Directions

A. Due Date: 25 Jan 2026 at 11:55pm

B. The homework may be done in groups of up to two students.

What to turn in

C. Turn in all related source code (.java files) along with any headers or supplemental libraries needed to compile the code.

D. The GitHub repository includes several GitHub Actions scripts that will compile your code and perform test runs on it. You can see the output from the actions scripts in the Actions tab on GitHub. Those outputs will be the basis for your grade.

- Do not submit compiled .class or .jar files. The Actions scripts will build them for you.
- Do not change the package names in the starter code.
- Do not change the name of the class with the `main` method in it. You made add more classes or packages as necessary, but you may not remove files.
- Do not submit output files or log files generated by the tests.

E. Turn in all related source code along with any headers or build files needed to compile the code in the GitHub repository.

F. Modify the provided README.md file to include the following information:

- Names and TZ of all students in the group
- Total number of hours spent on each part of the assignment
- Date of submission
- Comments or feedback on the assignment's requirements or complexity

G. A complete README.md is worth 5 points on the assignment.

How to submit

H. Turn in your submission via the starter repositories opened for you on GitHub via GitHub Classroom.

I. All submissions **must** be made in the starter GitHub repository for the assignment that is opened via GitHub classroom. Code placed anywhere else or submitted in any other manner - including via email or in unrelated repositories - will not be graded.

J. Indicate that your work is complete by performing a commit with the message "Submitted for grading" on each repository. The grade for the assignment will be based on the first commit with that comment, so create a submission with that comment only when you are finished.

TCP, Threads, and Domain Name System (DNS)

1 Overview

In this assignment, you will implement a multithreaded DNS server in Java that accepts DNS queries over TCP, caches responses, and forwards queries to an upstream DNS server when necessary. The assignment will give you hands-on experience with:

- TCP socket programming
- Multithreaded server design
- DNS protocol fundamentals using the `dnsjava` library
- Cache management with TTL expiration
- Concurrent data structures

2 Background

The Domain Name System (DNS) translates human-readable domain names (e.g., `www.example.com`) into IP addresses using a query-response format. Two sample conversations are shown in Figure 1.

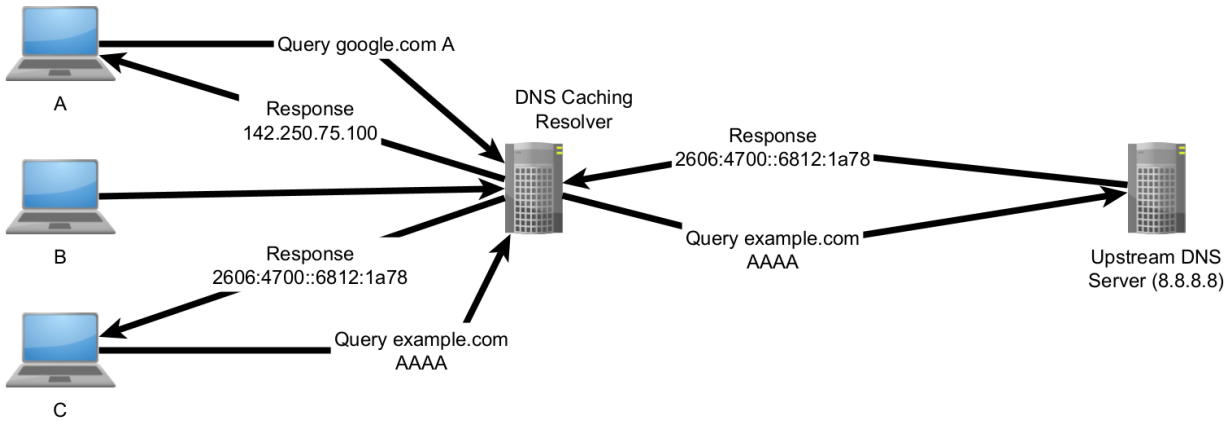

Figure 1: DNS resolving steps.

In the first conversation, C queries the DNS caching resolver for the IPv6 address for `example.com`. The resolver doesn't have a response for `example.com`, so it queries the upstream DNS server (8.8.8.8) to resolve. The DNS caching resolver then forwards the response back to C.

