Improving Protocol Performance by Dynamic Control of Communication Resources

Daniela Ivan-Roşu Karsten Schwan
{daniela, schwan}@cc.gatech.edu

GIT–CC–96-04

February 1996

Abstract

A problem frequently faced by complex distributed applications is to control the interaction of their communication and computational activities such that they jointly adhere to desired performance and timing requirements. This research presents the COMM^{adapt} communication infrastructure that can cope with dynamic variations in user programs' processing and QoS* requirements, by permitting the on-line adaptation of a protocol's resource usage to currently available resources and application requirements. A key feature of COMM^{adapt} is its dynamic (auto-)configurability, which is its support of on-line configuration transparent to application programs. Such configuration is performed by a heuristic that accommodates changes in a connection's resource requirements by reallocating resources based on its knowledge of actual resource usage of the active connections in the system. The heuristic's design and implementation are based on extensive investigations of the manner in which the assignment of protocol tasks to underlying processing resources can influence communication latency and throughput.

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332–0280

*Quality of Service

Improving Protocol Performance by Dynamic Control of Communication Resources

Daniela Ivan-Roşu* Karsten Schwan

{daniela, schwan}@cc.gatech.edu

Abstract

A problem frequently faced by complex distributed applications is to control the interaction of their communication and computational activities such that they jointly adhere to desired performance and timing requirements. This research presents the COMM^{adapt} communication infrastructure that can cope with dynamic variations in user programs' processing and QoS[1] requirements, by permitting the on-line adaptation of a protocol's resource usage to currently available resources and application requirements. A key feature of COMM^{adapt} is its dynamic (auto-)configurability, which is its support of on-line configuration transparent to application programs. Such configuration is performed by a heuristic that accommodates changes in a connection's resource requirements by reallocating resources based on its knowledge of actual resource usage of the active connections in the system. The heuristic's design and implementation are based on extensive investigations of the manner in which the assignment of protocol tasks to underlying processing resources can influence communication latency and throughput.

1 Motivation

For parallel or real-time applications, parameters of communication performance like end-to-end latency, delay jitter or loss probability, are as important as total communication throughput. For instance, the performance of scientific parallel codes has been shown to improve when computations and their associated communications are scheduled jointly, using compiler information[6] or using runtime knowledge of communication delays [25]. Similarly, both the performance and the predictability of real-time applications are improved if bounds on communication delays can guaranteed [23, 1].

This paper focusses on one element of end-to-end communication delays: the overheads of *communication processing*. Specifically, we posit that the reduction of such overheads requires the dynamic control of the resources used in communication processing. Dynamic control has become necessary due to the complexity and runtime variability of modern network platforms and of the increasingly complex multi-threaded and distributed applications using these platforms. In these contexts, it is too difficult to estimate runtime communication requirements and model network resource availability prior to program execution. In addition, overly pessimistic assumptions will have to be made if programs must specify connection needs at the time of connection establishment.

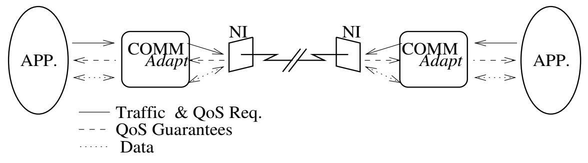


Figure 1: COMM^{adapt} interfaces the application(APP) and the network interface (NI)

This paper's solution to the problems presented above is twofold: (1) we attempt to allocate resources to satisfy stated communication requirements, but (2) such allocations are reviewed and revised while connections are active, thereby dealing with runtime variations in actual vs. estimated requirements and in required vs. available resources. Accordingly, the COMM^{adapt} software library presented and evaluated in this paper implements a communication infrastructure (CI)

*Funded in part by DARPA through the Honeywell Technology Center under Contract No. B09332478

[1]Quality of Service

(see Figure 1) that contains the mechanisms necessary for the dynamic allocation of resources to communication processing tasks. These mechanisms allow the application, the CI, or both to make runtime decisions in order to accommodate observed changes in communication requirements or resource availability. The goal of such decisions is to guarantee certain levels of service, improve the utilization of host and network resources, or prevent service disruption due to unexpected events. In addition, protocols implemented with COMM^{adapt} may be tailored to satisfy application-specific communication needs by their flexible composition from sets of protocol modules[10].

By permitting the CI itself to make runtime resource allocation decisions, (1) the reactivity of COMM^{adapt} protocols is improved, (2) application programs need not track changes in their own communication behaviors, and (3) programs also need not have knowledge of current resource availability and of the behavior of other applications using the same resource pool. In contrast, the CI may make allocation decisions based on current communication behavior and on actual levels of resource utilization and availability. This *auto-configuration* attribute of the CI distinguishes our research from other current and past work on configurable communication systems [10, 17, 22, 26, 27, 34, 16]. However, note that auto-configuration is also used in recent work that deals with dynamic changes in video traffic mapped to the QoS guarantees offered by ATM networks[30], and that it has proven useful in past research on guaranteeing the predictable behavior of real-time applications[3, 8].

The policies and algorithms making autonomous runtime resource allocation decisions for communication protocols comprise COMM^{adapt} 's *auto-configuration mechanism.* In the prototype presented in this paper, auto-configuration controls CPU resources. Namely, CPU resources are allocated at runtime with a load assignment heuristic that maps communication requirements to alternative configurations of protocol modules. Such decisions are triggered by stated or observed changes in program behavior (e.g., unforeseen rate increases or connection establishments) or by modifications in resource availability (e.g., due to varying network loads). Accordingly, we define the term *CI configuration* to represent currently available resources and their mapping to current connections supported by the protocol stack. Similarly, the term *configuration decision* denotes a resource allocation decision.

The benefits from using COMM^{adapt} are apparent in multiprocessor environments, where it is easy to guarantee dynamically negotiated CPU reservations. Systems permitting such reservations include commercial multiprocessors like SUN Solaris SMPs and supercomputers like the Convex machines, and they include systems offering real-time capabilities like the Rialto Operating System[14], the YARTOS kernel[11, 12], and RT-Mach with the processor capacity reservation-based scheduling described in [18]. This paper's measurements are performed on a commercial parallel machine, a KSR-2 supercomputer running multiple communicating processes each using the COMM^{adapt} prototype, with the KSR's interconnection network emulating the underlying network typically employed for inter-process communications. Since COMM^{adapt} is implemented with a portable user-level threads library[20], it can also be run on other parallel machines, including SUN Solaris and SGI multiprocessors.

The development of COMM^{adapt}has been driven by our experiences with both high performance and real-time applications [15, 19]. However, we expect to derive benefits from COMM^{adapt} s use for a wide variety of applications, including: (1) CSCW[2] applications in which multiple communication streams exhibit dynamic traffic and QoS characteristics and possibly, permit trade-offs among these streams' acceptable QoS levels; (2) video or data servers, where the aim is to maximize the number of connections with high throughput and predictable jitter; and (3) distributed real-time applications, where multiple streams of data are collected and processed subject to specific and possibly dynamic rates and timing requirements.

The remainder of this paper first outlines the COMM^{adapt}software architecture and its prototype (see Section 2). Next, insights on the design of the load assignment heuristics used for auto-configuration are derived from studies of the influence of alternative protocol configurations on communication processing latencies and throughput (see Section 3). COMM^{adapt}'s configuration mechanisms, an analytic model capturing suitable performance characteristics of COMM^{adapt} -built pro-

[2]Computer Supported Collaborative Work

tocols, and an auto-configuration heuristic based on this model are presented in detail in Section 4. A synthetic program derived from the aforementioned video server example is used to illustrate the functionality of the auto-configuration mechanism. COMM^{adapt}'s contributions are reviewed in comparison with related research in Section 5. Remarks on our future work conclude the paper (see Section 5).

