Network Working Group
Request for Comments: 916

G. Finn
ISI

October 1984

RELIABLE ASYNCHRONOUS TRANSFER PROTOCOL (RATP)

Status of This Memo

This RFC suggests a proposed protocol for the ARPA-Internet community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited.

This paper proposes and specifies a protocol which allows two programs to reliably communicate over a communication link. It ensures that the data entering one end of the link if received arrives at the other end intact and unaltered. The protocol, named RATP, is designed to operate over a full duplex point-to-point connection. It contains some features which tailor it to the RS-232 links now in common use.

Introduction

We are witnessing today an explosive growth in the small or personal computer market. Such inexpensive computers are not normally connected to a computer network. They are most likely stand-alone devices. But virtually all of them have an RS-232 interface. They also usually have a modem. This allows them to communicate over the telephone with any other similarly equipped computer.

The telephone system is a pervasive network, but one of the characteristics of the telephone system is the unpredictable quality of the circuit. The standard telephone circuit is designed for voice communication and not data communication. Voice communication tolerates a much higher degree of 'noise' than does a data circuit, so a voice circuit is tolerant of a much higher level of noise than is a data circuit. Thus it is not uncommon for a byte of data transferred over a telephone circuit to have noise inserted. For the same reason it is also not uncommon to have spurious data bytes added to the data stream.

The need for a method of reliably transferring data over an RS-232 point-to-point link has become severe. As the number of powerful personal computers grows, the need for them to communicate with one another grows as well. The new markets and new services that these computers will eventually allow their users to access will rely heavily upon the telephone system. Services like electronic mail, electronic banking, ordering merchandise from home with a personal computer, etc. As the information revolution proceeds data itself will become a commodity. All require accuracy of the data sent or received.

1. Philosopy of Design

Many tradeoffs were made in designing this protocol. Decisions were made by above all ensuring reliability and then by favoring simplicity of implementation. It is hoped that this protocol is simple enough to be implemented not only by small computers but also by stand alone devices incorporating microcomputers which accept commands over RS-232 lines. Sophisticated but unnecessary features such as dynamic window management [TCP 81] were left out for simplicity's sake. Having several packets outstanding at a time was eliminated for the same reason, and data queued to send when a connection is closed remotely is discarded. This eliminates two states from the protocol implementation.

The reader may ask why define this protocol at all, there are after all already RS-232 transport protocols in use. This is true but some lack one or more features vitally important or are too complex. See Appendix II for a brief survey.

- A protocol which can only transfer data in one direction is unable to use a single RS-232 link for a full-duplex connection. As such it cannot act as a bridge between most computer networks. Also it is not capable of supporting any applications requiring the two-way exchange of data. In particular it is not a platform suitable for the creation of most higher level applications. Unidirectional flow of data is sufficient for a weak implementation of file transfer but insufficient for remote terminal service, transaction oriented processing, etc.

- Some of the existing RS-232 transport protocols allow the use of only fixed size packets or do not allow the receiver to place a limit on the sender's packets. Where that block size is too large for the receiving end concentrator, that concentrator is likely to immediately invoke flow control. This results in many dropped and damaged packets. The receiver must be able to inform the sender at connection initiation what is the maximum packet size it is prepared to receive.

- Some protocols have a number of features which may or may not be implemented at each site. Examples are, several checksumming algorithms, differing data transmission restrictions, sometimes 8-bit data, sometimes restricted ASCII subsets, etc. The resulting requirement that all sites implement all the various features is rarely met.

Finally, the size of this document may be imposing. The document attempts to fully specify the behavior of the protocol. A careful

exposition of the protocol's behavior under all circumstances is necessary to answer any questions an implementor might have, to make it possible to verify the protocol, etc. This size of this specification should not be taken as an indication of the difficulty of implementing it.

1.1. The Host Environment

This protocol is designed to operate on any point-to-point communication link capable of transmitting and receiving data. It is not necessary that the link be asynchronous. Because neither end of a connection has control over when the other decides to transmit, the link should be full duplex. It is expected that in the vast majority of circumstances an asynchronous full-duplex RS-232 link will be used.

In practice this protocol could reside anywhere from the RS-232 driver software on a microcomputer in a concentrator all the way to the user software level. Ideally it properly resides inside the host operating system or concentrator. It should be an option associated with communication link which is selectable by the user program. If reliable data transmission were of great importance then the software would choose the option. Once the option were chosen the initial connection handshaking would begin.

There are many cases where this protocol will not reside in a host operating system (initially this will always be so). In addition there are many pieces of stand-alone equipment which accept commands over an RS-232 link. A plotter is such an example. To have a several hour plot ruined by noise on an unreliable data line is an all too often occurrence. The sending and receiving sides of the protocol should be as simple as possible allowing applications software and stand alone devices to utilize the protocol with little penalty of time or space.

1.2. Relation to Other Protocols

The "layering" concept has become the accepted way of designing communications protocols. Because this protocol will operate in a point-to-point environment it comprises both the datagram and reliable connection layers. No multi-network capability is implied. Where a link using this protocol bridges differing networks it is expected that other protocols like TCP will have their packets fragmented and encapsulated inside the packets of this protocol.

2. Packet Specification

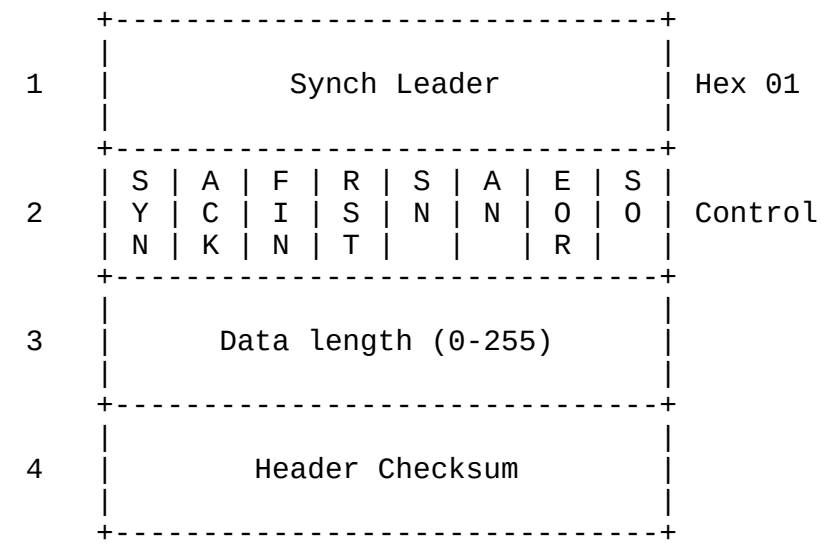
RATP transmits data over a full-duplex communication link. Data may be transmitted in both directions over the link. A stream of data is communicated by being broken up into 8-bit pieces called octets. These octets are serially accumulated to form a packet. The packet is the unit of data communicated over the link. The protocol virtually guarantees that the data transmitted at one end, if received, arrives unaltered and intact at the other end.

Within an octet all eight bits contain data. All eight bits must be preserved by the link interface and associated device driver. In many operating systems this is ensured by placing the connection into RAW or BINARY data mode. During normal operation packets are transmitted and acknowledged one at a time over the link in each direction. Each packet is composed of a HEADER followed by a DATA portion. The DATA portion may be empty.

> NOTE: There are some older operating systems and devices which do not permit 8-bit communication over an RS-232 link. Most of these allow restricted 7-bit communication. RATP can automatically detect this situation during connection initiation and utilizes a special packing strategy when full 8-bit communication is not possible. This is entirely transparent to any client software. See Appendix I for a discussion of this case.

2.1. Header Format

Byte No.



Header Portion of a Packet

2.1.1. Synch Leader

RS-232 provides a self-clocking communications medium. The wires over which data flows are often placed in 'noisy' environments where the noise can appear as added unwanted data. For this reason the beginning of a packet is denoted by a one octet SYNCH pattern. This allows the receiver to discard noise which appears on the connection prior to the reception of a packet. The SYNCH pattern is defined to be the one octet hex 01, the ASCII Start Of Header character <SOH>.

The SYNCH pattern should ideally be unlikely to occur as the result of noise. Differing modems, etc. have differing responses to noise so this is hard to achieve. The pattern chosen is thought to be a good compromise since many modems manifest noise by setting the high order bits. Situations will occur in which receiver is scanning for the beginning of a packet and a spurious SYNCH pattern is seen. To detect situations of this type a header checksum is provided (see below).

2.1.2. Control Bits

The first octet following the SYNCH pattern contains a 5-bit field of control flags and two 1-bit sequence number fields. The last bit is reserved and must be zero.

2.1.2.1. SYN - Synchronize Flag

Synchronize the connection. No data may be sent in a packet which has the SYN flag set.

2.1.2.2. ACK - Acknowledge Flag

Acknowledge number is significant. Data may accompany a packet which has this flag set as long as neither of SYN, RST, nor FIN are also set. Once a connection has been established this is always set.

2.1.2.3. RST - Reset Flag

Reset the connection. This is a method by which one end of a connection can reset the other when an anomalous condition is detected. No data may be sent in a packet which has the RST flag set.

