and any peer-to-peer (p2p) game will almost certainly need to perform NAT traversal through such devices. Our experiments suggest that while NAT hole punching techniques are relatively mature, they succeed only about 90% of the time and thus p2p games will inevitably need to employ NAT proxies to establish the remaining connections. We demonstrate with an implementation and a measurement study that using peers as NAT proxies is feasible for both UDP and TCP connections. We found that it is relatively easy to find peers capable of acting as proxies and that the performance achieved is comparable to that of server-based NAT proxies.

## I. INTRODUCTION

Since peer-to-peer (p2p) applications were first introduced about a decade ago, numerous applications and protocols such as Gnutella, BitTorrent and Skype have been developed. Recently, Gas Powered Games launched Demigod, a multi-player game that exploits p2p networking to improve game performance and, conceivably, to reduce server hosting costs. Upon its release, the game was plagued by a host of p2p related networking problems [1, 2]. Since extensive testing was almost certainly done prior to the launch, its teething problems highlight the unexpected difficulties of deploying p2p games over the Internet.

Many of the issues encountered by the Demigod developers can be attributed to the problems dealing with Network Address Translators (NAT) [3], which are known to be a major complication in the implementation of p2p protocols and systems [4, 5, 6]. Since NAT boxes prevent direct connections between peers, signaling servers on the public Internet are required to assist in punching a hole through them. If NAT hole punching fails, proxy servers would be required to bridge connections between the peers. Studies in 2005 found that NAT hole punching fails approximately 10 to 15% of the time [5, 7]. In our recent measurement study, we found that the situation has not improved significantly over the past few years. As such, game companies will likely always have to provision proxy servers to support p2p games.

In this work, we study the feasibility and effectiveness of using ordinary peers, even if they are behind a NAT, as NAT proxies. We evaluated our Java p2p network library implementing server and peer-based NAT proxies along with RakNet [8], the commercial game library used by Demigod [2], and found that peer proxies can achieve latencies and throughput comparable with server proxies. We also show

that there is a high likelihood of finding a proxy within a network and that a simple selection scheme of broadcasting requests to 5 peers and selecting the first respondent as the proxy is sufficient to provide good performance. Moving forward, peer proxies are thus an attractive solution for reducing the cost of provisioning dedicated proxy servers for NAT traversal.

The remainder of this paper is organized as follows: in Section II, we provide a review of existing and related work. Section III describes the architecture and implementation of our p2p network library. Finally, we describe our evaluation results in Section IV and conclude in Section V.

## II. RELATED WORK

NAT traversal for UDP was pioneered by Dan Kegel and his techniques have since been standardized to become the STUN (Simple Traversal of UDP Through NATs) [9] protocol. Subsequently, Guha et al. developed STUNT [10], which extended STUN for TCP NAT traversal. Since NAT traversal with STUNT is not always successful, the Traversal Using Relay NAT (TURN) protocol [11] was developed to use a relay node as a last resort if NAT hole punching fails. In 2005, Ford et al. obtained reports from volunteers with various NAT boxes, and found that 82% supported UDP hole punching while 64% supported TCP hole punching [5]. At around the same time, Guha et al. evaluated the effectiveness of STUNT both with a lab testbed and in the wild, and found that the success rate for TCP NAT traversal is about 88% [7]. They also measured the time taken to establish a TCP connection, but only for a single attempt without retrying in the case that port prediction failed. In this work, we have repeated some of these measurements and also measured the time taken to perform NAT traversal for both UDP and TCP.

Skype is a popular VoIP application that uses peers to relay VoIP traffic. Measurement studies by Guha et al. found Skype supernodes (i.e. potential relays) to be relatively stable when compared to nodes in other p2p networks and that the bandwidth consumed by an active relay is fairly low [12]. Another study classified relay nodes by domain name suffixes and found 22.4% of them to be ending in .edu, suggesting that the Skype network might be sustained by high-bandwidth non-NAT university networks [13]. In addition, Ren et al. proposed ASAP [14], an AS-aware peer relay protocol for VoIP, suggesting that one-hop peer relaying can provide higher quality VoIP transmission when compared to direct IP routing. These works suggest that the benefits of relaying are not limited to circumventing NAT and its effectiveness depends

on the nature of the application and the demographics of the users in the network.

