Project 2 DNS Server

Weight: 15% of the final mark

Due date: No later than 3:00 pm, Thursday 20 May, 2021 AEST.

Phases

This project should be executed in two phases: one starting now and one in a week's time. This document describes both phases; there is only one deliverable

Phase 1 of this project is to design and write as much code as possible without using sockets. This involves looking up the reference and other resources to understand the format of DNS messages, formatting of log entries (with timestamps), and thinking about how to achieve caching and non-blocking if those options are chosen. Example DNS requests and replies are avilable at comp30023-2021-projects/project-2 on Gitlab. It is strongly recommended that you get Phase 1 working by the *end of Week 9*.

Phase 2 is to write the socket code, and integrate it with the code written in Phase 1.

Why is it structured as two phases?

All the material for Phase 1 has already been covered in lectures. Hence, you can immediately start work on this phase. However, Phase 2 uses socket programming, which will not be covered until week 9.

The project is big enough that it is important to start it now, instead of leaving it until week 9.

If all this is confusing, you can treat the project as a single project, and accept the fact that sockets will not be covered in lectures for a while.

1 Background

Note: This project will contain some reading of standards as well as writing code. If you ever write an implementation of a protocol, you will need to read standards such as these. Therefore, becoming familiar with the format and terminology is an important part of the field of computer systems. You will be pointed to the relevant sections so that you do not spend your whole time reading the more arcane parts of the text.

The Domain Name System (DNS) provides, among other things, the mapping between human-meaningful hostnames like `lms.unimelb.edu.au` and the numeric IP addresses that indicate where packets should be sent. DNS consists of a hierarchy of servers, each knowing a portion of the complete mapping.

In this project, you will write a DNS server that accepts requests for IPv6 addresses and serves them either from its own cache or by querying servers higher up the hierarchy. Each transaction consists of at most four messages: one from your client to you, one from you to your upstream server, one from your upstream server to you and one from you to your client. The middle two can be sometimes skipped if you cache some of the answers.

The format for DNS request and response messages are described in [1].

In a DNS system, the entry mapping a name to an IPv6 address is called a AAAA (or "quad A") record [2]. Its "record type" is 28 (`QType` in [2]).

The server will also keep a log of its activities. This is important for reasons such as detecting denial-of-service attacks, as well as allowing service upgrades to reflect usage patterns.

For the log, you will need to print a text version of the IPv6 addresses. IPv6 addresses are 128 bits long. They are represented in text as eight colon-separated strings of 16-bit numbers expressed in hexadecimal. As a shorthand, a string of consecutive 16-bit numbers that are all zero may be replaced by a single "::". Details are in [3].

2 Project specification

Task: Write a miniature DNS server that will serve AAAA queries.

This project has three variants, with increasing levels of difficulty. It is not expected that most students will complete all tasks; the hard ones are to challenge those few who want to be challenged.

It is expected that about half the class will complete the Standard option (and do it well – you can get an H1 by getting full marks on this), a third to complete the Cache option, and a sixth to complete the Non-blocking option. If you think the project is taking too much time, make sure you are doing the Standard option.

You should create a Makefile that produces an executable named '`dns_svr`'.

Submission will be through git and LMS, like the first project.

2.1 Standard option

Accept a DNS "AAAA" query over TCP on port 8053. Forward it to a server whose IPv4 address is the *first* command-line argument and whose port is the *second* command-line argument. (For testing, use the value in `/etc/resolv.conf` on your server and port 53). Send the response back to the client who sent the request, over the *same TCP connection*. There will be a *separate* TCP connection for each query/response with the client. Log these events, as described below.

Note that DNS usually uses UDP, but this project will use TCP because it is a more useful skill for you to learn. A DNS message over TCP is slightly different from that over UDP: it has a two-byte header that specify the length (in bytes) of the message, not including the two-byte header itself [4, 5]. This means that you know the size of the message before you read it, and can malloc() enough space for it.

Assume that there is only one question in the DNS request you receive, although the standard allows there to be more than one. If there is more than one answer in the reply, then only log the first one, but always reply to the client with the entire list of answers. If there is no answer in the reply, log the request line only. If the first answer in the response is not a AAAA field, then do not print a log entry (for *any* answer in the response).

The program should be ready to accept another query as soon as it has processed the previous query and response. (If Non-blocking option is implemented, it must be ready before this too.)