2.1.2.4. FIN - Finishing Flag

This indicates that no more data will be sent to the other end of the connection. It also indicates that no more data will be accepted. No data may be sent in a packet which has the FIN flag set.

2.1.2.5. SN - Sequence Number

The Sequence Number associated with this packet.

2.1.2.6. AN - Acknowledge Number

If the ACK control flag is set this is the next Sequence Number the sender of the packet is expecting to receive.

2.1.2.7. EOR - End of Record

This bit is provided as an aid for higher level protocols which may need to fragment their packets. The Internet protocol for example often uses packets as large as 576 octets. A packet of such size would require fragmentation

when transported using this protocol. The EOR bit if set provides information to the higher level that a record is terminated in this packet. It is for information only and is the responsibility of the higher level to set/clear it when building packets to send. The interface to the protocol must provide a method of reading/setting/clearing this bit.

2.1.2.8. SO - Single Octet

One application thought to be of special importance is single character transmission --- a user communicates from the keyboard of a personal computer to another computer over an unreliable link. Since rapid interactive response is desirable it is expected that many of the characters typed will be transmitted individually. To minimize the overhead of this special case the SO control flag is provided.

The SO flag has no meaning if either the SYN, RST, or FIN flags are set. Assume none of those flags are set, then if the SO flag is set it indicates that a single octet of data is contained in this packet. Since the amount of data is known to be one octet the LENGTH field is superfluous and itself contains the data octet. The data portion of the packet is not transmitted.

The SO flag removes the need to transmit the data portion of the packet in this special case. Without the SO flag seven octets would be required of the packet, with it only four are needed and so transmission efficiency is improved by 40 percent. The header checksum protects the single octet of data.

2.1.3. Length

The second octet following the SYNCH pattern holds length information. If the SYN bit is present this contains the maximum number of data octets the receiver is allowed to transmit in any single packet to the sender. This quantity is called the MDL. A sender may indicate his unwillingness to accept any data octets by specifying an MDL of zero. In this case presumably all the data would be moving from the sender to the receiver. Obviously if data is to be transmitted both sides of a connection cannot have an MDL of zero.

If neither the SYN, RST, nor FIN flags are set this is an 8-bit field called LENGTH. In this case if the SO flag bit is set

then LENGTH contains a single octet of data. Otherwise it contains the count of data octets in this packet. From zero (0) to MDL octets of data may appear in a single packet. MDL is limited to a maximum of 255.

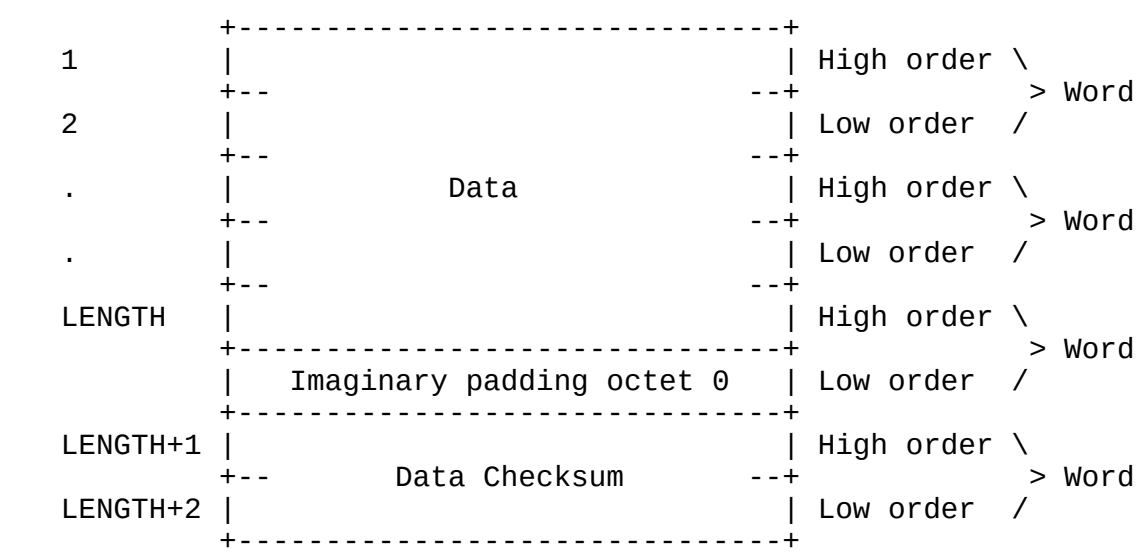
2.1.4. Header Checksum

The header checksum algorithm is the 8-bit equivalent of the 16-bit data checksum detailed below. It is built and processed in an similar manner but is eight bits wide instead of sixteen. When sending the header checksum octet is initially cleared. An 8-bit sum of the control, length, and header checksum octets is formed employing end-around carry. That sum is then complemented and stored in the header checksum octet. Upon receipt the 8-bit end-around carry sum is formed of the same three octets. If the sum is octal 377 the header is presumed to be valid. In all other cases the header is assumed to be invalid.

The reasons for providing this separate protection to the header are discussed in the chapter dealing with error handling. The header checksum covers the control and data length octets. It does not include the SYNCH pattern.

2.2. Data Format

The data portion of a packet immediately follows the header if the SO flag is not set and LENGTH > 0. It consists of LENGTH data octets immediately followed by two data checksum octets. If present the data portion contains LENGTH+2 octets.



Data Portion of a Packet

2.2.1. Data Checksum

The last two octets of the data portion of a packet are a data checksum. A 16-bit checksum is used by this protocol to detect incorrectly transmitted data. This has shown itself to be a reliable method for detecting most categories of bit drop out and bit insertion. While it does not guarantee the detection of all such errors the probability of such an error going undetected is on the order of 2**(-16).

The checksum octets follow the data to enable the sender of a packet to compute the checksum while transmitting a packet and the receiver to compute the checksum while receiving the packet. Thus neither must store the packet and then process the data for checksumming in a separate pass.

Order of Transmission

The order in which the 8-bit octets are assembled into 16-bit words, which is the low order octet and which is the high, must be rigidly specified for the purpose of computing 16-bit checksums. We specify the big endian ordering in the diagram above [Cohen 81].

Checksum Algorithm

The checksum algorithm chosen is similar to that used by IP/TCP protocols [IP 81] [TCP 81]. This algorithm has shown itself to be both reliable and relatively easy to compute. The interested reader may refer to [TCP Checksum 78] for a more thorough discussion of its properties.

The checksum algorithm is:

SENDER

The unsigned sum of the 16-bit words of the data portion of the packet is formed. Any overflow is added into the lowest order bit. This sum does not include the header portion of the packet. For the purpose of building a packet for transmission the two octet checksum field is zero. The sum formed is then bit complemented and inserted into the checksum field before transmission.

If the total number of data octets is odd then the last octet is padded to the right (low order) with zeros to form a 16-bit word for checksum purposes. This pad octet is not transmitted as part of the packet.

RECEIVER

The sum is computed as above but including the values received in the checksum field. If the 16-bit sum is octal 177777 then the data is presumed to be valid. In all other cases the data is presumed to be invalid.

This unsigned 16-bit sum adds 16-bit quantities with any overflow bit added into the lowest order bit of the sum. This is called 'end around carry'. End around carry addition provides several properties: 1) It provides full commutivity of addition (summing in any order is equivalent), and 2) If you apply a given rotation to each quantity before addition and when the final total is formed apply the inverse rotation, then the result will be equivalent to any other rotation chosen. The latter property gives little endian machines like a PDP-11 the go ahead to pick up 16-bit quantities and add them in byte swapped order.

The PDP-11 code to calculate the checksum is:

```
        CLR R0          ; R0 will get the checksum
                        ; R2 contains LENGTH count
LOOP:   ADD (R1)+,R0    ; Add the next 16-bit byte
        ADC R0          ; Make any carry be end around
        SOB R2,LOOP     ; Loop over entire packet
        COM R0          ; Bit complement result
```

2.3. Sequence Numbers

Sequence numbers work with acknowledge numbers to inform the sender that his last data packet was received, and to inform the receiver of the sequence number of the next data packet it expects to see. When the ACK flag is set in a packet the AN field contains the sequence number of the next data packet it expects from the sender. The sender looks at the AN field and by implication knows that the packet he just sent should have had a sequence number of:

<AN received-1 modulo 2>

If it did have that number that packet is considered to have been acknowledged.

Similarly, the receiver expects the next data packet it sees to have an SN field value equal to the AN field of the last acknowledge message it sent. If this is not the case then the receiver assumes that it is receiving a duplicate of a data packet it earlier acknowledged. This implies that the packet containing the acknowledgment did not arrive and therefor the packet that contained the acknowledgment should be retransmitted. The duplicate data packet is discarded.

The only packets which require acknowledgment are packets containing status flags (SYN, RST, FIN, or SO) or data. A packet which contains only an acknowledgment, i.e. <AN=n><CTL=ACK>, does not require a response (it contains no status flags or data).

