Improving Protocol Performance by Dynamic Control of Communication Resources

Daniela Ivan-Roşu* Karsten Schwan
Georgia Institute of Technology
College of Computing
{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 concerns communication infrastructures able to cope with the varying processing and QoS[1] requirements imposed on them by application programs. Specifically, we describe and evaluate COMM^{adapt}, a communication infrastructure enabling the on-line adaptation of a protocol's resource usage to currently available resources and application requirements. The 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 by other active connections. The heuristic's design and implementation are based on extensive investigations of the manner in which alternative assignments of protocol tasks to underlying processing resources can influence program-level latency and throughput requirements.

1 Motivation

For high performance 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 applications may be increased if communicating program components can exploit runtime knowledge of communication delays when scheduling computations and associated communications [3, 12]. Similarly, both the performance and the predictability of real-time applications may be improved if the communication infrastructure (CI) can guarantee bounds on communication delays.

This paper concerns the *communication processing* performed in distributed applications. We posit that the dynamic configuration of such processing is becoming increasingly important due to the complexity and runtime variability of underlying network platforms and of modern multi-threaded and distributed applications, which makes it difficult to model and estimate runtime communication requirements and network behavior. Specifically, we wish to eliminate the unnecessary restrictions imposed by current communication systems that require a strict match of connection-establishment time specifications and actual connection behavior. Moreover, for improved reactivity and and in order to shield application programs from changes in communication behavior, we aim to have the CI itself make configuration decisions based on its knowledge of actual and current resource availability.

Toward these ends, we develop COMM^{adapt} CI, a software library implementing a communication infrastructure that offers the resource management mechanisms necessary for dynamic configuration while also enabling the user-level composition of communication protocols. The mechanisms for dynamic resource allocation of CPU resources to communication processing tasks enable the application, the CI, or both to make on-line configuration decisions that support runtime changes in communication requirements. COMM^{adapt}'s reconfiguration decisions are made by a load assignment heuristic that maps communication requirements to alternative parallel configurations of communication modules. Such decisions are triggered by changes in program behavior (e.g., the unforeseen establishment of additional connections, or unexpected rate increases for existing connections) and modifications in resource availability(e.g., due to varying network loads).

The COMM^{adapt} prototype is evaluated on a shared memory multiprocessor (a KSR-2) using a portable user-level threads library (running on most existing multipro-

*Funded in part by DARPA through the Honeywell Technology Center under contract B09332478

[1]Quality of Service

cessl machines). The underlying network is simulated by the interconnect of the host multiprocessor.

COMM^{adapt} is intended for applications like (1) video or data servers, where the aim is to maximize the number of connections with high throughput and predictable jitter and (2) distributed real-time applications, where multiple streams of data are collected and processed subject to specific and possibly dynamic rates and timing requirements. While previous work in configurable communication systems [6, 8, 10, 13, 14, 18] addresses similar application domains, the maintenance of QoS requirements by on-line auto-configuration of protocols pursued in our research constitutes a novel contribution.

The remainder of this paper first outlines the COMM^{adapt} software architecture and its prototype (see Section 2). Next, insights on the appropriate design of the load assignment heuristics used for auto-configuration are derived from studies of the influence of alternative protocol configurations (see Section 3) on communication processing latency and throughput. Details of COMM^{adapt}'s configuration mechanisms and heuristic are presented in Section 4. Finally, COMM^{adapt}'s contributions are reviewed in comparison with related research in Section 5. Details on the COMM^{adapt} prototype and the evaluation of the load assignment heuristic can be found in [7].

