Networking Assignment 2025 (Draft)

Assignment description and communication scenario

It is often necessary to customize a protocol to fit an existing platform or develop a new one when starting a new project.

In this instance, you will work on constructing a custom DNS client and server application. The goal is for the client to send DNS lookup requests to the server. The server, in turn, will resolve the DNS lookups and reply to the client.

In this assignment, the custom DNS protocol will utilize a UDP connection. By following our guidelines and protocol design, you will observe how it differs from the traditional version, lacking several additional features, yet demonstrating how some theoretical concepts work in practice.

Both the client and the server must always handle errors and exceptions properly, ensuring they exit in a controlled manner. No exceptions should remain unhandled.

For this assignment, a low-level socket implementation is required. To send and receive UDP messages, the methods `Socket.SendTo()` and `Socket.ReceiveFrom()` must be used. Both methods operate on byte data. Proper object handling is necessary to send and receive messages effectively.

The protocol

The protocol works as follows

1. The client sends a *Hello*-message to the server.
2. The server replies with a *Welcome*-message.
3. The client sends a *DNSLookup*-message to the server. The DNSLookup content will contain partial information(see next item).
4. After receiving the *DNSLookup* the server will query the given JSON-file based on the *type* and the *name of the DNSRecord* included in the request (each *DNSLookup*-message must contain a ***type*** *and* ***name***).
5. The server now has two options:
    - [Option 1] If the query is successful and a record is found a complete *DNSRecord* will be created and sent as a *content* in the *DNSLookupReply*-message to the client. No new *MsgId* will be assigned to the reply message, but it will have the same *MsgId* as the *DNSLookup*-message.
    - [Option 2] If the query is unsuccessful an *Error* message will be created and sent to the client.
6. The client receives the *DNSLookupReply*-message and confirms it with an *Ack*-message. The same *MsgId* of the DNSLookup-message and *DNSLookupReply-message* must be included as a content in the *Ack*-message.
7. The client can then start sending the next *DNSLookup-message.*

Note: It is expected that your client sends at least four DNSLookup-messages.

- *At least* **two correct** *DNSLookup*-messages but have different *DNSRecord types* and *names.*
- At Least **two incorrect** *DNSLookup*-messages to test your implementation and how it handles different issues like errors and exceptions.

8. Once the server received the last *Ack*-message it will send an *End*-message to the client.
9. The client will terminate once the *End*-message is received.

10. The server will not terminate, once the End is sent. It will stay running to receive messages of the next client.

Sample Message Flow

1. **Handshake:**
   - Client → Server:
     `{ “MsgId”: “1” , ”MsgType": "Hello", "Content": “Hello from client” }`
   - Server → Client:
     `{ MsgId”: “4” , “MsgType”: "Welcome", “Content": “Welcome from server”}`

2. **DNS Lookup (Valid Request):**
   - Client → Server:
     `{ MsgId”: “33”, "MsgType": "DNSLookup", "Content": "www.example.com" }`
   - Server → Client:
     `{ MsgId”: “33”, "MsgType": "DNSLookupReply", "Content":{ 'Type': 'A', 'Name': 'www.example.com', 'Value': '192.168.1.1', 'TTL': 3600 }" }`
   - Client → Server:
     `{ MsgId”: “4112”, "MsgType": "Ack", "Content": “33”}`

3. **DNS Lookup (Invalid Request):**
   Scenario 1:
   - Client → Server:
     `{ MsgId”: ”1245” , "MsgType": "DNSLookup", "Content": "unknown.domain" }`
   - Server → Client:
     `{ MsgId”: ”7534445”, "MsgType": "Error", "Content": "Domain not found" }`

   Scenario 2:
   - Client → Server:
     `{ MsgId”: ”1245” , "MsgType": "DNSLookup", "Content": { 'Type': 'A', 'Value': 'www.example.com' } }`
   - Server → Client: `{ “MsgId”: ”7534445”, "MsgType": "Error", "Content": "Domain not found" }`

4. **End Communication:**
   - Client → Server:
     { MsgId”: “91377”, "MsgType": "End", "Content": “No Lookups anymore”}
   - Server: Keeps running for next client.

