



## 5. Übung zur Vorlesung Rechnernetze (Musterlösung)

Abgabe: 17.12.2004

### Aufgabe 1: HDLC (5 Punkte, 1 + 2 + 2)

- a) Welche Rahmenarten gibt es in HDLC und wofür werden sie verwendet?
- b) Untersuchen Sie die beiden folgenden HDLC-Pakete! Identifizieren Sie alle einzelnen Bestandteile des HDLC-Rahmens, und geben Sie für jeden Teil seine genaue Bedeutung an! Um welche Art von Rahmen handelt es sich jeweils? (Hinweis: Gehen Sie davon aus, dass das Adressfeld 1 Byte lang ist und die CRC-Checksumme mittels eines Generatorpolynoms vom Grad 16 ermittelt wurde).
- i) 7E 83 71 74 82 7E
- ii) 7E 83 00 A1 B3 82 4C 9D 6F E4 11 01 02 45 6B 71 80 7E
- c) HDLC verwendet ein Schiebefensterprotokoll zur Flusskontrolle mit der maximalen Fenstergröße 7. Nehmen Sie an, Stationen A und B seien über einen Kanal mit einer Kapazität von 100 Mbps verbunden. Die beiden Stationen seien 10 km voneinander entfernt. Die Ausbreitungsgeschwindigkeit des Signals betrage 200.000 km/s. Es wird von einer Rahmengröße von 1000 Bit inklusive Header ausgegangen. Jedes Paket wird vom Empfänger bestätigt. Gehen Sie ferner davon aus, dass die Länge der Bestätigungen zu vernachlässigen ist. Reicht die maximale Fenstergröße 7 bereits aus, um eine Effizienz von 100% zu erreichen, wenn keine Übertragungsfehler auftreten? Ab welcher Fenstergröße wird eine Effizienz von 100% erreicht, wenn keine Übertragungsfehler auftreten? *Hinweis: Verwenden sie die Formeln zur Berechnung der Effizienz aus der vergangenen Übungsstunde (siehe <http://www-vs.informatik.uni-ulm.de/teach/ws04/rn1/Fehlerkontrolle.pdf>). Auf den Folien wird der sendersseitige Timeout auf  $2 * \text{Signallaufzeit}$  gesetzt. In der Praxis wird meist ein längerer Timeout verwendet.*

### Lösungsvorschlag:

- a) *I-Frame (Information Frame): Datenübertragung, Huckepack-Transport von empfangenen Sequenznummern*  
*S-Frame (Supervisory Frame): Flusskontrolle. Explizite Bestätigungen falls kein Huckepack verwendet wird, um empfangene Sequenznummern zu signalisieren. Explizite Anforderung von Paketwiederholung.*  
*U-Frame (Unnumbered Frame): Versenden von Command/Reply zum Verbandsaufbau und -abbau, Einstellen des Response-Mode, Rücksetzen der Sequenzzähler, Austauschen von Identifiern, Senden von Unnumbered Information Frames.*
- b) *Der Rahmen beginnt und endet mit einem Flag zur Oktettsynchronisierung und Rahmenbegrenzung (7E), es folgt die 1 Byte lange Adresse der Sekundärstation (83) und das 1 Byte lange Kontrollfeld. Vor dem abschließenden Flag (7E) befindet sich eine 2 Byte lange CRC Checksumme (74 82 bzw. 71 80).*

i) 7E 83 71 74 82 7E

*Das Kontrollfeld 71 (= 01110001) des ersten Pakets endet mit „01“; somit handelt es sich um einen Supervisory Frame. Das Poll-Bit ist auf 1 gesetzt und der Command/Reply Code „00“ bedeutet Receive Ready, somit fordert die Primärstation mit dieser Nachricht ein Paket von der Sekundärstation an. Die Sequenznummer des erwarteten Pakets ist „011“.*

ii) 7E 83 00 A1 B3 82 4C 9D 6F E4 11 01 02 45 6B 71 80 7E

*Das Kontrollfeld 00 (= 00000000) des zweiten Pakets endet mit „0“, somit handelt es sich um einen Information Frame. Dieser kann theoretisch von der Primär- oder von der Sekundärstation stammen. Das nicht gesetzte Final-Bit bedeutet, dass es sich noch nicht um das letzte Paket handelt. Die Sequenznummer des Pakets ist „000“, und von der Gegenseite wird als nächstes auch die Sequenznummer „000“ erwartet.*

