How to Build A Gateway

-- C-Gateway: An Example

Lixia Zhang

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

April, 1987

Abstract

In packet-switched computer communication networks, a gateway is a device which connects two or more heterogeneous networks at packet transport level. This paper describes the design and implementation of a gateway, called the "C-Gateway", at MIT Laboratory for Computer Science. The C-Gateway aims at good real-time responsiveness, high throughput, and multiple, heterogeneous protocols accommodation. Thus it must resolve the commonly observed dilemma in network protocol implementations, that is, the conflict between the layer modularity and system performance. The C-Gateway employs a new system primitive, task (a scheduled procedure call), which gives fast system responses with a low system overhead, and facilitates the implementation modularity.

1. Introduction

Packet-switched computer communication networks have existed for almost two decades. Today the original motivations for connecting computers together are strongly pushing towards connections among various networks, to achieve an even broader range of information and resource sharing. The inter-network connections are accomplished by computers, called *gateways*. Many gateways have been built in the last few years; yet many more networks are awaiting inter-connections in the near future.

How does one build a gateway? Although much experience has been gained from practice, today there are still no general rules that one can follow. In this paper we introduce the design and implementation of a typical gateway, called the C-Gateway (CGW in short). We hope that our experience with building the CGW will help identify general requirements and design principles of gateway implementations.

We begin by specifying the goals of the CGW design, together with a briefing on the CGW's history and current state; we then proceed with a detailed description on the CGW's design and implementation, followed by a summary of our experience.

2. C-Gateway: History and Current Status

2.1. Why Need a Gateway

The CGW was designed and implemented at MIT Laboratory for Computer Science during 1980-1982. The lab has long been active in the computer networking research. And by the late 1970's, various types of networks were running in the lab, with various hosts speaking different protocols. Some of the networks were locally designed and

This is a revised version of the paper published in *the Proceedings of the Second International Conference on Computers and Applications*, June, 1987.

implemented, such as a high speed token-ring network and CHAOSnet; some were from external research networks, such as a number of ARPANET switches (called IMPs, the Interface Message Processor); the rest came from the commercial market place, such as Ethernets. Among the protocol languages hosts spoke, there were CHAOS protocol [7], the ARPA Internet Protocol suite [8], and the PUP protocol [1] (a close predecessor of XNS, Xerox Network System protocol [11]).

As a natural next step, work started on connecting the heterogeneous networks together, to create a multiple protocol backbone, so that the hosts, no matter what protocols they spoke, could all share the underlying network hardware for data transmissions. The CGW was designed to accomplish this goal.

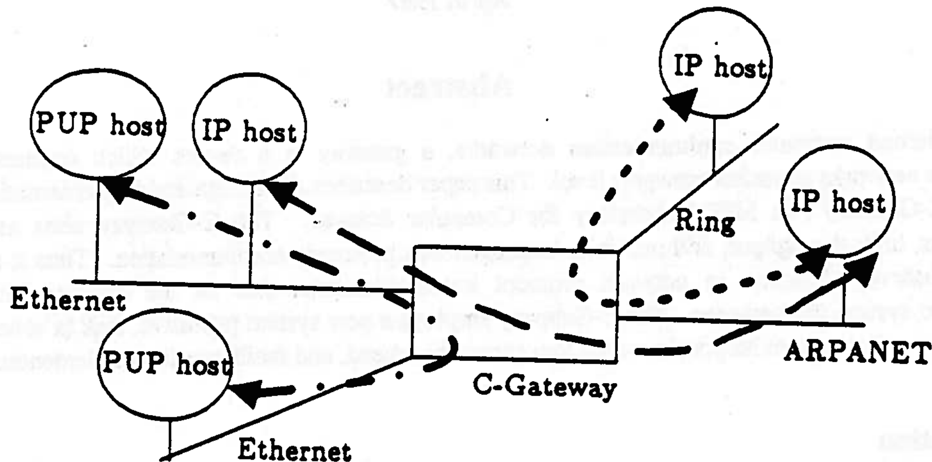

Figure 2-1: The goal of CGW: Sharing network hardware by multiple protocols.

