BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

PERFORMANCE MEASUREMENT TOOL FOR DNS SERVERS

NÁSTROJ PRO MĚŘENÍ VÝKONNOSTI DNS SERVERŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MATĚJ POSTOLKA**
AUTOR PRÁCE

SUPERVISOR **Ing. ROMAN VRÁNA**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Computer Systems (DCSY) Academic year 2019/2020

Master's Thesis Specification

Student: **Postolka Matěj, Bc.**
Programme: Information Technology Field of study: Computer Networks and Communication
Title: **Performance Measurement Tool for DNS Servers**
Category: Networking

Assignment:

1. Study the DNS application protocol.
2. Get acquainted with various implementations of DNS server applications.
3. Propose methods suitable for performance measurement of DNS request processing using both TCP and UDP protocol.
4. Implement a tool that is capable of performing proposed measurements.
5. Measure the performance of at least three different DNS server applications.
6. Discuss the achieved results and propose possible improvements to the created tool.

Recommended literature:

- According to the supervisor's recommendation.

Requirements for the semestral defence:

- Items 1 to 3 are required.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Vrána Roman, Ing.**
Consultant: Dražil Jan, Ing., UPSY FIT VUT
Head of Department: Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work: November 1, 2019
Submission deadline: July 31, 2020
Approval date: October 25, 2019

Abstract

This work deals with the design and implementation of a test framework for testing the performance of DNS servers on DNS traffic sent over the TCP and UDP transport protocols. It contains performance measurements of three different authoritative nameserver implementations. DNS query streams sent to the servers are composed of different types of DNS queries sent over varying network and transport protocols. A comparsion of the performance of individual nameserver implementations is performed.

Abstrakt

Tato práce popisuje návrh a tvorbu testovacího prostředí pro měření výkonnosti DNS serverů nad transportními protokoly TCP a UDP. Obsahuje výsledky výkonnostního měření tří různých implementací autoritativních DNS serverů nad síťovým provozem složeným z různých typů DNS dotazů zaslaných pomocí různých síťových a transportních protokolů. Je provedeno srovnání výkonnosti těchto implementací.

Keywords

DNS, TCP, UDP, DNS over TCP, Benchmarking

Klíčová slova

DNS, TCP, UDP, DNS over TCP, Benchmarking

Reference

POSTOLKA, Matěj. *Performance Measurement Tool for DNS Servers.* Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Roman Vrána

Performance Measurement Tool for DNS Servers

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mr. Roman Vrána. Additional information was provided by Mr. Jan Dražil. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Matěj Postolka
July 31, 2020

Acknowledgements

I would like to thank everyone who has supported me during work on this thesis.

Contents

Chapter 1

Introduction

The DNS application protocol is a key component of modern-day internet infrastructure. As the vast majority of applications which communicate over the internet cannot function without proper domain name resolution, DNS outages have severe, far-reaching consequences and should be avoided if at all possible.[3] Avoiding DNS outages requires a robust and diverse DNS infrastructure capable of processing high volumes of traffic and withstanding various types of attacks. These can range from Denial-of-Service-type attacks to the exploitation of bugs in DNS server software.

The DNS infrastructure is built around authoritative servers, which handle requests for specific zones, and resolvers, which enable users to perform lookups for arbitrary domain names.[4] As a result, a robust and resilient DNS infrastructure requires robust and resilient DNS server software, both with regards to authoritative servers and resolvers. Due to the hierarchical nature of DNS, authoritative servers on the root and .tld level receive the highest volume of traffic[3] and, consequently, have the most stringent requirements for resiliency and robustness, as outages of said servers impact the highest number of users.

Subjecting DNS server software to test scenarios which flood the server with different types and varying volume of traffic and measuring the rate at which the server responds under such conditions is one way to gauge the robustness of a given DNS server implementation.

The performance profiles of different DNS server implementations can then be used to identify weaknesses of the respective implementations, or to compare the performance of different implementations under various load scenarios. The profiles can also be used to select a combination of server implementations which meet a set of performance and stability criteria and are therefore a suitable choice for a diversified "server portfolio," which increases the robustness of a given DNS infrastructure.

Chapter 2

Domain Name System

The Domain Name System (usually abbreviated to DNS) is a distributed and hierarchical system;[4] hence, different performance requirements are placed on servers found at different positions within the DNS hierarchy. An understanding of the general principle of DNS lookups is necessary to understand the volume of traffic handled by the various servers.

2.1 Domain Namespace

DNS is, in essence, a distributed database indexed by *domain names*.[4] A *domain name* represents a path in an inverted tree, which is referred to as the *domain namespace* (see Figure 2.1).[4] Each node in the *domain namespace* has a text label, which must be from 1 to 63 characters long, with the exception of the root node, which has a special null (zero-length) label.[4]

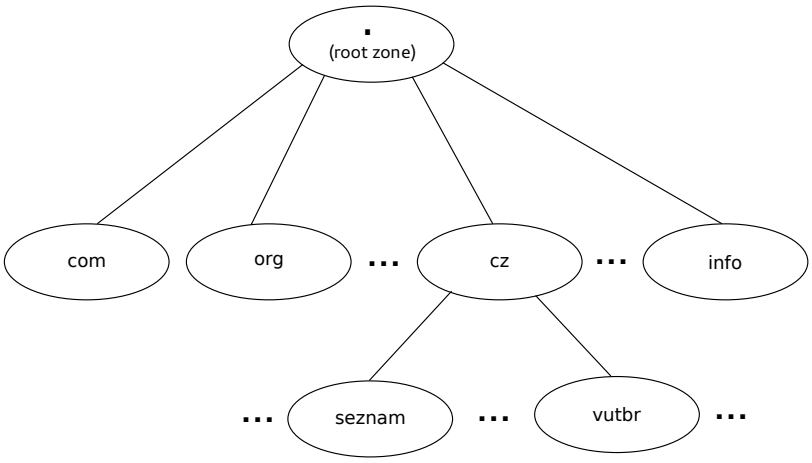

Figure 2.1: A subtree of the global *domain namespace*

The path from the root node to any given node in the *domain namespace* can be represented as a sequence of labels of the nodes which lie on the path from the chosen node to the root.[4] These labels, separated from each other by the dot (`.`) character, form a string which is called a *domain name*. Domain names are always read from the chosen node toward the root node (i.e. "up" the tree).[4]

The strings `"vutbr.cz."`, `"org."` and `"."` are all examples of valid domain names, each representing the respective paths $< vutbr, cz, . >$, $< org, . >$ and $< . >$ in the subtree illustrated in Figure 2.1.

Note that many user-oriented scenarios (domain name lookups, for example) leave out the final dot character representing the null label, so `"vutbr.cz"` and `"vutbr.cz."` are considered equivalent domain names.

2.2 Domains

As mentioned in Section 2.1, DNS is a distributed database indexed by *domain names*. A set of varying entries, called *resource records (RRs)*, can be associated with each *domain name* (see Chapter 3 for the description of the possible entry types).[4] These entries are kept in the zone files of authoritative nameservers responsible for the *domain* to which the *domain name* belongs.[4]

A *domain* is a subtree of the *domain namespace*. The *domain name* of a *domain* is the same as the *domain name* of the root node of a given *domain* subtree.[4] In the *domain namespace* subtree shown in Figure 2.1, the `"cz."` domain is the subtree which has the `cz` node as the root node, and contains the `seznam` and `vutbr` nodes, as well as all their children. Naturally, the `cz` domain also contains many other nodes, which is indicated by the `...` sign in Figure 2.1.

2.3 Zones

The servers which store information about *domains* are called *nameservers*. *Nameservers* usually have complete information about some part of the *domain namespace*, called a *zone*.[4] A *nameserver* responsible for a given *zone* is called the *authoritative nameserver* for that *zone*.[4]

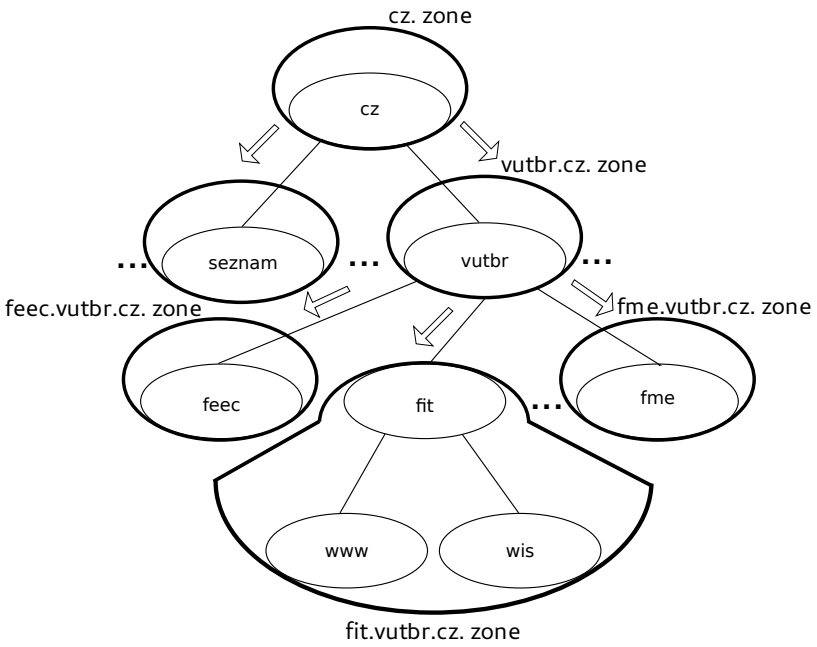

Figure 2.2: DNS zones and delegation

The principle of delegation is key to understanding the difference between *domains* and *zones*.[4] For example, the `cz.` domain contains a rather large number of subdomains, each of which can, in turn, contain more subdomains. Instead of managing the `cz.` domain as a single zone and storing *RRs* for every single domain name in the `cz.` domain on authoritative nameservers of the `cz.` zone, the `cz.` domain is broken up into multiple separate zones using delegation. This is illustrated in Figure 2.2; the arrows indicate the delegation of subdomains.

When a subdomain is delegated, a new zone is established. The responsibility, or *authority*, for managing the *RRs* of domain names in the newly established zone is then transferred to the authoritative nameservers responsible for the zone.[4] Top-level domains usually delegate all subdomains and top-level zones (such as `com`, `org` or `cz`) usually only contain information necessary for identifying the authoritative nameservers of the zones of the delegated subdomains.[4]

It should be noted that there can be multiple authoritative nameservers responsible for a zone, which is typically the case (for backup and load-balancing reasons).[4] Additionally, a single server can serve as an authoritative nameserver for multiple zones, which is also often the case in organizations which manage their own zones.[4]

2.4 Recursive Resolution

In order to access the *RRs* associated with a given *domain name*, it is necessary to query an authoritative nameserver responsible for the *zone* to which the *domain name* belongs (see Section 2.3). This server can be identified by using the *recursive resolution* strategy.[4]

Initially, the query for the domain name is sent to one of the root servers (authoritative nameservers responsible for the root zone). The root zone only contains *RRs* necessary for identifying authoritative nameservers of top-level domains (`com`, `org`, `cz`, ...), as all top-level domains are delegated to the respective top-level domain registrars.

As a result, a query for any domain name which contains one or more labels cannot be fulfilled by a root server. Instead, the root server will answer with the domain names of the authoritative nameservers responsible for the top-level domain the queried domain name belongs to. If the queried domain name is `www.fit.vutbr.cz`, the root server will answer with authoritative nameservers of the `cz` zone. The root server will also attach the IP addresses of these nameservers in a so-called "glue" record.[4]

This process will then be recursively repeated with the next authoritative nameserver, i.e. the same query (for `www.fit.vutbr.cz`) will be sent to one of the authoritative nameservers of the `cz` zone. In a similar fashion to the root servers, the `cz` domain also delegates all subdomains. As a result, the `cz` authoritative nameserver will answer with the domain names and IP addresses of the authoritative nameservers for the `vutbr.cz` zone. As seen in Figure 2.2, the `fit.vutbr.cz` domain is also delegated, which will cause another iteration of the process to take place, yielding the domain names and IP addresses of the authoritative nameservers for the `fit.vutbr.cz` zone. Finally, an authoritative nameserver for `fit.vutbr.cz` will be queried. Since `www.fit.vutbr.cz` belongs to the `fit.vutbr.cz` zone, the authoritative nameserver of the `fit.vutbr.cz` zone will be able to return the *RRs* requested in the query.

2.5 Resolvers

The relative complexity of performing domain name lookups using the recursive resolution strategy makes it inconvenient for use by regular users.[4] DNS resolvers are used to handle the recursive queries of users. Unlike authoritative nameservers, resolvers do not manage a well-defined set of zones, but attempt to correctly resolve arbitrary queries by recursively querying the authoritative nameservers using the method described in Section 2.4.[4]

In addition to performing recursive lookups, most resolvers are capable of caching responses to queries. As a result, repeated queries for the same *RRs* of a domain can be answered using the resolver's cache (assuming the cached data is still considered valid, see Chapter 3 for details) and the recursive lookup that would otherwise be necessary can be avoided. Caching greatly reduces the load on authoritative nameservers, but leads to other issues.

Note that terminology is not always consistent in the case of resolvers; resolvers as described above are, in fact, nameservers, however unlike authoritative-only nameservers, they are capable of executing recursive queries. A resolver is then understood to be a program capable of assembling a recursive query and sending the query to a nameserver willing to accept recursive queries (such resolvers are usually integrated into the operating system[5]). However, DNS server software which is labelled as a resolver is usually understood to be a nameserver capable of performing recursive queries (as is the case with *Knot Resolver*[1]). Some implementations also support both authoritative and non-authoritative nameserver (resolver) functionality, as is the case with *BIND*[2].

2.6 Performance Requirements

As a direct consequence of the recursive lookup strategy described in Section 2.4, the authoritative nameservers for the root zone (root servers) experience the highest volume of traffic (every single recursive lookup begins by querying a root server). The closer an authoritative nameserver is to the root in the domain namesapce, the higher the volume of traffic it must be able to process.[3]

In a similar fashion, the failure of a root server impacts the highest number of users, because all recursive lookups initiated with that particular server will fail. The failure of an authoritative nameserver for the zone of a top-level domain will have severe consequences as well, as all recursive lookups for a domain name in the given top-level domain sent to the faulty server will fail. As a result, root servers and authoritative nameservers for zones of top-level domains have the highest requirements for stability and resiliency, as outages of these servers have the potential to cause the most harm.

[1] https://www.knot-resolver.cz
[2] https://www.isc.org/bind/

Chapter 3

DNS Protocol

The DNS protocol is an application protocol used for querying the distributed DNS database for *RRs* associated with a given domain name.[4] Measuring the performance of authoritative nameservers and resolvers requires crafting and analyzing DNS messages. As a result, an understanding of the structure of messages exchanged as part of the DNS protocol is necessary.

3.1 Message Format

All communications inside the DNS protocol are carried in a singular format called a *message*.[5] The high-level format of messages is divided into 5 sections, as shown in Figure 3.1.[5] Some sections may be left out of certain messages.