2 The COMM^{adapt} Communication Infrastructure

2.1 The Software Architecture

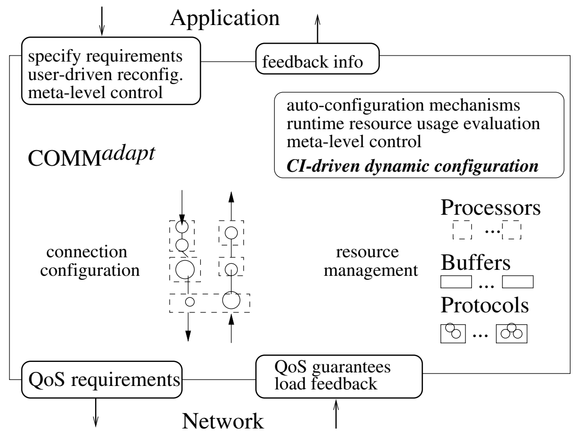


Figure 2: COMM^{adapt} architecture

The flexibility of a CI strongly influences the set of requirements it can accommodate. Consequently, a key issue in COMM^{adapt}'s design is to support flexibility in composing and configuring protocol stacks, and in managing resources and guarantees. COMM^{adapt}provides mechanisms for such configuration at the time of connection establishment, for user-driven reconfiguration at any time during the life of a connection, for specification of connection requirements, and for resource management (see Figure 2). The need for such mechanisms has already been established by previous work in configurable communication systems [10, 22, 27, 34]. Due to our interest in auto-configuration, the specific research issue addressed by COMM^{adapt} is the difficulty of accurately estimating and then responding to the runtime behavior of complex applications with numerous communication streams. Namely, the *dynamic configuration mechanisms* of COMM^{adapt}wish to address situations in which the actual connection requirements exceed those initially stated, but may be satisfied by use of spare resources recovered from other connections. This should result in improved resource utilization as it saves end users from using overly conservative estimates of their communication needs.

The COMM^{adapt} architecture consists of the following components:

- *Auto-configuration mechanisms*, which enable CI-level decisions about resource allocations to individual connections, based on protocol- and resource-specific algorithms/heuristics and transparent to applications. The goal of such configuration decisions is to satisfy each connection's performance needs. The runtime overheads of decision-making are kept within acceptable limits by trading off optimality for decision speed. In the future, we may also consider heuristics that learn from previous decisions and application behaviors.
- *Runtime usage evaluation mechanisms*, which provide information on the actual level of service observed by each individual connection. This information is used for auto-configuration, and it may be obtained from: (1) monitoring CI-internal activities like the time spent in given modules, the total message latency being experienced, etc., (2) network feedback, and (3) mechanisms specific to certain communication protocols in the configuration.
- *Configuration enactment mechanisms*, which instantiate configuration changes. The efficiency of such enactment strongly depends on the flexibility of the CI architecture and of its interface to the underlying resources.
- *Meta-level configurability control mechanisms*, which serve to adapt the decision mechanism (e.g., algorithms for resource management, decision-making policies) to specific applications or sets of requirements [3]. For example, an application may prefer that for specific connections, the decision algorithm focus on delay rather than on bandwidth optimization, or the CI itself may vary the quality and promptness of its decision making, by adjusting the amounts of information being monitored or the periodicity

of status checking.

The main components of the COMM^{adapt} software architecture together with its interfaces to the network and to application programs are depicted in Figure 2. Its current prototype providing auto-configurability and using the mechanisms described above is presented next.

2.2 Prototype Implementation

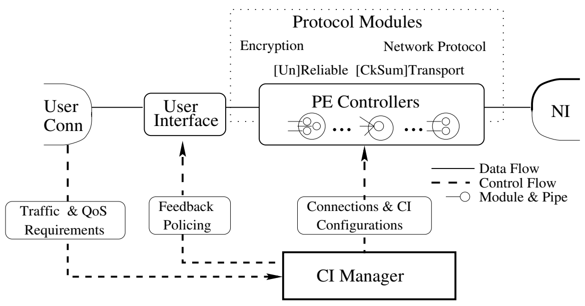


Figure 3: COMM^{adapt} prototype

The COMM^{adapt} prototype (see Figure 3) is implemented as a user-level library on Unix machines, and it is evaluated on a KSR-2 parallel supercomputer (a CC-COMA shared-memory multiprocessor, with a sequentially consistent shared address space and a 1GBytes/sec interconnect) over an emulated network. This environment is appropriate for studying the effects of CI configuration on communication performance for several reasons, including (1) the application programs performing send/receive operations may be isolated from other activities in the system, from the communication infrastructure and from the emulated network since the KSR-2 machine permits processor reservation, and (2) the emulated network itself offers behaviors observable in actual networks, such as asynchrony, actual communication latencies, etc.

The network environment, also called in the following "the network interface" (NI), consist of the network adapter emulated in software on an independent processor and the network fiber emulated by the KSR's memory and interconnect. The adapter offers a direct interface to the user space as done by most adapters with embedded processing resources [4, 5]. The interface available at the application level follows the model of the high performance ATM interface described in [24]: zero copy, gather/scatter facilities, early demultiplexing, per connection send and receive request queues, request driven receive (incoming messages are dropped if no posted receive). The network fiber is emulated by a set of shared memory buffers mapped into the sending and receiving address spaces. A point-to-point transmission is emulated by a set of remote memory operations that generate actual communications across the interconnect. For such a transmission, we use a "physical frame" with a fixed header specifying the destination socket and the payload length. For a send/receive operation, the network adapter has to "acquire" (lock) the channel, write/read the frame, and, finally, "release" (unlock) the channel. Checking the channel for an incoming message is a lock free operation. In this implementation of the emulated network offers an effective peak rate of about 68 MBits/sec.

The main components of the COMM^{adapt} prototype are: (1) the protocol library, which provides the building blocks (i.e., *protocol modules*) for constructing protocol stacks; (2) the "user interface", which controls the interaction with the application layer; (3) the communication task management, which includes mechanisms for inter-module communication (*pipes*) and for concurrent execution of protocol modules (*PE*[3] *Controllers*); and (4) the configuration mechanisms, which support application and COMM^{adapt} driven resource allocation decisions.

The protocol library includes the protocols required by the applications using COMM^{adapt}. In the current prototype, this library includes protocols for reliable transport (RTP), unreliable transport (UTP), fragmentation (NP), and encryption (EP) (in both serial and parallel implementations).

A COMM^{adapt} protocol is defined by a set of *modules*, each offering a specific service. The granularity of decomposition is implementation-specific. For instance, the UTP has only two modules, one for receives and the other sends, while the RTP has an additional module for handling ACKs. All modules in the library should have the same input interface (i.e., message connection, protocol state and CI state descriptors). To enable the evaluation of its per connection CPU requirements, each protocol module is described by the following overheads: null message processing, locally and remotely buffered 1KByte-message processing, and

[3]Processing Element

critical section processing.

The COMM^{adapt} prototype uses the layered protocol model. Protocol stacks are composed at the time of connection establishment, based on each connection's specification. The composition process consist of linking together the appropriate protocol modules in the order in which they should process a message. COMM^{adapt} 's protocol stacks are non-multiplexing. This preserves the connection identity at all levels of protocol processing, thereby enabling better support for connection-specific QoS [33] and for flexible composition of application-specific protocols.

The implementation of the transport protocols in the COMM^{adapt} library is similar to the TCP/IP suite with several exceptions: (1) the unreliable transport (UTP) is connection oriented; (2) the reliable transport (RTP) is message oriented, with selective retransmission and without adaptable window in the flow control mechanism. These differences are required by QoS management or were considered to be more appropriate for the emulated network environment in which the COMM^{adapt} prototype is exercised.