While there have also been proposals for finding and selecting appropriate peer relay nodes, previous proposals have only been simulated and have not been validated in “live” deployments. Dowling et al. suggested the use of the gradient topology to help in locating good quality relay peers in a generic p2p network [15], while Nguyen et al. proposed a relay selection algorithm to improve load balancing [16]. To the best of our knowledge, there has not been a comprehensive study on the use of peers as proxies for NAT traversal.

There are many libraries for developing p2p networked applications available to developers, and most of them have some form of NAT traversal support. Among them, RakNet [8], the game library used by Demigod, is by far the most popular library among game developers for traditional server-client games. P2P support has been recently added as plugins that support NAT traversal via hole punching and proxying. In view of its commercial popularity, we evaluate the performance of RakNet in addition to our network library.

### III. SYSTEM ARCHITECTURE

Our p2p network library allows applications to create communication endpoints called *nodes*, each of which is assigned a globally unique identifier by a central server called the *global tracker*. Two communication primitives are supported. The first is a UDP-based *application-layer framing* [17] interface where peers can send messages in the form of *application-data units* (ADUs). Different packet reliability and priority schemes can be set for each ADU sent. The second is a TCP-based stream primitive, useful for bulk transfers of data for certain applications (e.g. file sharing).

We implemented the library in two distinct layers: ALF and PAL. The ALF layer supports direct connection establishment using NAT hole punching and is so named because it also implements the basic abstractions for *application-layer framing*. The P2P Abstraction Layer (PAL) augments ALF by creating proxied connections transparently when direct connections fail. For brevity, we will refer to our network library as *PAL* in this paper.

#### A. Hole Punching

As a best-effort solution for establishing direct connections between peers, we implemented standard hole punching techniques with port prediction [5, 10], mostly following the STUNT [7] protocol. The global tracker serves as the signaling server for coordinating a hole punching attempt. In fact, two distinct machines are necessary. We call the second server the *buddy server*, since it pairs with the global tracker to assist peers during hole punching attempts.

Hole punching for UDP and TCP generally works as follows: connection request packets are sent to the known addresses of the destination peer, starting with the most private address first. Simultaneously, the destination peer is signaled via the global tracker to attempt the same procedure in reverse. If connection requests are received at either peer, a handshake

is performed and the connection is established. Otherwise, the global tracker and buddy server are queried by both peers to obtain the external address and ports assigned by the NATs for each server. With this information, the peers predict the next port that their NAT will assign and exchange their predictions via the signaling channel. If the NAT boxes assign the ports as predicted, sending connection request packets to the predicted address and ports of both peers will create “holes” in the NATs permit packet flow between them. Since port prediction can fail for a variety of reasons [10], we repeat this process a number of times if the connection does not get through.

The key difference between our implementation and the STUNT protocol is that we attempt both direct and reverse connections on the known addresses and ports of the peer first before attempting port-predicted connections. Port-predicted connection attempts can be time-consuming, especially if prediction accuracy is low. Our approach ensures that a connection can be established quickly if the peers have no connectivity issues. Another difference is that we use UPnP, if it is supported by the NAT, to explicitly open ports and increase the effectiveness of hole punching.

#### B. Proxying

When a peer connection is initiated, PAL will transparently create a proxied connection if NAT hole punching fails. One of the available peers will be chosen as the proxy, with a server-based proxy being used as a last resort. In our implementation, the global tracker functions as the server-based proxy. Since server-based proxying is simple and well understood, we will only discuss our implementation of peer proxying in detail.

The key challenge in supporting peer proxying is that peers can leave the network at any time. In other words, it is possible for data to be sent successfully to a proxy and yet be lost because the proxy departs before the data can be forwarded to the destination.