2 The COMM^{adapt} Communication Infrastructure

2.1 The Software Architecture

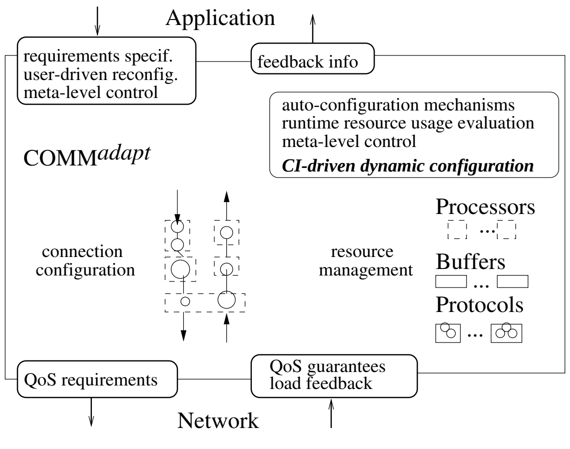


Figure 1. COMM^{adapt} architecture

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 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 1). The need for such mechanisms has already been established by previous work [6, 10, 14, 18]. In contrast to such work, COMM^{adapt} is addressing the difficulty of accurately estimating the runtime behavior of complex applications with numerous communication streams. Specifically, COMM^{adapt}'s *dynamic configuration mechanism* aims to accommodate situations where actual connection requirements exceed those initially stated, but may be satisfied by use of spare resources recovered by reallocation.

To support these goals, the COMM^{adapt} architecture offers:

- *Auto-configuration* mechanisms, which enable CI-level decisions about resource allocation to individual connections, based on protocol- and resource-specific algorithms/heuristics and transparent to applications. For example, dynamic increases in the traffic requirements of a specific connection may be satisfied by giving it resources (processing capacity, buffers, link bandwidth, etc.) obtained from other active communication streams that do not need them in order to satisfy their traffic requirements.
- *Runtime usage evaluation*, which provides the information for dynamic configuration.
- *Meta-level control* of CI configuration, which serves to adapt the decision mechanism (e.g., algorithms for resource management, decision-making policies) to specific applications or sets of requirements [2]. For example, the CI itself may vary the quality and promptness of its decision making (e.g., by adjusting the amount of monitoring or periodicity of status checking).

Figure 1 presents the main components of the COMM^{adapt} software architecture together with its interfaces to the network and to application programs. The next section briefly presents the COMM^{adapt} prototype.

2.2 The Prototype

The COMM^{adapt} prototype depicted in Figure 2 is implemented as a user-level library on Unix machines, and it is evaluated in a simulated network environment, on a KSR-2 parallel supercomputer (a CC-COMA shared-memory multiprocessor, with a sequentially consistent shared address space and a 1GBytes/sec interconnect). The network interface (NI) is a simulated network adapter enabling direct interface to the user space. It is modeled after the high performance ATM interface described in [11]. The network 'fiber' is simulated with shared memory buffers mapped into the sending and receiving address spaces.

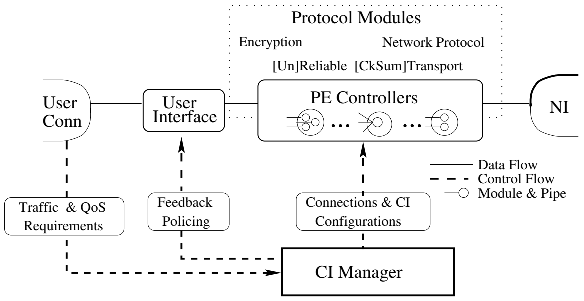


Figure 2. COMM^{adapt} prototype

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 interface with the application layer, called "user interface"; (3) communication task management, including mechanisms for inter-module communication (*pipes*) and for concurrent execution of protocol modules (*PE Controllers*); and (4) the configuration mechanisms.

The protocol library includes the protocols required by the specific applications. In the current prototype these protocols are: 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. For instance, the unreliable transport protocol has two modules, one receiving and the other sending, while the reliable protocol has an additional module for handling ACKs. 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 critical section processing. Application-specific protocol stacks can easily be constructed at runtime since all modules have the same input interfaces (i.e., message connection and CI state descriptors). 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 [17] and for flexible composition of application-specific protocols.

The user interface (UI) 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. The remote UI executes functions specific to the transition between user and CI layers (e.g., monitoring, usage control).

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 (i.e., no locking is needed on write/read if there is only a single writer/reader).

A module receives input from a pipe. The association between a protocol module and its input pipe defines a *module instance*. The flexibility in specifying and mapping module instances is the basis of the dynamic configuration mechanism in COMM^{adapt}. Multiple instances of the same protocol module may exist in the system. Also, one module instance may be assigned to several controllers at the same time, thereby enabling concurrent protocol execution.

