Slovak Technical University

STU

Faculty of informatics and information technologies in the STU

Lucas Daniel Espitia Corredor

**Assignment of the protocol p2p using UDP**

Computer and Communication Networks

PKS\_B

Bc. Marian Rohun

Thursday: 8 am

16/10/2024

**Index**

**1. Introduction**

**1.1 Structure of the Protocol**

**2. Structure of the Protocol**

**2.1 Protocol Header**

**2.2 Data Fragmentation**

**2.3 Integrity Verification**

**2.4 Message Retransmission (ARQ)**

**2.5 Connection Control and Keep-Alive**

**2.6 Error Simulation**

**3. Implementation Plan**

**3.1 Establishing the connection**

**3.2 Data Submission and Data Fragmentation**

**3.3 Verification and Assurance of Message Integrity**

**3.4 Maintaining Connection and Mutual Control (Keep-Alive)**

**3.5 Simulated Error Handling**

**3.6 Testing and validation**

1. **Introduction**

This paper proposes a customized P2P (peer-to-peer) communication protocol implemented over UDP (User Datagram Protocol). This protocol has been designed to establish a connection between two nodes, agree on the parameters of such connection, and allow reliable data transmission, ensuring integrity and retransmission in case of errors. A detailed description of the proposed mechanisms for message integrity verification, data retransmission in case of errors, maintenance of the active connection, and fault simulation is included.

1. **Structure of the protocol**

***2.1 Protocol Header***

The proposed P2P protocol is based on exchanging packets with a specific format. The header of each packet will include the following fields:

|  |  |  |
| --- | --- | --- |
| Field | Description | Size (bytes) |
| Version | Protocol version | 1 |
| Length | Package size, including header | 2 |
| Message ID | Unique identifier to distinguish messages | 2 |
| Current Fragment | Current fragment number if fragmented | 1 |
| Total Fragments | Total number of fragments | 1 |
| Message Type | Type of message sent: data, control, error, retransmission, etc. | 1 |
| Checksum | Checksum to ensure completeness | 2 |
| Data | Message data (variable) | Variable |

***2.2 Data Fragmentation***

Because messages may be too large to be sent in a single packet, the protocol allows the data to be fragmented into smaller parts. This is particularly useful when the packet size exceeds the limit of the underlying protocol (in this case, UDP).

The Current Fragment field indicates the index of the current fragment. The Total Fragments field indicates how many fragments make up the original message.

The receiver reconstructs the message from the received fragments, verifying that all fragments have been received before processing the message.

***2.3 Integrity Verification***

To verify the integrity of transmitted messages, the CRC16 algorithm is used. The checksum is calculated and inserted in the packet header. When a message is received, the receiver recalculates the CRC16 on the received content and compares it with the checksum sent. If the values do not match, the receiver requests a retransmission.

***2.4 Message Retransmission (W&S)***

The protocol uses the Wait and Stop (W&S) method for retransmission of lost or damaged messages. Instead of using explicit acknowledgment (ACK) messages, the W&S mechanism ensures that the sender waits for an implicit acknowledgment before sending the next message. The operation of this mechanism is described below:

* **Message sending**: the sender node sends a message and then enters a waiting state. It cannot send a new message until the transmission of the previous message is completed.
* **Implicit confirmation**: The sender waits for the receiver to respond with the next data message. The reception of this message is interpreted by the sender as an implicit confirmation that the previous message was received correctly.
* **Timeout**: If the sender does not receive any response from the receiver within a predefined time, it assumes that the message has been lost or that the receiver has failed. In this case, the sender resends the same message and restarts the wait timer.

Message retransmission: In case the timer expires without receiving a response, the message will be resent until the receiving node sends its data message, acting as an implicit acknowledgment.

***2.5 Connection Control and Keep-Alive***

The protocol implements a Keep-Alive mechanism to maintain the connection between the nodes when no data is actively being sent. If one of the nodes does not receive a Keep-Alive message within a time interval, it assumes that the remote node has disconnected.

Keep-Alive messages are empty packets with a specific message type (e.g., Message Type = 2). If a Keep-Alive is not received within the expected time, the node may decide to terminate the connection.

***2.6 Error Simulation***

To simulate transmission errors and test the protocol's ability to handle corrupted data, a function is introduced that intentionally generates errors in one of the messages sent. This can be done by altering the content of the data before calculating the checksum or by modifying the checksum itself so that the receiver detects an error.

