# General Bypass Architectures for High-Performance Distributed Applications

**C. M. Woodside, G. Raghunath**
Dept. of Systems and Computer Engineering
Carleton University, Ottawa, Canada K1S 5B6
cmw@sce.carleton.ca, raghu@scs.carleton.ca


**Abstract**

End-system performance limits are a barrier to achieving many of the planned applications for high-speed networks. We seek architectural principles for designing the end-system software which will make it easier to achieve high throughput and low delay. A bypass architecture, previously applied to protocol stacks, is generalized in this paper to any application which handles large volumes of information at a high rate. It is necessary to consider the applications as well as the protocol stack, to avoid having the processing bottleneck simply reappear at the higher level. The resulting architecture is modular, with attachments for bypass paths which traverse the system. The performance and workload impact of a bypass-based architecture may be dramatic.

**Index terms:** distributed computing, performance engineering, communications software, multimedia workstations.

## 1. Introduction

It is a commonplace observation that the use of high speed networks suffers from bottlenecks in the end-systems. Protocol execution alone is quite demanding, but various special optimizations have been found which make transfers in the range of some hundreds of Mbits/s feasible. Now similar optimizations are being investigated for the application design in particular kinds of end-systems, such as a video-on-demand server.

While specialized optimizations are useful in communications software, they often produce monolithic integrated designs that are very technology-dependent and hard to maintain. It would be better to find a general architectural principle with conventions which would make it easier to build end-systems from a variety of components, and also to apply optimizations across collections of components. The General Bypass Architecture is such a principle, which was developed originally for layered protocols in [1], [2], [3], [4]. This paper describes how it can be applied to application software of general modular structure, and explores its performance characteristics.

### The Bypass Concept

The notion of the bypass begins from the conventional notion of a “fast path” in performance tuning. Figure 1 shows a segment of processing with a fast path. The segment has a general case which takes $T$ sec to compute. Under “condition $B$”, it reduces to a simpler computation $F_B$ taking the smaller value $T_B$ sec. The relative frequency of the condition being true is $p_B$, and the execution cost reduction factor is $R_B$:

$$R_B = \frac{p_B T_B + (1 - p_B)T}{T} = p_B(T_B/T) + (1 - p_B) = 1 - p_B(1 - T_B/T)$$

$R_B$ is small when the improvement is large. We will call path $F_B$ the bypass path and $R_B$ the bypass workload ratio.

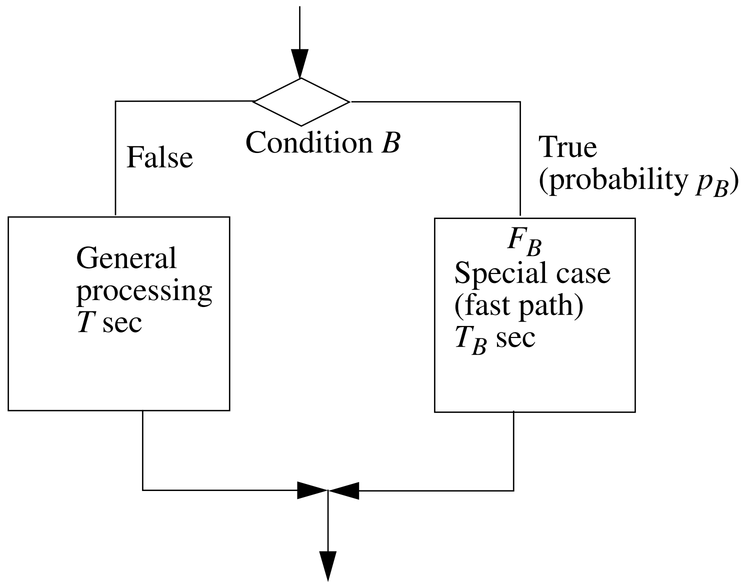


Figure 1. The Notion of a Fast Path or Bypass

In a protocol system the condition $B$ and its special processing are somewhat complex because of protocol state. In [1] it was demonstrated that a combined bypass for several protocol layers can be constructed for bulk data transfers, based on a condition $B$ depending on

- the input event and its arguments (e.g. data packets of a certain size)
- the protocol state (in “data transfer” state)
- the control state of the protocol entity (no data units being processed ahead of it).

The probability $p_B$ depends on the size of the bulk data transfers ($p_B$ =1 - 1/(data units per transfer)), provided there are no error packets or other exceptions in the flow. Thus $p_B$ can be very close to 1. $T_B$ also is very small, just a few machine instructions. The architecture of this protocol bypass is shown in Figure 2.

**Related Work**

Early work on efficient implementation of protocols was carried out by Clark, resulting in the widely used upcalls mechanisms [5]. Tuning a standard protocol stack such as TCP/IP was investigated by Jacobson et al., [6] culminating in the header-prediction algorithm. The protocol bypass is a generalization of the header-prediction algorithm that can be applied to both the sender and receiver stacks. More recently Clark and Tennenhouse [7] proposed an architecture for new generation protocols based on Integrated Layer Processing and Application Layer Framing. The main principle behind this architecture is the separation of data flows from control flows in the protocol