Both the AN and SN fields are a single bit wide. Since at most one packet is in the process of being sent/acknowledged in a particular direction at any one time a single bit is sufficient to provide a method of duplicate packet detection and removal of a packet from the retransmission queue. The arithmetic to advance these numbers is modulo 2. Thus when a data packet has been acknowledged the sender's next sequence number will be the current one, plus one modulo 2:

<SN = SN + 1 modulo 2>

The individual acknowledgment of each packet containing data can mislead one into thinking that side A of a connection cannot send data to side B until it receives a packet from B. That only then can it acknowledge B's packet and place in the acknowledging packet some data of its own. This is not the case.

As long as its last packet sent requiring a response has been acknowledged each side of a connection is free to send a data packet whenever it wishes. Naturally, if one side is sending a data packet and it also must acknowledge receipt of a data packet from the other side, it is most efficient to combine both functions in a single packet.

2.4. Maximum Packet Size

The maximum packet size is:

SYNCH + HEADER + Data Checksum + 255 = 261 octets

There is therefor no need to allocate more than that amount of storage for any received packets.

3. The Opening and Closing of a Connection

3.1. Opening a Connection

A "three-way handshake" is the procedure used to establish a connection. It is normally initiated by one end of the connection and responded to by the other. It will still work if both sides simultaneously initiate the procedure. Experience has shown that this strategy of opening a connection reduces the probability of false connections to an acceptably low level.

The simplest form of the three-way handshake is illustrated in the diagram below. The time order is line by line from top to bottom with certain lines numbered for reference. User events are placed in brackets as in [OPEN]. An arrow (-->) represents the direction of flow of a packet and an ellipsis (...) indicates a packet in transit. Side A and side B are the two ends of the connection. An "XXX" indicates a packet which is lost or rejected. The contents of the packet are shown on the center of each line. The state of both connections is that caused by the departure or arrival of the packet represented on the line. The contents of the data portion of a packet are left out for clarity.

```
Side A                                                  Side B

1. CLOSED                                               LISTEN

2. [OPEN request]
    SYN-SENT ->   <SN=0><CTL=SYN><MDL=n>     ...

3.                                              -->   SYN-RECEIVED
        ... <SN=0><AN=1><CTL=SYN,ACK><MDL=m> <--

4. ESTABLISHED <--
        -->    <SN=1><AN=1><CTL=ACK><DATA>   ...

5.                                              -->   ESTABLISHED
```

In line 2 above the user at side A has requested that a connection be opened. Side A then attempts to open a connection by sending a SYN packet to side B which is in the LISTEN state. It specifies its initial sequence number, here zero. It places in the LENGTH field of the header the largest number of data octets it can consume in any one packet (MDL). The MDL is normally positive. The action of sending this packet places A in the SYN-SENT state.

In line 3 side B has just received the SYN packet from A. This

places B in the SYN-RECEIVED state. B now sends a SYN packet to A which acknowledges the SYN it just received from A. Note that the AN field indicates B is now expecting to hear SN=1, thus acknowledging the SYN packet from A which used SN=0. B also specifies in the LENGTH field the largest number of data octets it is prepared to consume.

Side A receives the SYN packet from B which acknowledges A's original SYN packet in line 4. This places A in the ESTABLISHED state. Side A can now be confident that B expects to receive more packets from A.

A is now free to send B the first DATA packet. In line 5 upon receipt of this packet side B is placed into the ESTABLISHED state. DATA cannot be sent until the sender is in the ESTABLISHED state. This is because the LENGTH field is used to specify the MDL when opening the connection.

3.2. Recovering from a Simultaneous Active OPEN

It is of course possible that both ends of a connection may choose to perform an active OPEN simultaneously. In this case neither end of the connection is in the LISTEN state, both send SYN packets. A reliable bidirectional protocol must recover from this situation. It should recover in such a manner that the connection is successfully initiated.

```
      Side A                                                Side B

      1. CLOSED                                             CLOSED

      2. [OPEN request]
         SYN-SENT -->  <SN=0><CTL=SYN><MDL=n>       ...

      3.     ...                                          [OPEN request]
                    <SN=0><CTL=SYN><MDL=m>       <--  SYN-SENT

      4.                                            -->  SYN-RECEIVED
             ...  <SN=0><AN=1><CTL=SYN,ACK><MDL=m>  <--

      5. (packet finally arrives)
         SYN-RECEIVED  <--  <SN=0><CTL=SYN><MDL=m>

             -->  <SN=0><AN=1><CTL=SYN,ACK><MDL=n>  -->  ESTABLISHED
              ...       <SN=1><AN=1><CTL=ACK>       <--

      6. (packet finally arrives)
         ESTABLISHED <-- <SN=0><AN=1><CTL=SYN,ACK><MDL=m>
                     -->   <SN=1><AN=1><CTL=ACK>    ...
```

During simultaneous connection both sides of the connection cycle from the CLOSED state through SYN-SENT to SYN-RECEIVED, and finally to ESTABLISHED.

3.3. Detecting a Half-Open Connection

Any computer may crash after a connection has been established. After recovering from the crash it may attempt to open a new connection. The other end must be able to detect this condition and treat it as an error.

```
      Side A                                                Side

      1. ESTABLISHED                                        ESTABLISHED

                --->   <SN=0><AN=1><CTL=ACK><DATA>  ...
                                                    -->
      (crashes)

      2.        XXX    <SN=1><AN=1><CTL=ACK><DATA>  <--

      3. (attempts to open new connection )
                -->     <SN=0><CTL=SYN><MDL=m>      -->
                ...  <SN=0><AN=1><CTL=RST,ACK>      <--   (abort)
                                                          CLOSED

      4.        <--
      (connection refused)
         CLOSED
```

3.4. Closing a Connection

Either side may choose to close an established connection. This is accomplished by sending a packet with the FIN control bit set. No data may appear in a FIN packet. The other end of the connection responds by shutting down its end of the connection and sending a FIN, ACK in response.

```
      Side A                                                Side B

      1. ESTABLISHED                                        ESTABLISHED

      2. [CLOSE request from user]
         FIN-WAIT  -->     <SN=0><AN=1><CTL=FIN>    ...

      3.                                             -->  LAST-ACK
                   ...   <SN=1><AN=1><CTL=FIN,ACK>  <--

      4. TIME-WAIT <--
                   -->     <SN=1><AN=0><CTL=ACK>    ...

      5.                                             -->  CLOSED

      6. (after 2*SRTT time passes)
         CLOSED
```

In line 2 the user on side A of the fully opened connection has decided to close it down by issuing a CLOSE call. No more data

will be accepted for sending. If data remains unsent a message "Warning: Unsent data remains." is communicated to the user. No more data will be received. A packet containing a FIN but no data is constructed and sent. Side A goes into the FIN-WAIT state.

Side B sees the FIN sent and immediately builds a FIN, ACK packet in response. It then goes into the LAST-ACK state. The FIN, ACK packet is received by side A and an answering ACK is immediately sent. Side A then goes to the TIME-WAIT state. In line 5 side B receives the final acknowledgment of its FIN, ACK packet and goes to the CLOSED state. In line 6 after waiting to be sure its last acknowledgment was received side A goes to the CLOSED state (SRTT is the Smoothed Round Trip Time and is defined in section 6.3.1).

4. Packet Reception

The act of receiving a packet is relatively straightforward. There are a few points which deserve some discussion. This chapter will discuss packet reception stage by stage in time order.

Synch Detection

The first stage in the reception of a packet is the discovery of a SYNCH pattern. Octets are read continuously and discarded until the SYNCH pattern is seen. Once SYNCH has been observed proceed to the Header Reception stage.

Header Reception

The remainder of the header is three octets in length. No further processing can continue until the complete header has been read. Once read the header checksum test is performed. If this test fails it is assumed that the current SYNCH pattern was the result of a data error. Since the correct SYNCH may appear immediately after the current one, go back to the Synch Detection stage but treat the three octets of the header following the bad SYNCH as new input.

If the header checksum test succeeds then proceed to the Data Reception stage.

Data Reception

A determination of the remaining length of the packet is made. If either of the SYN, RST, SO, or FIN flags are set then legally the entire packet has already been read and it is considered to have 'arrived'. No data portion of a packet is present when one of those flags is set. Otherwise the LENGTH field specifies the remaining amount of data to read. In this case if the LENGTH field is zero then the packet contains no data portion and it is considered to have arrived.

We now assume that a data portion is present and LENGTH was non-zero. Counting the data checksum LENGTH+2 octets must now be read. Once read the data checksum test is performed. If this test fails the entire packet is discarded, return to the Synch Detection stage. If the test succeeds then the packet is considered to have arrived.

Once arrived the packet is released to the upper level protocol software. In a multiprocess implementation packet reception would now begin again at the Synch Detection stage.

5. Functional Specification

A convenient model for the discussion and implementation of protocols is that of a state machine. A connection can be thought of as passing through a variety of states, with possible error conditions, from its inception until it is closed. In such a model each state represents a known point in the history of a connection. The connection passes from state to state in response to events. These events are caused by user calls to the protocol interface (a request to open or close a connection, data to send, etc.), incoming packets, and timeouts.

