TEST 3: ARQ Process in RLC Acknowledged Mode (AM)

REFERENCE STANDARD: 3GPP TS 38.322

LAYER: RADIO LINK CONTROL (RLC)

DIFFICULTY LEVEL: ADVANCED

SUBMITTED BY:

GROUP 3 - TEAM MEMBERS:

- ***55984_Vikas Srivastava***
- ***58622_Harshinie M***
- ***58623_Shreyash Bhatt***
- ***58624_Jayavarshini G***
- ***58635_Yuvaraj P***

TABLE OF CONTENTS

S.NO	DESCRIPTION	PAGE NO
1	Introduction to RLC ARQ in 5G NR	3
2	RLC Architecture Overview	4
3	ARQ Process Flow	6
4	C++ Implementation	8
5	SAMPLE OUTPUT	19
6	KEY CONCEPTS EXPLAINED	20
7	COMPILATION & EXECUTION GUIDE	21
8	CONCLUSION	22

1. INTRODUCTION TO RLC ARQ IN 5G NR

In the 5G New Radio (NR) communication system, the Radio Link Control (RLC) layer plays an essential role in ensuring efficient and reliable data transfer across the wireless interface. It is situated between the Packet Data Convergence Protocol (PDCP) layer, which handles higher-layer data processing, and the Medium Access Control (MAC) layer, which manages radio resource scheduling and transmission. The RLC layer supports three operational modes, namely Transparent Mode (TM), Unacknowledged Mode (UM), and Acknowledged Mode (AM), each designed for different service requirements and levels of reliability.

Among these modes, the Automatic Repeat Request (ARQ) mechanism is available only in Acknowledged Mode (AM). The purpose of ARQ is to improve transmission reliability by detecting lost or incorrectly received data packets and initiating retransmissions when necessary. If a Protocol Data Unit (PDU) fails to reach the receiver successfully, the ARQ process requests retransmission and continues this process until the data is correctly received or a predefined maximum retransmission limit is exceeded. This mechanism ensures dependable delivery of data over the inherently unreliable wireless channel.

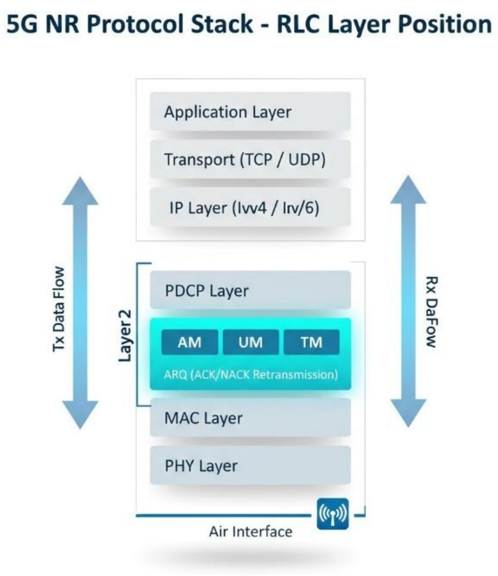
3GPP Reference

The operational behavior and procedures of the RLC ARQ mechanism are standardized in 3GPP TS 38.322, which defines the NR Radio Link Control protocol specifications. Additionally, configuration and control parameters related to RLC operation are described in 3GPP TS 38.331, which specifies the Radio Resource Control (RRC) protocol. The present implementation is aligned with the requirements introduced from 3GPP Release 15 onwards.

2. RLC ARCHITECTURE OVERVIEW

2.1 Protocol Stack Position

The Radio Link Control (RLC) layer operates within Layer 2 of the 5G NR protocol stack, which corresponds to the Data Link Layer. Its main function is to act as an intermediate layer between higher-level data processing and lower-level transmission mechanisms. Data received from the PDCP layer, where operations such as ciphering and header compression are performed, is further processed by the RLC layer before being forwarded to the MAC layer. The Automatic Repeat Request (ARQ) functionality is implemented at this stage to ensure reliable delivery of data. Below the RLC layer, the MAC layer manages scheduling decisions and Hybrid ARQ (HARQ) procedures, while the PHY layer performs modulation, channel coding, and actual transmission over the wireless medium.