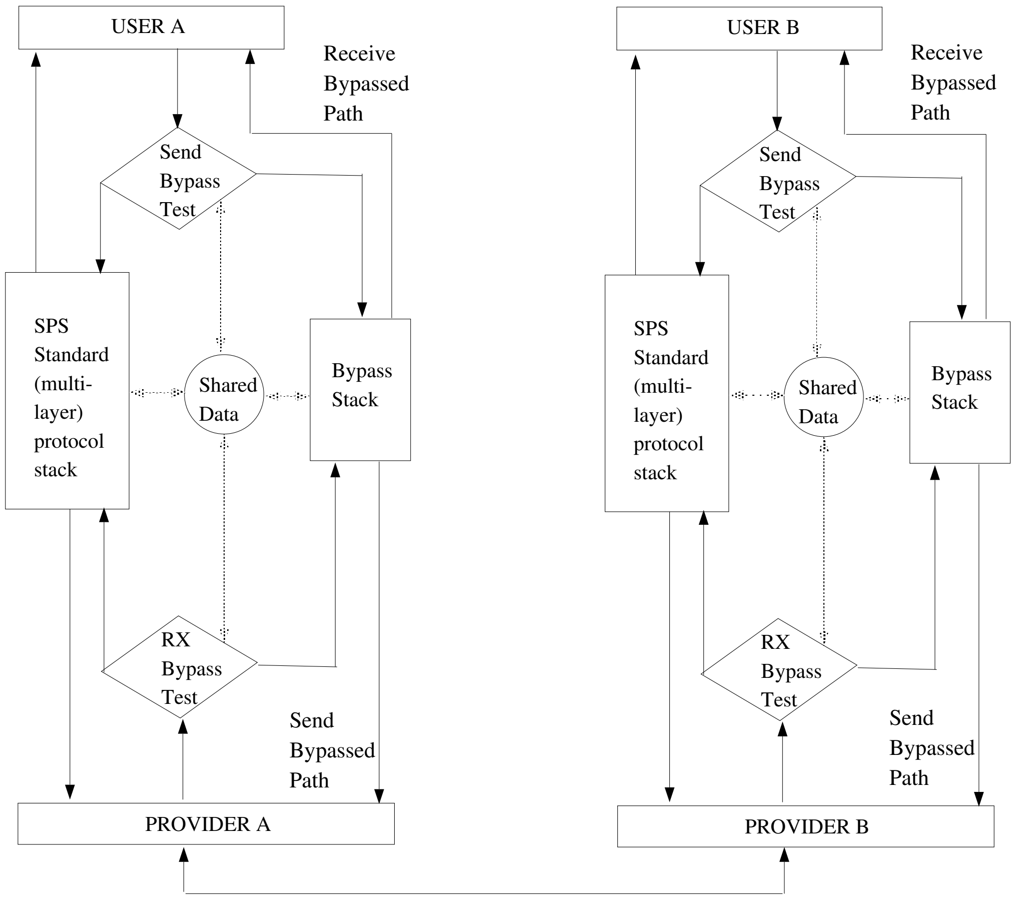


Figure 2. A Layered Protocol System with Bypass [1]

processing. Other work, at the operating systems level has also addressed the issue of efficient data movement, including the ideas of FBUFs of the X-kernel [8] and Container-shipping [9]. The protocol bypass architecture complements all these ideas and can be combined with them. It adds to performance improvements by identifying fast paths and allowing them to co-exist with the standard stack. One can view the protocol bypass as a design centered around the data path which is optimized for bulk data transfers.

The rest of the paper is organized as follows. Section 2 describes the further extension of the protocol bypass architecture to a general design principle for efficient implementation of high-performance distributed applications. It also provides an example of a Medical Consultation Application to illustrate these concepts. Section 3 discusses the various design options and their performance-cost trade-offs and provides expressions for quantifying them. It is important to consider the effect of bypass failures on the performance of systems, particularly in distributed applications requiring guaranteed bounds on delays. Section 4 discusses the causes of failures and develops analytical expressions for quantifying the delays. The discussion includes the three cases --- equal priorities, priority to bypass and priority among bypass streams.

## 2. Application Software Bypass

Layering and modularization are essential to deal with software complexity. The key to achieving performance gains in modularized software is to distinguish the data flows between modules from the control flows, and to improve the performance of data flows. There are three complementary ways to achieve these improvements -- customized monolithic implementations, general purpose mechanisms especially with support from operating systems and general design principles directly applicable to application components. The application bypass architecture falls under the last category. The application is analyzed to identify the modules and the data flows. Opportunities for bypassing are identified by finding relationships such as concatenated flows, common operations and common origins and destinations across layers. These data flows are reduced into simpler merged flows with minimal number of operations. The logical conditions under which these simpler flows can be used are also derived from the state machines of the interacting modules. Then a bypass path guarded by the bypass test is constructed. Further optimization of the bypass can then be done applying the operating systems mechanisms mentioned above.

**Example of a Medical Consultation Application (MCA)**

To illustrate these concepts consider a Medical Consultation Application (MCA). Medical consultation and remote diagnostics are some of the prime motivators for multimedia teleconferencing systems [10]. It has been recognized that teleconferencing is not merely an application but a *paradigm* of communication for many divergent applications. There are many differences in details between a multi-party business conference, a tele-lecture and a medical consultation application. However, all of them have much in common in terms of the structure of communications. Hence, it is reasonable to structure a medical consultation application on top of a teleconferencing service.

**Application Modules of the MCA**

A typical session of a medical consultation application may involve a patient, physician, radiologist and one or more specialists. The interactions may include text, voice, video and high-quality images. While the video and voice are live, images are usually stored in some server, possibly at the site of the radiologist. It is reasonable to partition the system into subsystems such as user-interface manager, conference manager, multicast transport service and image data base manager (Figure 3). If the underlying communication system provides multicast transport support, then that layer is not part of the application but the application interacts with that layer through a service interface.

Each of the modules provides a set of services. For example, the user-interface module may have such services as initiating a discussion, sending information to the other participants and presenting information received from the other participants in a manner suitable to the role of the participant. A patient, physician, or radiologist each may need to have different presentation styles. In addition, access rights may be associated with these roles and may have to be applied at run time.