2.2. Design Goals

At packet transmission level, a gateway appears as a host in each network it connects to. But instead of being the ultimate source or destination of data packets, it forwards packets from one network to another by encapsulation [10]. In particular, the CGW design had the following goals:

1. Considering the mega-bit per second speeds of the networks to be connected, the gateway should have very good real-time responses.
2. The gateway should be able to connect networks of different types. It should also speak different protocols at the inter-network layer, to forward packets of different protocol families.
3. It is important that the implementation follow the protocol modularity discipline, so that the system can easily accommodate the protocol heterogeneity, and provide a flexible configuration frame that can be easily tailored to a wide range of network environments. The implementation should also be in a high level programming language, to facilitate the portability and future extensions of the system.
4. Because of the capacity limitations of available hardware at the time, namely the memory space restriction, the implementation must be small in size.

2.3. Building Blocks

The C-Gateway is coded in the programming language C, implemented on top of the Micro Operating System [5], and running on an LSI-11 microprocessor. The selection of the hardware and the system was not a coincidence. LSI-11 is a small, inexpensive, and widely available machine; more importantly, off-shelf interface hardware for most of the networks to be connected could be attached to the LSI-11's Q-bus. The MOS was a small, simple

operating system designed for computer network applications. The choice of the C language was due to the flexibility of the language, the efficient object code, and the availability of a cross compiler.

2.4. Current Status

The CGW design and implementation successfully met all the specified goals. Today there are hundreds of CGWs running in universities, research institutes, and commercial companies all over the United States. At the network level, the CGW supports ARPANET, Ethernet, Proteon token ringnet, IEEE 802.5 token ringnet, Corvus Omninet, and synchronous serial lines; the X.25 protocol is currently being implemented. At the inter-network level, the CGW now supports the DoD Internet Protocol (IP)[1] [8], Xerox Network System Protocol (XNS) [11], the DECNET (Phase IV) routing layer protocol [3], and CHAOSnet Protocol [7]. Support to other protocol families, especially the OSI protocol suite, are also under consideration. As will be seen later, the basic CGW system frame can accommodate an arbitrary number of different networks and inter-network protocols.

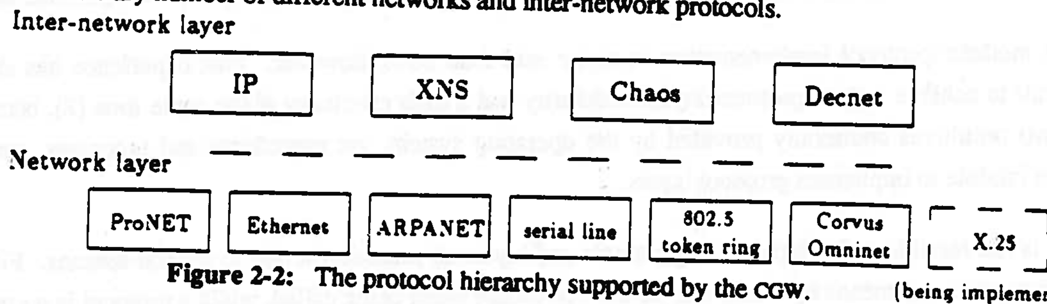

Figure 2-2: The protocol hierarchy supported by the CGW.

The CGW was made a commercial product by Proteon Inc. during 1985. Since then the implementation has been moved onto an MC-68000 microprocessor which has a faster CPU and a larger memory space, and has gone through a number of changes to further improve the performance. The main cove of the system, which was developed at MIT, remains intact, and will be described in the rest of the paper.

3. C-Gateway Design and Implementation

The MOS system is tiny and simple. It uses a round-robin process scheduling algorithm, with no preemption; all processes in the MOS run on a single address space. The simplicity alone, however, does not necessarily imply a good real-time property or high efficiency, as are required by the CGW. In fact we will show next why it does not, and how the CGW was designed to get around the MOS, in order to achieve the required properties.

