Exploratory Benchmarking of CurveCP

MSc Research Project Report

Title of original proposal:
"SSH/SCP Emulation as Meaningful Performance Benchmark for CurveCP"

Thorben Krüger
benthor@os3.nl

July 27, 2011

Abstract

The recently introduced CurveCP protocol for enabling secure packet-level communication has enjoyed little scientific attention to date. Here, I present my own research pertaining to (previously unanswered) questions about computational overhead, achievable bandwidth/latencies and practical decongestion behavior of the the alpha-test implementation of the protocol. Pitted against SSH and HTTPS on one particular machine, I found the software to achieve significantly lower payload-throughput while not fully utilizing available CPU time. Latency-penalties of CurveCP in one setup were found to be slightly higher than those imposed by SSH, their impact diminishing with the length of the travelled route. On an otherwise saturated link limited to 10MBit/s, the CurveCP implementation failed to achieve any significant bandwidth, while increasing the limit to 100MBit/s led to acceptable performance. Session persistence despite IP-address-roaming was found to work as advertised (although only informally tested). While these results probably do not fully quantify the properties of either CurveCP in general or its only current implementation in particular, they might still provide baseline and incentive for future research.

Contents

1 Introduction

In this document, I present my research about benchmarking the novel (and still obscure) CurveCP protocol, which aims to be a simple, secure alternative to TCP. I primarily investigated the protocol in terms of its potential use-case for providing transport for a simple remote shell/remote copy tool. In this role, the only available alpha-test CurveCP implementation[1] had to assert itself against popular TCP-based solutions, such as SSH[2]. Various performance and usability questions were investigated.

1.1 Introduction to CurveCP

CurveCP is a cryptographically enhanced protocol on top of UDP rather than TCP. As a consequence, it implements its own transmission control, thereby aiming to be a secure alternative to TCP in most of the latter's traditional use-cases.[7]. The core concepts of the protocol were first outlined publicly in a talk[1][3] held in December 2010 by Daniel J. Bernstein. A reference implementation was first made available in February 2011 along with a website which serves as the only protocol documentation so far. On this website, a number of claims are also made about various features of the CurveCP protocol. In the following paragraphs, I will summarize the most important of these claimed features.

1.1.1 Security Features

In order to always implicitly provide at least some degree of security, the protocol enforces server authentication in all cases. Client authentication remains optional. In case the server's correct long-term public key is already known to the client, man-in-the-middle attacks are rendered impossible. By consistent employment of nonces, replay attacks become ineffective.[9] In addition to this, CurveCP offers active and passive forward secrecy, both in terms of network traffic and in terms of (client) key material, making it impossible to retrieve information from recorded packets of a session that has since been invalidated.[4]

1.1.2 Availability Features

CurveCP is invulnerable to RST-style attacks (as can be conducted against TCP sessions) by keeping all transmission information (such as the equivalent of "sequence numbers") inside encrypted containers. Illegal or forged packets are simply discarded and not acted upon. This makes a CurveCP session hard to disrupt, short of blunt DoS. Attacks are further hindered by purposefully including randomness into the packet transmission schedule, which blurs patterns and makes traffic prediction (often required for targeted attacks) harder. Furthermore, un-authenticated client requests are designed/specified to exceed the standard server reply by a certain length, otherwise the request is discarded. This way, the server can not be abused to amplify the DoS resources of a (packet-forging) attacker against a third target. The server also avoids allocating memory before a handshake with a client has successfully completed, thereby making "SYN flooding"-inspired attacks largely ineffective. Finally, efforts are made to keep even the worst-case CPU load small, making it hard for a packet-forger to DoS-attack a CurveCP server without resorting to blunt attacks based on sheer bandwidth.[3]

[1] In correspondence, the CurveCP author points out that much of my research could be rendered obsolete by a more optimized beta-test implementation.[11]

[2] As one reviewer points out: Please note that I often use the term "SSH" synonymous with "TLS" on these pages.

[3] which I happened to attend

1.1.3 Efficiency

A CurveCP packet has a certain degree of overhead over a plain TCP packet. When used for short connections (i.e., transfer of a small amount of data) however, CurveCP uses significantly less traffic than either HTTPS or SSH. This has not been quantified yet.[6]

1.1.4 Decongestion Features

CurveCP tries to minimize packet-loss *and* significant increases in latencies, a behavior which in turn is claimed to decongest routers in response to the phenomenon of bufferbloat. In order to detect circular/regular patterns in resource consumption (as effected on the network by other, competing schedulers), the CurveCP scheduler (a.k.a "Chicago") keeps long-term congestion statistics. Thereby, even while running alongside an aggressive competing (e.g., common TCP) scheduler, CurveCP is claimed to be able to achieve a tolerable share of bandwidth. In addition, congestion-induced packet loss can be distinguished from similar effects due to a bad (i.e., lossy, e.g., wireless) connection.[5]

1.1.5 Addressing

Traditionally, a remote service is identified by the IP address of its host plus the port, which the service "listens" on. CurveCP addressing departs from this method of identifying a remote service. While the "traditional" components still constitute a part of the CurveCP address, two additional mandatory fields are required to identify a server address. A 16-byte extension can be arbitrarily chosen to be acted upon by the server-gateway. This facilitates deployment of a wide variety of services behind a single IP address and port, since the gateway can route incoming packets according to their extension. The final part of the CurveCP server address is its DNS-style domain name. Any packets sent to the server that do not contain its configured domain name get silently dropped. The domain name thereby plays its part in anti-aliasing the server from its IP address. Since a peer is never identified by its IP address alone, the latter can change in the course of a session without disruption. This facilitates "connection roaming" without affecting established sessions.[2]

A CurveCP client can know multiple alternative addresses of a service. In this case, it will try to initiate communication with each, establishing a session with the address that replies quickest. Since by design, non-completed handshakes are not an issue[3], this behavior is not seen as "bad practice". As a result, a client might not even notice a non-available host.[2]

1.2 Previous Research

Due to the considerable novelty of this protocol (both in terms of age and chosen approach), it understandably only enjoyed limited scientific attention as of time of this writing. Additionally, CurveCP has yet to be formally specified in full. To my knowledge, this work constitutes the very first instance of independent scientific scrutiny of the protocol.

1.3 Originally Proposed Research

At the beginning of my research, I sketched a rough outline of my approach and scope. Below, I give a short account of what I had initially proposed. The various departures from this plan will become evident later.

1.3.1 Approach

The intended approach was to develop a tool on top of CurveCP to provide SSH/SCP-like functionality. The result was then to provide the basis for conducting comparative performance measurements between SSH/SCP and the CurveCP test-implementation.

I envisioned a simple solution based on a gutted rsh/rcp implementation as frontend and a similarly gutted CurveCP-(HTTP)-proxy setup as a backend. This was deemed to be the least time-consuming approach. One added benefit would have been that the original rsh/rcp tools could have provided an upper boundary for performance if used in a scenario without encryption.

The gathered data was to be processed and presented in comparative plots, answering the various posed questions in an easily visible and straight-forward way.

1.3.2 Scope

Within the scope of this project, I intended to create a reproducible setup for benchmarking the alpha-test CurveCP implementation, pertaining to various performance questions (as posed below). From the beginning, it was not deemed to be feasible to test the protocol under a greater variety of conditions and setups, given the amount of time allocatable to the project. Any results obtained were therefore expected be of limited general validity only. Still, due diligence was to be paid towards answering all of the proposed questions as thoroughly as time allowed.

Any software developed conducting these experiments was to be published, regardless of general applicability.

---

2 Questions, Methods & Results

Unless otherwise noted, all experiments on the following pages were conducted using a host with an `Intel Pentium D CPU 3.00GHz` and 2 GB of RAM on the server side and a low end machine with an `Intel Atom 330 CPU 1.60GHz` and 2 GB of RAM on the client side.

