Implementation of TCP, UDP and IP on Wanda

T J Richardson

February 8, 1993

1 Overview of the TCP/IP Protocol Suite

TCP/IP was originally developed by DARPA to connect machines on the ARPANET together. The original TCP/IP implementation for the Unix operating system was done by the Computer Department at the University of California at Berkeley. This eventually became part of public domain software and most implementations of TCP/IP are based on this software.

Key documents describing the definition of the protocol are:

```
RFC 791 :- Internet Protocol (IP).
RFC 792 :- Internet Control Message Protocol (ICMP).
RFC 793 :- Transmission Control Protocol (TCP).
```

There have also been a number of books devoted to explaining TCP/IP including:

```
TCP/IP and NFS: Internetworking in a UNIX environment - M. Santifaller.
Unix Network Programming - W.R. Stevens.
The Simple Book: An Introduction of Management of TCP/IP Internets
         - M.T. Rose.
Interneting with TCP/IP: Volumes 1 and 2 - D. Comer.
```

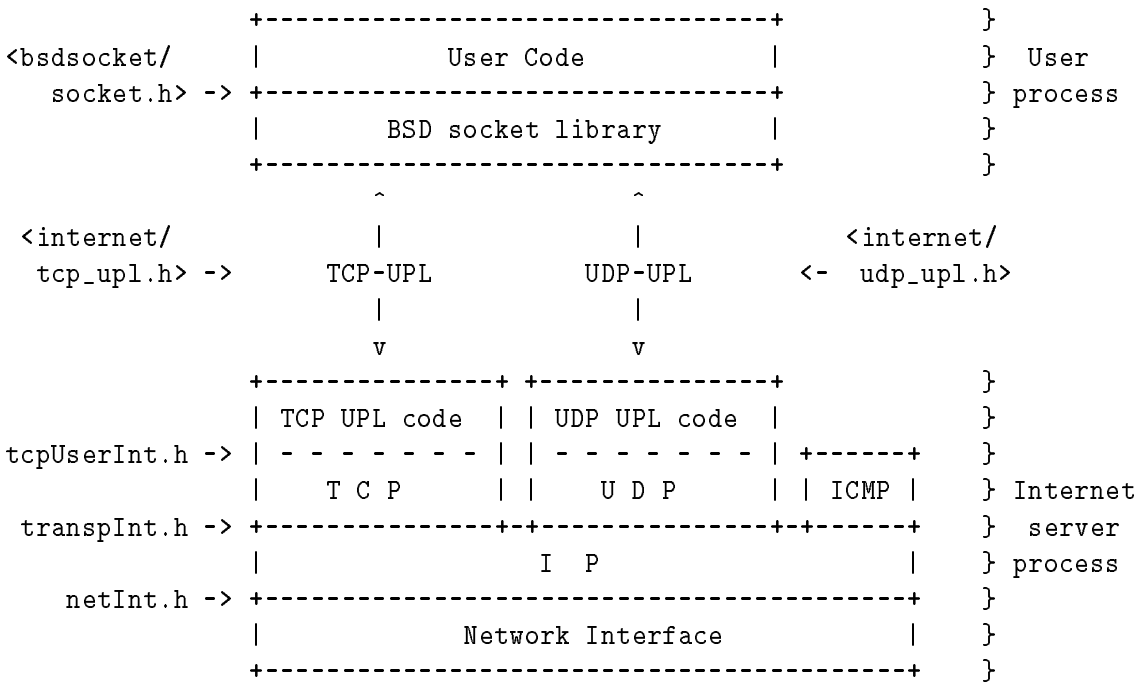
The Multi-Service Network Architecture (MSNA, see other documents in this collection) does not define a reliable transport layer so one needs to be implemented. It seems logical to implement the whole of TCP/IP then use "tunnelling" to treat MSNL as a single network as far as IP is concerned. This allows communication with any Unix machine which supports IP/MSNL, then by implementing network interfaces other than the MSNL "tunnel" (like ethernet), communication is also possible with Unix machines which do not support MSNL.

2 Previous Work

As his research project for the Third Year Computer Science Tripos, Anthony Gould (Fitzwilliam College) undertook the implementation of UDP and IP over Wanda based on raw Ethernet interface. This work was largely successful as an undergraduate project but lacked modularity and clean design making it difficult to extend or improve. The current implementation has no basis on this code and in fact much time was wasted trying originally to base the new implementation on this code.

