Implementation of a lightweight transport service over HIPPI on the IBM RISC System/6000

Gudrun R. Dalgeir, Flavio Sticozzi
L3 Experiment
CERN
Geneva, Switzerland

e-mail: dalgeir@cernapo.cern.ch, sticozz@cernapo.cern.ch

December 3, 1993

1 Introduction

The L3 experiment at CERN has been accumulating data on the production and decay of Z^0 particles since 1989. The current volume of acquired and processed data is close to 20 TBytes, of which approximately 1 TByte are highly active data. Any or all of several hundred physicists may use the active data in studies which need both substantial CPU power and excellent access to data. In the next two years the volume of active data is expected to increase by a factor of two. The usefulness of the fastest RISC processors for L3 data analysis is largely dependent on the bandwith with which they can access the active data sample.

The L3 HiPPI project originated in 1990 as consequence of the perceived inadequacy of general purpose network technologies (Ethernet, FDDI) when faced with the future needs of L3. Although, even in 1990, FiberChannel was considered a more elegant technology than HiPPI, the fact that HiPPI equipment was already becoming available lead to a decision to undertake a project on HiPPI-based communications. HiPPI was considered as a technology for bulk data transport within a cluster of processors, to be supplemented by general purpose networking technology for activities such as remote login and electronic mail.

HIPPI is fully standardised at the hardware level, but at the time when L3 started this project little software support existed to provide process-to-process communication over HIPPI. So in order to get started L3 had to develop its own communication stack and application programmer's interface, API, to run over HIPPI. One alternative would have been to implement the Internet protocol stack, TCP/IP, over HIPPI. The L3 HIPPI network should, however, not be used as a general purpose network, and we therefore felt that we should try to avoid the complexity of TCP/IP. Instead L3 decided to develop a simple transport protocol and a socket interface compatible with the BSD stream sockets.

In March 1992 a joint project was set up between IBM and California Institute of Technology on behalf of the L3 collaboration at CERN to provide a complete software system to support the attachment of a HIPPI device to the IBM 3090 mainframe and to the IBM RISC System/6000 workstations running AIX. L3 was responsible for developing the software, and IBM provided the hardware. In May 1992 two RS6000 workstations and the HIPPI device for the 3090 mainframe arrived at CERN.

The rest of this document will describe the software specification, the design and implementation which have been done for the IBM RS6000. The last part of the document contains the performance test results.

2 Software specification

The main requirements on the L3 HIPPI software were to be compatible with the BSD sockets interface, and to get a throughput at the application level as close as possible to the physical limitations. The L3 HIPPI network should not be a general purpose network, for that we would use the CERN local area network.

The rest of this section will give a summary of the software specification, for more information see Appendix A.

2.1 Single copy

In a standard TCP/IP implementation all data which are transferred from one application to another is touched six times, once to copy the data from the application's memory buffer into an operating system buffer, mbuf, then TCP has to read all the data from memory to calculate the checksum, and finally the data are copied from the mbuf to the network adapter's transmit buffer. At the receiving end the copying operations take place again, in reversed order. Consequently there is a lot of overhead introduced by moving data around when the message buffers are large. The L3 HiPPI software, which is optimised for large data transfers, is specified to avoid the use of a buffer pool at the transport level. The user data are copied directly between the application's buffer and the HiPPI adapter buffers.

2.2 No checksums

Due to the high reliability of the HiPPI physical layer and in order to reduce transport protocol overhead, the L3 protocol over HiPPI does not include a checksum. In case of bit errors of such a nature that the HiPPI physical layer can detect them, the L3 HiPPI software will be informed. The only possible action to take in case of detected bit errors will be to terminate the connection.

2.3 No network layer functionality

The main functions of a network layer are fragmentation and packet routing. These functions are omitted in the L3 protocol software to run over HiPPI. The addressing within a L3 HiPPI network is handled by the HiPPI physical layer which uses the HiPPI Switch Control standard, [9]. Fragmentation is not needed since a transport connection only will cross one physicial network, namely the HiPPI network, which supports an unlimited packet length.

2.4 Flow control

Due to the lack of buffering at the transport level, incoming packets can get lost. If the receiving application has not called the read socket routine by the time a packet for this application arrives, the HiPPI transport layer has no buffer to store this packet temporarily, and it will therefore be discarded. Thus to guarantee that all packets are received, the transport protocol includes a simple flow control scheme. The flow control scheme has been specified as a protocol option. Whether to use the option or not is decided in the connection handshake by the transport level.

The flow control option applies only to one direction, so if needed both ways, both ends have to set the flow control flag in their respective connection setup packets. Immediately after the connection has been established either end can start to transmit data as fast as HiPPI READY permits. If flow control has been chosen by the sender, all outgoing packets should contain a sequence number field to tag the data transmitted. Before the socket write routine returns control to the application, the transport layer has to ask the destination for an acknowledgement. The acknowledgement packet will tell the source the sequence number of the last byte of contiguous data delivered to the destination application. In case of lost or incomplete packets the source will have to retransmit the data starting from the byte which sequence number equals the acknowledgement number. Since there is no buffering done at the transport level, the data packet containing the acknowledgement request can also get lost. The HiPPI read call can therefore not return without sending an acknowledgment packet in case the flow control option is on. Similarly, HiPPI write cannot return before the complete buffer it is supposed to send has been acknowledged.

Acknowledgement packets do not carry user data, and they can therefore not get lost due to lack of an available application receive buffer.

2.5 Large packets, large windows

The maximum packet size and window size used by the L3 HiPPI transport protocol can be set when starting up the protocol software. In reality there is no upper limit defined for these two protocol parameters. Typically, if one has a network set up with several clients contending for access to a data server, one should configure the software to run with a relatively short packet and window size, lets say 64 KBytes, to guarantee fair share arbitration between the clients. While, if there is no contention for the use of a HiPPI link, one should optimize performance by using a large packet and window size, the packet size could be several megabytes. The window size tells how many bytes a sender can have outstanding, ie. how many bytes he can send before receiving an acknowledgement packet from the receiver. The window size should be set to a multiple of the packet size.

2.6 Addressing

In order to be compatible with the BSD Internet sockets the L3 HiPPI sockets use Internet address formats. Thus an end-system is identified by its 4-octet long Internet address, and a service is identified by a port number which can have a value between 1 and 64K. The port numbers are unique within a system. A transport connection is identified by the pair (local IP address, local port number), (remote IP address, remote port number).

The transport layer maps the Internet address pair into the corresponding HiPPI I-field (I-field, see [9]), to get from one system to another. Thus the Internet addresses are not part of the transport protocol header, they are just used by the socket interface to identify a host. The transport layer maintains a table mapping source and destination IP address pairs to HiPPI I-fields.

2.7 Multiple connections

The L3 HiPPI software is specified to support apparently simultaneous connections to multiple services at multiple machines.

An application process can have several open ports, and one can have several HiPPI applications running within one system. However, the BSD select call is not supported by the HiPPI socket library, so there is no way for an application to check several open sockets for read readyness. When an application calls the socket accept routine, the application will fall asleep till there is a connection established on the socket's corresponding port.

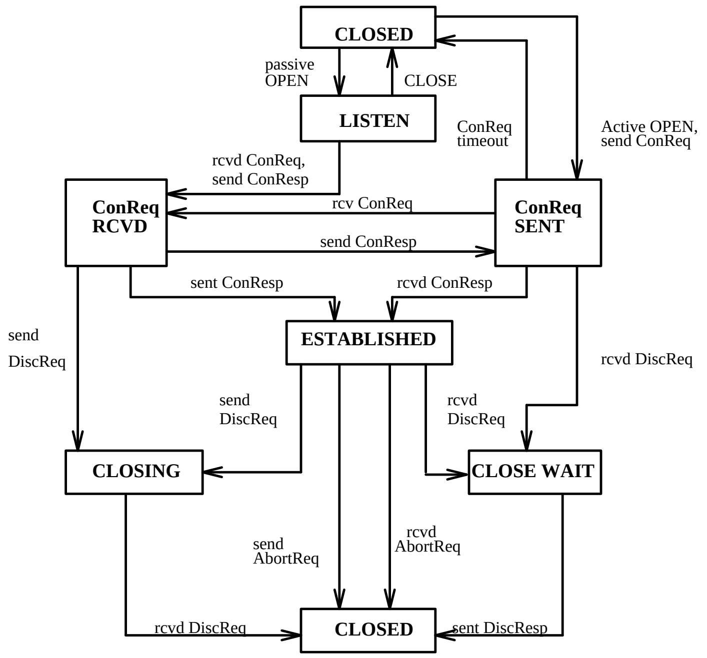

Figure 1: **Transport protocol state diagram**

An application can have several simultaneous open connections. The scheduling of connections within an application process should typically be implemented by using multi-processing techniques provided by the operating system, having one child process for each connection.

2.8 State diagram

Figure 1 shows the possible states and state transitions for a L3 HiPPI transport connection. The state diagram is a simplified version of the TCP state diagram as specified in [10].

2.9 Protocol header

As shown in Figure 2 the transport protocol header consists of the following fields:

Version A 8-bit version number (possible values 1 - 255). Two ends must have the same version number to communicate by use of the L3 HiPPI transport protocol.

Conn Connect flag. This bit is set in case of a connect request or connect respond packet.

Clos Close flag. This bit is set in case the user has requested the connection to be closed.

Abt Abort flag. This bit is set if either ends have a reason to abort the connection.

Type The type bit is used to distinguish between request and response packets. In case of a response packet the bit should be set, otherwise it should be off.

