URL Forwarding for NAT Traversal

Md. Ahsan Raza, Reaz Ahmed and Raouf Boutaba
School of Computer Science, University of Waterloo
Email: {m6raza, r5ahmed, rboutaba}@uwaterloo.ca

***Abstract*—Rapid technological advancement has induced widespread adoption of smart, powerful and multi-purpose end-user devices (e.g., phones, tablets, set-top boxes, gaming console etc.) in our daily life. Unfortunately, the available mechanisms for accessing, configuring and controlling these devices are still primitive, and require physical access to the devices. Enabling remote access to these devices through a widely used protocol, such as HTTP, can significantly improve their usability and can foster innovative applications. However, the end-devices are often NATed by home gateways, which is not suitable for HTTP access over the Internet. In this work, we present a mechanism for seamless traversal of HTTP traffic through home gateways. Our approach builds upon the existing Web-technology. It allows remote access using global DNS names, despite the end-devices having local IPs. As a proof of concept, we realize the proposed architecture on the well-accepted OpenWRT platform, and report experimental results on device access performance.**

I. Introduction

End user devices such as smart phones, tablets, TV set-top boxes and gaming consoles, have evolved significantly in processing power, storage, battery life and network connectivity. These devices are becoming cheaper every year, and offering wider range of capabilities including GPS, gyroscope and accelerometer. However, we need to have physical access to these devices to use their capabilities.

Remote access to the end-devices can foster a wide-range of applications such as hosting content directly from a device, remote control and configuration, remotely access on-device sensors and camera, etc. The impact can be significant. For example, we can achieve better privacy and control over shared personal content if we can host them from our own devices.

In order to grant remote access to the end devices, we have to address three eminent issues: a) access protocol, b) naming and c) NAT (Network Address Translation) traversal. First, end devices come from different vendors. They have heterogeneous capabilities. Thus, converging to a single protocol to access every end device is challenging. Secondly, a globally unique name is required for identification and remote access. The naming system has to efficiently handle billions of names, and frequent change in name to address association. Third, end devices are usually behind NAT boxes, which makes it very difficult to establish incoming connections for hosting services in these devices.

The first two issues, i.e., access protocol and naming, can be addressed by using Hypertext Transfer Protocol (HTTP). HTTP has become the de facto communication protocol for online services and applications, including video streaming (e.g., YouTube), peer-to-peer file sharing (e.g., BitTorrent) and online collaborative document editing (e.g., Google Docs). Programming tools for HTTP server and client development are available for almost every platform. The naming mechanism used by HTTP is URL, which is also widely accepted. It is a natural consequence to name the end devices using the host name part of an URL. Name resolution mechanisms offered by DDNS and pWeb [1] offer dynamic binding and efficient resolution of DNS names to end devices.

Unfortunately, the available mechanisms for addressing the third issue, i.e., NAT traversal, are not adequate. They require manual configuration or adhoc patches. In this work we propose *URL forwarding* – a simple mechanism for HTTP traffic traversal through NAT boxes (or home gateways). Our approach builds upon the existing Web-technology. It allows remote access using global DNS names, despite the end-devices having local IPs. We have made our solution available for public use at *pwebproject.net/url-forwarding*.

The rest of this paper is organized as follows. In Section II, we present an overview of URL forwarding and its benefits. Section III presents our solution for implementing URL forwarding within a home gateway. Our experimental testbed and performance evaluation are presented in Section V. An overview of the existing NAT traversal mechanisms and their comparison to URL forwarding is presented in Section VI. Finally, we conclude in Section VII.

II. URL Forwarding and Benefits

A. An Overview

DNS maps fully qualified domain names (e.g. www.google.com) to static IP addresses so that the users do not have to remember the IP address of the server. Instead, they can use easy to remember names. One drawback of using DNS is that not everyone willing to serve content and services has a static IP address. For example, regular Internet users receive temporary public IP addresses that change over time. A number of systems including pWeb and numerous DDNS providers allow end-users to obtain DNS names and dynamically attach IP addresses to those names. However, it remains the responsibility of the user to redirect incoming traffic to the devices behind their home gateways. Usually a home gateway implements a NAT mechanism to multiplex incoming requests to the local devices. There are different types of NATs and traversal mechanisms, but they share a common problem – local devices find it difficult to create server sockets for hosting any service for users outside the local network.