It is fairly straightforward to deal with this issue for UDP proxying. Headers containing the identifiers of the source and destination nodes and a unique message identifier are added to each packet. These allow forwarding to be performed on a per-packet basis, which makes it easy for us to change the UDP proxy “on-the-fly” if it fails. The unique message identifier is used to perform duplicate elimination for retransmissions when the reliable UDP delivery mode of our library is used.

TCP proxying, however, is more complicated since it guarantees the delivery and ordering of a continuous data stream to the application. Our solution is to split the data stream into 1,400 byte segments, each delimited by a 7-byte header. The headers are used to monitor and acknowledge the end-to-end delivery of the segments as they are relayed by the proxy and reassembled at the destination.

Our scheme for selecting a peer proxy is to pick the first peer capable of bridging the connection out of a random set of 5 candidate peers obtained from the global tracker. This is repeated until a proxy is found or there are no more candidates available. In our evaluation, we show that even though this

scheme is simple, it can find a proxy with reasonably good performance with minimal delay.

#### IV. EVALUATION

To obtain data for evaluating the performance of PAL, we deployed two Java games on Facebook: *Tankie MAX* [18], a real-time p2p tank game, and *Karkinos* [19], a single-player tower-defense game. The NAT connectivity statistics were collected using *Tankie MAX* as the game yielded more NAT hole punching attempts. It is also more popular, with over 750 users (about 65% from Singapore) playing the game during the period of 15 days in Sep 2009 when we collected the data. To avoid the interference of game traffic on our proxy performance measurements, we developed a separate standalone game, *Karkinos*, with an embedded PAL test client to conduct the experiments. 180 users played *Karkinos* over 2 separate sessions in Oct and Nov 2009, with a maximum of about 20 users online at the same time.

We wrote another test client in C++ using the RakNet library for comparison against our implementation. RakNet was chosen as it is perhaps the most popular open-sourced network library in use by commercial games. While it was initially designed for server-client architectures, p2p features such as NAT hole punching, proxy support and connection graph overlays have been recently added as plugins. Where possible, we tried to conduct experiments similar to those for PAL to obtain a fair comparison. However, RakNet does not natively support peer proxies and TCP, so our evaluation is restricted to just UDP and server-based proxies. We managed to get a total of 94 users to run the RakNet test client over 2 separate sessions in Jul and Oct 2009. We have much fewer data points for RakNet compared to PAL because it seems that volunteers are more wary and less inclined to run a standalone Windows executable than a Java-based Facebook game.

All our experiments were carried out over the Internet by volunteers who either played our Facebook games or ran our Windows-based RakNet test client. Most of the users who ran our PAL and RakNet test clients were from Singapore as well. We believe the localized data is still useful as it is not unreasonable to expect players to tend to join games with low latency, often within their own country. However, to ensure that the results are not skewed by users in our local university network, we filtered out such nodes from the data collected.

##### A. Connection Success Rates

The data collected from *Tankie MAX* for NAT hole punching is summarized in Table I. Our results are comparable to the findings by Guha and Francis in 2005 [7]. In addition, we also classified the attempted NAT connections by determining whether the endpoints were behind a NAT and whether they supported UPnP. It is not surprising that if one or more endpoints support UPnP, hole punching is generally more successful. However, we notice an anomaly for UDP, where if both NAT boxes support UPnP, hole punching performs slightly worse than if just one of them supports UPnP. At this point, we do not have a good explanation for this phenomenon.

TABLE I  
NAT HOLE PUNCHING SUCCESS RATES BY PAIRS OF ENDPOINTS.

Type of Endpoints	UDP		TCP	
	Success	Total	Success	Total
Both NAT w/o UPnP	86.6%	5573	87.7%	1328
Both NAT w/ one UPnP	96.5%	2451	92.3%	426
Both NAT w/ both UPnP	91.2%	431	100%	144
One NAT w/o UPnP	95.6%	856	100%	186
One NAT w/ UPnP	96.6%	207	100%	37
No NAT	100%	50	100%	16
Overall	90.4%	9568	90.8%	2137

TABLE II  
NAT HOLE PUNCHING SUCCESS RATES FOR RAKNET BY PAIRS OF ENDPOINTS.

