Design Issues in System Support for Programmable Routers

Prashant Pradhan, Kartik Gopalan, Tzi-cker Chiueh
Dept. of Computer Science, SUNY Stony Brook

Abstract

Placement of computation inside the network is a powerful computation model that can improve the overall performance of network applications. In this paper, we address the problem of providing sound and efficient system support for placing computation in a network router. We identify a set of requirements, related to protection, resource control, scheduling and efficiency, that are relevant to the design of this system support. We have developed a system that attempts to meet these requirements, and have used it to write a router application that performs aggregated congestion control.

1 Introduction

The tremendous success of data networks can largely be attributed to the simplicity and robustness of the network's service model. By placing all complexity in the end-systems and providing a simple, stateless forwarding service, the network can provide a communication substrate that is resilient to link and node failures. Good end-to-end algorithms, like TCP for congestion control, have further contributed to the adequacy of this model, by being able to control network stability using purely end-to-end mechanisms.

However, there is quantitative evidence to show that the ability to place computation inside the network leads to a fundamentally more powerful computation model. To begin with, such a capability allows one to control how the network processes and routes packets of certain applications, affording various optimizations not possible with a static forwarding service [1]. Computation placed in the network also has the ability to exploit topological advantage, for example, recovery latency for reliable multicast improves by performing local recovery in routers [2]. Similarly, the performance of heterogenous receivers receiving media flows improves by using a transcoding video gateway [3]. Further, access to multiple flows belonging to an application (or *global context*), typically affords some global information that is useful to the overall performance of that application. For example, knowledge of congestion state probed by independent flows is useful for new TCP flows, leading to better overall TCP performance [4], whereas global knowledge of session ID to cluster node mappings in an SSL server cluster leads to improved connection throughput [5]. In many cases, the ability to place computation even in a restricted set of network nodes (e.g. edge routers) can provide a large subset of the benefits of this paradigm.

However, the success of this paradigm in real networks critically depends upon the existence of carefully designed system support for programmability in routers. The service provided by the network relies on the functionality provided by its routers. Without appropriate resource control and protection mechanisms, dynamically added computation can effect the performance and integrity of the system in undesirable ways. Moreover, a router is a massively shared system, and its resources are used by a large number of flows. This makes effective arbitration of the router's resources between these flows an important requirement. Besides correct design, efficiency is a key requirement for making it practical for performance sensitive applications to use router extensions. Our goal in this paper is to discuss some of the requirements for sound and efficient system support for router programmability.

In the rest of the paper, we shall use the term ”router OS” in place of ”system support for programmable routers” and the term ”router application” in place of ”computation placed in programmable routers”[1]. In the following sections, we describe the main requirements that we believe should be taken into account while designing a router OS.

2 Efficient Memory Protection

Memory protection is a basic requirement for maintaining system integrity in the presence of dynamically installed functions. A dynamically added function may not necessarily be malicious, but it may perform unintended operations that compromise the safety of the system. Since the execution of a function, in general, may be dependent upon the

[1] Note that the terminology is not entirely accurate since these entities could take the form of a language runtime and a program in the language.

environment it executes in, it may not be possible to exhaustively test it for safe operation. Thus in general, a ”trusted function” may not be safe unless there are restrictions on what kind of computation the function may perform.

It is possible to write functions in a restricted programming language that guarantees safe execution. For certain kind of functions it may even be possible to statically determine safety, even though they are written in an unrestricted programming language. However, our interest is in an approach which does not restrict the expressiveness of the language in which these functions are written. Our experiences in writing two router applications that involved TCP congestion control and splice mechanisms [5] [4] show that router application code can be of significant complexity. We feel that writing these applications in a restricted language would have been substantially more complex. The generality of possible router extensions makes it difficult to come up with a language that captures all intended forms of computation, while guaranteeing safety. The key goal then becomes to provide safety efficiently when router applications are unrestricted.