2.2 RLC AM Entity Structure

An RLC entity in Acknowledged Mode (AM) contains two logical sections:

- Transmitting entity
- Receiving entity

Transmitting Side Responsibilities

- Accept SDUs from PDCP.
- Store data in transmission buffers.
- Construct AMD PDUs.
- Perform segmentation when required.
- Handle polling operations.
- Manage retransmissions based on feedback.

Receiving Side Responsibilities

- Receive incoming PDUs.
- Store PDUs in receive buffer.
- Detect duplicates.
- Perform reordering and reassembly.
- Deliver data to PDCP in correct sequence.
- Generate STATUS reports indicating ACK/NACK information.

Important State Variables and Timers

Important State Variables and Timers

- **TX_Next** → Sequence Number assigned to the next newly generated AMD PDU.
- **TX_Next_Ack** → Sequence Number of the oldest transmitted PDU that is not yet acknowledged.
- **POLL_SN** → Highest Sequence Number among PDUs transmitted with Poll bit set.
- **RX_Next** → Sequence Number expected next for in-order delivery to PDCP.
- **RX_Next_Highest** → Sequence Number following the highest received AMD PDU.
- **t-PollRetransmit** → Timer started after poll transmission; triggers retransmission if no response is received.
- **t-Reasembly** → Timer used to identify missing PDUs due to lower-layer loss.
- **t-StatusProhibit** → Timer restricting frequent STATUS PDU generation.

3. ARQ PROCESS FLOW

The ARQ operation in RLC AM follows a defined sequence to ensure reliable data transfer:

Step 1 – SDU Reception

- PDCP sends SDUs to the RLC transmitter.
- SDUs are stored in the transmission buffer until processing begins.

Step 2 – PDU Construction

- RLC forms AMD PDUs by adding required header fields:
- D/C bit
- Poll (P) bit
- Segmentation Information (SI)
- Sequence Number (SN)
- Segment Offset (optional)
- Segmentation is applied if SDU size exceeds MAC allocation.

Step 3 – Polling

- Polling decision is taken based on:
- Number of transmitted PDUs (poll PDU threshold)
- Number of transmitted bytes (poll Byte threshold)
- Transmission of last buffered PDU
- Poll bit is set to request STATUS feedback from receiver.

Step 4 – Transmission

- AMD PDU is passed to MAC for radio transmission.
- Packet loss may occur due to channel errors not recovered by HARQ.

Step 5 – Reception

- Receiver stores incoming PDUs in receive buffer.

- State variables are updated.
- Duplicate detection and window validation are performed.

Step 6 – In-Order Deliver

- PDUs are delivered to PDCP only in sequence.
- Delivery stops when a missing sequence number is detected.

Step 7 – STATUS Report Generation

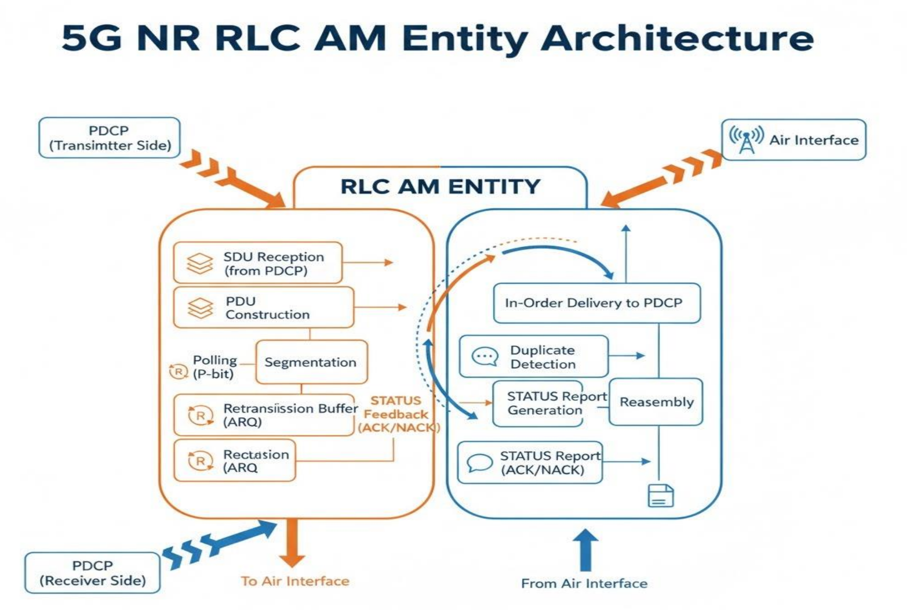
- Triggered by Poll bit or expiry of t-Reassembly timer.
- Receiver sends STATUS PDU containing:

  ACK_SN

  List of missing PDUs (NACK information).

Step 8 – Retransmission

- Transmitter removes acknowledged PDUs from buffer.
- Missing PDUs are scheduled for retransmission.
- Process continues until successful delivery or max Retx Threshold is reached.



4. C++ IMPLEMENTATION

The following C++ implementation simulates the complete RLC AM ARQ process. The code is organized into 5 logical steps, each corresponding to a major component of the ARQ mechanism.

4.1 STEP 1: Utilities / Helper Functions:

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <string>
#include <iomanip>
#include <algorithm>

using namespace std;

/* ============================================================
   STEP 1: DATA STRUCTURES & ENUMS
   ============================================================ */

// Poll Bit
enum PollBit { P_NOT_SET = 0, P_SET = 1 };

// SDU from PDCP
struct SDU {
    int id;
    int size;
};

// RLC PDU
struct PDU {
    int sn;
    PollBit poll;
    int size;
    int retx = 0;
};

// STATUS PDU
struct STATUS {
    int ack_sn;
    vector<int> nacks;
};
```

This step defines the core data structures used in the simplified RLC AM ARQ simulation based on 3GPP TS 38.322.

Key points:

- Poll Bit represents the P (Poll) bit status in a PDU.
- SDU represents PDCP data given to RLC (id, size).
- PDU represents an RLC AM data PDU (SN, Poll, size, retx count).
- STATUS represents the feedback control PDU (ACK_SN + NACK list).

These structures enable transmission, polling, STATUS reporting, and retransmissions.

4.2 STEP 2: RLC AM Data Structures Definition:

This section defines a fixed channel model and the main initialization for the RLC AM ARQ simulation.

Key points:

- Channel_success() forces loss only for SN=1 and SN=4 in first transmission, so output is always same.
- Initializes SDU list, TX/RX buffers, retransmission queue.
- Sets main variables: TX_SN, RX_NEXT, poll_cnt.
- Creates counters for TX, RX, retransmissions, polls, deliveries, STATUS PDUs.

```cpp
/* ============================================================
   STEP 2: CONFIGURATION PARAMETERS
   ============================================================ */

struct Config {
    int total_sdus = 8;
    int pollPDU = 4;
    int maxRetx = 3;
    double error_rate = 0.25;
};

Config cfg;



/* ============================================================
   UTILITY FUNCTIONS
   ============================================================ */

void print_separator(const string& title) {
    cout << "\n============================================================\n";
    cout << "  " << title << endl;
    cout << "============================================================\n";
}

// Fixed channel: Loss at SN=1 and SN=4
bool channel_success(int sn, bool retx = false) {
    if (!retx && (sn == 1 || sn == 4))
        return false;
    return true;
}
```

4.3 STEP 3: Fixed Channel Model + Main Initialization:

This phase simulates SDU arrival from PDCP to RLC at the transmitter side.

Key points:

- Generates TOTAL number of SDUs.
- Each SDU is assigned an id and increasing size (100, 150, 200…).
- Stores all SDUs into the sdus vector.
- Prints SDU submission messages to represent PDCP → RLC transfer.