The sender generates the error and sends it. When the receiver receives it, it will not be able to verify its integrity and will request a retransmission.

1. **P2P Protocol Implementation Plan**

The protocol will be implemented in several phases, according to the system requirements and the specified design. The implementation plan is described below:

* 1. **Establishing the Connection**

1. ***Node initialization:***

* Each node must specify a local port (for listening) and the IP address and port of the destination (peer) node.
* The protocol will allow both nodes to negotiate parameters such as the maximum data fragment size, the timer duration for the Wait and Stop (W&S) method, and other relevant details.

1. ***Exchange of initiation messages:***

* Initial “initiation” messages (similar to a handshake) are sent to confirm that both nodes are active and ready to receive messages.
* These messages will include basic information such as ports and agreed protocol parameters.
  1. **Data Submission and Data Fragmentation**

1. ***Data fragmentation***

* The message is divided into fragments, if necessary, respecting the maximum size allowed by the protocol.
* Each fragment has an associated sequence number and is sent one by one, in order.

1. ***Wait and Stop (W&S) method:***

* After sending a fragment, the sender node enters a waiting state until it receives the next message or an implicit confirmation that the fragment was successfully received.

If no response is received within a predefined time, the sender resends the fragment.

1. ***Error handling***

* If a data failure or corruption is detected, the Checksum field in the header will be used to verify integrity.
* If an error is detected, the receiver discards the corrupted fragment and waits for retransmission.
  1. **Verification and Assurance of Message Integrity**

1. ***Integrity verification with CRC16***

* Each transmitted message will have a Checksum (CRC16) calculated on the message content.
* The receiver will calculate the CRC16 to verify that the message has not been corrupted during transmission.

1. ***Detection and Retransmission of Lost Messages***

* The protocol assumes lost messages if the implicit acknowledgment (another message or reply) is not received within the defined time.
* In this case, the sender will retransmit the lost message until an implicit acknowledgment is received or until a maximum number of retries is exhausted.
  1. **Maintaining Connection and Mutual Control (Keep-Alive)**

1. ***Status control and Keep-Alive messages***

* A Keep-Alive mechanism will be implemented to check if the remote node is still active, especially during periods of inactivity.
* Nodes will exchange periodic messages including their current status.

1. ***Disconnection management***

If one of the nodes does not respond to Keep-Alive messages within a specified time, the connection will be considered to have failed and will be shut down.

* 1. **Simulated Error Handling**

1. ***Error injection***

The protocol will allow intentional errors to be injected into transmitted messages (e.g., by manually manipulating the Checksum field) to simulate corrupted data and to verify the robustness of the protocol in failure situations.

1. ***Error handling***

Upon detecting a message with an incorrect Checksum, the receiver will reject the corrupted fragment and notify the sender, who must retransmit it.

* 1. **Testing and Validation**

1. **Capture and analysis with Wireshark**

* Tools such as Wireshark will be used to capture and analyse messages transmitted between nodes.
* Headers, protocol fields, and correct message retransmission will be verified.

1. **Text and file transmission tests**

* A test system will be implemented to verify the correct transmission of data in both directions.
* Fragmentation of large messages, retransmission of lost or damaged fragments, and correct reception of files will be tested.

1. **Development**

The port connection has been successful, the following simulates the receiving and sending of messages from two different cmd, for the moment the code only provides the connection, likewise wireshark shows the protocol used in the network so the connection has been successful.

It is necessary to emphasize, that for now only the connection of the ports has been made using a “client” of means, the protocol is very precarious still, since it has not been implemented anything more than what has been described previously, but it fulfills the expectations of this first control point.

**P2P connection**

Screens screenshot of a computer screen

Description automatically generated

**Wireshark**

As you can see below the nodes have been successfully connected and at the same time, we can capture their movement through the network, thanks to wireshark, to verify that it is our protocol.

First, the ip ports correspond to the screenshots and second, the message or shared data is also shown in the screenshots.



**DATA**

A screenshot of a computer

Description automatically generated

**Final Solution**

In the end it has been decided to change the header of the protocol, due to inconsistencies that it had and also at the expense of the teacher.

In the following part we will explain the implementation of the code and what has been used.