The Encryption Protocol (EP) has an original implementation: the actual message processing may be serial or data-parallel according to the message length and the number of assigned Slave modules. The parallel processing doesn't incur any additional message copy and complies with the pipeline paradigm: The Master module specifies the message chunks to be process by the Slave modules and the last Slave to touch a message forwards it to the next module on the connection pipeline.

The modules in the prototype library are profiled in terms of their execution times in Table 1. The results presented in this table validate that the implementation of our protocols is reasonable with respect to their CPU requirements. Namely, processing overheads are relatively small for low-level protocol tasks like 'unreliable transport', overheads are dominated by user and network interfaces and message copying (see column 'extra cost per 1KByte'), and presentation level processing costs far exceed lower-level costs (e.g., see the 'encryption' costs). Such characteristics show that the COMM^{adapt}prototype is a suitable instrument for studying the effects of CI configuration on communication performance.

Module/operation	0 byte message (μsec)	extra 1KBytes (μsec)
NP In	62	0
NP Out	45	0
UTP In/Out	7	120
RTP In	62	120
RTP Out	27	120
RTP	70	0
Proxy UI, In	125	7
Proxy UI, Out	92	7
Remote UI, Out	14	0
Remote UI, In	21	0
EP Slave	1800	10000
Pipe Get No Locking	20	–
Pipe Get Locking	38	–
Send to different PE	21	–
Send on the same PE	4	–

Table 1: Processing overheads in COMM^{adapt}

The user interface (UI) implementation is designed for a multi-processor environment. Since it is likely that application- and communication-related processing are assigned to different processors, the UI is divided into a proxy and a remote component, which may execute on the same or on different processors. The proxy UI copies data between application and CI buffers, and always executes on the same processor as the user thread that issued the I/O request. The remote UI implements the remaining user interface functionality, such as monitoring and usage control, and it may execute on any available processor.

The interface offered to the user level is similar to the BSD socket interface. The main difference is related to connection establishment, where COMM^{adapt} allows to specify the traffic and QoS parameters.

Communication task management. The *PE Controllers* are mapped one-to-one to the processors to which COMM^{adapt} is assigned. They control the execution of the protocol modules associated with each active connection. The *pipes*, the inter-module communication links, are repositories for service requests with locking strategies that are dynamically adjusted to the current number of writers and readers. Namely, no locking is needed on write/read if there is only a single writer/reader.

A module receives its inputs from a pipe. The association between a protocol module and its input pipe defines a *module instance.* A module instance is shared by multiple connections. However, multiple instances of the same protocol module may exist in the system. In addition, one module instance may be assigned to several controllers at the same time, thereby enabling concurrent protocol execution. The flexibility in mapping module instances to controllers is exploited heavily by COMM^{adapt} s dynamic configuration mechanism.

A PE Controller executes a protocol module only when inputs are available on its corresponding pipe. In the current implementation, PE controllers follow a round-robin policy in checking pipes for available inputs, except when a request is *shepherded.* Namely, if the source and the destination modules are assigned to the same processor, then the destination module is executed as soon as the source has finished processing. By ignoring message priorities and deadlines, this implementation of PE controllers offers low and predictable overheads of message scheduling and pipe operations. More importantly, shepherding enables a configuration decision maker to control throughput and message latency, by controlling the granularity of protocol stack decomposition across the available processors. The pipe and scheduling overheads presented in Table 1 demonstrate the benefits of adaptive locking (much costly pipe operation if locking) and the low costs of protocol modularization when combined with shepherding.

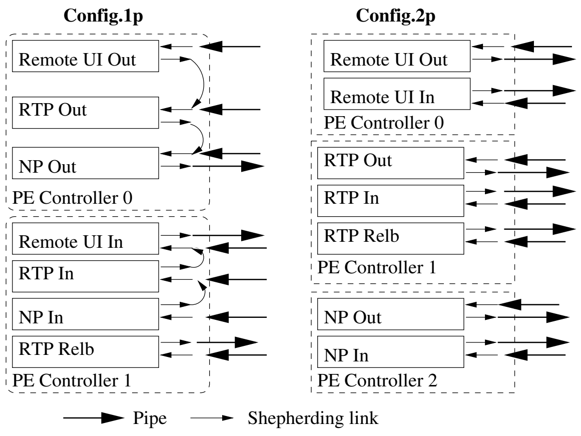


Figure 4: Examples of protocol stack assignments for a reliable connection (UI, RTP, NP)

Figure 4 illustrates two examples of how a protocol stack for unreliable communications (UI, UTP, NP) may be assigned to PE Controllers. These two assignments have different performance potentials, despite their identical processing requirements: Config.1p is characterized by a smaller latency because communication requests flow immediately across shepherding links instead of waiting in pipes for round-robin service. In contrast, config.2p, has a larger latency but offers a higher throughput, due to its improved per-stream load balance (see Section 3).

The configuration mechanisms implemented in the CI Manager (CIM) include the dynamic establishment of application-specific protocol stacks, management of processing capacity, performance control, feedback to the user, and support for both application-driven and auto-configuration. The main activities of the CIM are:

- connection establishment and user-level modification of connection specifications, including (1) the negotiation of QoS guarantees with the peer CIM and the underlying network, and (2) the decision of the configuration changes necessary to accommodate the new requirements; and
- on-line auto-configuration, including: (1) the periodic checking of the information collected at various points in the CI, and from network feedback, (2) the comparison of actual performance with the QoS and traffic requirements, and (3) decisions about the configuration changes that are necessary to accommodate any detected divergence.

Configuration decisions are enacted by informing PE Controllers of changes in their sets of serviced pipes. The actual changes are done independently by each PE Controller affected by the decision, without disturbing the processing of the messages currently in the system.

Important parameters of the CIM's activity are the frequencies of sampling and checking actual service levels, and the promptness with which the CIM activates the decision procedure when poor performance is detected. These parameters have major effects on the system's reactivity to unexpected events and on its overall performance. More specifically, the *activation delay* is the amount of time for which poor performance should be noticed in order to have the CIM trigger the decision procedure. The parameters controlling this delay are set by the meta-level control procedure in COMM^{adapt}. Their settings should result in low reaction times while avoiding reactions to meaningless short time variations, and retaining high performance in communication processing.

To summarize, the COMM^{adapt} prototype enables the construction of flexible protocol stacks, the on-line evaluation and monitoring of resource requirements and service qualities, and the accommodation of unexpected changes in connection requirements. In addition, the implementation choices made with respect to pipe management and scheduling (i.e., adaptive pipe locking and shepherding) improve the performance and predictability of communication processing.

3 Configurations and Communication Performance

The experimental results presented in this section provide insights into the ability of certain configurations (i.e., protocol modules assignments to processors) to influence performance metrics like maximum message rate, latency, predictability, and the efficiency of connections. More specifically, we are interested in how to achieve high throughput and resource utilization while keeping waiting times within predictable limits. The configurations considered in this study are (see Figure 5):

- *connectional parallel* connections (config.c), where a processor is handling only the messages of its assigned connections [29];
- *message parallel* connections (config.m), where any processor may process (completely) any message (from any connection) [29]; and
- *quasi-pipelined* connections (config.1p, config.2p, config.4p), where a processor is assigned one or more protocol modules and their processing is based on shepherding. In the classical pipeline [29], each protocol module is assigned its own processor. This makes the achievable peformance (i.e., throughput, latency) fixed with the implementation specific protocol decomposition. In contrast, with a quasi-pipelined approach, the implementation specific decomposition and the achievable performance are decoupled: with the same implementation, better performance may be achieved by appropriately reducing the actual granularity of decomposition by grouping modules.

These configurations are chosen to cover a range of different expected throughput and latencies. Differences are primarily due to changes in the distribution of processing loads across processors. For instance (1) the larger the difference between the maximum and minimum processor loads, the larger the waiting time in the CIM and (2) the better balanced the load, the better the throughput.