- c) *Maximale Fenstergröße:*  $W = 7$   
*Kanalkapazität:*  $C = 100 \text{ Mbps}$

Rahmenlänge:	$F = 1000 \text{ Bit}$
Ausbreitungsverzögerung:	$PROP = 10 \text{ km} / 200000 \text{ km/s} = 1/20000 \text{ s}$
Dauer, bis Rahmen auf der Leitung:	$TRANS\_P = F/C = 1000 / 100\,000\,000 = 1/100\,000 \text{ s}$
Erste Rückmeldung frühestens nach:	$T2 = 2 \cdot PROP + TRANS\_P = 1/10000 + 1/100000 = 0,00011 \text{ s}$
Sender darf ohne Unterbrechung senden, falls	$T2 \leq W \cdot TRANS\_P = 7/100000 \text{ s} = 0,00007 \text{ s}$
Sonst berechnet sich die Effizienz durch	$U = w / (1 + 2 \cdot (Prop / Trans\_p))$
	$U = 7 / (1 + 2 \cdot ((1/20000) / (1/100000))) = 7 / 11 = 63\%$
	$W \cdot TRANS\_P \geq 2 \cdot PROP + TRANS\_P$
→	$W \geq (2 \cdot PROP / TRANS\_P) + 1 = ((1/10000) / (1/100000)) + 1 = 11$

Ab einer Fenstergröße von 11 wird eine Effizienz von 100% erreicht, wenn keine Übertragungsfehler auftreten.

## Aufgabe 2: Programmieraufgabe File-Transfer (5 Punkte)

Wir wollen unser Programm vom letzten Übungsblatt erneut etwas verbessern. Laden Sie sich zunächst den Java-Code aus folgendem Archiv herunter: [http://www-vs.informatik.uni-ulm.de/teach/ws04/rn1/OSI\\_Aufgabe3.zip](http://www-vs.informatik.uni-ulm.de/teach/ws04/rn1/OSI_Aufgabe3.zip). (Achtung: Wir haben die Implementierung der Layer4 Klasse etwas verändert, sodass nun auch Bitfehler bei der Übertragung simuliert werden).

Erweitern Sie das Rahmenformat der Layer7-Pakete vom letzten Übungsblatt um eine 16-Bit lange CRC Checksumme am Ende des Pakets. Die CRC Checksumme soll mittels des folgenden Generator-Polynoms erzeugt werden:  $x^{16} + x^{12} + x + 1$ . Die Pakete sollen weiterhin eine Maximallänge von insgesamt 100 Byte haben. Abzüglich des 2-Byte langen Headers und der 2 Byte langen Checksumme kann ein Paket also maximal 96 Datenbytes enthalten. Entdeckt der Empfänger einen Übertragungsfehler bei einem Daten-, Start\_of\_file-, oder End\_of\_file-Paket, so soll er sofort ein explizites Negative\_Acknowledge (Pakettyp 5) mit der Sequenznummer des fehlerhaften Paketes an den Sender schicken, welcher das fehlerhafte Paket daraufhin sofort wiederholt anstatt den Timeout für das Acknowledge abzuwarten. Wird ein Übertragungsfehler bei einem Acknowledge oder Negative\_Acknowledge entdeckt, so soll das fehlerhafte Paket einfach ignoriert werden, (was zum Ablauf des Timeouts führt). Die Klasse Layer7 soll jedes Mal, wenn sie einen Übertragungsfehler entdeckt hat, eine entsprechende Meldung auf dem Bildschirm ausgeben, mit der Sequenznummer und dem Typ des fehlerhaften Pakets. Auf Senderseite soll außerdem bei jedem Erhalt eines Negative\_Acknowledge eine Meldung mit der Sequenznummer des fehlerhaften Pakets auf dem Bildschirm ausgegeben werden.