The conference manager has a more general notion of the structure and roles of the consultation. It may introduce roles such as organizer (initiator), manager (chair), ordinary participant, special invited participant and so on. It may provide services such as initiating a consultation, the assignment or election of persons for roles, managing the entry and exit of participants in the conversa-

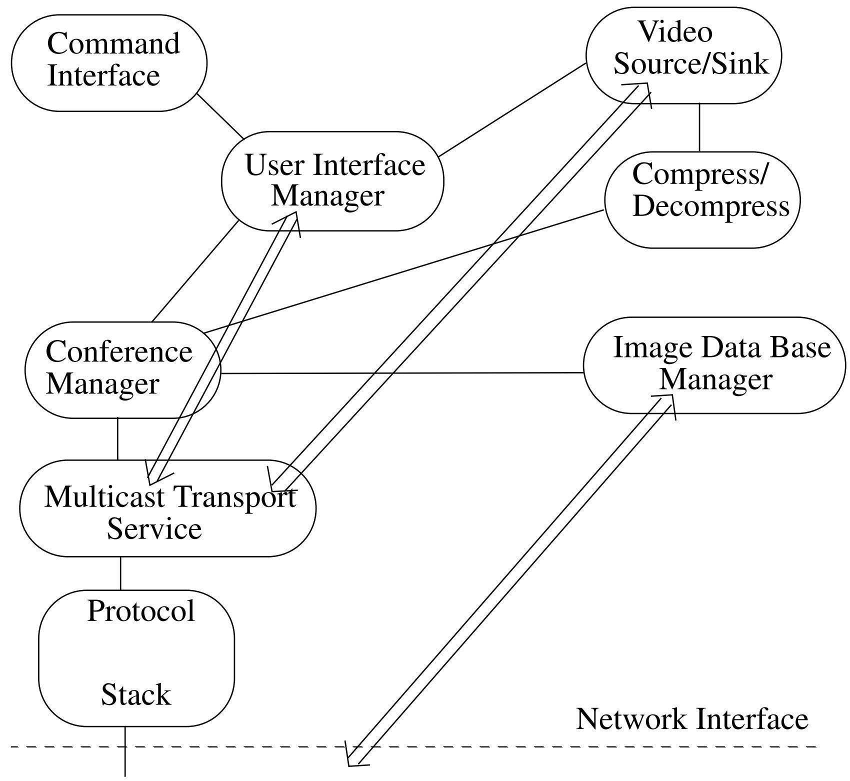


Figure 3. Modules for a Medical Consultation Application
(The double arrows represent paths which may be bypassable)

tion and floor control, apart from actual transmission of data [11].

The multicast transport layer provides services such as creation and destruction of groups, group membership management apart from transmission and reception of data.

If we treat the modules as independent while implementing them, it provides important flexibility in the design. However, it leads to redundant processing overheads. On the execution path of a single system operation, it often happens that the operations in one module are closely related to those in the next module along the path. Specifically, when an upper layer module is in data transfer state, it is normally the case that the lower layer modules are also in their data transfer states. These close relationships are exploited in a bypass path which efficiently combines the operations along the execution path.

**Implementation Possibilities**

Figure 3 contains several wide arrows representing data flows in MCA which provide opportunities for a bypass. They satisfy the criterion of high volume passing through several layers of applications, and simplification of processing in frequent but special cases. In the paths for video data for example the simplified processing could be a standard compression algorithm, and the bypass test could be passed by all video packets internal to a frame. Some bypass paths bridge several modules, which means they include functions from each one. These paths can be identified by considering the design trade-offs described in the next section. We can call these “customized bypasses.”

It is also desirable to consider standardized bypasses. These would be standardized modules which provide a defined, commonly-used fast-path service, such as a transfer from CD-ROM to IP/TCP packets, or from compressed video to a stated video display-buffer format. These optimized paths could be plugged together with full-service modules which handle exceptions and establish paths and groupings, to give a building-block approach to software engineering which also has good performance.

**Bypass Test Failures**

A bypass designed to transfer bulk data, along the lines of [4], uses a test which requires that each data unit conforms to a standard description (e.g. in size, encoding, and destination storage address). In a stream of image-data units to the conference manager, a control data unit (for instance, for floor management) would have to go to the general module logic rather than a bypass. When this happens, a rather longer processing operation involving several modules may follow, introducing a transient interruption or degradation to the Quality of Service (QoS) of the image-data stream. These transients, called “bypass glitches”, are considered in Section 4.

## 3. Execution Cost Trade-offs

Figure 4 illustrates the usual modular structure of software in an end-system. If a data-path lies through a sequence of several modules (illustrated in Figure 4), then bypass processing can be introduced in three different ways as described in Figure 5.

- In Figure 5(a) (“internal bypass”) each module handles its own fast path, internally.
- In Figure 5(b) (“communicating bypasses”) the fast path in one module accesses its continuation in the next directly, without a new test inside the second module.
- In Figure 5(c) (“integrated bypass”) the fast path is removed from the modules entirely into a separate integrated bypass module.

In module *i* the normal execution cost is $T_i$, and the fast path cost is $T_{Bi}$.

The execution cost implications of these three options will be summarized for an operation traversing a set of *n* modules. It is assumed first that a single test condition governs the special processing in all the modules, and this test can be applied in any module. The data stream is such that the test is true with probability $p_B$ as before. If we wish to add up the total execution cost it is useful to use the sums

$$T = \sum_{1}^{n} T_i, \quad T_B = \sum_{1}^{n} T_{Bi}$$

Also there is a cost of crossing a module boundary, of $T_{OH}$. This may represent procedure calling, object messaging or interprocess messaging overhead, depending on the design of the software. Finally there is a cost of applying a bypass test, of $T_{TEST}$.

By inspecting Figures 4 and 5 we see that without any bypass optimization the execution cost for *n* modules is $T + nT_{OH}$, and for the three options it is

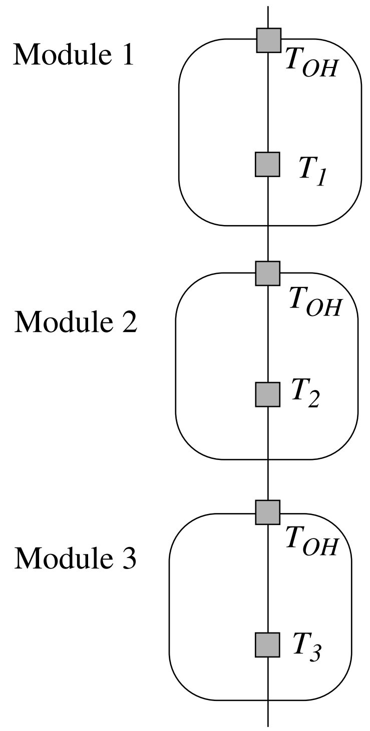


Figure 4. Flow Through a Modular Software System

- option (a), Cost = $T(1 - p_B) + T_B p_B + n(T_{OH} + T_{TEST})$
- option (b), Cost = $T(1 - p_B) + T_B p_B + T_{TEST} + nT_{OH}$
- option (c), Cost = $(T + nT_{OH})(1 - p_B) + T_B p_B + T_{TEST} + T_{OH}$