3 Design

The TCP/IP implementation can be divided into several layers as in other implementations (at each interface I have shown the header file describing the interface):



There are two main differences between this and many other implementations.

Firstly the implementation is done entirely in user space rather than in the kernel. This introduces extra complexity at the user interface, but it is believed that the modularity of this approach should compensate for any loss of performance. For example, one advantage of the user space internet server is that a machine which is short on memory but has a reliable MSNL connection to another machine could potentially use the other machine's internet server, thus needing only the BSD library code (though addressing problems may arise). Having said this, there is no reason why the code should not be put in the kernel without any major changes if desired. The main addition needed would be a `WandaIPC...` system call interface (replacing everything above the `tcpUserInt.h` line in the diagram).

Secondly no secondary buffering is used to minimise copying. Wanda IO buffers are used at all levels up to the socket code where the data is copied to/from the area of memory given by the user. This means that it is easy to run out of IO buffers if you are not careful. If several connections are likely to be needed then the default IO buffer configuration in the kernel may need to be changed (alter the file `kernel/include/kernel/ioBufConf.h`). Note also that since the UPL has no flow control, uncontrolled sending by the remote or local machine will cause IO buffer starvation (the solution to which is to replace the UPL with a wonderful MSRPC interface.... - any offers?!).

4 Implementation

The implementation is contained within the internet directory of the wanda applications (wapps) tree. This is divided into subdirectories include, server, bsdlib and programs

which contain header files, the server code, the BSD socket code and some test programs respectively.

4.1 Network interfaces

At present two are implemented, MSNL and Ethernet.

The MSNL interface's main job is to manage connections since it is simulating a connectionless datagram delivery service over a connection-oriented service. It is implemented in the file `netIntMsnl.c`. At present a connection is closed if 60 seconds elapse without receiving a packet.

The Ethernet interface includes an implementation of ARP. This is implemented in the files `netIntEther.c`, `arp.h`, `arp.c` and `arpTimer.c`. ARP cache entries are timed out after ten minutes. Note that by default user space raw ethernet access is not allowed, so the kernel used needs to have the raw ethernet wanda socket interface enabled. Also at present IP broadcast packets are ignored.

4.2 Interface between IP and network interface

Defined in `netInt.h`. In the network interface layer, a thread (or possibly several threads) waits for a packet to arrive over the network and then calls `IpReceive` with the corresponding IO buffer. On the way down a thread in the IP layer calls the network interface's transmit routine with an IO buffer.

4.3 IP layer

Implemented in `ip.h`, `ipReceive.c`, `ipTransmit.c`, `ipRoute.c`, `ipReassembly.c` and `ipTranspInt.h`. At present some features are unimplemented. These are: fragmentation and options. Note that fragmentation is needed for running as a gateway. For details of routing see section 5 on environment variables below.

4.4 Interface between IP and transport layer

Defined in `transpInt.h`. Datagrams/packets/segments are passed as a list of IO buffers at this level (since they may have been fragmented and reassembled - this is where our IO buffer system complicates things over other implementations). Packets in from the network result in the transport layer's receive routine being called with a list of IO buffers. Conversely when the transport layer wants to send it calls `IpTransmit` with a list of IO buffers.

4.5 ICMP

This is at the transport layer for modularity but is really part of IP. Implemented in `icmp.h`, `icmpReceive.c` and `icmpTransmit.c`. All received ICMP packets should be responded to correctly, but we really also need a user interface to run programs like ping.

4.6 UDP

UDP is much simpler than TCP (in fact it's almost a 'null' layer over IP). It just does intra-host multiplexing and optionally a checksum. Implemented in `udp.h`, `udpReceive.c`, `udpTransmit.c`, `udpTcb.c` and `udpUserInt.c`. The UDP UPL interface is defined in

`<internet/udp_upl.h>` and just provides binding to a port, send, receive and close (unbind).

4.7 TCP

TCP is the most complicated protocol in the suite. In retrospect I would say it's too complicated and certainly a disproportionately large amount of time was spent on it. Again use of IO buffers complicates things over other implementations since TCP deals with a stream of bytes. A much more efficient transport layer could be written over MSNL by referring to buffers rather than TCP's method of referring to ranges of bytes in a "sequence space".