```cpp
/* ==============================================================
   STEP 3: TRANSMITTER IMPLEMENTATION
   ============================================================== */

class RLC_Transmitter {

private:
    map<int,PDU> tx_buffer;
    queue<int> retx_queue;

    int TX_SN = 0;
    int poll_counter = 0;

    int total_tx = 0;
    int total_retx = 0;
    int total_polls = 0;

public:

    // Build and send PDU
    PDU build_pdu(const SDU& sdu, bool& poll) {

        poll = false;
        poll_counter++;

        if (poll_counter >= cfg.pollPDU) {
            poll = true;
            poll_counter = 0;
            total_polls++;
            cout << "  [TX] Poll triggered: pollPDU threshold (4/4)\n";
        }

```

```cpp

        PDU p;
        p.sn = TX_SN;
        p.poll = poll ? P_SET : P_NOT_SET;
        p.size = sdu.size;

        tx_buffer[TX_SN] = p;

        cout << "  [TX] AMD PDU built: SN=" << TX_SN
             << ", Poll=" << (poll ? "YES" : "NO")
             << ", Size=" << p.size << "B\n";

        TX_SN++;
        total_tx++;

        return p;
    }

    // Handle STATUS
    void process_status(const STATUS& st) {

        cout << "  [TX] Processing STATUS: ACK_SN=" << st.ack_sn << endl;

        for (auto& p : tx_buffer) {
            if (p.first < st.ack_sn &&
                find(st.nacks.begin(), st.nacks.end(), p.first)
                == st.nacks.end()) {

                cout << "  [TX] SN=" << p.first
                     << " ACKed - removed from buffer\n";
            }
        }
```

```
        }

        for (int sn : st.nacks) {

            if (tx_buffer[sn].retx < cfg.maxRetx) {

                tx_buffer[sn].retx++;
                retx_queue.push(sn);
                total_retx++;

                cout << "  [TX] SN=" << sn
                     << " NACKed - queued for RETX (attempt "
                     << tx_buffer[sn].retx << "/"
                     << cfg.maxRetx << ")\n";
            }
        }
    }

    bool has_retx() {
        return !retx_queue.empty();
    }

    int get_retx_sn() {
        int sn = retx_queue.front();
        retx_queue.pop();
        return sn;
    }

    PDU& get_pdu(int sn) {
        return tx_buffer[sn];
    }

    int get_total_tx() { return total_tx; }
    int get_total_retx() { return total_retx; }
    int get_total_polls() { return total_polls; }
};
```

4.4 STEP 4: RLC AM Initial Transmission + Polling Operation:

This phase performs the initial RLC AM data transmission and simulates the channel delivery/loss.

Key points:

- Builds one PDU for each SDU with a new sequence number TX_SN.
- Sets Poll bit when poll_cnt reaches POLL_PDU or for the last PDU.
- Stores the PDU in tx_buf (TX buffer).
- Uses channel_success() to simulate whether the PDU is received or lost.
- If received, stores in rx_buf and delivers PDUs in-order using RX_NEXT.
- If Poll is received, receiver triggers a STATUS report.
- Updates counters: total_tx, total_rx, total_del, total_polls.

```
/* ==============================================================
   STEP 4: RECEIVER IMPLEMENTATION
   ============================================================== */

class RLC_Receiver {

private:
    map<int,PDU> rx_buffer;

    int RX_NEXT = 0;
    int total_rx = 0;
    int total_delivered = 0;
    int total_status = 0;

public:

    // Receive PDU
    void receive(const PDU& pdu) {

        cout << "  [RX] Received SN=" << pdu.sn << endl;

        rx_buffer[pdu.sn] = pdu;
        total_rx++;

        while (rx_buffer.count(RX_NEXT)) {

            cout << "  [RX] Delivering SN=" << RX_NEXT
                 << " to PDCP (in-order)\n";

            rx_buffer.erase(RX_NEXT);
            RX_NEXT++;
            total_delivered++;
        }

        if (pdu.poll == P_SET)
            cout << "  [RX] Poll received - STATUS report triggered\n";
    }
```