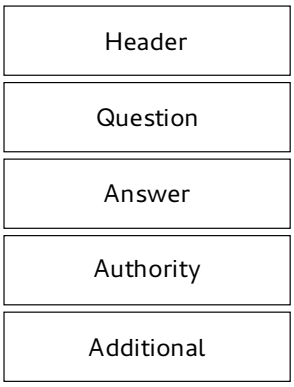

Figure 3.1: High-level format of DNS messages

The header section is present in every message and contains fields specifying which of the other sections are present.[5] Header fields also specify whether the message is a query or a response.

The names of the other sections are derived from their use in queries. The question section contains the request for the nameserver (domain name and the requested *RRs*). The answer section contains *RRs* which answer the question, the authority section contains *RRs* which point toward an authoritative nameserver and the additional records section contains *RRs* which relate to the query, but do not strictly answer the question.[5]

Header Section Format

The fields present in the header section are illustrated in Figure 3.2:

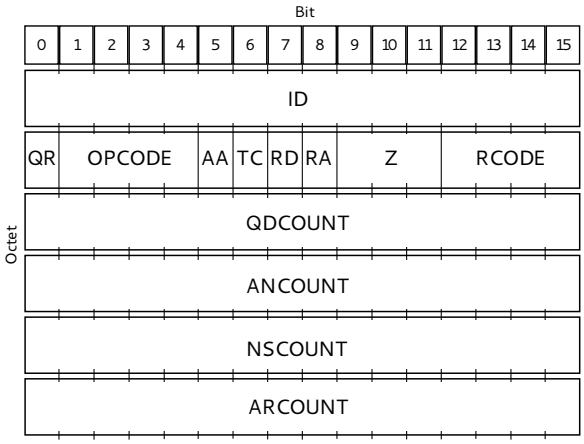

Figure 3.2: Header section format

The meaning of the individual fields is as follows:[5]

- **ID** – A 16-bit transaction identifier generated for every query. The identifier is copied into the reply and can be used by the requester to match the reply with the query.
- **QR** – A 1-bit field indicating whether the message is a query (0) or response (1)
- **OPCODE** – A 4-bit field specifying the kind of query the message carries. Set to 0 for standard queries.
- **AA** (Authoritative Answer) – A 1-bit field valid in responses, which indicates whether the responding server is an authoritative nameserver for the queried domain name.
- **TC** (Truncated) – A 1-bit field indicating whether the message was truncated due to the message exceeding the MTU of the transmission channel.
- **RD** (Recursion Desired) – A 1-bit field set in queries and copied into responses. Setting this bit directs the nameserver to resolve the query recursively.
- **RA** (Recursion Available) – A 1-bit field set or cleared as part of responses, which indicates whether the nameserver is willing to accept recursive queries.
- **Z** – Reserved field which must be set to 0 in all queries and responses.
- **RCODE** (Response Code) – A 4-bit field set as part of responses, with the following supported values:
  - *NOERROR* (0) – No error condition
  - *FORMERR* (1) – Format error; the nameserver was unable to interpret the query
  - *SERVFAIL* (2) – Server failure; the nameserver was unable to process the query due to an internal problem
  - *NOTIMPL* (3) – Not implemented; the nameserver does not support the requested type of query

  - *REFUSED* (4) – The nameserver refuses to perform the query (for example, an authoritative-only nameserver will refuse queries with the *RD* bit set)

- **QDCOUNT** – A 16-bit unsigned integer specifying the number of entries in the question section.
- **ANCOUNT** – A 16-bit unsigned integer specifying the number of *RRs* in the answer section.
- **NSCOUNT** – A 16-bit unsigned integer specifying the number of nameserver *RRs* in the authority section.
- **ARCOUNT** – A 16-bit unsigned integer specifying the number of *RRs* in the additional section.

Question Section Format

The question section is used to carry the question, which is a combination of the domain name and the requested *RRs* for the domain name. There is usually only one entry present in the question section of each message, as behavior for handling messages with $QDCOUNT \neq 1$ is not standardized.[6] As such, queries with $QDCOUNT \neq 1$ will receive an undefined response. The vast majority of nameservers, including *BIND*, *Knot DNS* and *NSD* do not support queries with $QDCOUNT \neq 1$ and respond with *FORMERR*.[6] The fields present in the question section are illustrated in Figure 3.3:

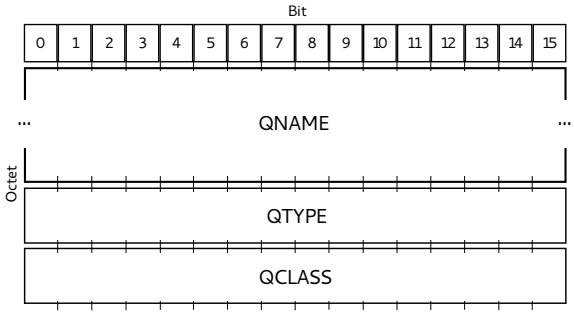

Figure 3.3: Question section format

The meaning of the individual fields is as follows:[5]

- **QNAME** – A domain name represented as a sequence of labels, where each label consists of a length octet followed by that number of octets. A zero-length octet (indicating the null label) is used to terminate the domain name.
- **QTYPE** – A 16-bit code specifying the type of the query. This can be the code of any *RR* type, or a special type (see Section 3.1).
- **QCLASS** – A 16-bit code specifying the class of the query. Today, all classes except *IN* (code 1) are obsolete.

Resource Record Format

The anwer, authority and additional sections are all made up of a variable number of *resource records*, where the number of records in each section is specified in the corresponding count field in the header.[5] The format of a *resource record* (*RR*) is illustrated in Figure 3.4:

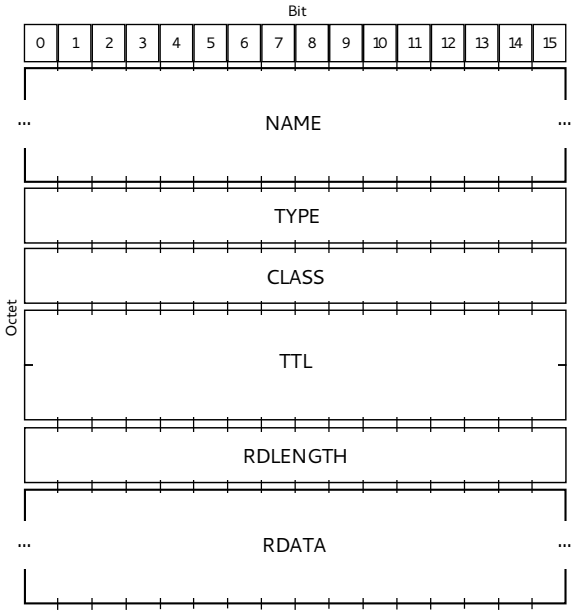

Figure 3.4: Resource record format

The meaning of the individual fields is as follows:[5]

- **NAME** – The domain name to which this record pertains.
- **TYPE** – A 16-bit code indicating the type of the *RR*. This code specifies the meaning of the data in the *RDATA* field.
- **CLASS** – A 16-bit code specifying the class of the *RR*. Today, all classes except *IN* (code 1) are obsolete.
- **TTL** – A 32-bit signed integer that specifies the maximum amount of time (in seconds) that this *RR* may be cached and returned by resolvers (non-authoritative nameservers), before an authoritative nameserver has to be queried again. A value of zero indicates that the *RR* may only be used for the transaction in progress and should not be cached (*SOA* records, for example, are always distributed with a zero *TTL*.[5])
- **RDLENGTH** – A 16-bit unsigned integer that specifies the length (in octets) of the *RDATA* field.
- **RDATA** – A variable-length field that contains the requested resource. The format of this field varies depending on the *TYPE* of the *RR*.

While there is a multitude of defined *RR* types, not all of them are currently in use (some defined in the original *RFC 1035* are considered obsolete). Additionally, more *RR* types

have been introduced over time, as is the case with the *AAAA RR* type for *IPv6* address records[9] or *RR* types used as part of DNS security extensions (*DNSSEC*).[2] Below are some of the most common *RR* types currently in active use:

- **A** (1) – An *IPv4* host address
- **AAAA** (28) – An *IPv6* host address
- **NS** (2) – An authoritative nameserver to which the requested zone is delegated
- **CNAME** (5) – The canonical name for an alias
- **SOA** (6) – Marks the start of a zone of authority and contains basic information about the zone
- **PTR** (12) – Pointer to a canonical name (used for reverse DNS lookups)
- **MX** (15) – A list of mail transfer agents for the domain name
- **TXT** (16) – Arbitrary text data

Additionally, there are several special types used to execute specific operations or request multiple resources:

- ***** (255) – Requests all records of all types known to the nameserver for the given domain name.
- **AXFR** (252) – Used by a secondary authoritative nameserver to request a zone transfer of the entire zone file from the primary authoritative nameserver.
- **IXFR** (251) – Used by a secondary authoritative nameserver to request an incremental zone transfer from the primary authoritative nameserver (the difference between the revisions of the zone on the primary and secondary server).
- **OPT** (41) – A pseudo *RR* type used to carry *EDNS* data.

Domain Name Compression

DNS messages utilize a compression scheme which eliminates the repetition of domain names.[5] This is used to reduce the size of messages, as the same domain name often occurs multiple times in the same message (with the first occurrence typically being in the question section and other occurrences in *RRs* in the answer, authority and additional sections). The compression scheme works by replacing a part of, or, alternatively, the entire, domain name (usually found in the *NAME* field of an *RR*) by a pointer to a prior occurrence of the same name.[5]

The pointer is a 16-bit sequence, where the two most significant bits are set to one. This effectively distinguishes a pointer from a label; labels are restricted to 63 characters or less, which means that the two most significant bits of a length octet for any label must always be zero.[5] The rest of the sequence specifies the byte offset from the start of the message (i.e. from the *ID* field of the header) where the referenced domain name occurs.

The compression scheme allows for any domain name in a DNS message to be represented as a standard sequence of labels ending in a zero-length octet, a pointer, or a sequence of labels ending with a pointer.[5]

3.2 Transport

Port 53 (both UDP and TCP) is the official IANA-assigned port for DNS.[5] Messages carried over UDP are restricted to a maximum size of 512 bytes (not counting the IP and UDP headers). If a message carried over UDP exceeds the 512 byte limit, the *TC* bit in the header is set and the rest of the message is truncated.[5]

Messages sent over TCP connections are prefixed with a 16-bit length field, which specifies the total length of the message, excluding the length field. This allows the server to determine when a message is fully assembled and ready for parsing. Due to the fact that DNS over TCP does not have the 512 byte message size limit inherent to DNS over UDP, TCP is used for handling more complex transactions, such as zone transfers.[5]

However, the vast majority of regular DNS queries is still currently sent over UDP. The zone of the Swiss (`ch`) *ccTLD* is a representative example; only about 2% of the total queries sent to the `ch` zone authoritative nameservers in 2019 were made over TCP.[1] With *DNSSEC* and *EDNS* becoming more prevalent, however, the proportion of TCP traffic is predicted to steadily increase, as some messages which contain *EDNS* data can exceed the 512 byte UDP limit. This is also one of the reasons for measuring the performance of DNS over TCP.

3.3 Master Files

Master files are text files which contain *RRs*.[5] Since the contents of a zone can be defined by a list of *RRs* (together with the domain name they correspond to), master files are used for defining zones on authoritative nameservers.

A master file is formatted as a sequence of entries placed on separate lines. Parentheses can be used to spread entries across multiple lines (this is often done with the *SOA* entry). Each line of the master file is made up of a domain name followed by an *RR* definition, which has the form of `[<TTL>] [<CLASS>] <TYPE> <RDATA>`.[5] The following is an example of a master file:

```
test.           86400    IN      SOA     ns.test. (
    test.test.  ; Administrator e-mail
    2019021300  ; Serial
    1800        ; Refresh
    900         ; Retry
    604800      ; Expire
    86400       ; Minimum
)

test.           86400    IN      NS      ns.test.
ns.test.        86400    IN      A       127.0.0.1
ns.test.        86400    IN      AAAA    ::1
a.test.         86400    IN      A       127.0.0.1
```

The above master file contains the definition of the `test` zone with one authoritative nameserver (`ns.test`).

[1] https://www.nic.ch/statistics/dns/udp-tcp/

Chapter 4

Nameserver Implementations

The number of actively maintained nameserver implementations in widespread use is relatively high, but the individual implementations often differ in the scope of supported features. This is due to the fact that some implementations, such as *BIND*, are designed to be feature-complete, while others are intended to be used in a specific scenario (*Knot DNS*, for example, is an authoritative-only nameserver designed for infrastructure use). Both authoritative and non-authoritative nameservers (resolvers) intended for infrastructure use are covered in this chapter and considered for benchmarking.

4.1 BIND

BIND is the oldest and most commonly deployed nameserver.[1] *BIND* is written in `C` and is, in essence, considered to be the reference nameserver implementaion. It is feature-complete and can be used in both authoritative and non-authoritative (resolver) scenarios, and has full *EDNS* and *DNSSEC* support.

BIND typically runs as a daemon and can be configured to serve multiple zones, in either primary authoritative nameserver (master) or secondary authoritative nameserver (slave) mode. Support for recursive queries can also be enabled in the configuration file. *BIND* has support for multi-threading; typically, *BIND* will detect the number of CPUs and create one worker thread per CPU (the number of worker threads can be modified using a parameter). *BIND* supports zone definitions in the format of master files.

While *BIND* is considered to be feature-complete and stable, it has poor performance in comparison to other implementations. For example, in benchmarks regularly conducted by *CZ.NIC*, the response rate of *BIND* is more than 75% lower than the response rate of *Knot DNS* in some benchmark scenarios.[2]

4.2 Knot DNS

Knot DNS is an authoritative-only nameserver implementation developed by *CZ.NIC*. Like *BIND*, it has full *EDNS* and *DNSSEC* support. *Knot DNS* does not support recursive queries.[3]

[1] https://www.isc.org/bind/
[2] https://www.knot-dns.cz/benchmark/
[3] https://www.knot-dns.cz/

Knot DNS is written in `C` and uses the *libknot* library for parsing and assembling DNS messages. It is possible to use *libknot* as a low-level DNS message processing library in other projects.

Like *BIND*, *Knot DNS* is feature-complete (with respect to features available on authoritative-only nameservers) and multi-threaded. *Knot DNS* also performs significantly better than *BIND* in all benchmark scenarios tested by *CZ.NIC*. *Knot DNS* is currently deployed as one of the authoritative nameserver implementations on several *ccTLD* zones and the root zone.

4.3 NSD

NSD is another authoritative-only nameserver implementation developed by *NLnet Labs*. In a similar to fashion to *Knot DNS*, it is a feature-complete and multi-threaded implementation intended for infrastructure use.[4] Like *Knot DNS*, it is one of the deployed implementations on multiple *ccTLD* zones and the root zone.

[4] https://www.nlnetlabs.nl/projects/nsd/about/

Chapter 5

Benchmarking Methodology

Creating realistic and comparable benchmarks of different nameserver implementations presents a number of challenges. Due to the design of DNS, infrastructure-level nameservers are subject to a large volume of varying queries from a large pool of clients. This must be reflected in the traffic generated for benchmarking purposes. Additionally, the measured *performance* of different implementations must be well-defined in order to be mutually comparable. As a minimum, the following must be taken into account:

1. It must be possible to generate diverse types of DNS queries, depending on what is being tested (response to large queries, response to TCP or UDP queries, a *DoS* scenario with a high volume of complex queries originating from a large set of hosts, or a different scenario).
2. It must be possible to send queries with a sufficiently high frequency to match and exceed the load typically placed on infrastructure-level nameservers.
3. The *performance* of a nameserver implementation under a given set of test conditions must be quantified in a way that makes it comparable with other implementations tested under the same set of conditions and with the *performance* under a different set of conditions.