Type of Endpoints	UDP Pairs	
	Success	Total
Both NAT	91%	391
One NAT	100%	32
No NAT	N/A	0
Overall	91%	313

RakNet’s native connection method assumes direct connectivity and does not perform NAT traversal. As such, UDP NAT hole punching is supported through a plugin, which has to be explicitly invoked by the application. We could have developed our NAT hole punching plugin or adjust the plugin’s settings, but we decided we wanted to have an “out of the box” comparison with RakNet. Unlike our implementation, there is no fixed protocol or NAT connection sequence for RakNet. To obtain a fair comparison, we introduced the following sequence in the RakNet client when establishing a connection: when peer *A* initiates a connection to peer *B* and fails, the server will notify peer *B* to attempt a reverse connection to peer *A*; if this fails as well, peer *A* will initiate NAT hole punching using the hole punching plugin at the default settings; if NAT hole punching fails, then the pair is deemed to be unconnectable.

RakNet does not have its own UPnP implementation, and we were not able to get consistent results with a recommended third-party library and hence did not employ UPnP for the RakNet tests. Nevertheless, as shown in Table II, the results are similar to that obtained for our Java-based library.

##### B. Connection Delays

With the data collected from *Karkinos* and the RakNet client, we plot the cumulative distribution of the time taken to successfully establish various types of connections in Fig. 1. UDP server-proxied connection times are omitted, because they are almost instantaneous. The graph shows that NAT traversal can add significant delay to the connection time, with 10% of direct PAL connections taking longer than 15 and 30 *s* for UDP and TCP respectively.

The RakNet client appears to establish UDP connections faster than PAL. We found that this is due to a combination of timeout settings in RakNet, which limits the time for any connection attempt to a maximum of about 10 *s*. Since our implementation shows that it may take longer than 10 *s* to

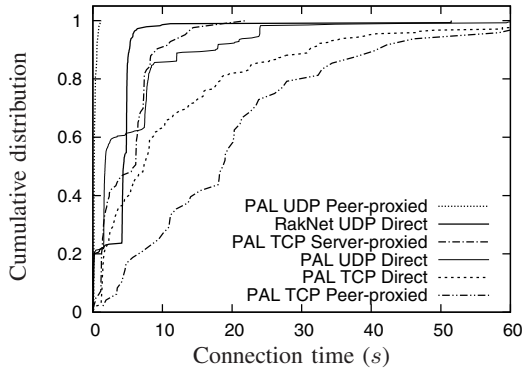


Fig. 1. Cumulative distribution of the time taken to establish connections.

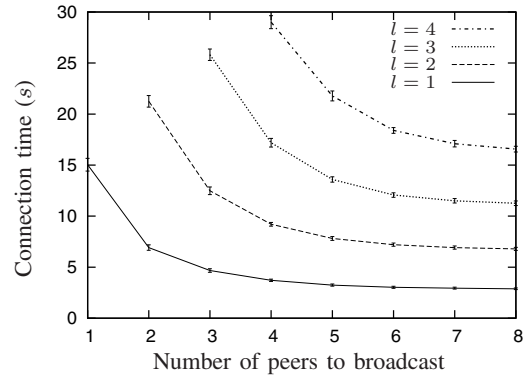
successfully perform a UDP hole punch, it may be the case that RakNet’s default settings are too conservative or our timeout values can be further optimized. We currently lack sufficient NAT traversal data for RakNet to provide a conclusive answer. It is important for game developers to have a good appreciation for these connection times in order to maximize their NAT traversal success rates.

It takes less time to create a connection through a server proxy for UDP and TCP than a direct connection because no hole punching is required to reach the server. While it may seem surprising that the establishment of a UDP peer proxy is also faster than a direct connection, this is due to the fact that requests are sent to several peers in parallel, which makes it more likely that the peers without NAT issues will fulfill the request earlier and reduce the overall connection time. Also, some proxies might have established the required UDP connections beforehand and so would not incur the time taken to perform NAT hole punching. On the other hand, setting up a TCP peer proxy can take exceptionally long, with about 20% of such connections taking longer than 30 s. This is expected though, as the process is more elaborate than setting up a UDP peer proxy. In addition, unlike UDP, NAT hole punching has to be performed for every new TCP connection request.