A TCB (transmission control block) is used to control each connection. Associated with it is a manager thread which divides its time servicing four types of event: a segment coming in from IP, an urgent segment coming in from IP, a segment from the user on the transmit queue being transmittable, and the occurrence of a timeout.

When a segment comes in from IP, the routine `TcpReceiveFromIp` is called by the IP layer. This puts the segment on either the receive queue or the urgent queue for the appropriate TCB, signals the manager thread and returns. The manager thread calls `TcpSegmentIn` (or `TcpUrgentSeg`) and processes the segment. This involves finding out the sequence numbers for the segment, resequencing segments if necessary, responding as appropriate according to the state of the connection. Incoming acknowledgements are used to stop segments from being retransmitted. Acknowledgements are sent for any data received and the data is passed to the user by calling the TCP UPL code's receive routine.

Going the other way, there is a thread in the TCP UPL code which calls the `TcpSend` routine when it wishes to transmit an IO buffer. This queues the segment on the transmit queue if there's room, signals the manager thread and returns. The manager thread checks if it can transmit the segment (whether it is within the send window) and if so then adds the TCP header and calls `IpTransmit`. Finally it sets a timer to retransmit the segment.

Looking in retrospect I think that the transmission for the first time of user data shouldn't be handled by the manager thread; the UPL thread which at present puts segments on the transmit queue should do the transmission itself, then put the segment on the queue for retransmission. This would simplify the manager thread but might require extra locking on access to the TCB (at present only the manager thread is allowed to access elements of the TCB other than those which were statically set up when the TCB was created).

The final event to consider is the timeout event. There is a single timer thread for all TCP connections. There are two types of timeout used at present: retransmission timeout and remove TCB timeout. The first type occurs if we do not receive an acknowledgement for a segment we have sent within the retransmission time. When this happens the manager thread is signalled and it retransmits the segment. The retransmission time for segments is calculated dynamically using Karn's algorithm. The remove TCB timeout is needed when the TCB is in the time-wait state. When this happens the TCB is removed and the manager thread exits. This is needed because when closing a connection is done in a certain order, it may be that stray segments are retransmitted, so the TCB and manager thread should still be around for a while afterwards to catch them.

4.8 TCP-User Protocol Level

The TCP server listens on a well-known MSNL port (`TCP_UPL_MAGICPORT`) for UPL connections from the user. There are commands to bind, listen, connect, send and close. As well as replies to the user for the bind and connect commands there are asynchronous messages to say a connection is established, data has been received, the remote host has closed or an error has occurred.

4.9 BSD Socket Library

The UPL is designed to make emulation of the BSD socket abstraction reasonably easy. The BSD socket interface gives some Unix compatibility.

The socket interface is implemented as library code which is included with each process using the internet server. The include files `<bsdsocket/socket.h>` and `<bsdsocket/internet.h>` contain the appropriate definitions for using the socket library. The socket calls supported so far are: socket, bind, listen, accept, connect, send, sendto, recv, recvfrom, close, select, getsockname, getpeername.

The code resides in the bsdlib subdirectory and is divided into protocol-independent code and protocol-specific code. The protocol-independent code is in `socket.c`, `sockstruct.h`, `util.c` and `util.h`. For TCP the protocol specific code is in `tcp.h`, `tcp_socket.c` and `tcp_recv.c`, for UDP in `udp.h`, `udp_socket.c`, `udp_recv.c`. Also there is support for MSNL sockets in `msnl.h`, `msnl_socket.c` and `msnl_recv.c` (use the include file `<bsdsocket/msnl.h>`).

On the receive side, a receiver thread takes IO buffers sent over the UPL connection by the internet server and queues them. Eventually the user must call recv or recvfrom when the data is copied out of the IO buffers into the user's buffer. It is important to realise that it is only at this point that the IO buffer is released This means that if a lot of data is sent to a very slow process, then all the system's IO buffers may be tied up and the system could crash. This issue needs to be addressed by an improved UPL which allows some kind of flow control. However this of course would be at the expense of passing more messages between the processes.

5 Using the code in your program

I'm assuming here that a post-snap4 version of the code is being used. The code in the snap4 tree is in a mess and should not be used. At ORL such a version can be found in the `/wanda/wanda.orl` tree. At the time of writing no wanda tree at the computer lab has a copy of the code, though the source can be found in `/homes/tjr10/wanda/internet`.