I conduct all my research using the only available CurveCP implementation (as part of version 20110221 of the Networking and Cryptographic Library (NaCl)[10]), which (as of time of this writing) currently is in alpha-testing stage only. [4]

2.1 CurveCP as Remote Shell/Remote Copy Service

As mentioned above, one intended “side-effect” of my research was the creation of a CurveCP-enabled remote shell/copy utility, which could then serve as a test-bed to benchmark the protocol against equivalent, more traditional SSH-based services and to verify some of the claimed features of the protocol.

Contrary to what I initially proposed, I did not set out to reproduce the “standard” CurveCP setup for the test implementation but satisfied myself with getting the most basic test-setup (described in the README file of the reference implementation) to work. I based my further work on this, being more comfortable with such a bottom-up approach.

My initial efforts to adapt rsh/rcp for my needs were hampered by the strongly deprecated status these tools “enjoy”, making it difficult to even obtain their source code. Contrary to my initial roadmap, I therefore set out to reimplement the required functionality myself (although the rsh source code which I eventually obtained proved helpful, too).

By nature of the CurveCP reference implementation, the application-layer client or server program can be entirely network-agnostic, enabling the programmer to concentrate on the core-functionality.

I succeeded in proto-typing a pseudo TTY service, which can be used as a server-application for a CurveCP transport, or a variety of other different backends, thereby providing a remote PTTY. (Further information plus source code can be found at `https://github.com/benthor/remotty`.) However, while this tool could certainly be used in place of SSH for remote-shell work (e.g., even curses-based applications behave as expected), it did not turn out to be very useful for *benchmarking* purposes. This was primarily due to my ambition to provide full, interactive PTTY access over CurveCP, while neglecting non-interactive use-cases, which ultimately would have been more useful for automated tests. Unfortunately, my PTTY solution does not easily lend itself to remote-copy functionality and my measurements pertaining to this feature had to be conducted using different means.

2.2 Latencies

2.2.1 Question

- How do latencies between CurveCP and TCP-based sessions compare?

[4] It has been pointed out that on the following pages, I sometimes fail to properly distinguish between aforementioned *implementation* (which I perform the measurements for) and the *protocol* per se. I have tried to rectify the most blatant of these slip-ups but may not have caught all.

2.2.2 Methods

ICMP ECHO exchanges (i.e., using ping) are commonly regarded as a method of choice to measure latencies. Even though ICMP is a very low-level protocol on the network/internet layer, I decided to bend it to my needs to measure message latencies over higher-level CurveCP or TCP-based sessions. To this end, I developed a tool to create a reasonably transport-agnostic layer 3 tunnel between two hosts. My tool simply forwards its standard input to the specified tunnel device and echoes everything received on the latter back to standard output. (Hence its name: stdxput2dev. It can be found at https://github.com/benthor/stdxput2dev) Using a named pipe, a simple SSH-based VPN-tunnel[5] can then be established like so:

```
mkfifo sshfifo
cat sshfifo |
python /path/to/stdxput2dev.py ssh0 10.0.0.1 10.0.0.2 |
ssh root@remote.host "python /path/to/stdxput2dev.py ssh0 10.0.0.2 10.0.0.1" > sshfifo
```

This creates a tunnel device (ssh0) on both the local and the remote host, with all layer 3 traffic between both machines being translated to and from ssh-packets.

Similar to the SSH-based tunnel, plain TCP (using e.g., netcat or socat) can also be used as transport.

Sending ICMP ECHO_REQUESTs over tunnels established with various transport backends in this fashion can now give a reasonable way to compare protocol latencies.

To rule out transient effects due to possible message fragmentation on the transport level, some tests pertaining to this issue were conducted. The length of ICMP payload data was varied and the amount and sizes of the respective transport-packets was monitored.

2.2.3 Setup

Three different tunnels were created between the local and remote host, using TCP/SSH/CurveCP as transport respectively. These were all established in parallel, before initiation of measurements. Measurements were both conducted over LAN as well as WAN connections (2 and 7 routing hops respectively). A latency baseline was established in all cases using raw (i.e., non-tunneled) ping.

To rule out transient errors which might skew the measurement, sets of 20 raw pings and pings over the different transports were conducted 10 times over, with every combination of payload lengths. Switching between the different transports was always closely interleaved to minimize the possible impact of transient effects in favor of any particular method.

The client machine for the WAN measurement was different from the default: Intel Core2 Duo P8600 CPU 2.40GHz and 2GB of RAM.

[5] OpenSSH also has a built-in facility to create tunnel devices. However, for ease of comparison, I exclusively use my own tool to create tunnel devices.

2.2.4 Results & Discussion

For SSH, each tunneled ICMP REQUEST/RESPONSE-pair corresponds to 3 exchanged SSH packets up to a payload length of 1378 bytes. When attempting to transmit between 1379 and 1472 bytes as payload, the number of exchanged packets increases to 7 for each ping-pair, since the added overhead of an SSH packet exceeds the MTU of 1500 and the ICMP packet has to be split over multiple messages.

With plain TCP, 3 packets per ICMP-message-pair get exchanged like with SSH, although "close-to-MTU" behavior using large data payloads could not be examined due to unspecified issues (probably pertaining to `netcat` internals) limiting sizes for tunneled packets to a maximum of 1024 bytes.

The very same issue was present for CurveCP which has an upper payload-bound of 1024 bytes in order for the resulting packet to still fit inside the MTUs of the most common data-link protocols, despite some added overhead[8]. CurveCP distributes each ICMP-message-pair over 4 exchanged UDP messages, regardless of chosen payload length. Both on LAN and WAN, latencies over the TCP tunnel exhibit a substantial standard deviation, with a mean significantly higher than the median. When plotting the latencies for a LAN tunnel, a pronounced and regular lower bound for TCP latencies becomes apparent, with a high number of seemingly random outliers above this baseline (see Figure 3). In order to not skew the following results too much against this buggy TCP-behavior, I conduct my analysis against the respective *median* latencies of the different transports instead of the mean.

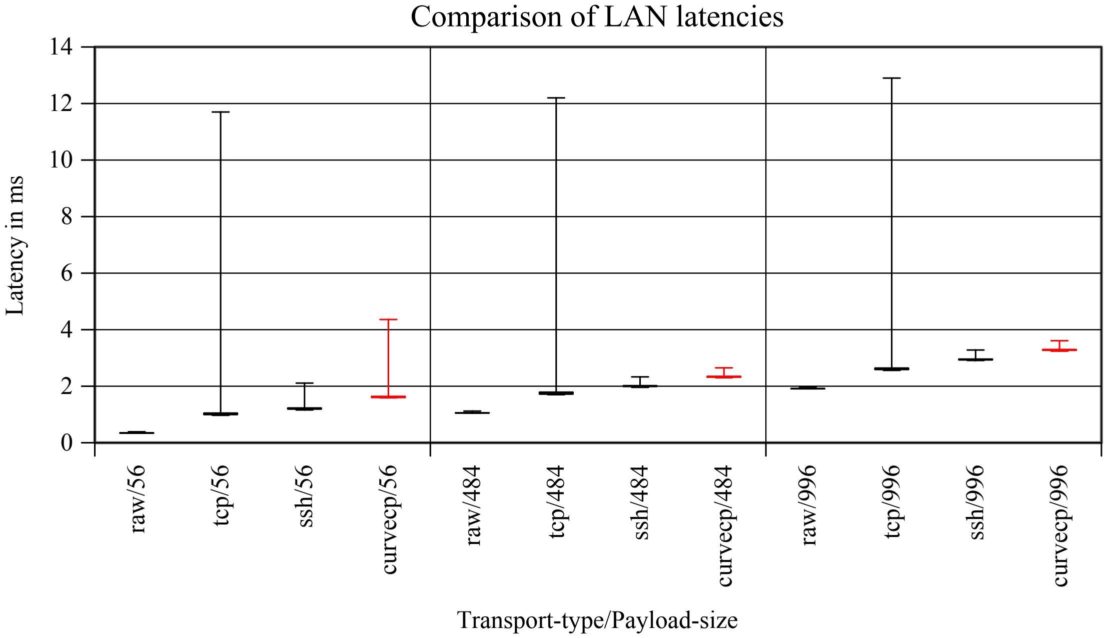

Figure 1: Latencies on a LAN, grouped by payload sizes

The alpha-test CurveCP implementation performs with a consistent, slight, but statistically significant ($p < 0.05$) latency penalty when compared to SSH or raw TCP[6].