3.1. Layering, Modularity, and Performance

A modular implementation is of the first importance for a gateway system; how to make one is the most valuable lesson we learned through the CGW experience. Prior to the CGW design, an prototype gateway was implemented without modularity consideration. It performs all packet processing functions in a single process, to save the process switch overhead, and processes each packet in a jumbo single step, starting from the reception till sending the packet out. Because no urgent event can be handled during this time period (about 4-5 msec), such an implementation results in poor responsiveness to the bursty traffic in a high speed LAN environment. Moreover, the unstructured implementation makes it very difficult to configure a gateway system for different network environments, or to add

[1] The implemented IP protocol module includes IP, ICMP [9], GGP [4], and EGP [6].

in new network protocols.

In general, modular programming is recommended for any system with a non-trivial size; in particular, since network protocols are specified as being divided to *layers* [13], implicitly each layer should be implemented as a distinct module of some sort, so that one can replace a layer with a new implementation without affecting other layers, so long as the interfaces in between do not change.

The challenge the CGW faces is more than just the layer modularity. In addition to the three protocol layers, the link, network, and inter-network layers, that a packet transport gateway must implement, at each layer the CGW must also accommodate multiple protocols, which may use any of the protocols in the neighbor layers. Keeping both the layer modularity and the protocol modularity is of essential importance, in order to make the CGW easily configure a specific realization for each different network environment, and easily accommodate environmental changes.

A modular protocol implementation is easier said than done, however. Past experience has shown that it is difficult to achieve a good protocol layer modularity and a high efficiency at the same time [2], because neither of the two primitives commonly provided by the operating system, viz procedures and processes, can be used as a proper module to implement protocol layers.

It is not feasible to implement the protocols and layers as procedures, due to several reasons. First, procedures do not provide any means by which a layer can run except when being called, while a protocol layer normally has its own activities independently from other layers, for example the network layer protocol may need to generate routing update messages periodically. Second, a procedure caller must suspend its own execution until the call is returned, however this may degrade its real-time responsiveness. Furthermore, a called procedure, after the execution, returns to the caller, whereas a protocol layer, after running, may want to activate some layer other than the one who called. For instance, assuming that a network input handler calls the internet forwarder to forward a packet, in this case we do not want the forwarder to return to the caller; instead, it is the network output handler that should be activated to take over the packet. A procedure-centric model best fits a hierarchical system, but does not provide a control flow that matches the packet flow in a gateway.

It is not feasible to implement protocols and layers as processes either, although this is a common approach taken by most protocol implementations. Processing a single packet usually requires passing it up and down through several protocol layers, while the process switching always involve a certain amount of system overhead. In addition, because the MOS employs a non-preemptive process scheduler, if a higher layer protocol is granted the CPU and runs for a long time period, lower layer processes would be blocked and fail to promptly handle real time events, such as new packet arrivals, which in turn would result in packet losses. It is possible to modify the MOS and make processes pre-emptable, but then the added overhead and complexity would, conceivably, reduce the gateway throughput.

It was clear at the beginning of the CGW design that a new system primitive was needed to resolve the conflict between the modularity and the performance. What should it be? To find out the answer, we need first to come up with a good gateway functional model.

3.2. Gateway as an Assembly Line

Imagine the process in a simple way, data packets pass through the gateway in a way similar to products passing through an assembly line. Packets entering the gateway are processed at each stage, then dispatched to the next, or to the outbound channels. In this model, gateway operations are packet-driven: it is packets that flow to each stage and trigger the operations.

