The Protocol Bypass Concept for High Speed OSI Data Transfer

C. M. Woodside*, K. Ravindran and R. G. Franks***

* Telecommunications Research Institute of Ontario, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada K1S 5B6

** Dept. of Computer and Information Sciences, Kansas State University, Manhattan KS 66506

Abstract

A protocol bypass is a fast processing path which is used for some data units, for instance for large data packets. It can provide conformance with standardized layered protocol system like OSI, together with some of the performance benefits of a "lightweight" protocol. The concept is discussed as it applies to the movement of user data by the OSI transport and session protocols, with some implementation experience, and an outline for an approach to formally deriving bypass specifications from protocol specifications is given. This outline uses some steps which must still be proven to be practical. Correct interleaving of data units from the two paths is a major concern, especially with multiple asynchronously specified layers. It seems that the difficulty can be overcome and the concept has promise. In the (rather limited) implementation, bypassing consistently outperformed parallel processing as a means of performance enhancement.

1 Introduction

The search for generality, flexibility and standardization in communications protocols has led to the OSI layered system [1],[2] and many offshoots such as MAP [3]. However, the slowness of execution of the protocol implementations, which is essentially due to the complex checking of conditions that is done at every operation, is becoming a limiting factor in some applications. Therefore a new generation of lightweight or high-speed data transfer protocols is now emerging.

The lightweight protocols exploit the low error rates of many networks, and use larger packets, reduced options to unclutter the data path, and more efficient methods for congestion control. They are partly motivated by the high speeds of new fiber-based networks, partly by high-throughput applications such as file system backups and full motion video, and partly by performance constraints already being felt with current protocols. Examples are Zwaenepoel's Blast protocol [4, 5], VMTP [6], and XTP [7].

Present-day "heavyweight" protocols such as OSI place notable performance constraints on distributed applications. Svobodova [8] surveys the status of transport protocols (OSI and others) running on LANs and finds throughputs up to a maximum of about 2 Mbits/s (other references are found in her paper). The goal of the current lightweight

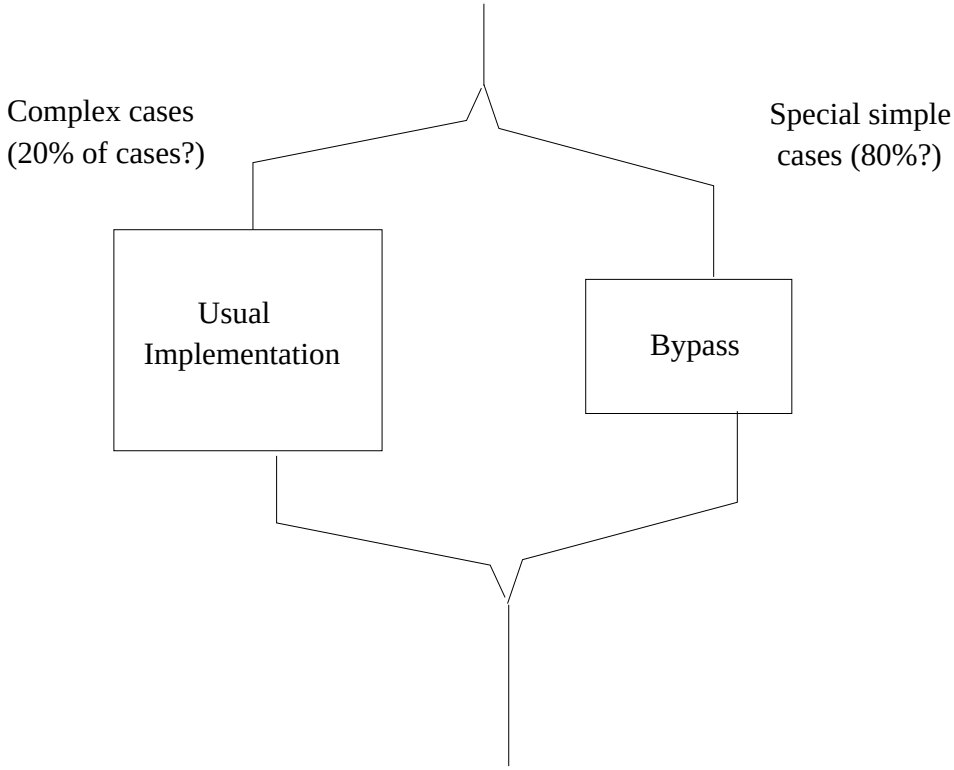


Figure 1: The Bypass Concept

protocols is to obtain an improvement by an order of magnitude now, and much greater increases later on faster processors and nets, into the range 100–1000 Mbits/s. The protocols do not have to be completely new ones; Clark et al [9] have analyzed the fundamental limitations in IP/TCP to show that it could be operated in the 100 Mbit/s. range if implemented properly. They describe a "fast path" concept in [9],[10] which has already, in an existing workstation environment, moved files at close to the capacity of an Ethernet. Jacobson has called his guiding principle for a fast receiving path "Header Prediction".