In the LAN setup and regardless of payload, the tested CurveCP implementation suffers a median 0.62 ms ($\sigma = 0.04$) latency penalty when compared to TCP. Against SSH, this penalty is reduced to a median 0.36 ms ($\sigma = 0.04$). (Raw

[6] This was previously erroneously reported to not be the case. Subsequent and more detailed statistical analysis of the collected data resulted in the above-mentioned findings.

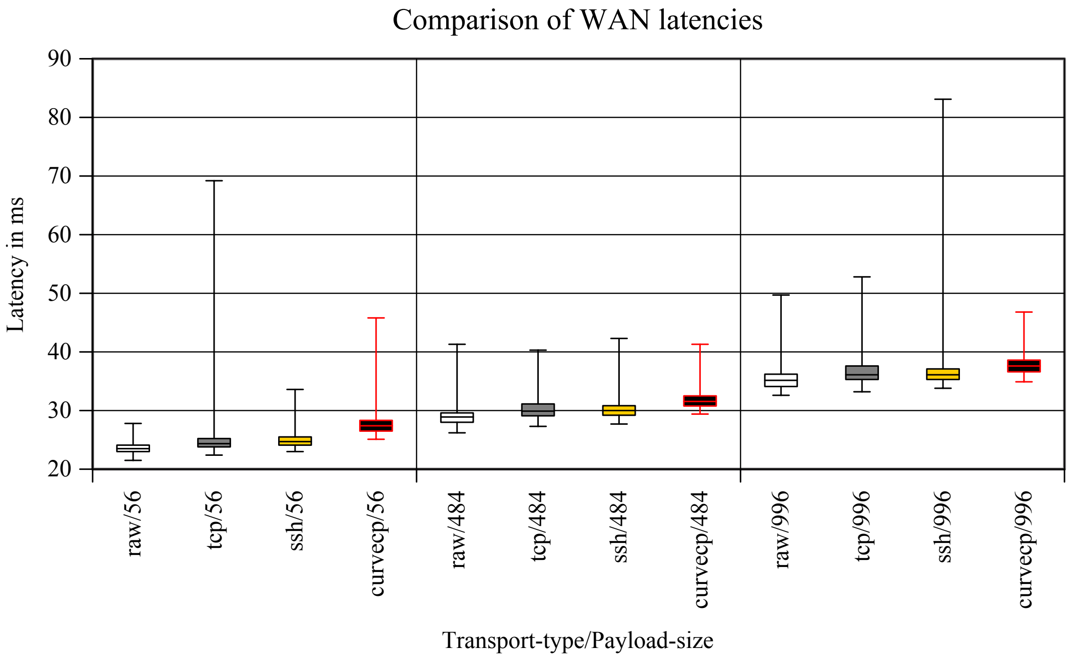

Figure 2: Latencies on a WAN, grouped by payload sizes

ping is on average 1.3 ms (σ = 0.06) faster.) However, for all transports, the *overall* latency increases with larger payloads, diminishing the impact of this static penalty in comparison. While a minimal payload takes 1.6 times longer for CurveCP than TCP, the biggest possible packet size takes less than 1.3 times as long. Compared to SSH, the overall impact decreases from factor 1.3 to factor 1.1 for small and large payloads respectively. Compare Figure 1.

Over a 7 hop Internet connection, the latency penalties see an increase. Interestingly, an increased amount of payload reduces the relative latency penalties of CurveCP compared to other transports. For TCP and SSH, the penalty factor again decreases for larger payloads, from 1.13 or 1.10 respectively to 1.04 in both cases. See Figure 2.

Comparing the LAN and WAN measurements, the latency penalty for CurveCP protocol seems to depend on the number of hops rather than cryptographic overhead (in which case the penalty should be more or less constant). This may be due to the fact that CurveCP relies on UDP instead of TCP for transport, and the former possibly being handled differently in internet routers.

Overall, for the current implementation, the CurveCP latency penalty, while present, seems negligibly small.

2.3 CPU usage/bandwidth

2.3.1 Question

- **In an ideal setting (no competing traffic), how does CPU usage relate to used bandwidth?**

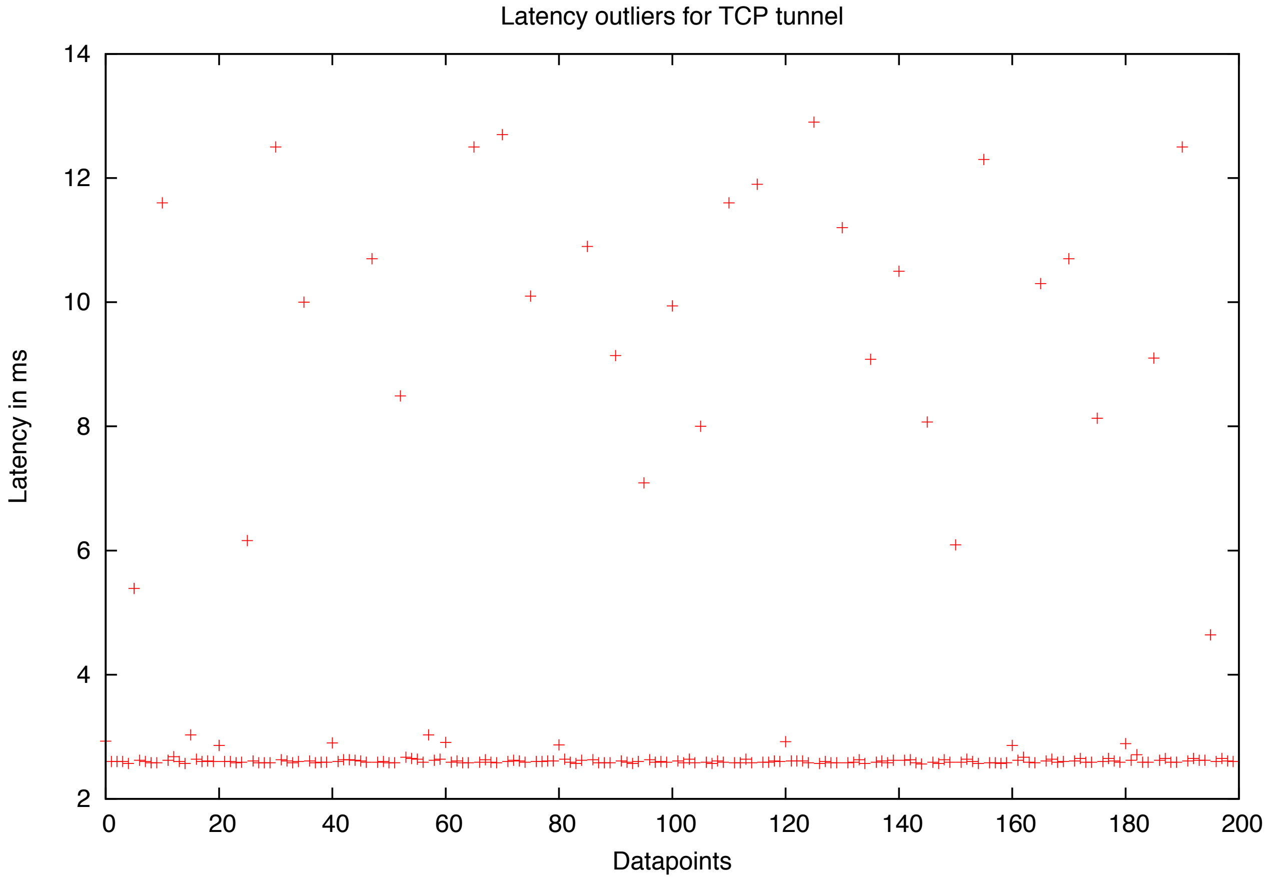

Figure 3: Scattered latency outliers for TCP transport on a LAN.