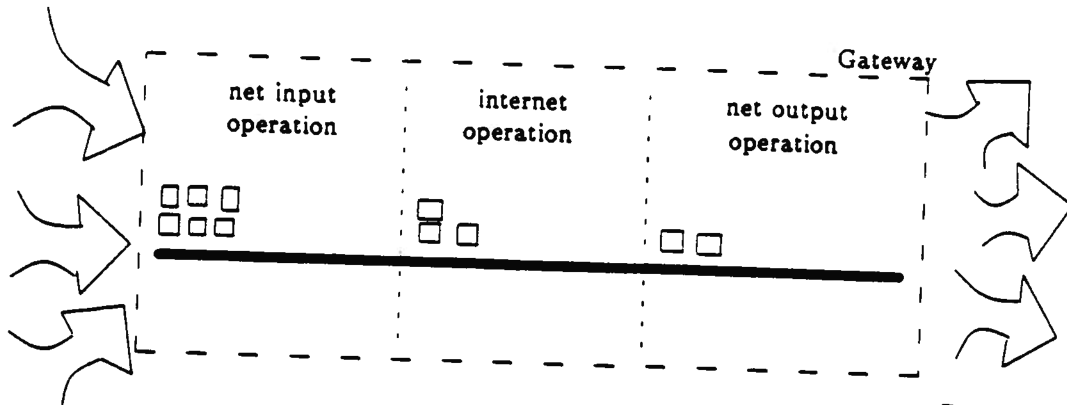

Figure 3-1: Gateway functional model: an assembly line.

It looks clearer now that the needed primitive should match to the operations on packets. Further, on a real assembly line operations at each stage go in parallel, while in the gateway all the operations will be performed by the single CPU. This implies a need for good operation scheduling, for the CGW must quickly respond to real time events, and process them in an order corresponding to each one's urgency. The CGW introduced *task* as the basic packet operation block, as we see next.

3.3. Task - The Basic Component in C-Gateway Implementation

A task is a dynamically generated request to perform a operation on packets; the running order of tasks should be assigned according to the urgencies of different operations.

The task is implemented as a scheduleable procedure call. The CGW creates a new data structure, called the *Task Block* (TB), to bind together all the information needed for running a task, as shown in Figure 3-2.

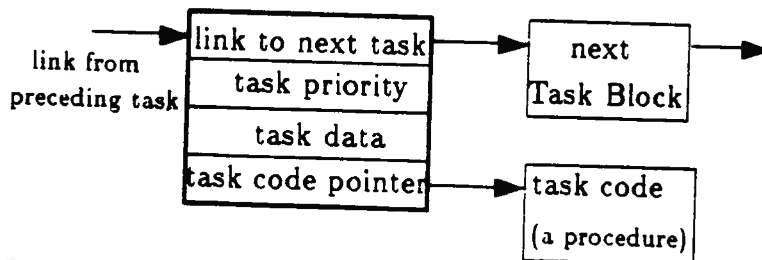

Figure 3-2: A Task Block.

The priorities of tasks are statically assigned according to the different time constraints of events they handle, as well as the lengths of their runtime. All tasks together create a priority-ordered running request queue. The gateway always picks up a new task from the head of the waiting queue to run. For the sake of implementational simplicity and efficiency, tasks are non-preemptable.

To achieve fast responses in a non-preemptable system, the running time of each task must be kept short. This implies that operations with long durations must be cut into smaller pieces. The CGW naturally divides the operations at the protocol layer boundaries, which both matches the layer modules and gives a fair division of work in each part. Forwarding a data packet consists of three tasks:

1. network input handling, which checks for network transmission errors, strips off the network header, and then passes the packet to a proper internet protocol forwarder;
2. internet forwarding, which performs the internet protocol error checking and routing functions; and
3. network output handling, which encapsulates the packet in the out-going network protocol, and turns the packet over to the device driver.

One task can schedule new tasks while running. When a network input task finishes with a data packet, for example, it schedules an internet forwarding task to run; or if one is already scheduled, it merely appends the packet to the packet queue of the forwarding task. The internet task will, in turn, schedule the network output task, so on and so forth. Packets flow through each task; there is never data copying inside of the CGW. By analogy, a task looks very much like an worker on an assembly line: when he finishes his part of the work, the worker wakes up the person at the next stage, who may have been idle for a while and fallen asleep, to continue on, or if that person is already busy, then just passes the product to his workbench. The CGW implements all the data packet processing tasks in one MOS process, named Gateway, to avoid the process switch overhead and scheduling delay completely.