The proposed URL forwarding mechanism allows seamless traversal of HTTP traffic through home gateways. We assume that a user has obtained DNS-compatible names for his/her end devices and has the means to associate their home gateway's public IP address to his/her device names. For our implementation we have used pWeb's naming system. However the

URL forwarding mechanism presented in this paper is equally applicable to any DDNS provider. Below is an outline of the proposed URL forwarding mechanism: a) We install a web request interceptor (WRI) module in the home gateway or on a standalone machine in the local network. For the second case, we need to add a port-forwarding entry in the home gateway that points to the machine running the WRI. In both cases, the WRI will be listening to the incoming HTTP traffic and will work as a reverse-proxy; b) A local device registers its DNS-compatible name and local IP address with the WRI using standard HTTP messages; c) All incoming HTTP messages are forwarding to the WRI, which extracts the target URL from the HTTP message header. Then it looks up its local registration database for a match and forwards the request to the registered local device.

B. Benefits

The proposed URL forwarding technique has the following benefits compared to conventional NAT traversal mechanisms.

1) Simplicity: NAT traversal mechanisms do not provide any explicit way for an end-device to notify a home gateway that it wants to listen (i.e., open a server socket) on a specific port. As a result, some complicated adhoc measure is required to install a forwarding rule at the home gateway. On the contrary, the proposed URL forwarding mechanism provides simple HTTP message for explicit device registration.

2) Identification: In NAT traversal, local devices are identified by their IP addresses. Every time a device reconnects it gets a new IP address. So, there is no way to identify a returning device. Whereas, for URL forwarding, devices are identified by their globally unique URL. A home gateway can use this URL for identifying a returning device and automatically register it.

3) Standardization: Since URL forwarding uses the standard HTTP protocol and a few well-defined messages, it will be easy to standardize and implement in all platforms.

4) Innovation: A home gateway can work as a local name resolver and thus can support many new applications, such as automatically syncing devices as they become available in the local network, managing devices from a single node in the network etc. On the contrary, NAT traversal mechanisms rely on IP address and allow requests from selective sources only.

C. Integration with the Internet

URL forwarding provides a mechanism for multiplexing HTTP request to multiple local devices through a home gateway. There are two prerequisites for URL forwarding: a) acquire DNS names for each end device and b) associate a local device name to its gateway's public IP address. DDNS and pWeb are two alternative ways to fulfill these two prerequisites. To use DDNS, one has to subscribe to a provider (e.g., *dyndns.com* or *noip.com*) and configure his home gateway (or a local machine) to bind the gateway's IP address to the DNS name given by the DDNS provider. Each DDNS provider uses its propitiatory communication protocol and are not compatible with each other. pWeb, on the other hand, is an open system. In pWeb a user can pick his own DNS name and can contribute to the name resolution process. In addition, pWeb allows a user to advertise his local services/contents to the World through an open search engine. Considering the flexibility in pWeb, we have made our prototype implementation compatible with pWeb messaging. However, the proposed concept and implementation can easily work with any DDNS provider.

III. The Proposed Approach

In this section we present the high level architecture of our solution. Figure 1 illustrates the functional components of the architecture and how they interact with each other. There are three major components in the solution: 1) Web Request Interceptor (WRI), 2) device tracker and 3) updater.