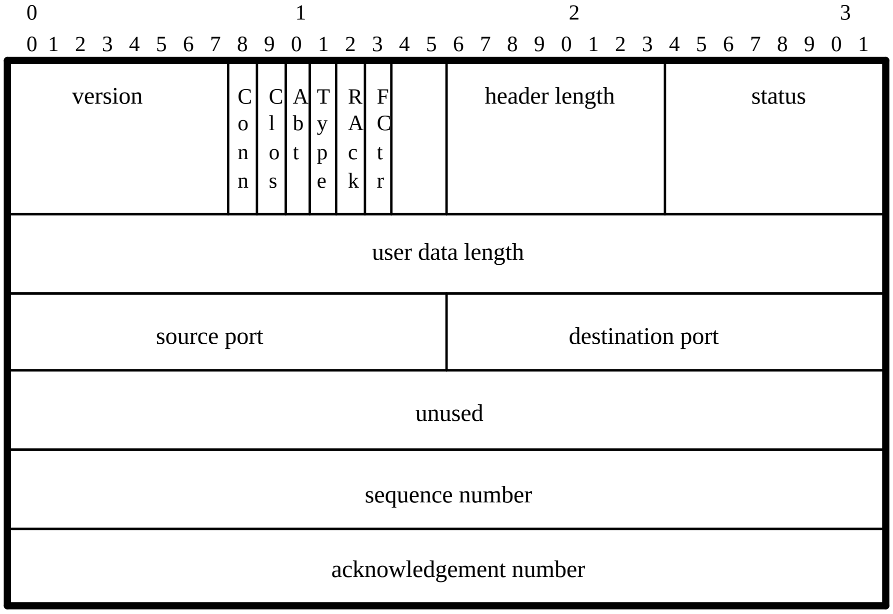

Figure 2: **Transport protocol header**

RAck Request Acknowledgement. The sender can set this bit to ask the receiver to acknowledge all bytes received contiguously.

FCtr Flow Control flag. If flow control is required this flag should be set in the Connect Request and/or Connect Response packets.

header length Because of the possibility of using the flow control option the length of the header can vary.

status The status field can be used to return error codes and other status information.

user data length Length of the user data in bytes, maximum length is 4.2 GBytes. The actual maximum packet length is implemented as a configuration parameter to the L3 HiPPI software.

source port This 16 bit number identifies the application service from which the user data originates. The port is bound to a local socket identifier.

destination port This 16 bit number together with the source port number identifies to which application at the destination host the user data should be delivered. The port is bound to a local socket identifier.

unused Keeps filling bytes.

The following header fields should only be present if an end has requested the flow control option during the connect handshake.

sequence number The sequence number of the first byte in this packet. The sequence number of the first data packet is the initial sequence number for this connection in the outgoing direction. The sequence number space is local to a transport connection.

acknowledgement The acknowledgement counter is one larger than the sequence number of the last byte delivered to the destination application, ie. the left window edge.

2.10 Socket interface

The L3 HiPPI socket library consists of the following routines which syntax is identical to the corresponding BSD socket routine.

h_socket Create a HiPPI stream socket.

h_bind Bind a name to a socket. A name is equivalent to an (IP address, port) pair.

h_connect Establish a transport connection over the HiPPI network.

h_listen Listen for connections on a socket.

h_accept Accept a connection on a socket.

h_write Send data over a transport connection from a socket.

h_read Read data from a socket.

h_send Equivialent to h_write, but communication options can be specified, ig. quality of service. These options are not supported by the HiPPI transport service.

h_recv Equivialent to h_read, but the non-supported communication options can be specified.

h_shutdown Initiate a gracefull shut down of a duplex transport connection.

h_close Discard a socket.

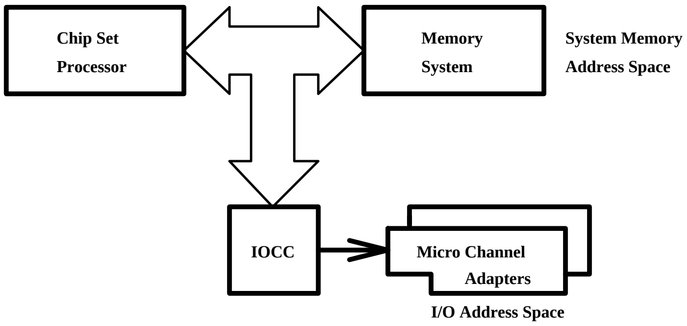

Figure 3: **System Block Diagram**

3 Description of the RS6000 HiPPI architecture

3.1 Microchannel organization

The Micro Channel architecture (Figure 3) provides a standardised hardware interface for adding I/O devices to an IBM computer. Implementation of this architecture provides a number of slots, which are electical connectors into which it is possible insert circuit cards (commonly called adapters).

The I/O Channel Controller (IOCC) is the interface between the bus interconnecting the CPUs and the Micro Channel bus. It is responsible for data buffering and protection.

Adapters can provide memory resources to the computer system but it cannot be used to provide main memory for the system processor. Memory resources provided by micro channel adapters can be mapped into virtual memory, but it resides in a different memory address space called bus memory address space, [3].

3.2 HiPPI adapter architecture

The IBM RS6000 HiPPI microchannel adapter enables the IBM RISC System/6000 to be attached to the standard HiPPI channel as defined by ANSI, [7].

The IBM HiPPI adapter supports:

- 32 bit wide data transfer at 100 MBytes/sec,
- HiPPI I-field, i.e. it can be connect to a HiPPI switch,
- the HiPPI Framing Protocol, [8].

The adapter subsystem conforms to IBM's Subsystem Control Block (SCB) architecture for communication between adapter and host. The HiPPI adapter is driven by its own device driver, the HiPPI device driver, which provides an SCB message interface to the HiPPI adapter for use by application device drivers implementing upper layer portocols. This HiPPI device driver interprets the SCBs coming from the application's device driver, and it is the only device driver having direct access to the HiPPI adapter, [4].

3.3 SCB architecture

Each SCB is made up of a header followed by a varying number of Parameter Descriptor Blocks (PDB) carrying information to or from the HiPPI device driver.

SCBs are passed between the HiPPI device driver and the application device driver using two queues, one incoming and one outgoing queue. The SCB header keeps information about the type of operation the application wants to issue. A valid operation is for example be to configure an entity, i.e. to declare an application device driver as known to the HiPPI adapter.

If the application would like to issue a read or a write operation, it must provide a buffer to the HiPPI adapter. The HiPPI adapter defines two possible ways to declare where the buffer is. The first method is to pass an SCB with no pointer to the data (inline data). Applying this approach, the buffer follows direcly after the PDB which tells that the buffer is passed inline, in the HiPPI device driver's incoming SCB queue. The same PDB gives the length of the buffer.

The second approach is to use an SCB with a pointer to the buffer. A PDB giving the buffer's address parameters is part of such an SCB. Once the application has build the SCB and passed it to the HiPPI device driver, the HiPPI adapter can begin to perform Direct Memeory Access operations on the buffer.

Whenever the HiPPI device driver has processed an SCB, the application device driver will be notified in terms of an interrupt.

3.4 HiPPI Adapter Address Space

The HiPPI adapter has its own memory. Before an application device driver can request a read or write operation the application must reserve some addresses in the HiPPI adapter's address space. The Micro channel addresses reserved must then be mapped into the the system's memory address space. This mapping is necessary to allow the HiPPI adapter to do Direct Memory Access (DMA) operations to and from of the system buffers.

The space reserved represents the total amount of Micro Channel address space that an application device driver can map to buffers at a given time. Obviously, the more address space an application reserves, the more transactions the application device driver can have outstanding at a given time. Although, this only represents address space rather than actual memory, it is a limited resource. An attempt to reserve too many addresses can fail, [4].

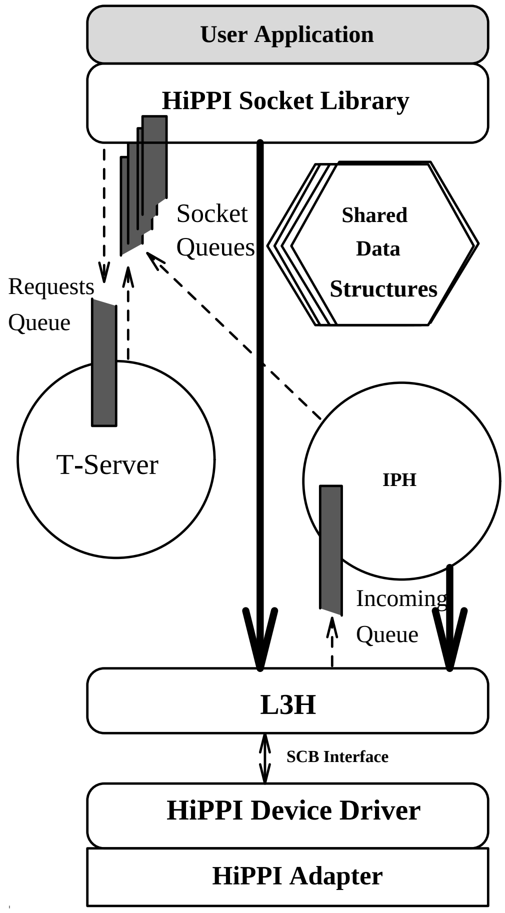

Figure 4: **L3 Processes Interaction**

4 Software Design

The L3 HiPPI software is made up of four components, a transport server (T-server), an Incoming Packet Handler (IPH), the HiPPI socket library and a HiPPI device driver (L3H), as shown in Figure 4.

The T-server, together with the IPH and the HiPPI library, implement the state diagram as shown in Figure 1.

Communication between the four components is achieved by using messages queues. The T-server reads requests from the library from an Inter Process Communications (IPC) queue which is known to all instances of the HiPPI library. For every new socket created there is also a new message queue created which gets associated with that socket. The queue is closed when the socket it serves get closed. These queues are used by the T-server and IPH to write messages back to the library instances. We call these queues a socket's input queue.

L3H uses a message queue to pass information to the IPH. Messages from L3H to the IPH come from the interrupt environment, and since calls to AIX queues cannot be issued in this environment, we implemented our own message queue at the kernel level.