Information about a connection must be maintained at both ends of that connection. Following the terminology of [TCP 81] the information necessary to the successful operation of a connection is called the Transmission Control Block or TCB. The user requests to the protocol interface are OPEN, SEND, RECEIVE, ABORT, STATUS, and CLOSE.

This chapter is broken up into three parts. First a brief description of each protocol state will be presented. Following this is a slightly more detailed look at the allowed transitions which occur between states. Finally a detailed discussion of the behavior of each state is given.

5.1. Protocol States

The states used to describe this protocol are:

LISTEN

This state represents waiting for a connection from the other end of the link.

SYN-SENT

This represents waiting for a matching connection request after having sent a connection request.

SYN-RECEIVED

This represents waiting for a confirming connection request acknowledgment after having both received and sent a connection request.

ESTABLISHED

This state represents a connection fully opened at both ends. This is the normal state for data transfer.

FIN-WAIT

In this state one is waiting for a connection termination request from the other end of the connection and an acknowledgment of a termination request previously sent.

LAST-ACK

This end of the connection has seen and acknowledged a termination request from the other end. This end has responded with a termination request of its own and is now expecting an acknowledgment of that request.

CLOSING

This represents waiting for an acknowledgment of a connection termination request.

TIME-WAIT

This represents waiting for enough time to pass to be sure that the other end of the connection received the acknowledgment of its termination request.

CLOSED

A fictional state which represents a completely terminated connection. If either end of a connection is in this state it will neither send nor receive data or control packets.

5.2. State Transitions

This section describes events which cause the protocol to depart from its current state. A brief mention of each state is accompanied by a list of departure events and to which state the protocol goes as a result of those events. Departures due to the presence of a RST flag are not shown.

5.2.1. LISTEN

This is a request to listen for any connection from the other end of the link. In this state, no packets are sent. The connection may be thought of as half-open. A STATUS request will return to the caller this information.

Arrived at from the CLOSED state in response to a passive OPEN. In a passive OPEN no packets are sent, the interface is waiting for the initiation of a connection from the other end of the link. Also this state can be reached in certain cases in response to an RST connection reset request.

Departures

- A CLOSE request is made by the user. Delete the half-open TCB and go to the CLOSED state.

- A packet arrives with the SYN flag set. Retrieve the sender's MDL he placed into the LENGTH field. Set AN to be received SN+1 modulo 2. Build a response packet with SYN, ACK set. Choose your MDL and place it into the LENGTH octet. Choose your initial SN, place in AN. Send this packet and go to the SYN-RECEIVED state.

5.2.2. SYN-SENT

Arrived at from the CLOSED state in response to a user's active OPEN request.

Departures

- A CLOSE request is made by the user. Delete the TCB and go to the CLOSED state.

- A packet arrives with the SYN flag set. Retrieve the sender's MDL he placed into the LENGTH field. Set AN to

be received SN+1 modulo 2. Build a response packet with ACK set, place in AN. Send this packet and go to the SYN-RECEIVED state.

- A packet arrives with the SYN, ACK flags set. Retrieve the sender's MDL he placed into the LENGTH field. Set AN to be received SN+1 modulo 2. Build a response packet with ACK set. Set SN to be SN+1 modulo 2, place SN and AN into the header. Remembering the other end's MDL, build data portion of packet. Send this packet and go to the ESTABLISHED state.

5.2.3. SYN-RECEIVED

Arrived at from the LISTEN and SYN-SENT states in response to an arriving SYN packet.

Departures

- A CLOSE request is made by the user. Create a packet with FIN set. Send it and go to the FIN-WAIT state.

- A packet arrives with the ACK flag set. This packet acknowledges a previous SYN packet. Go to the ESTABLISHED state. The TCB should now note the connection is fully opened.

- A packet arrives with the FIN flag set. The other end has decided to close the connection. Create a packet with FIN, ACK set. Send it and go to the LAST-ACK state.

5.2.4. ESTABLISHED

This state is the normal state for a connection. Data packets may be exchanged in both directions (MDL allowing). It is arrived at from the SYN-RECEIVED and SYN-SENT states in response to the completion of connection initiation.

Departures

- In response to a CLOSE request from the user. Set AN to be most recently received SN+1 modulo 2. Build a packet with FIN set. Set SN to be SN+1 modulo 2, place SN and AN into the header and send the packet. Go to the FIN-WAIT state.

- A packet containing a FIN is received. Set AN to be

received SN+1 modulo 2. Build a response packet with both FIN and ACK set. Set SN to be SN+1 modulo 2, place SN and AN into the header. No data portion is built. Send this packet and go to the LAST-ACK state.

5.2.5. FIN-WAIT

Arrived at from either the SYN-RECEIVED state or from the ESTABLISHED state. In both cases the user had requested a CLOSE of the connection and a packet with a FIN was sent.

Departures

- A FIN, ACK packet is received which acknowledges the FIN just sent. Go to the TIME-WAIT state.

- A FIN packet is received which indicates the other end of the connection has simultaneously decided to close. Set AN=received SN+1 modulo 2, and SN=SN+1 modulo 2. Send a response packet with the ACK set. Go to the CLOSING state.

5.2.6. LAST-ACK

Arrived at from the ESTABLISHED and SYN-RECEIVED states.

Departures

- An ACK is received for the last packet sent which was a FIN. Delete the TCB and go to the CLOSED state.

5.2.7. CLOSING

Arrived at from the FIN-WAIT state.

Departures

- An ACK is received for the last packet sent which was a FIN. Go to the TIME-WAIT state.

5.2.8. TIME-WAIT

Arrived at from the FIN-WAIT and CLOSING states.

Departures

- This states waits until 2*SRTT time has passed. It then deletes the TCB associated with the connection and goes to the CLOSED state.

5.2.9. CLOSED

This state can be arrived at for a number of reasons: 1) while in the LISTEN state the user requests a CLOSE, 2) while in the SYN-SENT state the user requests a CLOSE, 3) while in the TIME-WAIT state the 2*SRTT time period has elapsed, and 4) while in the LAST-ACK state an arriving packet has an ACK of the previously sent FIN packet.

In this state no data is read or sent over the link. To leave this state requires an outside request to open a new connection.

Departures

- User requests an active OPEN. Create a packet with SYN set. Choose your MDL and place it into the LENGTH octet. Choose your initial SN. AN is immaterial. Send this packet and go to the SYN-SENT state. The TCB for this connection is created. The connection may be thought of as half-open. A STATUS request will return to the caller this information.

- User requests a passive OPEN. The TCB for this connection is created. The connection may be thought of as half-open. A STATUS request will return to the caller this information. Go to the LISTEN state.

5.3. State Behavior

This section discusses in detail the behavior of each state in response to the arrival of a packet. In what follows a packet is not considered to have arrived until it has passed a number of tests (see the chapter entitled: Packet Reception).

The method chosen to describe state behavior is tabular. Each state is listed opposite a sequence of named procedures to execute whenever a packet has arrived.

STATE	BEHAVIOR
LISTEN	A
SYN-SENT	B
SYN-RECEIVED	C1 D1 E F1 H1
ESTABLISHED	C2 D2 E F2 H2 I1
FIN-WAIT	C2 D2 E F3 H3
LAST-ACK	C2 D3 E F3 H4
CLOSING	C2 D3 E F3 H5
TIME-WAIT	D3 E F3 H6
CLOSED	G

For example, in the ESTABLISHED state the arrival of a packet causes procedure C2 to be executed, then D2, then E, F2, H2, and finally I1. Any procedure may terminate the processing which occurs or cause a state change. Note that these procedures are executed in sequence, first C2, then D2, etc. The time ordering cannot be mixed.

The particular actions associated with each procedure are now described.

A ----------

This procedure details the behavior of the LISTEN state. First check the packet for the RST flag. If it is set then packet is discarded and ignored, return and continue the processing associated with this state.

We assume now that the RST flag was not set. Check the packet for the ACK flag. If it is set we have an illegal condition since no connection has yet been opened. Send a RST packet with the correct response SN value:

<SN=received AN><CTL=RST>

Return to the current state without any further processing.

We assume now that neither the RST nor the ACK flags were set. Check the packet for a SYN flag. If it is set then an attempt is being made to open a connection. Create a TCB for this connection. The sender has placed its MDL in the LENGTH field, also specified is the sender's initial SN value. Retrieve and place them into the TCB. Note that the presence of the SO flag is ignored since it has no meaning when either of the SYN, RST, or FIN flags are set.

Send a SYN packet which acknowledges the SYN received. Choose the initial SN value and the MDL for this end of the connection:

<SN=0><AN=received SN+1 modulo 2><CTL=SYN, ACK><LENGTH=MDL>

and go to the SYN-RECEIVED state without any further processing.

Any packet not satisfying the above tests is discarded and ignored. Return to the current state without any further processing.

B ----------

This procedure represents the behavior of the SYN-SENT state and is entered when this end of the connection decides to execute an active OPEN.