The bypass concept described here is an attempt to generalize Jacobson's "Header Prediction" to other protocols, particularly to layered protocol systems like OSI. Figure 1 shows the general notion of The bypass part is activated when a very limited set of conditions apply which may be executed quickly; it will be effective if the conditions are satisfied often enough. Hopefully, one can obtain both the performance benefits of a lightweight protocol, when operating within a restricted set of circumstances, and the standardization and functional benefits of a fully conforming standard protocol stack. To obtain the full potential performance, some adaptation of the OSI standards may be necessary (e.g., larger window sizes). The bypass is an implementation concept rather than a new protocol, but it raises interesting questions about specification, such as

- specifying the bypass path and control when several protocol layers are involved,
- specification properties which could constrain the applicability of the concept in

some cases,

- formal derivation of the bypass process specification from the protocol specification.

The bypass concept is based on a standard idea in performance enhancement, called the 'centering principle' by Smith [11] and sometimes called the "80–20 principle". In 'centering', special fast processing is provided for a frequently-occurring case. Although the idea is standard, and seems to be relatively straightforward in [11], it is nontrivial for layered protocols because of the sequential relationships generated by segmenting and reassembly of data, the complex interchanges supported by each layer, the web of relationships that may exist between entities at different layers due to multiplexing of connections, and especially because of the asynchronous nature of the separate layers.

The purpose of this paper then is to explore the feasibility of constructing a bypass for an arbitrary layered protocol, and more particularly for OSI. It begins with an informal outline of a particular bypass for the OSI Session and Transport layers, identifying difficulties associated with operating and controlling a bypass, that seem to be fundamental. Ways to overcome these difficulties are discussed, and have been demonstrated by some preliminary implementation experience discussed in Section 6. The formal derivation of bypass specifications is approached in Section 7 by outlining how the simple control strategy used in Sections 3, 4 and 5 could be described formally. The paper is intended to show that bypassing is a fruitful and important mechanism for combining standard and lightweight protocol concepts, and that this topic is worth pursuing further.

2 The Architecture of a Bypass

Figure 2(a) illustrates the architecture of a bypass, including the following nomenclature:

stack the set of entities implementing the standard protocol, which are to be bypassed,

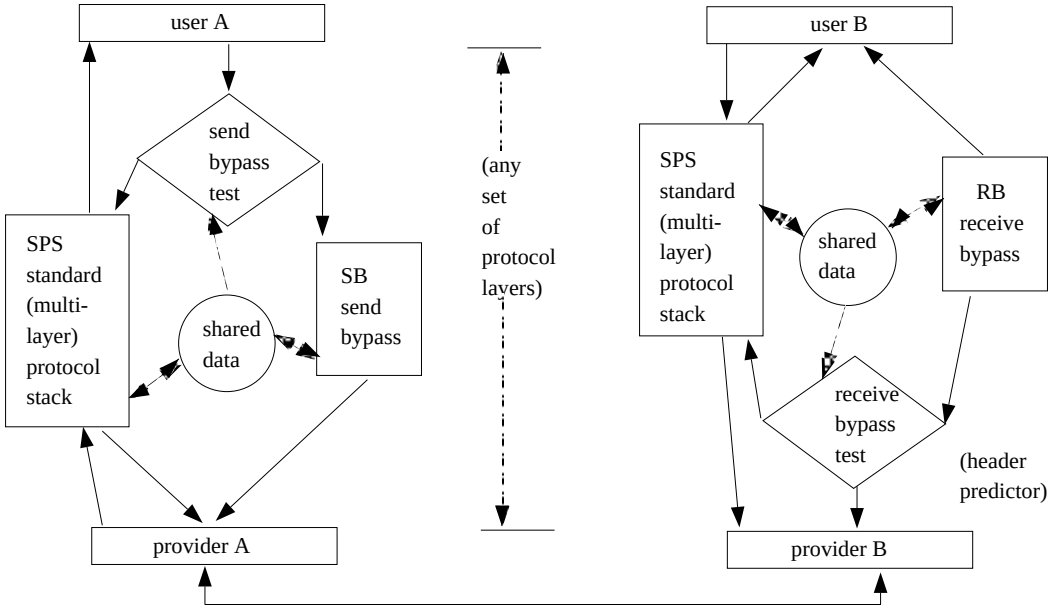
user the user of the topmost layer of 'stack', assumed to be unique,

provider the service provider entity below the bottom layer of 'stack', also assumed to be unique at each end. These embody a service layer serving both ends, and hiding the physical transmission of data, etc.