All of the experiments described in this section use multiple identical, concurrent connections between two "hosts", which are represented by synthetic applications running on sets of processors on different rings of the KSR2 machine. The application and the CI are assigned to disjoint processor sets, and the connections are independent of each other (i.e., each application thread involved in is assigned its own processor). Connection behavior and requirements do not vary during an experiment.

The *metrics* of interest are: connection throughput, message latency (i.e., for the outgoing path, this is measured from the time the message is copied into the CI buffers until it is sent to the CI), and efficiency (overall throughput with the number of processors). Each experiment runs until the metrics of interest reach the 95% confidence interval. Consistent with other studies reported in the literature[28, 21], the following *factors* are varied across experiments, : (1) protocol task assignments (see Figure 5), (2) number of concurrent connections; (3) message size, (4) protocol processing requirements (reliability, checksum, encryption), and (5) transmission rate (bounded 10Mbits/sec $\pm$ 0.05% or unbounded). *Fixed parameters* are: the size of processor pool for message parallel configurations, and the buffer pool and window size.

Experimental results are consistent with previously reported work, including the results in [28, 21]. For instance, at high communication rates, a message parallel configuration results in higher overall performance than a connectional parallel one (see Figure 7). More importantly, our experiments offer a number of basic insights directly related to our interest in dynamic QoS control. These insights are described next.

Latency ↓ / Max.PE Load ↑

	PE 0	PE 1	PE 2	PE 3-4
Config.c	..... Connectional parallelism			
Config.1p	Remote UI Out Encrypt Master Out RTP Out UTP Out NP Out	Remote UI In Encrypt Master In RTP In , Relb UTP In NP In	Encrypt Slave	Encrypt Slave
Config.4p	Remote UI In & Out RTP In, Out, Relb UTP In & Out	NP In & Out		
Config.2p	Remote UI In & Out	RTP In, Out, Relb UTP In & Out	NP In & Out	
Config.m	.....Message parallelism.....			

Figure 5: Examples of CI configurations

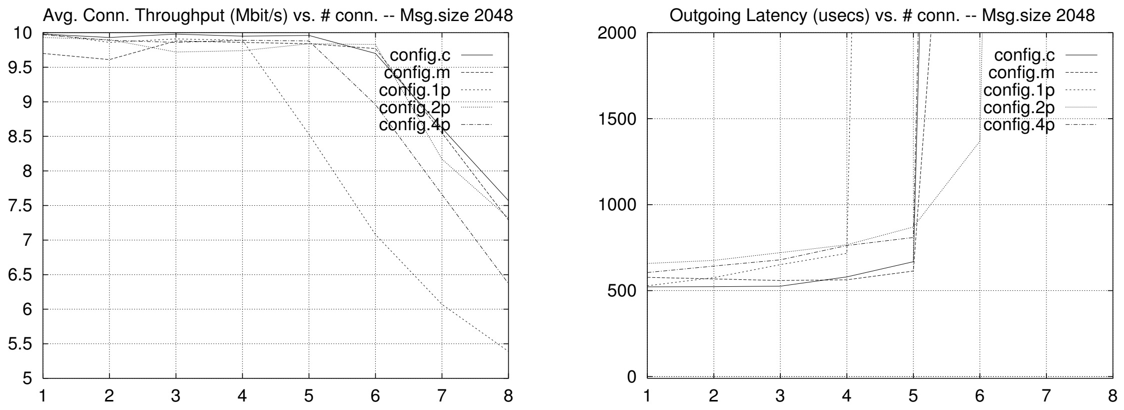


Figure 6: Effects of load distribution (unbounded rate, unreliable, checksum, no encryption)

• **Well balanced, quasi-pipelined configurations control message latencies** when the numbers of connections and transmissions increase. This is because a quasi-pipelined configuration can divide message processing (i.e., group the protocol modules) into appropriately sized chunks and then assign them to the available processors, thereby resulting in decreased waiting time and higher throughput for messages at high loads. For instance, in Figure 6, config.2p is better balanced than config.4p and config.1p, and consequently config.2p exhibits superior throughput and latency when more than 5 connections are active. However, for low processing requirements or low loads and when the processing chunks are small, the better 'balanced' configuration exhibits worse latency and no significant throughput improvement compared to the other configurations, primarily due to its pipe management operations and the lack of processing locality. For example, in Figure 6, for fewer than 5 connections, throughput is comparable for all three configurations, while the latency shown by config.2p exceeds the latency of config.1p by more than $100 \mu sec$.

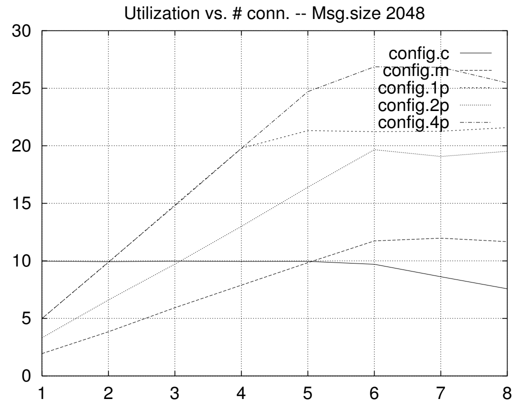


Figure 7: Configuration effects on efficiency (unbounded rate, unreliable, checksum, no encryption)

• **For reasonable processing requirements, quasi-pipelined configurations yield better overall throughput per number of processors than other types of configurations** (see Figure 7). Comparing configurations with the same number of processors for protocol stacks that do not require encryption (a very CPU intensive protocol), a quasi-pipelined configuration (e.g., config.2p) can support more concurrent connections than a connectional parallel configuration (config.c). In addition, the quasi-pipelined configuration enables lower waiting times (and better throughput) than a message parallel configuration (config.m). This feature is very important when only a small number of processors may be assigned to the CI.

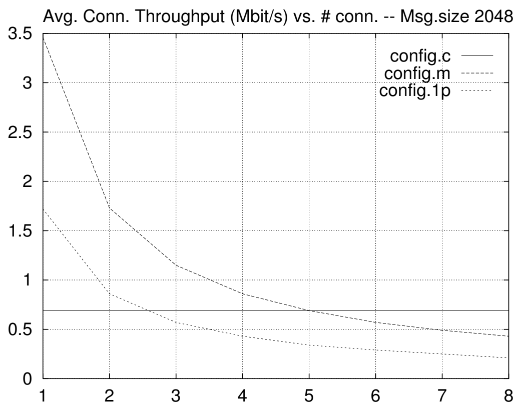


Figure 8: Parallelization effects on computationally intensive protocols (unbounded rate, unreliable, no checksum, encryption)

- **Computationally intensive protocols (e.g., encryption) are better serviced with message parallel configurations.** Figure 8 illustrates the advantages of config.m vs. config.c (serial encryption) and vs. config.1p, (parallel encryption, with 3 encryption slaves, out of the same overall number of processors as config.m). Message parallelism benefits from all processors' involvement in computationally intensive tasks, and from good locality in message processing.

- **Connectional parallel configurations offer the best predictability** (see Figure 6, config.c, up 5 connections when the NI overload occurs). The reasons are multiple: (1) reduced synchronization overhead at the user interface, (2) locality of message processing, (3) no interconnection influences, and (4) a natural message ordering. However, the drawback of connectional parallelism is the poor CPU utilization.