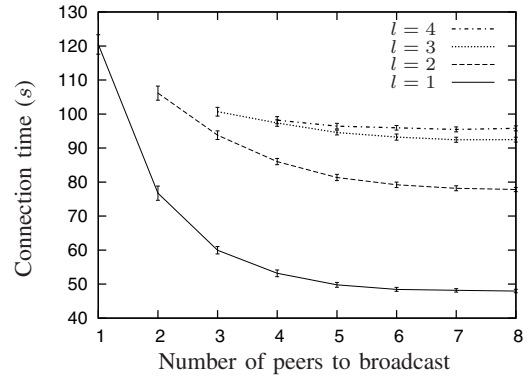
### C. Finding Peer Proxies

Since we are studying the feasibility of using peer proxies in a p2p setting, the two natural questions are: (1) *how do we find suitable peer proxies* and (2) *are peer proxies readily available in a typical network?* A trivial solution to the first question would be to send a message to every peer to check if it can be a proxy. While this might work in a small network, broadcasting to every peer in a large network would result in significant overhead and congestion.

To choose a proxy, our approach is to broadcast a proxy request to  $k$  randomly selected peers and choose the best peer after we have heard from at least  $l$  of them. If none of these peers are able to be a proxy, we repeat this process with another  $k$  peers until we exhaust all available peers and need to rely on a proxy server. To derive the optimal values for  $k$  and  $l$ , we collected data on the connection times, throughputs and latencies achieved between all pairs of nodes that were online at the same time and conducted simulations on the data set



(a) UDP



(b) TCP

Fig. 2. Effect on connection time with varying number of peers to broadcast per round ( $k$ ) and number of responses to wait for ( $l$ ). Error bars indicate the standard deviation.

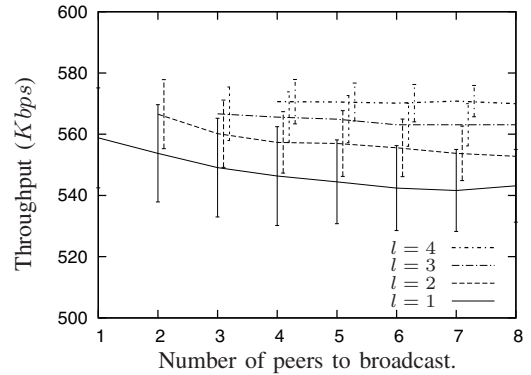


Fig. 3. Plot of maximum possible throughput for UDP proxy, resulting from varying number of peers to broadcast per round ( $k$ ) and number of responses to wait for ( $l$ ). Error bars indicate the standard deviation.

to obtain the estimated performance with different values of  $k$  and  $l$ .

Fig. 2(a) and 2(b) shows the average time taken to find a proxy in our simulations for both UDP and TCP respectively. Broadcasting to more peers decreases the connection time until  $k = 5$ , after which there is no significant improvement in increasing  $k$ . The graphs also show that waiting for more than one response ( $l$ ) would increase the connection time significantly. To evaluate if it would be helpful to wait for additional responses, we plot the maximum throughput attain-

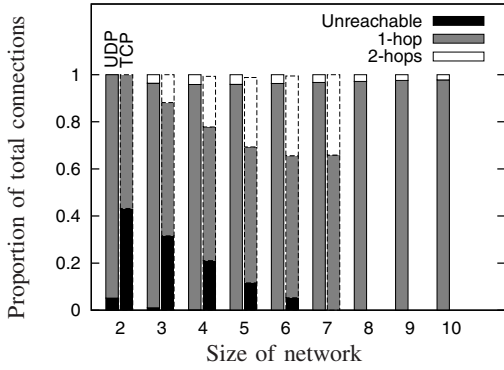


Fig. 4. Average proportion of degree of separation (minimum number of hops) separating each pair of peers for varying network sizes, both for UDP (on the left) and TCP (on the right). TCP results for more than 7 peers were not available from our data set.