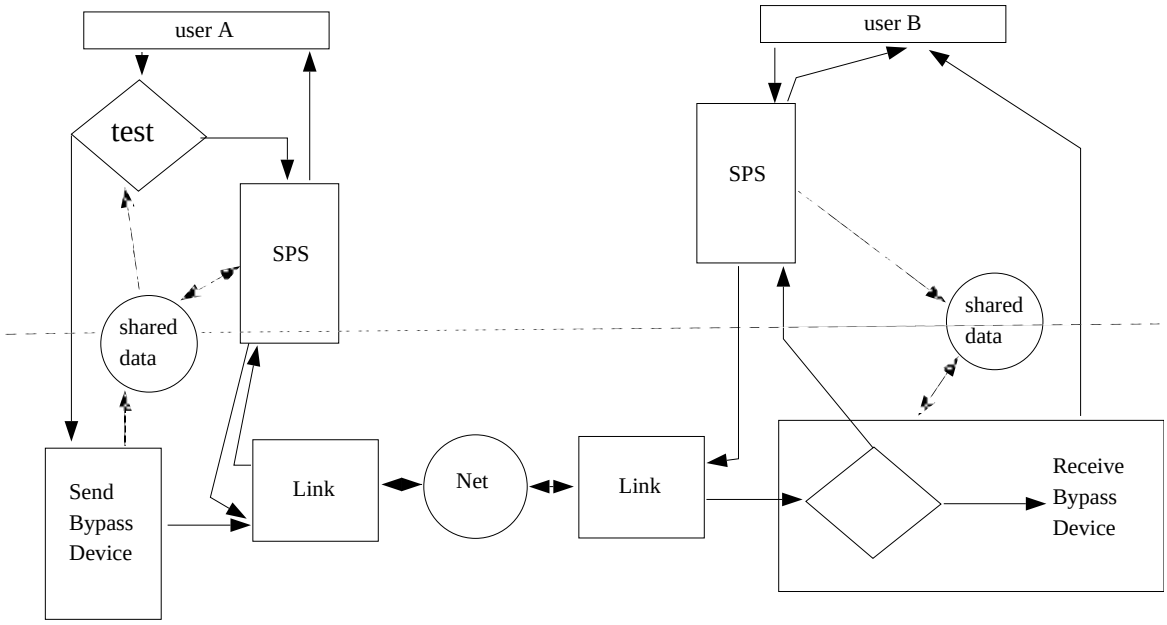
The Figure shows a system to support one-way data transfer, in which data flows from the 'Data Sender' end A to the 'Data Receiver' end B; note that control packets are also involved and flow in both directions.

As a data unit flows from left to right, it is filtered by tests and if it qualifies, is processed by the fast path. The fast path is envisaged as a single process without internal concurrency, cutting through all the layers of 'stack', while 'stack' may have internal concurrent processes. Jacobson's 'Header Prediction' conforms somewhat to this model, for in his system a received packet is examined for a match against a stored predicted header at the test labelled RBYPASSTEST, but there is no SEND test.

One of the potential values of isolating the fast path rather than just optimizing the code in 'stack' is the excellent prospect of implementing the bypass in hardware. Figure 2(b) shows a suggested division when the service entities are hardware-implemented link controllers such as Ethernet controller chips. SBP and RBP are the 'send' and 'receive' bypass devices. The receive test RBYPASSTEST is in hardware, to avoid breaking the hardware path to the user level, and the RBP device writes data into addresses supplied in memory accessed by the device. The send test does not have such compelling reasons to be in hardware, and so is not.



(a) Architecture of Bypass for Sender and Receiver



(b) A Hardware Implementation Strategy with two bypass devices
(for the case where the provider is the link controller)

Figure 2: Bypass Architectures

3 One-Way Bypass of OSI Session and Transport Layers

To focus the discussion this section examines a bypass around an OSI session layer combined with a Transport Class 2 layer, and used with extended TPDU-numbering, but without flow control.

3.1 Ideal Data Transfer

Further, bypassing will be provided only for those data units and transfers which satisfy the following conditions for "ideal data transfer",

- the session service data units are of type "data",
- there is no segmentation/reassembly, concatenation/separation, or multiplexing in the two layers,
- there are no piggybacked acknowledgements or window credits

Attention is also restricted to a *one-way* bypass, as illustrated in Figure 2(a). It could be used for instance in a file server with a send-only bypass loading pages across a network to diskless workstations with receive-only bypasses. Only one-way "fast paths" were considered in [9, 10]. Greater generality will be introduced in later sections.

In an "ideal data transfer" each SSDU corresponds to exactly one NSDU and entails these operations in the layers:

- increment PDU numbers by one in each layer at sender and receiver,
- send an acknowledgement, after a receive.
  The bypass must produce the same effects as a standard implementation.

  The general architecture of a one-way bypass system is as shown in Figure 2(a).