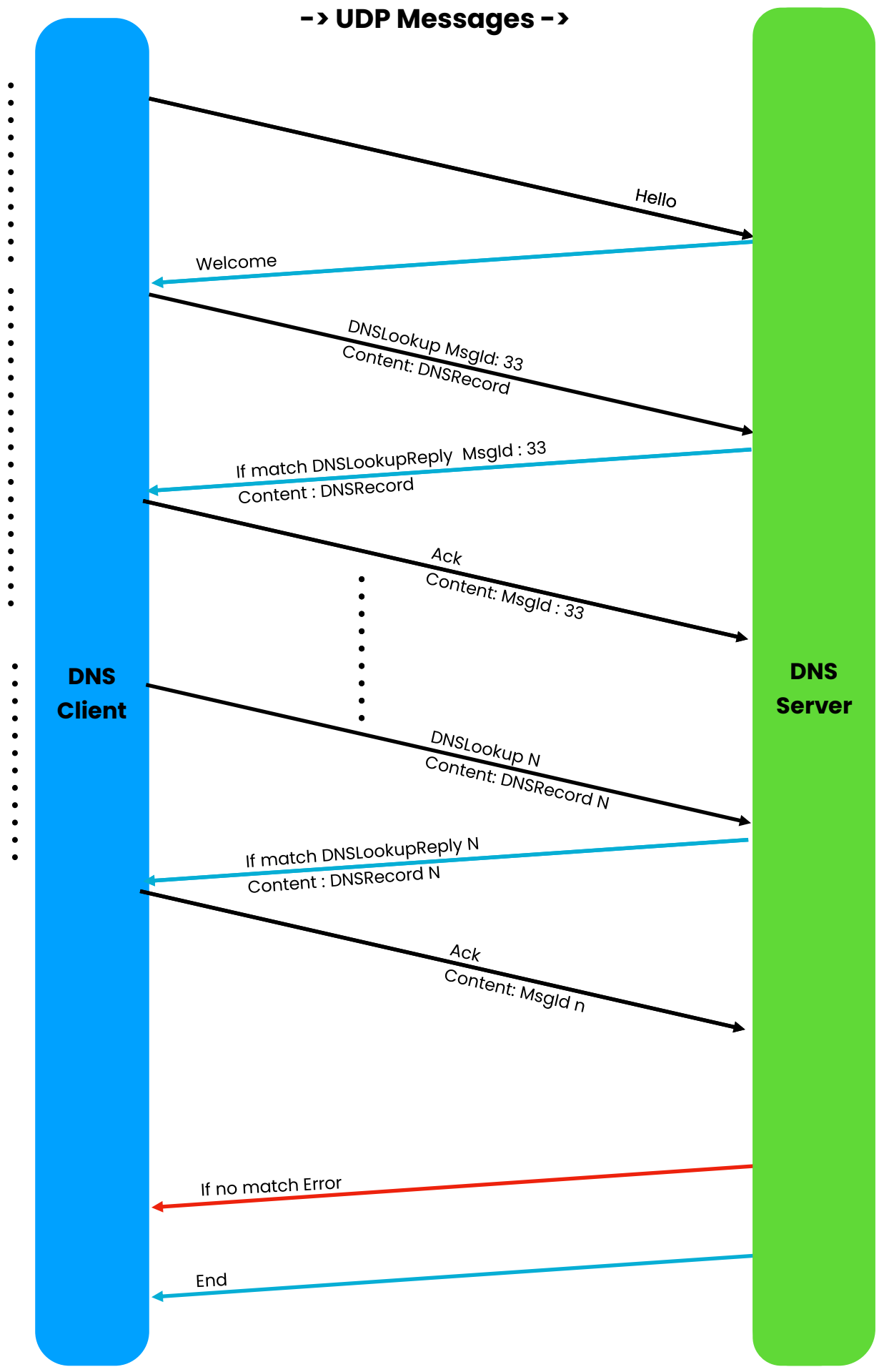
The Supported Messages

```
// NOTE: THIS FILE MUST NOT CHANGE

public class Message
{
    public int MsgId { get; set; }
    public MessageType MsgType { get; set; }
    public Object? Content { get; set; }
}

public enum MessageType
{
    Hello,
    Welcome,
    DNSLookup,
    DNSLookupReply,
    DNSRecord,
    Ack,
    End,
    Error
}

public class DNSRecord
{
    public string Type { get; set; }
    public string Name { get; set; }
    public string? Value { get; set; }
    public int? TTL { get; set; }
    public int? Priority { get; set; } // Nullable for non-MX records
}
```


-> UDP Messages ->
DNS Client
DNS Server
Hello
Welcome
DNSLookup MsgId: 33
Content: DNSRecord
If match DNSLookupReply MsgId : 33
Content : DNSRecord
Ack
Content: MsgId : 33
DNSLookup N
Content: DNSRecord N
If match DNSLookupReply N
Content : DNSRecord N
Ack
Content: MsgId n
If no match Error
End

Grading requirements

Additional requirement details can be found in the assignment description.

To achieve a sufficient grade:

- Your implementation must be compatible with the message data structure and comply with the custom protocol. Additionally, the client must send the expected DNS lookup messages and handle all types of errors, including:
    - Issues with socket creation.
    - Incorrect or incomplete messages (e.g., a received message missing an expected value).
    - The client or server sending messages in the wrong order (e.g., expecting "Welcome" but receiving "DNSLookup" instead).
    - Other protocol-related errors.
- The implementation must use a low-level UDP socket.
- **Both the client and server must print out detailed, well-formatted messages about the data they send and receive, as well as their actions.**
- The program must be written in .NET 8.x.
- No external libraries such as **UdpClient** or **UdpServer** is used.
- No installation of new libraries or extra files should be required to run the application.
- Your implementation must use the provided project template. Do not rename any existing files in the template.
- Do not write all the logic in a single method (e.g., **start()**), but follow OOP design.
- File paths and locations must be handled independently of the operating system (Mac/Windows/Linux).
- The solution must be the original work of the submitting group. Autogenerated code is explicitly prohibited (e.g., no use of ChatGPT or other AI tools).

- The submitting group may consist of **up to two students** with the same teacher. Retakers may choose any classmate as a partner.
- The submission MUST include the names and student numbers of the participants in the filename of the ZIP-file (sample naming is provided in the assignment).

Demonstration and Oral Assessment

Each individual or group will present and demonstrate their work to the assigned teacher. You may be asked questions about your implementation and must be able to explain your code. The teacher may also request you to test your logic using different error sequences in the protocol (i.e., valid formats with varying values).

The teacher will provide scheduling details during the lesson.

The delivery of the assignment

The delivery is expected before 04/04/2025 (17:00 pm).

Submission:

Submit the same template folder with your solution in it on **Brightspace**. The zip file should have the following name format:
student1name_number_student2name_number.zip.

Remarks: If you find errors or unclear parts let us know. This includes description, code and/or delivery.