For example, with the parameter values (normalized to the value of *T*) given by

$$T_B/T = T_{OH}/T = T_{TEST}/T = 0.05; \; n = 6$$

the values for Cost/*T* are 1.3 for the original case without any fast path, and for the three options the values are:

*(a)* Cost/$T$ = $1.6 - 0.95 p_B$

*(b)* Cost/$T$ = $1.35 - 0.95 p_B$

*(c)* Cost/$T$ = $1.4 - 1.25 p_B$

Figure 6 shows these values plotted against $p_B$, showing the marked advantage of option *(c)* as $p_B$ approaches 1.0. With these parameters, as $p_B \rightarrow 1.0$ the workload cost in case *(c)* is only one quarter of that in case (a), and one eighth of that with no bypass. Also, the bypass has multiplied the workstation capacity by a factor ranging from 2 in case *(a)* to 8 in case *(c)*.

So far we have considered one bypass for one path threaded through a set of modules. There may be

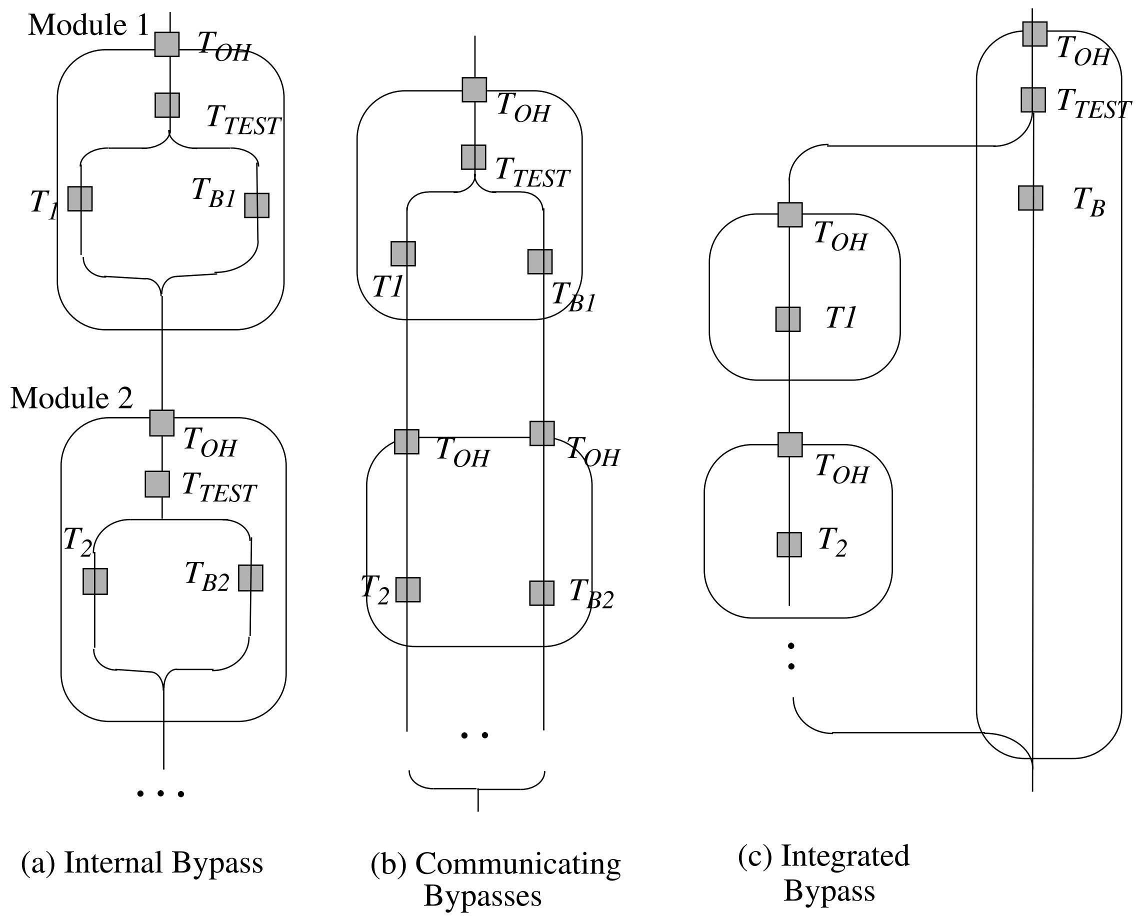


Figure 5. Bypass Paths in a Modular Software System

- other paths through the same modules, with a different bypass condition and different processing (for example, a different encoding/decoding step). The tests for the various bypasses could be combined, adding somewhat to the cost $T_{TEST}$,
- other bypass paths within the same module structure, with their own test points. (Figure 3, for example, indicates several distinct bypasses.)

Each bypass must be justified separately, and interaction between them is usually slight (consisting mainly of the possibility of combined bypass tests, as mentioned above).

## 4. Bypass Glitches

The main performance defect of a bypass is the “bypass glitch” which is a transient condition following a test failure. This places a transient in the otherwise smooth flow through the bypass, which may degrade the connection. When a bypass test fails, that data unit must be processed by the general version of the module or modules. A bypass test failure may be due to the end of the stage, to exceptional data, or to a communications error. We will assume that the following data unit, after waiting for the slow general processing, will pass the tests; that test failures are suffi-