Geben Sie bitte den kompletten Quelltext ausgedruckt oder per Email ab! Schicken Sie den gesamten Quelltext gezippt an: [schorr@informatik.uni-ulm.de](mailto:schorr@informatik.uni-ulm.de) mit dem Betreff „RN1 – Übung 5“.

*Lösungsvorschlag:*

*Die fett gedruckten Zeilen in dem folgenden Programmcode wurden gegenüber dem alten Programmcode neu hinzugefügt.*

```
public class Layer7
{
    static final int PacketMaxLen = 100;
    static final int Timeout = 500;

    private byte _sequenceNumber = 0;
    private Layer4 _layer4Instance;

    /**
     * CRC polynom, binary: 0000 0000 0000 0001 0001 0000 0000 0011,
     * hex : 0 0 0 1 1 0 0 3
     */
    static final int _crcPoly = 0x00011003;

    /**
     * Internal register for calculating the crc checksum
     */
    private int _register = 0;

    // ----- //

    /**
     * Constructor
     *
     * @param client If true, this Layer7 instance acts as a client which is
     * transmitting data to a server. If false, this Layer7
     * instance acts as a server, which is listening for incoming
     * data packets.
     */
    public Layer7(boolean client)
    {
        _layer4Instance = new Layer4(client);
    }
}
```

```

}

// ----- //

/**
 * Call AddBit() for all message bits. Before you call AddBit() the first
 * time, _register must be 0.
 *
 * @param nextBit Must be either 1 or 0.
 */
private void AddBit(int nextBit)
{
    int xorInput;

    // Shift the nextBit into the _register
    _register = (_register << 1) + nextBit;

    // Check whether register contains enough bits.
    if (_register >= 0x00010000)
    {
        // If yes, we can subtract (modulo 2) the _crcPoly
        xorInput = _crcPoly;
    }
    else
    {
        // If no, we cannot subtract.
        xorInput = 0x00000000;
    }

    // Subtraction modulo 2 is the same as a bitwise XOR operation.
    _register = _register ^ xorInput;
}

// ----- //

/**
 * Calculates and fills the CRC checksum field of a packet.
 *
 * @param packet Contains the data packet for which the checksum shall be
 *                be calculated (including the 16 appended zero-bits).
 * @param datalen The length of the data packet including the two bytes
 *                for the checksum. The actual size of the array may be
 *                longer than datalen.
 */
private void CalculateChecksum(byte[] packet, int datalen)
{
    _register = 0;

    // Iterate over all bytes of the packet except the last two.
    for (int i = 0; i < datalen; i++)
    {
        // Now iterate over all bits of the current byte.
        for (int j = 7; j >= 0; j--)
        {
            AddBit((packet[i] >>> j) & 0x01);
        }
    }

    // Now append the checksum to the packet.
    packet[datalen - 2] = (byte)((_register >>> 8) & 0x000000FF);
    packet[datalen - 1] = (byte)(_register & 0x000000FF);
}

// ----- //

/**
 * Test whether the checksum at the end of the packet is correct.
 *
 * @param packet Contains the data packet (including the checksum). The
 *                size of the array may be longer than the actual number of
 *                data bytes + checksum.
 * @param length The actual number of data bytes including the checksum.
 * @return Returns true if the checksum is correct.
 */
private boolean TestChecksum(byte[] packet, int length)
{
    _register = 0;