Furthermore, the T-server, the IPH and the library share a memory area where globally known information about all sockets and connections present in a system are stored. This area is not protected by the use of system semaphores. The only field in the global data structure which can be modified by more than one process, is the state variable for a socket and its associated connection. The correctness of the state variable is, however, guaranteed by the fact that while the IPH is active updating a socket's state variable, the library instance managing this socket is sleeping waiting for the IPH to finish the processing of a packet which can lead to a state transition of the connection associated with the actual socket.

If the state is modified by the T-server it is because the library has requested some operation from the T-server. Consequently the library will sleep while it is waiting for the T-server to accomplish the requested task, so there is no risk that the T-server and library will try to modify the state variable at the same time.

None of the defined state transitions can take place at the same time for a single socket/connection pair, and therefore it is not necessary to access the shared memory area by the use of semaphores.

The rest of this section gives a description of each software component.

4.1 HiPPI Socket Library

The socket library is the application programmer's interface to the L3 HiPPI network. The library implements most of the transport level. All outgoing data packets and some control packets are sent directly from the library to the L3H device driver. The T-server is only contacted to get a socket identifier, to bind a name to a socket, to accept a connection and to close a socket. All incoming data packets go directly from the L3H device driver, through the IPH to the appropriate application process. In case of an incoming data packet the only function performed by the IPH is to find out to which socket a packet is going to be routed. L3H has no means of communicating directly back to the applications, so that is why the packets are propagated through the IPH to the library.

The following two subsections will describe the send and receive operations in more detail.

4.1.1 Send

Whenever the application calls h_write or h_send on a socket the library will call L3H to lock the user's buffer in memory, ie. to prevent the memory pages containing the user's buffer being swapped out of the physical memory. If there is not enough memory available to lock the amount of data going into a packet, h_write will simply return an error saying that the system is lacking buffer space. Following a successfull lock operation the library will build the packet header. If the user's buffer exceeds the maximum packet size several packets will have to be sent before the library can return the control to the application. Before sending a packet, the user data part of the packet must be locked in memory.

After the header has been built the library calls the L3H driver directly to send the packet. The packet header is passed as inline parameters while the buffer is passed by its address and length parameters. Thus the user data part of the packet is not copied. The library will then fall asleep till there is a WRITE COMPLETE message from L3H on this socket's input queue. This message contains the status of the write operation and the number of bytes written to the HiPPI link. If the operation failed, h_write will return an error to the application. Otherwise h_write will continue with sending packets till the last byte of the user's buffer has been sent. Before sending a new packet or returning to the application the library will call L3H to unlock the user data of the last packet sent from the memory.

If the outgoing flow control option is on, each data packet header will contain a sequence number. The library will set the header request acknowledgement flag when the amount of outstanding data equal the window size or whenever the packet it is about to send will contain the last byte of the user's buffer. After having requested an acknowledgement the library will wait for an ACKNOWLEDGEMENT message to arrive from the IPH on this socket's input queue. Before entering the wait state, the library will start a timer. If this timer expires before the ACKNWOLEDGEMENT messages arrives, the library will start to retransmit from the last acknowledged point of the outgoing stream. If the same retransmission is performed a certain number of times, h_write will finally give up and return an error to the application. However, if there is no timeout the library will compare the acknowledgement number received from its peer with the next sequence number it expects to send to see if any data have been lost. In the case of lost data, the library will start to retransmit from the point given by the acknowledgement number, and when it again reaches the end of the window or user's buffer the library will request an acknowledgement. The library will call L3H to unlock the memory buffer and return the h_write (or h_send) call when the complete user buffer has been acknowledged.

4.1.2 Receive

As for the send operation the library will call L3H to lock the user's receive buffer in physical memory. The library will then call L3H to establish this buffer as a HiPPI receive buffer, ie. a buffer which the HiPPI adapter can copy data into on the request from L3H. If the buffer is bigger than the maximum buffer size the library will only attempt to lock and establish a buffer of the predefined maximum buffer size. If either the lock or establish buffer operation fail h_read

(or h_recv) will simply return an error.

As packets arrive from the network the user data part of the packets will be put into the established HiPPI buffer by the HiPPI adapter on request from L3H. However, the library is informed by each incoming packet in terms of a message containing the packet header on this socket's input queue from the IPH. The library can then analyse the header to see how much data it has received and if necessary, check the flow control fields.

If the incoming flow control option is on the library will compare the sequence number of the incoming packet with the last acknowledgement number it has sent to see if this packet is the one it is expecting to receive. If the sequence number is correct, the library will advance its window by increasing the acknowledgement number by the length of the incoming user data field. Otherwise the acknowledgement number remains unchanged. In this case, when the packet expected has been lost and another packet has arrived, the data in the HiPPI receive buffer is out of sequence. h_read will therefore return with the number of bytes received pointing to the last point in the receive buffer which is in sequence. Before returning, the library will call L3H to release the HiPPI receive buffer, unlock the memory buffer and send an acknowledgement packet to the other end. It is then up to the application to call h_read (or h_recv) again to receive those packets which have been discarded.

If the incoming flow control option is on and if the incoming packet header contains a request for acknowledgement the library will create an acknowledgement packet and write it to L3H. The acknowledgement packet contains neither data nor a sequence number.

L3H will inform the library in terms of a READ COMPLETE message sent through the IPH to this socket's input queue when the receive buffer is full or nearly full. By nearly full it is meant that the remaining space in the buffer is not big enough to keep an incoming user data field of maximum length. Whenever the library receives a READ COMPLETE message it will call L3H to release the buffer, unlock the memory buffer and return the number of bytes received to the application.

To avoid blocking an application process for ever in case there is no data arriving on a transport connection, there is a timer associated with every receive buffer established at the L3H level. The timer is started by L3H every time a buffer is established and re-started after the arrival of a packet. If no packet has arrived before the timer expires L3H will send a READ COMPLETE message to the library through the IPH to this socket's input queue. Thus the library can then terminate the read operation and return the control to the application.

4.2 The Transport Server

The T-server manages local resourses like sockets and ports. When the T-server receives a request for a new socket it allocates a message queue, and it initialises a Transport Control Block (TCB) element in the shared memory. The new socket identifier is the primary key field to the TCB element. The message queue identifier needed to communicate from the T-server and the IPH to the new socket is kept in the TCB element.

Ports are a resource managed exlusively by the T-server. A port is bound to a socket before a connection is established (h_connect) or on demand (h_bind).

Several sockets can be bound to the same port if there are several connections to a single service port within a system. There is a new socket created for every connection accepted by the server application, but all the sockets remain bound to the same port. However, all sockets bound to a single port must be owned by a single process, or be member of the same process family. An application cannot request a port to be bound to a socket if the port is already in use.

4.3 Incoming Packet Handler

The IPH handles the processing of all incoming transport packets. Furthermore, it forwards the local write and read confirm messages from the HiPPI device driver to the correct socket input queue.

The IPH can receive the following transport protocol packets from the network:

- Connect Request: if possible, a connection is established. When a connect request arrives the IPH will check the status of the socket associated with the requested port. Three cases are possibles:
  1. Socket listens. The IPH will create and send a Connect Response packet, and the connection will be marked as ESTABLISHED.
  2. Socket listens, but no further connections can be accepted for this port. The IPH will send a Disconnect Request with the status flag set to 'Connection Refused'.
  3. Socket does not listen. The request is ignored, and the remote end will time out.
- Connect Response: IPH will mark the connection as ESTABLISHED and send a CONNECT CONFIRM message to the library instance which sent the corresponding Connect Request packet.

- Disconnect Request: If the state of the connection is ConReq SENT, then the IPH will create and send a Disconnect Response packet and mark the connection as CLOSED. Otherwise the IPH will mark the connection as CLOSE WAIT.
- Disconnect Response: IPH will mark the connection as CLOSED.
- Abort: The IPH will mark the connection as CLOSED.
- Acknowledgement: The IPH will forward the packet to the input queue of that socket which is associated with the connection as identified by the source and destination ports found in the packet header.
- Data: The packet header is forwarded to the input queue of that socket which is associated with the connection as identified by the source and destination ports found in the header.

4.4 The L3 HiPPI Device Driver

The main task of L3H is to allow the applications to send and receive data to and from the HiPPI adapter.

The HiPPI device driver assumes that each application driver has two major components, a receive and an unsolicit routine. These interrupt routines are called by the HiPPI device driver when it has to inform L3H about some event. Thus the L3H must operate in asyncronous mode.

The receive routine is called by the HiPPI device driver to inform L3H about the completion of an operation. This does not mean that the operation was successfull, but only that it has been attempted. The SCB returned from the HiPPI device driver contains an operation status field which has to be analysed.

The unsolicit routine is called by the HiPPI device driver whenever a HiPPI packet arrives from the network with the ULP-id (ULP-id, see [7]) set to identify L3H.

4.4.1 Device Driver Write Operation

When the library calls L3H to write a packet to the network, L3H will not copy the user data of the packet anywhere in its own address space. However, it will attach the buffer's address to its own address space (in the system memory), and mark it as unpageable to the paging manager. Then, L3H has to map the actual buffer's address into the HiPPI Microchannel address space. Once this mapping has been done, L3H will build an SCB that contains all the information about the destination and the buffer address, and pass the SCB to the HiPPI device driver's input queue. L3H does not wait for the confirm of the operation. At this point, L3H can return control to the HiPPI library from which it has been called. Later, L3H receives the result of the write operation from the HiPPI device driver which it propagates, through the IPH, to the calling socket's input queue.

4.4.2 Device Driver Read Operation

The read operation is more complex than the write operation because it consists of two steps:

1. the establishment of an application receive buffer, ie. to tell the HiPPI device driver to where incoming data should be copied, and
2. the actual copying of the data into the application's receive buffer.

