AN X.PC/TCP PROTOCOL TRANSLATOR

Jose M. Rodriguez
Unisys Corp.
7929 Westpark Drive
McLean, VA 22102
jrodrig@edn-vax.arpa

ABSTRACT

The design of a computer communications device is presented whose purpose is to act as a translator between two virtual circuit protocols: Tymnet's X.PC, and DOD's TCP. Protocol translation in general and details on how the X.PC/TCP translation is done are discussed. Problems found in the task of protocol translation are also discussed.

1.0 Introduction

The purpose of this paper is to present the design of a network protocol translator that translates between two virtual circuit protocols: Tymnet's X.PC [1] and the Department of Defense's Transmission Control Protocol (TCP) [2]. X.PC is based on CCITT's X.25 [3] and therefore this design would be useful in the design of an X.25 to TCP translator.

This design was part of an internal R&D program whose goal was to design and build an architecture for connecting personal computers to the Department of Defense's Defense Data Network through telephone (POTS) lines [4].

The translator was implemented on a dedicated microprocessor system consisting of 2 CPU cards with Intel 8086 microprocessors on a Multibus I chassis. This system has eight serial asynchronous ports for personal computers and a HDH interface for a connection to a Bolt, Beranek and Newman C/30 packet switching node (PSN).

A brief overview of X.PC and TCP should be useful. TCP is an end-to-end layer 4 transport protocol designed to operate in a wide variety of communications subnets. It requires little of its communications subnets. X.PC is a simplified X.25 so it falls between layers 3 and 4 of the ISO model. X.PC is designed to operate in a point-to-point serial asynchronous link. TCP in principle can support a total number of connections between two to the sixteenth power to two to the sixty-four power. X.PC is limited to 16. Commands in TCP are implemented as bits in the TCP header. The format of this header is the same for all TCP PDUs. In X.PC commands are binary encoded octets located in the X.PC header. X.PC PDUs differ in format depending on the command in the PDUs.

The TCP protocol specification includes a canonical upper level interface that all implementations must follow. This interface includes the "listen" request. This request is used to indicate to the TCP implementation that an attempt by some remote TCP to establish a connection should be replied positively and that the connection should be established. The "listen" request requires a 16 bit parameter, called the local port number, that is used to differentiate between distinct connection attempts made by remote TCPs. When the "listen" request is not used, the local TCP automatically refuses any attempted connections by remote TCPs.

Out of band signaling in TCP is done through the "urgent" bit that requests higher priority processing of data in the data stream. In X.PC there there is an interrupt PDU that can be used to send expedited data. Their similarity is exploited by the translator.

The differences between X.PC and X.25 in terms of features are few. For example, the call establishment and call closing phases of both protocols are the same. The biggest differences are in the timers - X.PC uses fewer, and the number of simultaneous connections supported: X.PC supports 16 while X.25 supports

0308 CH2534-6/88/0000-0308 $1.00 © 1988 IEEE

1024. Another difference is that the X.25 specification indicates a standard for addressing the remote computers being called, while the X.PC specification ignores how to do this. In terms of location of functions there is a big difference between X.25 and X.PC. While X.25 relies heavily on the features of the LAPB protocol, X.PC's link layer is very simple and provides few facilities. This layer only provides packet boundary information and data integrity information using a checksum.

2.0 Protocol Translation in General

There are several types of real time protocol translation methods. There is the "meat grinder" type where fields in a protocol data unit (PDU) of protocol A generate through formulas and bit field rearrangements a PDU of protocol B. There is the pure translation type where the translation is done by a client process, client of the implementations of protocol A and B on the translating device.

Then there is the hybrid translation method where the translating process resides on top of the protocol implementations but is able to manipulate the internals of the implementations of the protocols being translated. The design presented here is of the hybrid type. The above list of types is not meant to be exhaustive.

Currently there is a great interest in protocol translation particularly between the DOD protocol suite and the ISO protocol suite. For example, the National Bureau of Standards is building under contract from the Defense Communications Agency a protocol translator between DOD's File Transfer Protocol and ISO's FTAM [5]. Other groups are working at protocol translation at the network (or internetwork) layer, from DOD's Internet Protocol (IP) to ISO's IP. The purpose of this translation would be for networks and gateways (routers in ISO parlance) of one protocol suite to be able to act as transit systems for systems of a different protocol suite.

3.0 Description of the Protocol Translation in this Design

The type of protocol translation presented in this paper can be characterized as a pure PDU to PDU translation. There is no encapsulation, and no staging. The two users ends are not even aware that two different virtual circuit protocols are being used to provide the requested service.

