Last-mile IPv6 Teredo Tunnelling Performance Evaluation

Boris Espinoza-Kalchev

Computer Science
Jacobs University Bremen
Campus Ring 1
28759 Bremen
Germany

Type: *Bachelor Thesis Report*
Major: *Computer Science*
Date: *May 28, 2012*
Supervisor: *Prof. Dr. Jürgen Schönwälder*

Abstract

On the edge of the inevitable transition to the IPv6 protocol, it is clear that the outright 'jump' to the newer Internet protocol is impossible. Thus, the actual goal is not a transition from IPv4-only to IPv6-only systems, but rather a transition to dual-stack systems, where the main load is carried over the newer protocol. A low-performance of IPv6 compared to IPv4 could strongly delay its adoption. This makes Internet monitoring and performance measurement an important aspect for ensuring that the newer protocol performs at least as good as its older brother. Accordingly, the aim of this research was to further extend the toolbox for Internet performance measurements by development of tests providing reliable statistics on the performance of Teredo tunnelled IPv6 client connections. The first part of this report presents several large Internet benchmarking projects, some of which are still running and in process of development. Then, you can find the initially planned research investigation. The last part of the report focuses on what we have done, and the obstacles, which we encountered, and document details about the development process.

Contents

1 Introduction

Over the past two decades, along with the huge expansion of computer technologies, the field of computer networks and distributed technologies has undergone even faster growth. It is hard to imagine that just 20 years ago network speeds were bound to the painfully slow 'dial-up' with less than 14.4kb/s throughput. Today, high-level broadband networks are available to millions of residential subscribers around the globe. The exponential growth of broadband speeds has increased the pressure on the whole Internet infrastructure. However, there is no clear assessment about the quality and reliability of the network. The lack of common standards and tools for performance measurements often force the users to rely on web-based speed tests that provide one-time and unreliable measurements.

It is clear that broadband penetration is to increase further, and that more and more people will rely on their connectivity for a wide range of activities. Thus, it is necessary to develop measurement systems capable of evaluating and monitoring the home network access performance. The information collected from those platforms will be not only useful for home-users, but also for researchers, policymakers and Internet Service Providers (ISPs). This data will provide accurate information about how the Internet functions, and where the bottlenecks in the whole Internet infrastructure are. Currently there is still a lack of useful information for effectively understanding the state of broadband networks. Thus, it is important to further develop suitable testing methods.

It is crucial, however, that such measurement data reflects network performance over an extended period of time. There exist a large amount of web-based network benchmarking tools, which end-users could use to rate their Internet performance. However, mainly because of their incidental character, those tools are not capable of providing reliable data that could point out the modern challenges of the Internet. The key to properly measure the network performance is establishing continuous test running on the edge of the ISPs' networks, i.e., the users' homes. It is also important that the test suites to be used include all kinds of active network measurements and span multiple ISPs around the globe.

In order to address those problems, several large research projects have been already developed. They are aiming in collecting knowledge about a range of services - from low-level packet switching, to considerations for the networking expectations of individual applications. More information about them will further follow in this report.

However, there exist several possible pitfalls regarding such wide-scale measurements. First, deploying software-based testing suites on multiple host-machines can be fast and inexpensive, but since the host-machines are often turned off, there exists no guarantee for collecting reliable and consistent data. That is why, most of the currently running projects choose to use separate hardware devices, which

considerably increase the deployment costs and time of the projects. Furthermore, all data from the measurements should be collected in a way that will not violate the privacy or security of the users [1]. The techniques to be used should meet the ethical and legal norms (e.g., no passive monitoring on the users' traffic should be done).

Accordingly, the main focus of our research and thesis was also providing useful benchmarking tools, which will provide accurate measurement data from multiple vantage points. The measurements were supposed to contribute for the fast and easy transition to IPv6 connectivity. Specifically, we planned to provide benchmarking tools that would evaluate the performance of the Teredo protocol based on tests performed directly from the end-users' home routers. Teredo is a widely used IPv6 tunnelling protocol. Due to the fact that it can provide IPv6 connectivity even behind a network address translator (NAT), it has received worldwide adoption. However, recent research shows that although it is one of the major contributors to the world IPv6 traffic, it's performance and reliability are questionable. Thus, a long term evaluation of its performance could lead to better understanding of the problems that exist. The research thesis perfectly fitted to the ideas and goals of RIPE Atlas project [2], and was strongly influenced by it. A detailed overview on the Atlas project and the intended research will follow.

2 Statement and Motivation of Research

This section provides general overview on several research projects, some of which are currently running and still under further development. They all aim at providing data about and examining the current state of the global network.

2.1 SamKnows

The SamKnows measuring platform is one of the major measuring systems that is trying to provide a picture of the global broadband performance [3]. The SamKnows project was started by Sam Crawford while he was at university as part of his computer science degree. The project is providing UK broadband availability information since 2003 [4]. Initially, Sam's idea was to develop a testing system that could accurately measure the user's Internet performance, and to provide statistics that are available to individual users (hosts), so that they could compare their actual Internet performance with the speeds that were advertised by their Internet providers. Since the proper way of performing such tests is from users' homes, i.e., at their routers, the tests were embedded into customer-premises-equipment (CPE) devices that incorporate all software tools needed for performing test and forwarding the results to a centralised database. The project called those probing devices "Whiteboxes".

Currently, the Whiteboxes execute a series of broadband tests and collect continuous statistics about the users' broadband connection. The results are later reported to a centralised database. The connection between a Whitebox and the database server is encrypted, allowing secure transmission of the results. The tests are running either against predefined test nodes, or against real applications hosted on the Internet, e.g., web sites, web streams. After each testing cycle the results are transmitted to the back-end infrastructure. In case of network failure, the test results are stored locally, and are later re-submitted to the Data Collection Service, once the connection is restored [5].

The projects utilises hardware from several modern manufacturers for the Whiteboxes. The firmware of the devices is designed and built by the project engineers in collaboration with the manufacturers of hardware. All CPE devices operate a 2.6 series Linux kernel [5]. All Whiteboxes can operate in two modes - either as a router, or as an Ethernet bridge. In the first case all home devices that are connected to the Internet should connect directly to the Whitebox, while in the bridge case only wired devices are connected. This is important since the Whiteboxes are 'sensing' the network usage, so that the tests do not interfere with the users' experience. In the case, when the Whitebox is used as a bridge for wired device, the Whitebox cannot estimate the network usage that accurately and thus it is just passively monitoring the strongest wireless signal for traffic. If a threshold man-

ager detects download or upload traffic above certain limits, the running test will be aborted and delayed.

All communication between the Whitebox and the Data Collection center is encrypted over SSL. Regular updates of the test suite and test schedules are also automatically downloaded from the Data Collection center.

Each Whitebox performs tests against predefined test nodes both over UDP and TCP. As a TCP congestion control algorithm the SamKnows engineers have chosen TCP Reno algorithm with active Selective Acknowledgement. All SamKnows performance tests are written in C and are currently closed-source.

At the time when this thesis report is written, the performance evaluation is based on 10 groups of tests. More information is available in the SamKnows Methodology Whitepaper [5].

1. **Web Browsing**
   Measures the time to fetch a HTML page and all referenced resources, images and stylesheets, including the time spent on DNS resolution. The tested web pages are hosted on existing and popular web sites - e.g., facebook.

2. **Video Streaming**
   The Video Streaming tests are also performed against popular web-sites with streaming services - e.g., YouTube. The test operates over TCP and uses a proprietary client and server side component.

3. **Voice over IP**
   This test is similar to Video Streaming test, however it operates over UDP and measures the performance of bi-directional streaming connections with a predefined server.

4. **Availability Test**
   Availability Test measures the availability of the network. It is continuously running in the background and monitoring for network failures.

5. **UDP Latency and Packet Loss**
   Tests the round trip time (RTT) of small UDP packages that are sent to a test node, which is configured to echo them back to the Whitebox. This test is also permanently running in the background and the Whitebox summarises the average, minimum and maximum results before submitting them to the Data Collection Service.

6. **Data Usage Test**
   The Data Usage Test monitors the inbound and outbound traffic passing through the Whitebox.

7. **Speed Test**
   Tests the upload and download speed of the connection to a target test node.

8. **ICMP Latency and Packet Loss**
   Measures the RTT of the ICMP echo requests that are sent to a test node. The minimum, maximum and average results are again summarised and the data is sent to the Data Collection Server.

9. **ICMP Latency Under Load**
   This test measures the RTT of ICMP echo requests while the system is performing downstream and upstream speed tests.

10. **DNS Resolution**
    Tests the time for DNS lookups of popular web site domains.

11. **Peer-to-peer**
    The peer-to-peer test emulates the BitTorrent transfer of a 10 MB binary file, which is downloaded from various seeding servers around the world. The test registers the average and peak throughput, the number of established connections, the total number of pieces transferred, and the number of TCP connection resets.

12. **FTP transfer**
    The test measures the throughput of performing an FTP transfer of a arbitrary sized file to/from a designated FTP server. A single TCP connection is used.

13. **Email Relaying**
    This test measures how long it takes for an email to be sent via the ISP's public SMTP relay and to reach a third-party (SamKnows-hosted) email server.

As indicated above, the Whiteboxes perform most of the tests against predefined testing nodes (servers). There exist two main types of test nodes: off-net and on-net. The former type of test nodes are built upon the Measurement Lab infrastructure. They are called 'off-net' since they are outside ISPs' networks. The second type of test nodes are hosted within a participating ISP's network. Their purpose is to measure how the network connection is affected outside the particular ISP's network.

Each Whitebox obtains a list of active test nodes. It then determines the closest off-net and on-net nodes, against which it performs the predefined tests. Afterwards the results are sent via an encrypted connection to the back-end infrastructure of SamKnows (Data Collection Service), where the results could be accessed through a Web 2.0 interface.

2.2 M-Lab

Measurement Lab (M-Lab) is another Internet performance benchmarking platform determined to bring better Internet transparency and provide users with useful data about their broadband access links. M-Lab is a community-based, open,

distributed server platform on which different researchers can deploy Internet measurement tools [6, 7].

M-Lab was co-founded by New America Foundation's Open Technology Institute, Planet-Lab consortium, Google Inc. and researchers in order to address several existing problems associated with studying broadband networks and Internet performance from the 'last mile'. Those problems include lack of widely-deployed servers (test nodes) that are available to individual researchers for network testing and evaluation. Moreover, multi-user network testing requires good connectivity and enough server resources to sustain active network measurements. Another problem that M-Lab tries to address is the lack of a platform for data sharing that could be used by the researches to easily compare and share the results of their experiments.