First, check the packet for the ACK flag. If the ACK flag is set then check to see if the AN value was as expected. If it was continue below. Otherwise the AN value was unexpected. If the RST flag was set then discard the packet and return to the current state without any further processing, else send a reset:

<SN=received AN><CTL=RST>

Discard the packet and return to the current state without any further processing.

At this point either the ACK flag was set and the AN value was as expected or ACK was not set. Second, check the RST flag. If the RST flag is set there are two cases:

1. If the ACK flag is set then discard the packet, flush the retransmission queue, inform the user "Error: Connection refused", delete the TCB, and go to the CLOSED state without any further processing.

2. If the ACK flag was not set then discard the packet and return to this state without any further processing.

At this point we assume the packet contained an ACK which was Ok, or there was no ACK, and there was no RST. Now check the packet for the SYN flag. If the ACK flag was set then our SYN has been acknowledged. Store MDL received in the TCB. At this point we are technically in the ESTABLISHED state. Send an acknowledgment packet and any initial data which is queued to send:

<SN=received AN><AN=received SN+1 modulo 2><CTL=ACK><DATA>

Go to the ESTABLISHED state without any further processing.

If the SYN flag was set but the ACK was not set then the other end of the connection has executed an active open also. Acknowledge the SYN, choose your MDL, and send:

<SN=0><AN=received SN+1 modulo 2><CTL=SYN, ACK><LENGTH=MDL>

Go to the SYN-RECEIVED state without any further processing.

Any packet not satisfying the above tests is discarded and ignored. Return to the current state without any further processing.

C1 ----------

Examine the received SN field value. If the SN value was expected then return and continue the processing associated with this state.

We now assume the SN value was not what was expected.

If either RST or FIN were set discard the packet and return to the current state without any further processing.

If neither RST nor FIN flags were set it is assumed that this packet is a duplicate of one already received. Send an ACK back:

<SN=received AN><AN=received SN+1 modulo 2><CTL=ACK>

Discard the duplicate packet and return to the current state without any further processing.

C2 ----------

Examine the received SN field value. If the SN value was expected then return and continue the processing associated with this state.

We now assume the SN value was not what was expected.

If either RST or FIN were set discard the packet and return to the current state without any further processing.

If SYN was set we assume that the other end crashed and has attempted to open a new connection. We respond by sending a legal reset:

<SN=received AN><AN=received SN+1 modulo 2><CTL=RST, ACK>

This will cause the other end, currently in the SYN-SENT state, to close. Flush the retransmission queue, inform the user "Error: Connection reset", discard the packet, delete the TCB, and go to the CLOSED state without any further processing.

If neither RST, FIN, nor SYN flags were set it is assumed that this packet is a duplicate of one already received. Send an ACK back:

<SN=received AN><AN=received SN+1 modulo 2><CTL=ACK>

Discard the duplicate packet and return to the current state without any further processing.

D1 ----------

The packet is examined for a RST flag. If RST is not set then return and continue the processing associated with this state.

RST is now assumed to have been set. If the connection was originally initiated from the LISTEN state (it was passively opened) then flush the retransmission queue, discard the packet, and go to the LISTEN state without any further processing.

If instead the connection was initiated actively (came from the SYN-SENT state) then flush the retransmission queue, inform the user "Error: Connection refused", discard the packet, delete the TCB, and go to the CLOSED state without any further processing.

D2 ----------

The packet is examined for a RST flag. If RST is not set then return and continue the processing associated with this state.

RST is now assumed to have been set. Any data remaining to be sent is flushed. The retransmission queue is flushed, the user is informed "Error: Connection reset.", discard the packet, delete the TCB, and go to the CLOSED state without any further processing.

D3 ----------

The packet is examined for a RST flag. If RST is not set then return and continue the processing associated with this state.

RST is now assumed to have been set. Discard the packet, delete the TCB, and go to the CLOSED state without any further processing.

E ----------

Check the presence of the SYN flag. If the SYN flag is not set then return and continue the processing associated with this state.

We now assume that the SYN flag was set. The presence of a SYN here is an error. Flush the retransmission queue, send a legal RST packet.

If the ACK flag was set then send:

<SN=received AN><CTL=RST>

If the ACK flag was not set then send:

<SN=0><CTL=RST>

The user should receive the message "Error: Connection reset.", then delete the TCB and go to the CLOSED state without any further processing.

F1 ----------

Check the presence of the ACK flag. If ACK is not set then discard the packet and return without any further processing.

We now assume that the ACK flag was set. If the AN field value was as expected then return and continue the processing associated with this state.

We now assume that the ACK flag was set and that the AN field value was unexpected. If the connection was originally initiated from the LISTEN state (it was passively opened) then flush the retransmission queue, discard the packet, and send a legal RST packet:

<SN=received AN><CTL=RST>

Then delete the TCB and go to the LISTEN state without any further processing.

Otherwise the connection was initiated actively (came from the SYN-SENT state) then inform the user "Error: Connection refused", flush the retransmission queue, discard the packet, and send a legal RST packet:

<SN=received AN><CTL=RST>

Then delete the TCB and go to the CLOSED state without any further processing.

F2 ----------

Check the presence of the ACK flag. If ACK is not set then discard the packet and return without any further processing.

We now assume that the ACK flag was set. If the AN field value was as expected then flush the retransmission queue and inform the user with an "Ok" if a buffer has been entirely acknowledged. Another packet containing data may now be sent. Return and continue the processing associated with this state.

We now assume that the ACK flag was set and that the AN field value was unexpected. This is assumed to indicate a duplicate acknowledgment. It is ignored, return and continue the processing associated with this state.

F3 ----------

Check the presence of the ACK flag. If ACK is not set then discard the packet and return without any further processing.

We now assume that the ACK flag was set. If the AN field value was as expected then continue the processing associated with this state.

We now assume that the ACK flag was set and that the AN field value was unexpected. This is ignored, return and continue with the processing associated with this state.

G ----------

This procedure represents the behavior of the CLOSED state of a connection. All incoming packets are discarded. If the packet had the RST flag set take no action. Otherwise it is necessary to build a RST packet. Since this end is closed the other end of the connection has incorrect data about the state of the connection and should be so informed.

If the ACK flag was set then send:

<SN=received AN><CTL=RST>

If the ACK flag was not set then send:

<SN=0><AN=received SN+1 modulo 2><CTL=RST, ACK>

After sending the reset packet return to the current state without any further processing.

H1 ----------

Our SYN has been acknowledged. At this point we are technically in the ESTABLISHED state. Send any initial data which is queued to send:

<SN=received AN><AN=received SN+1 modulo 2><CTL=ACK><DATA>

Go to the ESTABLISHED state and execute procedure I1 to process any data which might be in this packet.

Any packet not satisfying the above tests is discarded and ignored. Return to the current state without any further processing.

H2 ----------

Check the presence of the FIN flag. If FIN is not set then continue the processing associated with this state.

We now assume that the FIN flag was set. This means the other end has decided to close the connection. Flush the retransmission queue. If any data remains to be sent then inform the user "Warning: Data left unsent." The user must also be informed "Connection closing." An acknowledgment for the FIN must be sent which also indicates this end is closing:

<SN=received AN><AN=received SN + 1 modulo 2><CTL=FIN, ACK>

Go to the LAST-ACK state without any further processing.

H3 ----------

This state represents the final behavior of the FIN-WAIT state.

If the packet did not contain a FIN we assume this packet is a duplicate and that the other end of the connection has not seen the FIN packet we sent earlier. Rely upon retransmission of our earlier FIN packet to inform the other end of our desire to close. Discard the packet and return without any further processing.

At this point we have a packet which should contain a FIN. By the rules of this protocol an ACK of a FIN requires a FIN, ACK in response and no data. If the packet contains data we have detected an illegal condition. Send a reset:
<SN=received AN><AN=received SN+1 modulo 2><CTL=RST, ACK>

Discard the packet, flush the retransmission queue, inform the user "Error: Connection reset.", delete the TCB, and go to the CLOSED state without any further processing.

We now assume that the FIN flag was set and no data was contained in the packet. If the AN field value was expected then this packet acknowledges a previously sent FIN packet. The other end of the connection is then also assumed to be closing and expects an acknowledgment. Send an acknowledgment of the FIN:

<SN=received AN><AN=received SN+1 modulo 2><CTL=ACK>

Start the 2*SRTT timer associated with the TIME-WAIT state, discard the packet, and go to the TIME-WAIT state without any further processing.

Otherwise the AN field value was unexpected. This indicates a simultaneous closing by both sides of the connection. Send an acknowledgment of the FIN:

<SN=received AN><AN=received SN+1 modulo 2><CTL=ACK>

Discard the packet, and go to the CLOSING state without any further processing.

H4 ----------

This state represents the final behavior of the LAST-ACK state.

If the AN field value is expected then this ACK is in response to the FIN, ACK packet recently sent. This is the final acknowledging message indicating both side's agreement to close the connection. Discard the packet, flush all queues, delete the TCB, and go to the CLOSED state without any further processing.

Otherwise the AN field value was unexpected. Discard the packet and remain in the current state without any further processing.

H5 ----------

This state represents the final behavior of the CLOSING state.