able amongst the proxies selected in Fig. 3 for UDP. While the average throughput appears to increase with  $l$ , the large spread of the data points suggests that the actual gain might not amount to much. We obtained similar results for TCP, which has an average throughput that is about  $50 Kbps$  lower. The estimated minimum latencies between the chosen proxies does not vary significantly as well. In light of these results, we chose to broadcast proxy requests to 5 peers concurrently and pick the first responding peer as the proxy.

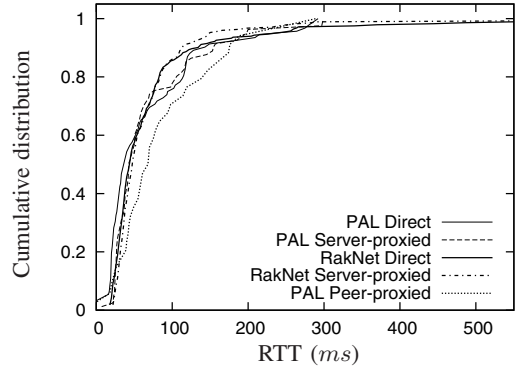
In Fig. 4, we plot the number of hops separating pairs of peers for networks of different sizes. We found that the UDP hole punching success rate for the Karkinos data set at 95% is similar to that for Tankie MAX, as shown in Table I, while the TCP success rate is significantly lower at less than 60%. When nodes are separated by two hops, a proxy will be required. We see from these results that for UDP, a proxy can always be found when there are at least 4 nodes, while for TCP, a proxy can always be found when there are at least 7 nodes. Since the Karkinos data set is relatively small and has TCP hole punching success rates that are significantly lower than that for the larger Tankie MAX data set, it is plausible that TCP proxies might be easier to find in practice.

#### D. Proxy Performance

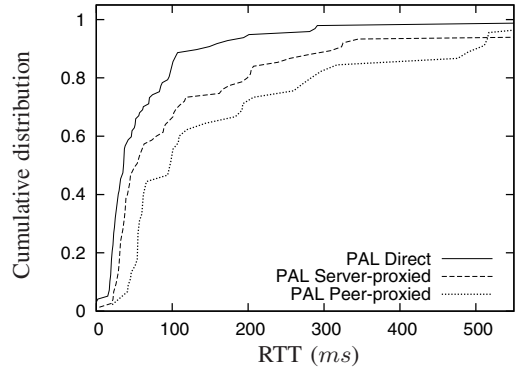
In Fig. 5(a), we plot the round-trip latencies observed for different types of UDP connections for both PAL and RakNet. On the whole, we see that direct and server proxied connections are comparable with peer-proxied connections, which are slower by about 20 to 30  $ms$ . We believe that this is due to the extended queuing and processing delays in a peer proxy over a dedicated server.

In Fig. 5(b), we compare the measured round-trip latencies observed for direct and proxied TCP connections by sending a ping signal down the channel and measuring the time it takes to receive a response. Results are similar to that for UDP, though proxied TCP connections have significantly higher latencies. This is likely a result of the segmentation and buffering in our TCP proxy implementation.

We also investigated how NAT proxies would affect the achieved throughput. In Fig. 6(a) and 6(b), we plot the mea-



(a) UDP



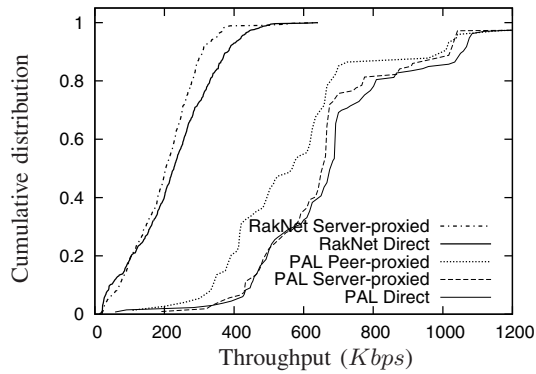
(b) TCP

Fig. 5. Cumulative distribution of RTT for direct and proxied UDP and TCP connections.