Thus, the goal of the project is deploying a network of servers with sufficient resources for active network testing (infrastructure, connectivity and measurement tools). Those servers will provide passive server-side resources for client-initiated active network measurement of users' Internet connections, and will be used as a back-end infrastructure. Additionally, the data sharing platform will allow individual researchers to easily publish their results and collaborate with one another.

Researchers who would like to deploy measurement tools on the M-Lab platform should adhere to several requirements [8]. Since M-Lab is intended to be an open-platform, the source code both for client-side and server-side tools should be made publicly available and accessible to individuals who would like to develop client-side measurement software. Researchers should also provide an easy and well-documented techniques for data collection from the testing hosts (i.e., end-users). Additionally, no processing or modification of the results is allowed on the test hosts.

The platform is open not only for individual researchers, but also for different companies, academic institutions, or even similar projects, e.g., SamKnows, BISMark. M-Lab was developed as a sub-part of the PlanetLab Consortium (PLC)[9] and makes use of the infrastructure and network that has been already developed by it. PLC provides a distributed software architecture as well as an operations and management framework that is highly relevant for M-Lab. However, the servers of M-Lab are separate and distinct from those of PLC [7], since although PLC supports bigger range of experiments than M-Lab, they are not well-suited for M-Lab goals. There is no guarantee that PLC servers have sufficient resources for active network measurements. On the other hand, M-Lab servers limit the number of tools that can run on each processor. At most 1.5 tools can run on a single core at any time, ensuring that the results are not degraded by the M-Lab servers. Additionally, each test node has at least an eight-core Intel Xeon processor running at 2 or more gigahertz and three or more gigabytes of main memory. Furthermore, each server is connected to one or more ISPs with 1 gigabit per second upstream capacity [6].

All existing tools on the M-Lab platform are performing active network measurement tests. This means that the tools are examining end-to-end performance of the connection between servers and clients and no clients' network traffic is passively monitored.

At the time this report is written M-Lab has announced nine existing tools [10]. Some of them are developed as part of different projects (e.g., BISMark, Neubot) and are using the M-lab platform as a back-end infrastructure to collect data from a broad range of network end-users.

1. **Network Diagnostic Tool (NDT)**
   This tool measures TCP throughput between a client running at the user's host and an M-Lab server. During test execution the tool is also collecting additional meta-data with information about the client OS and kernel-level TCP connection information.

2. **Glasnost**
   This tool emulates a BitTorrent client and measures the performance back to the test node. The basic idea is detecting if the ISP is performing application specific traffic shaping.

3. **Network Path and Application Diagnosis (NPAD)**
   This tool uses TCP to measure end-to-end throughput and information about the switch/router queues along the path. The tool attempts to measure the network problems that affect last-mile broadband performance.

4. **Pathload2**
   The tool measures the residential capacity (available bandwidth or maximum bit rate the client can send before the link gets congested) of the connection between the client and M-Lab test node.

5. **ShaperProbe**
   This is another tool that is designed to detect ISP traffic shaping. The tool is using specifically created UDP packet trains. It also measures the maximum upload/download speeds of the link.

6. **WindRider**
   WindRider is designed for modern smart-phones and mobile devices. It attempts to measure if the mobile network provider is performing application or service specific differentiation, i.e., prioritizing or slowing traffic to certain websites, applications, or content.

7. **SideStream**
   This tool measures TCP performance, and collects Web100 statistics while the user browses the M-Lab web site. It runs on the same M-Lab slice with NPAD.

8. **Neubot**

Neubot is a background daemon (bot) that runs continuously and measures network performance and application-specific throttling. It using various application level protocols (e.g., emulate HTTP or BitTorrent)

9. **BISmark**
   BISmark is actually a separate project that aims to provide a Broadband Internet Service BenchMARK. The project is similar to SamKnows, but it differs in that it is directed in examining how the choice of home modem or router affects the network performance for the end-users. Like the SamKnows project BISMark provides the end-users with a customised gateway, that connects with M-Lab servers and perform series of performance tests (latency, jitter, packet loss, available bandwidth).

2.3 RIPE Atlas

RIPE (**Réseaux IP Européens** - from French **"European IP Networks"**) is an open participation, not-for-profit forum ensuring the administrative and technical coordination that is necessary to enable the operation of the Internet [11]. Since RIPE is an open-participation forum there is no formal membership. Anybody interested can participate through the meetings [12] or mailing lists [13].

On the other hand, **Réseaux IP Européens Network Coordination Centre** (**RIPE NCC**) is a separate entity from RIPE [11]. The RIPE NCC provides administrative support to RIPE, such as the facilitation of RIPE Meetings and providing administrative support to RIPE Working Groups. RIPE NCC is one of the five **Regional Internet Registries** (**RIRs**) in the world, which are responsible for allocation and registration of Internet numbered resources (i.e., IP addresses and AS numbers) that they receive from IANA [14]. RIPE NCC region of service covers Europe, the Middle East, and Central Asia. RIPE NCC also operates K-root - one of the worlds root nameservers [15].

The RIPE NCC has been conducting active measurements from a network of approximately 100 Test Traffic Measurement (TTM) Test Boxes. The majority of them are located inside Internet Service Provider (ISP) infrastructure [16, 17]. The test-boxes perform measurements including one-way delay and traceroute to major high-level DNS servers. TTM aims to help ISPs with monitoring their networks and planning transmission capacity. However, recently RIPE has also seen the necessity for shifting the focus of Internet measurements from the ISPs to the end users. Time has changed and if twenty, or even ten years ago, the limiting bottleneck of the network performance was in the slow dial-up connections and mediocre hardware of the end-users, now the problem has shifted from the last-mile to the middle mile. According to Daniel Karrenberg [16] more comprehensive measurements are necessary to answer many of today's questions about the Internet. Those measurements should cover the whole RIPE region. This means that multiple van-

tage points are necessary that are both located in each Autonomous System in the RIPE region, and in each significant geographical area (i.e., continent). Those measurements should answer questions such as "How is this DNS root server reachable from these Autonomous Systems?" or "How are services in Germany reachable from Ukrainian cities?".

In order to address those challenges, in November 2010 RIPE NCC launched the RIPE Atlas project. RIPE Atlas is a next generation active Internet measurement system. Currently it is in a prototype stage [18], but it generates and collects huge amounts of active measurement data that can be analysed both by researchers and by policymakers. This data can further be used to develop tools for monitoring and troubleshooting that enable network operators to cut operational costs, and contribute to building a better Internet [19]. The goal of the project is to produce a collection of live Internet maps (latency maps, reachability maps, etc.) with great detail. The 'atlas' of maps [20] should be generated by thousands of test nodes (at least 50 000 for meaningful active measurements [2]) that spread around the RIPE NCC region. These test probes will provide real-time data about Internet performance from the edge of the network (i.e., end-users' homes).

The measuring probes that RIPE Atlas is using are tiny devices (nearly three times the size of an ordinary RJ-45 connector), which are powered by an USB connector. Currently, for their measurement probes, the project uses Lantronix XPortPro modules [21]. They have a customised powering board and additional protective housing. The probes connect to the Internet via standard RJ-45 Ethernet interfaces, which speak both 10base-T and 100base-TX Ethernet. Each probe runs a series of tests in the RIPE Atlas system and reports these measurements to the data collection components.

Figure 1 shows a picture of a RIPE Atlas testing probe.

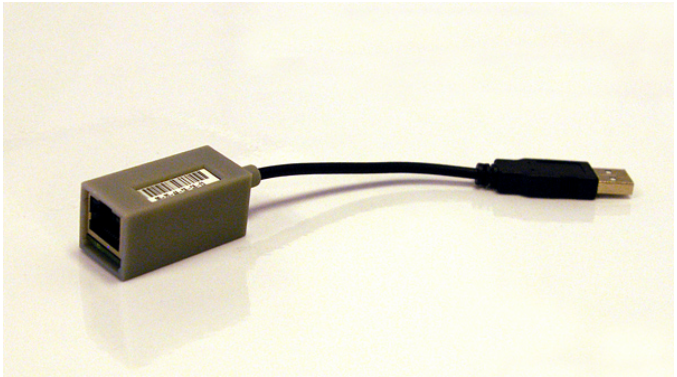

Figure 1: Lantronix XPortPro testing probe [22]

At the end of December 2011, RIPE Labs reported more than 1024 active (continuously measuring) probes [19]. Further, the project aims in deploying another

1K of operational probes by no later than 30 June 2012. On 23. December Atlas project announced the beta version and later in April the final version of their **User Defined Measurements** (UDM) feature [23, 24], which allows users to individually set and schedule network measurements. Each user can hold certain amount of credits, and could spend them on getting some slice of the available resources (testing probes), with which he or she could schedule different types of measurements. Currently the credit earning system is limited only to RIPE Atlas probe hosts and RIPE Atlas Sponsors. They receive credit points automatically based on the time-length during which the probe is online or respectively on the number of sponsored probes. A graphical user interface is used to design the customised tests, and the results can be later reviewed and visualised. After setting, and scheduling UDMs, each user can review which test nodes were randomly selected, as well as the results produced by the measurements. One can also export the data as a CSV file, or visualise it. For example, with the UDM feature one could monitor the reachability of his or her network or of a specific server from dozens of probes all over the world. However, the users are limited to perform tests using only the already deployed testing tools (e.g., traceroute). The system is still in development, and in the future it should enable additional features and greater resources to the users.

Currently, more than one 1500 probes collect data, while performing their built-in tests. The testing suite includes the following measurements [18]:

1. Probe's own network configuration information
2. Current uptime, uptime history and total uptime
3. Ping measurements to RIPE NCC infrastructure and all of the root name servers supporting ICMP echo requests (all except e,g and h).
4. RTT (minimum, maximum and average) and packet loss measurements to the first and second hops after the probe
5. RTT (minimum, maximum and average) and packet loss measurements to a number of predetermined destinations
6. Traceroute measurements to major DNS servers.
7. DNS (anycast) measurements that for each root name server try to determine, which instance of the nameserver the probe ends up connecting to. This allows to build maps about root DNS server "gravity radiuses" which give an insight into the client base for the root DNS operators [25].

In figure 2 you can see an example of a map generated from the results of RTT measurements to the k-root nameserver. Each probe is represented by a triangle and colour of the triangle indicates the RTT: small RTT is represented by green colour while large RTTs are shown by red triables.

Figure 2: Map showing round trip times from Atlas probes to k-root server in Europe [26]

.

2.4 Testing Teredo by RIPE Labs

In April 2011, Geoff Huston published an article [27] on the RIPE Labs website, reporting a recent experiment on the performance of the Teredo tunnelling protocol [28]. The test was performed by attaching a JavaScript object to a web page. This JavaScript, when executed on the client machine, directs the client to perform a number of additional fetch operations of a simple 1x1 image. The measurements concentrate on IPv6 and Teredo performance.