In the second conversation, A queries the DNS caching resolve for the IPv4 address for `google.com`. In this case, the resolver already has the answer from a previous query (not shown in the figure) and so the response is sent back without querying the upstream resolver.

2.1 Query TTL

Part of every DNS response is a time to live (TTL) field that indicates how long the response can be considered accurate. We will use the TTL provided in the responses to keep responses only for their indicated validity periods. If a response doesn't have a TTL for some reason, use a default value of 3 minutes for it.

2.2 Query and Response IDs

Due to issues with DNS cache poisoning (you can read about it here), DNS queries are sent with a random query ID inside. If the response doesn't have the same query ID as the query, the receiver ignores the response. Since we're going to be implementing a caching resolver, we need to be careful to use the query ID correctly.

If a query is a **cache miss** and the upstream DNS server must be queried, you can just use the same query ID from the client and pass the response back when it returns from the upstream server.

If a query is a **cache hit**, our cached response might have the query ID from a previous query inside it still, so we must make sure to prepare and return a response with the query ID from the current query. If we don't, the querying client will reject the response.

3 DNS Caching Resolver Using TCP

While DNS typically uses UDP on port 53 for queries, it also supports TCP for larger responses and zone transfers. We will use this functionality to create our multithreaded DNS resolver. The server you will write shall:

1. Listen for incoming TCP connections from DNS clients
2. Parse DNS query messages using the `dnsjava` library (Code at `https://github.com/dnsjava/dnsjava`, JavaDoc at `https://javadoc.io/doc/dnsjava/dnsjava/latest/index.html`)
3. Check if the requested domain is in the cache
4. If cached and not expired, return the cached response
5. Otherwise, forward the query to an upstream DNS server via TCP
6. Cache the response with appropriate TTL
7. Return the response to the client

Since TCP is stream oriented and not packet oriented like UDP, DNS over TCP must delineate where a request or reply begins and ends. DNS over TCP uses a length field to prefix its requests and responses so the client and server know where data boundaries are. The prefix is a 2-byte length field (in network byte order) indicating the length of the DNS message that follows. The framing allows TCP to properly delimit individual DNS messages in the stream. The format is shown in Figure 2.

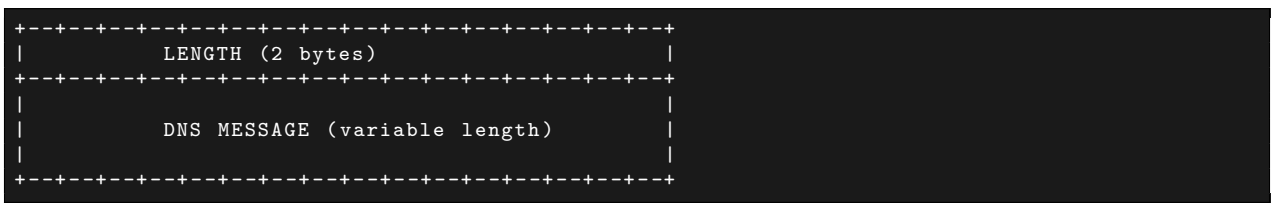

Figure 2: DNS over TCP message format

4 Server Requirements

TCP Server Create a TCP server that listens on a configurable port. The port must be provided at the command line.

Multithreading Handle each client connection in a separate thread to support concurrent queries. You can use Thread or Runnable for the threading implementation.

DNS Query Parsing Since DNS queries are binary messages and non-trivial to parse, use the `dnsjava` library to parse incoming DNS query messages and extract the queried domain name and query type.

Cache Implementation Implement a thread-safe cache using appropriate Java concurrent data structures. You can use `ConcurrentHashMap` for example to store responses using a cache key that consists of the domain name and query type.

Store both the DNS response and its expiration time in the cache to implement TTL based expiration.

Upstream DNS Resolution If a query arrives that is not in the cache or has expired, connect to an upstream DNS server to resolve. For example, you might use Google's `8.8.8.8:53` server using TCP connections. In your query, forward the original query message from the client.

You must handle and parse the response yourself using `dnsjava`. **Do not** use Java's built in resolution methods.