3.2 The Bypass Tests, without Window Flow Control

Correct execution of the protocol by a bypass depends on two key operations – the identification, at the bypass-condition tests in Figure 2(a), of those data units which which will give an ideal transfer, and correct interleaving of data units which follow the two paths. On sending a user data unit, the conditions to be met are that it be small enough to fit into one TPDU, and that it be a 'data' type of unit (rather than control). The connection must also meet the conditions stated above (eg., no multiplexing). Therefore the bypass test will be only partly based on headers (as it was in Jacobson's method); it will also include some aspects of the state of the protocol layers, shown as the "shared data" in Figure 2(a).

Packets which fail the bypass test follow the normal processing path through the full implementation. This raises the problem of ensuring correct interleaving of data units from the two paths where they rejoin. This is a non-trivial problem because in principle the separate layers in OSI are asynchronous entities. In this work interleaving is ensured by further assuming or enforcing that the protocol execution for an ideal data transfer is atomic with respect to all other operations (send or receive, and by either layer) for the same connection. This extra assumption is called 'relative atomicity' of processing. Then there is a simple toggle in the flow, switching from one path to the other, and the

initiation of the new path at each switch is delayed until the former path is empty. This delay is simple to control when switching away from the bypass because it is a relatively simple process; the delay must be carefully enforced for switching the other way.

To be effective, the bypass must go around every layer with an important processing load, and the window size must be large enough to make the extra speed useful. Larger window parameter values than those now standard in OSI would be needed to fully exploit the concept. Bypassing of link processing would assume a capability for large link-level data units, to avoid segmenting at the link level. In [9], for example, link data units of 8K bits or more were used.

Initially, the tests will be given for a case without window flow control; it will be added later in this section.

Sender

Besides the usual protocol entities for Session and Transport layers, there is a send bypass entity SB, as shown in Figure 2(a), which only operates when sending a data unit from the user. The bypass entity will be associated with the session service access point, and will communicate directly with the network (provider) SAP for the connection that is being bypassed. A particular set of entities (session, transport and bypass) will be associated with a given user-to-user connection by an initiation mechanism considered below.

The following protocol entity state data must be shared with the bypass entity (remember that we are ignoring window operation at this point):

- session layer major state (read-only by bypass process)
- SPDU-NR, next session PDU number for sending (read-write by bypass process)
- session connection identifier (read-only by bypass process)
- transport layer major state (read-only by bypass)
- TPDU-NR, next transport PDU number for sending (read-write by bypass)
- max TPDU size (read-only by bypass)
- transport connection identifier (read-only by bypass)
- network entity information for sending (eg. SAP identifier) (read-only by bypass)

Besides the above, two test conditions must be accessible to the bypass test:

- session entity empty (entity is idle and no data units are ready for processing in either direction)
- transport entity empty (entity is idle and no data units are ready for processing in either direction)

Then the bypass test and operation for send has the form:

```
SBYPASSTEST = (session layer state = DT) and
                (session entity empty) and
                (transport layer state = DT) and
                (transport entity empty) and
                (predicted TPDU length <= TPDUmax)

if SBYPASSTEST then addheader; send NSDU; update; else send SSDU;
```

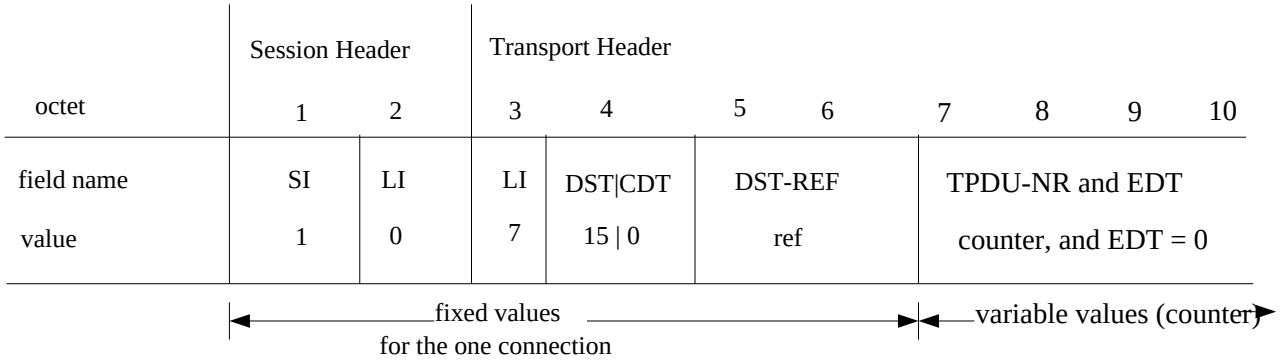


Figure 3: Session and Transport Header Format for the Case of Section 3
(class 2 transport with extended TPDU-NR and EDT format, no segmenting/reassembly so EDT=0, no flow control so CDT=0)

where "send NSDU" sends to the network entity, while "send SSDU" sends to the session entity (for non-bypassed units), and "update" increments SPDU-NR and TPDU-NR. The operation "addheader" adds the combined headers for the two layers to the data in the form shown in Figure 3, to give exactly the same TPDU that would be constructed by the two layers together. In the restricted operational cicumstances of this bypass a fixed header template, with only one variable field for TPDU-NR, can be retained for fast processing. Acknowledgements may be handled by SPS.