**Header**

|  |  |  |
| --- | --- | --- |
| **Field** | **Size (bytes)** | **Description** |
| **Type** | 2 | Specifies the type of the packet or message, used to identify its purpose. |
| **Port src** | 4 | Source port number, indicating the sender's port. |
| **Dst src** | 4 | Destination port number, indicating the receiver's port. |
| **Total length** | 4 | Total length of the message, including header and data. |
| **Offset** | 4 | Offset for fragmented packets, specifying where this fragment starts. |
| **Length Fragment** | 4 | Length of the current fragment in bytes. |
| **Crc32** | 4 | Checksum value for error-checking the message integrity. |
| **Data** | X(variable) | Payload of the message, containing the actual data being transmitted. |

As you can see in the table above a new protocol has been created, things that were unnecessary such as the message id and version have been removed, it is not necessary, it has been changed literally by a flag, message type which can get certain values and depending on this the main receiver will route these packets.

**Flags Explanation**

The **TYPE** field (2 bytes) determines the purpose of the message.

Each value has a specific meaning:

|  |  |  |
| --- | --- | --- |
| **Flag Value** | **Meaning** | **Description** |
| 0 | Message Data | The payload contains regular message data. |
| 1 | File Data | The payload contains file data, often part of a larger transfer. |
| 2 | |  | | --- | |  |   Keep-Alive Message | A signal to maintain the connection alive without transmitting additional data. |
| 3 | Keep-Alive Response (ACK) | A response to a keep-alive message, acknowledging the connection's status. |
| 4 | Final ACK for Fragment Chain | Acknowledges successful transmission of the entire chain of fragments. |
| 5 | Message and Fragments Received Confirmation | Confirms that the entire message and its fragments were received without issues. |
| 6 | |  | | --- | |  |  |  | | --- | | Request for Lost/Corrupted Fragments (NACK) | | Indicates fragments were lost or corrupted. Includes indices of the problematic fragments for retransmission. |

**Key Notes**

1. **Keep-Alive Messages (Flags 2 & 3):** Used in long-lived connections to prevent timeouts when no actual data is being sent.
2. **Final ACK (Flag 4):** Ensures that both sender and receiver agree on the successful completion of data transmission.
3. **NACK (Flag 6):** Provides robust error handling by pinpointing specific fragments that need retransmission.

**Fragmentation**

The protocol uses a configurable variable called fragment\_size to handle data transmission efficiently. This variable defines each data fragment's maximum size (in bytes) during transmission. Fragment\_size is set to 1024 bytes by default, but it can be dynamically adjusted during program execution based on requirements or network conditions.

It is important to note that **fragmentation only applies to the data payload**. The 26 bytes reserved for the protocol header are **excluded** from the fragment size calculation. This design ensures that the header remains consistent and is transmitted intact with each fragment

**Maximum Number of Fragments**

A 4-byte field determines the maximum possible number of fragments in the protocol. Since this field uses 4 bytes, it allows for fragments **4,294,967,295 fragments** in total.

This exceptionally high limit ensures that even the largest data transmissions can be accommodated within the protocol's design. Each fragment can carry up to the defined fragment\_size (default: 1024 bytes), meaning the theoretical maximum amount of data is 4.4 terabytes. Although that amount of packets is quite large and in practice it could take a long time + network packet loss.

**Fragment Transmission and Reception**

The protocol implements a **Selective Repeat ARQ (Automatic Repeat Request)** mechanism for reliable fragment transmission. This strategy is designed to optimize throughput while ensuring data integrity. Below is an explanation of how this process works:

1. Initial Transmission:

All available fragments are transmitted at once, to be clear the amount of fragments to be sent is obtained from a simple math len(data) // fragment\_size, so we will get an integer with the nearest index and we will know how many fragments will be used.

1. Finalization Indicator:

To signal the end of the transmission, the sender includes a **special flag (4)** in the header of the final packet, it is another packet, it is not the same as used in the payload. Basically it is a ping without data.

This packet serves as a delimiter, letting the receiver know that no more fragments are expected beyond this point.

1. Reception and Validation:

Upon receiving the final packet with the special flag (4), the receiver performs a check to determine if all expected fragments have been received.

If all fragments are present, the receiver acknowledges successful reception (using flag 5) and sends this packet to the initial sender, in other words, it is another package without data, just confirm that you have all the packages.

1. Retransmission:

The sender retransmits only the requested fragments, minimizing redundant transmissions and improving overall efficiency.

In this case, the indexes that the receiver has not been able to handle correctly are transferred directly, either because of corruption where its crc32 has been affected or because of packet loss over the network.