2.3.2 Methods

In order to put bandwidth and CPU usage into context, I went beyond the posed question and conducted the measurements not only for CurveCP but for SSH and HTTPS/plain TCP (see below) as well. Since resource measurement is tied to additional resource-consumption as well, a comparative approach can be used to make meaningful remarks about *relative* performance.

Two approaches to relate CPU usage to bandwidth were taken.

Firstly, both client and server were launched on the same machine, with the intention to obtain results not bound or influenced by possible (side-)effects of (physical) networking. The server/client processes were thus purposefully CPU-bound. In this setup, the standard POSIX `time` utility could be used to obtain CPU utilization data of the tested client/server components.

The second setup was designed to explicitly include a physical network acting as a bottleneck, distributing server and client over different hosts. However, additional changes needed to be made to the setup: The `time` utility failed to provide any meaningful statistics for the server process in this setup, which was suspected to be due to the `forking` behavior of the tested CurveCP implementation. Upon every new connection in this setup, the server `forks` off a new process, which will then take up the actual work while not being tracked by `time`. Due to this, direct comparability to the results obtained in the first approach had to be forgone. Instead, a custom stream copy tool[7] (in the spirit of `dd`) was created, which would periodically report (global) CPU-usage statistics and I/O-bandwidth. Unfortunately (by virtue of being programmed in a high-level language) this tool introduces some significant CPU overhead (pagefaults/cache-misses), rendering it useless for providing meaningful statistics in CPU-bound settings. For this reason, it could not be meanigfully applied in the non-networked setup. Similarly, no useful measurements could be performed on a 100MBit connection due to a too high impact of the measurement on available CPU-time.

In addition to that, a stream copy tool does not easily lend itself to HTTP file-serving setups, which is why HTTPS was dropped from consideration in favor for a plain (i.e., unencrypted) `netcat`-based file transfer.

2.3.3 Setup

Because only a multicore machine was available to benchmark this (and the classic POSIX tools tend to report conflicting information once several cores are available), all but one CPU cores were disabled.

For the second setup, network device throughput was artificially limited to 10 MBit to perform the measurements (using `ethtool`).

The default configuration for both HTTPS and SSH were not modified. The results for HTTPS below have been obtained with `tlsgatling`[8] (v. 0.12) acting as server and `wget` (v. 1.12) acting as client, both using `libssl` (v. 0.9.8). Traffic analysis showed that the negotiated cipher suite (via TLS 1.0) was 256-bit AES with Cipher Block Chaining and SHA-1 as hash function. No HTTP compression was negotiated.

SSH by default negotiated 128-bit AES with Cipher Block Chaining and MD5 as hash function. For both the server and the client, the SSH implementation and version used was OpenSSH 5.1. Again, no compression was negotiated.

In both cases, the handshakes employed RSA public key cryptography.

Lacking a true, CurveCP-based remote copy tool, a 100MB file containing random data was simply streamed to the client as soon as the session was established. Rather than simply using SCP, a similar approach was used for SSH. A non-interactive SSH session was to be established, simply issuing a `cat` command on the server for the same 100MB file, piping it to the client.

2.3.4 Results

In case of deployment on the same host (no bandwidth limitations), the tested **CurveCP** client and server implementations on average shared available CPU time in the following manner (based on 3 trials):

- CPU (server): 30.3% ($\sigma = 3.2$)

[7] An implementation of which will be released to github shortly.

[8] A high performance single-threaded HTTP server implementation found here: `http://www.fefe.de/gatling/`

- CPU (client): 51.3% ($\sigma = 1.5$)
- *CPU (both):* 81.6% ($\sigma = 3.0$)

- *bandwidth:* $7.0MB/s$ ($\sigma = 0.4$)

In a comparable setting (based on 3 trials), **SSH** client and server shared available CPU time like so:

- CPU (server): 56.3% ($\sigma = 0.6$)
- CPU (client): 42.0% ($\sigma = 0.0$)
- *CPU (both):* 98.3% ($\sigma = 0.6$)

- *bandwidth:* $31.3MB/s$ ($\sigma = 0.2$)

Finally, 4 **HTTPS**-based trials gave the following results:

- CPU (server): 44.0% ($\sigma = 0.0$)
- CPU (client): 49.3% ($\sigma = 1.0$)
- *CPU (both):* 93.3% ($\sigma = 0.6$)

- *bandwidth:* $24.9MB/s$ ($\sigma = 0.1$)

The second, explicitly *networked* setup produced the results shown in Figure 4.

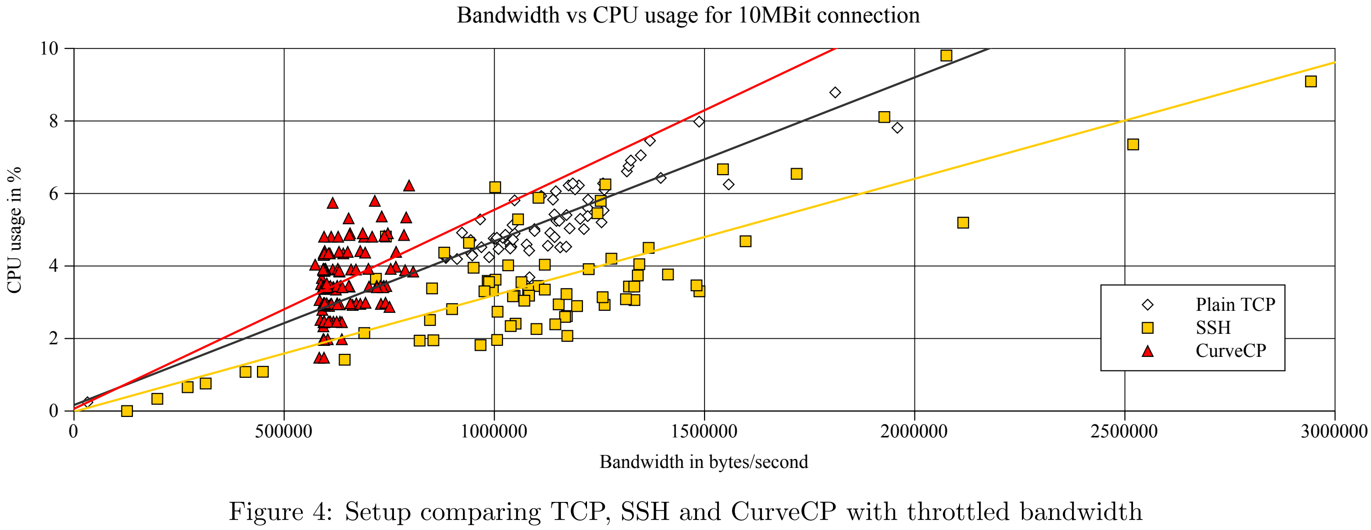

Figure 4: Setup comparing TCP, SSH and CurveCP with throttled bandwidth

2.3.5 Discussion

While the CurveCP implementation does not consume all available CPU time, the achieved bandwidth in this setup is still significantly less than what SSH or HTTPS offer in a CPU-bound mode.

In the case of a deliberately non-CPU-bound setting (i.e., using a throttled network device) and server-side-only resource-consumption measurements, this contrast does not appear to be so drastic any more, although CurveCP still performs worst.

SSH achieved more than quadruple bandwidth compared to CurveCP, HTTPS more than three times as much. At the same time however, both the HTTPS and the SSH server processes used a significantly higher share of CPU time than the CurveCP server.

2.4 Packet Scheduling

2.4.1 Questions

- How much bandwidth can a fresh CurveCP session achieve alongside a (saturating) TCP file transfer (on the server/client side)?
- How much bandwidth can a fresh TCP session achieve alongside a (saturating) CurveCP file transfer (on the server/client side)?