Your server should not shut down by itself. `SIGINT` (like CTRL-C) will be used to terminate your server between test cases.

You may notice that a port and interface which has been bound to a socket sometimes cannot be reused until after a timeout. To make your testing and our marking easier, please override this behaviour by placing the following lines before the `bind()` call:

```
int enable = 1;
if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int)) < 0) {
  perror("setsockopt");
  exit(1);
}
```

2.2 Cache option

As above, but cache the five (5) most recent answers to queries. Answer directly if possible, instead of querying the server again.

Note that the ID field of the reply must be changed for each query so that the client doesn't report an error. The time-to-live (TTL) field of the reply must also be decremented by the amount of time which has passed.

Note also that cache entries can expire. A cache entry should not be used once it has expired and a new query should be sent to the recursive server.

If a new answer is received when the cache contains an expired entry, the expired entry (or another expired entry) should be overwritten. If there is no expired entry, any cache eviction policy is allowed.

Do not cache responses that do not contain answers (i.e., the address was not found). If multiple answers are returned, only cache the first one.

If you implementing caching, include the line "`#define CACHE`" in your code. Otherwise, this functionality will not be tested/marked.

2.3 Non-blocking option

It can sometimes take a while for the server that was queried to give a response. To perform a recursive DNS lookup, many servers may need to be contacted (when starting from an empty cache, from a root DNS server down to an authoritative DNS server); any of them may be congested. Meanwhile, another request may have arrived, which the server cannot respond to if it was blocked by the completion of the first request.

This option extends both options above and mitigates this problem, by enabling the processing of new requests while waiting for prior requests to complete.

This may be done using multi-threading, or `select(3)/epoll(7)`. Using multithreading may require explicit locking of shared resources, such as the cache, whereas a single threaded implementation using `select()` will not require that. However, using `select()` may require significant refactoring if it is not considered in the initial code design.

If you choose this option, you are expected to read further about multi-threading and/or `select()` on your own (extending from week 3 and 10 practicals). This is training for what computer professionals are expected to do.

Please use only a single process; do *not* use `fork()` or other means to create additional processes.

Remember that this is an advanced option; you are able to get 90% without doing it. Your time is probably better spent ensuring you did well on the other parts. However, if you are realistically aiming for 15/15, then this option is for you.

If you implement non-blocking, include the line "`#define NONBLOCKING`" in your code. Otherwise, this functionality will not be tested/marked.

3 Submission

All code must be written in C (e.g., it should not be a C-wrapper over non C-code) and cannot use any external libraries or call upon external programs (e.g. exec and parse output). You may use standard libraries (e.g. to print, parse command line arguments etc.). As a rule of thumb, if you don't need "`-l`" then the library is standard. The maths library, which requires "`-lm`", is also standard. If you are unsure about a library, ask on Piazza.

Your code must compile and run on the provided VMs and produce deterministic output. Signal handlers which catch SIGSEGV will be removed.

The repository must contain a Makefile which produces an executable named `dns_svr`, along with all source files required to compile the executable. Place the Makefile at the root of your repository, and ensure that running `make` places the executable there too. Make sure that all source code is committed and pushed.

Executable files (that is, all files with the executable flag set that are in your repository) **will be removed** before marking. Hence, ensure that none of your source files has the executable flag set.

For your own protection, it is advisable to commit and push your code to git regularly. git history may be considered for matters such as special consideration, extensions and potential plagiarism. Your commit messages should be a short-hand chronicle of your implementation progress and will be used for evaluation in the Quality of Software Practices criterion.

You must submit two things:

- The full 40-digit SHA1 hash of your chosen commit to the Project 2 Assignment on LMS.
- Your commits to the repository named comp30023-2021-project-2 in the subgroup with your username of the group comp30023-2021-projects on gitlab.eng.unimelb.edu.au.

You will be allowed to update your chosen commit. However, only the last commit hash submitted to LMS before the deadline will be marked without late penalty even if you update your `git` repository after that. You should ensure that the commit which you submitted is accessible from a fresh clone of your repository. For example:

```
git clone https://gitlab.eng.unimelb.edu.au/
    comp30023-2021-projects/<username>/comp30023-2021-project-2
cd comp30023-2021-project-2 && git checkout <commit-hash-submitted-to-lms>
```

Late submissions will incur a deduction of 2 mark per day (or part thereof).