A PE Controller executes a protocol module only when input is available on the corresponding pipe. In the current implementation, PE controllers follow a round-robin policy in checking pipes for available input. However, when for some request, 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. Although this simple *shepherding* policy ignores message priorities and deadlines, it reduces the overheads with scheduling and pipe operations. More importantly, shepherding enables to control the throughput and message latencyperformance by controlling the granularity of protocol stack decomposition across the available processors.

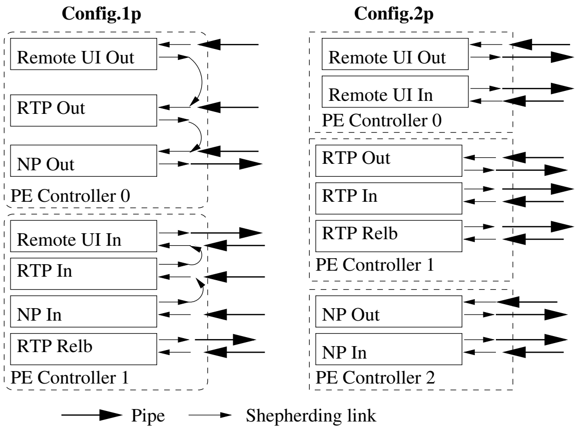


Figure 3. Examples of protocol stack assignments for a reliable connection (UI, RTP, NP)

Figure 3 illustrates sample assignments of protocol stacks for unreliable communications (UI, UTP, NP). Config.1p is characterized by its smaller latency because communication requests flow immediately over the shepherding links instead of waiting in the 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 connection QoS parameters with the peer CIM and underlying network, (2) the decision of configuration changes in order to accommodate the new requirements; and
- on-line auto-configuration, including: (1) the periodical checking of the information collected at various points in the CI, and the network feedback, (2) the comparison of the actual performance with the QoS and traffic requirements, and (3) the decision of appropriate configuration changes to accommodate detected divergence.

Configuration decisions are enacted by informing PE Controllers about the changes to their module sets.

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, 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 optimize performance metrics as per connection maximum message rate, latency, predictability, and efficiency. The configurations considered in this study are (see Figure 4):

- connectional parallelism (config.c), where each connection is assigned its own processor [16];
- message parallelism (config.m), where a message (in any connection) may be (completely) processed on any processor [16]; and
- quasi-pipelined (config.1p, config.2p, config.4p), where more than one protocol module may be assigned to the same processor, and where messages are shepherded through all of those modules before a processor is relinquished.

These configurations are chosen to cover a varied range of expected throughput and latency, where differences in both arise primarily from the distribution of processing loads across processors: the bigger the difference between the maximum and minimum processor loads, the larger the waiting time in the CI; the better balanced the load, the better the throughput.

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

The following factors are varied across experiments, consistent with other studies reported in the literature[15, 9]: (1) *protocol task assignments* (see Figure 4), (2) *number of concurrent connections*; (3) *message size*, (4) *protocol processing requirements* (reliability, checksum, encryption), and (5) *transmission rate*, either bounded (10Mbits/sec $\pm$ 0.05%) or unbounded. The metrics of interest are: *connection throughput*, *message latency* (i.e., for the outgoing path, time since the message is copied into the CI buffers and until it is sent to the CI), and *efficiency* (overall throughput by number of processors). Each experiment is run until the metrics of interest reach the 95% confidence interval.

A number of basic insights may be derived from these experiments:

• Pipelined configurations yield better performance under high NI loads. Consider the measurements presented in Figure 5: at high loads (number of processors), connectional (config.c) or message (config.m) parallelism experience performance degradation since both tend to generate higher contention for NI services or for interconnect access. This is due to the larger number of processors directly interfacing the NI. With pipelined configurations, such contention may be avoided by assigning only one or two processors to directly interface the NI (e.g., config.2p, config.1p).