```

```

// Iterate over all bytes of the packet.
for (int i = 0; i < length; i++)
{
    // Now iterate over all bits of the current byte.
    for (int j = 7; j >= 0; j--)
    {
        AddBit((packet[i] >>> j) & 0x01);
    }
}

if (_register == 0) { return true; }
else                { return false; }
}

// ----- //

/**
 * Transfers a file over the network
 *
 * @param ipAddress The IP address of the receiver.
 * @param filename  The name of the file.
 * @return void
 */
public void SendFile(String ipAddress, String filename)
{
    byte[] packet;
    int    available;

    // The sequence number of the first packet should always be 0.
    _sequenceNumber = 0;

    try
    {
        // Open the file.
        FileInputStream in = new FileInputStream(filename);

        // Create a new network packet (this is only the data part of the packet,
        // the SendUnitDataRequest() method will add the header).
        if (filename.length() > PacketMaxLen - 4)
        {
            System.out.println("Sorry, filename mustn't be longer than "
                + (PacketMaxLen - 4) + " bytes.");
            return;
        }
        packet = new byte[filename.length()];
        // Copy the filename into the network packet.
        for (int i = 0; i < filename.length(); i++)
        {
            packet[i] = (filename.getBytes())[i];
        }

        // Send start_of_file packet.
        SendUnitDataRequest(ipAddress, packet, (byte)1);

        // Read file content until we have reached the end of the file.
        while ((available = in.available()) > 0)
        {
            if (available > PacketMaxLen - 4)
            {
                packet = new byte[PacketMaxLen - 4];
                in.read(packet, 0, PacketMaxLen - 4);
            }
            else
            {
                packet = new byte[available];
                in.read(packet, 0, available);
            }

            // Send the data packet.
            SendUnitDataRequest(ipAddress, packet, (byte)2);
        }

        // Create an end-of-file packet (contains no data).
        packet = new byte[0];
        // Send end_of_file packet.
        SendUnitDataRequest(ipAddress, packet, (byte)4);

        // Close the input file.

```

```

        in.close();
    }
    catch(FileNotFoundException e)
    {
        e.printStackTrace(); System.exit(-1);
    }
    catch (IOException e)
    {
        e.printStackTrace(); System.exit(-1);
    }
}

// ----- //

/**
 * Transmits a packet to the specified receiver address.
 *
 * @param ipAddress The IP address of the receiver.
 * @param buffer The packet
 * @param type Defines the packet type (1 = start_of_file, 2 = data,
 *         3 = acknowledge, 4 = end_of_file)
 * @return 0 if operation was successful
 */
public int SendUnitDataRequest(String ip, byte[] buffer, byte type)
{
    int i, j;
    boolean receivedAck = false;

    // Create a buffer capable of storing the 2-byte header, the 2-byte
    // checksum and the payload.
    byte[] packet = new byte[buffer.length + 4];
    // Create a buffer capable of storing the incoming acknowledge packet.
    byte[] ack = new byte[4];

    // Set the sequence number of the data packet.
    packet[0] = _sequenceNumber;
    // Set the packet type number.
    packet[1] = type;
    // Copy the payload into the data packet.
    for (i = 0; i < buffer.length; i++)
    {
        packet[i + 2] = buffer[i];
    }
    // Add 16 zero-bits.
    packet[buffer.length + 2] = 0;
    packet[buffer.length + 3] = 0;
    // Now calculate the CRC checksum.
    CalculateChecksum(packet, buffer.length + 4);

    // Now send the message to the receiver.
    _layer4Instance.SendUnitDataRequest(ip, packet);
    System.out.println("- Layer7: Sent packet number: " + packet[0]);

    while (receivedAck == false)
    {
        if (_layer4Instance.CheckForUnitDataConfirmation(Timeout, ack) == 0)
        {
            System.out.println("- Layer7: Timeout occured while awaiting ack: "
                + _sequenceNumber);
            // Now send the message to the receiver.
            _layer4Instance.SendUnitDataRequest(ip, packet);
        }
        else if (ack[1] == 5)
        {
            System.out.println("--- Layer7: Received NACK for packet: " + ack[0]);
            // Now send the message to the receiver.
            _layer4Instance.SendUnitDataRequest(ip, packet);
        }
        else if ( (ack[0] != _sequenceNumber) || (ack[1] != 3)
            || (TestChecksum(ack, 4) == false))
        {
            System.out.println("- Layer7: Received corrupt ACK");
        }
        else
        {
            System.out.println("- Layer7: Received correct ACK number: " + ack[0]);
            receivedAck = true;
        }
    }
}