The library calls L3H to establish a buffer. In order for L3H to know which buffer to use for an incoming packet, every buffer established is associated with a descriptor. The parameters of the buffer descriptor are identical to the address parameters of an incoming packet, ie. the I-field and the source and destination port numbers. As already explained in the section describing the library, L3H also associates a timer with every buffer established. The timer is started after the buffer has been established, and re-started after the arrival of a packet to this buffer. The timer is droped when the buffer is released.

When a HiPPI packet with the ULP-id set to indentify the L3H entity arrives, the HiPPI device driver calls L3H's unsolicit routine. This routine identifies two different types of SCBs, namely with or without pointer to data. SCBs without a pointer to data contain transport packets without any user data. So in case of a control packet, the complete packet is passed in the SCB coming from the HiPPI device driver. All transport packet headers are forwarded directly from L3H to the IPH.

If a pointer to data are present L3H will extract the address parameters from the incoming packet header, and then it will search through its table of buffer descriptors to see if there is a buffer established matching the address parameters of the packet. If a buffer exists L3H will request the HiPPI device driver to copy the user data part of the transport packet to this buffer. The user data are always kept in the D2 area (D2 area, see [7]) of the HiPPI frame while the header is kept in the D1 area. It is a feature of the RS6000 HiPPI adapter we use that the D1 area can be copied inline to the application device driver before waiting for the arrival of all the D2 area bursts of a HiPPI packet.

If L3H does not find any buffer established with address parameters equal those of the incoming packet, L3H will remove the packet from the adapter by requesting a copy to a dummy buffer.

L3H implements the following policy in order to decide when to return a receive buffer to the application.

1. Whenever a packet has been written to the buffer L3H will check if the remaining space in the buffer is big enough to receive a user data field of maximum length. If it is, L3H re-starts the timer and waits for the next event. Otherwise, L3H will mark the buffer as written and send a READ COMPLETE message through the IPH to the socket which has established the buffer. The timer will be stopped.

2. If a timer expires L3H will mark the associated buffer as written and return a READ COMPLETE message through the IPH to the socket which has established the buffer. The timer will not be re-started.

When the library calls L3H to release a buffer L3H will delete the buffer descriptor and timer.

4.4.3 Buffer alignement

If the application send and receive buffers are word aligned they can just be translated into Micro channel addresses. Otherwise, the HiPPI library will create a word aligned buffer. In case of write the library will copy the user's buffer into the aligned buffer. For read the library will establish the aligned buffer to receive the data, and before returning copy the data into the user's buffer. Consequently, to avoid this overhead the user applications should always call the HiPPI library with word aligned buffers.

5 Performance results

To test the performance of the L3 HiPPI software we ran memory-to-memory transfers between a Risc System/6000 model 550 and a Risc System/6000 model 590. The two systems were interconnected through an NSC PS32 HiPPI switch. Both hosts ran version 3.2.1 of the HiPPI software. The 550 ran with the AIX version 3.2.4, while the 590 ran with AIX version 3.2.5. The maximum speed of the IOCC chip controlling the microchannel of the 550 we had for the test was 16 MBytes/sec, while the microchannel of the model 590 supports 80 MBytes/sec. Consequently our end-to-end test could not go beyond 16 MBytes/sec. IBM has tested their TCP/IP implementation running under AIX 3.2.4 between two 550s over the same HiPPI version as we had available achieving 9.0 MBytes/sec. However, with the latest AIX version, 3.2.5, which implements the TCP/IP options for high bandwidth networks (RFC1323), and with the HiPPI version 3.2.2, IBM has achieved 38 MBytes/sec memory-to-memory between two 590s.

The author of the application program we used to measure the performance is Leslie Roberston, working in the Computing and Networking Divison at CERN. The application client simply generates a buffer of a given size decided by the user, opens a stream socket, and then it enters

Packet size	Buffer size	Throughput	CPU usage (client)	CPU usage (server)
64 KBytes	64 KBytes	5.5 - 6.8 MBytes/sec	38%	13%
64 KBytes	32 KBytes	3.0 - 4.0 MBytes/sec	41%	14%
64 KBytes	128 KBytes	5.5 - 7.0 MBytes/sec	30%	9%
2 MBytes	2 MBytes	12.0 - 13.8 MBytes/sec	46%	1%
2 MBytes	1 MBytes	13.3 MBytes/sec	12%	4%
2 MBytes	4 MBytes	12.1 - 14.0 MBytes/sec	65%	1%
4 MBytes	4 MBytes	13.0 - 14.0 MBytes/sec	70%	3%
4 MBytes	2 MBytes	13.8 MBytes/sec	46%	3%

Table 1: **End-to-end throughput and CPU utilisation for 550 <-> 590**

an indefinite loop. Within the loop the client writes the buffer to the socket. The application calls h_read and h_write with word aligned buffers. After every buffer written the client calls a routine which calculates the time it took to transfer this buffer and how much CPU time was spent by the application program linked to the HiPPI library, both executing in user mode and in system mode. This information is retrieved by using the Unix routine getrusage. The CPU usage it took to transfer one buffer is calculated as the percentage of the elapsed time between two calls which equals the total CPU time spent (total CPU time = user mode time + system mode time). The application server simply reads the buffer from its stream socket, and after every read operation it calls the same routine as the client to get the transfer speed of the last buffer received and the CPU time it took to receive this buffer. The program updates the average transfer speed after every buffer sent/received.

The T-server does not take any of the CPU when the connection is in the ESTABLISHED state, ie. when the data transfer takes place. To measure the CPU time taken by the IPH we used the resource monitor running under AIX. When using a packet size of 64 KBytes the IPH consumes about 8% of the CPU, while for a large packet size of 2 MBytes the IPH takes only 1.5% of the CPU on the 550 and not more than 0.2% on the 590 running the receiver. The highest CPU consumption we saw by the IPH when running with small packets of 32 KBytes on the model 550 was 12%.

5.1 End-to-end performance results

Table 1 lists the results of the end-to-end tests between the 550 and the 590. For all sessions the window size was equal the socket buffer size. The applciation client always ran on the 550 and the application server on the 590. The CPU usage is the avarage CPU usage for all the samples of a session. A sample is defined as the transfer of one application buffer. What can be

Packet size	Buffer size	Throughput	CPU usage (client)	CPU usage (server)
64 KBytes	20 KBytes	5.5 MBytes/sec	29%	13%
64 KBytes	64 KBytes	5.4 MBytes/sec	32%	11%
64 KBytes	2 MBytes	5 MBytes/sec	27%	10%

Table 2: **End-to-end TCP/IP throughput and CPU usage for 550 <-> 590**

seen is that the client always consumes much more of the CPU than the server. However, the client ran in the least powerful machine.

When using a relativly short packet size of 64 KBytes (+ the number of bytes taken by the transport packet header), the peak throughput is at 7 MBytes/sec, which is 44% of the physical limitation defined by the IOCC of the 550. What can be seen is that the throughput does not increase much by doubling the length of the application buffer, but the CPU time spent goes down. The effect of doubling the buffer size is that the server can receive two packets before returning the h_read, and the client can send two packets before waiting for an acknowledgement.

The maximum throughput of 14 MBytes/sec, which is 87% of the possible microchannel speed, is achieved by running with a packet size of 2 MBytes and a buffer and window size of 4 MBytes. The same peak, but with less variations between the samples, is also achieved by running with a packet size of 4 MBytes. In case of a packet size of 2 MBytes, there is a big variation between the sessions in the CPU time spent by the client on the model 550. We don't have any explanation for this, but one possibility could be that the adapter has 1 Megabyte of physical buffer space, and a socket buffer of the same size could give the optimal result in terms of CPU usage. However, whenever we get a test setup available again, we would like to spend some more time to investigate this result.

We ran the same application program between the model 550 and the 590 to test the TCP/IP performance over HiPPI. Since the AIX version 3.2.4 does not support RFC1323 the maximum window size was limited to 64 KBytes. The maximum IP packet size used over HiPPI is also 64 KBytes. We sat the socket send and receive buffer space to 64 KBytes. With this setup the maximum throughput we got was 5.5 MBytes/sec, see Table 2.

5.2 RS6000 model 590 in loopback

When connecting the HiPPI source to the HiPPI destination of the model 590, we got a very unstable result varying from 2.7 MBytes/sec to 11 MBytes/sec using a packet and buffer size of 64 KBytes, see Table 3. This could be due to the fact that both the application client, server and the IPH have to be scheduled once as the active process for every packet transfered,

Packet size	Buffer size	Throughput	CPU usage (client)	CPU usage (server)
64 KBytes	64 KBytes	2.7 - 11 MBytes/sec	12%	20%
2 MBytes	2 MBytes	27.0 MBytes/sec	8%	5%
2 MBytes	4 MBytes	28.2 MBytes/sec	10%	8%

Table 3: **Model 590 in loopback, throughput and CPU utilisation**

and since the packets were relativly short, the system must have been quite busy doing context switches. When increasing the packet size to 2 MBytes we got 27 MBytes/sec writing buffers of 2 MBytes and 28.2 MBytes/sec when writing buffers of 4 MBytes. What the software could have achieved in terms of throughput if it could have been run between two 590s will only be guesswork, but when running the model 550 in loopback with 2 MBytes packets, we got half of the maximum performance achieved end-to-end. Thus if the same would turn out to be the case for the 590 end-to-end we would have got 56 MBytes/sec, which is 70% of the microchannel speed. In loopback the IPH takes around 2% of the 590's CPU.

By tuning the TCP/IP HiPPI parameters avaialable under AIX 3.2.5 we got 19 MBytes/sec when running our application program over TCP/IP in loopback over HiPPI on the model 590. Both the client and server reported to take almost 50% of the CPU. This result is consistent with IBM's result of 38 MBytes/sec between two 590s.

6 Summary

A simple transport protocol has been developed to run on the IBM Risc System/6000 under AIX over switched HiPPI. The programmer's interface to the transport service is compatible with the standard BSD Internet sockets. The HiPPI socket library implements the blocking version of socket, bind, connect, listen, accept, shutdown, close, send/write and recv/read.