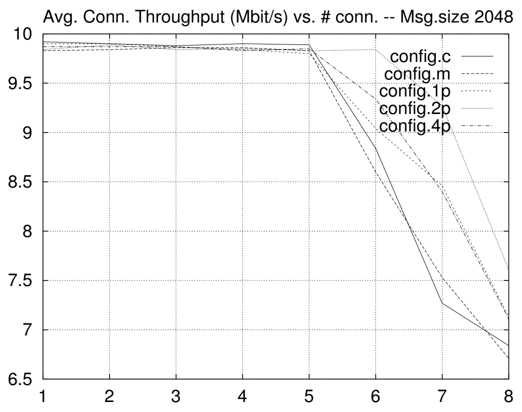
- **The performance of the network interface is influenced by the number of processors directly accessing it.** Consider the measurements presented in Figure 9: at high loads (number of processors), connectional (config.c) and message (config.m) parallel configurations experience significant performance degradation. This is due to the large number of processors directly accessing the NI and generating high contention for NI services and for the interconnect. Specifically, at high contention levels, the NI's overheads of connection request queue management and the waiting time for a request completion will increase significantly. With pipelined configurations, such contention may be avoided by assigning only one or two processors to directly access the NI (e.g., config.2p, config.1p).

The main idea conveyed by the experimental results presented in this section is that no single configuration can provide for 'best' performance (in terms of throughput, latency, predictability) under a large variety of application requirements. This suggests that a high performance CI should support runtime (auto-)configuration. More specifically, these results provide the following insights that may be used to develop effective heuristics for auto-configuration:

- Quasi-pipelined configurations are easily structured to match given throughput and latency requirements. In addition, perturbation effects may be kept under control by isolating incoming from outgoing streams, by grouping connections with similar QoS requirements. or by balancing loads.
- Message parallel configurations are useful for execution of computationally intensive tasks.
- Connectional parallel configurations result in low message latency and high levels of predictability.

4 Auto-Configuration

Auto-configuration is the fashion in which COMM^{adapt} achieves its main goal, which is to accommodate dynamic changes in communication requirements while complying with stated QoS requirements. Configuration decisions are made at the time of connection establishment and dynamically, when changes in traffic are detected. The goal of such decisions is to guarantee the required QoS and traffic characteristics based on the appropriate assignment of processing capacity to exist-



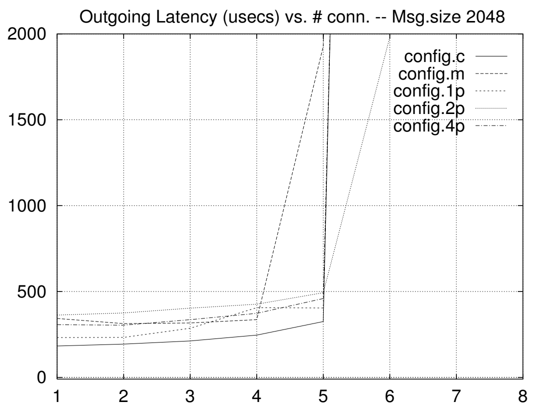


Figure 9: Effects of NI bottleneck (10Mbits/sec, unreliable, no checksum, no encryption)

ing connections (i.e., mapping of module instances to the available processors). The most important parameters of the decision model embedded in COMM^{adapt} are:

- *connection requirements*, which are described by the set of modules on the protocol stack, and by the traffic and QoS requirements;
- *connection guarantees*, which are described by the traffic characteristics and the QoS currently guaranteed for each connection;
- *connection configuration*, which is defined by its current resource assignment[4]; and
- *available processing capacity*, which describes the processing capacity available for communication tasks on each processor in the CI configuration. We assume that some system-level scheduler guarantees this capacity.

A configuration is considered *acceptable* if for all of its active connections, the estimated performance satisfies the *actual* traffic and QoS requirements. As the goal in COMM^{adapt} is to accommodate unexpected requirements with its available resources, configuration decisions are based on the assumption that the near past is a better model for the future than specifications stated when a connection is established. Therefore, we will use reserved but unused resources resulting from overestimated connection-time specifications or from runtime traffic fluctuations to satisfy new requirements.

The load assignment heuristic presented in this section generates acceptable CI configurations. Such configurations are not restricted to the few evaluated in Section 3: each connection may have its specific assignment of modules to processors.

[4]The current prototype considers only the assignment of module instances to processors.

The following text first presents some details about the conditions required for an acceptable configuration, then describes the heuristic and the online configuration procedures. A simple experiment illustrates the operation of COMM^{adapt} 's auto-configuration mechanism.

4.1 Configuration Model

In COMM^{adapt}, a connection is defined by its input and the output streams, each with its own traffic and QoS requirements. Similar to [1], for each stream the traffic requirements are described by average inter-arrival time (X_{ave}), minimum inter-arrival time (X_{min}), averaging interval (I), and maximum message size (S_{max}). QoS requirements are described by an upper bound on the end-to-end delay (D_{max}) and maximum loss burst (B_{max}). To estimate actual traffic characteristics, the metrics monitored on a per connection basis are message inter-arrival time, message size, and message latency (within CI's confines).

At the time of connection establishment, based on connection-specific requirements, COMM^{adapt} defines the structure of the protocol stack, evaluates the overhead of message processing at each protocol module, and allocates appropriate buffer space. During the connection's lifetime, its per-module overheads X_{ave}, X_{min}, S_{max} may change based on monitoring information.

In our model, the CI is seen as a network of queues where the servers are the PE Controllers and the queues are the (inter-module) pipes. Specific to our model is that (1) each queue is receiving input from a single stream, and (2) a round-robin policy is used by the server in servicing its queues. This features enable a simpler, easy to evaluate at run-time model for the message time in the system than a the general network of queues model.

The result of a heuristic-based configuration decision is a CI configuration where all active connections are guaranteed the required QoS and traffic parameters. As stated earlier, this is called an *acceptable configuration*. In order to define the conditions that should be met by such a configuration, the following notations and definitions are introduced:

- $\mathcal{P}$ – the set of processors assigned to the CI
- $\mathcal{I}$ – the set of communication streams (2 per connection)
- $X_{ave}(i)$, $X_{min}(i)$, $I(i)$, $S_{max}(i)$ – the current traffic descriptor for stream i
- $Avail(p)$ – capacity percentage assigned to CI processing on processor p.
- $Exec(\mathcal{C}, p, i)$ – the overhead of processing a message in stream i, on processor p, given the assignment of module instances in configuration $\mathcal{C}$. This includes the overhead of processing one or more modules, of synchronization, of pipe management, and of shepherding. The actual processing overhead of a module instance is divided by the number of processors assigned to process it. This assumption ignores the difference of load among processors, but it results in any easily evaluated approximation of actual overheads.
- $Exec^*(\mathcal{C}, p, i)$ – the actual time necessary to execute $Exec(\mathcal{C}, p, i)$ given the assigned CPU percentage:
$$Exec^*(\mathcal{C}, p, i) = \frac{Exec(\mathcal{C}, p, i)}{Avail(p)}$$
- $\mathcal{U}$ – the utilization of processor p determined by CI processing
$$\mathcal{U} = \sum_{i \in \mathcal{I}} \frac{Exec(\mathcal{C}, p, i)}{X_{avg}(i)}$$
- $Proc(\mathcal{C}, i)$ – the set of processors assigned module instances of stream i (i.e., $p \in P(\mathcal{C}, i)$ iff $Exec(\mathcal{C}, p, i) \neq 0$).
- $Conn(\mathcal{C}, p)$ – the set of streams with modules assigned to p (i.e., $i \in Conn(\mathcal{C}, p)$ iff $p \in Proc(\mathcal{C}, i)$
- $Share(\mathcal{C}, i)$ – the set of streams which share processors with stream i
$$Share(\mathcal{C}, i) = \bigcup_{p \in Proc(\mathcal{C}, i)} Conn(\mathcal{C}, p) \setminus \{i\}$$
- $PComm(\mathcal{C}, i, j)$ – the set of processors shared by streams i and j:
$$PComm(\mathcal{C}, i, j) = Proc(\mathcal{C}, i) \bigcap Proc(\mathcal{C}, j)$$
- $prob(i, j)$ – the probability that a message in stream i must wait in the pipe for a message in stream j to be serviced, at some processor in CI. Given the round-robin service policy, we can approximate $prob$ by the inverse of the number of j messages that fit into i's inter-arrival time:
$$prob(i, j) = \frac{1}{\left\lceil \frac{X_{avg}(i)}{X_{avg}(j)} \right\rceil}$$
- $Wait_{max}(i, j)$ – the worst case waiting time incurred on a message in stream i by some message of stream j:
$$Wait_{max}(i, j) = \max_{p \in PComm(\mathcal{C}, i, j)} Exec^*(\mathcal{C}, p, i)$$
- $Wait_{avg}(i, j)$ – the average waiting time incurred on a message in stream i by some message of stream j:
$$Wait_{avg}(i, j) = \max_{p \in PComm(\mathcal{C}, i, j)} (Exec^*(\mathcal{C}, p, i) \cdot prob(i, j))$$
- $\mathcal{X}_{max}(i)$ – the maximum time between two consecutive messages in the stream i leave the CI is:
$$\mathcal{X}_{max}(i) = \max_{p \in Proc(\mathcal{C}, i)} Exec^*(\mathcal{C}, p, i) + \sum_{j \in Share(\mathcal{C}, i)} Wait_{max}(i, j)$$
- $\mathcal{X}_{avg}(i)$ – the average time between two consecutive messages in the stream i. It is computed exactly as $\mathcal{X}_{max}(i)$, except that $Wait_{avg}$ values are used. In a less conservative approach, the $\mathcal{X}_{avg}$ may be used in place of $\mathcal{X}_{max}$.