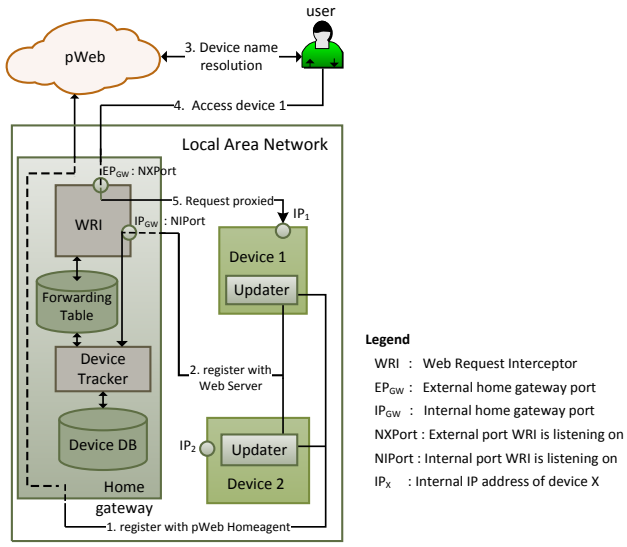


Fig. 1: Proposed System Architecture

1) Device Tracker: The device tracker runs in home gateway and maintains a database of all devices in the network that are registered with WRI. Every time an Updater in an end device calls the Device Tracker, which updates the device database (DB) with the device's DNS name, internal IP address, listening port and a PREFIX_LEN (more details below). Based on the content of the database, the device tracker updates the forwarding table. Then, the device tracker sends a command to WRI and instructs it to reload the updated forwarding table. With its configuration up to date, WRI then plays its role to proxy all incoming traffic to the appropriate device.

2) WRI: The WRI runs on the home gateway and is a crucial component in our solution. In principle, the WRI is a web server that works as a reverse proxy in our solution. The WRI listens on a standard port (e.g., 80 or 8080). It intercepts the incoming HTTP messages and finds the target URL by inspecting the HTTP request header. Then it looks up its internal forwarding table to find the local IP address of a registered device that matches the target name. Each entry in the forwarding table has a PREFIX_LEN that governs the number of DNS name prefixes to match. Full DNS name is matched if the PREFIX_LEN is set to zero. Devices using pWeb should set the PREFIX_LEN to 3 since names in pWeb are of the form *devicename.username.servername.dns-suffix*. In pWeb, the *dns-suffix* points to a DNS server that

works as a gateway between DNS and pWeb. For example, the name *nexus.bob.uwaterloo.dht.pwebproject.net* refers to the *nexus* phone of user *bob*, who is registered to the *uwaterloo* server. Here, *nexus.bob.uwaterloo* is a unique name in pWeb and *dht.pwebproject.net* points to a DNS gateway. There can be many DNS gateways, and the *dns-prefix* will change accordingly. For example, if a user uses *mygw.ca* as the DNS gateway then *bob*'s home gateway will see the name *nexus.bob.uwaterloo.mygw.ca* in HTTP header. After inferring the target device, the WRI forwards the request to the target device which sends a response back to the requester.

3) Updater: The updater runs on an end-device and registers the device with WRI and with pWeb. When a user turns on his device, the updater is loaded and the user provides his or her pWeb credentials and other meta-information. The updater then invokes the Device Tracker to register itself with WRI. It then updates its network address (IP:port) with pWeb.

IV. Functional Overview

A. Device registration

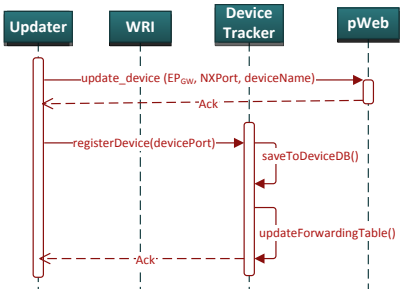


Fig. 2: Registration with device tracker and pWeb

Figure 2 illustrates the process to register a device with WRI and pWeb. Upon start up, the updater software collects user's pWeb credentials and the port number of any web service that the user wants to host from his device. The updater then registers the device using the following steps:

First, the updater invokes the *update_device* message from pWeb's REST API (see *developer.pwebproject.net* for details). The *update_device* message requires several pieces of information including the public IP address (EP_{GW}) of the home gateway, the listen port (NXPort) of WRI and the user's pWeb credential. EP_{GW} can be extracted in different ways, such as calling a public server to return the address or using one of pWeb's servers. The NXPort is retrieved using the web interface exposed by WRI. The user's pWeb credentials are collected from the user in the very first step.

