SEG3155: Lab 3

Medium Access Control using HDLC

Sam Shteinman

Daniel Godfrey

Date: March 11th, 2011

Contents

[Introduction 2](#_Toc287275911)

[Objectives 2](#_Toc287275912)

[1. Get familiar with medium access control. 2](#_Toc287275913)

[2. Further understand the importance of protocols 2](#_Toc287275914)

[3. Master sliding window mechanism 2](#_Toc287275915)

[4. Study a concrete second-layer protocol, the High-level Data Link Control (HDLC). 2](#_Toc287275916)

[Assumptions 2](#_Toc287275917)

[Discussion 2](#_Toc287275918)

[Communication Flowchart 2](#_Toc287275919)

[Client 2](#_Toc287275920)

[Server 2](#_Toc287275921)

[Frame 2](#_Toc287275922)

[Flag Field: 2](#_Toc287275923)

[Address Field: 2](#_Toc287275924)

[Control Field: 2](#_Toc287275925)

[Information Field: 2](#_Toc287275926)

[Conclusion 2](#_Toc287275927)

# Introduction

## Objectives:

### Get familiar with medium access control.

### Further understand the importance of protocols

### Master sliding window mechanism

### Study a concrete second-layer protocol, the High-level Data Link Control (HDLC).

In this lab we successfully completed the four objectives by developing a server/client Java program that incorporated medium access control and HDLC. We mastered the sliding window mechanism by incorporating it into our application to transfer frames from client to client, using the server as the intermediate. Finally after completion of the system we understood the importance of protocols when dealing with communication between two end points.

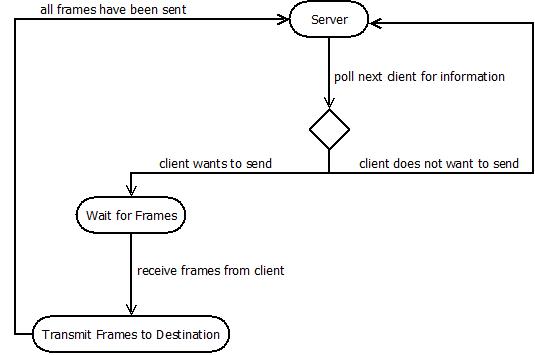
## Assumptions:

1. All communication strictly between the Primary Station and a single Secondary Station (such as polling and other cues) do not require the use of frames and can simply be sent as a string.
2. We are only dealing with 7-bit ASCII and not extended ASCII.
3. Bit stuffing is not implemented.
4. Extensive Error control is not necessary.
5. The Primary Station does not require an address
6. The Primary Station assigns all Secondary Stations their address
7. The users of all Secondary Stations are aware of the addresses of all other Secondary Stations
8. Validating the FCS of a received frame is not required.
9. Since the Primary Station must poll the Secondary Stations before they can send, only one Secondary Station will be transmitting at a time
10. When a Secondary Station is polled, it sends **ALL** the information in its buffer, not just a single frame
11. ACK Frames can be sent with out being polled
12. The words “Server” and “Primary Station” can be used interchangeably
13. The words “Client” and “Secondary Station” can be used interchangeably

Can you put some assumptions in here?

# Discussion

## Communication Flowchart:



After all connections have been established the server begins iterating through its list of connected clients and polls the first client in the array for information by sending a message to the client asking if it wants to send any frames. If the client has a message to send it replies to the server that it wishes to send and the server waits for frames[[1]](#footnote-1). The HDLC sliding window protocol is then used between the source client and the destination client while the server acts as an intermediate that passes the frames along. After the message and acknowledgments have been exchanged, the server and polls the next client in the list. Once the server has reached the end of the array, it goes back to the beginning of the array and continues to poll, this results in an infinite loop until server termination.

## Client

**The client contains three threads in addition to the main thread, each of which is implemented as a nested class that extends the Thread class:**

### Main:

### SocketIOManager:

The SocketIOManager acts as the main point of contact between the client and the server. Actually, despite its name, it only manages the **input** coming from the server to the client. The output stream from the client to the server is simply located in the SocketClient class as static variable so all nested classes can use it.

The reason for this abstraction is so that the input can then be redirected to one of the remaining two threads based on its nature. To achieve this, the SocketIOManager reads the input stream and the writes it to one of two output streams (one for the ClientMessageListener and one for the ServerMessageListener). Each of these output streams is then piped to an input stream, which is then read by the respected threads.