How is a task originally scheduled at the arrival of a real-time event? The CGW introduced *handler* to bind an event to the task(s) handling it. A handler is a small subroutine; one handler is written for each possible network event: I/O completions, transmission errors, etc.; and handlers have a running priority preceding all the tasks. The MOS system signals I/O events by sending signal messages to processes; each process has a queue of all messages addressed to it. Whenever its message queue is non-empty, the process Gateway locates and runs the corresponding handler of each message, which checks and decides whether one or more tasks need be scheduled to process that message. For example, a packet arrival triggers a MOS signal message to be sent to Gateway; the handler of this message will then schedule a network input task to run, or add the new packet to the packet queue of that task if it is already scheduled, or discard the packet if there is a hardware error or a buffer shortage. Gateway keeps looking in its message queue until all the messages have been handled, then turns to run the scheduled tasks, and will re-check the message queue after finishing each packet operation.[2] All its information about its task queue, handler array, etc., is kept in its database, *Handle-Task*, as shown in Figure 3-3.

By introducing the *task*, the CGW achieves the following desired properties:

1. Because message handling has the highest priority, all event signals are processed as soon as possible. Even though a newly arrived signal may wait for a running task to finish, all tasks have a guaranteed short running time (less than a milli-second) by the coding convention.
2. Since the MOS does not order messages by priority, the process message queue is fetched in a FIFO order, but the messages are processed in a priority order by the prioritized task scheduling.
3. The implementation preserves the protocol layer modularity. Each protocol (at one layer) is a module of an interface procedure and one or more tasks that fulfill the specific protocol functions; each protocol layer consists of a set of protocol modules. Inter-layer communications are done through calling the interface procedures, with data passing through the shared memory space (provided the

---

[2] It is possible that one task may run for long, not because of the long operation but because of a long packet queue to be processed; to avoid this, the task checks Gateway's message queue after processing each packet, and reschedules itself if there are new messages.

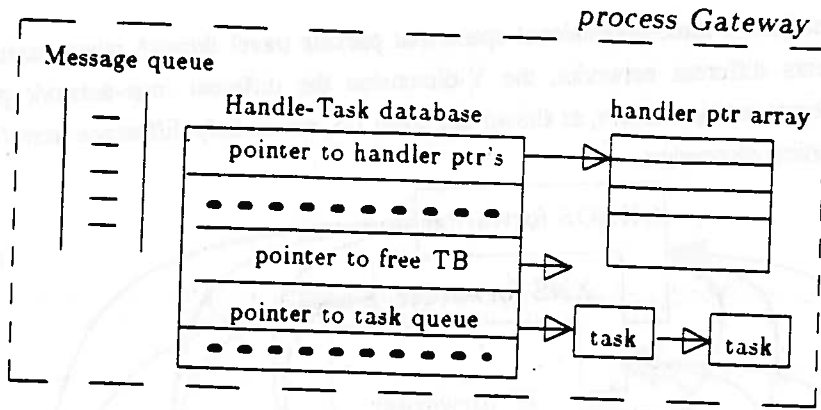

Figure 3-3: The message queue and handle-task database of the process Gateway.

MOS) (see Figure 3-4).