In addition, a list is created where these requests are stored, however, a request is not sent one by one, it is sent one request per block, this is to improve the efficiency of the code.

**how is the size of the block calculated?**

The maximum block size for retransmission requests is constrained to 256 indices due to the protocol's maximum fragment size of 1 MB (1024 bytes). Since each index occupies 4 bytes (as an integer), the total size of the indices list is calculated by multiplying the number of indices (256) by the size of each index (4 bytes). This results in a total of 1024 bytes for the indices in a single retransmission request. Therefore, the maximum block size for retransmissions is 256 indices, which fits within the 1024-byte limit for each fragment. This ensures that the retransmission request remains within the size constraints and can efficiently manage the retransmission of up to 256 fragments in a single request.

**What happens if there are more than 256 missing indexes?**

Nothing, when sending the request for packets, the receiver (the original sender) unpacks the data and filters it to send the corresponding fragments. If more than **256 indices** are needed on the other side, the process is straightforward: another **ACK of type 4** is sent to signal that the receiver should begin verifying if all the expected data has been received. If any fragments are still missing, the receiver will initiate another request with the missing indices. This process is repeated until all the fragments have been successfully received, ensuring complete and reliable data transmission.

It should be noted that it is not necessary for these indices to be next to each other following a GBN approach, the word block is used but these indixes may or may not be next to each other, their position does not really matter, they have been selected one by one for repetition.

**But how is it done, where is the magic?**

To achieve our objective and determine which type of fragment needs to be sent, we must understand the UDP protocol used. In this case, the offset field will contain the sequence number, which helps track the order of fragments. Notably, when there is no fragmentation, the offset is 0, but once fragmentation occurs, it starts with 1 and continues up to x (the total number of fragments). This makes it easy to track and manage each fragment in the sequence.

Furthermore, on the sending side, the program will be designed to block until the entire transmission is complete, ensuring that the fragment list is not prematurely deleted or modified. This prevents potential errors, ensuring that all fragments are properly transmitted and accounted for before the program proceeds.

**Keep alive**

For the keep-alive mechanism, a series of timers and monitoring techniques have been implemented to ensure the connection remains active. The maximum allowable time without receiving a response is 15 seconds, but there are specific exceptions to this rule.

The keep-alive process starts and begins to adjust its timing only after the initial handshake has been completed. Before the handshake, the connection is not considered to be in an active state, and therefore, the protocol will simply monitor for the first connection attempt. Once the handshake occurs and the connection is established, the keep-alive mechanism will start sending ping messages at regular intervals when no other messages are being transmitted. These pings are sent to confirm that the connection is still active and that the other side is responsive.

If no messages are being sent, the keep-alive will regularly send ping packets, and it will expect an ACK response confirming that the other side is still there. If no response is received, the protocol will attempt to send up to three ping messages, each at 5-second intervals.

If the receiver still does not respond after these three attempts, the connection is considered to have been lost, and the protocol will terminate the connection. This ensures that idle connections are properly maintained and that the system can detect when one side of the connection has become unresponsive, allowing for a timely cleanup of inactive or lost connections.

It is important to highlight that the **timer** will be reset every time a **packet** is received. This is a crucial design decision to prevent the **keep-alive** messages from interrupting the normal flow of the system. By resetting the timer upon receiving a packet, the system ensures that the **keep-alive** mechanism is only triggered when there is truly no activity, preventing unnecessary pings from being sent while data transmission is ongoing. This approach helps maintain smooth communication without interference, optimizing system performance while still ensuring that the connection remains active when needed.

**Conclusion**

The implemented protocol has been quite complex to implement, despite the setbacks, it has been possible to create a program base that is at least executable and quite intuitive where two nodes communicate with UDP protocols, in addition to that the LUA script has been created, please read README.txt before starting the program.

It should be noted that although it has not been done exactly perfectly, for example selective repeat has been used, and not GBN with SR, the program is quite functional, there are some cases that could not be resolved and it is understood that the note can be reduced for that, in addition to small bugs within it.

-> Lua script will only work if accessed with ports 10 and 20 respectively.

-> To save a file it is necessary to write the input twice, since the first one will send a message instead of saving the file.

-> After a sudden drop while receiving packets, the connection can be reestablished but the part so that the fragments continue to be sent has not been worked on. (Although I had an idea of ​​how to do it, I didn't have enough time).

Thank you.