2.4.2 Methods & Setup

Since I have found no way to throttle/limit the loopback device, these measurements were done on a LAN, with the server NIC throttled to 10 and 100MBit/s respectively. The saturating TCP connection was established between the same two hosts that would also conduct the CurveCP session.

2.4.3 Results

As can be observed in Figure 5, there is irregular and sparse throughput when the NIC is throttled to 10MBit/s. Packet sniffing alongside the measurement reveals that there is a slow but comparably constant trickle of CurveCP UDP *packets* between server and client, while a successful transmission of an actual *payload* message happens much more rarely.

For a 100 MBit/s limit (not plotted), the situation is different however. Here, CurveCP achieves a regular share of bandwidth of 111KiB/s ($\sigma = 26.5$).

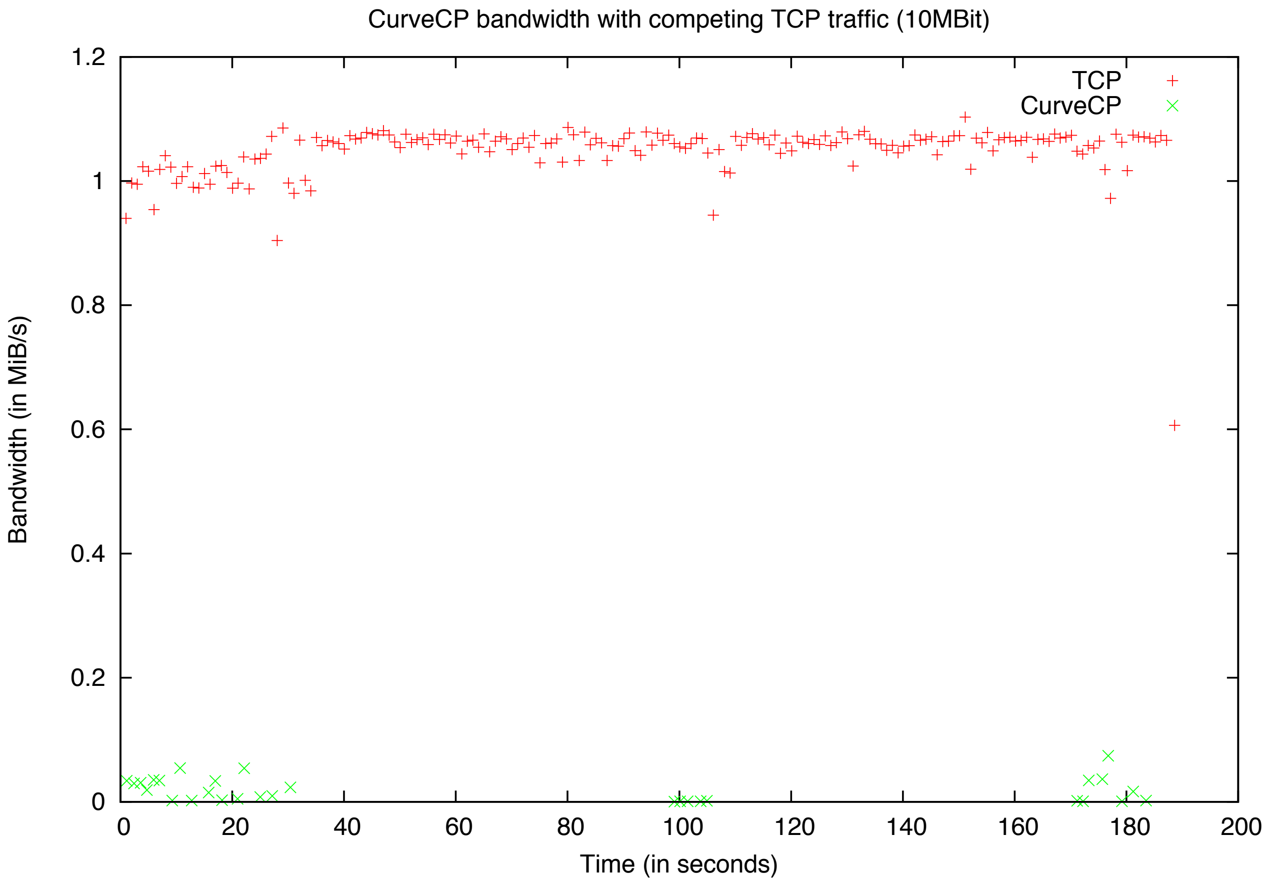

Figure 5: A NIC limited to 10 MBit/s, with the CurveCP session initiated shortly *after* the TCP connection saturating the link was made

It should be noted, that bandwidth for the tested CurveCP implementation degrades to the described degrees, *no matter* whether communication started before or after the TCP session was initiated. (I.e., these results apply to both questions posed above.) In both cases, TCP session succeeds in claiming next to all available bandwidth for itself.

2.4.4 Discussion

As evident in Figure 5, the CurveCP implementation fails to receive any tolerable/reliable share of the bandwidth with a NIC limited to 10MBit/s. Occasionally, a set of messages does get through to the client-side, although between these "exceptions", there are lengthy (usually somewhere between 90 and 230 seconds) gaps of absolute silence. The constant trickle of UDP packets, which takes place despite this puzzling. Due to the packet payloads (including any data pertaining to transmission control) being encrypted and thus completely opaque, I can only speculate about this issue. The simplest explanation which presents itself is that both sides might attempt to send messages which are simply not received at the other end.

Compared to the abysmal performance for a 10MBit/s limit, the situation for higher achievable bandwidths is more tolerable. This might indicate problems in the CurveCP decongestion algorithm/scheduler, which warrants further investigation.

2.5 Handshakes

2.5.1 Question

- How do SSH and CurveCP handshakes compare in terms of exchanged packets (and their sizes)?

2.5.2 Results/Discussion

The CurveCP handshake is *very* simple, well defined and only consits of three exchanged packets (2 sent by the client and 1 sent by the server). The Client Hello packet (i.e., the first) has only fixed fields of certain sizes. The data-part of this UDP packet is always 224 bytes long (plus a typical 20 byte IPv4 and 8 byte UDP header). The Server Cookie (i.e., the reply) packet always contains 200 bytes of data fields. With a third packet (sent again by the client), the session is then fully established. This "Client Initiate packet" completes the handshake and will also contain the first encrypted CurveCP (payload) message. The non-payload "handshake-overhead" in this packet is fixed at 544 bytes.

All in all, the client sends 768 bytes as part of the session handshake, while the single Server Cookie packet of 200 bytes is sufficient to establish an encrypted and authenticated session. This trivially equals a sum of 968 transferred bytes (i.e., less than a kilobyte) required to facilitate the handshake. Also adding the typical 8 byte UDP headers increases the handshake byte count to 992, or 1052 if typical IP headers are considered as well.

In contrast, an SSH session already requires the exchange of three packets for the classic 3-way TCP handshake. These packets typically don't contain any data payload, putting their sizes between 20 and 60 bytes (plus a 20 byte IPv4 header). (In my experiments, the SYN and SYN ACK packets always had 40 byte headers, while the final ACK contented itself with 32 bytes. The TCP headers for all packets following this handshake were fixed at a length of 32 bytes)

The overhead of the remainder of the handshake will largely depend on the selected/preferred cryptographic features. Therefore, much of the following is based on observation (i.e., traffic analysis) and conjecture.

Once the TCP session is established, variable-length SSH-version strings are exchanged between server and client. In my experiments, these were at most 32 bytes long, although I could not find any specification about an upper limit. As a practical lower limit, I assume 9 bytes, which would for instance be required to represent the string `SSH-2.0\r\n`. Each side ACK's the other one's version packet, resulting in a best/worst case of 73/128 bytes sent on each side.

Both peers subsequently exchange SSH2_MSG_KEXINIT packets, containing a collection of variable-length strings denoting all supported cryptographical protocols/features. Typically, this packet is rather big, although one could certainly set out to explicitly disable most protocols/features. In practice, I have seen between 800 and 1200 byte-long packets at this stage.