This design does not take a PDU of protocol A, rearranges the bits in the PDU and come up with a PDU of protocol B. Given the different timers, sequence and acknowledgement numbers, etc. of TCP and X.PC, this would not work. The reason it is a pure PDU-to-PDU translation is that for each PDU of one protocol received, an equivalent PDU in the other protocol is shortly generated and sent. See table 1 and table 2 for equivalent PDUs between TCP and X.PC.

XPC PDU		TCP request
CALL REQUEST	->	OPEN REQUEST
RESET REQUEST	->	TCP ABORT
CLEAR REQUEST	->	CLOSE REQUEST
XPC DATA	->	TCP SEND
INTERRUPT	->	URGENT SEND
CALL CONFIRM	\	
RESET CONFIRM	\	
CLEAR CONFIRM	\	
RECEIVE READY	>	not translated
RECEIVE NOT READY	/	
RESTART REQUEST	/	
RESTART CONFIRM	/	

Table 1. Translation of X.PC PDUs into TCP requests.

TCP response		X.PC PDU
OPEN ID	\	CALL CONFIRM *
OPEN SUCCESS	/	
OPEN FAILURE		CLEAR REQUEST
TERMINATE		CLEAR REQUEST
CLOSING		CLEAR REQUEST
DELIVER		X.PC DATA
TCP ALLOCATE		not translated

Table 2. Translation of TCP messages into X.PC PDUs.

* - OPEN ID and OPEN SUCCESS must be in sequence for the translation to be done.

During the design stage of this project, a different approach to the one described here was considered. This approach involved the use of X.PC as an interface protocol that was not directly translated to TCP requests. This scheme consisted of establishing X.PC connections between the X.PC end and the translator without opening a TCP connec-

tion at the same time. After opening an X.PC connection, commands would be sent through it as data. This approach is very similar to those used in loosely coupled network front ends. The commands over the X.PC channel would form a Service Access Protocol (SAP) [6]. The advantage of this approach is that the SAP can be designed to fit exactly the interface of a TCP implementation. This approach was discarded because its highly programmed behavior was not needed and would introduce a protocol layer between the DOD applications and X.PC in the personal computer.

4.0 Design of the Translator

The design of the translator was influenced by two considerations. First, there was an operational and tested TCP implementation for the system planned to be used as the translator. This implementation has a MIL spec conformant upper level interface. Second, all connections will be initiated by the X.PC end system (i.e. the translator will never send a CALL REQUEST PDU to the X.PC end system). This does not preclude the use of TCP listens though.

The first consideration suggested that a general purpose implementation of X.PC could be written for the translating device and that the translating process could be a client process accessing both X.PC and TCP through their upper layer interface. For several reasons, one of them expediency, this was not done. The X.PC implementation in the translating device has no upper layer interface and is closely tied to the protocol translation process. Whenever the translator receives X.PC PDUs, the necessary X.PC functions are done and then a request to the TCP implementation is generated.

The second consideration is acceptable given that for the purposes of this design, the X.PC end system would initiate all virtual circuit connections and would request "listens" from the translating device whenever connections are expected to be initiated by remote end systems.

The translating device or translator consists of 4 processes connected in a pipeline:

1) A process to handle the link layer of X.PC. This process verifies checksums of received packets, checks for disconnected asynchronous lines, creates and sends X.PC link layer packets.

2) The translator process which is in charge of the protocol translation. The X.PC network layer implementation is also part of this process.

3) A previously developed TCP implementation.

4) A previously developed IP implementation.

The device has two types of interfaces: 8 asynchronous RS232 ports used by X.PC and an HDH interface for a connection to a BBN C/30 packet switching node.

5.0 The Translator Process

The main idea behind the translator process is simple: to generate protocol events in protocol B because of the occurrence of equivalent events in protocol A. Protocol events are defined as requests from a client to a protocol implementation, reception of PDUs by protocol implementations or the reception of messages by a client from a protocol implementation. There must be a high degree of equivalence between protocols A and B or else the translation becomes difficult or impossible. While the translation of all events may not be necessary (and therefore events without equivalent ones may be ignored), untranslated events must be unimportant and not used by the protocols' clients. All events used by old, well established clients must be translated.

A more detailed view of the protocol translator process in this design can be described as a four step process:

1) Reception by the translator of a protocol A PDU from system A.

2) One or more protocol requests made to protocol B's implementation.

3) One or more replies received from protocol B.

4) A reply sent back to system A.