• Well balanced quasi-pipelined configurations show superior performance for a large number of connections. This is because waiting time increases more slowly for better balanced configurations. For instance, in Figure 6, config.2p is better balanced than config.4p and config.1p, consequently exhibiting superior throughput and latency when more than 5 connections are active. However, for reduced processing requirements, the better 'balanced' configuration shows worse latency and no significant throughput improvement: in Figure 6, for less 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 $100 \mu sec$.

• Quasi-pipelined configurations on a small number of processors are more efficient than other types of config-

	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.....			

Latency

Max.PE Load

Figure 4. Examples of CI configurations

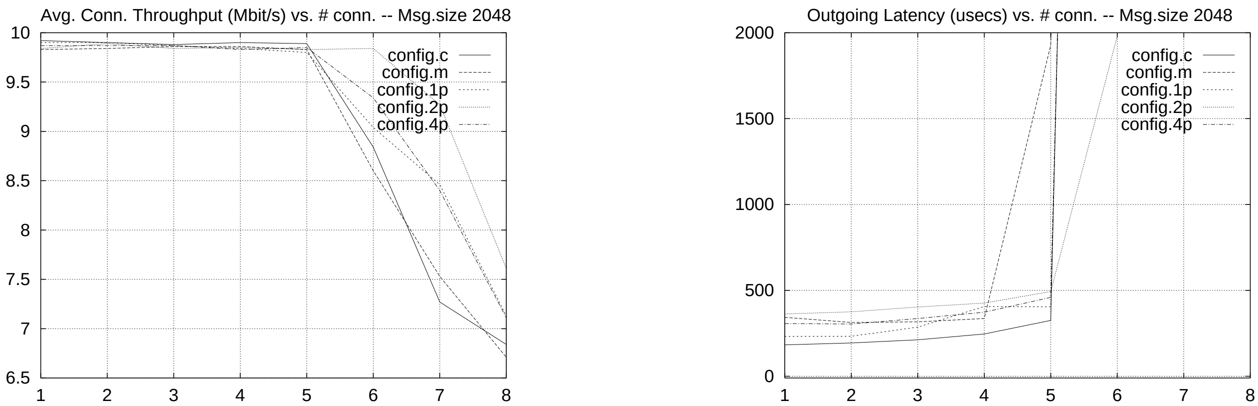


Figure 5. Effects of NI bottleneck (10Mbits/sec, unreliable, no checksum, no encryption)

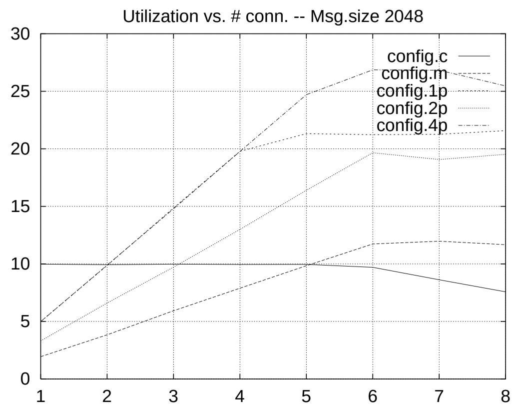


Figure 7. Configuration effects on efficiency (unbounded rate, unreliable, checksum, no encryption)

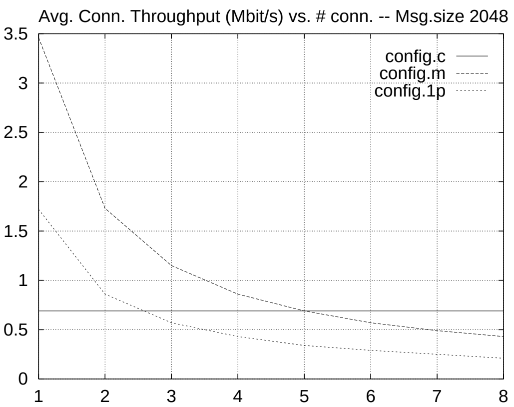
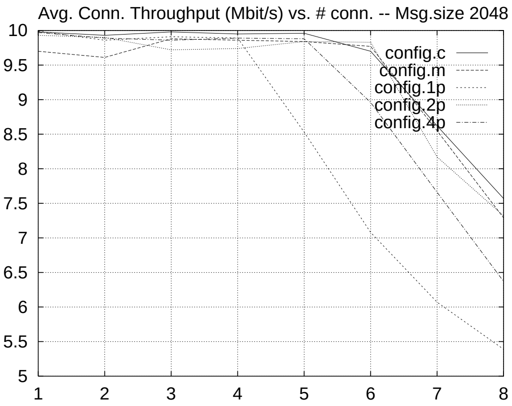