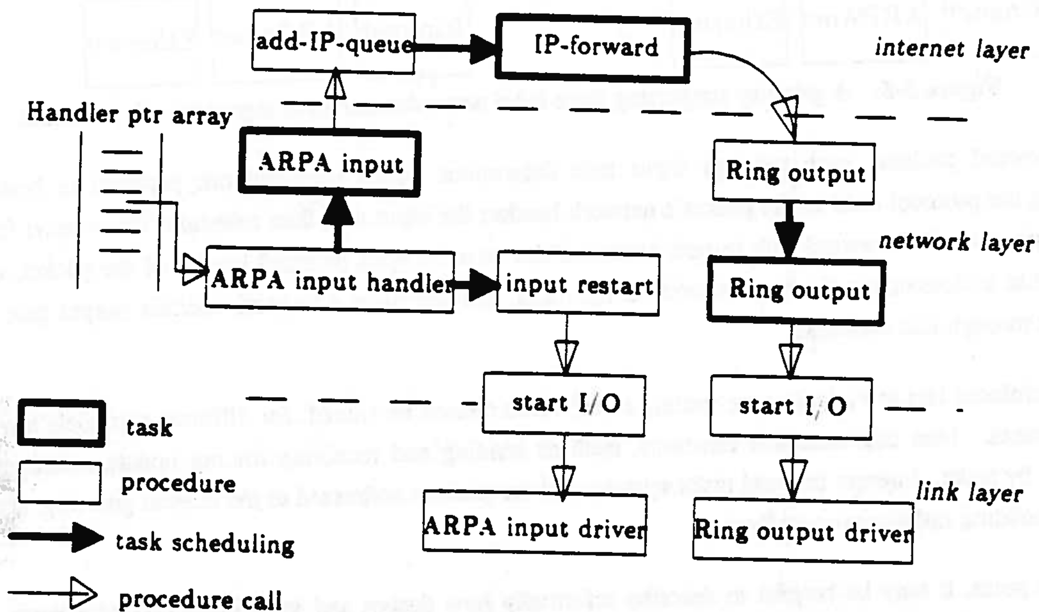

Figure 3-4: A snapshot of task scheduling and across layer procedure calls upon a packet arrival, with ARPANET and Ringnet being the inbound and outbound networks, respectively.

This last point gives the CGW a modular configuration; a gateway realization for a specific network environment can be easily configured by plugging in the needed protocol modules at each layer. We have more to discuss on this point next.

3.4. Supporting Multiple Protocols

The implementation of a gateway is often incremental, in the sense that after the first launch, requests for supporting more heterogeneous networks and protocols will keep coming continuously, therefore the implementation will keep growing by adding new networks or protocols. One of the top gateway design goals is a flexible frame structure; together with a well-disciplined, modular implementation, later modules can then be easily plugged in to support new protocols.

The CGW was built with such a good system frame. Protocols and layer boundaries are clearly preserved. To

illustrate, let us imagine a three-dimensional space that packets travel through when passing the CGW. The X-dimension represents different networks, the Y-dimension the different inter-network protocols, and the Z-dimension the different protocol layers, as shown in Figure 3-5, whose only difference from Figure 3-1 is by giving protocol layer a vertical dimension.

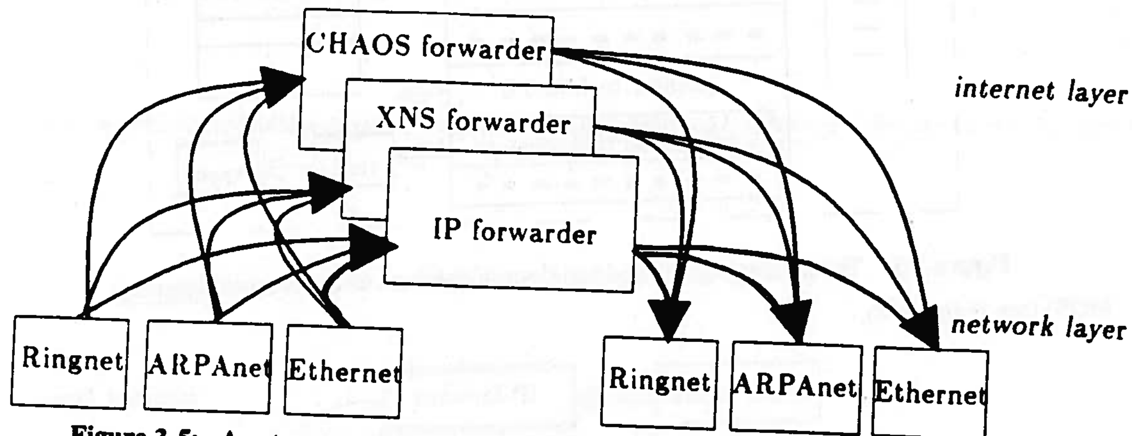

Figure 3-5: A gateway supporting three local networks and three inter-network protocols.

To forward packets, each network input task determines which inter-network protocol is being used by examining the protocol field in the packet's network header; the input task then schedules the internet forward task of that protocol. The forward task in turn examines the inter-network protocol header of the packet, consults its routing table to determines the next network in the route, and schedules a network-specific output task to forward the packet through that network.

Each protocol has to keep its own routing table which cannot be shared, for different protocols have different address spaces. Non data-transport functions, such as sending and receiving routing update messages, are also performed by tasks. Internet forward tasks separate out the packets addressed to the current gateway, and schedule the corresponding tasks to process them.