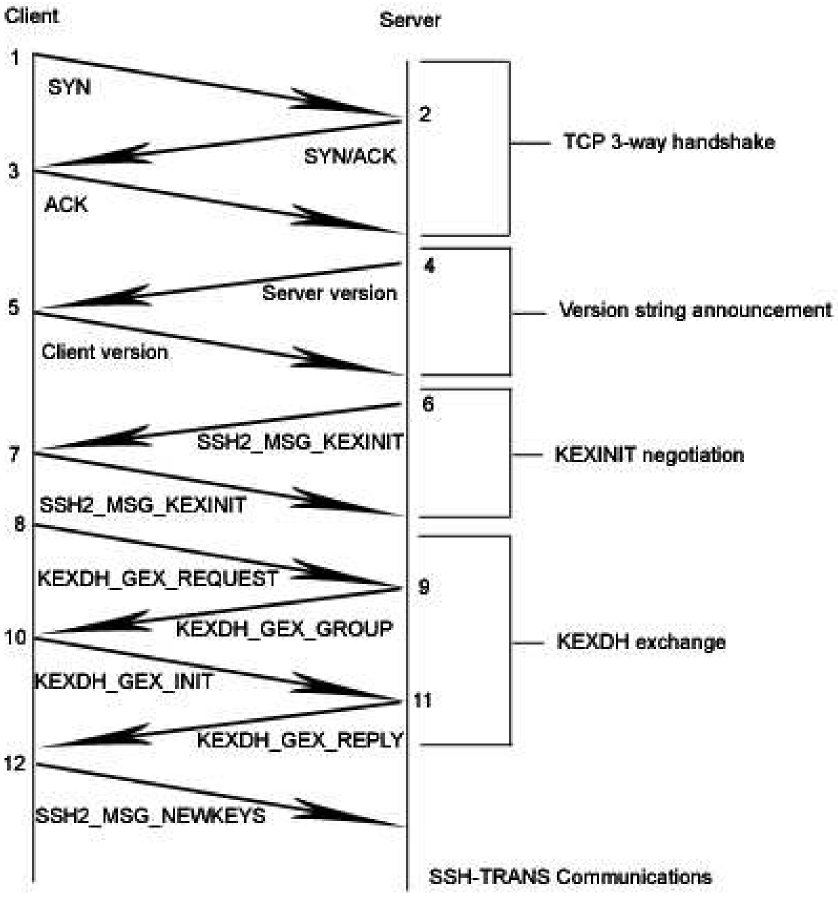

Figure 6: Typical packet exchange[13] at the beginning of an SSH-handshake

At this point, the SSH handshake has *already* required more than double the amount of packets used for a CurveCP handshake. The number of transferred bytes can be considered to be about equal, but contrary to CurveCP, the SSH handshake is far from being complete at this point:

In order to generate a shared secret, some form of Diffie-Hellmann group exchange typically takes place next. Client and server exchange 4 messages before a shared secret is determined. The client begins by sending a (small, e.g., 52 bytes long) packet describing constraints for the Diffie-Hellman exchange. Within these constraints, the server then chooses Diffie-Hellman modulus (P) and base (G), which are sent back to the client in a somewhat larger packet (observed to be ca 200 bytes long). Based on these values, the client computes e based on P and G before sending the result in another packet (also about 200 bytes long) to the server. The server computes f, which is the final piece of information needed by the client to derive the shared secret. The packet containing f also contains the server's public key and a signature (a cryptographic hash over all constituents of the exchanges so far). This packet has been observed to be about 732 bytes long. Upon processing this, the client will send a short packet (44 bytes long), indicating that encryption can now go into effect.[13] (Also compare Figure 6)

At this stage, only the server is authenticated, but client authentication can now take place encrypted. I refrain from going into further detail at this point, since it does not add much to the verdict of the SSH-handshake being much more resource intensive, requiring many times over the number of packets used for the CurveCP handshake and a significantly higher amount of exchanged data.

2.6 Overhead

2.6.1 Question

- How large is the general (packet) overhead of CurveCP over TCP (theory vs practice)?

2.6.2 Methods

Traffic dumps from the above-mentioned tunneled-ping experiments could be reused to investigate the practical validity of the claims made about the CurveCP protocol

2.6.3 Results & Discussion

CurveCP overhead has to be distinguished on two different levels. Firstly, the cryptographic layer on top of UDP introduces some overhead, as has been mentioned above. However, the encrypted payload of these UDP packets is not raw data but data structures commonly referred to as CurveCP *messages* (as opposed to *packets*). These datastructures provide fields for information that facilitates reliability features known from TCP, in addition to containing the raw data.

Since the CurveCP handshake has been discussed above, I will concentrate on the overhead of CurveCP session that has already been established. It should be noted that the CurveCP protocol deliberately breaks with the traditional connection symmetry (known e.g., from TCP).

A common “Server Message packet” always contains exactly 64 bytes of (cryptographic) overhead plus a CurveCP message that is an integer multiple of 16 bytes in length (max 1088 bytes). A common “Client Message packet” always contains exactly 96 bytes of (cryptographic) overhead plus a CurveCP message that is an integer multiple of 16 bytes in length (max 1088 bytes).

A common CurveCP (payload) message per specification has 48 bytes worth of data fields for purposes of transmission control. A maximum of 1024 bytes of raw data can be transferred in a message. If the message length is not a multiple of 16 bytes, it is zero-padded. Zero padding is not restricted between 0-15 bytes, additional blocks of 16 bytes can be arbitrarily added within the maximum message size of 1088 bytes. This can hamper packet analysis attacks somewhat.

With 8 byte UDP headers, 20 byte IPv4 headers and 16 bytes zero padding, a Server Message packet would have

156 bytes overhead. For a full-length payload of 1024 bytes, a whole packet would have a maximum size of 1180 bytes (on the IP layer). The additional overhead would constitute about 13.2 % of the whole packet. Conversely, a Client Message packet would have 188 bytes overhead, which would make up 15.5 % of the whole packet and (with 1024 bytes of payload) add up to a maximum size of 1212 bytes (on the IP layer).

All the above findings have been verified to the exact number of bytes by examining captured traffic from the above-described latency experiments with tunneling large ICMP payloads.

In comparison, a plain TCP packet with a 1024 byte payload, a standard header size of 32 bytes and 20 bytes IPv4 header should add up to 1076 bytes on IP level. This was found to be true in case of the above experiments as well. This overhead only makes up 4.8 % of the entire packet, about 3 times less than the CurveCP overhead for the same payload.

The question for theoretical packet overhead for SSH was deliberately not asked here, since the protocol is quite complex. However, analyzing traffic dumps could in and of itself give a useful hallmark. Tunneling a 1024 byte ICMP packet over an SSH-based channel takes 1124 bytes on the IP level. The 100 bytes difference divide into 20 bytes for the IP header and 32 for the TCP header, leaving 48 bytes of additional encryption overhead. The encryption overhead is 4.3 % of the entire packet, with the TCP and IP overheads adding up to 8.9 %

2.7 Addressing

2.7.1 Questions

- How feasible is the possibility of using a single IP address and port for multiple servers?
- How feasible are persistent CurveCP sessions when switching client IPs?
- In failure situations, how does the failover happen and what are the parameters involved?

2.7.2 Results & Discussion

Since the reference implementation does not provide a gateway to parse CurveCP-extensions, running multiple servers behind a single IP address and UDP port could not be put to the test. However, the theory of this use-case does not have any obvious inadequacies, since the approach is not too dissimilar to well-understood, traditional NAT.

Similarly, the reference client does not allow to specify more than a single IPv4 address/UDP port, so it can not be used to verify the claims about rapid failover. Again however, the theoretical/technical approach is well argued and easily understood, so its viability is still very likely.

Some practical evidence was found in favor of the claims about persistent CurveCP sessions in case of client (IP) address roaming. However, not much efford was spent in vetting this for different use-cases, which would have required some significant work in order to create a suitable infrastructure for detailed tests. Perhaps of note: while address roaming was found to work, CurveCP sessions get automatically invalidated if idle for more than 2 minutes, due to specific design decisions of the CurveCP protocol. This is seen (and argued) to be a (security) feature rather than a short-coming.