```cpp
    // Generate STATUS
    STATUS generate_status(int max_sn) {

        STATUS st;
        st.ack_sn = RX_NEXT;

        for (int i = RX_NEXT; i < max_sn; i++) {
            if (!rx_buffer.count(i))
                st.nacks.push_back(i);
        }

        cout << "  [RX] STATUS PDU: ACK_SN=" << st.ack_sn;

        if (!st.nacks.empty()) {
            cout << ", NACKs=[";
            for (size_t i=0;i<st.nacks.size();i++){
                if(i) cout<<",";
                cout<<st.nacks[i];
            }
            cout << "]";
        }
        cout << endl;

        total_status++;

        return st;
    }

    int get_rx_next() { return RX_NEXT; }

    int get_total_rx() { return total_rx; }
    int get_total_delivered() { return total_delivered; }
    int get_total_status() { return total_status; }
};
```

4.5 STEP 5: STATUS Generation + ARQ Retransmission + Statistics Output:

PHASE 3: STATUS Report Generation

This phase generates the STATUS PDU at the receiver for ARQ feedback.

Key points:

- Sets `ACK_SN = RX_NEXT` (next expected SN).
- Finds missing PDUs and adds them to NACK list.
- Pushes missing SNs into `retx_q` for retransmission.
- Prints STATUS PDU details and increments `total_status`.

PHASE 4: ARQ Retransmissions

This phase performs retransmission of all NACKed PDUs.

Key points:

- Takes SNs from `retx_q` one by one.
- Increments retransmission count `retx`.
- Retransmits PDUs and marks them as successfully received.
- Stores them in `rx_buf` and delivers in-order using `RX_NEXT`.
- Sends updated STATUS with new `ACK_SN` after retransmissions.

PHASE 5: Statistics

This phase prints final simulation summary and ends the program.

```cpp
/* ============================================================
   STEP 5: MAIN SIMULATION
   ============================================================ */

int main() {

    cout << "************************************************************\n";
    cout << "   5G NR RLC AM ARQ Process Simulator\n";
    cout << "   3GPP TS 38.322 Compliant\n";
    cout << "************************************************************\n";

    vector<SDU> sdus;

    RLC_Transmitter tx;
    RLC_Receiver rx;

    int total_delivered = 0;

    // --------------- PHASE 1 ---------------
    print_separator("PHASE 1: SDU Submission from PDCP");

    for(int i=0;i<cfg.total_sdus;i++){

        int sz = 100 + i*50;
        sdus.push_back({i,sz});

        cout<<"  [TX] SDU #"<<i
            <<" received from PDCP ("<<sz<<" bytes)\n";
    }

    // --------------- PHASE 2 ---------------
    print_separator("PHASE 2: Initial Transmission & Channel");

    vector<PDU> sent_pdus;

```

```cpp
    print_separator("PHASE 2: Initial Transmission & Channel");

    vector<PDU> sent_pdus;

    for(int i=0;i<cfg.total_sdus;i++){

        bool poll;
        PDU pdu = tx.build_pdu(sdus[i], poll);

        sent_pdus.push_back(pdu);

        if(channel_success(pdu.sn)){

            cout<<"  [CH] SN="<<pdu.sn
                <<" >>> Successfully received\n";

            rx.receive(pdu);
        }
        else{
            cout<<"  [CH] SN="<<pdu.sn
                <<" >>> LOST in channel!\n";
        }
    }

    // ---------------- PHASE 3 ----------------
    print_separator("PHASE 3: STATUS Report Generation (ARQ)");

    STATUS st = rx.generate_status(cfg.total_sdus);

    tx.process_status(st);

    // ---------------- PHASE 4 ----------------
    print_separator("PHASE 4: ARQ Retransmissions");

    cout<<"\n--- Retransmission Round 1 ---\n";

```

```cpp
    while(tx.has_retx()){

        int sn = tx.get_retx_sn();