At this point, it may be helpful to describe informally how design and implementation of a gateway system seems to proceed, as typified by the experience with the CGW. The first step is to get a highlight model of the system operation; the second step is to outline the basic frame structure of the system; the next decision has to do with choosing a system primitive that can be used as the building block to implement protocols and layers in a modular way. In the CGW case, packets passing through a gateway is modeled as products passing through an assembly line, which shows a packet-flow driven system consisting of a number of operation stages; a new system primitive, task, is then defined to perform packet operations. Once these design decisions have been made, the general shape of the system is fully determined, and more detailed design decisions can be undertaken.

Up to now we have discussed the functional aspect of the CGW. In the next few subsections we discuss the packet buffering strategy and programming style of the CGW.

3.5. Packet Buffering in the CGW

A few lessons are learned from the cgw packet buffering strategy. Notice that there potentially exists a packet queue with every task, although it may be empty most of the time. Recall that all long functions are broken down into smaller pieces of tasks; each task, because it is scheduled to run rather than running immediately whenever being called, must have a place to hold its packets temporarily. What is the impact of this buffering?

First, by dividing the packet processing into small tasks, the CGW is able to suspend the current processing at task boundaries and turn to handle new events. Although the tasking mechanism does not directly increase the gateway throughput, it achieves a good real-time response with a minimal overhead, for it gets away with the complexity associated with process switching and pre-emption, as commonly seen approaches for good system responsiveness. The low overhead boosts the CGW's high throughput.

Second, as a consequence of suspending the current processing, each task prepares a packet buffering queue, which resembles the workbench in front of each worker on an assembly line. Tasks are stateless; it is those packet queues in front of tasks that show the running state of the gateway. By looking at these queues, one can easily find out where resides the bottleneck operation; and when congestion occurs, decisions on buffer pool management can be easily made. In comparison, an alternative way of buffering would be to attach one packet to one run of a task. This approach was rejected, because it embeds packets in the tasks, which not only increases task scheduling overhead (one scheduling per packet per operation), but made it difficult to see the state of the gateway -- one cannot easily find how many packets there are, or where they are.

Third, the size of system buffer pool is a crucial factor in the gateway performance, especially in an LAN environment, where the data traffic is highly bursty but has a low average volume. The original implementation on the LSI-11 processor could provide no more than 20 packet buffers; when data packets flow into the gateway in bursts, this small pool may overflow easily, resulting in packet losses. Research on network congestion control also concludes that an adequate buffer space is a necessary condition for effective traffic control [12]. The move to MC-68000 processor is an important step towards further enhancing the CGW functionality and performance.

3.6. New Data Structures

It is worthwhile mentioning some important data structures and coding conventions in the CGW. We have seen the structure of the *Task Block*. Now let us look at another core data structure of the CGW, the *NET*.

A *NET* is implemented with the data type *Struct* in the C language (as is the *Task Block*). Compared with other high level programming languages, C does not have a rich set of data or control structures, nor is it strongly typed. The data type *Struct* in C is generally used to contain pure data only. In the CGW implementation, however, *Struct* becomes a useful tool for building modular programs. Modeling a process or a device as an object, the design exploited the idea of so-called object-oriented programming, that is, the CGW binds all the information of an object, such as data, status, as well as procedures which manipulate the object, together in one block. Object-oriented programming helps build a clear, simple, well structured, and modular system. It also facilitates efficient parameter passing -- objects and their operations are passed together so that the receiver can manipulate an object without knowing its specific representations.

NET is a typical example: it binds an abstract object, a network module, with the operations performed on that

object, e.g. the driver of the network. There is one *NET* object for each network, which contains network specific information: the network drivers, the lengths of input and output queues (at the driver layer), pointers to packet queues at the network layer, the network status, statistical information, and so on. In the CGW, object types are generic, such as *NET*, while realizations of objects are specific, such as ProNET and Ethernet, which are very different. Objects of the same type are bound with different operations by using the procedure pointer feature of C. All the major structures in the CGW are coded in the similar way.