Figure 8. Parallelization effects on computationally intensive protocols (unbounded rate, unreliable, no checksum, encryption)

urations (see Figure 7). Comparing configurations with the same number of processors, a quasi-pipelined configuration (e.g., config.2p) can support more concurrent connections than a connectional parallel configuration (config.c), and it can enable smaller waiting times (and better throughput) than a message parallel one (config.m).

• **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 the involvement of all processors assigned to



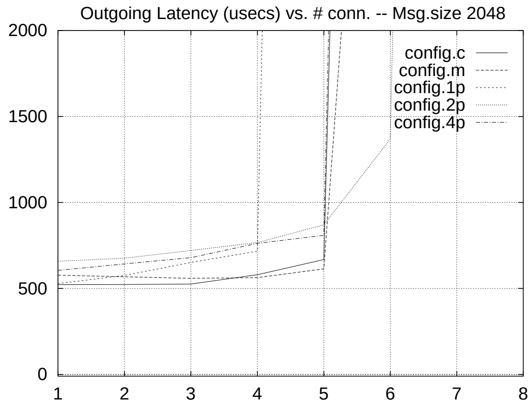


Figure 6. Effect of load distribution (unbounded rate, unreliable, checksum, no encryption)

the CI in computationally intensive tasks. It also benefits from locality in message processing.

Conclusions from experimental results. The main idea conveyed by the experiments discussed above is that no single configuration can provide for 'the best' performance under variable application requirements (e.g., rate or message size). This suggests that a CI should have the ability to perform on-line (auto-)configuration. More specifically, the results presented above demonstrate some properties of protocol configurations based on which heuristics may be developed. First, with message parallel configurations, the perturbation induced by other connections can be significant (especially in the presence of bursty traffic and long message sizes), whereas with connectional parallelism, such perturbation is completely avoided. However, message parallel configurations are appropriate for servicing multiple, similar connections requiring long service times. Second, quasi-pipelined configurations are more easily structured to match the required performance guarantees for all active connections (e.g.,by isolating incoming and outgoing streams, by grouping connections with similar QoS, or by balancing load). Last, despite poor processor utilization, connectional parallelism might be an acceptable choice when latency and predictability requirements are very strict.

4 Auto-Configuration

Configuration decisions may be made at the time of connection establishment and dynamically, at the time 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 (i.e., the mapping of module instances to available processors). The most important parameters of the decision model developed in this research are:

• *Connection requirements*, which are described by the set of modules on the protocol stack, traffic and QoS requirements, and guarantees (rate and latency). Traffic requirements include rate (pick, average), averaging interval, and maximum message size. These may vary in time, in response to detection of changes in connection traffic by the monitoring mechanism.

• *Connection configuration*, which is defined by the current assignment of modules to processors.

• *Current CI and network load*, which are defined by the estimated utilization and waiting time for the NI and for each processor to which COMM^{adapt} is mapped. The estimation is based on the observed traffic.

A configuration is considered "acceptable" if for all active connections, the estimated performance matches the *actual* requirements. When establishing such actual requirements (ie., traffic levels), we make the assumption that the near past is a better model for the future than the connection-time specifications. This enables us to accommodate new requirements by using the reserved but unused resources of other connections resulting from overestimated connection-time specifications or from runtime traffic fluctuations.

The load assignment heuristic generates CI configurations where each connection receives a specific type of assignment, either connectional, message parallel or quasi-pipelined. This is different from the unique configuration approach considered in our experiments (see Section 3), but it is a natural way of concurrently supporting a large variety of communication requirements.

The following sections briefly present the heuristic and the configuration procedures, while details including the evaluation in the context of a synthetic applications, appear in [7].