The first thing that the author noted is the relatively small shares of Teredo in the global connectivity. Only about 0.02% of dual-stack object fetches were performed using Teredo. Even more surprising is the result for the IPv6-only object - only about 0.2% of the clients used Teredo tunnels to obtain the object. This seems to be unreasonable, since Teredo has been deployed and by default is enabled on all machines running MS Windows Vista or MS Windows 7. However, as the author points out, although Teredo is 'on' by default on those systems, they will not query the DNS repository for an AAAA record if the local IPv6 interface is link-local or Teredo. This means, that although most Windows systems (even behind NAT boxes) are IPv6-enabled, they will not access an IPv6-only URL, because they will not query the DNS for an IPv6 address of a domain name.

As a side-note the author also points out another peculiar fact about the Windows systems. Although a Windows client might have a native dual-stack connectivity (i.e., no tunnelling is involved), the system will query the DNS server for an AAAA record only if it receives a response about the A record of the corresponding domain. This means that if the query for the A record fails or times-out, the client

systems will not query the DNS for an IPv6 address.

In order to measure the hidden Teredo clients, the experiment bypasses the DNS by directly using an IPv6 address in the URL of the fetched object. The test reveals that about 20% to 30% of the clients are IPv6-capable and can retrieve the object using an IPv6-only URL; also 16% to 26% of all Internet clients are IPv6 capable using Teredo as tunnelling mechanism.

To evaluate the performance of the Teredo protocol, results from the clients using Teredo are compared with the available results for clients using 6to4 [29], and the results from IPv4 object fetching. The test concludes that Teredo performance is poor. The time to perform the fetch operation over a Teredo tunnel is highly variable and on average is up to 3 seconds longer than the time needed for the fetch operation over IPv4. These results are even worse than 6to4, which consistently needs 1.2 seconds more than IPv4.

For both tunnelling protocols this delay consists of two components the time to set up the tunnel and the RTT. One reason for the low-performance of Teredo is that, while 6to4 clients have local 6to4 relays, Teredo clients are usually behind NAT boxes. Thus, Teredo tries to lock the client into the server's choice of Teredo relay, and this way creates a symmetric path. In the case, when a client is situated behind a restricted NAT, more than one RTT is necessary for the tunnel setup. The observations from the test show that for a large portion of the clients the set up of the Teredo tunnel takes from 0.7 s up to 3 s. For about 20% of the Teredo users, the RTT is comparable to the RTT of an IPv4 connection, but for another 30% of the clients the RTT is with 300 ms longer than the RTT of an IPv4 connection.

The experiment also measured the Teredo failure rate by counting the number of connections that are unable of completing the 3-way handshaking and submitting the HTTP GET request. It turns out that Teredo connections are highly unreliable, and about 37% of them do not complete. This percentage includes only failed connections after the successful tunnel establishment, thus the failure rate is actually much higher.

The author also explains that by looking into the 'dark' IPv6 traffic, 18 out of 20 packets were Teredo ICMP connection attempts. Since the operating systems prefer using IPv4 over the tunnelled IPv6, then most of this traffic can be attributed to peer-to-peer connections. Also, he proposes that a big portion of the traffic over IPv6 today is generated by peer-to-peer applications and that probably Teredo is one of the major contributors to the IPv6 traffic.

According to the experiment a major problem with the tunnelling protocol responsible for its poor performance is that it is unable to choose the 'closest' relay to the client. This is because AS routing policy is to pick one exit for each network prefix. Since multiple nodes advertise routing to the same Teredo prefix 2001:0::/32, each AS will pick its best route, which will leave many clients relying on relays that are far away from the actual closest relay.

The author concludes that the tunnelling can never be the solution to IPv6 connectivity. Additionally, NATs and the stateful devices on the Teredo data paths are problematic, and the mere fact that the protocol works is an achievement.

2.5 Deployment and Performance Evaluation of Teredo and ISATAP over Real Test-bed Setup

Deployment and Performance Evaluation of Teredo and ISATAP over Real Test-bed Setup [30] is an experimental setup for comparing the performance of two automatic tunnelling protocols, i.e., Teredo [28] and ISATAP [31]. The test bed used consisted of five nodes: two hosts, one DNS server, two routers, and allows stable network conditions for performance evaluation.

During ISATAP performance evaluation one node (a dual-stack ISATAP node) resides inside an IPv4 network and has to communicate to another dual-stack node, which is residing in an IPv6-enabled part of the network. An ISATAP router is used to forward IPv6 packets between ISATAP hosts on the IPv4 network and IPv6 hosts on the IPv6-enabled part of the network. Figure 3 depicts the test-bed setup: H2 is an ISATAP dual-stack node residing in the IPv4-only subnet. It communicates with H1, which is a dual-stack node on the IPv6-enabled subnet 1, via an ISATAP router (R1). Subnet 2 and 3 were configured as IPv4-only enabled, and subnet 1 is both IPv4 and IPv6 capable. For the Teredo examination the same configuration was used, plus additional Teredo server and NAT server.

Performance was evaluated based on several attributes related to tunnelling and routing: throughput, round trip time, jitter, end-to-end delay.

UDP audio streaming was used for measuring throughput, and the average of four runs was calculated for the test. The conclusion from the evaluation is that ISATAP performs better in respect of throughput. It got an average of 476.88 Kbps throughput in comparison to Teredo, which achieved 454.67 KBps on average. As a reason for the lower performance of Teredo, the authors gave the two-level Teredo encapsulation of the IPv6 packets, i.e., each IPv6 packet is encapsulated in an UDP packet, which is later encapsulated in an IPv4 packet.

End-to-end delay was also calculated on UDP audio streams. Again, Teredo performed worse on the test, and incurred more delay. The additional second encapsulation of the packets, which also allows Teredo clients to work behind NAT boxes, is responsible for the extra overhead. The average measured delay for Teredo was 1.7534 ms, while the average delay for ISATAP was 1.2427 ms.

For the jitter test, the measured Teredo performance was better than the performance of ISATAP. The authors proposed that the more frequent tunnel refresh packets, which ISATAP uses after every 13 data packets, as the reason for the lower performance of the ISATAP protocol. During the refresh of the tunnel, the ISATAP

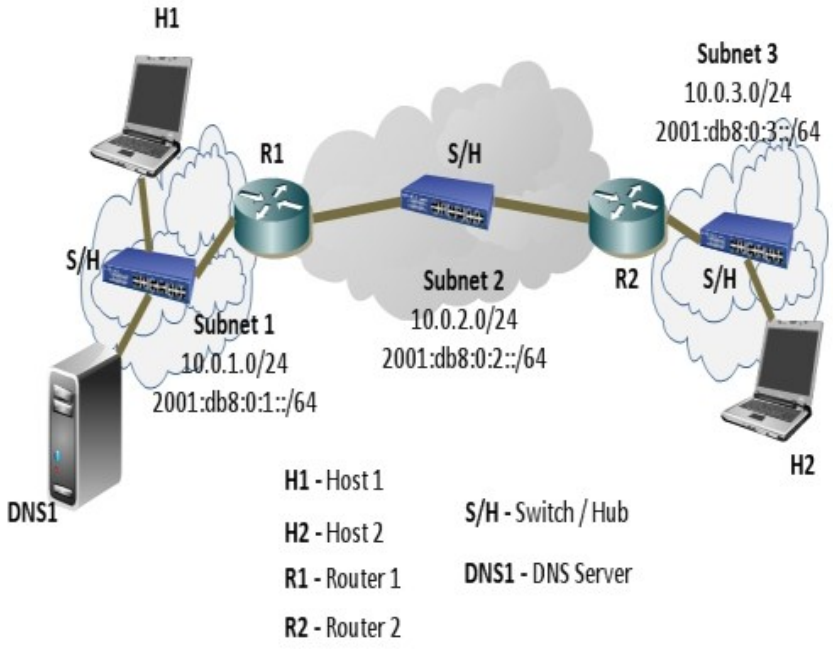

Figure 3: Test Bed setup [Source: [30]]

.

data traffic is halted for a while, incurring more jitter. The average jitter time in milliseconds for Teredo was 0.008 ms, and the average jitter time for ISATAP was 0.029 ms.

The fourth measured attribute was RTT. In this test, ISATAP performed again better than Teredo with 0.5516 ms of RTT on average, compared to 1.0048 ms on average for Teredo.

The authors conclude that comparing ISATAP and Teredo, ISATAP performs better on all of the test parameters, except jitter. However, future work is necessary, because the test bed provides stable network conditions that are insufficient to point out how the two tunnelling protocols are going to perform in the real Internet environment.

3 Initially Planned Investigation and Research Interests

This section presents the problems, which this research wanted to address, and outlines the basic tests that a successful measurement system should accomplish.

The main goal of the research project was to further improve and extend the range of measuring tools that are in use for Internet performance evaluation. In this direction there exist a broad variety of tests that could be developed. The data from such experiments would be valuable for understanding the future state of the Internet. Currently, as we are on the edge of the transition to IPv6, it is important that the whole Internet community (researchers, policymakers, ISPs, etc.) clearly understand the requirements that will ensure the scalability of the global network.

As IANA has already announced that the last /8 block had been assigned to the RIRs, it is of no question that transition to IPv6 cannot be further avoided. Until now it was of no interest to the ISPs or the end-system vendors to invest in earlier IPv6 deployment. Although recommendations about transition techniques existed as early as the beginning of the century [32, 33], it is now actually going to be done in practice.

Due to the fact that the simultaneous deployment of the IPv6 protocol on all network nodes around the globe is a task, which does not scale, transition to IPv6-only connectivity cannot be done promptly. In fact, eveb if IPv6-only connectivity is possible, it will probably take decades to accomplish. The main goal now is to move the Internet load from the IPv4 infrastructure to the IPv6 infrastructure. Thus, mechanisms that support the concurrent use of both protocols, i.e., dual-stack capability, have been developed and they are essential for global adoption of IPv6.

Most modern operating systems have successfully implemented tools for dual stack connectivity. However, there exist several problems with such systems. First, although functions such as getaddrinfo(3) allow DNS queries for both Internet protocol addresses ('A' and 'AAAA'), there still exists the problem that many of the widely used domain-names do not have any IPv6 entries available on the domain name service. Thus, the Internet Society and multiple large Internet companies (e.g., Google, Microsoft, Facebook) announced the 'IPv6 Launch For Real' day on June 6, 2012. By this date all of the participating parties have promised to permanently enable access to all of their services over IPv6 on their main domain-names. For example, one should be able to query www.google.com over IPv6.