- $\Lambda_{max}(i)$ – the maximum time necessary for a message in stream i to be completely processed, given the stream i *is not bursty*.

$$\Lambda_{max}(i) = \sum_{p \in Calc(\mathcal{C},i)} Exec^*(\mathcal{C},p,i) + \sum_{j \in Share(\mathcal{C},i)} Wait_{max}(i,j)$$

  $\Lambda_{avg}(i)$ is defined analogous.
- $\beta(i)$ – the maximum burst of stream i:

$$\beta(i) = \left\lceil \frac{I(i)}{X_{ave}(i)} \right\rceil$$

- $\mu(x) \overset{def}{=} \max\{0,x\}$ – used to simplify formulas
- $\omega(i,k)$ – the maximum waiting time of the k-th message in a burst of stream i is :

$$\omega(j,k) = \mu(\mathcal{X}(j) - X_{min}(j)) \cdot (k-1)$$

  Due to the round-robin scheduling of pipes by PE Controllers and due to the use of multiple, independent module instances, the worst-case latency of a given stream is affected only by its own burstiness, not by the behavior of other streams.
- $\mathcal{L}(i)$ – the maximum latency observed by some message in stream i. Note that the largest waiting time is observed by the last message in a burst and that no more than $B_{max}(i)$ (maximum acceptable loss burst for the stream i) can be dropped.

$$\mathcal{L}(i) = \Lambda(i) + \omega(i, \mu(\beta(i) - B_{max}(i)))$$

The primary requirements for an acceptable configuration is to comply with the processor capacity assignment. The following should hold for each processor p and for the NI:

$$\mathcal{U}(p) < Avail(p) \qquad (1)$$

For a stream, i, to always meet its average rate requirements, the following should hold:

$$\mathcal{X}_{max}(i) < X_{avg}(i) \qquad (2)$$

To guarantee the required QoS, given the B_{max} acceptable loss burst, the sum of latencies $\mathcal{L}$ at the two hosts and the network overhead should be smaller than D_{max}. To be fair, we consider that each of the hosts and the network are "entitled" to at most a third of the user-specified loss burst (i.e., $B_{max}/3$ will be used the evaluation of $\mathcal{L}$).

Note that the above model assumes that the system level scheduler is guaranteeing the assigned CPU percentage. Such schedulers have been presented in [18], [14] and [12].

Given this model, the goal of the auto-configuration decision in COMM^{adapt} is to comply with the available resources (see 1), rate requirements (see 2), and to minimize message latency. The module assignment heuristic presented next attempts to achieve these goals.

4.2 Connection-Time Configuration Heuristic

The goal of connection-time configuration is to allocate resources to a new connection such that it will meet its traffic and QoS requirements and such that all other connections continue to maintain their current traffic and QoS guarantees. The configuration decision is part of the connection establishment procedure, which is coordinated by the CI Manager (CIM). The connection establishment procedure consists of: (1) establishing the corresponding set of protocol modules and estimating their specific overheads, (2) evaluating the levels of guarantee at source and destination hosts, and considering what the underlying network can offer, and (3) informing the user about achievable guarantees. A host approves the connection, provided an acceptable configuration is found. Otherwise, it rejects the connection request and returns feedback on the level of guarantees it can offer.

When evaluating the levels of guarantees a given connection may be granted, the CIM first tries to assign to this connection all necessary resources. The goal is to meet the traffic and loss burst requirements while minimizing the predicted worst case latency. Towards this end, the incoming and outgoing connection streams are assigned incrementally by a *three-phase algorithm*, where a phase is entered only if the previous phase failed to define an acceptable configuration.

Phase 1. In the first phase, the goal is to meet the required rate, while maintaining low latency. Protocol modules are assigned one at a time, in protocol stack order, so as to construct a quasi-pipelined configuration. The intent is to attain low latency by exploiting *processing locality* while maintaining a global *load balance.* As a result, the algorithm first attempts to assign a module to the same processor as its predecessor. If this does not yield an acceptable configuration, then the module is assigned to the least loaded processor. With this heuristic, connections with small processing requirements and tight latencies may be completely assigned to a single processor, while several such connections may share the same processor provided their requirements are guaranteed.

Phase 2. In the second phase, the goal is to achieve rate requirements beyond those achievable in the first phase, while minimizing the effects on other connections. This is attained by uniformly spreading the load of the new connection over the smallest possible number of processors as follows: (1) the entire protocol stack is assigned to the two least loaded processors, and then (2) processors are added to the connection set until either an acceptable configuration is found or no more processors are available. We call this a *partially message parallel* configuration, as it is similar to a message parallel configuration but may span only part of the available processor set. When extending such a configuration (i.e., adding processors), the achievable throughput improves. Interestingly, latency degrades in comparison to a first phase assignment only if total processing time is comparable to total pipe locking overhead (see Figure 9, config.m) or if the load on the corresponding processor set is too high.

Phase 3. In the third phase, the goal is to increase the resources available to the new connection by improving the load balance of the entire processor set. With the new connection extended over the entire processor set (as at the end of the second phase), the configurations of other connections are modified by extending them to partial message parallel configurations. Connections are considered in increasing order of their ratios of experienced vs. guaranteed latency. This action will reduce the load on the most loaded processors while preserving the guarantees of already active connections.

To summarize, the connection assignment heuristic provides a first fit with respect to the available processing capacity (1) and the rate requirements (2) while keeping the latency small by either locality of processing or load balancing (i.e., small waiting time).

4.3 On-line Configuration Heuristic

On-line configuration is triggered by the QoS evaluation mechanism whenever some connection receives inappropriate service or is shut down. In the context of our COMM^{adapt} prototype, poor service is caused by some connection increasing its load (number of messages, message size) over its guaranteed traffic level.

The goal is to accommodate a connection's load increase by taking advantage of currently unused resources. The misbehaving connection is subjected to the same load assignment heuristic as the one used for connection-time configuration (see Section 4.2). If no acceptable configuration is found, buffer management or policing are used to restrict traffic to remain within appropriate bounds. Moreover, COMM^{adapt} informs the application program about the traffic violation, enabling it to either adjust the traffic (by application-level adaptation) or explicitly negotiate new communication requirements.