The L3 HiPPI device driver runs in kernel space, while the library, T-server and IPH runs in user space. In theory further performance improvements could have been expected by implementing the transport layer in kernel space.

The maximum data transfer rate achieved by this implementation is 14 MBytes/sec which is 87% of the physical limitations of the test setup. In comparison, IBM has tested TCP/IP over the same HiPPI version between two RS6000s model 550 achieving 9 MBytes/sec. By using our test setup we got a maximum transfer rate of 5.5 MBytes/sec over TCP/IP. The TCP/IP protocol stack includes many functions which we have omitted, but on the other hand TCP/IP runs completly in kernel space and is therefore not losing transfer speed due to context switches.

In general purpose networks most packets are small, not more than a few 100s of bytes, and in that context our comparison with TCP/IP when running the L3 HiPPI software with a packet size of 2MB might seem unreasonable. However, the purpose of this project was to make a highway for transfer of large data sets, and the test results clearly show that for this purpose our solution is better than the available TCP/IP implementation over HiPPI.

The result achieved in loopback on the model 590 indicates that two 590s running the L3 HiPPI software end-to-end possibly could have transferred data at around 56 MBytes/sec. Again, this is far better than TCP/IP which has been tested by IBM to run at 38 MBytes/sec between two 590s using the HiPPI version 3.2.2. We confirmed this data rate over TCP/IP by getting 19 MBytes/sec when running the model 590 in loopback.

7 Acknowledgements

We are very grateful to Tim Bell, IBM Geneva for his help on AIX issues, his interactions with IBM and his efforts which in the end let us have the model 590 to run the end-to-end tests. We would also very much like to thank Craig Stoops at IBM, Austin by whom we have communicated mainly through electronic mail, for helping to sort out technical difficulties we have had while developing the device driver. Finally we would like to thank the many people at California Institute of Technology and within IBM who made this project possible, including Richard Mount, California Institute of Technology, Tito Sorlini, IBM Geneva, Karen Woolhouse, Henry Brandt and George Kremls, all from IBM Kingston.

A proposal to the L3 HIPPI Network Software Architecture

Gudrun R. Dalgeir
L3 Experiment
PPE Division CERN
1211 Geneva 23, Switzerland

March 11, 1992

A.1 Introduction

The L3 experiment plans to set up a distributed computing environment to be used for off-line analysis centred around an IBM 3090 mainframe, a VAX and Unix workstations based on Gigabit networking technology. The individual systems in this environment will be interconnected by switched HIPPI. The HIPPI network will only be used by data intensive applications. For other distributed applications like electronic mail, remote login, PAW, small file transfers and so forth the CERN local area network should carry the traffic. Hence all hosts on the L3 HIPPI network should have at least one other communication interface to either ethernet, the Apollo Domain ring or FDDI.

Distributed applications which should be supported in this environment are ZFTP, P3, Fatmen and other L3 software. The existing versions of these applications use the CERN developed TCPAW package, a simple C interface callable from FORTRAN to the BSD4.3 Internet sockets, to implement process-to-process communication over TCP/IP. Thus in order to minimise the required re-writing of these applications to make them run over HIPPI, it has been proposed to provide an application programming interface (API) to HIPPI which is compatible with the BSD4.3 Internet stream sockets, see [2]. Since HIPPI only will be used for large data transfers there will be no need for a datagram service. Thus datagram sockets will be redundant.

HIPPI is fully standardised at the hardware level, but little software support exists to provide process-to-process communication over HIPPI. Thus in order to implement the HIPPI sockets interface an end-to-end protocol which guarantees ordered, error-free delivery should be implemented. One alternative would be to run the Internet protocol stack, TCP/IP, over HIPPI. Since the L3 HIPPI network will not be used as a general purpose network and due to the complexity and overhead of TCP/IP, L3 will develop a light-weight transport protocol to be used over the HIPPI network. However, whenever products providing process-to-process communication over HIPPI giving the desired performance become available on the L3 HW platforms, we should be ready to replace our solution with something more standard. By chosing HIPPI for the physical layer and an API compatible with the BSD sockets, the goal should be to only have to re-link the applications in case of installation of new networking software.

A.2 HIPPI software requirements

To communicate with a non-HIPPI host existing communication software running over the CERN LAN should be used. This implies that the HIPPI transport protocol does not have to be compatible with any other transport protocol. Neither will we need a HIPPI/FDDI/ethernet gateway nor a network layer which basic functions are fragmentation and addressing. The addressing and routing will be handled by HIPPI-SC. Neither will fragmentation be needed since the transport protocol will only run over HIPPI links which all support an MTU equal a HIPPI packet of unlimited length. Taken these considerations into account the software requirements are the following:

- The transport protocol should give the application an end-to-end connection providing ordered delivery.
- The transport protocol should be designed to run on top of the HIPPI Framing Protocol (HIPPI-FP) which again should run over full duplex, switched HIPPI.
- It is up to the application to provide presentation and session services.
- The first version of the transport layer should have support for apparently simultaneous transport connections with multiple processes at multiple HIPPI hosts by using switched HIPPI, [1].
- The API should be compatible with the Internet stream sockets. The first version of HIPPI software will only support blocking sockets.

- To keep the HIPPI sockets compatible with the Internet sockets, the application should identify the destination by its Internet address. Thus the HIPPI sockets should support the Internet address family. The transport layer will provide the mapping between the Internet and HIPPI-SC addresses.
- The transport protocol software should be easy to port from one operating system (OS) to another.
- For the L3 computing environment the protocol performance is considered as more important than compatibility with existing protocol stacks. The transport protocol should therefore benefit as much as possible from the high quality of the underlying HIPPI links to reduce network overhead time in the transport layer.

A.3 Transport protocol issues

When specifying the HIPPI transport protocol the following problems must be taken into account:

- How to establish a transport connection over HIPPI. At any time the HIPPI host can only have one active HIPPI connection, but the transport level must be able to support several apparently simultaneous transport connections. The HIPPI transport software must guarantee fair share arbitration of the physical network.
- How to terminate and abort a transport connection over HIPPI.
- How to provide end-to-end flow control.
- How to recover from transmission errors.
- In order to provide process-to-process communication over HIPPI there must be support in the transport protocol for identifying a process on a remote HIPPI host.

A.4 HIPPI host software architecture

A HIPPI application will be linked together with the HIPPI sockets library. The library could either be the standard sockets library coming with the OS having support for a new HIPPI protocol family or a separate HIPPI sockets library. The sockets will act as a programmer's interface to the HIPPI transport service, (T)-service. The (T)-service will implement the HIPPI transport protocol which will handle the data delivery between two endsystems. If possible the (T)-service could be an integrated part of the OS, but it can also be implemented as an application communicating directly with the HIPPI driver. Some of the transport service functionality can also be part of the library. The Framing Protocol (FP) might have to be implemented as a system application, or it might reside in the HIPPI adapter provided by the vendor, see Figure 5. In the HIPPI adapters there will be transmit buffers (TB) and receive buffers (RB). Whether the transport service will need a buffer pool or not is left for the design study. It would, however, be desirable if the transport service could copy the data directly between the application's memory space and the adapter buffers to minimise the copying overhead, but again this depends on the system architecture of the host machine.

```
      System A                                  System B
---------------------                      --------------------
|                   |                      |                   |
| +---------------+ |                      | +---------------+ |
| |  Application  |<|----Appl. protocol----|>|  Application  | |
| +---------------+ |                      | +---------------+ |
| | HIPPI sockets | |                      | | HIPPI sockets | |
| +---------------+ |                      | +---------------+ |
| |HIPPI Transport|<|--Transport protocol--|>|HIPPI Transport| |
| |     Server    | |                      | |     Server    | |
| |               | |                      | |               | |
| +---------------+ |                      | +---------------+ |
| |   HIPPI-FP    |<|---Framing protocol+  | |  HIPPI Driver | |
| +---------------+ |                   |  | +---------------+ |
| |  HIPPI-Driver | |                   |  |         |         |
| +---------------+ |                   |  | ===system bus==== |
|         |         |                   |  |         |         |
| ===system bus==== |                   |  |  +-------------+  |
|         |         |                   |  |  |HIPPI Adapter|  |
|  +-------------+  |                   |  |  |             |  |
|  |HIPPI Adapter|  |                   +--|->|   HIPPI-FP  |  |
|  |             |  |                      |  |             |  |
|  |  HIPPI-PH   |<-|---HIPPI PH protocol--|->|   HIPPI-PH  |  |
|  |             |  |                      |  |             |  |
|  +-------------+  |                      |  +-------------+  |
|________||_________|                      |________||_________|
         ||                                         ||
          \\                                       //
            ============= HIPPI FABRIC ============
```

Figure 5: **HIPPI host architecture**

A.4.1 HIPPI sockets

The HIPPI API should be compatible with the BSD Internet sockets interface. The standard TCP way of identifying a connection, ie. by the pairs (local IP address, local port) and (remote IP address, remote port) will therefore be adopted by the HIPPI transport service. Similar to the Internet sockets the application servers will listen for connect requests on well known ports. It should be the responsibility of the transport service to assign unique port numbers within a host. Since the Internet address family will be used, the application developers can view the HIPPI network as an isolated internet.

The transport service will use socket identifiers to identify the local communication which has to take place between itself and an application. The first call an application will perform to establish this communication is HIPPI_socket. The transport service should allocate a unique socket identifier and return it to the application. Then, the application could either request the transport servcie to bind a name, ie. an (IP address,port) pair, to the socket, this will typically be done by the application server end, or to issue a connect call. In the latter case, if the socket is still unbound, the transport service should allocate a unique port number and bind it to the socket before it tries to establish the HIPPI transport connection.