This would be a huge leap in IPv6 transition, but does not solve the rest of the obstacles on the way. In the ideal situation a link to a dual-stack server works equally well no matter which Internet protocol is used. However, initially this might often not be the case, and nobody can be sure that choosing the AAAA record from the DNS reply to communicate with a server will always bring the expected network performance. Often the application might have to fall back to IPv4, because of a broken or unreliable IPv6 link. This might cause significant time-out compared to

an IPv4-only client, and can greatly impact the user experience. On the other hand, in the future, after vast IPv6 deployment, the opposite situation is possible, i.e., the client application chooses to communicate over IPv4, but the server is only IPv6 reachable. Thus, a common mechanism that allows applications to quickly determine the functioning path to a dual-stack server is necessary. One such technique drafted by the IETF is the Happy Eyeballs Dual Stack technique [34].

The basic idea of the Happy Eyeballs technique is to provide quick connection establishment to a dual-stack server. The algorithm first attempts to establish connection using an IPv6 address of the remote server. If that connection attempt is not quickly successful, it tries to connect using the older protocol. Additionally, the algorithm avoids thrashing the network, by not (always) making simultaneous connection attempts on both IPv6 and IPv4 (e.g., by caching the results from the attempts for a certain period of time and connecting previously successful addresses). Thus, an implementation of the Happy Eyeballs algorithm on a client application might remember that IPv6 always fails or is unreliable, and make subsequent connection attempts directly using the always successful IPv4 addresses. Consequently, although the user has an IPv6 connection, he or she is not using it due to performance issues. If sufficiently large amount of users faces this problem, it could lead to a delay of IPv6 adoption. In the worst case, as the demand on the Internet increases, due to the fact that the newer Internet protocol performs slightly worse than IPv4, ISPs will be forced to continue invest large amount of resource on IPv4 infrastructure, instead of concentrating on providing better IPv6 connectivity.

For example, consider a user browsing web sites that are hosted on dual-stack servers. Assume the user also has a dual-stack connectivity. For now ignore whether it is pure IPv4/IPv6 connection or one that uses some kind of tunnelling. The web browser of the client implements some sort of Happy Eyeballs technique. However, although the user has IPv6 connectivity almost all of the traffic goes over IPv4. This is because IPv4 connection establishment always outperforms the IPv6 connection establishment or just because the IPv6 link is very unreliable and the browser has learned to use the IPv4 set of addresses. Reasons for such low performance could be that, somewhere down the path, an IPv4-only router just drops IPv6 packets, and considerably scrutinise the performance of the newer Internet protocol on this particular link. Another reason could be that the client uses IPv6 tunnelling (e.g., a Teredo client) that considerably impairs the performance of the IPv6 connection, making it slow.

For those reasons it is of interest to perform IPv6 availability and performance measurements in greater depth. This was also the main target of our research. We intended to develop tools suitable for more detailed test on reachability and availability of dual-stack servers, and how Teredo tunnelling affects the performance of IPv6 links. We were motivated by the idea to get data that could help solving problems with IPv6 that prevent it from a wide usage (e.g., make it less attractive as a functioning path to a Happy Eyeballs algorithm). The reason behind choosing

especially testing Teredo [28] performance was that it is the most widely deployed tunnelling protocol. It is included in all distribution of Microsoft Windows Vista and Microsoft Windows 7, and is also running on many UNIX machines. More specifically, we wanted to evaluate the performance of tunnelled IPv6 client connections. This idea fits well to the goal of RIPE Atlas to measure Internet performance from the end-users' homes. End-user clients usually do not have native IPv6 connectivity and are behind NATs, so the Teredo protocol is usually used to tunnel IPv6 packets.

Suitable tests for measuring the performance of Teredo tunnelled connections would have been RTT, latency, DNS resolutions tests. Additionally, we had the idea to provide tests comparing the time needed for IPv4/IPv6 handshaking and connection establishment to predefined IP addresses/domain-names, and evaluating the reliability and performance of the connection. The data obtained by such tests should have been suitable for evaluation of the impact the tunnelling techniques have over the network performance and user experience.

Thus, the initial step, which we had to perform, was to deploy a Teredo client on the hardware (testing probes). As the probes are running a uClinux distribution [35] with 2.6.30 kernel, relatively high-level programs can be developed for such small embedded devices. However, there exist several differences compared to a standard Linux kernel, which make the compiling and installation of already existing software tools for standard Linux distributions not such an easy task. Details about this will further follow. Nevertheless, we decided to integrating the existing open-source miredo software as a Teredo client on our embedded Linux system. miredo is an Unix daemon program which mostly implements the Teredo protocol: tunnelling IPv6 over UDP through NAT servers. Teredo is an Internet proposed standard and details about it can be found in RFC 4380 [28]. Miredo can provide either client, relay or server functionality.

As part of the research proposal we thought about suitable technique for data collection. Ideally all test tools that were going to be developed were supposed to be fully compatible with RIPE Atlas probes. In this case, it would have been fairly easy to deploy them on a range of test probes around the globe and perform real measurements, from multiple AS-es, using the Atlas infrastructure. Even in future work, it is important that the data collecting mechanisms are backward compatible with Atlas project. However, the data collecting protocol that is used by the Atlas probes is still not publicly available.

4 The Hardware

As already explained, the project was intended to be closely related to the RIPE Atlas project, and backward compatible with the RIPE Labs measurement system. Thus, we used the same hardware as RIPE Labs. We chose to use the same

Lantronix XPort Pro [21] device servers (Figure 4) in the process of the research.

Lantronix developed the XPort and XPort Pro families with the main purpose to connect serial devices with Ethernet (serial to Ethernet translation), but due to its 32-bit architecture and the possibility of running an embedded Linux kernel, the XPort Pro probes allow big flexibility in application development. The XPort Pro devices come in two standard modes. In the first mode, they are equipped with Evolution OS [36], which is a network operating system with a fast and lightweight kernel. However, although Lantronix provide development tools for Evolution OS, this mode was not well suited for the goals we had. Moreover, the RIPE Atlas project also uses the second branch of XPort Pro devices, which are equipped with a customised uClinux kernel. uClinux provides a large range of applications and tools that are standard for GNU/Linux systems. Lantronix XPort Pro devices also come in two hardware modes - with 8MB and 16MB of SDRAM. The probes, which we had for our research had only 8MB of memory, which created a serious problem early in the development process. From the 8192K of available RAM, approximately 1500K were spend on kernel code, 200K on data, and after fully booting the Linux just a little more than 3000K of memory was left for user applications in **kernel + JFFS2 root** mode, or about 2300k in the default **ROMFS root** mode. Also, one should note that the free memory is not just one continuous block, but is fragmented into multiple pieces, which sometimes makes parts of it unusable.

Figure 4: Lantronix XPort Pro device [Source: [37]]

.

The probes used during this research have the following features:

- **Architecture:**

  Freescale ColdFire MCF5208 32-bit microprocessor

Core (System) clock - up to 166.67 MHz

Peripheral and External Bus Clock - up to 83.33 MHz

no MMU

no FPU

Instruction/Data Cache 8KB

Static RAM (SRAM) 16KB

- **On-board memory:**

  RAM: 8MB SDRAM

  Flash: 16MB

- **Network Interface:**

  10Base-T and 100Base-TX Link

  Connector: RJ45

  Protocols: TCP/IP, UDP/IP, ARP, ICMP, SNMPv2, TFTP, FTP, Telnet, DHCP, BOOTP, HTTP, SMTP, SSHv2, SSLv3, PPP, AutoIP, RSS, and SYSLOG

- **Serial Ports:**

  RS232-supporting high-speed serial port with all hardware handshaking signals.

  Software selective baud rate: 300 bps to 921600 bps.

  Note: additional board is used for serial communication.

- **Dimensions:**

  Length: 33.9 mm (1.33 in.)

  Width: 16.25 mm (0.64 in.)

  Height: 13.5 mm (0.53 in.)

  Weight: 9.6 g (0.34 oz)

- **Powering:**

  Input voltage: 3.3 VDC

  IO Max Rating 3.3v

  There is a step-down converter to 1.5 volts for the processor core. All voltages have LC filtering to minimise noises and emissions.

- **Software:**

  Boot loader: two-stage boot loader

  OS: custom uClinux distribution

  Linux Kernel: 2.6.30

Lantronix also provided a Linux Software Development Kit (SDK) [38, 39], which has all necessary tools for building and deploying applications on the Lantronix probes. The SDK also provides a range of implemented software that could also be used on the probes. The device communication is performed either through a serial interface, or a network link (Ethernet). In the process of this research, we used Lantronix Linux SDK version 2.0.0.2 and 2.0.0.3. The SDK also provides an application for remote network configuration, and also tools for flashing the main boot loader (only in Microsoft Windows environment).

Figure 5 shows an additional hardware board [40, 41], which is used for powering and serial communication with the probes. It is also manufactured and supplied by Lantronix. The board provides a serial connector to the device and capability for additional customisation (e.g., building a customised integrated circuit).

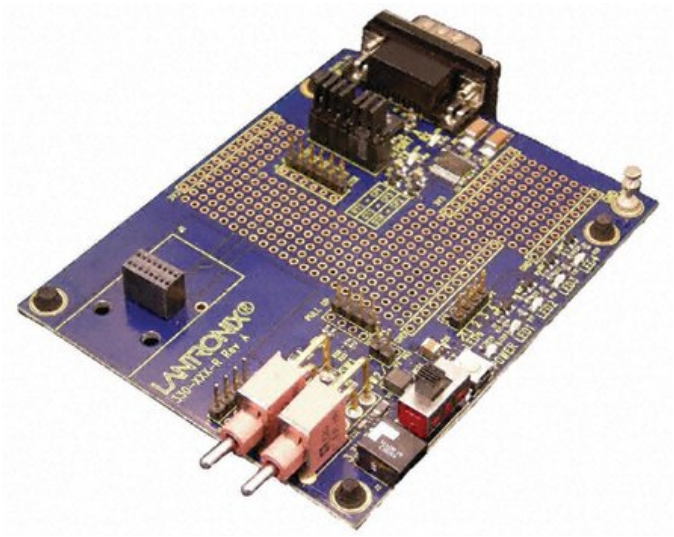

Figure 5: Lantronix Universal Demo Board [42]

.

The probe is running the customised uClinux [35] distribution. The uClinux project provides embedded Linux to systems without a Memory Management Unit (MMU). The Linux kernel used by the device is 2.6.30. uClinux provides a wide range of already available software tools and modules. However, all of them were specifically designed and implemented for uClinux. One reason for that is that there exist several big differences between uClinux and a standard Linux distribution. First, the

system does not implement the conventional fork(2), daemon(2) or brk(2) system calls. This is from where the first pitfall in miredo installation came: re-organising the code in such a way that it does not use fork(), but rather vfork() + execl() system calls. Additionally, the memory of the processes is not protected, allowing each program to corrupt other processes or even the kernel. All applications are statically linked, and no paging is provided, thus all processes should fit into the RAM.