Note that at each connection end-point, the 'host' is allowed to react independently when inappropriate service is noticed, as it is very likely that different configurations, loads or protocol processing overheads exist at the communicating sites Coordination among the hosts is needed only when the user explicitly renegotiates traffic and QoS requirements.

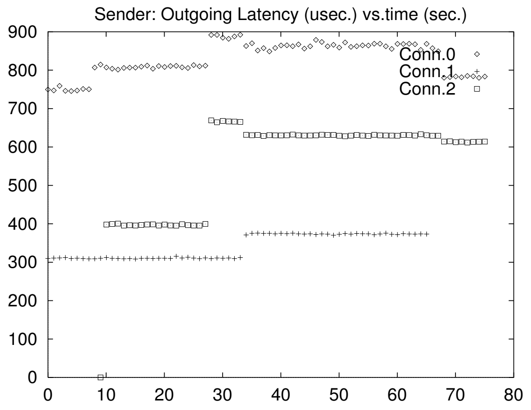
At connection shut down, the configuration mechanism will immediately redistribute any released resources. The configuration procedure (same as above) is invoked for those connections that were guaranteed worse performance than required.

4.4 Experiments With On-line Configuration

The experiments described in this section demonstrate the operation of the dynamic auto-configuration mechanism by using it with traffic characteristics like those experienced by video service applications. These characteristics are: (1) execution in a multiprocessor environment with two processors assigned exclusively to communication processing, (2) high bandwidth, unreliable, checksummed connections originating at

	Message (KBytes)	Rate (Mb/sec)	Pkts/sec	Latency μsec	Start Sec.	Active Sec.	sec.0	sec.10	sec.30	sec.70
conn.0	4	10	320	700	0	80	PE.2	PE.2	PE.2	PE.1,2
conn.1	1	11	1408	400	0	70	PE.1	PE.1	PE.1,2	—
conn.2	1	9	1152	400	10	70	—	PE.2	PE.1,2	PE.1,2

Table 2: Initial connection requirements and stream assignment with the auto-configuration mechanism



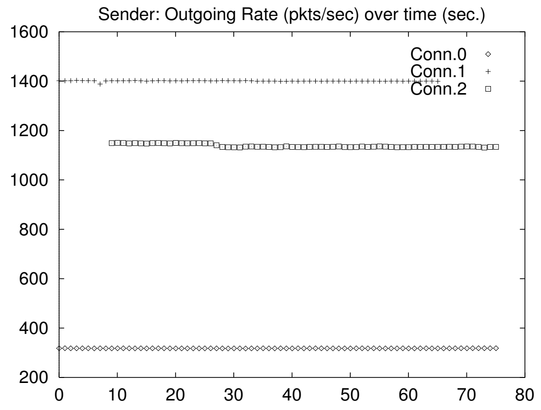


Figure 10: Latency and throughput, at sender site, with COMM^{adapt} auto-configuration.

the same source, (3) strict rate and latency requirements depending on each receiver's processing and buffering capabilities, (4) a potential for rate increases (as video stream characteristics change) supported by message size variation.

The experiment consists of running three connections with the connection-time requirements described in Table 2. The table includes the evolution of the connection configurations over time. Figure 10 present the evolution of the rate and the latency for the three connections.

Based on the heuristics described in Section 4.2, the *connection-time configuration decisions* are: (1) assign Conn.0 to a single processor (PE.0) due to its very tight latency requirements (in fact, Conn.0 requires a latency lower than the minimum required for its message size and processing requirements), (2) assign Conn.1 to PE.2, the least loaded processor, and (3) assign Conn.2 to the same processor as Conn.0 (PE.0) since the latter's low rate creates a lower load than for Conn.1's processor.

After 30 seconds, a perturbation occurs when Conn.2 doubles its message size, thereby negatively affecting the latencies of the other connections. Figure 10 depicts the effects of this perturbation (see 'sec.30') and of the *auto-configuration* decision (see 'sec.35'), which is to (1) transfer part of the extra load of Conn.2 to processor PE.1, (2) extend Conn.1 across both processors to keep its rate within required bounds, and (3) leave unchanged the assignment of Conn.0 as its estimated rate is at the guaranteed level. The parameters controlling activation delay are set such that this delay is fairly high (4 secs.), so that the effects induced by the traffic perturbation are more easily seen in Figure 10.

In this experiment, the on-line auto-configuration mechanism is triggered a second time when Conn.1 is shut down. This results in reducing the expected latency for Conn.0 closer to the required level, by extending Conn.0 to a message parallel configuration.

This straightforward experiment demonstrates that auto-configuration as performed by COMM^{adapt} s load assignment heuristic can help accommodate unexpected traffic changes while providing performance close to required levels. Importantly, such performance improvements do not require any changes in the application program.

5 Contributions, Related Research, and Future Work

Contributions. The goal of our research is to integrate the Communication Infrastructure (CI) with the host system and the network in order to improve service levels for high-performance and real-time connections. Specifically, we wish to eliminate the potential performance problems caused by current systems that require a strict match of specifications stated at the time of connection establishment with actual connection behavior over time. The solution approach demonstrated in this paper is based on *auto-configuration* – CI-level configuration decisions concerning the assignment of available resources. Benefits derived from runtime auto-configuration include: (1) improved processing and network resource utilization vs. user-specified configurations and connection-time configurations, (2) improved response time to unpredicted or unlikely requirements, and (3) compliance with specific application needs by use of customized configuration algorithms. The novel results presented in this paper are:

- the design and implementation of COMM^{adapt} software library for constructing configurable protocols, supporting user- and CI-directed dynamic protocol configuration;
- the model and heuristic for the allocation of CPU resources to protocol processing tasks, able to guarantee service levels to well-behaved connections while simultaneously accommodating runtime traffic variations elsewhere;
- interesting insights on the behavior of different CI configurations, including (1) that pipelined CI configurations on a small number of processors may yield performance comparable to message and connectional parallel configurations (in contrast to statements made in [27]) and (2) that the NI bottleneck may restrict the performance of the highly parallel communication architectures proposed in previous studies[28, 21].

The philosophy behind our auto-configuration approach may be summarized by the following statement: a prompt CI-level decision based on the observed traffic characteristics may keep performance within acceptable limits while accommodating an application program's dynamic resource requirements. Such configuration decisions are intended to prevent performance degradation until the application decides how to handle the observed traffic variation. These characteristics make COMM^{adapt} appropriate for servicing complex systems with dynamic and/or hard to accurately stated QoS requirements.

Related Work. The COMM^{adapt}prototype is loosely based on the previous work of [17] who only comment on the necessity of runtime adjustments in the mapping of protocol modules to processors.

COMM^{adapt} differs from previous work on configurable protocols in its support for constructing protocols that are auto-configurable at runtime in their functionality and performance. For instance, in the x-Kernel [10, 21], each connection is bound to a specific resource assignment/protocol stack at the time of connection establishment. DaCaPo [22] is an environment where the processes that execute the protocol related computations stacks can handle dynamic variations in the resource requirements, but no details are provided on the corresponding mechanisms. The ADAPTIVE system[27] enables only application-level adaptation decisions. The parallel STREAMS implementation in [7] offers only user-directed stream configuration, and the policy for runtime scheduling of communication related tasks ignores QoS requirements. The framework presented in [34] supports QoS requirements in the context of functionally decomposed protocol stacks by CI-level connection-time configuration decisions that minimize protocol functions and their processing requirements.

Our study of the various CI configurations focusing on per connection performance (throughput, latency and predictability) is different from similar studies reported in the literature which focus only on aggregate throughput. [28] reports on the influence of the synchronization and context-switching overheads on connectional and message parallel configurations; [21] analyzes the scalability of a parallel x-kernel implementation for message parallel configurations. Moreover, these studies ignore pipelined configurations, since those are considered [27] to have prohibitive synchronization and communication overheads. In contrast, we do consider such configurations and take advantage of their efficiency and flexibility, especially when the CI is run only on a small number of processors (e.g., 2-3).