Response Handling Return cached responses immediately if valid. If a query misses, add the response to the cache to make future queries faster.

Remember to properly frame all responses with the 2-byte length prefix. Methods to support reading and writing the response length in network byte order are found in the starter code.

Configuration The server must support command-line arguments for:

- -p: Server port (*e.g.* 5300) an positive integer [1025, 65535]
- -upstream: Upstream DNS server address (*e.g.* 8.8.8.8)
- -upstream-port: Upstream DNS server port (*e.g.* 53), a positive integer [1, 65535].

The parameters must be passed in by name and may be in any order. If any parameter is missing, quit with a usage message. If any parameter is invalid, quit with an error and usage message.

4.1 Technical Requirements

1. Use the latest `dnsjava` library for DNS message parsing and creation
2. Implement proper exception handling and logging
3. Use thread-safe collections for the cache

5 Using dnsjava for DNS Message Handling

When reading and writing DNS messages from TCP you'll need to use the following methods and classes:

1. `Utils.networkOrderArrayToLength` to convert the message length received in network byte order array (*e.g.* `[0x00, 0x32]`) to an integer (*e.g.* 50).
2. `Utils.lengthToNetworkOrderArray` to create the network byte order array for a message contents (*e.g.* length 50-byte array) to a network byte order array (*e.g.* `[0x00, 0x32]`).
3. `org.xbill.DNS.Message` Represents a DNS message (query or response)
   - `Message(byte[])` creates a Message object from the bytes received on the network
   - `toWire()` creates a network-ready version of the message for sending
   - `getQuestion()` gets the query part of the message as `Record` object

4. `org.xbill.DNS.Record` represents a resource record that can be a query or a response.
   - `getName()` gets the resource name (*e.g.* google.com)
   - `getType()` gets the type of resource requested (*e.g.* A, AAAA, TXT)
5. `org.xbill.DNS.Header` represents the query or response header. That's where the query ID can be get or set (`getID()`, `setID()`).

6 Documentation (5 points)

Add Javadoc documentation to every method and class. The documentation for methods must include:

- A 1-2 sentence summary of the method's purpose
- @param entries for all parameters, including what they are used for
- @return entry with a 1-2 sentence description of the return value
- @throws entries for any exceptions thrown.

The documentation for classes must includes:

- A 2-3 sentence description of the class' purpose
- @author the code's author
- @version a version for the class. Update on every commit to the repository.

7 Testing Your Implementation

You can test your server with command line tools from Linux and Windows. The following are instructions for how to test with dig and nslookup assuming your server is listening on 127.0.0.1 (localhost) and port 5300. Your server should successfully process the requests from the tools and return results they can understand.

7.1 Testing with dig

```
# Query your server using TCP
dig @localhost -p 5300 +tcp www.example.com

# Query for different record types
dig @localhost -p 5300 +tcp www.google.com A
dig @localhost -p 5300 +tcp google.com MX
```

7.2 Testing with nslookup

```
nslookup -vc -port=5300 www.example.com localhost
```

Note: The `-vc` flag forces TCP mode in nslookup.

8 Tests (90 points)

The tests will use the dig tool as a client for your DNS server. The points for the tests will be based on the behavior of the server as seen through dig. The tests shown below assume that the DNS server is listening on 127.0.0.1:5300:

```
java -jar DNSCacheServer-5786.jar -port=5300 -upstream=8.8.8.8 -upstream-port=53
```

8.1 Basic Resolution

Test resolution of a regular A DNS address:

```
dig @localhost -p 5300 +tcp www.example.com A +short
```

The result should be an IPv4 address.

8.2 Cache Hit

Test that the cache works. Two queries for the same domain should lead to the second query returning faster.

```
dig @localhost -p 5300 +tcp www.google.com A +short
sleep 1
dig @localhost -p 5300 +tcp www.google.com A +short
```

The result should be that the second query finishes significantly faster than the first one.