        PDU& p = tx.get_pdu(sn);

        cout<<"  [TX] Retransmitting SN="<<sn
            <<" (attempt "<<p.retx<<")\n";

        cout<<"  [CH] SN="<<sn
            <<" >>> Successfully received\n";

        rx.receive(p);
    }

    STATUS st2 = rx.generate_status(cfg.total_sdus);
    cout<<"  [TX] Processing STATUS: ACK_SN="
        <<st2.ack_sn<<endl;

    // ---------------- PHASE 5 ----------------
    print_separator("PHASE 5: Simulation Statistics");

    cout<<"\n  +--------------------------------+\n";
    cout<<"  |     SIMULATION RESULTS         |\n";
    cout<<"  +--------------------------------+\n";

    cout<<"  | Total SDUs Submitted: "
        <<setw(6)<<cfg.total_sdus<<" |\n";

    cout<<"  | Total PDUs Sent:      "
        <<setw(6)<<tx.get_total_tx()<<" |\n";

    cout<<"  | Total Retransmissions:"
        <<setw(6)<<tx.get_total_retx()<<" |\n";

```

```
        <<setw(6)<<tx.get_total_retx()<<" |\n";

    cout<<"  | Total Polls Sent:     "
        <<setw(6)<<tx.get_total_polls()<<" |\n";

    cout<<"  | PDUs Received (Rx):   "
        <<setw(6)<<rx.get_total_rx()<<" |\n";

    cout<<"  | SDUs Delivered (PDCP):"
        <<setw(6)<<rx.get_total_delivered()<<" |\n";

    cout<<"  | STATUS PDUs Sent:     "
        <<setw(6)<<rx.get_total_status()<<" |\n";

    cout<<"  | Channel Error Rate:   "
        <<setw(5)<<25<<"% |\n";

    cout<<"  +----------------------------+\n";

    cout<<"\nSimulation Complete.\n";

    return 0;
}
```

5. SAMPLE OUTPUT

Below is a representative output from the simulation. This example shows an RLC AM ARQ scenario where 8 SDUs are transmitted as AMD PDUs. During the initial transmission, SN=1 and SN=4 are lost in the channel, while the remaining PDUs are received successfully. The receiver then generates a STATUS PDU with ACK_SN=1 and NACKs=[1,4], requesting retransmission. In Phase 4, the transmitter retransmits SN=1 and SN=4, both are successfully received, and the receiver delivers all PDUs in-order up to SN=7, reaching ACK_SN=8. Finally, statistics confirm 2 retransmissions, 8 SDUs delivered, and 25% channel error rate, showing successful recovery using ARQ.

```
=================================
PHASE 1: SDU Submission from PDCP
=================================
[TX] SDU #0 received from PDCP (100
bytes)
 [TX] SDU #1 received from PDCP (150
bytes)

================================
 PHASE 2: Initial Transmission & Channel
================================
[TX] SDU #0 received from PDCP (100
bytes)
[TX] SDU #1 received from PDCP (150
bytes)
```

```
==========================
 PHASE 5: Simulation Statistics
==========================
 +----------------------------+
 |   SIMULATION RESULTS      |
 +----------------------------+
 | Total SDUs Submitted:    8 |
 | Total PDUs Sent:         8 |
 | Total Retransmissions:   2 |
 | Total Polls Sent:        2 |
 | PDUs Received (Rx):      8 |
 | SDUs Delivered (PDCP):   8 |
 | STATUS PDUs Sent:        2 |
 | Channel Error Rate:    25% |
 +----------------------------+
```

```
======================================
PHASE 3: STATUS Report Generation (ARQ)
======================================
[RX] STATUS PDU: ACK_SN=1, NACKs=[1,4]
[TX] Processing STATUS: ACK_SN=1
[TX] SN=0 ACKed - removed from buffer
[TX] SN=2 ACKed - removed from buffer