3 Bonus: Effectiveness of Decongestion

Although not originally planned, I had a limited opportunity to deploy CurveCP on a research system designed by some fellow students to measure the effects of bufferbloat. The system consisted of a daisy-chain of machines/routers, configured with artificial bottlenecks (via rate-limited NICs) along the way. A high-bandwidth network session communicating between both distant ends would quickly fill up packet buffers, resulting in increased round-trip times for packets. Round-trip times are then measured in a traceroute-inspired fashion to provide a per-link measure. For further details, see [12].

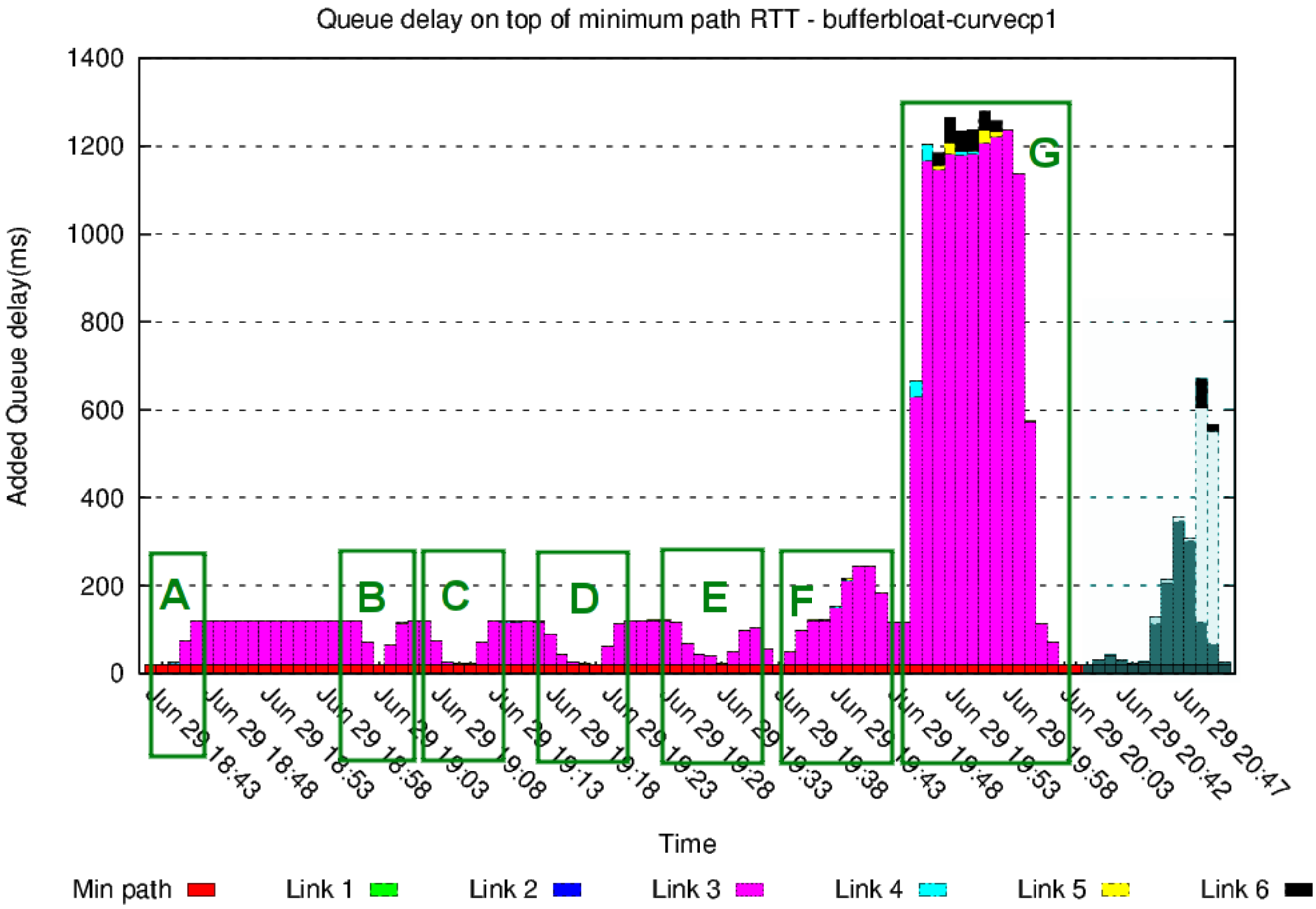

Figure 7: Queue delays over time/per link with different events marked: **A**: initiation of one CurveCP session, queue delay tops out at 120ms; **B**: queue delay spontaneously drops down to baseline before rising again; **C**: session started in *A* is terminated, a second session is started after a grace period, again tops out at 120ms; **D**: Previous session is stopped, a new one is initiated after queue delay normalization; **E**: queue delay spontaneously drops to baseline, rises again before session is terminated; **F**: A new session is initiated, after topping out at 120ms, *a second* session is established concurrently, queue delay doubles, tops out at 240ms before second session is terminated; **G**: 9 sessions are established concurrent to the existing one, queue delay blows up to 1200ms before experiment termination; the gray remainder of the plot was not part of these experiments

On this infrastructure, I was only able to conduct a single (but extensive) test run, during which I performed various informal tests. Figure 7 shows the plot generated by this test run, with various events marked and explained.

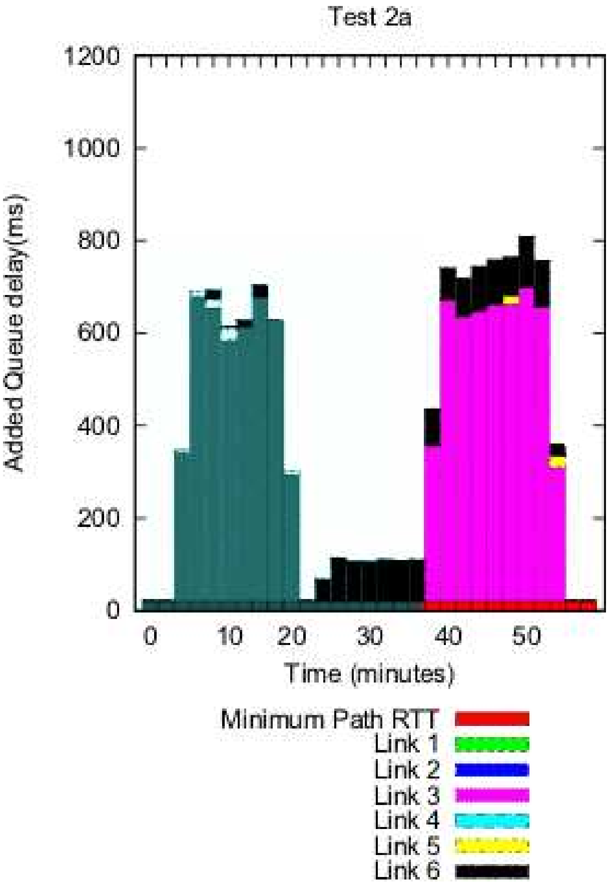

Figure 8: A plot taken from [12], showing the effects of TCP on queue delay. The non-gray part of the plot corresponds to measurements conducted with the same system setup as seen in Figure 7.

The CurveCP decongestion algorithm of the tested implementation achieves a commendably stable upper bound for queue delay in case of no competing traffic. Occasionally, the scheduler decides to back off far enough for the queue to completely empty (See events **B** and **E** in Figure 7).

However, queue delays appear to rise proportionally to the number of competing CurveCP sessions (events **F** and **G** in Figure 7). In such an environment, the Chicago decongestion behavior seems ineffective.

Still, most of the time, the queue delays of CurveCP compare favorably to those of a TCP session tested in the same configuration (see Figure 8). The TCP queue delays were seen to exceed those of a single CurveCP session several times over. In this respect, the decongestion behavior can certainly be seen as effective.