4.1 Connection-Time Configuration

The goal of connection-time configuration is to allocate resources to a new connection such that this and all other connections will meet current traffic and QoS requirements. 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 the specific overheads, (2) evaluating the levels of guarantee at source and destination hosts, and considering what the underlying network can offer, (3) informing the user about achievable guarantees. A host approves the connection provided an acceptable configuration is found. Otherwise, it returns feedback on the maximum possible level of performance.

When evaluating the acceptance of a certain connection, the CIM attempts to assign the connection all necessary resources. 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.

In the first phase, the goal is to achieve low latency, while also meeting the required rate and reducing possible effects on other connections. 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* and 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.

In the second phase, the goal is to achieve rate requirements beyond those achievable in the first phase while the effects on other connections remain small. 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 (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., by 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 5, config.m) or if the load on the corresponding processor set is too high.

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 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.

4.2 On-line Configuration

On-line configuration is triggered by the QoS evaluation mechanism whenever some connection receives inappropriate service or is shut down. In our context, poor service is caused by some connection failing to comply with its guaranteed traffic level.

The goal is to accommodate the load increase by taking advantage of currently unused resources. The misbehaving connection is subject to the same load assignment heuristic as for connection-time configuration is used (see Section 4.1). 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 correct the traffic (by application-level adaptation) or negotiate new communication requirements.

Note that at each connection end-point, the 'host' is allowed to react independently when inappropriate service is noticed, since it is very likely that different configurations, loads or protocol processing overheads exist at the two communicating sites[2]. Coordination among different hosts is needed only when the user explicity renegotiates traffic and QoS requirements.

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

5 Contributions and Related Research

The goal of this 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

[2] We are currently considering extensions for multicast communications.

paper is based on dynamic CI auto-configuration decisions. Benefits derived from such runtime configuration include: (1) improved processing and network resource utilization vs. user-specified configurations and connection-time configurations, (2) improved response to unpredicted or unlikely requirements, (3) compliance with specific application needs by use of customized configuration algorithms. The novel results presented in this paper are:

- a heuristic for the allocation of protocol processing resources, able to guarantee service levels to well-behaved connections while simultaneously accommodating runtime traffic variations elsewhere;

- the COMM^{adapt} software library for implementation of configurable protocols, supporting user- and algorithm-directed dynamic configuration;

- specific insights on the behavior of different CI configurations, including (1) that pipelined CI configurations on a small number of processors yield performance comparable to message and connectional parallel configurations (in contrast to [14]) and (2) that the NI bottleneck may restrict the performance of highly parallel communication architectures proposed in previous studies[15, 9].

COMM^{adapt} differs from previous work in its support for constructing protocols that are auto-configurable at runtime in their functionality and performance. For instance, in the x-Kernel [6, 9], each connection is bound to a specific protocol stack at the time of connection establishment. [10] presents an environment for dynamically configuring protocol stacks, but no details are provided on the runtime configuration mechanisms. The ADAPTIVE system[14] enables only application-level adaptation decisions. The framework presented in [18] supports QoS requirements by connection-time configuration decided by the CI. [8] only comment on the necessity of runtime adjustments in the mapping of protocol modules to processors. The parallel STREAMS implementation in [4] offers only user-directed stream configuration, and the policy for runtime scheduling of communication related tasks ignores QoS requirements.

We consider that dynamic auto-configurability can provide benefits to protocol architectures as HOPS [5], F-CSS [18], or micro-protocols [1], different from the layered protocol model used in COMM^{adapt}. Toward this end, the CI model should enable the estimation of the resource requirements and the actual usage for each specific connection.

Our future work will focus on addressing the QoS requirements of complex distributed real-time applications by designing novel configuration heuristics that integrate CPU, buffer and bandwidth management.

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

References

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

[2] 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.

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

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

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

[6] 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.

[7] D. Ivan-Rosu and K. Schwan. Improving protocol performance by dynamic control of communication resources. *Georgia Inst. of Technology, GIT-CC-96-04*, 1996.

[8] 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.

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

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

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

[12] 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.

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

[14] 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.

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

[16] 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.

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

[18] 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.