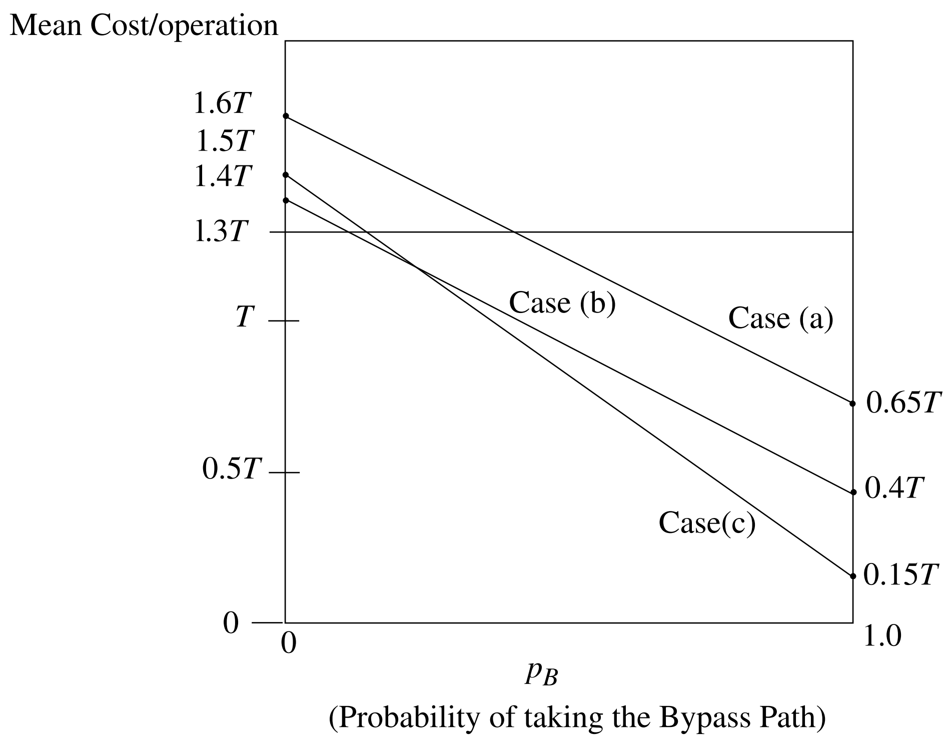


Figure 6. Execution costs vs $p_B$ (for 6 modules and $T_{OH} = T_{TEST} = T_B = 0.05T$)

ciently separated in time that the resulting transients do not overlap, and that the processing cost of both paths are deterministic. The glitch is the response to the slowdown for one data unit, and the backlog that builds up during that time. We will analyze the effect of glitches on quality-of-service (QoS) parameters.

We will consider communications processing as being carried out on several concurrent *streams*. In each stream some uniform operation (such as a video or graphics transfer) is carried out, for a period of time called a *stage*. We will consider one such stage, with bypassed operations, running perhaps in combination with other concurrent streams. Since a glitch gives a transient overload, a fluid approximation to the delay process [12] is appropriate for a first-order understanding of its effects. In a fluid approximation the total flow of work into the system for all streams, and the backlog of work to be done, is represented by a continuous deterministic flow. During a transient situation the delay values change with time. Consider a steady flow giving a total processor utilization of $\rho$, of which an amount $T_B/\tau$ is due to a particular bypassed stream with execution cost $T_B$ and arrival period $\tau$. At time 0, we have a pulse of work in this stream, of magnitude $(T - T_B)$, representing one bypassed operation of cost $T_B$ being replaced by one non-bypassed operation of cost $T$. (We will assume that all the additional work is at the CPU). During the remainder of the transient the CPU is continuously busy and new work is still arriving at rate $\rho$ units per sec, so the backlog declines at rate (1 - $\rho$)/sec, giving a work backlog transient as shown in Figure 7. Clearly the duration of the transient is $(T - T_B)/(1 - \rho)$. If the arrivals in our particular stream are spaced $\tau$ sec apart there are about $k = (T - T_B)/[\tau(1 - \rho)]$ operations during one transient.

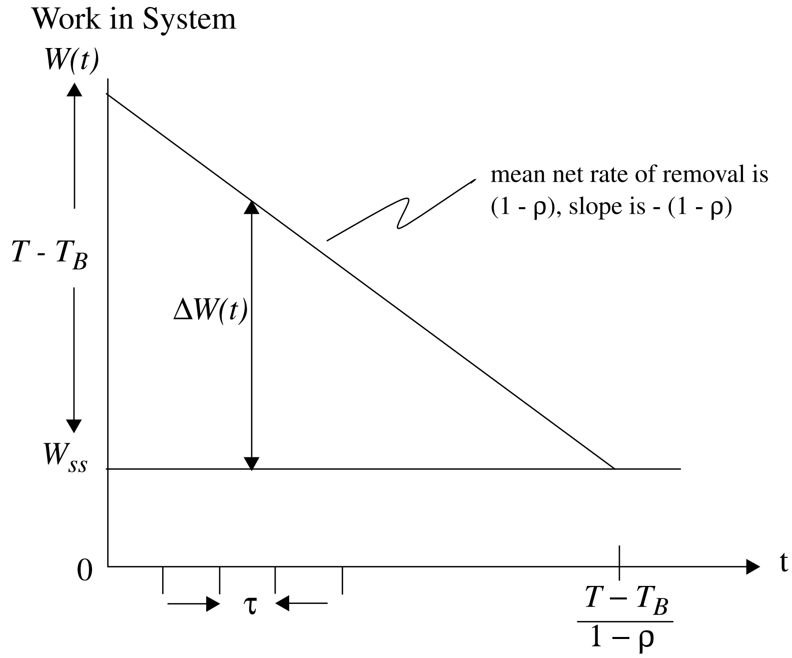


Figure 7.  Transient of Work in System for One Glitch

**Equal Priorities**

Consider the response-time consequences, first for cases where all work is of equal priority. Then the response time for the given stream is just the work-in-system at the arrival instant [13] plus the service time. The *additional* response time due to the glitch, for an arrival at time *t*, is the amount $\Delta W(t)$ shown in Figure 7. As already noted, there are about $k = (T - T_B)/[\tau(1-\rho)]$ responses during one transient, and from Figure 7 they have an average duration of $(T - T_B)/2$ .