The HIPPI software should have support for several interleaved connections to a single or several hosts. This makes it possible for an application server to provide support for several simultaneous clients, each through an open transport connection. Due to the fact that the existing socket applications used by L3 are invoked by the inetd or something equivalent on non Unix systems, the HIPPI software architecture could adopt the inetd mechanism. This will also simplify the connection establishment for the application developers and the multiplexing issue will be taken care of by the HIPPI inetd. The HIPPI inetd will open a socket and bind it to the appropriate port on behalf of each HIPPI application service. Then it will use the HIPPI_select call to check each socket for read availability, it will accept a call for a ready socket, and create a process which will execute the appropriate application server. See [13] for further details about how the inetd works. Alternatively the application servers can take care of the multiple connections by using multi-threading techniques. Under Unix an application server process can create a subprocess to serve a single connection.

The following synchronous socket calls should be supported:

HIPPI_socket Create a HIPPI stream socket. The transport service will return a socket identifier which is unique within this system. The socket is used to identify the local process-to-process communication between the application and the transport service.

HIPPI_bind Bind a name to a socket. A name is equivalent to a (host IP address, port) pair. The port number can be requested or allocated by the HIPPI transport service. The transport service will guarantee that the port number is unique for this host. If the HIPPI transport service is implemented as an application process, the HIPPI port numbers and the Internet port numbers will constitute two different port number domains.

HIPPI_getsocketname Get the name of a socket.

HIPPI_connect Should be used by a client to establish a connection over the HIPPI network. If the client's socket is unbound the HIPPI transport service must select and bind a name to it.

HIPPI_listen Listen for incoming connect requests.

HIPPI_accept Should be used by a server to accept a connection. The HIPPI (T)-service should return a new socket identifier to the application server to identify this connection.

HIPPI_write Send data to a socket.

HIPPI_read Receive data from a socket.

HIPPI_send Similar to write, but communication options can be specified, ig. quality of service. These options will not be supported in the HIPPI transport protocol. The reason for supporting this call is to be compatible with the BSD sockets interface.

HIPPI_recv Similar to read, but the non-supported communication options can be specified in the call to be compatible with the BSD sockets interface.

HIPPI_shutdown The HIPPI transport connection will be closed gracefully, so before deleting it all outstanding data will be delivered. The socket will be maintained, and the user can continue to read till the other end also shuts down the connection.

HIPPI_close Discard the socket, and in case of an open transport connection it will be aborted. Outstanding data can be lost.

HIPPI_select Should be used by an application server to multiplex I/O requests among multiple HIPPI sockets.

In addition there should be routines for looking up host addresses, protocol identifiers and services. A service is defined by port number and protocol identifier.

The complete specification of each routine with it's parameters should be the same as for the corresponding Internet stream sockets implemented under the various operating systems connecting to L3's HIPPI network.

A.4.2 HIPPI transport protocol

A.4.2.1 Transmission errors

According to [8] HIPPI-PH provides an odd parity bit for each byte. Thus an undetected error will require a 2-bit error in the same byte. The HIPPI-PH Length-Longitudinal Redundancy Check (LLRC) implements even parity across the individual bits of multiple words in a burst. Thus the LLRC will not detect errors where the same bit in an even number of words was incorrect. In case of corrupted packets recognized by HIPPI-PH, the HIPPI driver will simply inform the upper layer since a simplex HIPPI-PH layer has no means of retransmitting bursts. Neither can one be completely sure that the HIPPI-PH layer will detect all bit errors. However, due to the quality of the transmission media bit errors should be extremely rare. Besides if the HIPPI link starts to fail it would be reasonable to assume that the first errors will be of such a nature that the HIPPI-PH entities would detect them and inform the user. So taken the high quality of the media into account and at the same time trying to minimise the transport overhead and complexity, the first version of the transport protocol will be designed without a checksum and error retransmission mechanism. In case of a detected error the transport layer should merely restrict itself to notify the application.

A.4.2.2 Flow control

The protocol will not provide checksums, but if packets can be lost due to overflow acknowledgement and retransmission mechanisms are still needed. There is flow control implemented between two directly interconnected points in the HIPPI-PH protocol, see [7]. If the HIPPI receiver is slower than the sender and there is a switch in-between, the switch will not provide any buffering, and the switch is transparent to the endsystems, [9]. This means that the READY pulses generated by the destination every time it is ready to receive bursts will flow through the switch and be picked up by the source. Thus there is end-to-end flow control implemented between the HIPPI-PH entities.

Assuming there is no flow control implemented in the transport protocol, the receiving (T)-service should not be able to read data from the HIPPI adapter buffers before it has available buffer space, either in its own memory or in the application's address space. In theory this should slow down the HIPPI-PH sender which will not receive new READY pulses before the destination HIPPI-PH entity has empty buffers. However, the HIPPI adapter might discard the data if the receiving (T)-service fails in copying the data elsewhere whenever it is interrupted by the adapter for a read operation, and in that case flow control at the transport level will be mandatory. Besides one can expect the buffer space available in the adapter to be rather limited, so the adapter's memory should not be used as a (T)-level buffer pool on behalf of multiple transport connections. Incoming packets therefore have to be buffered, either in a (T)-level buffer pool or directly into the application's receive buffer. If the latter solution is chosen, packets will be lost if they arrive before the receiver has been scheduled to make its receive buffer available to the HIPPI system. This paper therefore proposes an optional flow control scheme in the transport protocol.

The flow control option applies only to one direction. Immediately after the connection has been established either ends can start to transmit data as fast as HIPPI READY permits. In case flow control has been chosen by the sender, all packets should contain a sequence number field to tag the data transmitted. Before the (T)-service can accept further input data from the user the (T)-service has to ask the destination for an acknowledgement. The acknowledgement packet will tell the source the sequence number of the last byte of contiguous data delivered to the destination application. In case of a lost or an incomplete packet the source will have to retransmit the data starting from the byte which sequence number equals the acknowledgement number. If there is no buffering done at the transport level the packet containing the acknowledgement request can be lost. The HIPPI_read call can therefore not return without sending an acknowledgment packet in case the flow control option is on. Similarly, HIPPI_write cannot return before the complete buffer it is supposed to send has been acknowledged.

A.4.2.3 HIPPI transport protocol header

The proposed HIPPI transport header consists of the following fields:

Version A 8-bit version number (possible values 1 - 255). Two ends must have the same version number to communicate by use of the HIPPI transport protocol.

Conn Connect flag. This bit is set in case of a connect request or connect respond packet.

Clos Close flag. This bit is set in case the user has requested the connection to be closed.

Abt Abort flag. This bit is set if either ends have a reason to abort the connection.

Type The type bit is used to distinguish between request and response packets. In case of a response packet the bit should be set, otherwise it should be off.

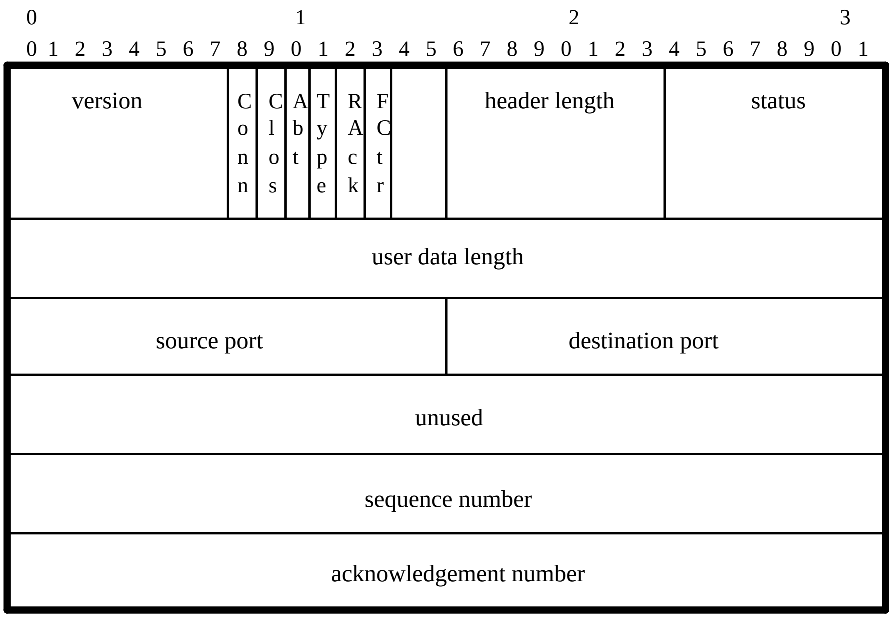

Figure 6: **HIPPI transport protocol header**

RAck Request Acknowledgement. The sender can set this bit to ask the receiver to acknowledge all bytes received contiguously.

FCtr Flow Control flag. If flow control is required this flag should be set in the Connect Request and/or Connect Response packets.

user data length Length of the user data in bytes, maximum length is 4.2 GBytes. The actual maximum packet length should be implemented as a configuration parameter to the HIPPI software.

header length Because of the possibility of using protocol options the length of the header can vary.

status The status field can be used to return error codes and other status information.

source port This 16 bit number identifies the application service from which the user data originates. The port is bound to a local socket identifier.

destination port This 16 bit number together with the source port number identifies to which application at the destination host the user data should be delivered. The port is bound to a local socket identifier.

unused Should keep filling bytes.