Receiver

At the receiver, the following protocol entity state data must be shared:

- session user entity information for delivering SSDUs
- session layer state (read-only by bypass process)
- SPDU-NR next session PDU number for receiving (read-write by bypass process)
- session connection identifier (read-only by bypass process)
- transport layer state (read-only by bypass)
- TPDU-NR next transport PDU number for receiving (read-write by bypass)
- last TPDU acked (read-write by bypass)
- transport connection identifier (read-only by bypass)
- network layer information for receiving (read-only by bypass)

The same two layer state test conditions as before must be accessible to the bypass test:

- session entity empty
- transport entity empty

The bypass test is executed on receipt of an incoming NSDU from the network:

```
RBYPASSTEST = (session layer state = DT) and
              (session entity empty) and
              (transport layer state = DT) and
              (transport entity empty) and
              (header match)
```

The header match condition ensures that the data is for the right connection, and has the expected sequence numbers. It is based on a predicted and stored representation, as illustrated in Figure 3. The bypass operation for receive has the form:

```
if RBYPASSTEST then stripheader; deliver SSDU; update; else deliver NSDU;
```

where "deliver NSDU" sends to the transport entity, while "deliver SSDU" sends to the session user entity. The operation "stripheader" removes the header component from the data as a standard, preprogrammed number of octets, and if necessary does the transport level cyclic redundancy check. The operation "update" increments TPDU-NR, SPDU-NR and issues acknowledgements as NSDUs (using a preconstructed format into which the PDU numbers are inserted). Each time an AK is issued the "last TPDU acked" value is updated.

Comments on Operation of a One-Way Bypass

The specification above assumes that the bypass entity shares data with the standard entities, and that process scheduling can enforce the relative atomicity of bypass processing with respect to the standard entities it is associated with.

Acknowledgements are received by the standard entities, and their handling is not accelerated. Since every packet is acked this may become a significant load, and the traffic through the standard entities may interfere (through the "empty entity" conditions) with availability of the sending-side bypass. Ideally, by the appropriate use of priorities or by some other mechanism, the receipt of the AK by the standard entities should always be completed before the test for the next send is executed. Then a long sequence of user data packets which meet the conditions on length should all pass through the bypass.

Both the above points indicate how the specification of a bypass interacts more closely with implementation considerations than is the case with the OSI standards. This is to be expected when the existence of the bypass is governed by performance goals. The intention here however is to specify the implementation considerations only as formal constraints, without saying how they are to be satisfied.

3.3 Additional Features

Adding Window Management

We can bypass with window flow control by adding

1. to the shared data at the sender:
   - available transport window credit (read-write by bypass process)
2. to the SBYPASSTEST
   - the condition (window credit available)

The actual window management can still be done by the transport entity when it handles an AK, or other incoming PDU with window credit attached. The sender bypass rejects the unit if there is no window credit, and the data is queued by the transport entity to be sent by it on receipt of credit. Until the backlog is cleared the (transport entity empty) condition will fail.

This description provides window functionality, but if the window ever closes the slower path will be used and the sender may never catch up with the backlog, and may never be able to re-open the bypass. To avoid this problem the bypass would itself have to handle AKs and window credits, as discussed under the two-way version below.

Initiation of Bypass Flow

Bypass operation is most useful for certain situations, such as bulk data transfers. At other times the test overhead may make it unprofitable. The appropriate conditions for initiating it may be detected and operated on in various ways, such as:

- user initation, by an action from the application level;
- from the quality of service parameters (QOS), as interpreted by the transport layer
- by an automated process based on observing the packet stream.

The details of the initiation process are not a major concern of this paper; any of the above approaches could be taken. They have different implications for affecting the implementation of the standard stack, however.

4 Two-way Bypass

Here we consider doing the bypass in both directions. The above specification of send and receive bypasses are combined into a single process (still atomic, but now triggered by events either at the session service user or the network service provider) so that each end has both the send and receive features. Then data transfer will be bypassed in either direction while AK and window operations will be carried out by the standard entities. We will now examine some details of the changes that would be required.