During a long period of $K$ operations there are, on average, $(1-p_B)K$ glitches. The addition to the total response time due to glitches in the same stream is

$$(1 - p_B)Kk(T - T_B)/2$$

and the addition to the mean response time of the stream is

$$\Delta R_G = (1 - p_B)(T - T_B)^2/[2\tau(1-\rho)] \quad . \qquad \text{(Equal priorities)}$$

For example if $\rho = 0.7$, $(T - T_B) = 95$ msec, and $\tau = 15$ msec, then there are about 20 operations during one glitch and overall, $\Delta R_G \approx 1000 p_B$ msec.

Depending on how QoS is defined for a data stream, the important impact may either be an increased mean delay through the system, or operations which fail to meet deadlines. The effect $\Delta R_G$ on the mean delay as calculated above may be masked if the achieved end-to-end mean is normally less than the QoS target. However the figure calculated above serves as a pessimistic bound. Similarly, the effect on deadlines can be bounded by the number of operations affected by glitches, which during a long period of $K$ operations is $N_G = (1-p_B)Kk$, giving

$$P_G = \text{Prob}\{\text{operation misses deadline}\} \quad \leq (1-p_B)k = p_B(T-T_B)/[\tau(1-\rho)]$$

For the parameters given just above, this has value $21.3(1-p_B)$.

These equations show that the impact of bypass failures is considerably increased when the CPU is heavily loaded, in that the probability of a QoS impact is much larger than $(1-p_B)$ per operation, and the mean delay impact is much more than $(1-p_B)(T-T_B)$ .

The above analysis is highly simplified, so it was compared to results for more realistic simulated response times. Figure 8 shows response times of packets following a glitch. The simulation has $\tau = 0.5$, $T = 1.0$, $T_B = 0.05$, a background workload sufficient to make $\rho = 0.7$, and equal priorities. The arrival processes are Erlang-10, so they have some limited degree of random variation, the bypassed stream has deterministic service, and the background workload has Erlang-10 service. The figure shows the mean of 100 transients, with 90% confidence intervals, plotted against the prediction shown as a dashed line. The agreement is excellent, and was equally close for other parameter values.

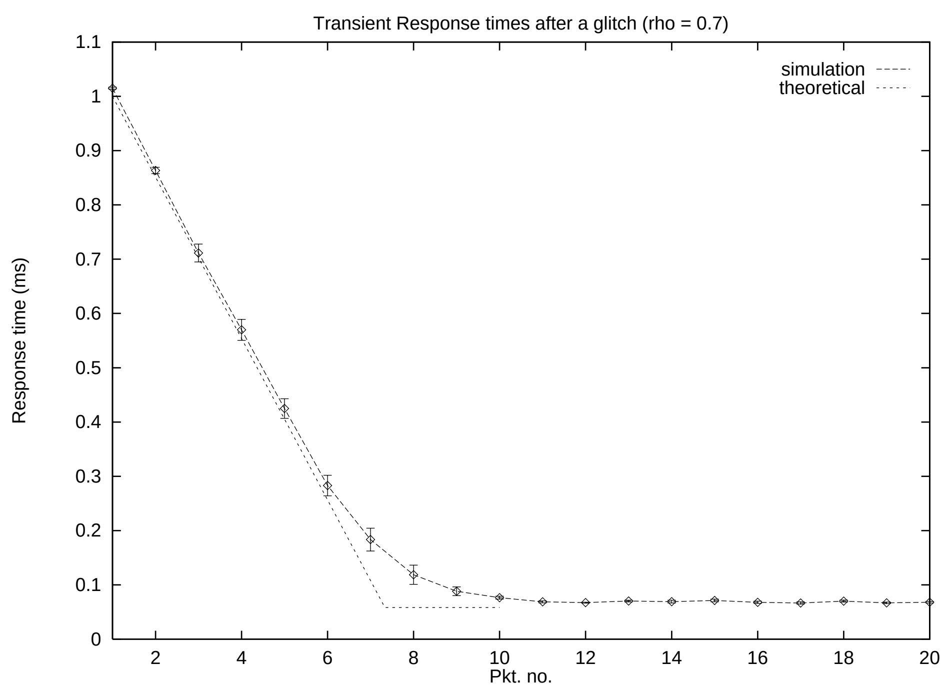


Figure 8. Transient Response times after a glitch ($\rho = 0.7$)
(Simulation vs the theoretical fluid approximation)

## Priority to the Bypass

The impact of glitches on the same bypassed stream may be reduced if a part of the CPU workload other than this stream can be preempted (by giving priority to the communication-oriented processing), or by modifying the recovery from a glitch (perhaps by dropping some data). If the bypassed stream with partial utilization $\rho' = T_B/\tau$ has preemptive priority over the remaining "background" workload, then it does not "see" the background at all. Therefore, the same formulae are applied with $\rho'$ replacing $\rho$. Thus

$$\Delta R_G = (1 - p_B)(T - T_B)^2 / [2\tau(1 - \rho')] \quad \text{(Priority)}$$

$$P_G = (1 - p_B)(T - T_B) / [\tau(1 - \rho')] \quad \text{(Priority)}$$

**Multiple Bypassed Streams: Glitch Effects**

The above analysis only considers one bypassed stream, among all the various workload components of the CPU. If there are several bypassed streams running at the same priority, a glitch in any one will affect the others. The glitch rate will be the sum of the rates for all streams. Applying a subscript $i$ for the $i$th stream, it has parameters $p_{Bi}$, $t_i$, $T_i$ and $T_{Bi}$. When all work has the same priority, glitches in stream $i$ contribute an amount which can easily be shown to be

$$\Delta R_{ij} = (1 - p_{Bi}) \frac{(T - T_{Bi})^2}{2\tau_i(1 - \rho)}$$

to the mean response time of stream $j$, so for any one stream $j$ the total addition is

$$\Delta R_j = \sum_i (1 - p_{Bi}) \frac{(T_i - T_{Bi})^2}{2\tau_i(1 - \rho)}$$

One bypass stream with low $p_B$ can negatively affect all of the processing on the end-system. This is the most serious problem with bypassing, and indeed with any kind of optimistic tuning of any component of work.

If the bypassed streams have priority over background processing and contribute a total of $\rho''$ to the utilization, the above expressions have $\rho''$ in place of $\rho$.