5.1 If the include files, library and internet server are not installed

I'm assuming here that (a) your program is called `yourprog.c` (b) it is in a wanda apps tree, and (c) this tree has an internet subdirectory in which the BSD library and internet server have been built. By default, when you build your apps tree these will be links to the (non-existent) installed versions, so you'll have to do "make local", "make depend" and "make all" in the internet directory to build them.

Firstly, `yourprog.c` should `#include <bsdsocket/socket.h>` and `<bsdsocket/internet.h>` (or `<bsdsocket/msnl.h>` for MSNL sockets).

Now put in your Imakefile:

```
INCLUDES = $(TOP)/internet/include
NormalProgramTarget(yourprog, yourprog.o, , $(TOP)/internet/bsdlib/libbsd.a,)
```

This will ensure that the BSD library is linked in with your program.

If you are using internet sockets then you also need to make sure that the internet server is running on the wanda machine in addition to your program. This is done by splicing `$(TOP)/internet/server/internet_svr` onto the kernel in addition to your program (or use (sas)wrun if you're using the procsvr).

5.2 If the include files, library and internet server are installed

This is slightly easier. Put in the Imakefile:

```
NormalProgramTarget(yourprog, yourprog.o, , $(LIBDIR)/libbsd.a,)
```

Then use `$(BINDIR)/internet_svr` as the internet server.

5.3 Environment Variables

There are several environment variables used by the TCP/IP implementation. Firstly the network interfaces need to know their IP addresses. The msnl network interface expects to be given its address in dotted decimal notation in "`MSNLIPADDR`". The ethernet interface expects its address in "`ETHERIPADDR`". If these environment variables are not present then the corresponding network interface will not be registered with IP.

IP routing information is specified in the hostinfo file by the environment variable "`IPROUTE`". Several entries may be given, separated by commas. The format of each entry is `<mask>-<compare>-<netInt>-<nextHop>`, where `<mask>` and `<compare>` are in the dotted decimal notation, `<netInt>` is a string ("ether" or "msnl" at present) and `<nextHop>` is either "d" for direct delivery of the address of the next hop gateway in dotted decimal notation.

For example,

```
do addenv("IPROUTE=255.255.240.0-128.232.0.0-ether-d,".
          "255.255.240.0-128.232.64.0-msnl-d,".
          "255.255.255.0-192.5.239.0-ether-128.232.0.2,".
          "0.0.0.0-0.0.0.0-ether-128.232.0.1");
```

Datagrams with destination of the form `128.232.<0-15>.<0-255>` will be delivered directly over the ethernet, `128.232.<64-79>.<0-255>` will be delivered directly over MSNL, `192.5.239.<0-255>` will be sent to the gateway `128.232.0.2` and anything else will be sent to the gateway `128.232.0.1`.

NB the order is important - if the last entry had appeared first then everything would be sent to `128.232.0.1`.

The test programs also use environment variables to find out their arguments - see below.

6 Test Routines

I have written 13 test programs that can be used to test the package. These programs were written with the view to testing the code between two Wanda machines or with a unix machine. The test programs that run on wanda can be found in the programs subdirectory while the corresponding files that run on the unix machine can be found in the internettest directory of the tools tree.

The test programs check parts of the implementation, from connection, sending ordinary data, sending urgent data, and clever things like doing "select" on more than one socket. They use certain environment variables which need to be set in the hostinfo file: `PEER-ADDR`, `PEER-PORT`, `MY-PORT`, `SEND-BYTES`, `RECV-BYTES` and `WAIT-TIME`. For each test only a subset of the environment variables are actually used but they all need to be present to run any of the programs.

7 Things still to do

```
Wanda kernel - rewrite WandaIPCSendAndKeep to be efficient instead of
               copying the IO buffer.

IP      - fragmentation (-> gatewaying);
          options processing;
          broadcast packets;
          loopback address.

ICMP    - user access.

TCP     - persistent probing of a zero send window;
          avoiding unnecessary ACK if we're going to transmit anyway.

UPL     - add an abort connection command;
          complete rewrite - MSRPC?

Sockets - add listen queue length limiting (needs UPL abort command).
```