The dynamic auto-configurability mechanism in COMM^{adapt} is designed for the layered protocol model, but we posit that similar mechanisms can provide benefits to protocol architectures like HOPS [9], F-CSS [34], or to the micro-protocols described in [2]. Toward this end, the minimal requirements imposed by the CI model is to permit the runtime estimation of resource requirements and of the actual usage for each connection in the system.

Future Work. Our future work will address the QoS requirements of complex distributed real-time applications by designing novel configuration heuristics that integrate CPU, buffer and bandwidth management. As part of this work, we will integrate the COMM^{adapt} infrastructure into CORBA-compliant, high performance, object-based middleware now being constructed at Georgia Tech, called COBS[31]. The intent of COBS is to permit applications to configure object implementations to their specific needs. Large-scale applications with which this system is being evaluated include groupware and collaborative applications transporting significant amounts of data and performing substantial processing on such data [32], traditional high performance codes like the parallel global atmospheric model described in [15], and embedded real-time applications, as described in [13]. These applications are being constructed jointly with other faculty in the confines of the 'Distributed Laboratories' project at Georgia Tech. This project's general goals are to enable multiple scientists to jointly solve their problems with computational instruments residing on heterogeneous, networked computing engines. Some of these engines may require COMM^{adapt} s support for parallel protocol execution, due to their high input and output rates when using their multiple processing elements, whereas others may require different configuration algorithms embedded in COMM^{adapt}and suitable for single-CPU machines.

Also as part of the integration of COMM^{adapt} and COBS, we will develop support for the simultaneous use of multiple network devices and resources by application programs. In addition, we are considering using COMM^{adapt} for real-time and high performance communications via an optimized ATM device interface described in[24].

Acknowledgments. We acknowledge Prof. Sudhakar Yalamanchili for helpful discussions on mapping algorithms.

References

[1] A. Banerjea and D. F. et al. The tenet real-time protocol suite: Design, implementation, and experiences. *IEEE/ACM Transactions on networking vol.4, no.1*, Feb. 1996.

[2] N. Bhatti and R. Schlichting. A system for constructing configurable high-level protocols. *Proc. SIGCOMM*, 1995.

[3] T. Bihari and K. Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems, vol.9, no.2*, pages 143–174, May 1991.

[4] P. Druschel, L. Peterson, and B. Davie. Experiences with a high-speed network adapter: A software perspective. *Proc. SIGCOMM*, pages 2–13, 1994.

[5] A. Edwards, G. Watson, J. Lumley, et al. User-space protocols deliver high performance to applications on a low-cost gb/s lan. *Proc. SIGCOMM*, pages 14–23, 1994.

[6] E. Felten. Protocol compilation: High-performance communication for parallel programs. *U. of Washington, Dept. Computer Science and Eng., TR 93-09-09*, 1993.

[7] A. Garg. Parallel streams: a multi-processor implementation. *Proc. USENIX*, pages 163–176, Winter'90.

[8] P. Gopinath and R. Gupta. Compiler-assisted adaptive scheduling in real-time systems. *7th IEEE Workshop on Real-Time Operating Systems and Software*, pages 62–69, 1990.

[9] Z. Haas. A protocol structure for high-speed communication over broadband isdn. *IEEE Network Magazine*, pages 64–70, Jan. 1991.

[10] N. Hutchinson and L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. on Software Engineering, 17(1)*, pages 64–76, Jan. 1991.

[11] K. Jeffay. The real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems. *Proc. ACM/SIGAPP Symposium on Applied Computing*, pages 769–804, 1993.

[12] K. Jeffay and D. Bennett. A rate-based execution abstraction for multimedia computing. *Proc. 5th Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, 1995.

[13] R. Jha, M. Muhammad, S. Yalamanchili, D. I.-R. K. Schwan, and C. DeCastro. Adaptive resource allocation for embedded parallel applications. *conference submission*, Aug. 1996.

[14] M. Jones, P. Leach, R. Draves, and J. Barrera. Modular real-time resource management in the rialto operating system. *Proc.of the 5th Workshop on Hot Topics in Operating Systems*, pages 12–17, 1995.

[15] T. Kindler, K. Schwan, et al. A parallel spectral model for atmospheric transport processes. *Georgia Institute of Technology, College of Computing, TR GIT-CC-95-17*, 1995.

[16] R. Kravets, K. Calvert, and K. Schwan. Dynamically configurable communication protocols and distribued applications: Motivation and experience. Technical Report GIT-CC-96-16, Georgia Institute of Technology, 1996.

[17] B. Lindgren, B. Krupczak, M. Ammar, and K. Schwan. Parallel and configurable protocols: Experiences with a prototype and an architectural framework. *Georgia Institute of Technology, GIT-CC-93-22*, 1993.

[18] C. Mercer and H. Tokuda. An evaluation of priority consistency in protocol architectures. *Proc. IEEE 16th Real-Time Systems Symposium*, Oct.1991.

[19] Mori and N. N. I. S. L. Wasano. A benchmark for advanced real-time systems. *personal communication*, 1994.

[20] B. Mukherjee. A portable and reconfigurable threads package. In *Proceedings of Sun User Group Technical Conference*, pages 101–112, June 1991.

[21] E. Nahum, D. Yates, J. Kurose, and D. Towsley. Performance issues in parallelized network protocols. *Proc. 1st Symposium on OSDI*, pages 125–137, 1994.

[22] T. Plagemann, B. Platter, et al. Modules as building blocks for protocol configuration. *Proc. International Conference on Network Protocol*, 1993.

[23] J. Rexford, J. Dolter, and K. Shin. Hardware support for controlled interaction of guaranteed and best-effort communication. *Workshop on Parallel and Distributed Real-Time Systems*, pages 188–193, April 1994.

[24] M. Rosu. Processor controlled off-processor i/o. *Cornell University, Dept.Computer Science TR 95-1538*, 1995.

[25] J. Saltz, G. Edjlali, et al. Data parallel programming in an adaptive environment. *Proc. of the 9th International Parallel Processing Symposium*, pages 812–819, 1995.

[26] S. Saxena, J. K. Peacock, et al. Pitfalls in multithreading svr4 streams and other weightless processes. *Proc. USENIX*, pages 85–96, Winter'93.

[27] D. Schmidt, D. Box, and T. Suda. Adaptive. a dynamically assembled protocol transformation, integration, and evaluation environment. *Journal of Concurrency: Practice and Experience*, Jun.93.

[28] D. Schmidt and T. Suda. Measuring the performance of parallel message-based process architectures. *IEEE INFOCOM*, 1995.

[29] D. Schmidt and T. Suda. Transport system architectures for high-performance communications subsystems. *IEEE Journal on Selected Areas in Comm., vol.11,No.4*, May 93.

[30] P. Schneck, E. Zegura, and K. Schwan. Drrm: Dynamic resource reservation manager. *Proc. of IC3N*, 1996.

[31] K. Schwan and M. Ahamad. Cobs - configurable objects for high performance systems. *http://www.cc.gatech.edu/systems/projects/COBS/*, 1995.

[32] K. Schwan, G. Eisenhauer, V. Martin, B. Schroeder, and J. Vetter. From interactive high performance applications to distributed laboratories. *journal submission*, July 1996.

[33] R. Sharma and S. Kesav. Signaling and operating systems support for native-mode atm applications. *Proc. SIGCOMM*, pages 149–157, 1994.

[34] M. Zitterbart, B. Stiller, and A. Tantaway. A model for flexible high-performance communication subsystems. *IEEE Journal on Selected Areas in Comm., 11(4)*, May 1993.