Extension policy: If you believe you have a valid reason to require an extension, please fill in the form accessible on Project 2 Assignment on LMS. Extensions will not be considered otherwise. Requests for extensions are not automatically granted and will be considered on a case by case basis.

3.1 Continous Integration

As for project 1, git has been set up to run some tests on your code every time you push. This is to help in your debugging. To access the detailed results of the tests, navigate to your project repository on Gitlab and click the green tick (or red cross) to the left of the short commit hash (above the listing of files). Then, click the tick/cross under Stages and select the first dropdown item.

The CI will 'fail' (and you may receive corresponding emails) if you did not pass all the visible test cases for tasks 1-4.

There are some tests for Phase 1. If your Makefile produces an executable called phase1 then the tests will be performed on it. There are no marks allocated to the phase1 executable; these tests are only to help you to debug. The phase1 program must read in a packet from **stdin**, parse it, and then output the log entry (to the file `./dns_svr.log`) that would have been written if you had received the packet over the network. A command line argument "query" or "response" will be given to indicate where the packet would have come from. i.e. we will run command `./phase1 [query|response] < <path-to-packet-file>` to invoke your phase1 program.

4 Assessment criteria

4.1 DNS responses

The finished project should be a functional DNS server, for AAAA queries.

4.2 Log

Keep a log at `./dns_svr.log` (i.e., in current directory) with messages of the following format:

`<timestamp> requested <domain_name>` – when you receive a request

`<timestamp> unimplemented request` – if the request type is not AAAA

`<timestamp> <domain_name> expires at <timestamp>` – for each request you receive that is in your cache

`<timestamp> replacing <domain_name> by <domain_name>` – for each cache eviction

`<timestamp> <domain_name> is at <IP address>` – see 2.1

The functions `inet_ntop()` and `inet_pton()` are useful for converting between the text version of the IP address and the 16-byte binary version in the packets. Use man to learn how to use them. Note that the maximum length of the text version is given by `INET6_ADDRSTRLEN`, defined in `<arpa/inet.h>`. See [6] for a discussion.

- The field timestamp must not contain spaces. It should be in the ISO8601 format that is generated by `strftime()` with the format "`%FT%T%z`".
- Fields must be separated by a single space.
- Line endings must be the Unix standard LF (\n) only.
- The log file must contain ASCII characters only. There is no need to perform Punycode conversions.
- Flush the log file after each write (e.g. using `fflush(3)`). (By default, files on disk are block buffered, which means that lines written to the log would not appear for a long time. This makes it difficult for a sysadmin to use the logs to track down problems.)
- Log entries due to cache operation (if implemented) must be printed *before* those relating to the cached packet itself.
- Output to stdout and stderr will not be assessed.
- The log file must be readable and writable by the current user (e.g., file permission 0644).

4.3 Error handling

When a request is received for something other than a AAAA record, respond with Rcode 4 ("Not Implemented") and log "`<timestamp> unimplemented request`". Do not forward valid non-AAAA responses to your client; send nothing except what is needed to send the Rcode.

Assumptions that can be made:

- Requests from client and responses from server will always be well-formed **DNS** packets
- Clients will always send query (QR 0), Server will always send response (QR 1)
- OPCODE is 0, TC is 0, RD is 1
- The server will always be reachable

You may respond with other Rcodes if any of these assumptions are detected to be broken, though this is not required nor examined.

Real systems code has to be very robust to errors; a bug in the kernel will crash the entire computer, which may be supporting many VMs, or running a bank's entire payment system. However, to keep this project simple, no further error processing is required here.

4.4 Build quality

Running `make clean && make -B && ./dns_svr <server-ip> <server-port>` should start the submission.

If this fails for any reason, you will be notified and be allowed to resubmit (with the usual late penalty).

- Do not commit `dns_svr` or any other executable or object file (see Practical 2). A 0.5 mark penalty will be applied if your final commit contains any such files. Executable scripts (with `.sh` extension) are excepted.
- `make clean` must be functional (i.e. it runs successfully, and removes all executable, .o and log files).
- Compiling using `-Wall` should yield no warnings.

4.5 Quality of software practices

Factors considered include quality of code, based on:

- proper use of version control, based on the regularity of commit and push events, their content and associated commit messages (e.g., repositories with few commits pushed near the deadline, non-informative commit messages will attract mark deductions)
- choice of variable names, comments, formatting (e.g. indentation, line length, line spacing/whitespace)
- abstraction and modularity (making reasonable effort to avoid e.g. duplicate code, long functions, global variables, repeated uses of magic numbers).