With all the necessary information available, the updater module invokes a pWeb's server (aka home agent), e.g., *uwaterloo.pwebproject.net*, to updates the device's network address. pWeb only supports HTTPS to register devices so that the user's credentials are encrypted when they are sent over the Internet. After a successful update, pWeb adds a DNS entry for the device (e.g., *nexus.bob.uwaterloo*) that points back to the the home gateway (i.e., $EP_{GW} : NXPort$).

After updating pWeb, the updater registers with the device tracker. For this registration, the updater extracts metadata information such as the local IP address of the home gateway. The updater calls a server process running on the home gateway (device tracker in our case) that will register the device. The updater provides the relevant peer data (collected or computed previously) to this process: the device DNS name in the pWeb system, the listening port on the device and the private IP address of the device. The process does some basic data validation (e.g. ensures the device port is valid) then goes on to update its own internal database containing the devices that are registered with WRI. If a device with the same DNS name already exists, the IP address is updated. If, for any reason, the pWeb registration process fails, the registration with the reverse proxy server on the home gateway is halted.

B. Name Resolution and Content Request Process

Figure 3 illustrates the process of accessing a device behind a home gateway. First an external user, willing to access a NATed service, makes a DNS request to resolve the device name (e.g., *nexus.bob.uwaterloo.dht.pwebproject.net*). The DNS forwards the resolution request to the pWeb name resolution system via a DNS gateway (here *dht.pwebproject.net*). pWeb in turn finds the home agent (here *uwaterloo*) and retrieves the network of address ($EP_{GW} : NXPort$) of the home gateway. It should be noted that this two step name resolution process remains transparent to the end users.

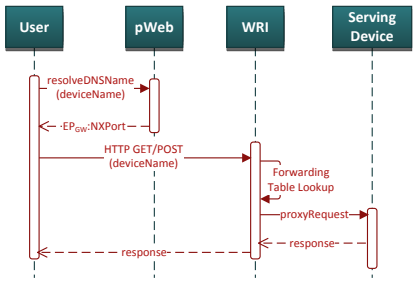


Fig. 3: Name resolution and device access process

After learning the home gateway's network address, the external user (usually a web browser) makes a HTTP GET/POST request to the home gateway. Upon receiving the HTTP request, the WRI inspects the "Host" header field and obtains the global DNS name of the target device. This information is enough for the WRI to know which device to forward the request to. The WRI maps unique DNS names to the internal IP addresses of the registered devices within the network. It received this information, along with the port on which the peer's server process is listening on, during the peer registration process. With the internal IP address and port of the producer peer known, the WRI proxies the request to that IP address and port. The web service running on the serving device processes and responds to the request. The WRI receives the response and sends it back to the external user. The WRI listens on a single port, yet it may forward the request to multiple devices in its local network.

C. Auto-register and Auto-unregister Process

The idea behind auto registering and auto unregistering devices is two fold. First, it makes the solution more user-friendly in that users do not need to register their devices

constantly whenever they move in and out of the same network. Second, it lets the WRI only serve the devices that are still on the network, which improves performance. Any device that exits the network will be automatically unregistered.

When a device successfully registers itself for the very first time, the updater persists the state to a local persistent store. The updater then registers the device with the WRI every 5 minutes. The device tracker module in the home gateway registers the device. In addition, it stores the time of registration. The updater adds itself to the operating system's startup sequence (if applicable). Next time the operating system boots up, the updater reads from its persistent store, and sees that it had previously registered a user and therefore it re-registers that user. The device tracker re-registers the device with effectively no change, however, it updates the device registration time and network address.

The home gateway uses soft registration. If a device's last registration time in the forwarding table is older than 20 minutes, it is automatically unregistered and the corresponding entry in forwarding table is removed. This ensures freshness of all devices registered with the reverse proxy server.

D. Security

In this section we discuss a few of the security aspects that may affect our solution. We briefly discussed one security issue in Section IV-A, i.e., a malicious user masquerading as a genuine user and being able to receive the requests targeted to the genuine user. This vulnerability can exist if a device can be registered with WRI without any authentication. A genuine user may register a device with pWeb and then with WRI. A malicious user can subsequently register his/her device with WRI by invoking a simple HTTP GET request. If the malicious user knows the genuine user's pWeb username, device name and home agent name, (s)he can register for the device to receive requests targeted to the genuine user.