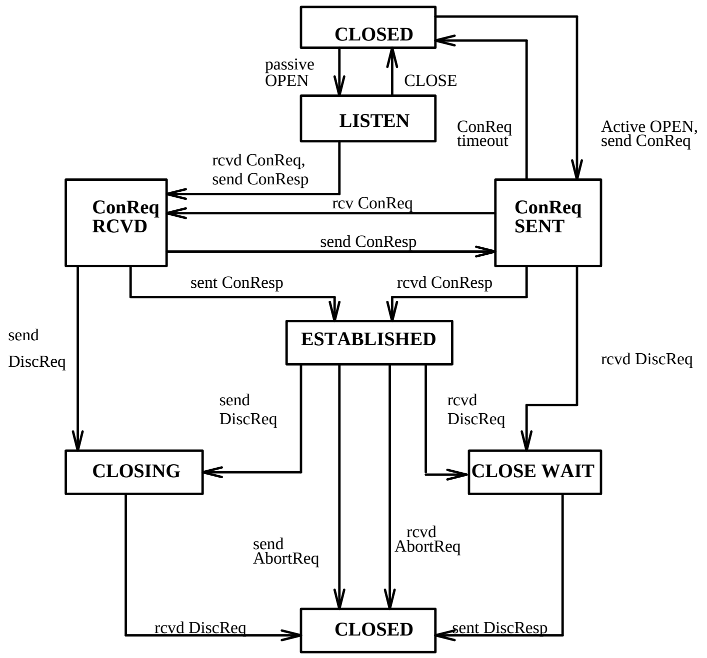

Figure 7: **State diagram for the HIPPI transport protocol.**

The following header fields should only be present if an end has requested the flow control option during the connect handshake.

sequence number The sequence number of the first byte in this packet. The sequence number of the first data packet is the initial sequence number for this connection in the outgoing direction. The sequence number space is local to a transport connection.

acknowledgement The acknowledgement counter is one larger than the sequence number of the last byte delivered to the destination application, ie. the left window edge.

The source and destination addresses will be kept in the I-field of the first HIPPI-PH burst so the transport protocol does not have to carry other address information than the port numbers.

A.4.2.4 Connect

When the application client calls the (T)-service through the HIPPI_connect call the (T)-server will

1. If the client's socket is unbound, assign a port number to it.

2. The (T)-service will send a transport request packet with the connect flag set and the type bit off to indicate that this is a request packet. If outgoing flow control is needed the flow control bit should be set. The sender enters the CONREQ SENT state.

```
Application |     Transport Service      | Application
client      |                            | server
            |                            |
            |                            |<- h_listen
h_connect ->|                  h_listen->|
            |connect.req->               |
            |               ->connect.ind|
            |              <-connect.resp|<- h_accept
            |                  h_accept->|
            |connect.conf<-              |
            |<-h_connect                 |
            |                            |
            |       Data Transfer        |
```

Figure 8: **Connect time sequence diagram**

3. The destination (T)-service should ignore the connect request if there is no application listening to the destination port. The other end will eventually time out and return to the CLOSED state.

4. If there is an application listening to the requested port, the destination (T)-service should put the connect request in this port's backlog queue, and mark the state of the queued connection as CONREQ RECV. The destination application will be woken up from the HIPPI_accept call whenever there is a connect request in its backlog queue.

5. If the connect request was put in the destination's backlog queue, the destination (T)-service should send a response packet with the connect flag set, the status code to OK, and the type bit set to indicate that this is a response packet. If outgoing data from the destination must be acknowledged, the flow control bit in the header must be set.

6. The sending (T)-service will receive the connect response packet. If the connect request completed successfully data transfer can begin, and both ends enter the ESTABLISHED state.

7. If the connect response does not arrive within a certain time limit, the sender should return a timeout status to the application client, and the sender returns to the CLOSED state.

Due to the lack of mandatory sequence- and acknowledgement numbers in the proposed transport header the type bit is used to distinguish between connect request and connect response

```
 Application |      Transport Service      | Application
 client      |                             | server
             |                             |
h_shutdown ->|                             |
             |<-h_shutdown                 |
             |disc.req->                   |
             |                  ->disc.ind|
   h_read  ->|                             |<- h_send
             |                  <-send.req|
             |send.ind<-          h_send->|
             |<-h_read                     |<- h_shutdown
             |                h_shutdown->|
   h_read  ->|                 <-disc.resp|
             |disc.conf<-                  |
             |<-h_read                     |
             |                             |
```

Figure 9: **Example of a disconnect time sequence diagram**

packets.

A.4.2.5 Disconnect

A transport connection has to be shut down by both ends. The meaning of shutdown is that "I have no more data to send". Before dropping the transport connection any data in the pipe should be delivered. Thus the user who requests the connection shutdown may continue to receive data until he is told that the other end also has shutdown. [10].

When either ends call the local (T)-service through the HIPPI_shutdown call, the (T)-service should

1. If any outgoing data is queued for this connection, transmit the user data.

2. Send a disconnect request packet with the close flag set, and the sender enters the CLOSING state.

3. When the destination (T)-service receives a disconnect request it should tell the local application that there is no more data to come in terms of a HIPPI_read return code. The destination enters the CLOSE WAIT state.

```
Application |      Transport Service      | Application
client      |                             | server
            |                             |
h_shutdown->|                             |<- h_shutdown
            |disc.req->       <-disc.req|
            |<-h_shutdown   h_shutdown->|
            |disc.ind<-       ->disc.ind|
            |disc.resp->     <-disc.resp|
            |disc.conf<-     ->disc.conf|
```

Figure 10: **Simultaneous close sequence diagram**

4. When the destination application has sent its last message, it should also call HIPPI_shutdown (or HIPPI_close).

5. The destination (T)-service will send a disconnect response packet with the close flag and type bit set, wake up the local application and release all resources associated with this connection. The destination returns to the CLOSED state.

6. If the source (T)-service receives the disconnect reply within a certain time it knows that all data has been delivered in both directions and that the transport connection can be deleted. The source can also enter the CLOSED state.

7. If the application has called HIPPI_read it should be waken up with the status code set to no more data. To discard the socket the application has to call HIPPI_close.

8. In case the disconnect response never arrives, the (T)-service should after a reasonable long timeout release all resources reserved for the given connection.

If both ends receive a disconnect request packet while having an outstanding disconnect request on the same connection, one has the scenario shown in Figure 10.

What can be seen from Figure 10 is that both ends should reply to the incoming disconnect request. If the application has called read after shutdown the (T)-service should return with the status set to no more data regardless of whether the disconnect response packet has arrived or not.

```
Application |     Transport service     | Application
client      |                           | server
            |                           |
  h_close ->|                           |
            |abort.req->                |<- h_send
            |<- h_close     ->abort.ind |
            |                  h_send ->|
```

Figure 11: **Abort time sequence diagram**

A.4.2.6 Abort

Both source and destination can abort a transport connection. Outstanding data in the pipe between the two applications can be lost. Neither HIPPI_read nor HIPPI_send can be accepted by the (T)-service after the application has requested a socket close. However, client-server applications using BSD sockets typically implements a session layer on top of TCP in which they negotiate a gracefull termination of a session before either ends can call close. This practice could also be adopted by the L3 HIPPI applications.

Whenever an application calls the (T)-service through the HIPPI_close call the (T)-service should

1. If there is an open transport connection associated with this socket, send a packet with the abort flag set and the type bit off, and enter the CLOSED state.

2. The (T)-service should delete the transport connection.

3. The socket identifier and the associated port should be marked as free and the calling application should be waken up from the HIPPI_close call.

4. Any system which receives an unsolicited abort packet from the network should tell the local application that the connection has been aborted, stop all data transfer over the given connection and delete the transport connection. The socket at the other end is kept till the remote application also calls HIPPI_close.

A.4.2.7 Data Transfer

Either ends can issue HIPPI_send and HIPPI_ read calls in any order.

```
Application |     Transport Service     | Application
client      |                           | server
            |                           |
   h_send ->|                           |<- h_read
            |send.req->                 |
            |ack.req->        ->send.ind|
            |                  ->ack.ind|
            |                  h_read ->|
            |                 <-ack.resp|
            |ack.conf<-                 |
            |<- h_send                  |
            |                           |
            |          ...              |
            |          ...              |
   h_read ->|                           |
            |<- h_read                  |<- h_send
   h_read ->|                 <-send.req|
            |send.ind<-        h_send ->|
            |<- h_read                  |
```

Figure 12: **Example of a data transfer time sequence diagram**

Whenever a user calls the (T)-service through the HIPPI_read call on an open socket, the user process should block until the (T)-service has moved the data which is queued for this connection into the user's buffer space or till the user's buffer is full. The (T)-service should return a byte counter indicating the actual length of the received data. If there is no data, the HIPPI_read call should return after a certain time to avoid deadlock situations. In case the other end has requested the connection to be shut down, ie. there is no more data to come, or in case of an error the (T)-service should return a read status code indicating closing or the type of error. If the bursts are corrupted and the errors have been detected by the HIPPI layer, it is up to the application to decide what action to take, i.g. connection abort.

Whenever a user calls the (T)-service through the HIPPI_send call on an open socket, the (T)-service will create a TPDU and try to transmit the TPDU in a HIPPI-FP packet. The application will block till all the data either has been transmitted or queued in the local HIPPI transmit buffer pool. If both ends try to send before doing a read and if there is no buffering done at the transport level, the transport connection will block forever. This is an application protocol problem.

If flow control has been chosen, the sending (T)-service must before returning the HIPPI_send call ask the destination to send an acknowledgement packet. If the entire user buffer was not received, the missing data must be retransmitted, and the sender has to ask for another acknowledgement packet. In case the acknowledgement packet does not arrive within a certain time limit, the sender should should retransmit the last packet with the request acknowledgement bit set. The sender will go on like this till it receives and acknowledgement or till the same packet has been retransmitted a maximum number of times. In case the upper retransmit limit has been reached, HIPPI_send should return an error.

In case of flow control and no buffering done at the transport level, the HIPPI_read call should never return before an acknowledgement packet has been sent. Under normal conditions the reader will receive a packet with the request acknowledgement bit set, and the reader will create and send the acknowledgement packet. However, if something starts to fail the HIPPI_read call can time out because no more data is received or the receive buffer is full before a request acknowledgement packet arrives, and in that case an acknowledgement has to be sent before HIPPI_read can return.

The sender will not be informed about any transmission errors which could have corrupted the TPDU.