If the line read from the socket’s input stream is identified as a frame (by checking if it starts with a *flag*) then it is written to the output stream for the ClientMessageListener. Otherwise, it is written to the output stream of the ServerMessageListener.

All of this is required so that ACK frames can be received concurrently with the transmission of information frames. For example, if the information the client wants to send requires 18 frames and the window size is 7, clearly frames need to be acknowledged before all 18 frames can be sent. With this implementation, the client can begin sending the 18 frames and also receive ACK frames at the same time.

### ServerMessageListener:

The ServerMessageListener thread listens to the cTosDIS input stream for messages that the SocketIOManager flags as a message directly from the server. Currently, the only messages being sent directly from the user a requests or “poll” for the client to send data. The server sending the string “request-for-you-to-send” denotes a “poll”.

When the ServerMessageListener receives a “poll”, it either responds with the command “want-to-send” if the buff is not empty or “don’t-want-to-send” if buff is empty. If the client does want to send, the ServerMessageListener then begins constructing frames (one frame per 64 bytes of information) as per the guidelines outlined and sends them to the server following the sliding window protocol. Details on the implementation of both these concepts can be found in their respected sections.

### ClientMessageListener:

The ClientMessageListener thread listens to the cTocDIS input stream for messages that the SocketIOStream flags as a message originating from another client (though sent from the server). These messages are received in the form of Frames and are either new messages being sent to the client to be displayed or acknowledgments (ACK Frame) confirming that a message the client sent was received.

If the frame received is an ACK Frame, then the counter of un-acknowledged frames for that client is decremented and a confirmation message is printed to the user.

If the frame received is a message, then it is decoded and displayed to the user and an ACK frame is sent to the client at the frames source address.

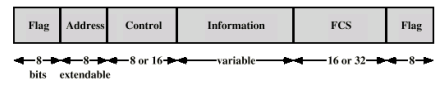
### Console:

A while loop in the main() function is also constantly checking for user input from the console, if the user inputs a message it will add that message to a string buffer. Once the client is polled from the server the buffer is checked, if it is empty the client responds with a “do not want to send” message. If the buffer contains a message, then the client responds with a “want to send” message and then encodes the message to frames and transmits them.

## Server

The server constantly listens and accepts connections from new clients. After connection has been accepted the client is added to an array list and a thread is run to listen for incoming messages. Once a message has been acquired, the server checks if it contains a flag, if yes, then the message that has been received is a frame and the server will extract the destination address and pass the frame along to the destination client. If the message does not contain a flag, then it is a command message from a client1. The command message is processed and the proper methods are executed. A log of the communication is printed to the console.

## Frame



A frame class is created to organize all the fields and provided encoding/decoding functionality for the user inputted messages.

**The frame contains the following fields:**

### Flag Field:

An 8 bit strings containing the value: 01111110

### Address Field:

The address field is extended to 16 bits to support the destination address and the source address. The first 8 bits represent the destination address, while the last 8 represent the source address.

### Control Field:

The control field is 8 bits and is fully implemented with I-Frames.

### Information Field:

The information field is a maximum of 64 bytes and encoded in binary.[[2]](#footnote-2)

### FCS Field:

A function in the frame class is called to create a 16-bit FCS which is append to the frame.

## Frame Functions

**Several functions are implemented in the frame class in order to encode and extract information:**

### getPorFbitFromFrame:

Returns the P/F bit from the frame.

### getSourceAddressBinaryFromFrame:

Returns the source address from the frame.

### getDestinationAddressBinaryFromFrame:

Returns the destination address from frame.

### getFrameType:

Checks the first two bits of the control field and returns the appropriate frame type.

### getCommandNameFromFrame:

First checks the frame type and then returns

### getDecodedInfo:

Returns the decoded the information field from binary to a string message.

### getNR:

Returns the N(R) number of the frame.

### getNS:

Returns the N(S) number of the frame.

# Sliding Window

# Conclusion

In conclusion we effectively integrated an HDLC protocol with sliding window into a server/client communication network using the Java IDE.

1. Assumption 1 [↑](#footnote-ref-1)
2. See assumption 2. [↑](#footnote-ref-2)