One way to potentially fix this issue is to pass the user's pWeb credentials to the server process that performs the registration with WRI and have it ensure that the user's pWeb credentials are valid before registering with WRI. This will ensure that only the genuine users can register. However, this also means that one should use encryption (e.g. HTTPS) to transport the data between the device and the registration process, since the user's pWeb credentials are now being sent over the network.

Another vulnerability can exist if the auto register functionality is implemented, and the user's pWeb credentials are saved in a persistent store without being encrypted. There are many techniques used by browsers to encrypt users' passwords before being saved to a persistent store. Zhao et al. [12] presented a technique that can be used by browsers to save users' password in a secure manner to a persistent store. Similar techniques should be used to encrypt users' passwords when they are saved by the updater module.

V. Evaluation

A. Experimental Setup

We setup an experimental environment based on the architecture proposed in Section III. We first needed a home gateway platform that could support a reverse proxy web server as well as dynamic server-side languages such as PHP. OpenWRT, a linux variant, is such a home gateway platform. It supports a large number of home gateway hardware[1]. Most importantly, it supports running reverse proxy web server software and PHP[2] within its environment. Given these characteristics, we decided to use OpenWRT as the home gateway platform for our experiment.

Given that we need a web server that can work as a reverse proxy, we chose nginx that is a well-known, high performance web server. PHP is a widely used server-side language that is easily deployable in linux environments, so we chose PHP for developing the server-side processing software.

To setup a working environment, we deployed a virtual network using Oracle Virtual Box. The host machine had Windows 8.1 with an out of the box Intel Core i7-3770 processor at 3.4 GHz and 12 GB RAM. We setup an OpenWRT VM for emulating the home gateway, and provided it with a single core processor and 256 MB of RAM. The OpenWRT VM had two network interfaces: one that allows the VM to connect to the Internet, while the other one allows local ndevices to connect to the OpenWRT VM and make the OpenWRT VM their home gateway. To simulate devices on the network, we setup up to 7 Ubuntu VMs, each with two cores and 1 GB RAM. The devices had a single network interface to connect to the OpenWRT VM.

We then installed nginx and PHP on the OpenWRT VM. Nginx listened on port 8080 for external requests. For registration requests from within the network, nginx listened on port 8980 locally. On the Ubuntu VMs, we ran Apache web server to listen to and respond to requests from the external users.

We wrote a device tracker as described in Section III using PHP. The device tracker script uses a binary file as its device database. When a device requests to register with nginx, the device tracker saves the device information to its internal database and updates nginx configuration file. Then it sends a reload signal to nginx, which forces nginx to reload its configuration. To register the Ubuntu VMs with pWeb and nginx as described in Section III, we developed the Updater module in Java. After registering the device with pWeb and nginx, we were able to receive requests using the global device name from users outside the network and respond to those requests.

B. Performance Results

1) Performance Results for the Proposed Architecture:

Test case - basic RTT: We first evaluate the performance of basic RTT: the end-to-end time it takes for the response to a request to return to a peer after they issue the request through the device's pWeb name. We control for two variables in the response times: effects of pWeb DNS resolution time and effects of webpage caching.

Test methodology: We host a webpage of negligible size on 4-7 Ubuntu VMs, with each VM representing a device, and measure the response time. From two separate machines,

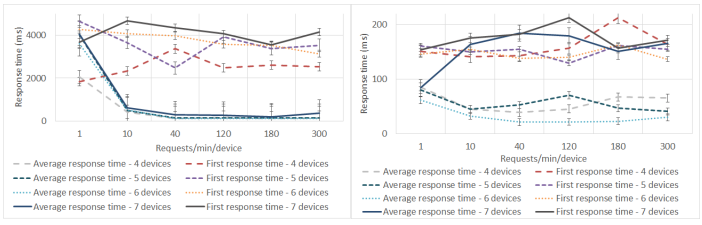
[1] http://wiki.openwrt.org/toh/start

[2] http://wiki.openwrt.org/doc/howto/http.nginx

placed 150 kilometers away from each other, we ran a process on each laptop lasting 3 minutes that simultaneously sent a varying number of requests per minute to each device. This process was then terminated, and a new instance of the process started. The process ran for a total of 10 times. We averaged the results of the 10 processes and then averaged the results from the two laptops. The final value calculated represents a single data point on a graph. The amount of requests sent per minute per device varied from 1 - 300 depending on the test.