Full AK Handling

Further performance improvement might result, in cases with window flow control and error control, if AKs and window management could be handled by the bypass. For AKs, the receive bypass makes a second test if RBYPASSTEST fails, as follows:

```
RAKTEST = (session layer state = DT) and
          (session entity empty) and
          (transport layer state = DT) and
          (transport entity empty) and
          (header match)
          (packet matches AK template)
```

which leads to an alternative operation when RBYPASSTEST fails, as:

```
if RAKTEST then ack-update;
```

Here "ack-update" updates the "last TPDU acked" to the value in the AK, and updates flow control window credits as conveyed by the AK. Similarly, on RBYPASSTEST the "update" operation ends by processing piggybacked credits in the same way.

On receipt of window credits in cases where there is a backlog of transport data units, the standard transport entity must then be forced to process its waiting data, as it would do on receipt of window credits in an AK.

Window Management

If on the other hand the bypass attempts to also send the waiting data units as part of a single 'relatively atomic' operation, it produces messy scenarios. On receipt of window credits the queue of waiting data may include units that cannot be bypassed, mixed with some that can. To sort out the bypassable units, a different bypass test is needed because the data has already been partly processed. Further, when one of the waiting units is sent through the standard entity it breaks up the atomicity of the bypass processing which was originally initiated by the receipt of the data unit with the window credits.

A simple-minded solution is to block all data units at the bypass entrance when window credit is exhausted, and then when credit is received to force their processing. This forced processing would activate the bypass test on each of the waiting data units in turn until either the credit or the queue of waiting data is finished. Data units taking the standard path would be processed to completion before the next was tested, as discussed earlier, to permit the next one to be bypassed.

5 Experience

A bypass for the OSI Session and Transport Layers along the general lines described above has been programmed into a protocol performance testbed described elsewhere [12]. The standard entities in the testbed were implemented on parallel processors, and this both biased the test against the bypass and also complicated the sharing of data and the entity empty tests, which were implemented with inter-process messages. Nonetheless, even in this rather hostile framework, the bypass roughly doubled the throughput of a long sequence of data packets.

The task architecture of the implementation is shown in Figure 4. A real-time kernel called Unison [13] was used; Unison has parallel tasks with send-receive-reply messages and a task can send to or receive from specific ports, shown as little parallelograms within the larger shapes representing tasks. The processors were MC68020's on a VME bus. The two ends of the connection were run on the same backplane, so the net work delay is insignificant and flow control is not a problem; in any case only Transport level 0 was implemented so flow control was not an issue.

The protocol bypass concept was applied to an implementation of the OSI session and transport layers written in C and running on a common bus multiprocessor. The original software consisted of four layers: a user aplication, a session and transport layer, and a vestigial network layer. (The latter was simply used to provide the proper interface to the transport layer and to connect the source and destination together.) This design, including the modification for bypassing, is illustrated in Figure 4.

The protocol software must be capable of handling traffic in both directions simultaneously, therefore layers cannot send messages directly to one and another. If they did,

deadlock would arise when two layers tried to communicate with each other at the same time. Consequently, each layer consists of up to three tasks. The first task implements the actual functionality of the layer while the others are used to transport data from one layer to the next.

For protocol bypassing, the original design was augmented by changing the network and application layer software to reroute messages directly to each other. The bypass is designed so that the user layer always attempts to send data using the bypass route, the exception occuring when data must be segmented. Incoming packets are initially routed through the main protcol software. However, once a data connection is established, the network layer, through the bypass software, also attempts to route data through the bypass channel. Packets which fail to match the bypassing criteria are routed in the normal manner. The modifications to the architecture are indicated by shaded portions in Figure 4.

Experimental Configurations

In the original design, up to eight processors were used: one processor was assigned to each of the session and transport entities, to each half of the user-level software, and to each half of the interface between protocol stacks. This configuration exploited the maximum degree of parallelism on the hardware. The protocol software could also be partitioned to run on only two processors. In this configuration, all of the software making up the source half of the connection ran on one processor, while the destination half ran on the other.

The bypassing variant of this software was also designed to run on multiple processors. Since the bulk of the protocol bypass software is found in the user and network layers, no additional processors are needed. Furthermore, when the bypassing is operational, only four of the eight processors are active because the session and transport layer tasks are not used.

Results

Tables 1 and 2 present the throughput of the system when configured with one processor per party (two processors in the testbed) or four processors per party, (eight processors in the testbed) respectively. In the latter case each major task in Figure 4 had its own processor. These tests were conducted using a Class 0 transport connection sending data in one direction only. The performance improvement by using bypass ranges from 120% for small packet sizes to 75% for larger packet sizes, when using just two processors. When tested using the eight-processor configuration, the throughput improvement ranges from 50% down to 25%.

It is notable that a bypass using only one processor per party was uniformly better than a parallel implementation on four processors per party without bypassing. This is due to the inherent efficiency of the bypass concept. For messages of 500 bytes and over the multiprocessor implementation with the bypass performed worse than the uniprocessor implementation. This appears to be because of bus saturation, since the bypass coordination and inter-layer data-passing both placed loads on the bus in the multiprocessor case.