A.4.3 Byte ordering

In a HIPPI-PH ordered byte stream byte 0 is the first byte, byte 1 is the second, etc. The

```
+---+---+---+---+
| 0 | 1 | 2 | 3 |
+---+---+---+---+
D31             D00
```

Figure 13: **Ordered byte stream to HIPPI-PH**

most significant byte of a word should be byte 0. Thus the most significant byte of a 32-bit word is mapped onto the highest numbered signals as shown in Figure 13, [8].

Within each byte the highest numbered signal shall be the most significant bit of the byte. The data transferred by HIPPI-FP and/or the (T)-service shall be an ordered HIPPI-PH byte stream where the most significant bit of a field is transmitted first. When a field is a multi-octet quantity the most significant octet is transmitted first.

A.5 How to map the Transport protocol onto the Framing protocol

The HIPPI-FP defines the data framing for a simplex point-to-point HIPPI link. This framing protocol should be implemented in the L3 HIPPI network. Only one service is provided by the HIPPI-FP layer, namely the FR_TRANSFER service, which lets you transfer a FP packet over a simplex HIPPI-PH link. Connection establishment and termination is taken care of by the underlying HIPPI-PH layer.

The D1 area of a HIPPI-FP packet which is only present in the first burst of the packet and is delivered separately to the upper layer, should be used to carry the HIPPI transport header. The D2 area should contain the user data of the transport packet. Since the HIPPI transport header cannot be longer than 256 bytes the D2 area could start in the first burst of a FP packet. Thus the header bit Start D2 On Burst Boundary should be off.

Since the IBM 3090 HIPPI-PH does not support short bursts, the framing protocol packets should only consist of full bursts, class 4a packets as shown in Figure 14.

Hd The Framing Protocol header, 8 bytes.

D1_Area D1 Area, contains D1 data set, if present. D1 Area should contain control information associated with the user data. The D1 Area will be delivered to the upper layer without waiting for the arrival of other bursts of the packet, [8].

```
+-----------------+--------------+
|Hd     D1_Area   |    D2_D      |
+-----------------+--------------+
|             D2_D               |
+--------------------------------+
                ....
                ....
+--------------------------------+
|             D2_D               |
+--------------------------------+
|             D2_D          (F)  |
+--------------------------------+
```

Figure 14: **Class 4a FP packet composed of full bursts**

D2_D D2 data set, this is the user data of the FP packet.

F Fill bytes are used to pad out the D2 data set to the end of a physical burst.

A.5.1 Connect

A (T)-connect.request should be mapped onto a FP_Transfer.Request with the Keep Connection flag set, the CCI to identify the HIPPI source and destination, the ULP-id to identify the HIPPI (T)-service, the D1_Size to be the length of the complete transport request packet, the D1_Data_Set to contain the whole transport packet, the D2_Size set to zero and the rest of the burst should be padding. The local HIPPI-FP entity will acknowledge the request by issuing a FP_Transfer.Confirm.

The destination (T)-service will receive a FP_Transfer_D1.Indicate, which it should acknowledge locally by a FP_Transfer.Response. The destination (T)-service should reject the HIPPI connect request if

1. The destination (T)-service for some reason is not able to receive packets.
2. There is not enough space in the destination HIPPI adapter buffer pool to receive a full transport packet.
3. The I-field is invalid.
4. The I-field was received with incorrect parity.

A (T)-connect.response should be mapped onto a FP_Transfer.Request with the Keep Connection flag set, the CCI to identify the the HIPPI host from which the corresponding request originated, the ULP-id to identify the HIPPI (T)-service, the D1_Size to be the length of the complete transport response packet, the D1_Data_Set to contain the whole transport packet and the D2_Size set to zero. The rest of the burst should be filling bits.

To make the share arbitration of the physical network as fair as possible the HIPPI-SC camp-on bit should be set whenever requesting a HIPPI connection to prevent the fastest host from always grabing a particular route. The only exception from this rule should be if the (T)-server has packets queued for several destinations. The use of the camp-on bit prevents connection rejects due to the temporary busy case, and it is therefore the only reliable method to detect whether a reject was due to an error condition or not, [12].

A.5.2 Data Transfer

The sending (T)-service should transmit packets, TPDUs, as quickly as READY permits. To obtain fair share arbitration of the HIPPI links, however, the source should release the HIPPI connection when

- 65 bursts, ie. a TPDU of maximum length, have been sent since the connection was established,
- the time it takes to send 65 bursts has expired,
- no TPDU is ready for immediate transfer or
- the next TPDU cannot be sent without exceeding the 65 burst limit.

Alternatively the HIPPI connection burst limit could be implemented as a network configuration parameter. If it is a small network with little competion for getting the same link to a certain destination, the maximum packet size should be larger. A large packet size will minimize the packet processing overhead and the number of interrupts.

A data transport packet should be mapped onto a FP_Transfer.request with the transport header in the D1_Data_Set. The ULP-id should identify the (T)-service, and the CCI field should contain the HIPPI addresses of the source and destination hosts. The D2_Size should contain the length of of the user data. The user data should start from the D2_Data_Set in the first burst. If the TPDU ready to be sent will reach the burst limit the Keep Connection flag should be off, otherwise on. The local FP entity will acknowledge the transfer request locally by issuing a FP_Transfer.Confirm.

The destination (T)-service will receive a FP_Transfer_D1.Indicate containing only the transport header of the sent TPDU. When the user data of the length given by the D1 field D2_Size has arrived the (T)-service will receive yet another a FP-Transfer.Indicate for the D2 bursts. The (T)-service should acknowledge each indicate locally by issuing two FP-Transfer Responses.

A.5.3 Disconnect

A disconnect request TPDU should be mapped onto a FP_Transfer.Request with the Keep_Connection flag off, the ULP-id to identify the (T)-server, the CCI field to contain the source and destination HIPPI addresses, the D1_Size to the length of the TPDU, the D1_Data_Set to contain the TPDU and the D2_Size set to zero. The rest of the burst should be filled out.

The destination (T)-service receiving the FP_Transfer_D1.Indicate for the disconnect request should when the application at its end is ready to close the connection create a transport disconnect response packet. The response packet is mapped in a similar fashion as the request packet onto a FP_Transfer.Request, also with the Keep_Connection flag off. Thus after the disconnect response packet has been sent the duplex HIPPI-PH connection should have be dropped in both directions.

A.5.4 Abort

A transport abort request packet is mapped in the same way as a disconnect packet onto a FP_Transfer.Request. The destination receiving a FP_Transfer_D1.Indicate carrying a transport abort request does not have to create a transport response packet, but if it has an open HIPPI connection to the origin of the abort packet, it has to drop this connection.

A.6 Summary

1. Due to the lack of software support for process-to-process communication over HIPPI L3 will develop a HIPPI transport protocol and a HIPPI API. The API should be compatible with the Internet stream sockets.

2. The HIPPI transport software should run under VM/CMS, VMS and various flavours of Unix. It is therefore important to keep the software as portable as possible.

3. The first version of the HIPPI transport protocol will not provide a checksum mechanism. The error detection provided by HIPPI-PH cannot detect all bit errors, and we therefore run the risk of transferring corrupted data. The application will be notified of errors detected by HIPPI. However, transmission errors are expected to be rare, and it is likely to assume that they will appear in bursts whereof the HIPPI-PH layer will be able to detect some of the errors.

4. If it is possible the transport level should avoid buffering data. It should simply move data between the application's memory space and the HIPPI adapter buffers to minimise copying overhead.

5. The HIPPI-PH layer implements flow control based on burst counts. However, if packets can be lost due to lack of receive buffers at the higher level, flow control must be included in the tranport protocol. The transport protocol will therefore include an optional flow control scheme, optional in the sense that it does not have to be applied on every transport connection, but mandatory in the sense that all implementations of the protocol must support it.

6. The first implementation of the L3 HIPPI software should be a minimum to be able to run only one application over a single HIPPI link between two endsystems. So the HIPPI inetd and the HIPPI_select call could for example be left out in the first version.

References

[1] R.P. Mount, Proposed Socket Interface for Communication over HiPPI.
February 8, 1991.

[2] R.P. Mount, Use of HiPPI/Fiber Channel to Build an Integrated Computing Environment Centered on IBM Mainframes.
November 23, 1990.

[3] Writing a Device Driver for AIX Version 3.2
IBM - GG24-3629-01
Second Edition - January 1992

[4] High-Performance Parallel Interface User's Guide and Programmer's Reference Manual
IBM - SC23-2492-00
First Edition - March 1993.

[5] Janet I. Egan - Thomas J. Teixeira
Writing a Unix Device Driver
John Wiley & Sons, Inc.

[6] Keith Haviland - Ben Salama
Unix System Programming
International Computer Science Series

[7] American National Standards Institute, Task Group X3T9.3
X3T9.3/88-127 REV 8.0 High-Performance Parallel Interface - Mechanical, Electrical, and Signalling Protocol Specification (HiPPI-PH),
Draft American Standard X3.183-199x, January 22, 1991

[8] American National Standards Institute, Task Group X3T9.3
X3T9.3/89-146 REV 4.2 High-Performance Parallel Interface - Framing Protocol (HiPPI-FP),
Preliminary draft proposal, June 24, 1991

[9] American National Standards Institute, Task Group X3T9.3
X3T9.3/90-106 REV 1.5 High-Performance Parallel Interface - Physical Switch Control (HiPPI-SC),
Working draft proposal, January 14, 1991

[10] Information Science Institute, Univeristy of Southern California.
Transmission Control Protocol - Darpa Internet Program Protocol Specification, RFC-793, September 1981.

[11] V. Jacobson, R. Braden, D. Borman
TCP Extensions for High Performance, RFC-1323,
May 1992.

[12] Internet Engineering Task Force, Network Working Group
J.Renwick, A. Nicholson - Cray Research, Inc.
IP and ARP on HIPPI. January 1992.

[13] S.J. Leffler, R.S. Fabry, W.N. Joy, P. Lapsley, S. Miller, C. Torek
An Advanced 4.3BSD Interprocess Communication Tutorial.