Not all PDUs received by the translator get translated. For example X.PC's Receive Ready (RR) packets have no exact match in TCP. The translator replies to a received RR packet with its own Receive Ready or Receive Not Ready

packet to the original party and updates its send counter. If this results in data buffers being freed then an allocation request would be made to the TCP implementation (because translator process can receive more data from the local TCP implementation).

6.0 Details of the Translator Process

The translator process is designed as an event driven process. There are three types of events: X.PC events, TCP events, and timer events.

X.PC events consist of the reception of some type of X.PC related PDU or action. In the final design there were only two types of X.PC events: the reception of an X.PC PDU or some action at the layer 1 or 2 of the X.PC interface like line connected, line disconnected, etc. TCP events consist only of TCP messages like replies, data received and status messages received from the TCP implementation. Timer events are created when a count down timer reaches zero, these timers are started by the translator process.

The translator process is formed by the implementation of a finite state machine that defines the states an X.PC connection can be. The implementation consists of a number of routines each corresponding to the state the connection is in and the type of event it handles. For example if a connection is in the CONNECTED state and a TCP event is received then a routine is called that exclusively handles TCP events while in the CONNECTED state.

The translator is implemented as a single process with three entry points. Each entry point forms a light weight sub-process. These sub-processes share the data space with each other. The process and its sub-processes never die and control must be relinquished by the sub-process because the system's scheduler is a non-preemptive one. Each sub-process is in charge of one type of event. The sub-process main body is a routine that calls the routine associated with the current state.

The X.PC implementation is merged with the translation procedures. Every time that an X.PC PDU is received that has a TCP equivalent, the implementation handles the event and then makes a request to the TCP implementation through the TCP interface. Those requests often result in the TCP implementation sending TCP PDUs. The following is an example of the translation process in pseudo-code form taken from the routine xpc_event_disconnected_action() handling the open requests from X.PC:

```
if (event is a CALL_REQ packet) then
   update_send_counter( event);
   result = open_protocol(
         protocol_control_block);
   if (result == CONNECTED) then
         protocol_control_block.
         connection_state = CONNECTED;
   else
         protocol_control_block.
         connection_state =DISCONNECTED;
         free( protocol_control_block);
   endif
endif
```

The following is the translation from a TCP event, a message containing received data, to the X.PC data PDU. It is part of the routine tcp_event_connected_action().

```
if (TCP event is a DELIVER message) then
   record TCP send allocation given
       in the TCP event;
   send any XPC to TCP data remaining;
   while (there is data in the
       TCP message)
   begin
       form an XPC data holding packet
          by removing data from the TCP
          message;
       queue the XPC data holding
       packet for future delivery;
   end
   send all XPC packets to the remote
       XPC end;
   set the T15 (data retransmission)
       timer;
endif
```

7.0 TCP and X.PC buffering and acknowledgement

Because the translator acts as an intermediate storage node for data flowing between the two user ends, the buffering and acknowledgment strategies used for handling the two virtual circuits are important issues.

Given that the translator uses a self-contained TCP implementation, the buffering and acknowledgement strategy of the TCP side of the translation consists of knowing when to send data and allowing the TCP implementation to receive data.

The TCP implementation gives a credit to its client in units of packets to be sent. The client gives to the TCP process credits (in number of bytes) for receiving data. In other words the client controls the reception of data and the TCP implementation controls the transmission of data.

The translator sends X.PC data to TCP when TCP grants a credit. Because of the speed difference between TCP and X.PC in the translator, TCP grants credits as fast as it receives X.PC data. In case the TCP traffic was slowed down or halted, if X.PC packets were received, the first one is buffered and the X.PC flow control mechanism is used to stop the flow. This mechanism consists of sending to the remote X.PC end a Receive-Not-Ready packet. Further X.PC packets received are dropped and not acknowledged. When TCP provides new credits for transmission, a Receive-Ready packet is sent to the remote X.PC end. This restarts the flow of X.PC data.

Data originating at the remote TCP end is sent and buffered in the translator until the remote X.PC end acknowledges it. This allows for the retransmission of data if not acknowledged on time. As soon as it is acknowledged and TCP to X.PC buffers are freed, the translator gives new receive credits (sends an ALLOC message) to the TCP implementation.

All the above refers to one virtual circuit formed by one X.PC and one TCP virtual circuit. The translator supports 24 X.PC-TCP virtual circuit pairs. Buffering, credits and acknowledgements between connections are completely independent.

8.0 Problems in the design of the translator

The biggest problem this design has, which is common with most packet switching protocol translator systems, is a mismatch between features of the protocols being translated. This mismatch can be very serious, preventing otherwise similar protocols from being translated. In the translator system presented here, the biggest mismatch is that there is no equivalent in X.PC (or in X.25) of the TCP passive listen request. This operation is very important for the implementation of server processes and is used in user FTP. Without it, user FTP could not be implemented in the personal computers which was a principal goal of the R&D project. Requests of TCP listens could not left out from the translator device.