The device boots using a two-stage boot loader. The first stage is the Lantronix boot loader. This boot loader is also available on the XPort Pro modules that are equipped with Lantronix Evolution OS. The second stage is the dBUG boot loader, which is responsible for loading the Linux kernel. It is also used for flashing new Linux images. The dBUG boot loader itself could be flashed using a serial connection and a provided Lantronix application for Windows machines. The Lantronix boot loader, on the other hand, is marked as read-only and cannot be modified or deleted.

The dBUG boot loader provides several configuration options of the probe that allow the setup of parameters for the network or serial connection. Also, tools for boot failure detection and remote network configuration are provided. One can get access to the dBUG console both over the serial interface as well as remotely over the network using the netcon tool, which is provided in the SDK. In the later case, the dBUG netcon option should be enabled. The boot loader configuration allows the users to change the partitioning of the Flash memory. It supports two modes of partitioning - dual Bank and single bank. In the first case, the flash memory is divided into two banks. The embedded Linux boots using one of the banks and keeps the other one unused. The second bank could be later used automatically during firmware upgrades, and provides redundant flash storage space that is useful in recovery scenarios when firmware upgrade fails. When dual bank is disabled, Linux can use the entire flash memory for the kernel and root file system [38]. The default flash layouts of the probes are shown in Figure 6.

The uClinux distribution on the devices supports several types of file systems. ROMFS is the default root filesystem type for the uClinux distribution. It is a read-only uncompressed file system with minimal overhead. In this case the boot loader copies or decompresses the Linux kernel and the ROMFS into RAM and then jumps to its execution. The second major file system type is JFFS2. It is a sophisticated writable log-structured file system that supports wear-levelling. One disadvantage of the JFFS2 root file system is that it is extremely difficult to upgrade the firmware from within the Linux without sufficiently available flash space. uClinux also supports the mounting of directories from a remote computer via NFS. This could be very beneficial during the development process since it eliminates time consuming manual flashing for each change. On the other hand, it is resource expensive. There is also a possibility to mount the root file system as NFS, so that all changes done to the applications do not require a separate flashing of the de-

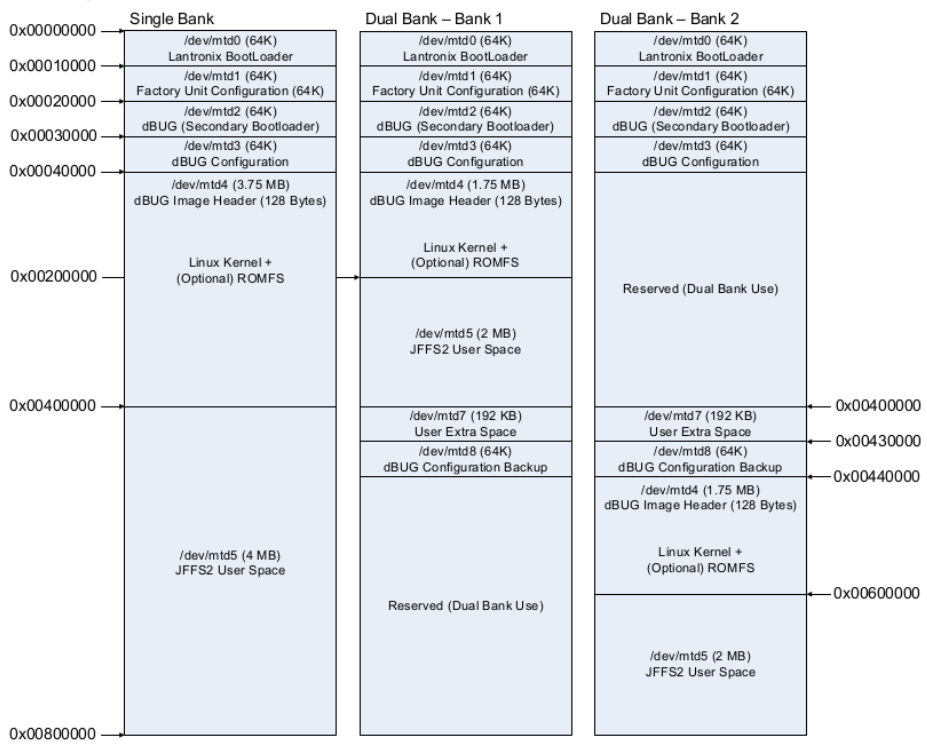

Figure 6: Lantronix XPort Pro Flash Layout [38]

.

vice. This could be extremely helpful for cutting down the deployment time during development stages.

In dBUG the user also has to choose the proper root-file system type depending on the configuration and the Linux image that is in use (e.g., flashed on the probe). When one cross-compiles the Linux image, the SDK produces several image files, which could be flashed on the probe. Depending on the configuration, which should be accomplished, the user/developer should choose the image or images that are suitable for this exact configuration. Three important image files that are produced by the build process: `linux.bin`, which is an image containing just the Linux kernel; `romfs.img`, which is just the ROMFS image; and `image.bin`, which contains both Linux kernel and ROMFS. The first two are used in kernel + JFFS2 root configuration mode, and the third one is used in kernel + ROMFS configuration mode.

The last one is also the default and easiest deployable configuration. Here is an overview how one could make advantage of it, during development:

It is important that dBUG is set properly, so that it knows that the root file system type is romfs. In cases when dBUG tries to load romfs, but the device is for ex-

ample flashed with a JFFS2 root image, the Linux booting will fail. In the default mode (kcl is set to rootfstype=romfs) the device can even boot over teh network using TFTP server. A sample dBUG configuration for this mode (remote boot) is:

```
dBUG> show
    watchdog: on
  silentboot: off
 romfs_flash: off
    bootbank: Single
    safebank: 0
      netcon: on
     tftpsrv: on
        base: 16
        baud: 115200
    autoboot: Autoboot from network
      server: 10.100.2.149
      client: 10.100.3.237
     gateway: 0.0.0.0
     netmask: 0.0.0.0
    filename: image.bin
    filetype: Image
     ethaddr: 00:20:4A:DE:4C:54
         dns: 0.0.0.0
      bootfc: 0
   maxbootfc: off
         kcl: rootfstype=romfs
dBUG>
```

The important lines in this configuration are:

- `autoboot: Autoboot from network` - which sets dBUG to try to load the kernel and root image from the provided tftp server address. One can change the autoboot option by using the command `set autoboot network | stop | flash`
- `server: 10.100.2.149` - which provides the IP address of remote tftp server (usually the host machine that is used for cross-compiling the Linux images). One can change it by `set server <ip-addr>`
- `client: 10.100.3.237` - which is the IP address to be used by the probe while in dBUG. One can change it by `set server <ip-addr>` ; in the case when dBUG should use BOOTP protocol for obtaining an address, just set the ip to 0.0.0.0.
- `gateway` and `netmask` specify the default gateway and the subnet mask of the probe while in dBUG. One can change them by `set gateway <ip-addr>` and `set netmask <ip-addr>` respectively.
- `filename: image.bin` - The SDK automatically generates several files that store the embedded Linux + romfs images.

- `image.bin` is one of the generated images. It includes Linux kernel + romfs. There are two other suitable variations for this configuration. First `imagez.bin`, which is gzip compressed image.bin, and `imageu.bin`, which has compressed Linux kernel and uncompressed romfs.
- `kcl: rootfstype=romfs` - which specifies that the root filesystem type is romfs. This could be changed by `set kcl rootfstype=romfs`.

On each reboot, dBUG will try to access the TFTP server, download `image.bin` and load the Linux kernel from it. This configuration is advantageous for development, since no flashing of the device is necessary after each change in the Linux image. When one builds a new version of the image on his/her host machine (using the SDK), a single reboot of the device is enough to see the changes. On the other hand ROMFS root 'eats up' valuable amount of RAM, since during boot time romfs is copied to the RAM, so that flash area where it resides can be easily overwritten. This allows reliable firmware updates from inside the Linux.

However, if one wants to overwrite and flash the firmware on the device, it is also fairly easy in this configuration mode. While in dBUG, just issue `dnfl image.bin`, and this will download the Linux kernel + romfs into the RAM. dBUG then asks you if you would like to continue and flash the device: `Continue (yes | no) ? yes`. After successfully erasing and writing the appropriate sectors, dBUG reports that the flashing has been completed, and you can boot from the new Linux firmware using the `gfl` command.

The second main root filesystem configuration is **kernel + JFFS2 root**. This one was also used during most of the development, since it frees up RAM, and is used for applications that require bigger amount of memory. The downside of this mode is that it is very hard to implement a reliable firmware upgrade process from within Linux. It is also prone to corruption of important files, since everything is write-enabled. Another disadvantage is that it is not well suited for development, since each time there is a change in an application, the device should be flashed in order to see the effect. Together with compiling, this takes up to 6-7 minutes. On the other hand, for our Teredo client we need to create **/dev/net/tun** for our TUN/TAP node, since this is the default place that it resides. Also, it should be writable, but in romfs the whole **/dev** filesystem sub-tree is read-only (same applies to **/etc**). Thus this is another reason why we chose to use a write-enabled root file system.

In this mode, there are two important configuration setting in dBUG:

```
dBUG> show
    watchdog: on
    ...
         kcl: noinitrd rw rootfstype=jffs2 root=/dev/mtdblock5
dBUG>
```

The first line specifies that the watchdog timer should be turned on. The watchdog timer monitors the CPU and prevents it from blocking by resetting the de-

vice. The rest defines that the root filesystem type is JFFS2, and that the root partition is /dev/mtdblock5, where the location of /dev/mtdblock5 depends on the flash layout. The flash layout is not fixed, and one can make adjustments to it. The partition definitions are available through the SDK and are located in the file: <install directory>/linux/linux-2.6.30/drivers/mtd/maps/m520x.c, where <install directory> is the directory at which the Linux SDK is installed on the host machine.