Dividing source code into multiple C files is encouraged; marks will be deducted for any source file more than 500 lines. When having multiple source files, compilation should occur in stages.

4.6 Caching

Cache behaviour will be evaluated by checking the correct processing of TTL fields, ID fields, and eviction behaviour, by both the log and the DNS responses.

4.7 Non-blocking operation

Marks for non-blocking operation will be allocated if `dns_svr` is able to process up to multiple requests for the user (up to, say, 100), and the matching replies from the server, regardless of the order of these (except, of course, that replies must be after the requests they are replying to).

5 Assessment process

The marking in this project is mainly based on meeting criteria in a testing framework. This is similar to test-driven development (TDD). This is sometimes called "automated marking", but thinking of it as TDD reflects the fact that this approach is widely used outside of university.

Most of the tests are available in Gitlab's Continuous Integration (CI) feature, and so it is important that you commit regularly and check the CI output. Similar to project 1, there are also hidden tests.

The feedback on CI is indicative of the marks that you will receive **for these cases**.

Of the 11 marks for program execution, 6 come from cases in CI, and 5 come from tests not in CI.

The marks are broken down as follows:

Task #	Marks	Task	In CI	Not in CI
1	3	Correctly sending DNS responses	1.85	1.15
2	4	Correct log output, excluding cache	2.30	1.70
3	1	Error handling	0.30	0.70
4	2	Build quality	2	0
5	2	Quality of software practices	0	2
6	1.5	Caching	0.75	0.75
7	1.5	Non-blocking operation	0.80	0.70

6 Collaboration

You may discuss this project abstractly with your classmates but what gets typed into your program must be individual work, not copied from anyone else. Do *not* share your code and do *not* ask others to give you their programs. Do *not* post your code on subject's discussion board Piazza. The best way to

help your friends in this regard is to say a very firm *"no"* if they ask to see your program, pointing out that your *"no"*, and their acceptance of that decision, are the best way to preserve your friendship. See https://academicintegrity.unimelb.edu.au for more information.

Note also that solicitation of solutions via posts to online forums, whether or not there is payment involved, is also Academic Misconduct. You should not post your code to any public location (e.g., `github`) while the assignment is active or prior to the release of the assignment marks.

If you use any code not written by you, you must attribute that code to the source you got it from (e.g., a book or Stack Exchange).

Plagiarism policy

You are reminded that all submitted project work in this subject is to be your own individual work. Automated similarity checking software may be used to compare submissions. It is University policy that cheating by students in any form is not permitted, and that work submitted for assessment purposes must be the independent work of the student concerned.

Using git is an important step in the verification of authorship.

7 Questions for reflection

These do no need to be submitted, but will help you understand the subject. You are welcome to discuss them.

1. Why should requests be accepted on port 8053 by default? Why not 53? Why not 8000?
2. What is a AAAA record?
3. How secure is DNS? How could it be attacked?
4. How efficient is DNS?
5. What are the xxd and od utilities? How could they be used for debugging?
6. If you have your project running on your VM, what will the following command do?
   ```
   $ dig +tcp @127.0.0.1 -p 8053 AAAA example.com
   ```

References

[1] J. Routley, "Let's hand write DNS messages", https://routley.io/posts/hand-writing-dns-messages, retrieved 9 February, 2021.

[2] S. Thomson, C. Huitema, V. Ksinant and M. Souissi, "RFC3596: DNS Extensions to Support IPv6" https://tools.ietf.org/html/rfc3596

[3] R. Hinden and S. Deering, "IPv6 Addressing Architecture", Section 2.2, https://tools.ietf.org/html/rfc4291#section-2.2

[4] P. Mockapetris, "Domain names - Implementation and Specification", Section 4.2.2, https://tools.ietf.org/html/rfc1035#section-4.2.2

[5] J. Dickinson et al., "DNS Transport over TCP - Implementation Requirements", Section 8, https://tools.ietf.org/html/rfc7766#section-8

[6] "Why is INET6_ADDRSTRLEN defined as 46 in C?", https://stackoverflow.com/questions/39443413/why-is-inet6-addrstrlen-defined-as-46-in-c/39443536#39443536

You can refer to any documentation on DNS, but not copy C code.

The authoritative references are the relevant Requests For Comments (RFCs), but many are heavy going and I don't recommend starting with them, except [2] above.