5.1 Performance Quantification

The high-level approach to determining the *performance* of a nameserver implementation involves generating a volume of DNS queries of a certain type (depending on what is being tested) and measuring the *response rate*, i.e. how many of the generated queries the server was able to successfully process. This process is then repeated for different types and volumes of queries in a given time frame to determine the overall *performance* of the implementation.

Let $s = \{q_1, q_2, ..., q_n\}$ be a query scenario. The query scenario s is defined by a set of queries, where each query $q_1, q_2, ..., q_n$ represents a valid DNS message with the header QR bit set to 1. This message contains exactly one entry in the question section. The *performance* of an implementation is then evaluated for a given set of query scenarios $S = \{s_1, s_2, ..., s_n\}$ with queries generated at frequencies defined by the frequency set $F = \{f_1, f_2, ..., f_x\}$. A *response rate* is measured for each test case $c = (s, f) \in S \times F$ where queries from s are selected randomly with a uniform distribution and generated at frequency f for a constant amount of time. The *response rate* $r = q_p/q_s$ where q_p denotes the number

of queries successfully processed by the nameserver and q_s denotes the number of queries generated for a given test case c. The *performance* of a nameserver implementation is then defined by a set of ordered pairs $P = \{(c_1, r_1), (c_2, r_2), ..., (c_n, r_n)\}, c_i \in S \times F$.

For the sake of comparing the performance of different nameserver implementations, it is possible to define a "better than" relation $\gg$ where $P_a \gg P_b \Leftrightarrow \forall(c_a, r_a) \in P_a \forall(c_b, r_b) \in P_b : c_a = c_b \Rightarrow r_a > r_b$. In order for the performance sets to be comparable, it must also hold that $\forall(c_a, r_a) \in P_a \exists(c_b, r_b) \in P_b : c_a = c_b$. Similarly, $\forall(c_b, r_b) \in P_b \exists(c_a, r_a) \in P_a : c_a = c_b$.

In addition to maintaining the same set of query scenarios and frequencies, all implementations must be tested on identical hardware, compiled with the same level of optimizations and their runtime configurations must match as closely as possible. In the case of authoritative nameservers, it is also necessary to load identical zone files, as all the implementations should be forced to take the same pipeline when attempting to resolve queries from s.

5.2 UDP Benchmarking

The approach described in Section 5.1 is technically feasible for benchmarking UDP traffic. Due to the stateless nature of the UDP protocol, the frequency at which queries can be generated and sent to the server is only limited by the capabilities of the generator and the underlying L1 technology. The performance of the implementation is then determined purely by the (c, r) pairs.

A Spirent TestCenter (*STC*) device can be used to generate UDP traffic at a bitrate of up to 10 Gb/s. The traffic generator supports the use of *modifiers*, which are able to randomize certain parts of a packet before it is sent. This makes it possible to generate queries at a rate of up to 10 Gb/s, with the query scenarios being very extensive due to the possibility to randomize source IP addresses, ports and data in the question section of DNS messages.

The *STC* also contains a traffic analyzer, which can be used for measuring the response rate. The traffic analyzer can track and classify incoming packets according to the values of bits specified by a bitmask and offset from the start of a given section of the packet. It is therefore possible to, for example, group the replies from the server by their *RCODE*.

Since the *STC* provides access to a Python scripting API, it is possible to create a Python framework capable of creating large query scenarios. The queries from those scenarios are then sent by the *STC* at desired frequencies for a constant amount of time. The traffic analyzer is used to measure the response rate for every (s, f) pair. The process can be repeated for other query scenarios to obtain a detailed *performance* profile of the given implementation.

5.3 TCP Benchmarking

Due to the stateful nature of the TCP protocol, the approach described in Section 5.1 is not sufficient to fully assess the performance of an implementation. In DNS over TCP, all queries must be sent over established TCP sessions. When the server is unable to establish new sessions, a query cannot be sent. As a result, more factors need to be taken into account when assessing overall nameserver performance on TCP; apart from the (c, r) pairs, the rate of successfully established sessions must be considered. Additionally, TCP keep-alive can be used to send multiple queries over a single session, which must also be taken into account when assessing overall *performance*. It is important to note that while TCP session

management is key in determining the performance of any given implementation, session management is the responsibility of the operating system kernel and not the application.

Since the *STC* cannot handle TCP connections, a different approach to generating queries is necessary. It is possible to use a custom DNS traffic generator and analyzer which makes use of the *Dataplane Development Kit* to generate traffic at a sufficient bitrate. The *Dataplane Development Kit* (*DPDK*) is a low-level packet acceleration framework.[1] Processing packets with *DPDK*'s pipeline is significantly faster than using *BSD* sockets and the regular kernel pipeline. *DPDK* can, therefore, be used to create an application capable of generating a high volume of DNS queries and analyzing the server's response.

In this case, the overall performance of a given implementation will not be determined by (c, r) pairs, but by (c, r, t) tuples, where $t = t_e/t_o$. t_e denotes the count of successfully established TCP sessions (over which a query is subsequently sent) and t_o denotes the total number of sessions which the generator attempted establish. Values in the frequency set F then determine the frequency of opening new TCP connections instead of the frequency of sending queries. Every successfully established session is used to send at least one query. More queries can be sent over a single session if TCP keep-alive is used.

[1] https://www.dpdk.org/

Chapter 6

STC Framework

This chapter describes the main features and mechanics of the Spirent TestCenter (*STC*) framework used for benchmarking the performance of implementations over UDP. The framework can be used for testing multiple combinations of query scenarios to determine the overall *performance* of an implementation under such scenarios. The framework is designed in a modular, reusable fashion, which makes it possible to add more scenarios of varying complexity.

The framework is written in Python 3 and made up of several key components, namely a simple TCP client and wrapper which exposes Python-equivalents of the TCL-based Spirent Automation API and provides a layer for communication with a server running on a machine with a working installation of the Spirent Test Center Application. This server then relays the commands it receives from clients to the actual Spirent device using the API exposed by the Spirent Test Center Application.

The rest of the test framework aims to provide full abstraction from the underlying STC Automation API by providing a series of classes which allow the creation, scheduling and execution of different test scenarios and subsequent result export and analysis. These scenarios can be set-up and launched without the use of any specific Spirent Automation API handles and commands.

6.1 Test Environment Components

A typical benchmarking setup involving the use of this framework will be made up of four distinct components, as shown in Fiugre 6.1:

Test Server

The Test Server runs the test framework and collects results. It sends commands and receives raw data from the STC Server (which is used to control the Spirent device). While it can run on a stand-alone machine, it can also run on the DNS Server subject to testing or any other machine which is able to open a TCP connection to the STC server (including the STC server itself).

STC Server

The purpose of the STC Server is to relay the Python API calls made by the Test Server to the Spirent device. It must be a machine with a working native Spirent TCL API

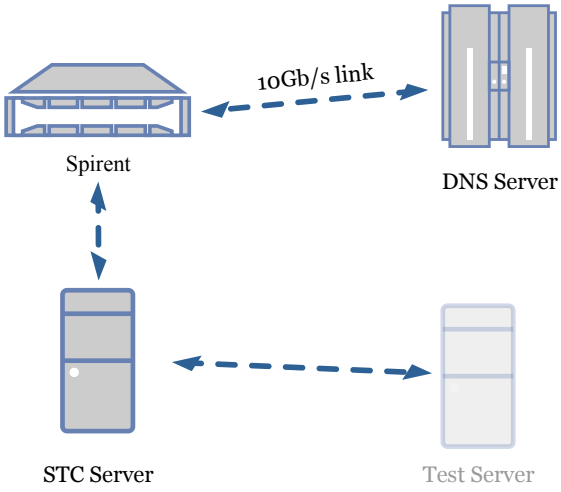

Figure 6.1: STC Framework Components

installation able to connect to the Spirent device and an active TCP relay server accepting connections from Test Server clients.

Spirent

The Spirent device is used for generating traffic and analyzing the DNS server's responses. While the setup illustrated in Figure 6.1 uses a 10 Gb/s link to connect the Spirent device with the DNS server subject to testing, it is possible to use line cards with a different link speed, as the framework scales the query generating frequency according to the percentage of the maximum link speed.

Outgoing queries are modified based on the the query scenarios configured on the Test Server and response traffic statistics are grouped according to *RCODEs* of the DNS server's responses. Response traffic statistics are read by the Test Server for further processing.

6.2 STC Automation API

The STC Automation API provides components for controlling the STC. The test framework works as an abstraction layer from STC Automation API calls and components, but still relies on them internally to be able to interface with Spirent. Additionally, some objects provided by the test framework mirror STC Automation API components to a large extent. As a result, a basic understanding of some relevant STC Automation API components and concepts is necessary to understand how the framework functions.

Overview

The API provides a hierarchical, object-oriented model for controlling the STC.[1] API objects are organized in a tree structure with the root of the tree anchored at the `System1` object. Objects can be configured by manipulating their respective attributes and can be referred to using handles generated by the STC when instantiating the objects [1]. Handles are strings which take the form of `<ObjectType><Identifier>`, so the first port object created in a project will have the handle `Port1`. A handle is valid until a given object or

its parent is destroyed. Objects are instantiated in the memory of the STC Server and pushed to the Spirent device when the `apply` API call is executed or when state-changing commands are run (see Section 6.2).

The root of the object tree is always referenced by the `System1` handle. All other objects are children of `System1`. Programming the STC entails creating child objects of `System1` and setting their attributes to the desired values. Some objects are created automatically; creating a `Port` object will, for example, automatically create `Generator` and `Analyzer` objects associated with the `Port` object. An example of a valid object hierarchy is shown in Figure 6.2:

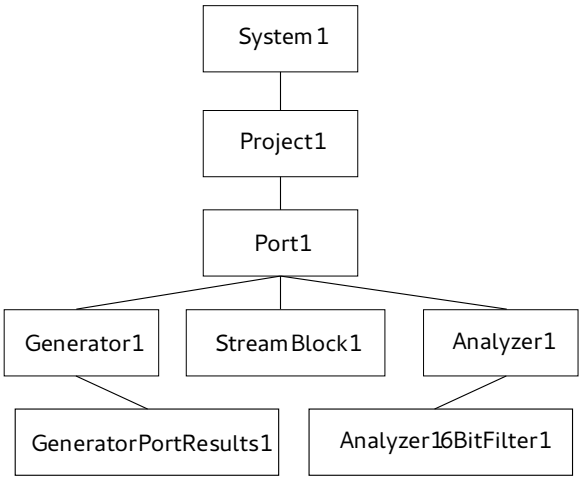

Figure 6.2: Example STC Object Hierarchy

Project Objects

`Project` objects serve as parent containers for a given configuration. They do not contain any configuration attributes significant for traffic generation or analysis. A full STC configuration can be loaded or exported by loading or exporting a given `Project` and its children.

Port Objects

`Port` objects serve as the parent objects for any port-related configuration. They are used for configuring Spirent-specific internal parameters of the physical port, most notably the port's location within the STC chassis. `Port` objects contain the following relevant configuration attributes:

- `Location` - Location of the port within the STC chassis

Generator Objects

`Generator` objects are automatically created as children of `Port` objects and represent the traffic generator of the physical port. `Generator` objects do not contain any configurable attributes, as the traffic frames are defined by `StreamBlock` objects, but serve as parent objects for `GeneratorPortResults` objects, which contain statistics on generated traffic, and `GeneratorConfig` objects, which contain configuration related to traffic generation.

GeneratorPortResults Objects

`GeneratorPortResults` objects can be created as children of `Generator` objects and used track generated traffic statistics. These objects contain purely read-only attributes. The relevant attributes are:

- `TotalBitCount` - Total number of bits generated.
- `TotalFrameCount` - Total number of frames transmitted.

GeneratorConfig Objects

`GeneratorConfig` objecs are automatically created as children of `Generator` objects and used to configure the parameters of traffic generation:

- `Duration` - Length of the packet transmission.
- `DurationMode` - Defines the unit of `Duration`. The framework uses seconds.
- `FixedLoad` - Load of the generator.
- `LoadUnit` - Defines the unit of `FixedLoad`. The framework uses `PERCENT_LINE_RATE`, which specifies load as a percentage of the maximum bandwidth available on the port.

StreamBlock Objects

A `StreamBlock` can be used to define an arbitrary frame which is then sent out by the `Generator` as a stream at a bitrate specified by `GeneratorConfig`. Frames are assembled from individual PDU objects using an XML configuration string parsed by the STC.

Below is an example of a DNS query configuration string:

```
<frame>
 <config>
  <pdus>
   <pdu name="eth1" pdu="ethernet:EthernetII">
    <srcMac>AA:AA:AA:AA:AA:AA</srcMac>
    <dstMac>BB:BB:BB:BB:BB:BB</dstMac>
   </pdu>
   <pdu name="ip_1" pdu="ipv4:IPv4">
    <sourceAddr>1.1.1.1</sourceAddr>
    <destAddr>2.2.2.2</destAddr>
   </pdu>
   <pdu name="udp_1" pdu="udp:Udp">
    <sourcePort>1234</sourcePort>
    <destPort>53</destPort>
   </pdu>
   <pdu name="dns_1" pdu="custom:Custom">
    <pattern>a0a000000001...</pattern>
   </pdu>
  </pdus>
 </config>
</frame>
```

The STC supports certain builtin PDU types, such as *Ethernet*, *IPv4* or *UDP* headers. However, DNS is not supported, so the DNS header and question sections are defined in the `pattern` element of the *Custom* PDU using a hexadecimal string.

`StreamBlock` objects contain the following relevant configuration attributes:

- `FrameConfig` - XML configuration string defining the PDUs in the stream's frames.
- `FrameLengthMode` - Defines whether the frame length is set to a fixed value, calculated automatically or variable.
- `FixedFrameLength` - Length of the frame if fixed frame length mode is used.

RandomModifier Objects

`RandomModifier` objects can be created as children of `StreamBlock` objects and are used to randomize parts of the frames sent in the stream of the given `StreamBlock`. Modifiers can be attached to specific PDUs by setting their `OffsetReference` attribute to the `name` attribute of the PDU in the XML configuration string.

`RandomModifier` objects contain the following relevant configuration attributes:

- `Mask` - Byte mask that specifies the data bytes to be modified.
- `Offset` - Offset in bytes to which the mask is applied.
- `OffsetReference` - String which specifies the PDU to which the offset and mask are applied. PDUs are specified using their `name` attributes (for example `dns_1`).
- `Seed` - Seed value of the random number generator.

Analyzer Objects

`Analyzer` objects are automatically created as children of `Port` objects and handle incoming traffic on the physical port. They do not contain any relevant attributes, but serve as containers for `AnalyzerPortResults` and `Analyzer16BitFilter` objects.

AnalyzerPortResults Objects

`AnalyzerPortResults` objects are automatically created as children of `Analyzer` objects and track overall incoming traffic statistics. The relevant attributes are:

- `TotalBitCount` - Total number of bits received.
- `TotalFrameCount` - Total number of frames received.