Effects of pWeb DNS name resolution time: We attempt to reach the producer peer's device through two paths: through the device's pWeb name and directly through the home gateway's public IP address. To reach the device directly through the home gateway's public IP address, we send an HTTP request as usual but use the home gateway as the proxy server. This method retrieves the page directly from the server without going through pWeb. We can measure the performance of the system with and without the latency caused by pWeb DNS name resolution. Figure 4a considers only the requests that were made via the device's pWeb name. Figure 4b considers only the requests that were sent directly to each device through their home gateway's IP address.

Effects of webpage caching: We measure the response times depending on whether the webpage we are attempting to retrieve has been cached or not. To measure the caching metric, we divide response time into two sets: average response time of all requests and the average response time of the first requests. The latter requests are uncached and therefore give us a measure of how fast an uncached request returns. The uncached requests appear as "first response time" data points in the graphs. "Average response time" is the average response time for all requests, cached or uncached. "First response time" is the average response time of those requests that are the first to run when the test process starts. For example, when we send 10 requests per minute per device for a duration of 3 minutes, this is equivalent to 1 request to each device every 6 seconds. The "first response time" only considers the requests issued in the first 6 seconds. We repeat this test 10 times and average the response times to ensure an outlier does not seriously skew the results.



(a) Request through pWeb (b) Direct request

Fig. 4: Response times (ms) for HTTP requests

Results: Figures 4a - 4b show the results of the tests. It is clear from the results that the latency introduced by pWeb name resolution is significant when the results are uncached: the final response time is between 2000 - 4500 milliseconds (ms). With the results cached, however, using the pWeb name gives the response in a negligible amount of time: between few hundred to 2000 ms. Directly accessing content from the device through the public IP address always returns the response in a negligible amount of time. The uncached response time is between 100 - 250 ms. The response time is even smaller when the requests are cached: between 20 - 200 ms. It is clear from these results that the latency introduced by the proposed architecture is minimal to negligible.

It is interesting to note the drop in the pWeb response time as the number of the requests per minute increase. This is likely due to cached DNS records in any number of DNS servers that are used in the name resolution.

2) Files of Non-Trivial Size:

Test case - upload capacity fairness and utilization: In this section, we evaluate two key metrics: the fairness of the system upload capacity distribution and its utilization.

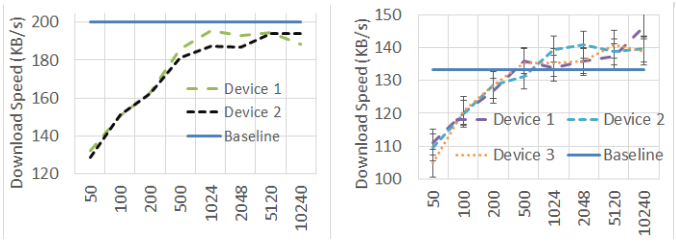
Test methodology: We run the test in an environment that has an upload speed of 400 KB/s. We run the test against varying number of Ubuntu VMs, with each Ubuntu VM emulating a device, and host an Updater on each one of the VMs. We deploy a PHP script in the VM that generates garbage content of arbitrary size. The script takes a single parameter x, a positive integer, and generates content of size x KB.

From a single machine, we sent simultaneous requests to each VM to produce content of varying size and recorded the upload times. This process was repeated 10 times, after which the upload times from the 10 runs were averaged. The averaged upload times were converted to kilobytes per second. Each machine's average upload speed represents a single point on a graph. In this test, we only sent requests directly to the IP address using the same method outlined in section V-B1 so as to not have the results affected by pWeb DNS name resolution time. The number of Ubuntu VMs per test were between 2-5, depending on the test case. The "Baseline" shows the optimum upload speed per machine - it is the upload capacity of the network divided by the number of machines attempting to upload files simultaneously.