If the AN field value was expected then this packet acknowledges the FIN packet recently sent. This is the final acknowledging message indicating both side's agreement to close the connection. Start the 2*SRTT timer associated with the TIME-WAIT state, discard the packet, and go to the TIME-WAIT state without any further processing.

Otherwise the AN field value was unexpected. Discard the packet and remain in the current state without any further processing.

H6 ----------

This state represents the behavior of the TIME-WAIT state. Check the presence of the ACK flag. If ACK is not set then discard the packet and return without any further processing.

Check the presence of the FIN flag. If FIN is not set then discard the packet and return without any further processing.

We now assume that the FIN flag was set. This situation indicates that the last acknowledgment of the FIN packet sent by the other end of the connection did not arrive. Resend the acknowledgment:

<SN=received AN><AN=received SN+1 modulo 2><CTL=ACK>

Restart the 2*SRTT timer, discard the packet, and remain in the current state without any further processing.

I1 ----------

This represents that stage of processing in the ESTABLISHED state in which all the flag bits have been processed and only data may remain. The packet is examined to see if it contains data. If not the packet is now discarded, return to the current state without any further processing.

We assume the packet contained data, that either the SO flag was set or LENGTH is positive. That data is placed into the user's receive buffers. As these become full the user should be informed "Receive buffer full." An acknowledgment is sent:

<SN=received AN><AN=received SN+1 modulo 2><CTL=ACK>

If data is queued to send then it is most efficient to 'piggyback' this acknowledgment on that data packet.

The packet is now discarded, return to the ESTABLISHED state without any further processing.

5.4. Timers

There are three timers associated with this protocol. Their purpose will now be briefly discussed as will the actions taken when a timer expires. The particular nature these timeouts take and the methods by which they are set is the responsibility of the protocol implementation.

5.4.1. User Timeout

For practical implementation reasons it is desirable to have a user controllable timeout associated with the successful opening of a connection, successful acknowledgment of data, and successful closing of a connection. Consider the situations in which a connection is so noisy that no data gets through, or a connection is physically cut. Without an overriding timeout these situations would result in unbounded retransmissions.

When this timeout expires the user is informed "Error: Connection aborted due to user timeout.", all queues are flushed, the TCB is deleted, and the CLOSED state is entered.

5.4.2. Retransmission Timeout

This timer ensures that any packet sent for which the SN is significant is acknowledged. When such a packet is sent it is placed in a retransmission queue and the retransmission timer is begun. If an acknowledgment has not arrived within the timer's period then the packet is retransmitted and the timer is restarted. If the acknowledgment does arrive in time then the timer is stopped and the packet is removed from the retransmission queue. The next packet with a significant SN may now be sent.

This timeout is expected to operate in conjunction with a counter which keeps track of the number of times a packet has been retransmitted. Normally an upper limit is set on retransmissions. If that limit is exceeded then the connection is aborted. This event is similar to the user timeout. The user is informed "Error: Connection aborted due to retransmission failure", all queues are flushed, the TCB is deleted, and the CLOSED state is entered.

5.4.3. TIME-WAIT Timeout

This timeout is used to catch any FIN packets which might be retransmitted from the other end of a connection in response to a dropped acknowledgment packet. The timeout period should be at least as long as 2*SRTT. After this timeout expires the other end of the connection is assumed to be closed, the TCB is deleted, and this end enters the CLOSED state also.

6. Data Error Handling

This chapter discusses in detail the types of data errors an established connection may encounter. These are distinct from protocol errors discussed above. In order of discussion these are:

- Framing Errors
- Missing SYNCH pattern
- Unacknowledged packets
- Bad packets
- Duplicate packets
- Outside flow control
- Packets that are too large
- Packets that are too small

6.1. Framing Errors

The RS-232 specification provides framing only for an individual octet. Link level protocols for computer networking normally provide framing for each packet. The SYNCH pattern provides a boundary for the beginning of a packet. No similar pattern was chosen to mark the end and completely frame the packet.

Any bit pattern can appear in the data portion of a packet. For any particular pattern to reliably mark the end of a packet that terminating pattern cannot be allowed to appear in the data. This is usually accomplished by the sender altering any occurrence of the terminating pattern in the data so that it is both no longer recognizable as that pattern and also restorable upon receipt. Both the sender and the receiver are required by this technique to examine all the data. In the absence of a protocol chip to perform this function, it is a source of some overhead.

6.1.1. Synthetic Framing

In the absence of framing, the end of the packet must be synthetically determined. The start of a packet is indicated by the SYNCH pattern. The expected end of a packet can now only be determined by examining the LENGTH octet of the header. It is important to know whether or not the LENGTH data can be

trusted. This is accomplished by employing a one octet header checksum to cover the first two octets following the SYNCH pattern. If the header passes the checksum test and neither the SYN, FIN, RST, nor SO flag bits were set then LENGTH is trusted and the number of octets expected beyond the header is LENGTH+2. (For those packets in which any of the above flag bits are set the packet length is fixed and includes only a header portion.)

If the header fails the checksum test we are in some difficulty. The length is incorrect so it may be too small or too large. To recover from this error do the following. Beginning immediately after the SYNCH pattern rescan looking for the next SYNCH pattern. Throw away all octets until a SYNCH is seen and then attempt to reinterpret it as a packet. The sender's retransmission timeout guarantees that a new copy of the packet will be transmitted. This ensures that in discarding the initial SYNCH pattern, the SYNCH pattern from the beginning of the retransmitted packet will eventually be seen.

6.1.2. Costs of Synthetic Framing

This framing strategy causes no overhead unless data errors occur in the packet. This is presumed to be a low probability occurrence. In addition it removes the overhead of both sender and receiver passing over the data to process any termination pattern which might appear in the data.

The worst case behavior would require a packet header to fail its checksum, a new SYNCH pattern to appear in the next few octets, that header failing its checksum, etc., until the SYNCH pattern of the retransmitted packet were finally seen. Consistently bad behavior of this type indicates an extremely noisy communications link.

6.2. Missing SYNCH Pattern

Any valid packet must begin with the SYNCH pattern. Any receiver must discard all input octets until the SYNCH pattern is seen. The data which immediately follows a SYNCH pattern is interpreted as a packet. The header checksum test is applied, then LENGTH+2 octets are read, the data checksum test is applied, etc.

6.3. Unacknowledged Packets

If an ACK for a packet is not obtained within the retransmission timeout interval that packet is retransmitted. Because significant variability in response can be expected from either end of a connection it is best to dynamically calculate the retransmission timeout interval. An example of such a calculation is provided below. The protocol will operate successfully, although not with as high an effective transmission rate, if a realistic upper bound time is used instead.

A realistic upper bound time depends upon the packet size and line speed. If the baud rate of the connection is 300 or above let B be the baud rate (for clarity assume it is the same in both directions), let L be the MDL of the receiver, let P be the packet processing time of the receiver. Then an Upper Bound for the Reception Time (UBRT) is:

UBRT = L/(B/10) seconds + P seconds

and a realistic upper bound time is 2*UBRT seconds.

6.3.1. Calculation of Retransmission Timeout Interval

For the purpose of detecting retransmission time out the protocol must have access to a clock which provides at least single second resolution. One technique for calculating the round trip time is:

Measure the elapsed time between sending a packet with a particular SN and receiving an ACK with an AN which covers that SN. The measured elapsed time is the Round Trip Time (RTT). Next a Smoothed Round Trip Time (SRTT) is calculated as:

SRTT = (ALPHA * SRTT) + ((1- ALPHA) * RTT)

and based upon this you compute the Retransmission Time Out (RTO) as:

RTO = min[UBOUND, max[LBOUND, (BETA * SRTT)]]

where UBOUND is an upper bound on the timeout (e.g., 1 minute), LBOUND is a lower bound on the timeout (e.g., 1 second), ALPHA is a smoothing factor (e.g., .8 to .9), and BETA is a delay variance factor (e.g., 1.3 to 2.0).

6.4. Bad Packets

A bad packet is received when it fails either the header or data checksum tests. When this happens the sender will retransmit the packet after the retransmission timeout interval.

6.5. Duplicate Packets

A duplicate packet is a packet which passes the checksum tests but for which the SN received is significant but not the expected value. This is normally caused when the sender did not get the ACK last sent by the receiver. This situation is diagrammed below.

```
Side A                                                  Side B

ESTABLISHED                                             ESTABLISHED

1.      --> <SN=1><AN=0><CTL=ACK><DATA>          ...
                                                 -->

2.      XXX <SN=0><AN=0><CTL=ACK><OTHER-DATA> <--

3. (after SRTT)
        --> <SN=1><AN=0><CTL=ACK><DATA>          ...

4.                                               -->
        ... <SN=0><AN=0><CTL=ACK><OTHER-DATA> <--

5.      <--
```

In line 2, B's packet was lost in transit, it may have failed its checksum tests when it reached A or its initial SYNCH pattern was smashed, etc.. In line 3 side A comes to the decision that its packet from line 1 was not received after SRTT time passes and retransmits that packet.