Analyzer16BitFilter Objects

`Analyzer16BitFilter` objects can be used to differentiate incoming frames by comparing up to 16 bits of data selected by a mask at a specified offset. This allows tracking statistics of different *RCODE*s in received DNS replies, as the filter creates separate result statistics objects for every "key" defined by the bits selected by the mask. The relevant attributes are:

- `Mask` - 16-bit mask which specifies which bits to differentiate.
- `Offset` - Offset in bytes to which the mask is applied.
- `LocationType` - Defines where `Offset` is referenced from. The framework uses `START_OF_TCP_UDP_HDR`.

ResultDataSet Objects

`ResultDataSet` objects are created for key values filtered out by filters (such as the `Analyzer16BitFilter`). One object is created for every key value detected by the filter. Similarly to `*PortResult` objects, they contain attributes which can be read to obtain statistics for incoming frames which fall under the given key:

- `BitCount` - Total number of bits received for the given key.
- `FrameCount` - Total number of frames received for the given key.

Commands

While objects are configured by setting their respective attributes, actions involving objects are executed using commands. The STC API supports commands which can be performed on various types of objects. Below are the relevant commands used in the framework:

- `GeneratorStartCommand` - Start the generators passed in the `generatorlist` argument.
- `GeneratorWaitForStopCommand` - Blocks until the generators passed in the `generatorlist` argument have stopped.
- `GeneratorStopCommand` - Stop the generators passed in the `generatorlist` argument.
- `AnalyzerStartCommand` - Start the analyzers passed in the `analyzerlist` argument.
- `AnalyzerStopCommand` - Stop the analyzers passed in the `analyzerlist` arguments.

6.3 Framework Modules

The framework is designed in an object-oriented, modular fashion. The framework's modules contain classes which are used to encapsulate native STC API objects, assemble various test scenarios or handle the export of results. Some modules can be used independently of the rest of the framework. The modules which make up the framework are shown in Figure 6.3.

Wrapper Objects Module

The *Wrapper Objects* module contains wrapper classes for STC API objects. Wrapper classes are provided for `Generator`, `Analyzer`, `Streamblock`, `Analyzer16BitFilter` and `ResultDataSet` objects. The main purpose of the wrapper classes is to hide the STC API objects and allow the use and configuration of these objects in a simple and consistent manner.

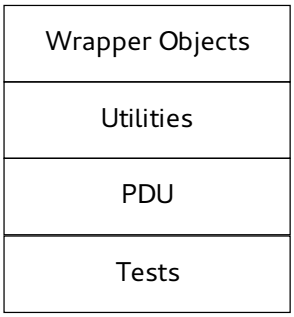

Figure 6.3: Framework Modules

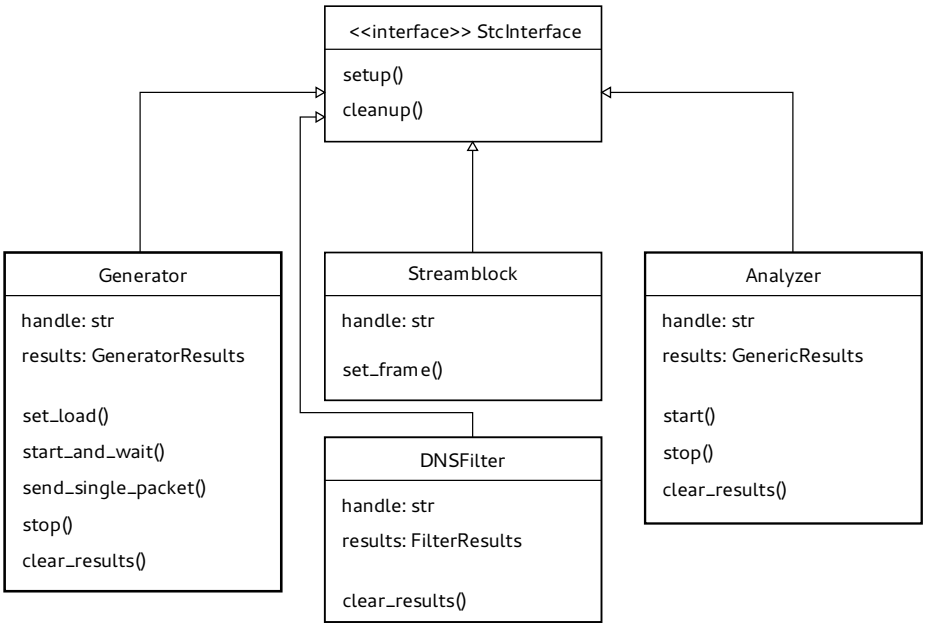

Figure 6.4: *Wrapper Objects* Class Diagram

As shown in the class diagram in Figure 6.4, the `Generator`, `Streamblock`, `Analyzer` and `DNSFilter` classes all implement the `StcInterface` interface. `StcInterface` consists of two abstract methods: `setup()` and `cleanup()`.

All wrapper classes use the same lifecycle model; instantiation only initializes the object's internal state (sets default attributes and retrieves the STC API handle for the given object if applicable). No direct changes to the Spirent device configuration are made or applied. Calling the `setup()` method will then apply the object's internal state (defined by the values of its attributes) to the hardware on the Spirent device. Similarly, calling the `cleanup()` method will remove any changes applied to the Spirent device by the given object and its children.

Aside from wrapper classes, the module also contains classes used for storing relevant result statistics tracked by `GeneratorPortResults`, `AnalyzerPortResults` and `ResultDataset` STC API objects. Figure 6.5 illustrates the three result classes; `GenericResults` and the `GeneratorResults` subclass are used for storing statistics from the `AnalyzerPortResults` and `GeneratorPortResults` objects. The `FilterResults` class is used for storing combined statistics from `ResultDataset` objects; statistics for individual keys are returned from the

`get_results()` method, which takes the result key as an argument. This is used to obtain statistics for individual *RCODE*s.

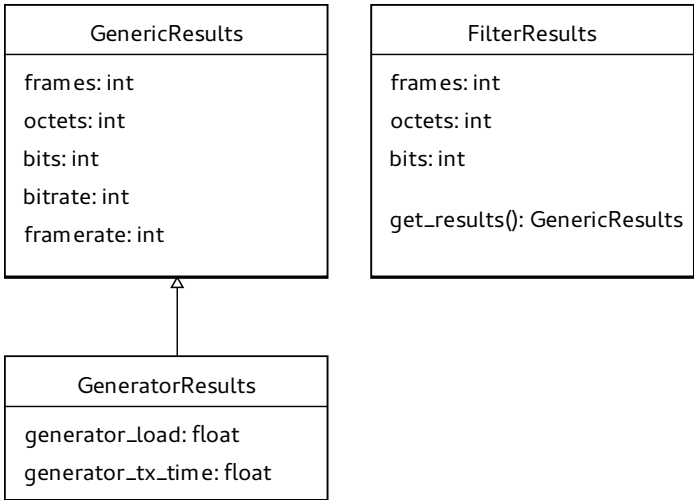

Figure 6.5: *Wrapper Objects* Result Classes

Utilities Module

The *Utilities* module contains utility classes used across modules. The most notable of these is the `StcModifier` static class, which contains methods for adding different types of modifiers to `Streamblock` objects. The class contains methods for adding modifiers to source IP addresses, source UDP ports, the DNS transaction ID, and the *QNAME* in the question section.

PDU Module

The *PDU* module contains classes which are used to assemble frames and generate the XML configuration string defining these frames, which can be passed to `Streamblock` objects via the `set_frame()` method. The frames are subsequently transmitted by Spirent when the `Generator` object is started. Classes provided by the *PDU* module mirror the PDU units used in regular ethernet frames. The class diagram of the *PDU* module is shown in Figure 6.6.

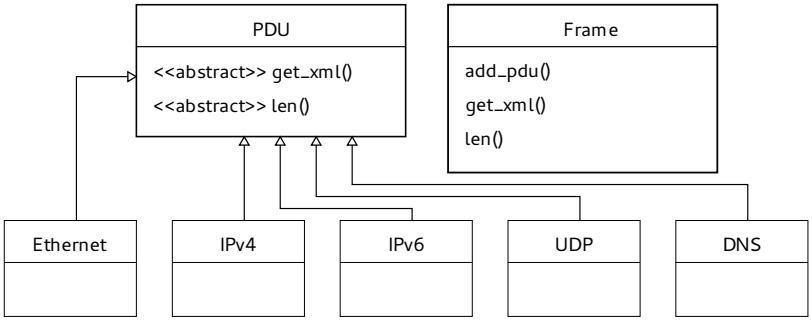

Figure 6.6: *PDU* Class Diagram

All PDU classes inherit from the base abstract `PDU` class, which contains two abstract methods: `get_xml()` and `len()`. The `get_xml()` method returns the STC XML configuration string (as described in Section 6.2) of the given PDU unit. The `len()` method returns the total length of the given PDU unit in bytes, which is used when configuring a fixed frame length on `Streamblock` objects.

The `DNS` PDU class internally relies on the `custom` STC header, as DNS headers are not natively supported by the STC. The hexadecimal string which defines the DNS payload is obtained from the `DnsQuestion` and `DnsQuery` helper classes, which provide user-friendly methods for assembling DNS queries. The `DnsQuestion` class accepts the *QNAME*, *QTYPE* and *QCLASS* parameters as constructor arguments and returns the hexadecimal string of the entire question in the `hexstring` property. The `DnsQuery` class accepts the transaction identifier as a constructor argument and allows appending a question section by passing a `DnsQuestion` object to the `add_question()` method. The `hexstrig` property of the `DnsQuestion` class returns the hexstring of the entire DNS payload (the header and question sections).

Assembling a complete frame is achieved by encapsulating individual PDUs in a `Frame` object. The `Frame` class allows appending `PDU` subclasses with the `add_pdu()` method. The `xml` property of the `Frame` class then returns the full XML configuration string of the frame, while the `len` property returns the length of the entire frame. These can be passed to `Streamblock` objects to fully define frames.

Tests Module

The *Tests* module is the most extensive module of the framework and contains classes which define different test traffic scenarios. All test classes inherit from the `BaseTest` class which implements `StcInterface`. As a result, the `BaseTest` class contains the `setup()` method, which is used to initialize `Generator`, `Analyzer` and `Streamblock` objects before the test begins, and the `cleanup()` method, which returns the objects to their initial state after the test has completed. The `StcInterface` lifecycle model allows multiple test objects to be instantiated at the same time while ensuring that only one object's configuration is actively loaded in the STC. The `cleanup()` method also ensures that any result objects are cleared, which prevents statistics from various test scenarios from interfering with each other.

The module contains test scenarios for *A*, *AAAA* and *MX* *QTYPE*s. Each *QTYPE* has scenarios with no randomization, randomized *QNAME*s, randomized DNS transaction identifiers, randomized source IP addresses, randomized source ports and the combinations of the respective randomization options. This allows creating extensive query scenario sets.

The methodology of the test is defined in the `run()` method of the `BaseTest` class. The default methodology consists of sending traffic bursts lasting 10 seconds and incrementing the load of the `Generator` object by 2.5% of the maximum link bandwidth after every burst, until 100% link bandwidth is reached. Additionally, a 5 second delay is placed between individual bursts to prevent traffic belonging to different bursts from mutually impacting statistics. The test methodology can be changed in individual tests by overriding the `run()` method of the `BaseTest` class.

The default frame used by the `BaseTest` class does not contain any PDUs. Individual tests must add the desired PDUs by overriding the `_frame_setup()` method and appending PDUs to the `Frame` object stored in the `_frame` attribute. If modifiers are used, they must be added in an overridden `setup()` method, as inserting modifiers immediately creates new STC objects and the `_frame_setup()` method is called from the `BaseTest` constructor;

inserting modifiers on instantiation would violate the `StcInterface` lifecycle model and potentially ruin the state of currently set-up objects.

Apart from individual test scenarios, the *Tests* module contains the `ResultSet` class, which is used to group `Generator`, `Analyzer` and `DnsFilter` results generated by the tests, and the `TestSet` class, which is used to group multiple tests into a larger testing scenario. Test scenarios in `TestSet`s are run consecutively and `ResultSet`s are stored for each scenario. A `TestSet` can also be labelled with additional metadata and exported into an HTML report, which contains performance graphs generated by the *Plot.ly*[1] library and `ResultSet` data in JSON format.

[1] https://plotly.com

Chapter 7

DPDK TCP Generator

The STC framework can be used for benchmarking the performance of nameserver implementations over UDP, but cannot be used for TCP benchmarks, as management of TCP connections is not supported by the available Spirent TestCenter hardware. The *DPDK TCP Generator* is a custom application capable of generating DNS queries over TCP connections at a high bitrate and analyzing response statistics in a fashion similar to the STC framework. *DPDK TCP Generator* makes use of the *DPDK*[1] packet acceleration framework.

7.1 Data Plane Development Kit

The *Data Plane Development Kit* (*DPDK*) is an acceleration framework designed for fast packet processing.[7] The main advantage of using the *DPDK* to handle network I/O is the possibility to completely bypass the operating system kernel and directly access network interface queues [7]. Bypassing the kernel network stack removes overhead associated with sockets, system calls and buffer operations [7].

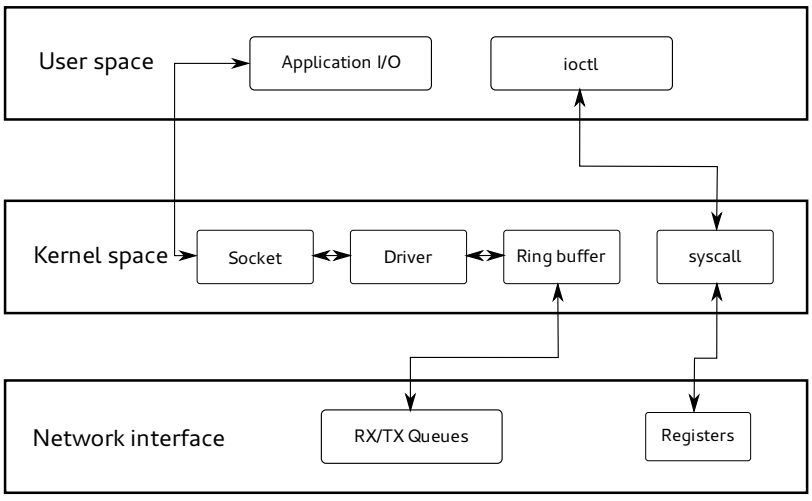

Figure 7.1: Kernel packet processing pipeline

[1] https://www.dpdk.org

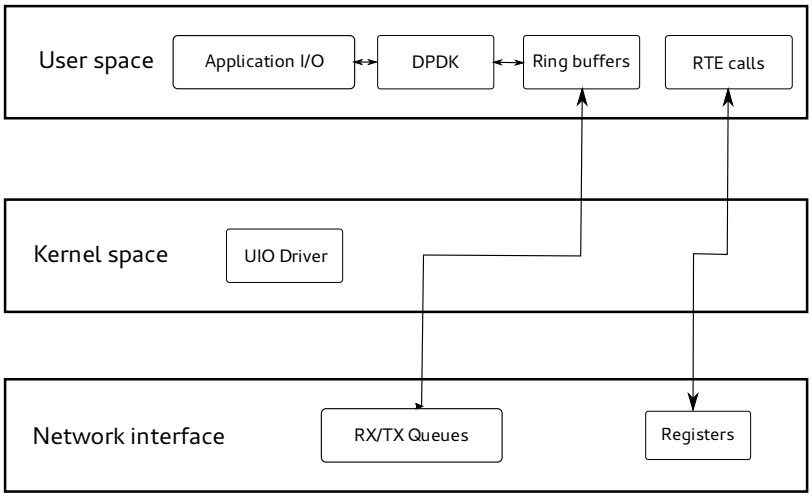

Figure 7.2: *DPDK* packet processing pipeline

Figures 7.1 and 7.2 illustrate the difference between the standard kernel packet processing pipeline and the *DPDK* pipeline. When using *DPDK*, the entire kernel network stack is bypassed and network interface queues are manipulated directly from the application using *DPDK* library methods. The userspace I/O (*UIO*) driver is used to expose the registers and queues of the NIC to the *DPDK* library by mapping them to the application's memory. The *DPDK* library contains userspace *Poll Mode Drivers* (*PMD*s) for specific NIC hardware. *PMD*s continuously poll *RX* queues of network interfaces for incoming packets; this removes overhead associated with interrupt-driven I/O.