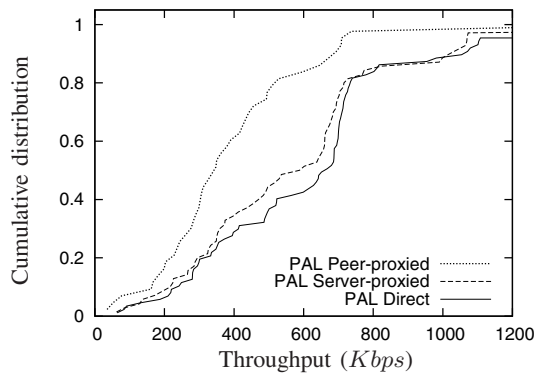
sured throughputs for direct and proxied connections for UDP and TCP respectively. For PAL, UDP throughput is estimated by sending as much data as possible within an interval of 2.5  $s$ , while TCP throughput is estimated by sending 2  $MB$  of data between end-hosts. The values of 2.5  $s$  and 2  $MB$  were chosen because we found that these were the minimum values required to achieve good measurement accuracy after a calibration test. The graphs show that peer-proxied connections are generally slower than direct and server-proxied connections, but still achieve reasonably high throughput.

Fig. 6(a) also shows the UDP throughput for RakNet, which was measured differently because of its built-in flow control mechanism. We filled the transmit buffer with 400  $KB$  of data and let RakNet control the actual sending rate. We see that the achieved throughput for RakNet is much lower than PAL. The graph is also smoother as the send rates do not seem to reach the bandwidth limits of the users' Internet connections. The effectiveness of this flow control mechanism is apparent in Fig. 7, which shows that RakNet achieves significantly lower loss rates for UDP packets as compared to PAL.

We also investigated the performance of our protocol in the case that a proxy fails and an alternative peer is required to restore the proxied connection. We found that in our implementation, a proxy connection can typically be restored within 6  $s$  for UDP and within 18  $s$  for TCP.



(a) UDP



(b) TCP

Fig. 6. Cumulative distributions of observed throughput for direct and proxied connections for UDP and TCP.

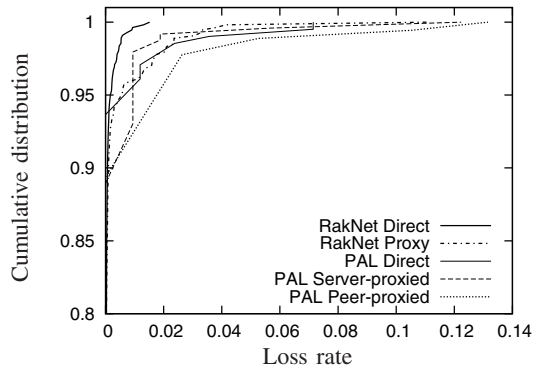


Fig. 7. Cumulative distributions of observed loss rates for UDP connections.

### E. Overhead

As with any high-level network abstraction or framing, PAL incurs some overhead. As the relative overhead is dependent on the actual amount of data transferred, we assume packets are filled to the maximum transmission unit (MTU) size of 1,500 bytes for Ethernet. UDP packets over direct connections would thus require a minimum overhead of 4%. This increases to 7% for proxied connections. In contrast, RakNet has a minimum overhead of only 2% for both direct and proxied connections because it uses a connection-based proxying protocol. PAL, on the other hand, uses a stateless proxying protocol for easier failure recovery.

For our TCP proxy implementation, the additional headers added into each segment results in a minimum overhead of 0.5% (assuming each segment is of optimal MTU size) excluding the native overhead of TCP. This is a small trade-off in allowing transparent failure recovery in an otherwise connection-based protocol.