Results: Figures 5a - 5d show the results of the tests. The trend lines show that the upload capacity distribution is fair across the devices: the upload speed experienced by any individual device at any given point does not significantly differ from the upload speed experienced by other devices. The upload capacity utilization approaches baseline as the file upload size increases. With smaller file sizes, the time to establish the connection takes a great chunk of the total upload time and thus brings down the overall upload time. As the file size increases, the time to establish the connection has less of an impact on the upload speed and they appear and stabilize around the baseline. Files of size 500 KB - 1 MB and greater have a fair and full utilization of the upload capacity.

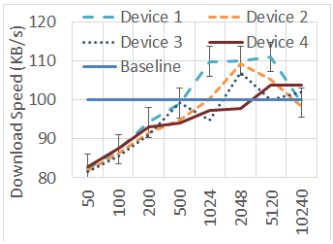
3) OpenWRT Scalability:

Test case - OpenWRT Scalability: In this section, we evaluate the scalability of OpenWRT in face of large number of requests. To evaluate this, we measure the RTT of requesting a page of negligible size when the number of requests per minute to OpenWRT are abnormally high and the number of devices on the network is on the higher side, though not unusually high.

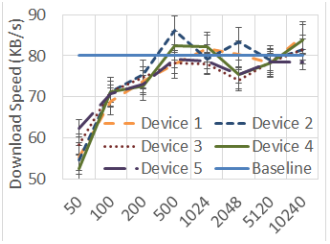


(a) Requesting files from 2 devices simultaneously

(b) Requesting files from 3 devices simultaneously



(c) Requesting files from 4 devices simultaneously



(d) Requesting files from 5 devices simultaneously

Fig. 5: Speed for uploading files from 2-5 devices simultaneously (file sizes in KB)

Test methodology: We host a webpage of negligible size on 7 Ubuntu VMs, with each VM representing a device on the network. We use a methodology similar to that used in section V-B1. From a single machine, we ran a process lasting 3 minutes that simultaneously sent a varying number of requests per minute to the OpenWRT VM. The nginx instance on OpenWRT forwarded the requests to the relevant Ubuntu VM and the VM responded to the request. This process was then terminated and a new instance of the process started. The process ran for a total of 10 times. The final value calculated represents a single data point on a graph. In this case, however, we only sent requests directly to the IP address using the same method outlined in section V-B1. This ensures that the pWeb DNS resolution time does not affect the results of the experiment. The amount of requests sent per minute varied from 7 - 2100 depending on the test.

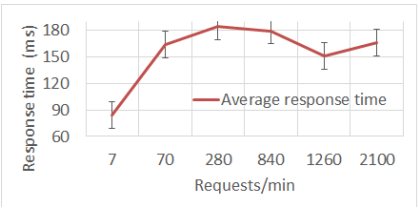


Fig. 6: Response times (ms) for direct requests to OpenWRT

Results: Figure 6 show the results of the test. Starting with 7 requests per minute and amping it up to an unreasonable 2100 requests per minute, the response time basically remains unchanged. Given the high number of devices on the network (7) and the unreasonable requests per minute (2100), it is clear that the scalability of OpenWRT is not affected by a high number of requests or the number of devices on the network.

VI. Related Works

NAT traversal allows outside hosts to see and communicate with the devices behind a NAT [3]. There are a wide variety of NAT traversal solutions, each solution being very different from the other.[2][8] Here we discuss some of these solutions.

1) Universal Plug and Play (UPnP): offers a simple and robust way to connect devices from different vendors [3]. It can automatically forward ports on an UPnP-enable home gateway for NAT traversal. UPnP has known security issues. Malwares can use UPnP to open ports and have unfettered access to the local devices [4].