7.2 Principle of Operation

The *DPDK TCP Generator* (*tcpgen*) application operates by using randomized source IPv4 and IPv6 addresses for establishing new TCP sessions with the target DNS server and using open sessions to send DNS queries. Responses sent by the server are classified in a fashion similar to the *STC Framework*, with the possibility to export results in JSON format.

Application Configuration

The majority of the possible *tcpgen* configuration options can be specified in a configuration file. The following configuration file sample demonstrates the possible configuration options:

```
source-mac de:ad:be:ef:ca:fe
destination-mac 90:e2:ba:00:00:01

ipv6-source-network fcaa:dead:beef:cafe::
ipv6-source-netmask ffff:ffff:ffff:ffff:ffff:ffff::
ipv6-destination-ip fcbb:1::1

ipv4-source-network 10.10.64.0
ipv4-source-netmask 255.255.192.0
ipv4-destination-ip 10.99.0.1
```

```
destination-port 53

ip-ipv6-probability 0.5
udp-probability 0.5
tcp-keepalive-probability 0.5
```

The individual configuration options have the following meaning:

- `source-mac` - Source MAC address used in frames sent to the server. The server's ARP and NDP tables must contain a corresponding entry for this address in order for the server to be able to send responses.
- `destination-mac` - Destination MAC address used in frames sent to the server. Must be set to the MAC address of the server's NIC.
- `ipv6-source-network` - Used together with `ipv6-source-netmask` to specify the subnet of randomized source IPv6 addresses used for establishing new TCP sessions.
- `ipv6-destination-ip` - Destination IPv6 address used in packets sent to the server. The server's network interface must be configured with this address.
- `ipv4-source-network` - Used together with `ipv4-source-netmask` to specify the subnet of randomized source IPv4 addresses used for establishing new TCP sessions.
- `ipv4-destination-ip` - Destination IPv4 address used in packets sent to the server. The server's network interface must be configured with this address.
- `destination-port` - Destination port used in datagrams sent to the server. By default, port 53 (DNS) is used.
- `ip-ipv6-probability` - Probability of any given new TCP session being opened over IPv6 or UDP query being sent over IPv6 specified in the interval $< 0, 1 >$.
- `udp-probability` - Probability of sending a UDP query instead of opening any given new TCP session specified in the interval $< 0, 1 >$.
- `tcp-keepalive-probability` - Probability of reusing an open TCP session to send another query instead of closing the session and opening a new session specified in the interval $< 0, 1 >$.

Other configuration options are specified as command line arguments:

```
tcpgen [EAL options] -- -p PORTMASK -c CONFIG --pcap PCAP
       [-g TCP_GAP] [-r RUNTIME] [--results RESULTS]
```

- `PORTMASK` - Specifies which DPDK-enabled network interfaces will be used for generating traffic.
- `CONFIG` - Path to the configuration file.
- `PCAP` - Path to the PCAP file with reference queries. The PCAP file must contain standard UDP DNS queries. *tcpgen* will strip L2-L4 headers and use the DNS payloads in generated queries.

- `TCP_GAP` - Specifies the minimum gap between opening new TCP sessions.
- `RUNTIME` - Stop generating queries after the specified runtime.
- `RESULTS` - Path to file containing response rate statistics in JSON format.

Target Server Network Setup

In order for *tcpgen* to function correctly, the network interface on the target DNS server needs to be set up to correctly reflect the subnet configuration specified in the configuration file. IPv4 and IPv6 addresses of the server's network interface cannot overlap with the subnets specified by the respective (`ipv4-source-network`, `ipv4-source-netmask`) and (`ipv6-source-network`, `ipv6-source-netmask`) pairs; doing so would cause ARP and NDP lookups to be performed for received queries. Instead, the server's interface must reside in a disjoint subnet and the server's routing table must route the source *tcpgen* subnets via an IP address on the same subnet as the server's network interface.

Any IP address in the same subnet as the server's network interface can be used as the next hop; it is, however, necessary to create corresponding ARP and NDP table entries mapping the IP address to the `source-mac` configured in *tcpgen*'s configuration file. This will cause the server to send all responses over the network interface connected to *tcpgen* with the specified destination IP address. *tcpgen* will ignore the destination IP address and classify all non-malformed DNS responses into response statistics.

Reference PCAP File

Unlike the *STC Framework*, *tcpgen* does not contain a library for crafting DNS queries, but relies on a PCAP file which contains reference queries with the desired payload. The PCAP file must contain standard, valid UDP DNS queries. *tcpgen* will parse the PCAP file, strip L2, L3 and L4 headers and store the payloads of DNS queries in memory. When queries are sent over TCP, the DNS header is modified to contain the length byte.

When generating queries, *tcpgen* opens TCP sessions from randomized source IP addresses over both IPv4 and IPv6, based on the parameters specified in the configuration file. The DNS query payloads from the reference PCAP file are then inserted into the appropriate section of the generated frames. As reference PCAP payloads are stored in a cyclic linked-list structure and selected in a round-robin fashion, it is possible to use small reference PCAP files when generating queries at a high bitrate.

Query Stream Parameters

The parameters of the stream of queries generated by *tcpgen* depend on the supplied configuration options. By default, only TCP queries are sent with a $1:1$ IPv4 to IPv6 ratio, assuming both IPv4 and IPv6 configuration options are supplied in the configuration file. Unless the `tcp-keepalive-probability` option is specified, TCP keepalive is not enabled and only one query per TCP session is sent.

The `udp-probability` option enables mixing in regular UDP queries. If a query is to be sent over UDP, *tcpgen* will not attempt to establish a TCP session and send a regular UDP query. UDP responses are also processed and factored into response rate statistics.

7.3 Architecture

The *tcpgen* application is multi-threaded and can be used with multiple network interfaces. The number of threads used is specified by the `coremask` *DPDK EAL* argument. Enabled interfaces are specified by the `portmask` argument. One *TX* and *RX* queue is set up for each thread on every enabled interface.

After launch, once *EAL* arguments are successfully parsed, the master thread parses configuration options and the reference PCAP file. A global configuration structure shared across all threads is initialized. Response rate statistics are kept separately for every thread and aggregated when traffic generation is stopped.

Slave threads are started once initialization on the master thread completes. Every thread, including the master, then follows the same operating pipeline; new TCP connections are opened with a delay of `TCP_GAP`, unless UDP queries are sent instead. Upon receiving a *SYN-ACK* datagram, an *ACK* datagram is sent to complete the three-way handshake, along with the current query payload in the reference PCAP linked list. The reference PCAP payload data is read-only and shared between all threads; every thread, however, keeps a local linked list structure over the reference PCAP payload data, which preserves the same order of queries sent by each respective thread.

Incoming frames are distributed between threads using a receive-side scaling (*RSS*) hash function, which uses the <`SRC_IP`, `SRC_PORT`, `DST_IP`, `DST_PORT`> tuple to compute the value of a hash associated with a given frame. The *RSS* hash function used in *tcpgen* is not symmetrical, but since *tcpgen* does not rely on an internal state to keep track of TCP sessions, a query can be sent over an open session by a different thread than the one which opened the session.

When `tcp-keepalive-probability` is set, existing sessions are reused with the configured probability. A new query is sent over a reused session with a delay of `TCP_GAP`. Additionally, *tcpgen* will skip opening a new TCP session for every session that is reused. The number of simultaneously opened TCP sessions is limited to ensure source IP address and port randomization even when a high `tcp-keepalive-probability` is set. If a session fails the die-roll and is not reused, it is immediately closed and a *FIN* flag is sent to the server with the last query.

Chapter 8

Authoritative Nameserver Benchmarking

This chapter contains benchmarking results of three authoritative nameserver implementations: *Knot-DNS 2.8.4*, *NSD 4.2.2* and *BIND 9.14.6*. All benchmarks were conducted on the same hardware setup, both with regards to the target nameserver and the traffic generator. The same benchmarking methodology was used in all scenarios to create comparable *performance* profiles.

8.1 Hardware Setup

The nameserver implementations were tested on an Intel Core i7-6700 machine clocked at 3.40GHz with hyperthreading enabled with 16GiB of physical memory. The CPU scaling governor for all eight logical cores was set to *performance* mode. Affinity of per-core NIC queue IRQs was manually distributed across all eight logical cores, with each core handling a single IRQ.

Knot-DNS and *BIND* were permitted to automatically determine the number of worker threads. As this feature is not supported in *NSD*, the number of workers was manually set to 7, as this is was the value automatically selected by both *Knot-DNS* and *NSD*.

8.2 Methodology Summary

A common query scenario set S was selected for all benchmarks. The scenarios $s_1, s_2, ..., s_n$ vary in the count of queried *QNAME*s, which ranges from a single *QNAME* to 1 million unique queried *QNAME*s. The scenarios also vary in the types of records queried (*A*, *AAAA*, *MX*), the size of the source IPv4 and IPv6 subnets, the size of the source port ranges and the size of source DNS message ID range.

8.3 UDP Benchmarking

The following selection of test scenarios consists of homogeneous UDP traffic. All test scenarios consist of 1 million randomized queried *QNAME*s. The existence of the corresponding *A*, *AAAA* and *MX* records in the zone files of the given authoritative server is guaranteed.

Knot-DNS 2.8.4

Single-client scenario

No randomization of the DNS message ID, source IP addresses or source port occurs, but *QNAME*s are still randomized. This scenario is designed to prevent the operating system from load-balancing incoming traffic. As a result, the resulting response rate statistics effectively demonstrate single-thread performance of the nameserver implementation.

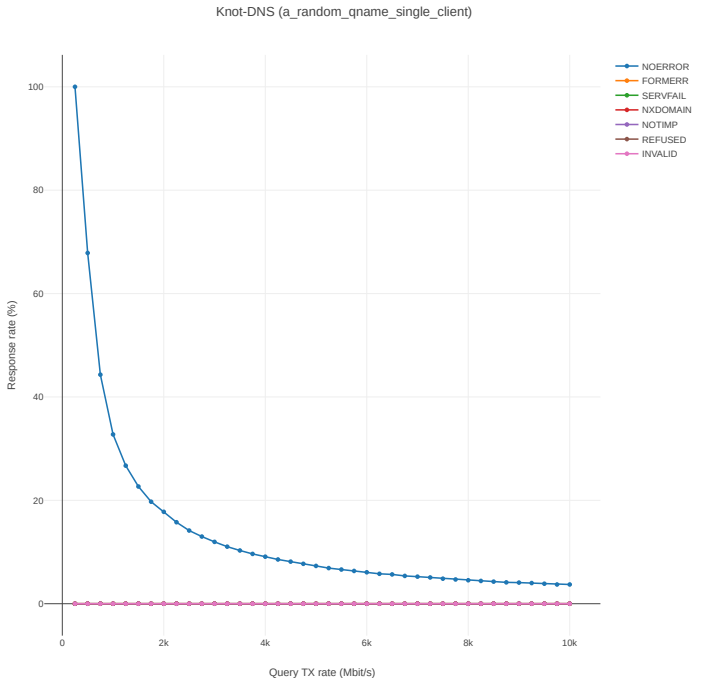

Figure 8.1: Random *A* queries in single-client scenario response rate

Figure 8.1 demonstrates the proportion of resolved queries for UDP query TX rates from 0 to 10 Gbit/s. Figure 8.2 demonstrates the relation between queries per second and responses per second.

Figures 8.3 and 8.4 demonstrate the same response rate statistics for *AAAA* queries, while Figures 8.5 and 8.6 demonstrate the same response rate statistics for *MX* queries.

Random client scenario

Unlike in the single client scenario, DNS message ID, source IP addresses and source ports are randomized in addition to *QNAME*s. In this scenario, traffic is load-balanced across threads in a uniform fashion. Figures 8.7 and 8.8 demonstrates the proportion of resolved queries for UDP query TX rates from 0 to 10 Gbit/s and the relation between queries per second and responses per second respectively. Figures 8.9 and 8.10 demonstrate the same for *AAAA* queries and Figures 8.11 and 8.12 for *MX* queries.