8.3 Multiple Concurrent Queries

Test that the server can handle multiple concurrent sessions. The test will run 10 queries all at once and ensure that the results are returned for all of them.

```
dig @localhost -p 5300 +tcp test1.example.com A &
dig @localhost -p 5300 +tcp test2.example.com A &
dig @localhost -p 5300 +tcp test3.example.com A &
dig @localhost -p 5300 +tcp test4.example.com A &
dig @localhost -p 5300 +tcp test5.example.com A &
dig @localhost -p 5300 +tcp test6.example.com A &
dig @localhost -p 5300 +tcp test7.example.com A &
dig @localhost -p 5300 +tcp test8.example.com A &
dig @localhost -p 5300 +tcp test9.example.com A &
dig @localhost -p 5300 +tcp test10.example.com A &
```

8.4 Different Record Types

Test that the resolver can handle multiple record types, not just A.

```
dig @localhost -p 5300 +tcp www.google.com A +short
dig @localhost -p 5300 +tcp www.google.com AAAA +short
dig @localhost -p 5300 +tcp google.com MX +short
```

The result should be responses for all three record types.

8.5 Cache TTL Expiration

Test that the resolver removes entries from the cache after they expire. This test will take a few minutes since it must wait for the resolver cache to expire.

```
dig @localhost -p 5300 +tcp www.example.com A
sleep 1
dig @localhost -p 5300 +tcp www.example.com A
# Sleep for longer than the TTL should be
dig @localhost -p 5300 +tcp www.example.com A
```

The second query should be faster than the first one due to the caching. The third query should be slow again due to the TTL timeout.

8.6 TCP Framing

Tests that the resolver uses the proper length prefix on its messages. To check that, the test will manually create a DNS query for example.com and check its response header and length.

8.7 Invalid Domain

Tests that the resolver properly returns responses for an invalid domain name (NXDOMAIN):

```
dig @localhost -p 5300 +tcp this-domain-definitely-does-not-exist-12345.com A
```

8.8 Large Response

Tests that the resolver can handle large responses, larger than 512 bytes.

```
dig @localhost -p 5300 +tcp google.com TXT +stats
```

The response includes the size of the message from dig and we check that it is valid. When I tested, the response size was 886 bytes.

8.9 Connection Handling

Tests that it can properly handle 10 sequential connections without a problem.

```
dig @localhost -p 5300 +tcp test1.example.com A
dig @localhost -p 5300 +tcp test2.example.com A
dig @localhost -p 5300 +tcp test3.example.com A
dig @localhost -p 5300 +tcp test4.example.com A
dig @localhost -p 5300 +tcp test5.example.com A
dig @localhost -p 5300 +tcp test6.example.com A
dig @localhost -p 5300 +tcp test7.example.com A
dig @localhost -p 5300 +tcp test8.example.com A
dig @localhost -p 5300 +tcp test9.example.com A
dig @localhost -p 5300 +tcp test10.example.com A
```

All connections should work.

8.10 Parameters and Validation

Checks that the server can handle parameters in any order and outputs proper responses when invalid or missing parameters are given:

8.10.1 Parameter Order

```
java -jar DNSCacheServer-5786.jar -upstream=8.8.8.8 -port=5300  -upstream-port=53
```

The server should start as usual.

8.10.2 Missing Parameter

```
java -jar DNSCacheServer-5786.jar -port=5300  -upstream-port=53
```

The server should exit with a usage message:

```
Usage: DNSCacheServer-5786
  -port=<port>             Server port
  -upstream=<address>      Upstream DNS server
  -upstream-port=<port>    Upstream DNS port
```

8.10.3 Invalid port (negative)

```
java -jar DNSCacheServer-5786.jar -upstream=8.8.8.8 -port=-1  -upstream-port=53
```

The server should exit with an error and a usage message:

```
Error: Server port must be between 1025 and 65535
Usage: DNSCacheServer-5786
  -port=<port>              Server port
  -upstream=<address>       Upstream DNS server
  -upstream-port=<port>     Upstream DNS port
```

8.10.4 Invalid Port (not an number)

```
java -jar DNSCacheServer-5786.jar -port=abc -upstream=8.8.8.8 -upstream-port=53
```

The server should exit with an error and a usage message:

```
Error parsing server port: For input string: "abc"
Usage: DNSCacheServer-5786
  -port=<port>              Server port
  -upstream=<address>       Upstream DNS server
  -upstream-port=<port>     Upstream DNS port
```