In line 4 side B receives the packet. It detects a duplicate because it already sent a packet acknowledging A's SN=1 (although that packet was lost). B now discards the duplicate and immediately retransmits its last packet to A. Side A finally receives the retransmitted packet in line 5.

6.6. Outside Flow Control

There are many large computer systems which make use of flow control to regulate their input side of an RS-232 link. Flow control based upon two special characters such as <Ctrl-S> (ASCII DC3) and <Ctrl-Q> (ASCII DC1) is almost universally in use today. So it becomes important for the protocol to be able to either:

(1) Recognize and obey the flow control of the host computer(s), or

(2) Ignore the flow control but still guarantee reliable data reception.

It is the latter approach which this protocol takes. This decision was made because the number of differing flow control characters in use would make it difficult to obey them all.

There is a particular type of flow control with which this protocol will not operate. The ENQUIRE, ACKNOWLEDGE method of flow control requires that the receiver of an inquiry respond with an acknowledge before any more data will be sent to it. This type of flow control also usually prohibits unrestricted 8-bit data transmission because the inquiry character is forbidden as a data byte.

For the other class of flow control methods a proof is required that data may still be reliably transmitted and received if flow control is ignored. For the purposes of this discussion assume <Ctrl-S> is sent when the receiving end of the connection wishes the sender to stop transmitting. A <Ctrl-Q> is sent when the receiver wishes the sender to resume. The choice of these particular two characters is arbitrary. If the sender does not immediately cease transmission upon receipt of the <Ctrl-S>, characters may be discarded. Since this protocol chooses to ignore the flow control characters any part of a packet may be discarded.

More precisely stated consider X to be the receiver and Y to be the sender. The packet sent is represented by the string abc where a, b, and c are data segments of unspecified size. X may receive one of:

1. abc
2. ab
3. ac
4. bc

For case [1] the correct data is received and no special action need be taken.

For cases [2], [3], and [4] we have a situation identical to data dropped during transmission. This is handled by the same checksum, time-out and retransmission strategy already described.

Assume Y is not now in the act of receiving a packet, then Y sees the two characters <Ctrl-S> and <Ctrl-Q> appear as input in that order. Y is waiting for a message to appear and so expects to see a SYNCH pattern. If the two characters "<Ctrl-S><Ctrl-Q>" are not part of a SYNCH pattern then they will be immediately discarded. If Y is receiving a packet then the <Ctrl-S> and <Ctrl-Q> are seen to be added noise characters and would be detected by the checksum tests. The packet being received would require retransmission.

The question of which character to pick for the SYNCH pattern is slightly muddied by the above observation. To the author's knowledge <SOH> is rarely if ever picked for flow control. This is part of the motivation in using it as the SYNCH pattern.

How does one guarantee that any data will actually arrive successfully? The initial choice of maximum data counts during connection establishment is very important. Some knowledge of one's own operating system must be assumed. If it is known for example, that streams of data in excess of a certain length will often trigger flow control at the connection baud rate, then the maximum data count should be chosen sufficiently lower that flow control rarely will be employed. An intelligent choice of the maximum data count will guarantee that some packets will arrive without encountering flow control.

6.7. Packets that are too Large

Assume a packet arrives which passes its header checksum test but whose LENGTH is larger than the MDL of the receiver. In such a case the sender has violated the protocol or a packet has a data error in the LENGTH octet and has passed the header checksum test. The latter is unlikely so that we assume the former. The receiver will abort his connection. The sender must inform the user "Error: Connection aborted due to MDL error", and go to the CLOSED state.

When the MDL is exceeded the receiver will transmit a legal reset:

<SN=received AN><CTL=RST>

6.8. Packets that are too Small

Assume that a packet has passed its header checksum test but some of the data octets have been dropped by the link. In such a case the receiver's routine which reads data and builds packets is expecting octets which do not arrive. After SRTT the sender will retransmit this packet to the receiver. The receiver will now have enough data to complete the packet. Almost certainly however it will fail the data checksum test. As with any bad packet the receiver will rescan from the octet immediately following the SYNCH pattern for the next SYNCH pattern. In this manner the receiver will eventually see the SYNCH pattern of the retransmitted packet.

I. Inability to Transmit/Receive 8-bit Data

There are some older operating systems and devices which do not permit 8-bit communication over an RS-232 link. Most of these allow restricted 7-bit communication. Where this is an unavoidable problem both ends of the connection must have a protocol layer beneath this protocol. This lower layer will unpack packets it sends over the RS-232 link. It will also repack packets it receives over the RS-232 link. RATP will automatically determine whether or not full 8-bit or restricted 7-bit communication is being used (see below).

The strategy chosen for restricted 7-bit communication is called 4/8 packing. That is, each octet to be sent will be broken up into two 4-bit nibbles. The order of transmission is the high order four bits followed by the low order bits. Each octet to be received will be repacked by the inverse function. The high order nibble will be received first then the low order nibble. These two nibbles will be reassembled into an octet.

I.1. Encoding for Transmission

For those systems which are incapable of 8-bit data transmission over RS-232 links, there are operating systems which in addition place special restrictions on the non-printable ASCII characters. The encoding for 4/8 packing should restrict itself to transmitting data only in the printable 7-bit ASCII range.

I.2. Framing an Octet

The seventh and highest order bit of a transmitted 7-bit ASCII byte is a flag used to indicate whether the high or low order nibble of an octet is contained in this character. This flag bit if set implies that a new octet is being received and that this printable ASCII character contains the high order nibble of an octet in its four low order bits. In addition it implies the next ASCII character received should not have its highest order bit set.

This high order flag bit is set by adding the ASCII character "@" (octal 100) to a data byte. Thus the first nibble of an octet is always transmitted with "@" added to its value. The high order nibble will be transformed into the characters "@" through letter "O".

The lower order nibble of an octet is transmitted with zero "0" added to its value. The low order nibble will be transformed into

characters "0" through "?". When receiving 4/8 packed data, any characters not within the range "0" through letter "O" are discarded.

The octet whose octal value is 45 will be transmitted as two 7-bit printable ASCII characters:

```
           +-------------+
High order |1|0|0|0|1|0|0| First transmitted ("@" + data) = D
           +-------------+
Low order  |0|1|1|0|1|0|1| Second transmitted ("0" + data) = 5
           +-------------+
```

Since data bytes may be dropped or added at any time it is important to know always which portion of an octet is expected and to deliver only complete octets to the higher protocol level. If a single 7-bit character were completely dropped without being noticed the data stream delivered to the higher level could be shifted by an odd multiple of four bits. In the worst case this condition could remain indefinitely and the higher level would never receive an octet correctly. In such a case no packets would be correctly received, leading to an unusable connection.

To avoid this problem octets are assembled using a state machine driven by the presence of the high order flag bit. The presence of that bit in the 7-bit printable character indicates the beginning of a new octet. The two state machine which assembles octets is described below. A byte received with the high order flag bit set is called "HIGH", the byte without "LOW".

State 0

[Start state] Read a byte from the legal restricted set. This is determined by seeing if the byte is in the legal range "@" to the letter "O". If it was not discard the byte and return to this state.

A HIGH byte was read. Place the four low order bits of the byte into the four high order bits of the assembled octet and go to state 1. Otherwise discard the byte and return to this state.

State 1

Read a byte from the legal restricted set. This is determined by seeing if the byte is in the legal range zero "0" to the letter "O". If it was not discard the byte and return to this state.

If a LOW byte was read subtract zero "0" from the byte placing the four low order bits of the result into the four low order bits of the assembled octet. A full octet has now been assembled. Pass the octet to the higher level and go to state 0.

Otherwise a HIGH byte was read. Place the four low order bits of the byte into the four high order bits of the assembled octet and return to this state.

Utilizing this state machine to receive 4/8 packed data ensures that the data stream delivered to the higher level will not permanently remain shifted an odd multiple of four bits. The restriction placed upon bytes read removes obviously bad data and in some cases would handle uncontrolled padding or blocking insertion.

I.3. Automatic Detection of 8-bit or 4/8 Packed Data

It is an unavoidable problem that some machines cannot handle unrestricted 8-bit data. Since this is given, it is desirable to be able to automatically detect whether unrestricted 8-bit or restricted 4/8 packing is being used to transmit data on a connection. For the purposes of this discussion those machines capable of transmitting and receiving both unrestricted 8-bit and 4/8 packed data are called smart. Machines are called dumb if they can only transmit and receive 4/8 packed data.

When initiating a connection there are four possible machine configurations and they are:

1. A (smart) opens a connection to B (smart).

2. A (dumb) opens a connection to B (smart).

3. A (dumb) opens a connection to B (dumb).

4. A (smart) opens a connection to B (dumb).

Each case is examined and extensions to the behavior for the LISTEN and SYN-SENT states are provided which allow both types of machines to initiate or receive a connection.

Cases 1 and 2: LISTEN Behavior for a Smart Machine

In these cases machine A initiates a connection to B who is assumed to be in the LISTEN state. B must be able to passively detect whether 8-bit or 4/8 packing is being used and respond accordingly. The method B uses relies upon the detection of a valid first packet. In the LISTEN state B attempts to simultaneously treat the incoming data as if it were both unrestricted 8-bit and 4/8 packed.