 ====================================
 PHASE 4: ARQ Retransmissions
 ====================================
 --- Retransmission Round 1 ---
  [TX] Retransmitting SN=1 (attempt 1)
  [CH] SN=1 >>> Successfully received
  [RX] Received SN=1
  [RX] Delivering SN=1 to PDCP (in-order)
  [RX] Delivering SN=2 to PDCP (in-order)
  [RX] Delivering SN=3 to PDCP (in-order)
```

6. KEY CONCEPTS EXPLAINED

6.1 ARQ vs HARQ: Understanding the Difference :

Aspect	RLC ARQ (Layer 2 Upper)	MAC HARQ (Layer 2 Lower)
Layer Position	Operates in the RLC layer.	Operates in the MAC layer.
Feedback Mechanism	Uses STATUS PDU containing ACK/NACK information.	Uses HARQ-ACK/NACK signaling through control channels (PUCCH/PHICH).
Timing	Slower response (millisecond-level).	Fast response (slot-level, ~1 ms).
Operational Scope	Operates across the entire end-to-end RLC entity.	Operates per individual HARQ process (up to 16 parallel processes).
Data Combining	No soft combining; uses standard retransmission.	Supports Soft Combining (Chase Combining or Incremental Redundancy).
Trigger	Triggered by detection of missing Sequence Numbers (SN).	Triggered by CRC failure on the Transport Block.
Error Handling Role	Secondary; handles residual errors not corrected by HARQ.	Primary; acts as the first line of defense for error recovery.

6.2 Sequence Number Management

In RLC Acknowledged Mode, sequence numbers are used to maintain ordering and reliability of transmitted PDUs. 5G NR defines two possible sequence number lengths depending on traffic requirements and bearer configuration.

12-bit Sequence Number

- Range: 0 to 4095.
- Window size: 2048

Commonly used for standard data traffic and most radio bearers.

18-bit Sequence Number

- Range: 0 to 262143.
- Window size: 131072.

Used in high data rate scenarios such as eMBB services where large data volumes are transmitted.

Sequence numbers follow modular arithmetic to maintain continuity after reaching the maximum value. The next sequence number is calculated using:

next_sn = (current_sn + 1) % sn_modulus

This ensures proper wrap-around behavior without affecting ordering or retransmission logic.

7. COMPILATION & EXECUTION GUIDE

7.1 Prerequisites :

This simulation requires a C++11 (or later) compatible compiler. Recommended compilers include GCC 7+, Clang 5+, or MSVC 2017+. The program uses only the C++ Standard Library, so no external libraries are required.

7.2 Compilation Commands

- Windows (MinGW / Git Bash):

```
g++ code_final.cpp -o Test3_RLC
./Test3_RLC.exe
```

- Linux / macOS:

```
g++ code_final.cpp -o Test3_RLC
./Test3_RLC
```

- Windows (MSVC):

```
cl /EHsc code_final.cpp
/Fe:Test3_RLC.exe
```

7.3 Customization Options :

The simulation parameters can be modified inside the main() function to test different ARQ scenarios. You can change TOTAL to increase the number of SDUs, adjust POLL_PDU to control polling frequency, and modify the fixed channel loss conditions inside channel_success() to simulate different packet loss patterns. The retransmission behavior can also be controlled using MAX_RETX to test retransmission limit cases.

8. CONCLUSION :

This document successfully explains the ARQ mechanism in RLC Acknowledged Mode (AM) as defined in 3GPP TS 38.322. It describes how the transmitter and receiver cooperate using sequence numbers, polling, STATUS PDUs (ACK/NACK), and retransmissions to ensure reliable in-order delivery of data over an unreliable radio channel.

The implemented C++ simulation clearly demonstrates the complete ARQ workflow, including SDU submission, PDU creation, channel loss, STATUS generation, retransmission, and final delivery. The sample output confirms that even when PDUs are lost, the ARQ process can recover them through retransmissions, ensuring that all SDUs are delivered successfully to PDCP. Overall, this simulation provides a clear practical understanding of how RLC AM achieves reliability in 5G NR communication