Inter-Office Memorandum

To	Communications Protocols	Date	December 20, 1978
From	David Boggs	Location	Palo Alto
Subject	TeleSwat Protocol	Organization	Parc

XEROX

Filed on: [Maxc1]<Pup>TeleSwat.bravo

There may not be a debugger in Peoria, but there is a debugger in Palo Alto and a way to move packets between Palo Alto and Peoria.

This memo documents the protocol by which Swat communicates with a remote Swatee. I maintain a lot of software which is used at other Xerox sites, and I often get calls from users when a program lands in Swat. The caller is rarely facile with Swat's command language, and it is very difficult to guide him through the steps necessary to gather debugging information (assuming for the moment that he has the symbol file on his disk). Many programs are also available as boot files, for which the only debugger available until now was SSD: a super simple octal debugger. The need for a cross-net debugger has been clear for a long time. Pup gateways have a rudimentary debugger server in them, but the effort required to build the remote user has prevented development, and it has remained only a toy. Bolt Beranek and Newman has developed several cross-net debuggers. I know of two: one for debugging ArpaNet Imps and one for debugging PDP-11 BCPL programs.

There are many ways to do remote debuggers. A useful way to classify them is to ask where the logic of the debugger runs: in the machine being debugged or in the machine where the human debugger is. I considered putting a Telnet server into Swat, and using Chat to control the remote Swat. Swat's keyboard and display stream would then be paralleled with the Telnet streams. The advantages of this approach are that it would use existing software, and Swat could then accept commands from either the remote Chat or the local keyboard, allowing joint debugging by the guy with the broken machine and the remote guru. The disadvantage is that it takes a lot of code to implement a Telnet server and Swat, like all Alto programs, is short on space. A server built on the standard Pup package would consume about 6K. Building a Telnet server from scratch is too much work. That eliminated this approach.

The interface between a debugger and a debugee is usually quite simple, and the other approach where the debugger runs in the remote machine with just a small nub in the debugee looked promising. The debugger must be able to fetch and store memory locations in the debugee, and tell it to stop and go. In addition there must be conventions for saving and restoring the debugee's state, and interrupting (manually and by breakpoints) and resuming. Since there is not always network software present in our machines and because of our dislike of omnipotent operating systems, it is not possible to force a remote machine into the debugger without its consent, so that eliminates the stop command. This approach greatly reduces the amount of mechanism in the remote machine - it is possible to write a server which implements Fetch, Store and Go in a few hundred instructions. The prospect of including such a debug nub in boot files and then being able to symbolically debug them with Swat convinced me that this was the way to go.

How Swat does it

Swat contains a very simple Pup Level 0 Ethernet driver, a Level 1 Raw Pup dispatcher, and a Level 2 TeleSwat User and Server. The ^Z command specifies the address space which Swat peers into: any bank of memory, or any file created by OutLd (usually, of course, this is Swatee), or any host in the internet, in which case Swat becomes a TeleSwat user and the remote host is assumed to implement the server half of this protocol.

The $$^Y command makes Swat a TeleSwat server: it then ignores the keyboard and answers Fetch, Store and Go packets from the net. If the target address space is set to Swatee using the ^Z command before making it a server, then things work as you would expect, except remotely. If the server's target address space is set to bank 0, the user Swat will be examining the server Swat's running core image (in principle the server's target address space could even be another host...). Servers other than Swat are not expected to be this fancy; the address space which the server references on behalf of the user is outside of this protocol.

The TeleSwat user is where the debugger logic is running and most of the Swat in the server is not being used. A boot file, which usually doesn't hook up to the local disk and therefore can't call Swat, can contain a tiny debug nub which implements the server half of the TeleSwat protocol and which gets control when the program tries to call Swat. In this case Fetch and Store requests go to the running core image.

The Protocol

The protocol is designed to minimize work for the server, placing the burden on the user. The server is completely passive: it only does something in response to a command from a TeleSwat user, and the only packets it generates are acknowledgements. Duplicate suppression, retransmission of lost packets, etc is the responsibility of the user. The protocol is connectionless: except for the Go command, the server is not required to maintain any state from one packet to the next.