In the second configuration mode, flashing new firmware takes more effort. However, this configuration frees almost 1MB more RAM than the default one, and is suitable for applications, which need more operating memory like our Teredo client (miredo). The flashing procedure looks like this:

1. Download and flash the Linux kernel image with `dnfl` command, which automatically downloads the specified image to the RAM and then writes it to the flash, starting at address 0x40000.

```
dBUG>
dBUG> dn  fl linux.ib  bin
Address:  0x4001FF80
Downloading Image 'linux.bin' from 10.100.2.149
TFTP transfer completed
Read 1687680 bytes (3297 blocks)
Must erase complete sectors (0x00080000 to 0x0021FFFF)

Continue (yes | no)? yes
.............
Flash Erase complete. 0x1A0000 bytes erased
Program successfully flashed...
```

2. Download the JFFS2 root file system image to the RAM using `dn` command. This will automatically download the specified image to the RAM, starting at address 0x4001FF80 (RAM starting address is 0x40000000). Then use flash write command to write the image to the flash. The command reads 0x400000 bytes, starting at address 0x4001FF80, and writes them to flash memory, starting at 0x00400000.

```
dBUG> dn rootfs.img
Address:  0x4001FF80
Downloading Image 'rootfs.img' from 10.100.2.149
TFTP transfer completed
Read 4194304 bytes (8193 blocks)
dBUG> fl w 0x00400000 0x4001FF80 0x400000
.............................
Flash Write complete. 0x400000 bytes written
```

3. After successful completion of the download you could load the Linux, using the `gfl` command.

```
dBUG> gfl
kernel_info->code_segment_dest = 40020000 SDRAM_ADDRESS = 40000000
Copying 1687552 bytes from 0x80080 to 0x40020000
```

```
Shutting down IO
...
```

Note that both flashing methods are downloading the images over the network, so both dBUG and the host TFTP server should be properly configured. The necessary image files should be available at the TFTP server directory (e.g., /tftpboot). Additionally, the users are not bound to using only those two configuration. One can customise the flash layout by modifying the partition definitions. This way you can create a configuration that would better fits your needs and effectively uses the available hardware resources.

Another important customisation that one might want to do is in the start up scripts. Especially, if you are using **kernel + romfs** configuration mode, both **/etc** and **/dev** directories will be read-only, once you have booted your linux. Thus, one might want to adjust uClinux start-up scripts or just add his or her own scripts. For example, miredo uses a client-hook script in order to add a new network interface, and it should be placed in the **/etc/miredo/** directory together with the **miredo.conf** configuration file. You could access existing scripts or add new scripts using the build environment under the directory:
<install directory>/linux/vendors/Lantronix/XPort_Pro/romfs_extra/
There, you could find several main uClinux start-up scripts:

- **/etc/inittab** - which controls the configuration of the init process and which scripts are called on start-up or shutdown.
- **/etc/init.d/rcS** - which is responsible to mount the probe's file systems and call the start script.
- **/etc/start** - calls rest of the start-up scripts and resets dBUG boot failure counter
- **/etc/netstart** - netstart script is called by /etc/start and it initialises the networking.

Using those scripts you could create automatically loadable scripts on your own, just add some operations or programs that should be executed during probe's start-up (or shutdown).

The Lantronix SDK provides the basic tools that are necessary for development of new application and cross-compiling firmware images. The most important files and directories in the directory tree are:

- <install directory>/linux/ : containing uClinux kernel and user applications
  - <install directory>/linux/images/ : here is where the compiled firmware images are automatically placed
  - <install directory>/linux/linux-2.6.30 : uClinux kernel

  - <install directory>/linux/uClibc : C library directory that is used instead of GNU C Library
  - <install directory>/linux/users : Containing many user applications that could be installed on the device

- Makefile : responsible for generating and assembling the whole images based on the user configuration
- env_m68k-uclinux : environment file

For successfully building uClinux firmware image you should follow these steps:

```
$ cd <install directory>
$ . env_m68k-uclinux
$ make menuconfig
$ make
```

The first two lines are straightforward. The next command **make menuconfig** allows the developer to customise the configuration profile for the next build. **menuconfig** uses ncurses terminal windows, but it is also possible to use graphical windows (via **make xconfig** or **make qconfig**) or just plain terminal by **make config**. While preparing the configuration, developers can adjust the kernel settings (e.g., module support, specifying kernel modules, network option and supported protocols, device drivers). Also, one can choose which applications should be included in the firmware image. uClinux already provides a huge list of predefined applications. Most of them are standard Linux applications, but are developed especially for uClinux. Additionally, one can force library builds (e.g., libpng, libssh), but usually this is not necessary, since Kconfig is smart enough to include them automatically.

A central application is BusyBox [43]. It combines tiny versions of many common UNIX utilities into a single executable. It provides replacements for most of the utilities you usually find in GNU fileutils, shellutils, etc. The utilities have been optimised for space by supporting fewer options; flash and RAM space have also been saved by eliminating the overhead of having multiple binaries. One example on how BusyBox works is that typing the command **ls -a** in the shell, actually transforms to **busybox ls -a**. BusyBox then runs the module responsible for ls. It is possible to add custom applications to BusyBox, since it is extremely modular.

However, in order to create a custom separate application, one should follow several steps. For example, in order to create a simple hello world program, you could do:

1. First create a separate directory for the application sources. Although the path of new application directories is not limited to just one location, usually new source directories should be placed in **<install directory>/linux/user/lantronix**.

   ```
   $ cd <intsall directory>/linux/user/lantronix
   $ mkdir helloworld
   ```

```
$ cd helloworld
- Add the source file. For example:
#include <stdio.h>
int main() {
  puts("Hello World!");
  return 0;
}
```

2. Now create a Makefile. Use a Makefile from another lantronix program as reference: e.g., **<install directory>/linux/user/lantronix/chkstk**.

3. Add your application to <install directory>/linux/user/lantronix/Makefile: add the following line of code there (again search for chkstk for reference): **dir_$(CONFIG_USER_LANTRONIX_HELLOWORLD) += helloworld**.

4. Now edit <install directory>/linux/user/Kconfig and add your hello-world program in the same manner as the rest of the lantronix applications. For example, insert the following code segment:

```
config USER_LANTRONIX_HELLOWORLD
  bool "My Hello World Program"
  select USER_LANTRONIX_APPS
  help
    Here is a description and help that
    will appear in the menuconfig window.
```

5. Run **make menuconfig** from <install directory>. Under the menu for lantronix applications, you could find a check-box for **My Hello World Program** that allows you to add the application to your next firmware build.

In similar manner one can develop higher-level applications with more than one source file. Also, the developers could choose a different approach and even place the application sources in <install directory>/linux/user/ or create a new sub-directory and branch on their own.

5 Development Process

The initial step was to setup the hardware (the testing probes) and to install the provided Lantronix Linux Software Development Kit. The SDK was installed on an Ubuntu 10.04 Linux machine as described in the Linux SDK [38]. The next step was to set up the appropriate TFTP server settings for the host machine, i.e., the computer on which we installed the SDK. Apart from some configuration problems on the TFTP server, everything went as expected and without any problems.

The next crucial step that was mandatory for starting the development process was the integration of a Teredo client to the probe. For our Teredo client we decided to use the open-source miredo project [44]. Miredo includes functional implementations of all components of the Teredo specification: client, relay and server. It runs on the GNU/Linux kernel, FreeBSD, NetBSD and MacOS X. Since miredo implements the Teredo protocol, it can be used to provide IPv6 connectivity to users who are behind NAT devices or broadband routers. Most of these device do not support IPv6 and do not allow forwarding of proto-41 (including 6to4).

For our development, we took the source from the latest available at that moment miredo version: **miredo-1.2.5**. Here is where the problems started. The first problem is that miredo compilation strongly relies on executing a configuration script. This script is responsible to check for the existence of some C libraries or for the existence of some C library functions (e.g., clock_nanosleep(3)). Some of them are mandatory for successful compiling and installation, but for others, as is the case with clock_nanosleep, there is a provided fix-up. In case the distribution on which miredo is installed misses a certain library or function, there are provided fix-up code segments. In fact, the code is actually existing in the individual sources, but its compilation is controlled by multiple macro definitions (e.g., HAVE_CLOCK_NANOSLEEP). The main problem was that there is no way to run the configuration script from within the SDK, since it should be ran from inside the linux distribution, on which miredo is about to be installed. There exists command-line arguments for specifying a different compiler, i.e., cross-compiling, but the config script reports an error even right on the first tests of the C compiler. However, still even if this compiler was not a problem, the config script doesn't have a clear overview on what libraries are actually included on the embedded distribution. The config script is also responsible for preparing the necessary Makefiles that control, the compiling and installation of miredo executables and shared libraries.

On the other hand, right from the beginning it was clear that miredo source wouldn't run out-of-the-box, since both miredo and miredo-server are daemon processes that rely on the usage of the fork(2) system call. Since fork(2) is not available in uClibc, but rather only vfork(2) could be used, there was some work to be done before even trying to compile the sources. The difference between fork(2) and vfork(2) is that the later does not copy the parents page table. Many times a complete copy of

the fork caller's data space is not needed, because after forking an exec(3) is done, using fork(2) incurs a considerable overhead. Thus, uClibc, which is designed for small embedded systems without MMU, does not implement conventional fork(2). After vfork(2), the child process should not modify any other variable than of type pid_t. Immediately after successfully executing vfork(2), the child should use the exec family of system calls to execute a separate process. Otherwise, the behaviour is undefined. Also, if exec(3) fails for some reason, the child should not use the conventional exit(3), but rather directly issue the system call to _exit(2).

The miredo (client/relay) application originally forks several times. One of the forks is for daemonising, another creates a separate concurrent child process, another is for creating a sub-process and yet another is used for executing a separate program - miredo-privproc. Miredo-privproc is a service program that miredo uses to execute a client-hook script. This service program itself forks several times and executes the client-hook script. For the miredo-privproc source, it was fairly safe to exchange the fork(2) calls with vfork(2) calls, since the miredo-privproc child is just used to create a separate process in which the client-hook script is executed using execl(3). For the same reasons, the fork in miredo client/relay process that creates the miredo-privproc process was also safe to be done using the vfork(2).

However, the forks for daemonising the miredo main process and the other one for creating a separate child process were a bit more tricky to fix. Nevertheless, daemonisation could be accomplish with vfork(2) + exec(3) calls. The idea is to fork a child process and execute the same program, but with a command line argument, which tells the process that it is already a daemon. The other fork we managed to eliminate by fusing both processes into one.

First, let's take a look at what miredo sources provide. The source of miredo is done in such a way that the code is reused multiple times. Standardly, there are two main executables that are installed after successful configuring and compiling. The first one is **miredo-server** (Teredo server) and the second one is **miredo**, which is a Teredo client and relay. The mode of the client/relay application depends on the miredo.conf file that holds the miredo configuration. The second main miredo executable is miredo-server, which obviously implements the Teredo server. To lower the executables' sizes the compiling/installation Makefile actually creates two dynamic libraries: libteredo, libtun6. We needed only a Teredo client, so we decided to strip-out the server code and leave only the necessary sources for the client/relay program. Actually, for the client/relay we also needed the miredo-privproc program, which manages client-hook script execution, which on the other hand is responsible for creating a new network interface and hooking it to the kernel. Originally, most users would not really realise that miredo installation, installs more than just the two programs - miredo and miredo-server. Programs as miredo-privproc and miredo-mire (used by the server) are also deployed on the user's system, but as library executables.

Thus, we decided to leave only the necessary code for miredo and mired-privproc.