There were two possible options for implementing the TCP listen request. One, whenever a CALL REQUEST X.PC PDU was received that should be a listen request, for the translator to hold on to the CALL CONFIRM X.PC PDU until a connection was established on the listen request. This was not possible because X.PC has a timer on how long a CALL REQUEST can remain unanswered. After a few minutes, X.PC sends a CLOSE REQUEST indicating that the call request has taken too long and the request should be canceled.

The second option was whenever a CALL REQUEST PDU requesting a listen was received, the translator process would make a listen request from the translator's TCP and would reply immediately to the X.PC end with a CALL CONFIRM message. As soon as a TCP connection is established a single null byte is sent to the X.PC remote end. This indicates that the full virtual circuit from one end to the other has been established. The problem with this is that while the first hop of the total virtual circuit has been established, the second hop, the TCP connection, has not been established and therefore any data received from the X.PC end has no place to go. There must be an agreement between the application implementation and the translator to send no data until the application receives the single byte from the translator. Any data received by the translator is discarded.

A similar problem of semantic mismatch is the handling of out of band (OOB) signaling. X.PC has an interrupt packet that can be used as an OOB signal, while TCP has the PUSH and URGENT bits. Because the URGENT seems the closest to the interrupt message, in this design these two PDUs are translated into each other. Currently, the translator ignores the PUSH flag when it is received in a TCP connection.

9.0 Conclusion

Currently the X.PC to TCP translator is operational. The following are some observations made during its implementation and testing. There is little

performance degradation caused by the translation process because one of the interfaces (the serial asynchronous one) is very slow - 1200 bps. The translator requires a fair amount of memory to support some useful number of simultaneous connections, each connection pair requires around 4 kilobytes.

This paper presents the design of a successful protocol translator. It succeeded because the two protocols being translated provide very similar services to their clients. One important feature mismatch, TCP's listen request, indicates the importance of protocol equivalence for a successful translation.

REFERENCES

[1] X.PC Protocol Specification, Publication Number - 269, Tymshare, Inc. 8 September 1983

[2] Military Standard Transmission Control Protocol, MIL-STD-1778, 12 August 1983.

[3] Recommendation X.25, Fascicle VIII.2, CCITT, 1980.

[4] Jose M. Rodriguez, "A Portable Computer Access Architecture for the Defense Data Network", IEEE Infocom '87, 1 April 1987.

[5] Michael Wallace, "Gateway Architecture Between FTP and FTAM", National Bureau of Standards ICST/SNA-86-6, 1986.

[6] John Day, WWMCCS Host to Front End Protocols: Specifications Version 1.0, DTI Document 78012.C-INFE.14, Digital Technology Incorporated, 5 November 1979.

[7] X.25/PLP - DDN/TCP Gateway SDL Technical Specification, REPORT NCS-0034-113, Booz Allen & Hamilton Inc., 5 May 1986.

[8] Irene Hu, "Managing the Transition from TCP/IP to ISO Protocols", Uniforum 1987, 20 January 1987.

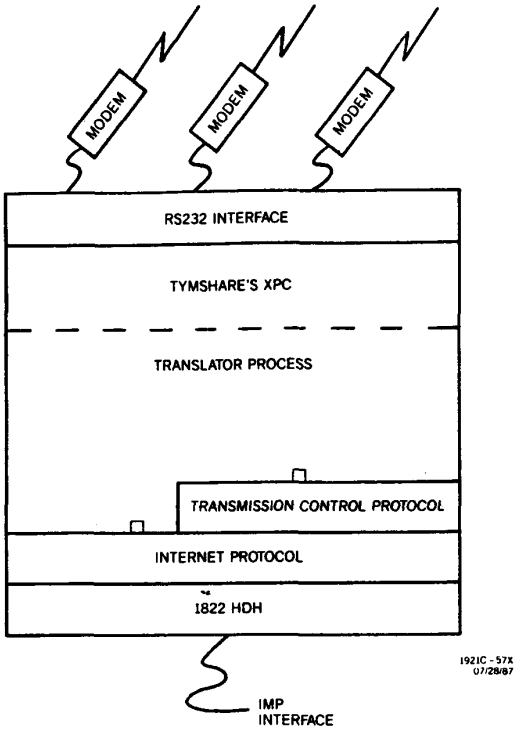

Figure 1. Translator Device