The incoming data is in effect fed to two different receiving algorithms. The detection of a valid header will occur to one of these algorithms before the other. If the first valid header was read assuming unrestricted 8-bit data then any resulting connection is assumed to use unrestricted 8-bit data for the life of the connection. If the first valid header assumed 4/8 packing then the resulting connection is assumed to use 4/8 packing for the life of the connection. In the case of the detection of illegal condition in the LISTEN state the protocol will reply with a RST packet in kind.

Case 3: LISTEN Behavior for a Dumb Machine

In this case machine B is the recipient of a connection request and is capable of handling only 4/8 packed data. The LISTEN behavior for machine B assumes that all connections are 4/8 packed. It never deals with unrestricted 8-bit data. As a result it will refuse to open a connection request from a smart machine (see case 4 below).

Case 4: SYN-SENT Behavior for a Smart Machine

In this case machine A attempts to open a connection to machine B. However, A has no knowledge of B’s capabilities. A will send its connection request assuming B is smart using unrestricted 8-bit transmission. It will await a reply assuming the response will be unrestricted 8-bit also. If B is in fact dumb it will not return a SYN-ACK because of the restriction imposed by case 3 above. If no connection is made with B using 8-bit data the entire connection initiation is restarted assuming B is dumb, 4/8 packing is used and the response is assumed to be 4/8 packed as well.

The cost of this approach is a longer time to determine whether or not it is possible to open a connection to B. It is twice as long. The advantages of being able to automatically adjust to either unrestricted 8-bit or 4/8 packed data out weigh this disadvantage. RATP will not exhibit the schizophrenic behavior of many other asynchronous protocols when dealing with both classes of machines.

II. A Brief Survey of Some Asynchronous Link Protocols

II.1. DDCMP

DDCMP, Copyright (c) 1978 Digital Equipment Corporation [DDCMP 78], is a reliable point-to-point and multi-point transmission protocol is used by many of that manufacturer's computers. DDCMP does provide reliable asynchronous two way data transmission.

Some of the decisions taken in the design of DDCMP reflect its orientation toward multi-point data links. This leads to headers which are substantially longer than needed for two way point-to-point communications.

DDCMP allows as many as 255 outstanding unacknowledged messages. DDCMP does specifically mention that a particular end of a connection may choose to limit the send queue to one outstanding unacknowledged message. It also allows sending a stream of outstanding unacknowledged packets. Unless all RS-232 implementations of DDCMP were limited to a single outstanding packet, the collision with existing flow control restrictions could lead to very low thruput. (DDCMP is assumed to have control over the link driver. Dealing with various differing flow control mechanisms is not a consideration.)

DDCMP uses a CRC polynomial for data protection which is difficult to calculate for many machines without special hardware [TCP Checksum 78]. Many Digital Equipment computers have such hardware.

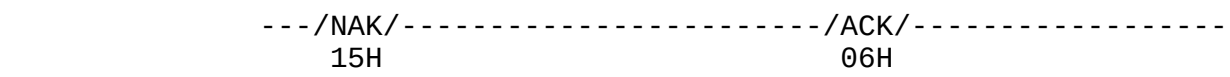
DDCMP does not provide the receiver with the ability to restrict incoming packet size. It is true that all the higher level protocols built on top of DDCMP could separately negotiate packet size. But this burden would then be moved away from the link level where it properly resides.

Generally, a full implementation of DDCMP is too complex for consideration. If one were to implement 'part' of the protocol then issues of compatibility with already existing implementations on other computers are raised.

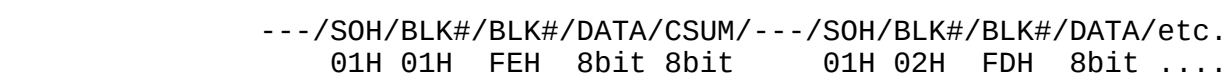
II.2. MODEM Protocol

This is a protocol in common use amongst microcomputers. The description here comes from

> MODEM/XMODEM Protocol Explained by Kelly Smith, CP/M-Net "SYSOP" January 8,1980
>
> .... Data is sent in 128-byte sequentially numbered blocks, with a single checksum byte appended to the end of each block. As the receiving computer acquires the incoming data, it performs its own checksum and upon each completion of a block, it compares its checksum result with that of the sending computers. If the receiving computer matches the checksum of the sending computer, it transmits an ACK (ASCII code protocol character for ACKNOWLEDGE (06 Hex, Control-F)) back to the sending computer. The ACK therefore means "all's well on this end, send some more...".
>
> The sending computer will transmit an "initial NAK" (ASCII protocol character for NEGATIVE ACKNOWLEDGE (15 Hex, Control-U))...or, "that wasn't quite right, please send again". Due to the asynchronous nature of the initial "hook-up" between the two computers, the receiving computer will "time-out" looking for data, and send the NAK as the "cue" for the sending computer to begin transmission. The sending computer knows that the receiving computer will "time-out", and uses this fact to "get in sync"... The sending computer responds to the "initial NAK" with a SOH (ASCII code protocol character for START OF HEADING (01 Hex, Control-A)), sends the first block number, sends the 1's complement of the block number, sends 128 bytes of 8 bit data, and finally a checksum, where the checksum is calculated by summing the SOH, the block number, the block number 1's complement, and the 128 bytes of data.
>
> Receiving Computer:



> Sending Computer:



This process continues, with the next 128 bytes. If the block was ACK'ed by the receiving computer, and then the next sequential block number and its 1's complement, etc. ....

As can be seen from this partial description the MODEM protocol is unidirectional, data can only pass from the sender to the receiver in a stream. In order for data to flow simultaneously in the other direction another connection over another RS-232 line would be required.

In addition this protocol is restricted to a fixed 128 octet packet size. Many front-end concentrators are unable to service such large incoming packets. It has been observed many times that the concentrator of a busy DECsystem-20 can invoke flow control on input at 1200 baud for packets as small as 64 characters.

II.3. KERMIT System

The KERMIT system, Copyright (c) 1981 Columbia University, is a file transfer environment developed recently. It has implementations which run on DECsystem-20, IBM 370 VM/CMS, 8080 CP/M based systems, and the IBM PC among others.

KERMIT combines both the reliable transfer and file transfer into a single package. Extension to other applications and higher level protocols would be possible but the boundary between the reliable transfer and application layers is very indistinct. It violates the layering design strategy the Internet employs.

There is a limitation of transmission to the restricted printable ASCII set for certain computers but not for others. This leads to confusion. KERMIT allows both restricted ASCII and 8-bit transmission.

The KERMIT protocol does have a method of setting MDL at connection initiation. It is limited to a smaller maximum packet size, 96 as opposed to 261 octets. Kermit originally used a checksumming algorithm limited to six bits. This is considered to provide too low a level of error detection capability for data packets. Kermit now allows two other checksumming algorithms in addition to the original. There must be a negotiation between sender and receiver regarding which algorithm to use.

The KERMIT protocol does not appear to make provision for both sides of a connection attempting an active open simultaneously. One side must be an initial "sending Kermit" and the other a "receiving Kermit". The code published as a KERMIT implementation

guide cannot recover from simultaneous active opens, it immediately ABORTs. This reflects a bias towards unidirectional data flow.

The KERMIT packet type (similar to RATP control flags) specifies whether an ACK/NAK is contained in the packet, or data, etc. These are mutually exclusive and piggybacking an ACK on a data packet is not possible. This can be a source of overhead. In addition KERMIT restricts the sender to a single outstanding unacknowledged packet as does RATP. It allocates an entire byte to the sequence number which is unnecessary.

On the subject of error recovery, the size of a packet is contained in the second byte of the packet and is not protected by a header checksum. If the length field was in error due to noise on the link, it could be longer than the correct packet size. The code published as the KERMIT implementation guide relies upon the detection of the <SOH> character anywhere in a packet to indicate the beginning of a packet header. It re-SYNCHs using this technique. This is only possible if binary data in a packet is quoted. If full eight bit data is transmitted it does not appear that the KERMIT protocol rescans for a new MARK (SYNCH) character within the bad packet data just consumed. It will under these circumstances throw away the retransmitted packet or portions thereof. Re-SYNCHing under such conditions is problematical.

REFERENCES

[Cohen 81]

Cohen, D. On Holy Wars and a Plea for Peace. IEEE Computer, October, 1981.

[DDCMP 78]

DDCMP AA-D599A-TC edition, Digital Equipment Corporation, 1978. Version 4.0.

[IP 81]

Postel, J. DOD Standard Internet Protocol [RFC-791] Defense Advanced Research Projects Agency, 1981.

[TCP 81]

Postel, J. Transmission Control Protocol [RFC-793] Defense Advanced Research Projects Agency, 1981.

[TCP Checksum 78]

Plummer, W. W. TCP Checksum Function Design. Technical Report, Bolt Beranek and Newman, Inc., 1978.

EDITORS NOTES

This memo was prepared in essentially this form in June 1983, and set aside. Distribution at this time is prompted by the the "Thinwire" proposal described in RFC-914.

--jon postel