4 Conclusion

I set out to obtain sufficient data from various measurements to be able to reason about various practical advantages and disadvantages of using CurveCP-based transports over SSH-based transports. In particular, I expected to either confirm or deny various, as of yet unfounded, feature claims about the CurveCP protocol and conduct some performance measurements on the alpha-test implementation.

I managed to create a functional proof-of-concept remote shell utility, suitable for SSH-like tasks. At least subjectively, this demonstrates the viability of CurveCP being employed in place of SSH.

Regarding latencies, my admittedly indirect approach to their measurement nevertheless produced results that counter-indicate any significant issues for the current CurveCP implementation in that department, if a latency penalty not exceeding a couple of milli-seconds is acceptable for a given use-case.

In my tests with running clients and servers on the same host, the CurveCP test-implementation does not compare favourably to either SSH or HTTPS in terms of achieved bandwidth for large file transfers. In a more "real world", networked setup, on the server side, these differences diminish slightly without changing the overall picture. For smaller/shorter transfers/sessions on the other hand, the extremely terse CurveCP-handshake could compensate for the lower bandwidth, especially when compared to SSH with its much more expansive handshake. Based on this, the current CurveCP test-implementation could so far only be recommended for exceedingly short-lived or inherently low-bandwidth connections. However, if the claims made about the protocol's *availability/addressing features* are true, a smaller usable bandwidth on the same hardware can be an acceptable trade-off for data transfers that are in return much harder to disrupt (either maliciously or through imprudent connection-roaming).

I have found the packet scheduling behavior of the CurveCP test-implementation to be unstable and occasionally buggy. Especially during my experiments for a bandwidth-limited setup with a competing, aggressive TCP scheduler, the CurveCP scheduler gave a rather mixed performance. While some simple parameter adjustments in the underlying algorithm might prove sufficient to rectify these problems, deployment of the current CurveCP implementation alongside TCP in bandwidth-constrained environments can not be fully recommended at this point (or should at least be thoroughly tested).

In terms of cryptographic overhead, my results fully agree with the statements made on the CurveCP website. Concerning initial handshakes, the protocol is extremely terse, both in terms of required packets and transferred bytes. On the other hand, for long-running sessions, CurveCP suffers from a bit more general overhead (as expected).

4.1 Future Research

My experiments mostly lacked a proper research-infrastructure, the measurements should have been taken in a much more controlled environment or preferably on a much wider range of machines (for gathering quantitatively meaningful statistics). My research was exploratory, intended to touch on as many different aspects of the CurveCP protocol and implementation as possible in order to paint a crude picture of their respective issues, limits and capabilities. Indeed, while I have demonstrated the protocol's applicability as a transport for a remote shell, I have also identified the current incarnation of the Chicago packet scheduler as behaving erratically on occasion and the entire alpha-test implementation to be somewhat short of breath in the bandwidth department. (As Bernstein points out, much of this could change for a beta-test implementation, rendering these findings obsolete.[11])

Future research should put a strong focus on qualifying Chicago's behavior, which in my opinion has the most pressing issues. Possibly, these issues turn out to be the price to pay for a "well-behaved" packet scheduler and necessary in order to stay remotely competitive next to naive, "abusive' schedulers.

Needless to say, my prototyped software for CurveCP enabled remote shells could be further enhanced. Crucially, the essential feature of client authentication has so far been neglected, forcing me to resort to "security-by-obscurity" for my various tests. Additionally, my crude VPN-for-CurveCP tool could also benefit from the attention of a more experienced Unix programmer than myself. Forks/improvements of both pieces of software would therefore be greatly appreciated. The latter can be found at `https://github.com/benthor/remotty` and `https://github.com/benthor/stdxput2dev` respectively.

4.2 Personal Remarks

Finally, from a subjective point of view, I have found the protocol to be exceedingly well designed[9]. The source code of the reference implementation is refreshingly readable and the architecture is sensibly divided into different components with clear-cut scopes. Even the principles of the cryptographic session establishment can easily be comprehended by non-experts. The entire thing is a lesson in elegance and prudence.

[9] I almost succeeded in keeping myself from using the word "sexy" in this report.

References

[1] Daniel J. Bernstein. High-speed high-security cryptography: encrypting and authenticating the whole internet. talk slides, December 2010. http://cr.yp.to/talks/2010.12.28/slides.pdf.

[2] Daniel J. Bernstein. Curvecp: Usable security for the internet – addressing. website, February 2011. http://curvecp.org/addressing.html.

[3] Daniel J. Bernstein. Curvecp: Usable security for the internet – availability. website, February 2011. http://curvecp.org/availability.html.

[4] Daniel J. Bernstein. Curvecp: Usable security for the internet – confidentiality and integrity. website, February 2011. http://curvecp.org/security.html.

[5] Daniel J. Bernstein. Curvecp: Usable security for the internet – decongestion. website, February 2011. http://curvecp.org/decongestion.html.

[6] Daniel J. Bernstein. Curvecp: Usable security for the internet – efficiency. website, February 2011. http://curvecp.org/efficiency.html.

[7] Daniel J. Bernstein. Curvecp: Usable security for the internet – introduction to curvecp. website, June 2011. http://curvecp.org/index.html.

[8] Daniel J. Bernstein. Curvecp: Usable security for the internet – messages in curvecp. website, February 2011. http://curvecp.org/messages.html.

[9] Daniel J. Bernstein. Curvecp: Usable security for the internet – nonces in curvecp. website, February 2011. http://curvecp.org/nonces.html.

[10] Daniel J. Bernstein. Nacl: Networking and cryptography library. website, May 2011. http://nacl.cace-project.eu/.

[11] Daniel J. Bernstein. Re: Some exploratory benchmarking of curvecp... email, July 2011. http://permalink.gmane.org/gmane.network.curvecp/17/.

[12] Danny Groenewegen and Harald Kleppe. Detecting and quantifying bufferbloat in network paths. online - PDF, July 2011. http://ext.delaat.net/sne-2010-2011/p46/report.pdf.

[13] Michael Ligh. Cryptography of ssh. website, 2006. http://www.mnin.org/write/2006_sshcrypto.html.

A Problems

A.1 (Reverse) Heisenbugs

On some occasions, trying to measure CurveCP performance had a negative impact on the latter. In particular, active SSH sessions (e.g., running `top`) to a server machine running CurveCP appeared to affect decongestion behavior in a negative way. Similarly, while collecting operating-system level UDP statistics, a much lower bandwidth was achieved than in a similar but unobserved run. Unfortunately, I have discarded most data collected during these occasions, so these findings can not be taken into consideration.

A.2 Python

I used the Python programming language to implement most of my tools and software prototypes for this research. While this language allows for very rapid prototyping and is easy to use, I hit some unexpected obstacles, especially when it comes to lower-level intricacies of operating-system interaction.

Had I known about these problems in advance, I probably would have written my programs either in Perl or Lua[10], or would have used this as an excuse to wrap my head around Go.

A.2.1 Sockets

Python sockets can not be flushed and on top of that are *not* always flushed implicitly at the right time. While this might simplify some rapid prototyping, it really makes lower-level "getting one's hands dirty" difficult. In the course of developing my tool, I have found a small hack which seems to circumvent the problem. The trick is to obtain a *file descriptor* representing the socket and create a corresponding file object from that. The writes to that file object can then be flushed.

```
import socket,os
s = socket.socket()
s.connect(ADDRESS)

sfd = s.fileno()

sfile = os.fdopen(sfd, 'w')
sfile.write('23')
sfile.flush()
```

[10] LuaJIT currently makes Lua *the fastest* "scripting" language available. Together with its ability to allow calling external C functions and using C data structures from pure Lua code, this would probably have been the better tool for the job.

A.3 FreeBSD

I could not get CurveCP to work properly under FreeBSD. Even only running the simple CurveCP-sanity test described in the README failed for me. Although sessions could be easily established, they all closed/froze prematurely, i.e., before having transferred the complete contents of the file echo'd into the stream from the server side.