There are three commands and five packet types. The user should employ the Pup ID as a packet sequence number for duplicate suppression. To respond to a request, the server need only set the type to 'ack', exchange the source and destination ports, append the requested information if any, and send the Pup back to its physical source (note that the server doesn't have to worry about routing).

In Fetch and Store commands, the user may optionally request that the server send back a block of words surrounding the word being fetched or stored. The server may ignore this and only send back the requested word, but if it complies, the user can then implement a cache and reduce the number of packets it generates. When the machines are directly connected via an Ethernet, the cache doesn't buy much, but when they are separated by a 9.6 KB hop, it wins big. I have experimentally determined that the optimum block size is about 16 or 32 words; Swat asks for 32 word blocks. The size of the surrounding block of memory requested by the user can be any power of two up to 256; the server may choose to send a block of a different size (for example: the server may have a small packet buffer which can hold a block of up to 32 words; if the user requests a 256 word block the server may respond with a 32 word block). The base address of a block is the address of the fetch or store command which the ack packet is acknowledging ANDed with the negative block size in the ack packet.

The Go command involves a 3-way hand-shake to protect against lost acknowledgements. The problem is similar to closing a connection. The user says Go; if the packet is lost, no ack comes back and the user retransmits. If the ack is lost, the user also retransmits since he can't distinguish this from having the Go command clobbered. If the server resumes the Swatee as soon as it receives a Go, but its ack is lost, the user will be unsure of whether the server heard it, since the server has stopped listening and isn't around to retransmit acks. So when the server receives a Go command, it acknowledges it and then *dallies* for up to 10 seconds, so that it can retransmit a lost ack. When the user gets the ack, it sends a GoReply packet (the third packet in the hand-shake sequence). If the server gets this and the previous packet was a Go, it stops dallying and resumes the Swatee. If it gets any other command, it abandons the Go command. If the GoReply command is lost, the server dallies for 10 seconds and then resumes the Swatee. The Pup ID of a

GoReply packet should be one greater than the previous Go packet.

Details

All numbers are octal. The well known TeleSwat server socket is 60. For this description (and historical reasons) the data words and bytes in a Pup are numbered from one, not zero.

Store (user to server)

> Pup Type: 200
> Pup ID: arbitrary (server sends it back in the ack)
> Pup Contents: the first data word of the packet contains the address of the word to be stored into. The second data word is the value to store. If the third data word is non-zero, then the user is requesting the server to send a block of that many words surrounding the address specified in word 1. The block should reflect the result of the store.
>
> Ack: The first data word of the ack is the address of a word in the debugee. The second data word is ignored. If the third word is non-zero, then a block of that many words of the debugee begins in the fourth data word. The base address of the block is (word 1) AND - (word 3). Note that the server doesn't have to send a block, and may choose to send a block of a different size, usually smaller.

Fetch (user to server)

> Pup Type: 201
> Pup ID: arbitrary (server sends it back in the ack)
> Pup Contents: the first data word is the address to fetch. The second data word is ignored. If the third data word is non-zero, it is a request for a block of that many words as in the Store command.
>
> Ack: The first data word is the requested address. The second data word is the contents of that address. If the third data word is non-zero, then the interpretation is as for the Store command.

Go (user to server)

> Pup Type 202
> Pup ID: arbitrary (server sends it back in the ack)
> Pup Contents: none
>
> Ack: no contents

GoReply (user to server)

> Pup Type: 203
> Pup ID: the ID of a previous Go command plus one
> Pup Contents: none
>
> Not acknowledged

Acknowledgement (server to user)

> Pup Type: 204
> Pup ID: same as corresponding request
> Pup Contents: See descriptions above.

Caution : the length of the data portion of a Fetch or Store packet can be as little as 2 (fetch) or 4 (store) bytes, in which case word 3 (block length) is meaningless. Similarly for acks: if the length is less than 6 bytes, no block follows.

Revision History

December 11, 1978: first release.

December 20, 1978: the second data word in a Store ack is unspecified.