### V. CONCLUSION

NAT proxies will almost certainly be required to support a p2p game because NAT hole punching does not always succeed. We showed with a measurement study that it is feasible to use peers as proxies even with a simple and conservative NAT hole punching implementation since it is relatively easy to find a peer that can act as a proxy. The performance achieved with peer NAT proxies is also comparable with server-based implementations. Our current peer NAT proxy implementation is still somewhat naive and further optimizations remain as future work.

### REFERENCES

- [1] J. Callahan. (2009, Apr.) Demigod still dealing with multiplayer networking issues. [Online]. Available: <http://news.bigdownload.com/2009/04/28/demigod-still-dealing-with-multiplayer-networking-issues>
- [2] B. Wardell. (2009, May) Demigod: So what the hell happened? [Online]. Available: <http://forums.stardock.com/352561>
- [3] P. Francis and R. Gummadi, "IPNL: A NAT-extended Internet architecture," in *Proc. SIGCOMM'01*, 2001, pp. 69–80.
- [4] P. Francis, "Is the Internet going NUTSS?" *IEEE Internet Computing*, vol. 7, no. 6, pp. 96–95, 2003.
- [5] B. Ford, P. Srisuresh, and D. Kegel, "Peer-to-peer communication across network address translators," in *Proc. ATEC'05*, 2005, p. 13.
- [6] K. Pussep, M. Weinert, A. Kovacevic, and R. Steinmetz, "On NAT traversal in peer-to-peer applications," in *Proc. WETICE'08*, 2008, pp. 139–140.
- [7] S. Guha and P. Francis, "Characterization and measurement of TCP traversal through NATs and firewalls," in *Proc. IMC'05*, 2005, p. 18.
- [8] RakNet. [Online]. Available: <http://www.jenkinssoftware.com>
- [9] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, "Session Traversal Utilities for NAT (STUN)," RFC 5389 (Proposed Standard), Internet Engineering Task Force, Oct. 2008. [Online]. Available: <http://www.ietf.org/rfc/rfc5389.txt>
- [10] S. Guha, Y. Takeda, and P. Francis, "NUTSS: A SIP-based approach to UDP and TCP network connectivity," in *Proc. SIGCOMM'04*, 2004.
- [11] J. Rosenberg, R. Mahy, and C. Huitema, "Traversal Using Relay NAT (TURN)," Internet-Draft, Internet Engineering Task Force, Sep. 2005. [Online]. Available: <http://tools.ietf.org/id/draft-rosenberg-midcom-turn-08.txt>
- [12] S. Guha, N. Daswani, and R. Jain, "An experimental study of the Skype peer-to-peer VoIP system," in *Proc. IPTPS'06*, 2006.
- [13] W. Kho, S. A. Baset, and H. Schulzrinne, "Skype relay calls: Measurements and experiments," in *Proc. INFOCOM'08*, 2008.
- [14] S. Ren, L. Guo, and X. Zhang, "ASAP: an AS-aware peer-relay protocol for high quality VoIP," in *Proc. ICDCS'06*, 2006, p. 70.
- [15] J. Dowling, J. Sacha, and S. Haridi, "Improving ICE service selection in a p2p system using the gradient topology," in *Proc. SASO'07*, 2007, pp. 285–288.
- [16] H. X. Nguyen, D. R. Figueiredo, M. Grossglauser, and P. Thiran, "Balanced relay allocation on heterogeneous unstructured overlays," in *Proc. INFOCOM'08*, 2008, pp. 126–130.
- [17] D. D. Clark and D. L. Tennenhouse, "Architectural considerations for a new generation of protocols," in *Proc. SIGCOMM'90*, 1990, pp. 200–208.
- [18] J. Yong, J. Bai, D. Seah, A. Razeen, H. Nguyen, J. Liew, Z. H. Koh, and B. Leong. (2009) Tankie MAX. [Online]. Available: [http://apps.facebook.com/tankie\\_max](http://apps.facebook.com/tankie_max)
- [19] A. Razeen, D. Seah, and B. Leong. (2009) Karkinos. [Online]. Available: [http://apps.facebook.com/karkinos\\_alpha](http://apps.facebook.com/karkinos_alpha)