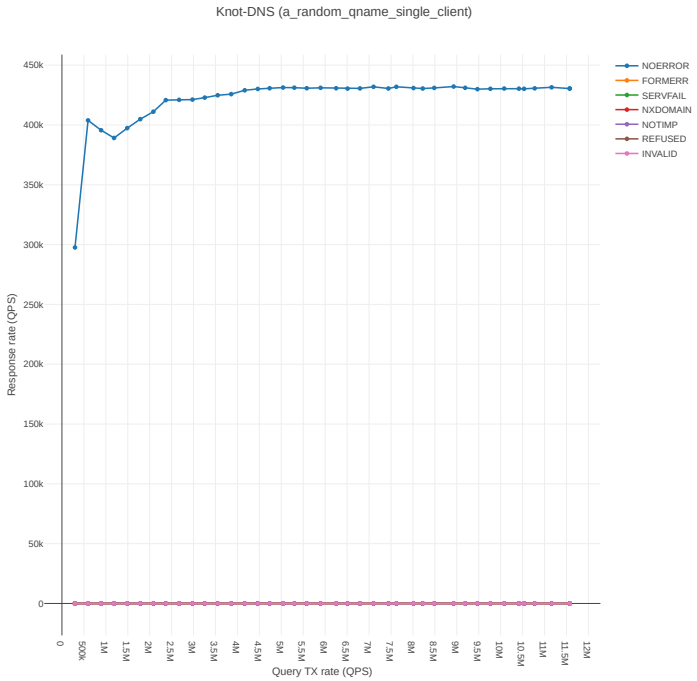

Figure 8.2: Random A queries in single-client scenario QPS to RPS rate

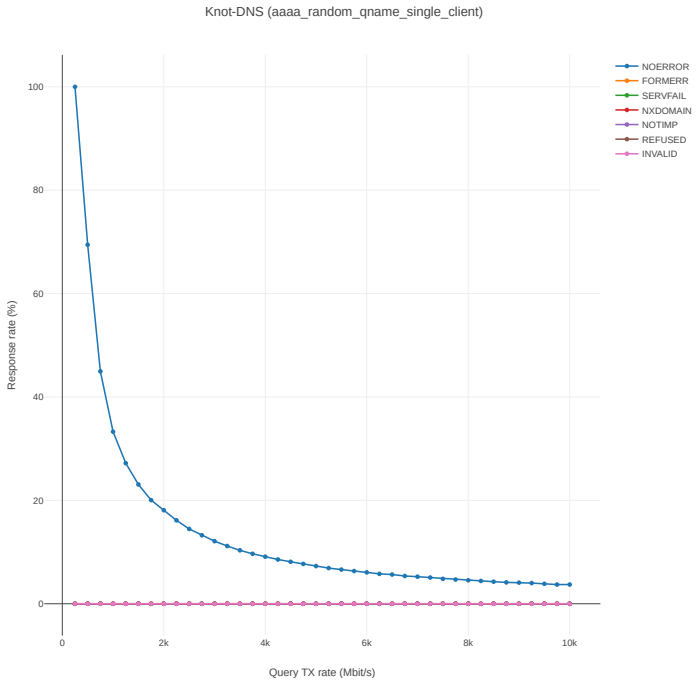

Figure 8.3: Random $AAAA$ queries in single-client scenario response rate

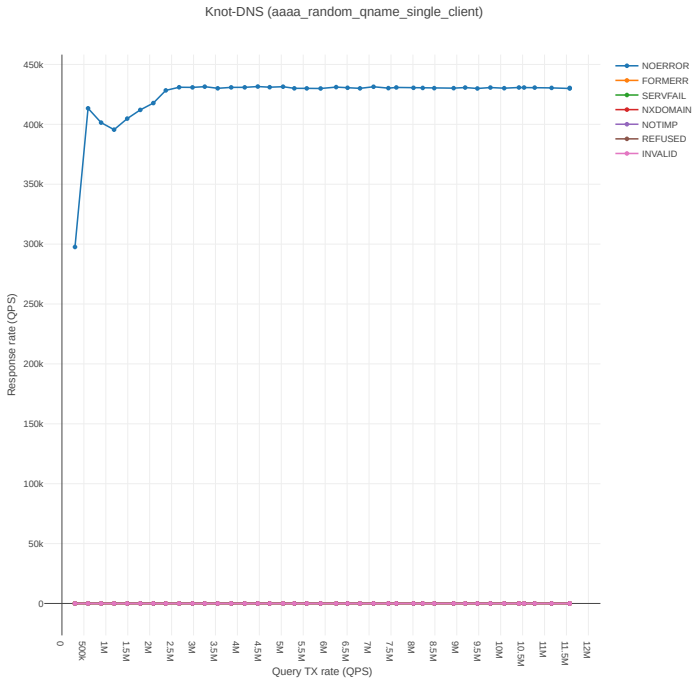

Figure 8.4: Random $AAAA$ queries in single-client scenario QPS to RPS rate

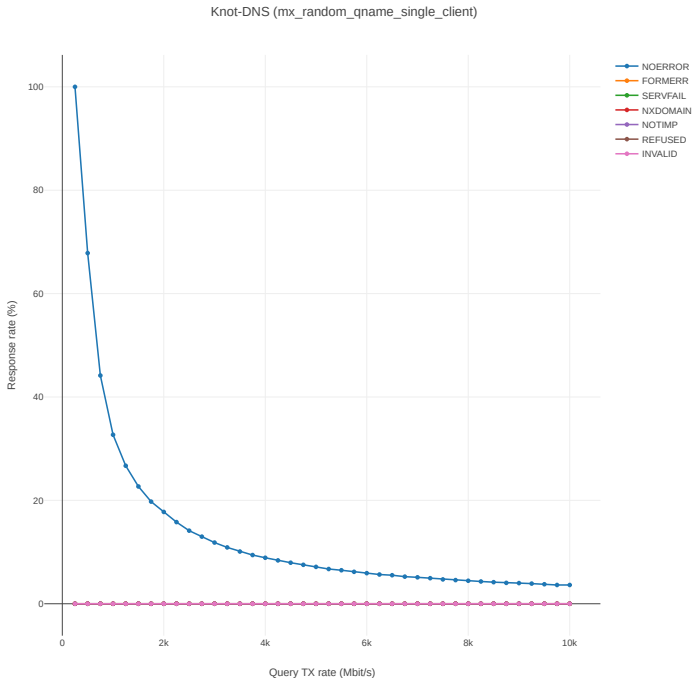

Figure 8.5: Random MX queries in single-client scenario response rate

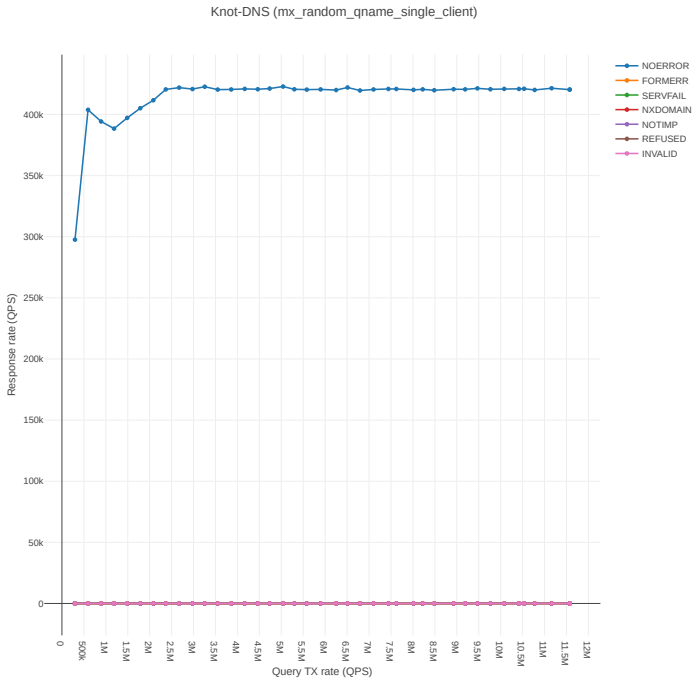

Figure 8.6: Random *MX* queries in single-client scenario QPS to RPS rate

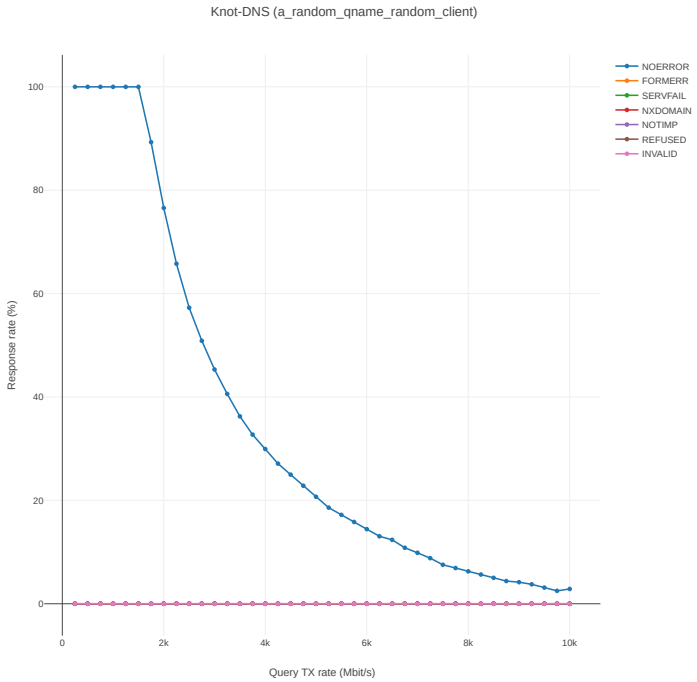

Figure 8.7: Random *A* queries in random-client scenario response rate

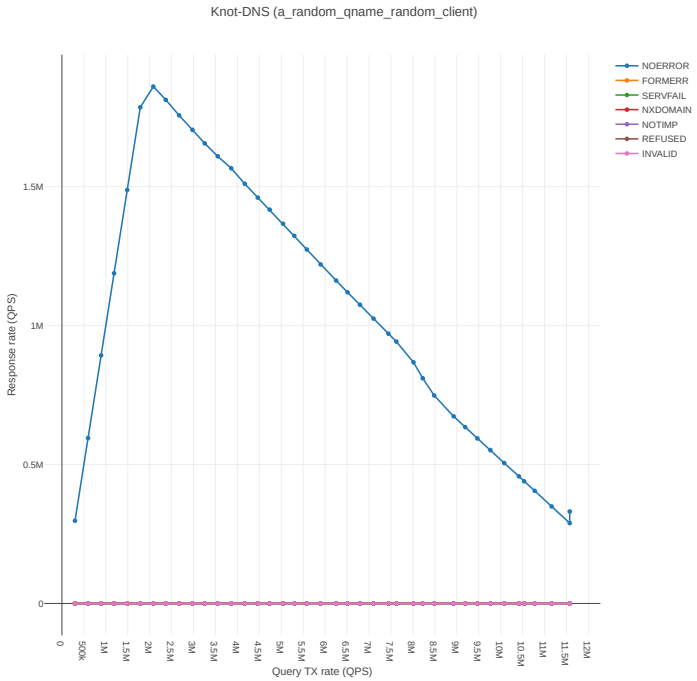

Figure 8.8: Random A queries in random-client scenario QPS to RPS rate

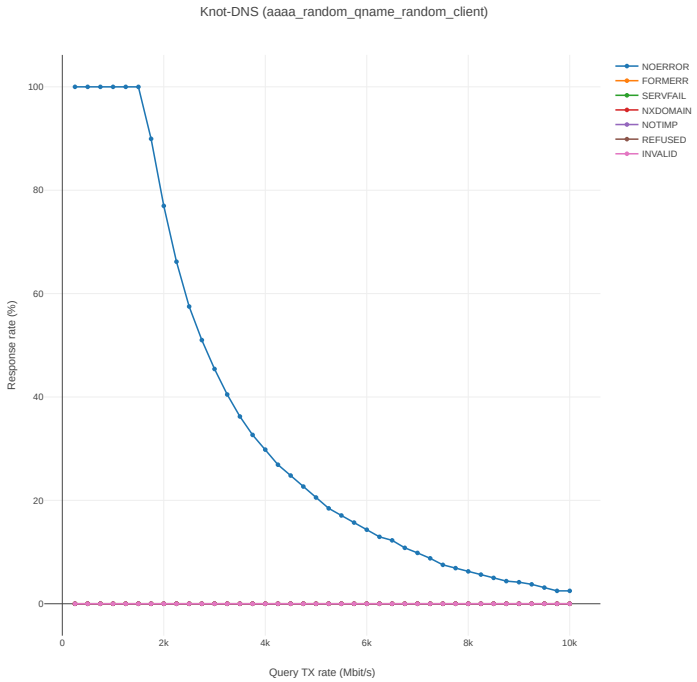

Figure 8.9: Random $AAAA$ queries in random-client scenario response rate

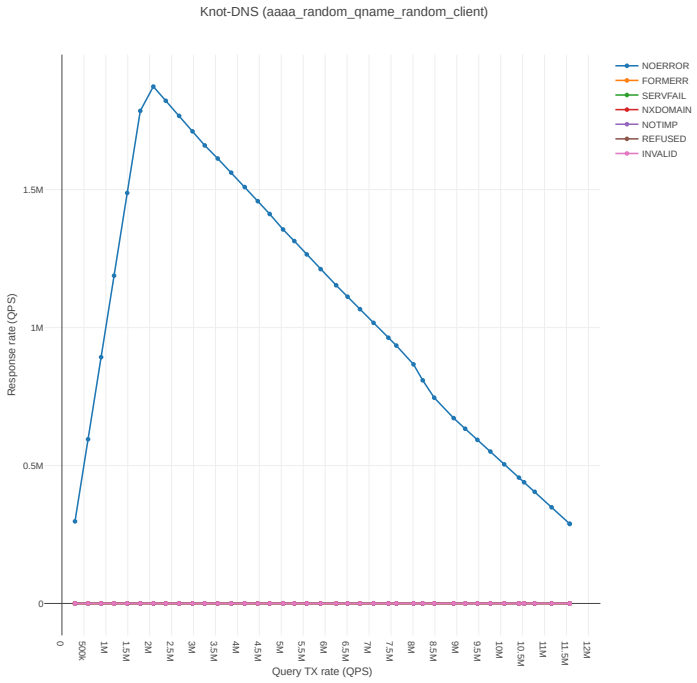

Figure 8.10: Random *AAAA* queries in random-client scenario QPS to RPS rate

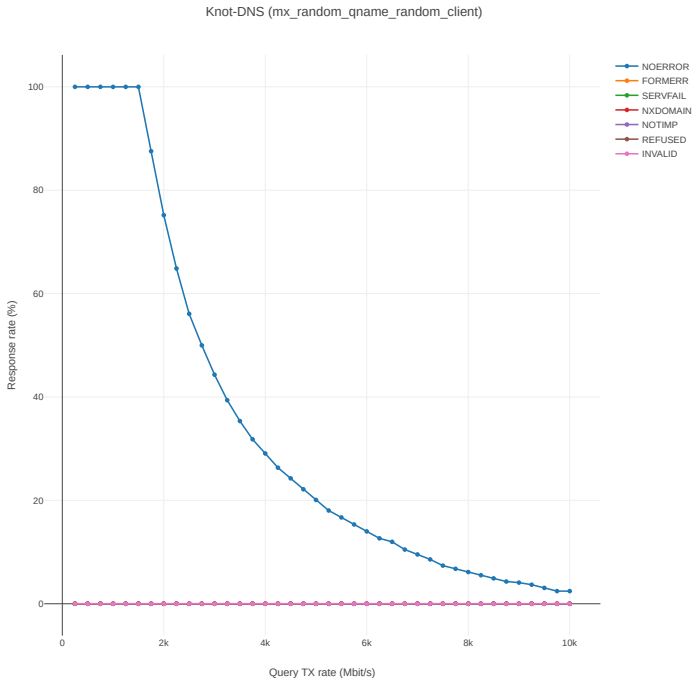

Figure 8.11: Random *MX* queries in random-client scenario response rate

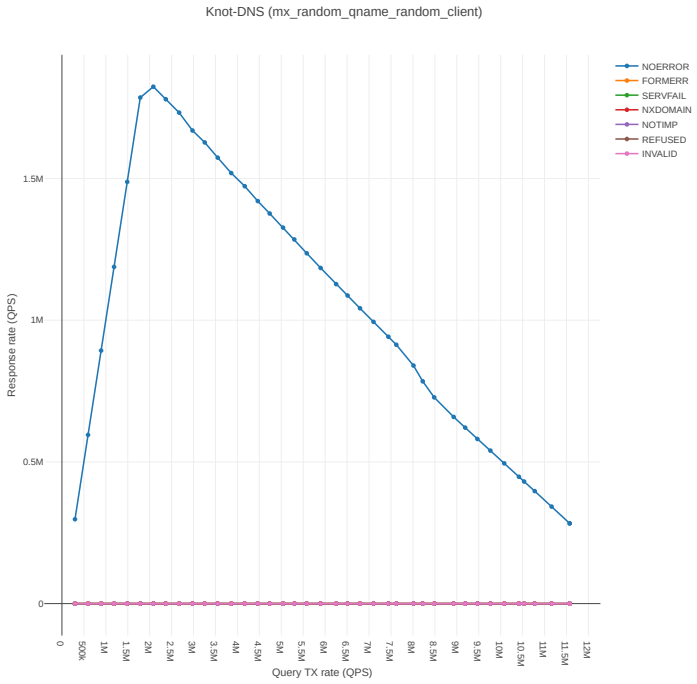

Figure 8.12: Random *MX* queries in random-client scenario QPS to RPS rate

NSD 4.2.2

The same scenarios that were used with *Knot-DNS* were tested with *NSD*. Since there is no significant variance in response rates for *A*, *AAAA* and *MX* queries, only *AAAA* query response rate graphs are presented.