```

```

    // Now we can increment the sequence number.
    _sequenceNumber++;

    return 0;
}

// ----- //

/**
 * Waits for a file transfer and stores the incoming file at the specified
 * destinationDirectory.
 *
 * @param destinationDirectory The directory, where the incoming file will
 *                             will be stored.
 * @return void
 */
public void ReceiveFile(String destinationDirectory)
{
    byte[]          packet = new byte[PacketMaxLen];
    int             packet_length;
    String          filename;
    FileOutputStream out;

    _sequenceNumber = 0;

    // Wait for incoming network packet.
    packet_length = ReceiveUnitData(0, packet);
    // The first packet must be a start_of_file packet.
    while (packet[1] != 1)
    {
        packet_length = ReceiveUnitData(0, packet);
    }

    try
    {
        filename = new String(packet, 2, packet_length - 4);

        // Create a new file in the specified directory.
        out = new FileOutputStream(destinationDirectory + filename);

        // Wait for incoming data packets.
        packet_length = ReceiveUnitData(0, packet);
        while (packet[1] == 2)
        {
            out.write(packet, 2, packet_length - 4);
            // Now wait for the next packet.

            packet_length = ReceiveUnitData(0, packet);
        }

        // Check whether the last packet was an end-of-file packet.
        if (packet[1] != 4)
        {
            // If not, we have a problem.
            System.exit(-1);
        }
        else
        {
            System.out.println("Received end_of_file.");
            // Close the output file.
            out.close();
        }
    }
    catch (IOException e)
    {
        e.printStackTrace();
        System.exit(-1);
    }
}

// ----- //

/**
 * Waits for an incoming network packet.
 *
 * @param timeout The amount of time (in milliseconds) that we want to wait
 * @param buffer The incoming data packet (including the header) will be put
 *               into this buffer.
 * @return The packet length
 */

```

```

*/
private int ReceiveUnitData(int timeout, byte[] buffer)
{
    // Create a buffer capable of storing the incoming data packet
    byte packet[] = new byte[PacketMaxLen];
    // Create a buffer capable of storing the acknowledge packet.
    byte ack[] = new byte[4];
    // Create a buffer capable of storing a negative acknowledge packet.
    byte nack[] = new byte[4];
    int packetLength;
    boolean checksumOK;

    // Now we wait for incoming packets.
    packetLength = _layer4Instance.CheckForUnitDataIndication(0, packet);
    // Check whether the checksum is correct.
    checksumOK = TestChecksum(packet, packetLength);

    // Now check, whether the packet is correct.
    while ((packet[0] != _sequenceNumber) || (checksumOK == false))
    {
        // If checksum is not correct, send a NACK packet.
        if (checksumOK == false)
        {
            // Set the header fields and append 16 zero-bits.
            nack[0] = packet[0]; nack[1] = 5; nack[2] = 0; nack[3] = 0;
            // Now calculate the CRC checksum.
            CalculateChecksum(nack, 4);

            // Send the NACK message to the sender.
            _layer4Instance.SendUnitDataResponse(nack);
            System.out.println("--- Layer7: Sent NACK for packet number: " + nack[0]);
        }
        else
        {
            // Although the sequence number was wrong, we still send an ACK.
            // Otherwise, the sender would transmit the same packet again.
            ack[0] = packet[0]; ack[1] = 3; ack[2] = 0; ack[3] = 0;
            // Now calculate the CRC checksum.
            CalculateChecksum(ack, 4);

            // Send the ACK message to the sender.
            _layer4Instance.SendUnitDataResponse(ack);
        }

        // Now we wait again for the next packet.
        packetLength = _layer4Instance.CheckForUnitDataIndication(0, packet);
        // Check whether the checksum is correct.
        checksumOK = TestChecksum(packet, packetLength);
    }

    // Now we construct an acknowledge packet. The acknowledge shall bear the
    // same sequence number as the last packet received.
    ack[0] = packet[0]; ack[1] = 3; ack[2] = 0; ack[3] = 0;
    // Now calculate the CRC checksum.
    CalculateChecksum(ack, 4);

    // Now send the acknowledge packet to the other peer.
    _layer4Instance.SendUnitDataResponse(ack);
    _sequenceNumber++;

    // Now put the received data into the buffer provided by the user.
    for(int i = 0; i < packetLength; i++)
    {
        buffer[i] = packet[i];
    }

    return packetLength;
}
}

```