**Priorities Between Bypasses**

Priorities can reduce the impact of glitches on a high-priority stream, at the cost of worse impact on other bypassed streams. Consider two streams with the same bypass failure probability, arrival time and service times ($p_B$, $\tau$, $T$, $T_B$), but stream 1 has pre-emptive priority over stream 2, and both have priority over the background. During a stream-1 glitch, stream 1 only waits for its own component of load, with the backlog $\Delta W_{11}(t)$ illustrated in Figure 9. It returns to normal after time $(T - T_B)/(1 - \rho')$, but during all this time stream 2 is stopped by the priority system. Remember that $\rho' = T_B/\tau$.

After a stream-1 glitch, stream-2 operations are totally blocked until the stream-1 backlog is cleared (i.e. until $(T - T_B)/(1 - \rho')$ ). At this point there is a stream-2 backlog of $\rho'(T - T_B)/(1 - \rho')$ , which must be cleared; the backlog is reduced at rate $(1 - 2\rho')$ . Thus the stream-2 backlog is $\Delta W_{12}(t)$ as shown in Figure 9.

For a low-priority stream like stream 2 the response-time is not the backlog at the arrival instant; it includes some high-priority arrivals that come after but leave first. If $\Delta W_{12}(t)$ is the backlog, an arrival at time $t$ has response time

$$R_{12}(t) = \Delta W_{12}(t) + \rho' R_{12}(t) = \Delta W_{12}(t)/(1-\rho') \quad .$$

Thus for the right side of Figure 9, for $t < (T-T_B)/(1-\rho')$, this makes the response time of an arrival at time $t$ equal to:

$$\begin{aligned} R_{12}(t) &= \left[\rho'\frac{(T-T_B)}{1-\rho'} - (1-2\rho')\left(t - \frac{(T-T_B)}{1-\rho'}\right)\right]\frac{1}{1-\rho'} \\ &= [(T-T_B) - (1-2\rho')t]/(1-\rho') \end{aligned}$$

which reaches zero (ending the glitch) at $t = (T-T_B)/(1-2\rho')$.

For $t < (T-T_B)/(1-\rho')$, $R_{12}(t)$ is the time until the stream-1 backlog clears, plus a term due to the stream-2 backlog at time $t$, which is $\Delta W_{12}(t) = \rho' t$. Thus:

$$\begin{aligned} R_{12}(t) &= \frac{T-T_B}{1-\rho'} - t + \frac{\Delta W_{12}(t)}{1-\rho'} \\ &= [T-T_B-(1-2\rho')t]/(1-\rho') \end{aligned}$$

which makes $R_{12}(t)$ a straight line from $(T-T_B)/(1-\rho')$ at $t = 0$ to zero at $(T-T_B)/(1-2\rho')$. Figure 10 shows simulation results for $R_{11}(t)$ and $R_{12}(t)$, for a case with $\rho' = 0.3$, $\tau = 0.167$, $T = 1.0$, $T_B = 0.05$.

The mean response-time increment to stream 2 can be found, after some manipulation, to be:

$$\Delta R_{G,12} = (1-p_B)\frac{(T-T_B)^2}{2\tau(1-\rho')^2(1-2\rho')} \quad .$$

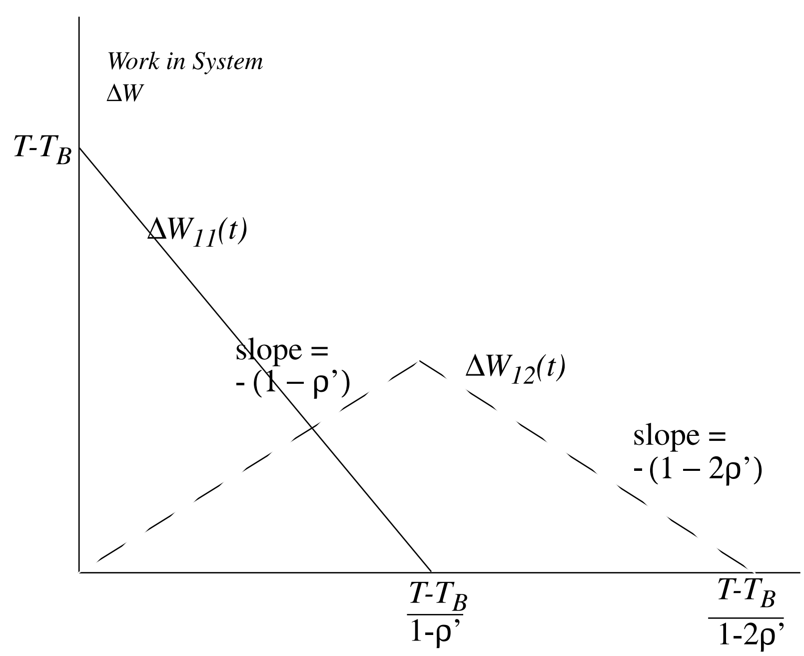


Figure 10. Response to a stream-1 glitch.

A stream-2 glitch gives a simpler analysis. Stream 1 is unaffected; stream 2 sees a backlog $\Delta W_{22}(t)$ starting at $(T - T_B)$ and dropping to zero at $t = (T - T_B)/(1 - 2\rho')$ . So:

$$\Delta W_{22}(t) = (T - T_B) - t(1 - 2\rho')$$

$$R_{22}(t) = \Delta W_{22}(t)/(1 - \rho')$$

$$\Delta R_{G,22} = (1 - p_B)(T - T_B)^2/[2\tau(1 - \rho')] \quad .$$

Overall then the effect on the mean response time is

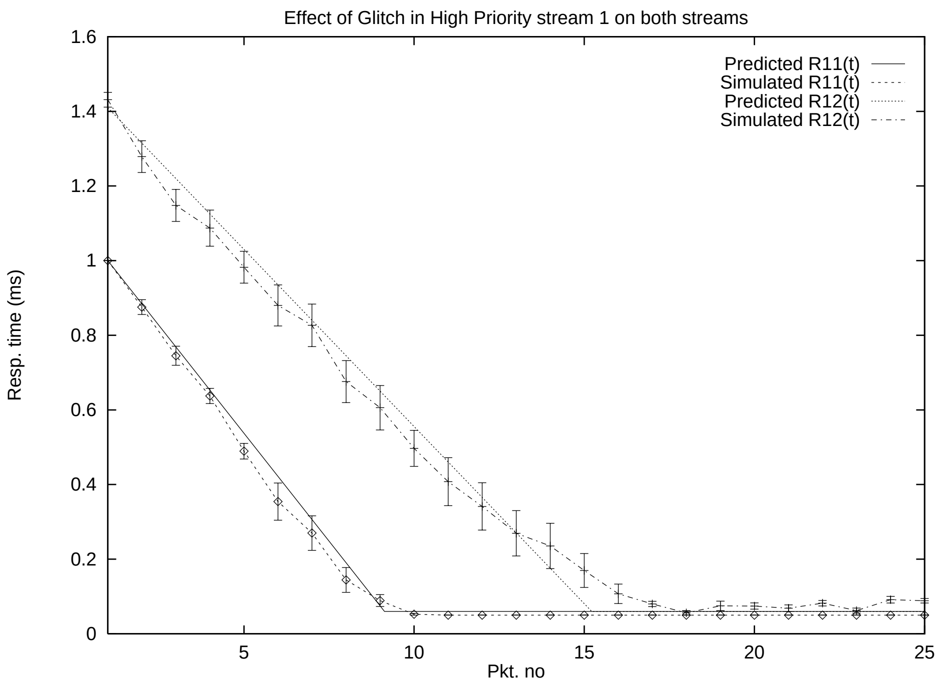


Figure 11. Response Times of Both Streams after Glitch in Stream-1

$$\Delta R_{G,1} = p_B(T - T_B)^2/[2\tau(1 - \rho')]$$

$$\Delta R_{G,2} = \Delta R_{G,12} + \Delta R_{G,22}$$

$$= (1 - p_B)(T - T_B)^2(2 - \rho')/[2\tau(1 - \rho')^2(1 - 2\rho')] \quad .$$

As an example, suppose $\rho' = 0.3$, $\tau = 0.5$, $T - T_B = 0.95$, then $\Delta R_{G,1} = 1.29(1 - p_B)$, and $\Delta R_{G,2} = 7.83 p_B$. If the two streams have equal priority, however, $\Delta R_G = 4.512(1 - p_B)$.

## 5. Conclusions

We have identified the possibility for exploiting the bypass idea, previously described for layered protocols, in general modular communications software such as a medical consultation system. Calculations for the execution-cost advantage of a bypass were given, for various alternative bypass architectures. Substantial cost advantages, or one or more orders of magnitude, are possible.

A *glitch* in a data stream is a transient condition resulting from the need to apply the full processing logic to the data stream. If the end-system is heavily loaded, glitches may have significant impact on the quality of service parameters of a stream. We considered the effect on the mean end-system response, and the number of data units that are affected. The effect on certain streams can be reduced by the use of priorities, and this was investigated. When a bypass is designed the frequency and magnitude of glitches must be investigated as part of the assessment of its performance potential. If bypassing becomes important in practice it may be necessary to include QoS parameters that describe the relative immunity of the stream, as required by the user.

## Acknowledgments

This research was supported by the Telecommunications Research Institute of Ontario, and by the Commonwealth Scholarships.

## References

[1] C.M. Woodside, K. Ravindran and R.G. Franks, "The Protocol Bypass Concept for High Speed OSI Data Transfer", in *Proc. of Second IFIP International Workshop on Protocols for High-Speed Networks*, November, 1990 (Published by North-Holland as *Protocols for High-Speed Networks II*, 1991, pp. 107-122.)

[2] Y.H. Thia and C.M. Woodside, "High Speed OSI Protocol Bypass Algorithm with Window Flow Control" in *Proc. 3rd IFIP Workshop on Protocols for High Speed Networks*, Stockholm, May, 1992.

[3] Y.H. Thia and C.M. Woodside, "A Reduced Operation Protocol Engine (ROPE) for a Multiple-Layer Bypass Architecture" in *Proc. 4th IFIP Workshop on Protocols for High Speed Networks*, Vancouver, August, 1994, pp.203-218.

[4] C.M. Woodside and Y.H. Thia, "A Parallel Optimistic Bypass Architecture (POBA) for High-Speed Bulk Data Protocol Processing", Report SCE-94-03 of the Dept. of Systems and Computer Engineering, Carleton University, November, 1993.

[5] D.Clark "The structuring of systems using upcalls", In *Proc. 10th ACM Symposium on Operating System Principles*, pp. 171-180, December, 1985.

[6] D.D. Clark, V. Jacobson, J. Romkey and H. Salwen, "*An Analysis of TCP Processing Overhead*", IEEE Communications Magazine, vol. 27, no. 6, pp. 23-29, June, 1989.

[7] D.D.Clark and D.Tennenhouse, "Architectural Considerations for a New Generation of Protocols", in *Proc. ACM SIGCOMM*, pp. 200-208. 1990.

[8] P.Druschel, M.B.Abbot, M.M.Pagels and L.L.Peterson, *"Network Subsystem Design*", IEEE Network,7(4) July, 1993.

[9] J.Pasquale, E. Anderson and P.Keith Muller, *Container Shipping - Operating System Support for I/O-intensive Applications*, IEEE Computer, March 1994, pp. 84-93.

[10] L.Orozco-Barbosa, A.Karmouch, N.D. Georganas and M.Goldberg, "*A Multimedia Inter-hospital Communications System for Medical Consultations*", IEEE Journal on Selected Areas in Communications, vol.10, No.7, September 1992, pp.1145-1157.

[11] ITU Study Group 8, *T.GCC: Generic Conference Control for Audiovisual and Audiographic Terminals and Multipoint Control Units*, May 23-5, 1994 (ITU Telecommunications Standardization Sector).

[12] G.F. Newell, "*Applications of Queueing Theory*", Chapman and Hall Ltd., London, England, 1971.

[13] L. Kleinrock, "*Queueing Systems*", vol. 1, John Wiley and Sons, 1976.