Single-client scenario

Single-client scenario resulsts are presented in Figures 8.13 and 8.14.

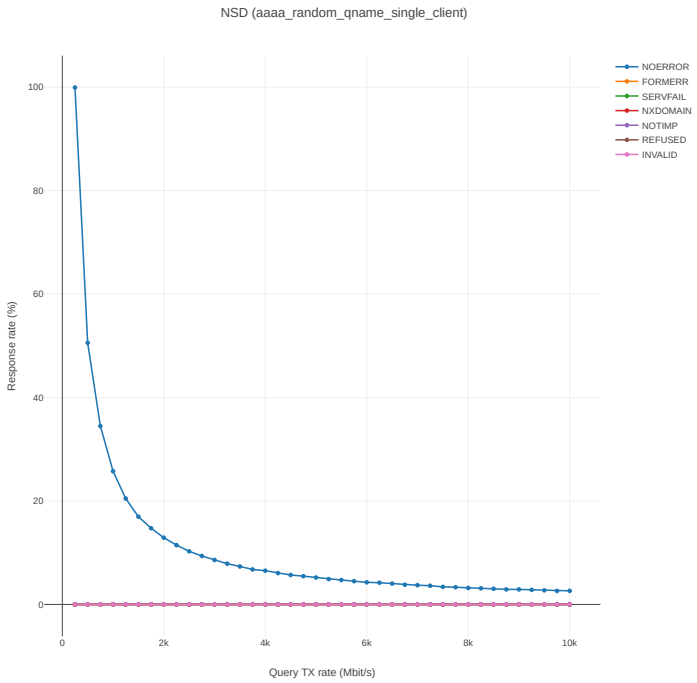

Figure 8.13: Random *AAAA* queries in single-client scenario response rate

Random client scenario

Random client scenario results are presented in Figures 8.15 and 8.16.

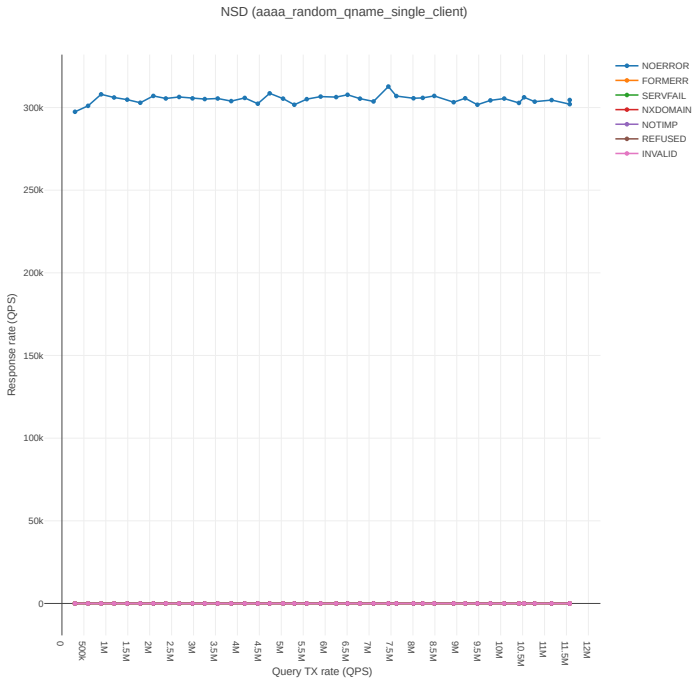

Figure 8.14: Random $AAAA$ queries in single-client scenario QPS to RPS rate

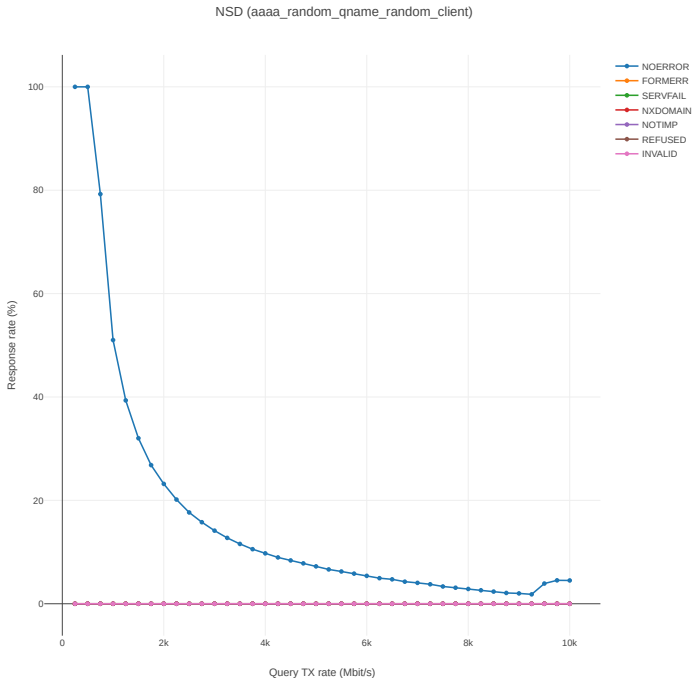

Figure 8.15: Random $AAAA$ queries in random client scenario response rate

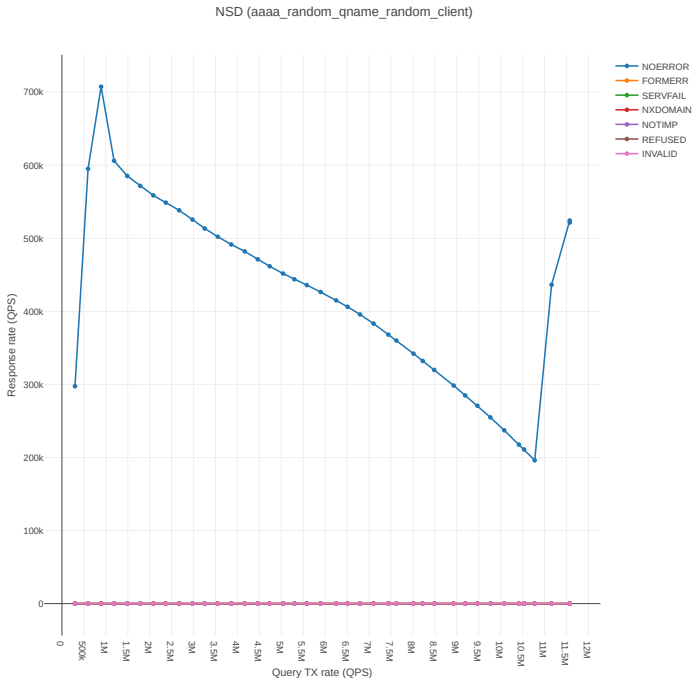

Figure 8.16: Random $AAAA$ queries in random client scenario QPS to RPS rate

BIND 9.14.6

Due to *BIND* being the worst-performing implementation by a significant margin, only random client scenario results for *AAAA* queries are presented.

Random client scenario

Random client scenario results are presented in Figures 8.17 and 8.18.

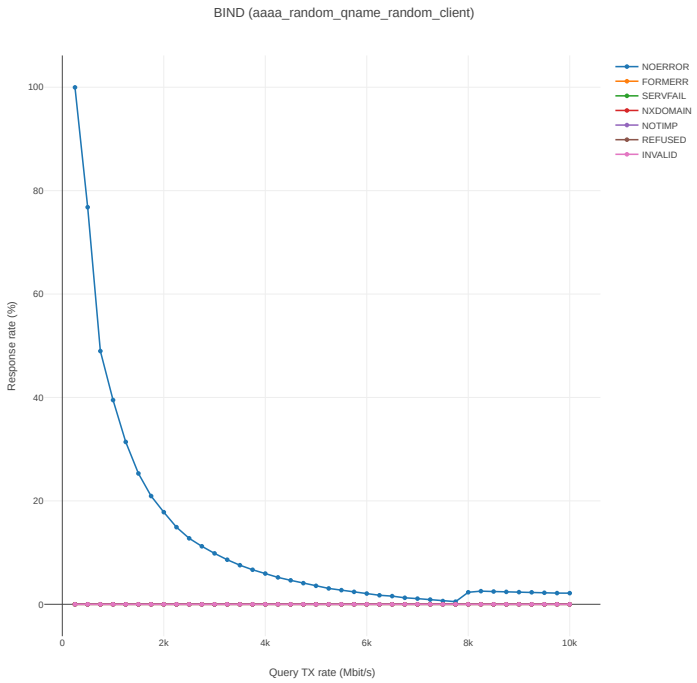

Figure 8.17: Random *AAAA* queries in random client scenario response rate

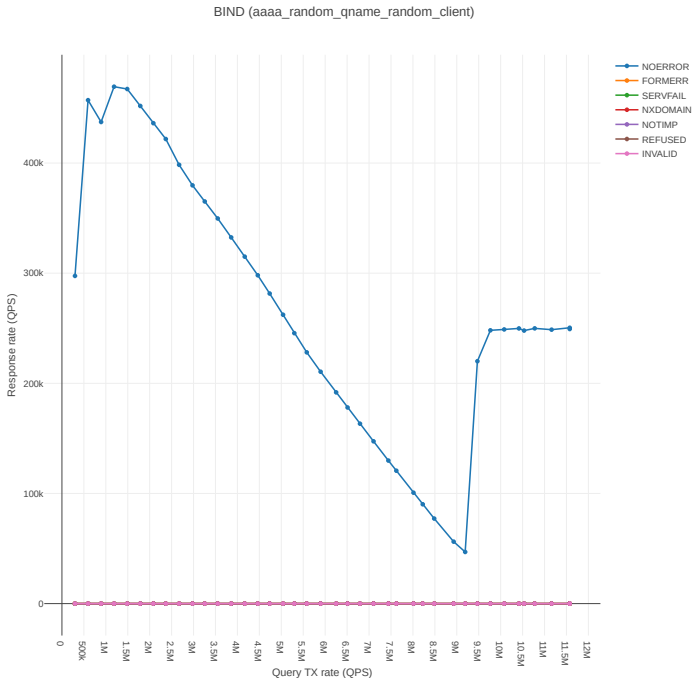

Figure 8.18: Random $AAAA$ queries in random client scenario QPS to RPS rate

Performance Comparison

Graphs in this section contain combined response rates of the tested implementations to both random and single client query test scenarios.

Random client scenarios

For completeness, response graphs of all query types are shown, but there is almost no variance between response rates to different types of queries (A, $AAAA$, MX). Comparison of the response to A queries is shown in Figures 8.19 and 8.20. $AAAA$ queries are shown in Figures 8.21 and 8.22 and MX queries are shown in Figures 8.23 and 8.24.

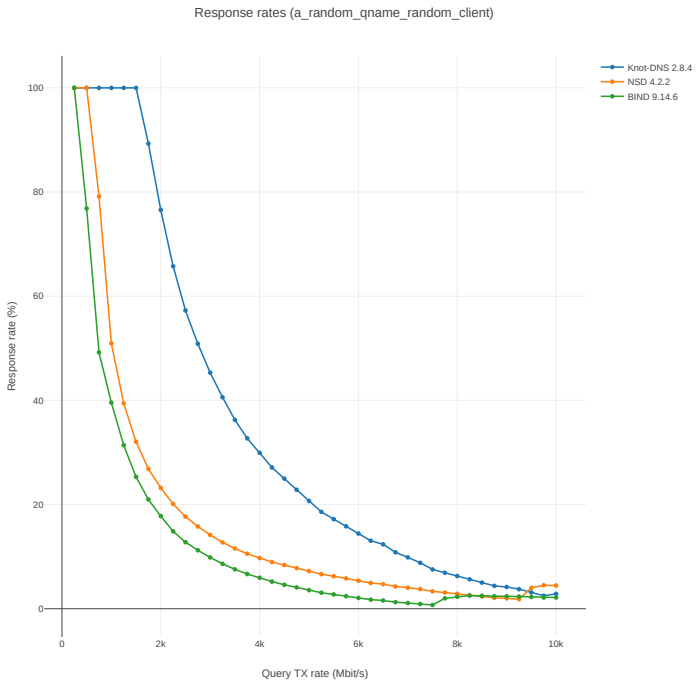

Figure 8.19: Random A queries in random client scenario response rate

Single-client scenarios

Single-client response rate comparisons are showcased for $AAAA$ queries in Figures 8.25 and 8.26.