Moreover, everything is compiled statically, so I decided to remove the dynamic libraries and use only the necessary sources from libteredo and libtun6, and to compile them directly as part of the miredo client program. To further cut the program, we removed the message translation. Although, the uClibc supports locale selection, and probably the translation would have worked with it, it was unnecessary, since the devices are not really expected to have accurate locale selection as well as a correct real-time clock.

So apart from changing the fork system calls, we had to also do the work that is usually performed by the config script. This didn't work out immediately, but happened in the process of compiling the sources for the device. The general miredo installation also takes care for creating and installing the client-hook script and miredo.conf file to /etc/miredo. The client-hook script that is used for Linux distributions is generic, so we just used the same one. The only necessary prerequisite was to have **iproute2** or at least the **ip** command. Although, Linux SDK 2.0.0.3 provides iproute2 as a separate program that could be installed on the probes, it failed to compile. I guess that is going to be fixed in future SDK releases. Thus, we used the **ip** applet that is provided as part of BusyBox application. Additionally, miredo requires the **TUN/TAP** device driver, which fortunately was available through the Linux SDK. All we had to do is to edit the uClinux start-up scripts so that the kernel module is loaded automatically.

Originally, the miredo client/relay program actually has several processes and threads running:

```
    (1)        (2)          (3)
|-miredo---miredo-+-miredo-privproc
|                 `-4*[{miredo}]
```

When the initial process (1) is created, it is used to parse command line arguments, to check that configuration file is existing, and to create a pid-file. Afterwards, it forks and spawns a separate second process (2). The second process is doing the actual tunnelling, while the first one handles incoming signals in a loop. Thus, as I already mentioned, we decided to reduce both processes to just one, and omit one additional fork that would have been harder to implement. Also, the less we have running on the device, the better, since everything should fit to the RAM. Signal handlers could be created and used to handle incoming signals, e.g., by making use of sigaction(2).

Our version of the miredo client/relay process looked like this:

```
    (1)          (2)
|-miredo-+-miredo-privproc
|        `-4*[{miredo}]
```

The running processes were reduced to just two. The first is performing the actual tunnelling, and the second is the service program that is used to execute the client-

hook script. In reality, while executing the client-hook script, an additional third process is created for **/bin/bash**, which on the other hand most likely creates a separate process for the **ip** shell command.

The first thing that miredo does after execution is to read and parse the configuration file. The config file specifies the type of the service that is performed by the program: client, relay, cone, restricted. Since the last three act like Teredo relays, we were interested only in the first option. The configuration also specifies the server address of the Teredo server, which the client is going to use, also the IPv6 prefix and the MTU and the name that should be used for the teredo network interface. There are additional options, but they are not required for successful execution. After reading the configuration file, miredo creates the necessary tunnel abstraction, i.e., spawns miredo-privproc, and creates a new tunnelling network interface.

The final step before actually completing the program start-up is creating four service threads.

- **miredo receiving thread**:
  This thread is responsible for receiving packets over the Teredo interface. When a UDP packet is received over the Teredo service port, the thread checks that it is encoded according to the packet encoding rules. If this is not the case, the packet is discarded. It also checks the type of packets and if it the received packet is router advertisement, it signals the maintenance thread to process it.

- **miredo maintenance thread**:
  This thread is responsible for ensuring that the mappings that the client uses remain valid. It does so by checking the packets that are regularly received from the Teredo server. The thread is actually waiting (with timer) on a condition variable **RECEIVED**, which is signalled by the receiving thread. If signal on the condition is received, it processes the packet (ROUTER ADVERTISEMENT). Also this thread is the key to establishing the Teredo address mappings and service port. The first valid router advertisement received is used for obtaining the needed information. The next valid router advertisements, which are received, are checked so that the mapped address and port correspond to the current Teredo address. If there is a change, the thread updates the current mappings. When the packet processing is complete, the thread signals the receiving thread, which is waiting on condition variable **PROCESSED**. Concurrently, on regular intervals, the maintenance thread is sending router solications.

- **miredo encapsulating thread**:
  The encapsulating thread perform encapsulation and transmission of IPv6 packets in IPv4/UDP packets.

- **miredo garbage collecting thread**:

Teredo receiving and transmitting packets uses a 'list with recent peers'. The list of peers is used to enable the transmission of IPv6 packets by using a 'direct path' for the IPv6 packets. The list of peers could grow over time, and thus a garbage collector is necessary to deleting the least recently used entries

According to the specification a Teredo client will maintain the following variables that reflect the state of the Teredo (miredo) service [28]:

- Teredo connectivity status
- Mapped address and port number associated with the Teredo service port
- Teredo IPv6 prefix associated with the Teredo service port
- Teredo IPv6 address or addresses derived from the prefix
- Link local address
- Date and time of the last interaction with the Teredo server
- Teredo Refresh Interval
- Randomised Refresh Interval
- List of recent Teredo peers

Before being able to send any packet, miredo should perform a start up procedure, which determines the connectivity status, the mapped address and port number. If the start-up qualification procedure is successful, the miredo client may use the service port to transmit and receive IPv6 packets, according to the transmission and reception procedures.

The miredo qualification procedure is performed as collaboration between the maintenance and receiving thread. Both threads are running concurrently. The maintenance thread is sending router solications and at the same time waits for router advertisements. It does not know if it has received a router advertisement, since the receiving thread is responsible for receiving packets. Thus, the maintenance thread basically performs is endlessly looping. On each loop cycle it waits on the condition variable (RECEIVED) for some time, and if the timer expires, it continues with execution. In order to set-up the associated Teredo service point, it should receive a valid router advertisement, so essentially it is waiting for the receiving thread to signal it on the condition variable (RECEIVED).

Using a lot of debug information one can see that the maintenance threads behaves as expected, it regularly sends router solications, waits for advertisement until its timer expires and then continues doing the same thing. After only four timer expirations, it logs that there is no reply from Teredo server, although it still continues to wait for a reply, and if it receives a valid router advertisement it will complete the initial setup. The proper functionality of the maintenance thread could be seen

even by checking the traffic with Wireshark. One can notice that router solications are properly sent from the miredo maintenance thread and have correct structure.

On the other hand the receiving thread behaves more than just strange. Let's see parts of the debug log and explain what actually happens.

```
miredo[83]: Starting...
...
miredo[89]: [RECV_THRD] teredo_recv_thread: Starting...
miredo[89]: [RECV_THRD] teredo_recv_thread: in for before teredo_wait_recv
miredo[89]: [RECV_THRD] teredo_recv_thread: in if before teredo_run_inner
miredo[89]: [RECV_THRD] teredo_recv_thread: calling teredo_run_inner
miredo[89]: [RECV_THRD] teredo_run_inner: Starting...
miredo[89]: [RECV_THRD] teredo_run_inner: tunnel and packet are not null
miredo[89]: [RECV_THRD] teredo_run_inner: Before Checks Packet...
miredo[89]: [RECV_THRD] teredo_run_inner: After Checks Packet...
miredo[89]: [RECV_THRD] teredo_run_inner: Before pthread_rwlock_tryrdlock...
```

The Teredo receive thread starts normally. Similarly to the maintenance thread, its functionality is performed by an endless for loop. The second debug message reports that the thread is already in the for loop and is about to call teredo_wait_recv(), which essentially waits for an incoming packet. The next debug message means that actually a packet has been received successfully, i.e., teredo_wait_recv() returned 0, and now the thread calls teredo_run_inner(), which takes care that the packet is a valid, and according to the author should return immediately otherwise. The function starts normally, tunnel and packet variables are not null, which means that they really exist. The next thing that the function does is to check if the packet is valid. Then, it reports that it is after the check, i.e., the packet is a valid Teredo packet. Then it tries to acquire a read lock on the tunnel state and there the debug output from this thread stops. On first look, this is OK, intuitively if it cannot obtain the lock, the thread should block. Initially, the original code was using the standard blocking function pthread_rwlock_rdlock(3):

```
teredo_state s;
pthread_rwlock_rdlock (&tunnel->state_lock);
s = tunnel->state;
pthread_rwlock_unlock (&tunnel->state_lock);
```

However, for debugging reasons, I changed the function to pthread_rwlock_tryrdlock(3), which according to POSIX man page should in no case ever block: it should return 0 if it successfully acquired the lock, and an error value otherwise that specifies the reason for which the lock cannot be obtained.

```
teredo_state s;
debug("[RECV_THRD] teredo_run_inner: Before pthread_rwlock_tryrdlock... ");
int err = pthread_rwlock_tryrdlock(&tunnel->state_lock);
switch(err) {
// debug messages here
}
debug("[RECV_THRD] teredo_run_inner: After pthread_rwlock_tryrdlock : Result %d\n", err);

if (err)
    pthread_rwlock_rdlock (&tunnel->state_lock);
s = tunnel->state;
pthread_rwlock_unlock (&tunnel->state_lock);
```

```
debug("[RECV_THRD] teredo_run_inner: After pthread_rwlock_rdlock... ");
```

Apparently, at least pthread_rwlock_tryrdlock(3) should return immediately with some result in the err variable. Then, we should see more debug messages, at least until pthread_rwlock_rdlock(3). However, those messages are missing.

Since some 'devil's advocate' might say that this is due to syslog not being flushed, I decided and change the messages, so that miredo directly outputs to standard error (flushes immediately) and additionally added some additional prints, just to show that the thread prints stop right before tryrdlock function. The result is the following:

```
...
miredo[89]: [RECV_THRD] teredo_run_inner: Starting...
miredo[89]: [RECV_THRD] teredo_run_inner: tunnel and packet are not null
miredo[89]: [RECV_THRD] teredo_run_inner: Before Checks Packet...
miredo[89]: [RECV_THRD] teredo_run_inner: After Checks Packet...
miredo[89]: [RECV_THRD] teredo_run_inner: Before pthread_rwlock_tryrdlock...
// here start stderr messages
[RECV_THRD] teredo_run_inner: Before pthread_rwlock_tryrdlock 1...
[RECV_THRD] teredo_run_inner: Before pthread_rwlock_tryrdlock 2...
[RECV_THRD] teredo_run_inner: Before pthread_rwlock_tryrdlock 3...
[RECV_THRD] teredo_run_inner: Calling pthread_rwlock_tryrdlock...
// should see more here, but unfortunatelly nothing
// if pthread_rwlock_tryrdlock successfully returned, this should be available:
//[RECV_THRD] teredo_run_inner: After pthread_rwlock_tryrdlock : Result <some result>"
```

The three 'Before pthread ...' messages are just placed one after another. The messages from the thread stop right before the call to tryrdlock function. Thus, I strongly believe that for some reason, pthread_rwlock_tryrdlock(3) function blocks and does not return. Additionally, I checked that the read-write lock is initialised successfully before being used. I have no explanation about this, it is hard to believe that there might be some bug in uClibc, but still a function that should return immediately no matter whatsoever, does not return. What is even more surprising is that although the receiving thread appears to be blocked, it uses almost 100% of the probe's CPU, as if it is spin-locked. Since the receiving thread signals the maintenance thread on **RECEIVED** shortly after obtaining the state of the tunnel, the maintenance thread never receives such signal, so the Teredo client setup cannot complete successfully.