3.7. Library and Code Sharing

The size of the PDP-11 CGW object code, including the MOS system, is about 20 Kbytes, depending on how many networks and inter-network protocols being supported by a particular gateway realization. Such a small system size is partly due to careful programming, and partly due to code sharing among different networks. The CGW builds a subroutine library to resolve the conflict between the desired functionality and the limited memory space, which was once a serious problem. The library contains commonly used subroutines that perform the bulk of the network interface handling work, such as allocating/deallocating packet buffers. All networks share the library code together: in each *NET* object, if a pointer to a network-specific subroutine is unspecified, as in most cases, a corresponding library routine will be used instead. Private routines are coded only for the special features of individual networks.

Although the adequate memory space is no longer an issue today, the code sharing library is still considered a good approach, for it reduces the amount of programming needed for adding new network protocols.

4. Summary

A gateway's basic function is to forward data packets across network boundaries. This function, although sounds simple, is not easily accomplished. The CGW is to operate in a high speed LAN environment, and to support multiple heterogeneous networks and protocols. In the CGW design and implementation, we found it particularly challenging to achieve good real-time response, high performance, and protocol modularity at the same time, as required by the CGW. Our solutions and experience are summarized as follows:

1. The CGW implementation largely avoids the concept of processes; instead, it uses a new system primitive, the task, to carry out gateway operations. Eliminating process switching overhead helps the CGW achieve a high throughput. The throughput is 300 packets per second on an LSI-11 machine, and above 500 packets per second on an MC-68000.
2. The real time responsiveness is of first importance for a gateway running in an LAN environment, where the data traffic is highly bursty but has a low average volume. The tasking mechanism fits this traffic feature well; it gives fast responses to real time event independently from the system load conditions.
3. The CGW achieves real-time responsiveness by limiting task runtime lengths, rather than by the usual approach of pre-emption mechanism, thereby avoiding the overhead and complexity associated with the latter.
4. The CGW breaks long processing into small tasks in a way corresponding to the protocol layer boundaries. Therefore the implementation preserves protocol modularity, and provides a modular configuration which can be easily tailored to meet a wide range of network environments.

Acknowledgments

I am grateful to J. Noel Chiappa, the implementor of the CGW, Mark Rosenstein, and Dave Feldmeier for their help during the writing of this paper. I am especially grateful to Dave Clark, whose insightful comments shaped the paper to its current state. Finally, I would also like to express my thanks to the anonymous reviewers for their valuable comments.

References

[1] Boggs, Shoch, Taft, Metcalfe.
Pup: An Internetwork Architecture.
IEEE Transactions on Communications , April, 1980.

[2] David D. Clark.
An Alternative Protocol Implementation.
Computer System Research Group RFC-223, MIT Lab for Computer Science.
May, 1982

[3] Digital Equipment Corp.
DECNET Digital Network Architecture (Phase IV) General Description
.
AA-N149A-TC.

[4] R. Hinden et al.
The DARPA Internet Gateway.
ARPA Internet RFC-823.
September, 1982

[5] James E. Mathis, Keith S. Klemba, and Andrew A. Poggio.
Terminal Interface Unit Notebook.
Project No. 6933, SRI International, April 1979.

[6] David L. Mills.
Exterior Gateway Protocol Formal Specification.
Network Information Center RFC-904, SRI International.
April, 1984

[7] David A. Moon.
Chaosnet.
Technical Memo 628, MIT AI Lab., June 1981.

[8] J. Postel.
DoD Standard Internet Protocol.
Network Information Center RFC-791, SRI International.
September, 1981

[9] J. Postel.
Internet Control Message Protocol.
Network Information Center RFC-792, SRI International.
September, 1981

[10] J. Shock, D. Cohen, and E. Taft.
Mutual Encapsulation of Internetwork Protocols.
Computer Networks 1981(5):287-300, 1981.

[11] Xerox System Integration Standard 028112.
Internet Transport Protocols.
December 1981.

[12] Lixia Zhang.
Congestion Control in Packet-Switched Computer Networks.
In *The Proceedings of the Second International Conference on Computers and Applications (Bejing, China).*
IEEE, June, 1987.

[13] H. Zimmerman.
OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection.
IEEE Transactions on Communications 28(4):425-432, April, 1980.