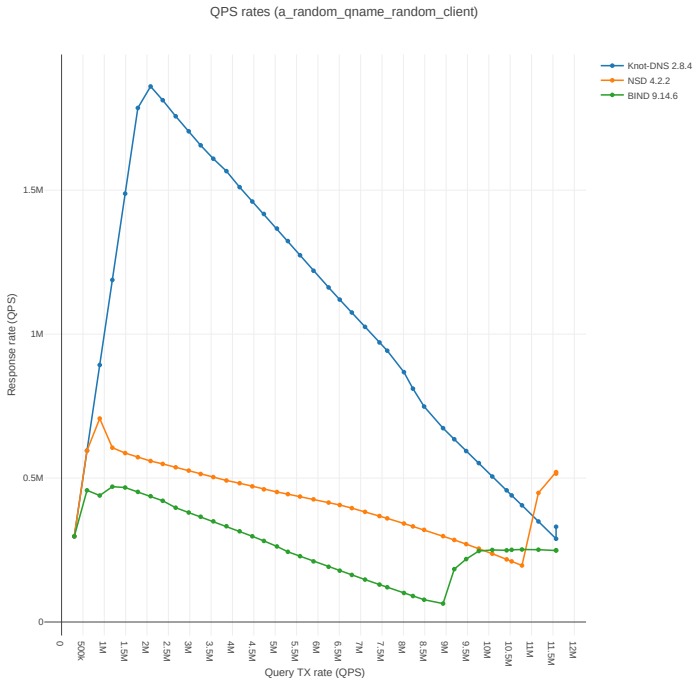

Figure 8.20: Random A queries in random client scenario QPS to RPS rate

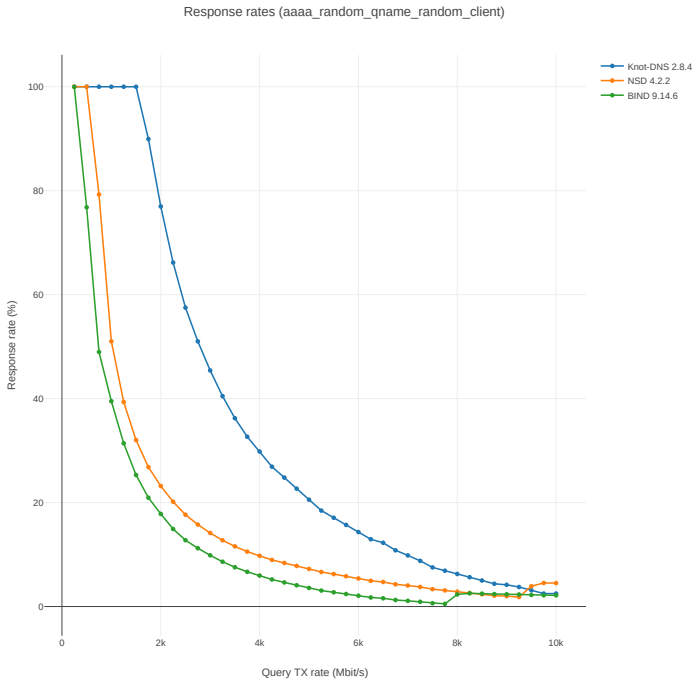

Figure 8.21: Random $AAAA$ queries in random-client scenario response rate

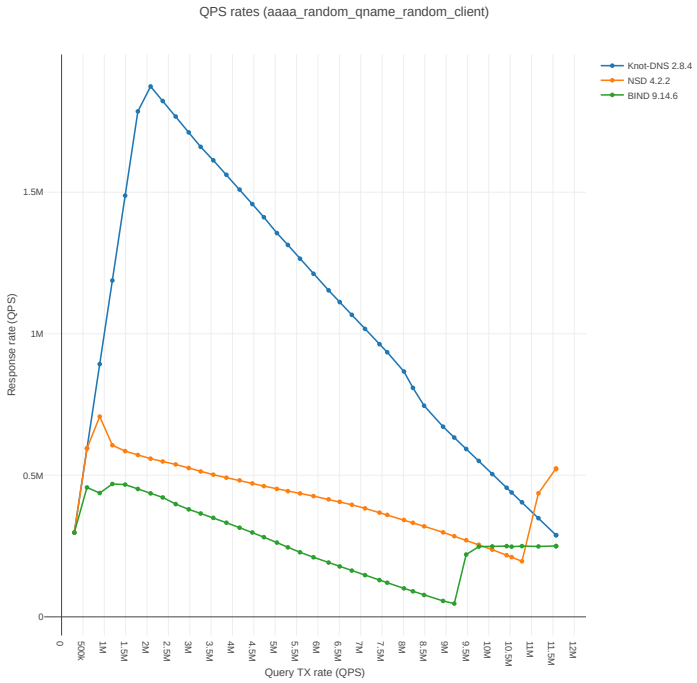

Figure 8.22: Random *AAAA* queries in random-client scenario QPS to RPS rate

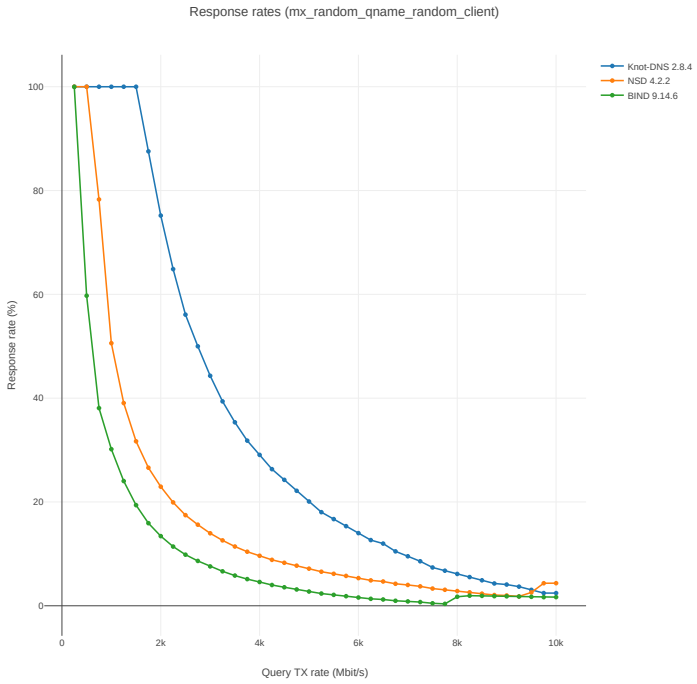

Figure 8.23: Random *MX* queries in random-client scenario response rate

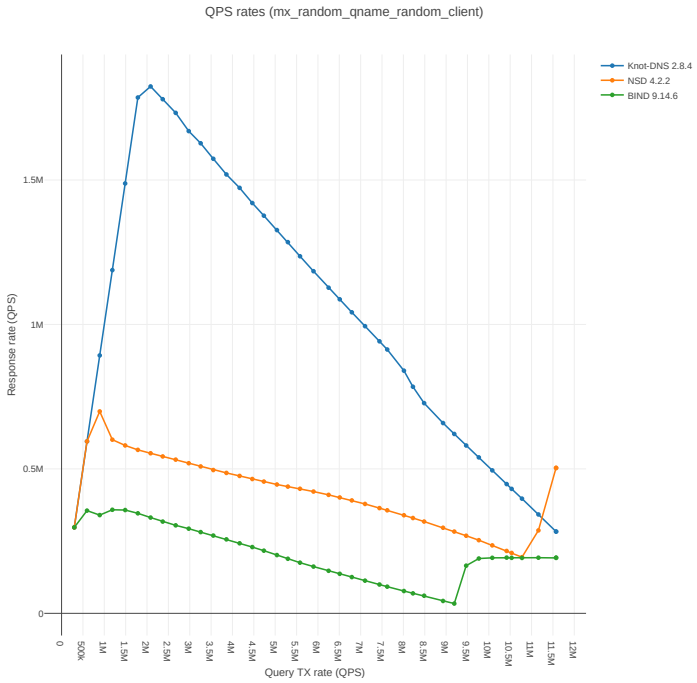

Figure 8.24: Random *MX* queries in random-client scenario QPS to RPS rate

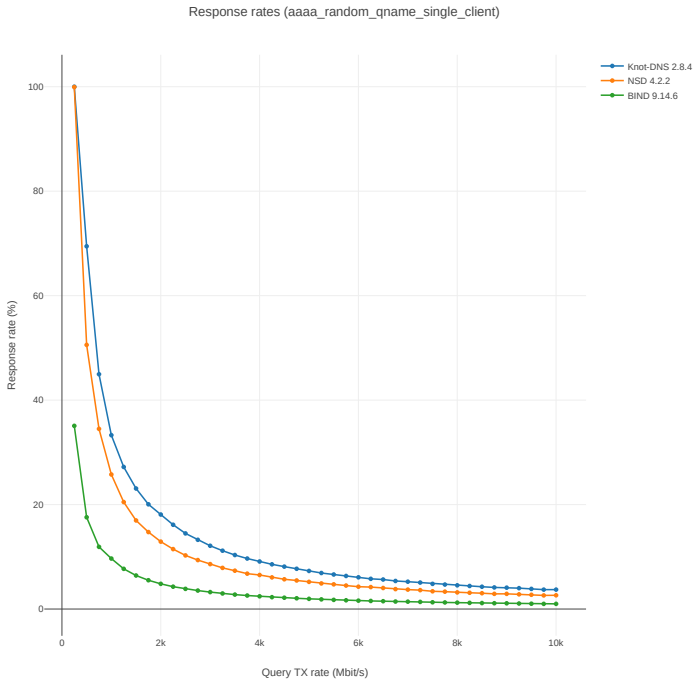

Figure 8.25: Random *AAAA* queries in single-client scenario response rate

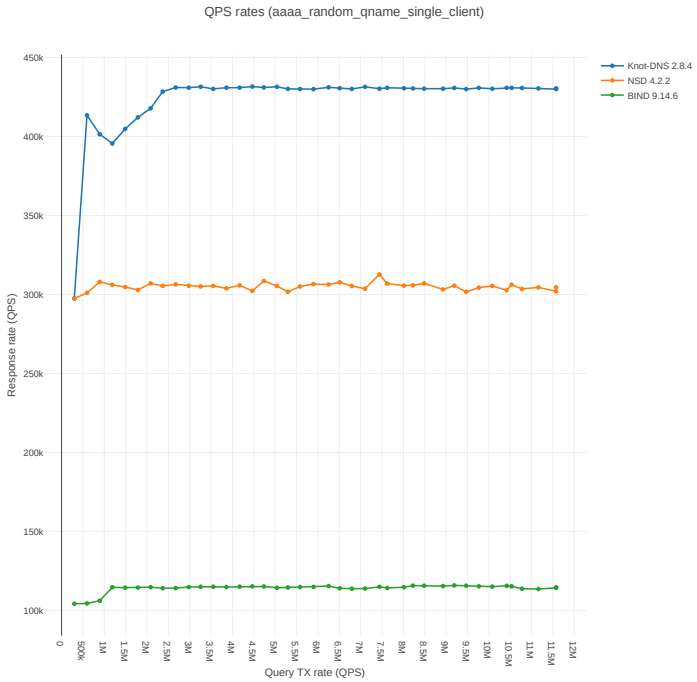

Figure 8.26: Random $AAAA$ queries in single-client scenario QPS to RPS rate

8.4 TCP Benchmarking

Figure 8.27 shows the response of *Knot-DNS* to a random-client *A* query scenario over TCP. The graph demonstrates the difference between the frames generated per second and received per second at a given bitrate. In an ideal scenario, frame counts should show the same trend regardless of the bitrate. The disparity at bitrates over 600 Mbit/s shows a massive drop in the server's ability to handle TCP session overhead and answer queries when flooding the server with a high volume of TCP traffic.

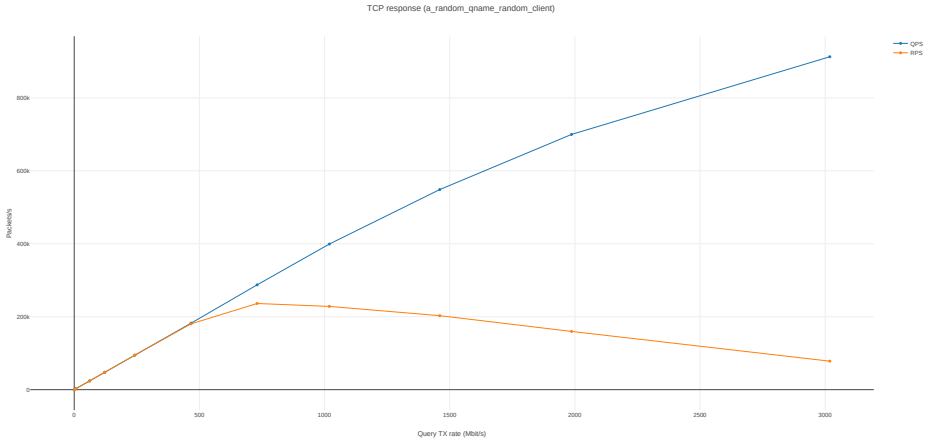

Figure 8.27: *Knot-DNS A* query TCP response rate

The showcased TCP benchmark scenario shows a significant performance drop when attempting to open a higher volume of TCP connections. Both *NSD* and *BIND* show the same trend, with only minor performance differences between implementations. Profiling carried out by CZ.NIC shows that there is severe lock contention in the Linux kernel when opening and closing TCP connections, which is currently the biggest bottleneck of DNS over TCP.[8]

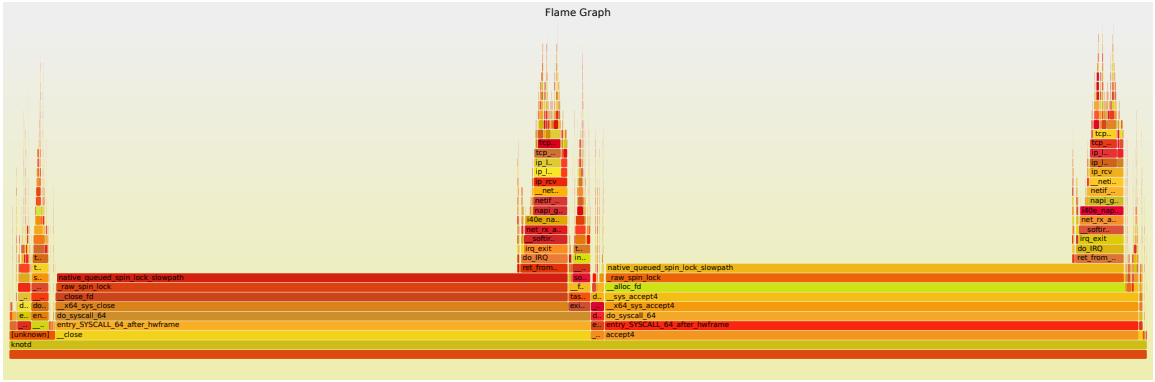

Figure 8.28: TCP connection lock contention in the Linux kernel[8]

Chapter 9

Conclusion

The showcased UDP benchmarks are composed of test scenarios which attempt to emulate realistic DNS traffic by randomizing L3, L4 and L7 elements of DNS packets. While it is possible to create more exhaustive test scenarios which diversify outgoing traffic even further, *Knot-DNS* is clearly superior with regards to performance in the showcased test scenarios. It cannot, however, be said that the performance P of *Knot-DNS* is in the $>>$ relation with the performance of other implementations, as both *BIND* and *NSD* outperform *Knot-DNS* in certain test scenarios. The performance of DNS over TCP was limited by issues in the Linux kernel rather than limitations of individual implementations.

The *tcpgen* tool is available on *github*[1] and was used by CZ.NIC to conduct TCP benchmarks of *Knot-DNS*[8]. A number of potential improvements to both *tcpgen* and the *STC Framework* are possible; one such improvement would be adding support for further customization of the stream of generated queries.

[1] https://github.com/thewhoo/dpdk-tcp-generator

Bibliography

[1] *Spirent TestCenter Automation Programmer's Reference*. 26750 Agoura Road, Calabasas, CA 91301, USA: Spirent Communications, Inc., september 2014.

[2] Arends R., L. M. M. D. R. S. *Protocol Modifications for the DNS Security Extensions*. RFC 4035. DOI 10.17487/RFC4035, march 2005. Available at: https://www.rfc-editor.org/info/rfc4035.

[3] Kröhnke, L., Jansen, J. and Vranken, H. Resilience of the Domain Name System: A case study of the .nl-domain. *Computer Networks*. Elsevier B.V. 2018, vol. 139, p. 136–150. ISSN 1389-1286.

[4] Liu, C. *DNS and BIND*. 5th ed.th ed. Beijing: O'Reilly, 2006. Help for system administrators. ISBN 0-596-10057-4.

[5] P., M. *Domain names - implementation and specification*. RFC 1035. DOI 10.17487/RFC1035, november 1987. Available at: https://www.rfc-editor.org/rfc/rfc1035.txt.

[6] R., B. *DNS Multiple QTYPEs*. IETF Draft. July 2017. Available at: https://tools.ietf.org/id/draft-bellis-dnsext-multi-qtypes-04.html.

[7] Rosen, R. *Network Acceleration With DPDK*. [Online; visited 12.06.2020]. Available at: https://lwn.net/Articles/725254.

[8] Salzman, D. Epic DNS Benchmarking. *The CZ.NIC Staff Blog*. CZ.NIC. 2019. Available at: https://en.blog.nic.cz/2019/12/10/epic-dns-benchmarking/. ISSN 2533-4727.

[9] Thomson S., K. V. and Souissi, M. *DNS Extensions to Support IP Version 6*. RFC 3596. DOI 10.17487/RFC3596, october 2003. Available at: https://www.rfc-editor.org/info/rfc3596.