Although, the program continues to run, it never really completes the necessary start-up qualification.

6 Future Work

Clearly, we did not achieve any of our initially planned research goals. This is due to the fact that the important Teredo client was not installed successfully on the hardware. It is interesting that there is no existing Teredo client software for uClinux. One reason for that might be that miredo is not well suited for this system, although it is supposed to work on modern 2.6.30 Linux kernel.

In my opinion the problem presented above clearly shows that adapting existing software for uClinux systems is not an easy task. In fact, I believe that even if this problem was solved, another one might have emerged afterwards. It is also interesting that nothing much has been changed in miredo functionality that might have caused some sort of a bug. The 'flow' of the program was kept as the original miredo client/relay program.

Clearly, future work would require installing some sort of Teredo clients on the devices. There are several possible solutions to this. First, it would be best if a Teredo client software is developed for uClinux and included in future versions of the Lantronix Software Development Kit. If this does not happen, future developers working on this research should spend some time assessing the usability of miredo for the existing hardware. It might be worth to create a new Teredo client application 'from-the-scratch'. It could still be based on miredo, but could be developed especially for uClibc. This will allow the future developers to actually publish his/her uClinux Teredo client, which could be later used by other users of uClinux. Another possibility is to create a Teredo client as a BusyBox module. Since BusyBox is extremely modular it is constantly expanding, and the project encourages individual developers to provide additional modules. A BusyBox implementation of a Teredo client will again allow its broader usage on such embedded systems. I am not completely sure how the development in BusyBox is organised, but one possible problem might be how to force installation of the necessary dependencies for the Teredo client (e.g., TUN/TAP driver, client-hook script).

Once we have appropriate solution for the Teredo tunnelling on the devices, development should continue as initially planned - creating new IPv6 performance evaluation tools.

7 Conclusions and Acknowledgements

The growth on the demand on the global network in the last decade is immense, but it is of no doubt that it will further increase in future. The Internet is adopting more and more critical functions that put great pressure on the reliability of the network. Thus, the performance monitoring platforms are becoming even more important, providing benchmark data from multiple vantage points, distributed both inside the middle mile of the Internet as well as on the edge of the network.

For a successful and seamless transition to the IPv6 Internet protocol, the whole Internet community agrees that we need a broader and more accurate view on the problems existing in the Internet infrastructure. The results from such performance measuring platforms will be valuable for eliminating the obstacles on the path to a successful IPv6 transition. The data would also give important insight to policy-makers, and enable developments that will scale to the requirements of the future network.

As part of the global community, we also wanted to positively contribute to the goal of making the Internet more reliable and secure. The measurement tools presented in this paper could have successfully been incorporated to a large benchmarking project like RIPE Atlas, and could have provided understanding on the performance issues associated with IPv6 on dual-stack systems, and particularly systems with IPv6 tunnelling.

Naturally, I am disappointed with the 'results', since they are essentially missing. However, I could also find many positive sides of the research - e.g., even small things as understanding why the Internet benchmarking is important.

I want to wish luck to any possible future researchers who decide to work on the same topic. Also, I want to express my acknowledgements to Prof. Dr. Jürgen Schönwälder, who provided me with a lot of feedback during the whole guided research period, and gave me many comments and advices both on my thesis proposal and on this paper. Special thanks to Mr. Anuj Sehgal, who was responsive, and also helped me with constructive ideas and comments.

References

[1] Vinton G. Cerf. Guidelines for Internet Measurement Activities. RFC 1262, October 1991.

[2] RIPE. Active Measurements - A Small Probe . https://labs.ripe.net/Members/dfk/a-small-probe-for-active-measurements. [Online; accessed Jan. 2012].

[3] SamKnows Limited. About SamKnows. http://www.samknows.com/broadband/about. [Online; accessed Jan. 2012].

[4] SamKnows Limited. UK Broadband Availability. http://www.samknows.com/broadband/broadband_availability. [Online; accessed Jan. 2012].

[5] SamKnows Limited. *Test Methodology White Paper*, July 2011. [Online; accessed Jan. 2012].

[6] Constantine Dovrolis, Krishna Gummadi, Aleksandar Kuzmanovic and Sascha D. Meinrath. Measurement Lab: Overview and an Invitation to the Research Community. *Computer Communication Review*, July 2010.

[7] Measurement Lab. M-Lab FAQ. http://www.measurementlab.net/content/faq. [Online; accessed Jan. 2012].

[8] Measurement Lab. M-Lab discussion document. http://www.measurementlab.net/sites/default/files/discussion_document_mlab_10.pdf, Jan. [Online; accessed Jan. 2012].

[9] Planet Lab. http://www.planet-lab.org/. [Online; accessed Jan. 2012].

[10] Measurement Lab. M-Lab tools. http://www.measurementlab.net/measurement-lab-tools. [Online; accessed Jan. 2012].

[11] RIPE. About RIPE. http://www.ripe.net/ripe/about. [Online; accessed Jan. 2012].

[12] RIPE. RIPE Meetings. http://www.ripe.net/ripe/meetings/ripe-meetings. [Online; accessed Jan. 2012].

[13] RIPE. RIPE Mailing-lists information. http://www.ripe.net/ripe/mail/mailing-lists/about.html. [Online; accessed Jan. 2012].

[14] IANA. Number Resources. http://www.iana.org/numbers. [Online; accessed Jan. 2012].

[15] Root Servers Information. http://root-servers.org/. [Online; accessed Jan. 2012].

[16] RIPE Labs. Active Measurements Need More Vantage Points. https://labs.ripe.net/Members/dfk/active-measurements-need-more-vantage-points. [Online; accessed Jan. 2012].

[17] RIPE Labs. Test Traffic Measurements FAQ. http://www.ripe.net/data-tools/projects/faqs/test-traffic-measurements. [Online; accessed Jan. 2012].

[18] RIPE Atlas. FAQ. https://atlas.ripe.net/faq. [Online; accessed Jan. 2012].

[19] RIPE Atlas. 1024 Active Probes. https://labs.ripe.net/Members/becha/ripe-atlas-has-reasons-to-celebrate. [Online; accessed Jan. 2012].

[20] RIPE Atlas. Visualisation Maps. http://atlas.ripe.net/atlas/maps_index.html. [Online; accessed Jan. 2012].

[21] Lantronix Inc. XPort Pro Product Information. http://www.lantronix.com/device-networking/embedded-device-servers/xport-pro.html. [Online; accessed Jan. 2012].

[22] RIPE Atlas. Probe Picture Source. https://labs.ripe.net/Members/dfk/active_measurements/probe2.jpg. [Online; accessed Jan. 2012].

[23] RIPE Atlas. User Defined Measurements Beta. https://labs.ripe.net/Members/becha/happy-beta-testers-of-atlas-udm. [Online; accessed Jan. 2012].

[24] RIPE Atlas. User Defined Measurements. https://atlas.ripe.net/doc/udm. [Online; accessed May. 2012].

[25] RIPE Atlas. DNS Measurements. https://labs.ripe.net/Members/kistel/dns-measurements-with-ripe-atlas-data. [Online; accessed Jan. 2012].

[26] RIPE Atlas. Map Picture Source. https://labs.ripe.net/Members/kistel/images/RTTmapkroot.png. [Online; accessed Jan. 2012].

[27] Geoff Huston. Testing teredo. https://labs.ripe.net/Members/gih/testing-teredo, April 2011.

[28] C. Huitema. Teredo: Tunneling IPv6 over UDP through Network Address Translations (NATs). RFC 4380 (Proposed Standard), February 2006.

[29] B. Carpenter and K. Moore. Connection of ipv6 domains via ipv4 clouds. RFC 3056 (Proposed Standard), February 2001.

[30] Mohammad Aazam, Syed Atif Hussain Shah, Imran Khan, and Amir Qayyum. Deployment and performance evaluation of teredo and isatap over real test-bed setup. In *Proceedings of the International Conference on Management of Emergent Digital EcoSystems*, MEDES '10, pages 229–233, New York, NY, USA, 2010. ACM.

[31] F. Templin, T. Gleeson, and D. Thaler. Intra-Site Automatic Tunnel Addressing Protocol (ISATAP). RFC 5214 (Informational), March 2008.

[32] Robert E. Gilligan and Erik Nordmark. Transition Mechanisms for IPv6 Hosts and Routers. RFC 2893, August 2000. Obsoleted by: RFC 4213.

[33] Robert E. Gilligan and Erik Nordmark. Basic IPv6 Transition Mechanisms. RFC 4213, October 2005.

[34] Andrew Yourtchenko Dan Wing. Happy Eyeballs: Success with Dual-Stack Hosts. Internet-Draft draft-ietf-v6ops-happy-eyeballs-07, IETF Secretariat, December 2011.

[35] uClinux Homepage. `http://www.uclinux.org./`. [Online; accessed Jan. 2012].

[36] Lantronix Inc. XPort Pro with Evolution OS User Guide. `http://www.lantronix.com/support/downloads/?p=XPORTPRO`. [Online; accessed Jan. 2012].

[37] Lantronix XPort Pro Picture Source. `http://www.engineersonline.nl/wosimages/nieuws_15283_16555_item_original.jpg`. [Online; accessed May. 2012].

[38] Lantronix Inc. XPort Pro with Linux User Guide. `http://www.lantronix.com/support/downloads/?p=XPORTPRO`. [Online; accessed Jan. 2012].

[39] Lantronix Inc. XPort Pro Integration Guide. `http://www.lantronix.com/support/downloads/?p=XPORTPRO`. [Online; accessed Jan. 2012].

[40] Lantronix Inc. XPort Pro Universal Demo Board. `http://www.lantronix.com/device-networking/embedded-device-servers/xport-evaluation-kit.html`. [Online; accessed Jan. 2012].

[41] Lantronix Inc. XPort Pro Universal Demo Board - User Guide. http://www.lantronix.com/pdf/XPort_Universal-Demo-Board_UG.pdf. [Online; accessed Jan. 2012].

[42] Lantronix Demo Board Picture Source. http://www.embedded.sphinxcomputer.de/images/xpdemoboard.gif. [Online; accessed Jan. 2012].

[43] BusyBox About. http://www.busybox.net/about.html. [Online; accessed May. 2012].

[44] Miredo Homepage. http://www.remlab.net/miredo/. [Online; accessed May. 2012].