2) Session Traversal Utilities for NAT (STUN) [11]: is not a NAT traversal solution by itself. However, it allows a device behind a NAT to discover its public IP address, public port and if it is located behind a NAT [6]. To utilize STUN, node A from its internal IP:port A_I invokes the STUN server. The home gateway translates A_I to the external IP:port A_E. The STUN server sends back A_E to node A. Node B then performs the same steps and gets the external port B_E. By means of a central server, nodes A and B now exchange their public ports. They then each connect to the public ports of the other peer. Since each node's respective home gateways have a mapping for these public ports, they allow the packets to get through to the respective node.

3) Traveral Using Relays around NAT (TURN) [10]: is a NAT traversal protocol that can work with most NATs, including symmetric NATs. At a high level, a TURN server essentially acts as a relay between the two nodes, i.e. all communication between the nodes go through the TURN server. TURN is a more complex protocol than STUN and incurs heavy load on the TURN server. It is only used as a last resort if STUN or direct connectivity does not work.

VII. Conclusion

In this paper, we set out to improve the way users can share their content with the World. The contemporary model of sharing content needs an overhaul, and our work is a step in getting there. We provided a solution that lets users share content from their devices without having to open ports or if necessary, open only a single port. Our solution integrates seamlessly with state of the art Web technology to provide remote access, no matter which network the device is on. We presented optional features that we believe can improve the end-user experience. We wrote a client software that is compatible with Android or any platform running JRE. This software lets users register with pWeb and with the system running the WRI. The performance results from our experiments show that the solution deployed in the serving device's network has excellent performance with no bottleneck caused by the WRI or home gateway, with fair upload capacity sharing and with full upload capacity utilization regardless of the size of the content or the request volume. To proliferate our solution for public use, we have made the installers and the OpenWRT VM images of both the main and the alternative architectures available at *pwebproject.net/url-forwarding*. This page also provides the device client software to the general public for deployment in their local networks. In the next release of the software in intend to improve security and to provide updater module for popular operating systems.

REFERENCES

[1] R. Ahmed, S. R. Chowdhury, A. Pokluda, M. F. Bari, R. Boutaba, and B. Mathieu. pWeb: A Personal Interface to the World Wide Web. In *IFIP Networking*, 2014.

[2] L. DAcunto, J. Pouwelse, and H. Sips. A measurement of NAT and firewall characteristics in peer-to-peer systems. In *Proc. 15-th ASCI Conference*, volume 5031, pages 1–5. Advanced School for Computing and Imaging (ASCI), 2009.

[3] Z. Haddad. Implementing a TCP Hole Punching NAT Traversal solution for P2P applications using Netty. Master's thesis, University of Stirling, 2010.

[4] C. Hoffman. HTG explains: Is UPnP a security risk? http://www.howtogeek.com/122487/htg-explains-is-upnp-a-security-risk/, Aug. 2010. How-To Geek.

[5] M. Holdrege and P. Srisuresh. Protocol Complications with the IP Network Address Translator. *RFC-3027*, Jan. 2001.

[6] Z. Hu. NAT Traversal Techniques and Peer-to-Peer Applications. In *HUT T-110.551 Seminar on Internetworking*, Apr. 2005.

[7] D. Maier and O. H. amd Jurgen Wasch. NAT Hole Punching Revisited. In *IEEE Conf. On Local Computer Networks (LCN)*, Oct. 2011.

[8] A. Müller, A. Klenk, and G. Carle. On the applicability of knowledge based NAT-traversal for home networks. In *NETWORKING 2008 Ad Hoc and Sensor Networks, Wireless Networks, Next Generation Internet*, pages 264–275. Springer, 2008.

[9] L. Popa, A. Ghodsi, and I. Stoica. HTTP as the narrow waist of the future internet. In *ACM SIGCOMM Workshop HotNets*, 2010.

[10] J. Rosenberg, R. Mahy, and P. Matthews. Traversal Using Relays around NAT (TURN). *RFC-5766*, Apr. 2010.

[11] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). *RFC-5389*, Oct. 2008.

[12] R. Zhao and C. Yue. All your browser-saved passwords could belong to us: A security analysis and a cloud-based new design. In *Proc. ACM DASP*, pages 333–340, 2013.