Figure 4: Protocol Software Task Architecture (one end) The parallelograms represent tasks; the arrows are for the direction of sending messages. The shaded components implement the bypass (sending on the right, receiving on the left).

Message Size (octets)	Throughput (bits/sec)	
	Standard	Bypassed
20	27000	59000
50	65000	143000
100	126000	273000
200	235000	502000
500	473000	1009000
1000	717000	1496000
2000	965000	1845000
5000	1216000	2146000

Table 1: Throughput - One Processor per Party (2 processors in all)

Message Size (octets)	Throughput (bits/sec)	
	Standard	Bypassed
20	47000	70000
50	114000	167000
100	215000	300000
200	387000	525000
500	742000	965000
1000	1057000	1322000
2000	1297000	1608000
5000	1485000	1844000

Table 2: Throughput - Four Processors per Party (8 processors in all)

6 Deriving a Formal Specification for a Bypass

In order to bypass a certain set of layers of a given protocol, with a known formal specification, we seek a method for formally deriving the bypass from the protocol. This would avoid hand-derivation, with its attendant likelihood of error. This section describes an approach which formalizes the discussion of sections 3, 4, 5. The approach is not the only one possible, and indeed has been deliberately restricted, for simplicity. It has two steps:

1. restrict the permitted concurrency in the standard protocol specification (let us call it SPS), without restricting its functionality, by imposing a "single-event" condition SE; this is always possible.
2. project the restricted specification into a subspecification B for a bypass path. It is guarded by defined conditions which become part of the bypass test for that path.

For brevity, the remainder of this section only indicates how a formal derivation could be constructed, based on a LOTOS-like FDT [14]. Define:

$SPS(u,l)$	the set of protocol entities to be bypassed, defined as a process communicating through an upper gate u and a lower gate l,
$user(u)$	the user of the topmost service provided by SPS,
$provider(l)$	the service provider to the bottom-most entity in SPS,
ES	the empty-stack condition, satisfied by SPS when all its protocol entities are blocked awaiting events and all its FIFOs are empty.

We consider ES to be applied to operations and data units for a single connection. When ES is true and an event arrives at either u or l it initiates processing by SPS, and the period until ES is true again is a busy period for SPS.

By adding some mechanism, such as a process scheduler or a screening process which filters the inputs to SPS, a restricted process SPS' (u,l) can be derived which always processes each input event at u or l until ES is true, that is until all possible interactions (internal and external) are exhausted, before accepting input again. Since protocol specifications make no assumption about the arrival time of the next event, this restriction can always be contructed in such a way to still satisfy the protocol specification and without introducing deadlocks. That is, SPS' is a correct implementation of SPS. We will not attempt to define SPS' formally here. If we denote a generic external event as 'ul' (meaning u or l) then one event is processed by $STACK$, defined as follows:

$$STACK = ul \rightarrow SPS'$$

This can also be expressed as a combination of a send process $SPSS'$ for u, and an alternative receive process $SPSR'$ for l:

$$\text{STACK} = ul \rightarrow SPS' = u \rightarrow SPSS' | l \rightarrow SPSR'.$$

We will define the above notation to permit shared data between the alternative processes $SPSS'$, $SPSR'$. Now we divide the event stream into components ub, lb and uln:

ub for those 'send' events u for which the data associated with the events satisfies bypass conditions for sending (e.g., a small enough data unit),

lb similarly for bypassable events l, for receiving,

uln for all the other events, that cannot be bypassed,

and we also introduce a pair of guard conditions gs and gr, which are satisfied if the internal state of SPS' is satisfactory for send bypass, and receive bypass processing, respectively. The process notation

$$ub(gs) \rightarrow SPSS'$$

represents an event ub and satisfaction of the guard condition gs together leading to $SPSS'$. gs and gr must satisfy

$$\begin{aligned} ub(gs) \rightarrow SPSS' &= ub \rightarrow SPS' \\ lb(gr) \rightarrow SPSR' &= lb \rightarrow SPS' \end{aligned}$$

Then

$$\begin{aligned} ul \rightarrow SPS' &= ub \rightarrow SPS' | lb \rightarrow SPS' | uln \rightarrow SPS' \\ ul \rightarrow SPS' &= ub(gs) \rightarrow SPSS' | lb(gr) \rightarrow SPSR' | uln \rightarrow SPS'. \end{aligned} \quad (1)$$

Finally we define bypass processes BS for send, BR for receive, in any way that satisfies

$$\begin{aligned} ub(gs) \rightarrow BS &= ub(gs) \rightarrow SPSS' \\ lb(gr) \rightarrow BR &= lb(gr) \rightarrow SPSR' \end{aligned} \tag{2}$$

The construction (2) implies that $SPSS'$, $SPSR'$, BS and BR have the same restriction on concurrency as SPS'. Combining (1) and (2) we obtain

$$\text{STACK} = ul \rightarrow SPS' = ub(gs) \rightarrow BS | lb(gr) \rightarrow BR | uln \rightarrow SPS'$$

With recursion the resulting process X is

$$X = (ub(gs) \rightarrow BS | lb(gr) \rightarrow BR | uln \rightarrow SPS');\ X$$

This discussion which ends here, opens up a number of further questions about formal descriptions:

- how to formally describe all the necessary process properties (i.e., is LOTOS able to do it?);
- how to specify SPS', $SPSS'$, $SPSR'$;
- how to specify the guard consitions gs, gr;
- how to split the event stream into the components ub, sb, etc.
- how to derive BS and BR from SPS'.

7 Effects of a Bypass on the Implementation of the Standard Stack SPS

An attractive notion is to be able to add a bypass to any standard stack implementation without modifying the latter at all. This is possible in principle because the tests are external to the stack; the only requirement is sharing of the stack state data. Because mutual exclusion is provided by the relative atomicity and empty stack conditions, only the location and coding of the necessary state must be known. Because our own implementation in Section 6 did data-sharing and established the empty stack condition with messages, it required extensive changes to the standard entitites.

In practice it is somewhat doubtful that the empty stack condition can be established without at least a thorough knowledge of the implementation, as it is not part of the protocol standard.

8 Conclusions

The present paper has described a bypass or "fast path" concept which may be used with layered protocols. It has described it informally, described implementation experience, and discussed how bypasses may be specified formally. It seems that, as an implementation technique, bypassing offers significant immediate performance payoffs. The specification problem raises some challenging issues, listed above.

An interesting question opened here is that of deriving a *projection* of a protocol or other asynchronous process, into a "subspace" of processes constrained by more-or-less arbitrary conditions such as those for bypassing. Also a better technique for connecting and controlling the bypass may be found, which would not require the ES condition, but rather would allow full concurrency on at least one path.

Acknowledgements

Discussions with Luigi Logrippo and Moshe Krieger were helpful in formulating the problem, as was a course-work project report by Cecilia Geldrez and Yueping Lu on a different version of the problem.

This research was supported by the Ontario government program of Centers of Excellence, through the Telecom Software Methods Project of TRIO, the Telecommunications Research Institute of Ontario.

References

[1] Int'l Standard Organization, *Information Processing Systems - Open Systems Interconnection – Part 1: Basic Reference Model, ISO7498-1.*

[2] W. Stallings, *Handbook of Computer-Communications Standards*, vol. 1. New York: Macmillan Publishing Company, 1987.

[3] M. Kaminski, "Protocols for communicating in the factory," *IEEE Spectrum*, April 1986.

[4] W. Zwaenepoel, "Protocols for large data transfers over local networks," in *ACM SIGCOMM Computer Communication Review, vol. 15, no. 4, Proc. of Ninth Data Communications Symposium*, September 1985.

[5] J. Carter and W. Zwaenepoel, "Optimistic implementation of bulk data transfer protocols," in *Performance Evaluation Review and Performance '89, vol. 17, no. 1*, ACM SIGMETRICS and IFIP, May 1989.

[6] E. Nordmark and D. Cheriton, "Experiences from VMTP: How to achieve low response time," in *Proc. IFIP Workshop on Protocols for High-Speed Networks*, pp. 1–15, May 1989.

[7] Protocol Engines Inc., *XTP Protocol Definition, version 3.3*, December 1988.

[8] L. Svobodova, "Measured performance of transport service in LANs," *Computer Networks and ISDN Systems*, vol. 18, pp. 31–45, 1989.

[9] D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An analysis of TCP processing overhead," *IEEE Communications Magazine*, vol. 27, pp. 23–29, June 1989.

[10] D. Clark, M. Lambert, and L. Zhang, "NETBLT: A high throughput transport protocol," in *Frontiers in Computer Communications Technology: Proc. of the ACM-SIGCOMM '87*, (Stowe, VT), pp. 353–359, Association for Computing Machinery, August 1987.

[11] C. Smith, "Independent general principles for constructing responsive software systems," *ACM Trans. on Computer Systems*, vol. 4, February 1986.

[12] G. Franks and C. Woodside, "Some software designs for OSI protocols in a Harmony/Unison environment," in *Proceedings of the Canadian Conference of Electrical and Computer Engineering*, (Montreal, P.Q.), September 1989.

[13] Multiprocessor Toolsmiths Incorporated, *Unison Real-Time Multitasking, Multiprocessing Operating System User's Guide*, 1987.

[14] O.S.I., "LOTOS: A formal description technique based on temporal ordering of observational behavior," *International Organization for Standardization, Information Processing Systems, Open Systems Interconnection*, vol. ISO8807, 1988.