Efficient memory protection can be provided by utilizing the low-level hardware protection features of the processor architecture in question. Most general-purpose processors provide hardware primitives for protection, where all associated checks are embedded in the micro-architecture and thus do not incur any extra overhead. These primitives, when exploited at the lowest level, can provide efficiency as well as hard protection guarantees. We have been able to implement efficient protection domains in a router OS by utilizing the segmentation hardware of the X86 architecture [6]. The protection subsystem of our router OS exposes the segmentation hardware at a low enough level that router applications can use it easily, while keeping invocation overheads close to that of a protected function call in hardware. Similar approaches have been tried with other architectures as well [7]. In general, by tuning its protection subsystem implementation to the processor architecture, a router OS implementation can provide efficient as well as strong memory protection without compromising expressiveness of router application code. In spirit, this design principle is similar to that of Exokernels [8], which would argue for exposing hardware protection features to the application.

3 Performance Protection

We distinguish between flows that are bound to some router application, called *application flows*, and flows that are processed by the router's standard forwarding code, called *generic flows*. We call generic flow processing and control plane processing as the router's *core tasks*. The goal of performance protection is to protect the performance seen by the router's core tasks in the presence of application flows. Performance protection limits the scope of the impact that dynamically added computation has on flows going through the router : application flows perceive the end-to-end effects of placing computation in the router (as desired), while the presence of this computation is transparent to generic flows.

Performance protection has two implications on a router OS. Firstly, core router tasks must be bound to an appropriate *core scheduling context*. This makes core task processing explicit in the scheduler, allowing it to deliver the appropriate performance guarantee. Secondly, the protection policy for the core tasks must be chosen, which may be prioritization or sharing. Recent studies using WAN traffic traces and inter-domain routing message traces show that traffic patterns and control plane processing load in internet routers is bursty and largely unpredictable. Thus, core router tasks are characterized by high short-term processing bandwidth, even if the long term processing bandwidth requirement may be small. Thus, to provide true isolation to these tasks, prioritization is the appropriate scheduling primitive. Prioritization ensures that in a programmable router, the processing demands of core tasks will be handled with zero latency in the presence of router applications[2]. For generic flows and control processing, this essentially simulates a router in which there were no applications running. Moreover, if application flows are scheduled among themselves using a proportional share scheduler, they will adapt gracefully to short-term reduction in available resource bandwidth (system virtual time will not advance while the prioritized task is being run).

4 Event-Driven Control Flow

An important characteristic of many useful router applications is the use of functions that carry state across invocations. Protocol stacks are one example, where every ”layer” is a stateful function. Similarly, any router application that exploits global state across flows must use stateful functions. Typically, a single stateful function would be used by many flows. Similarly, a single flow would use several stateful functions that act like a ”processing pipeline” for the flow. This model localizes state in the functions, and carries a flow's invocations from function to function. This is in contrast with the ”thread” model, where it is expected that a thread executing on behalf of the flow has access to all the state. If different functions are in different protection domains (because some functions are privileged, or installed by mutually untrusted authorities), the thread approach must either resort to state sharing through an interface (since it cannot directly read/write it), or there should

[2]This of course depends upon the scheduler getting control, which may happen on a function return or a timer. See section 4.

be a mechanism for a thread to "cross" protection domains. The latter essentially takes the form of an explicit invocation, as proposed in [9] through descriptor passing.

Thus, we argue that the computation and composition model for router applications should be event-based, as opposed to thread-based. Besides providing a closer match to a computation model that uses stateful shared functions, a key advantage of an event-based model is that all invocations are *explicit* and *asynchronous*. Since invocations carry the identity of the resource principal making the invocation, the resource principal associated with a piece of work is always explicitly known throughout the system. This gives the scheduler complete knowledge of pending work in the system for each resource principal, and allows it to schedule work correctly. Further, by being asynchronous, every invocation provides an instant when the scheduler gets control, leading to tighter resource control than that allowed in a constant time-slice based scheduler. Note that at every scheduling instant, the scheduler can look for invocations made in the core scheduling context (section 3), and prioritize them.

5 Integrated Resource Scheduling

An application flow requires CPU cycles as well as link bandwidth from the router to meet its performance requirement. However, the router application can only specify a flow's requirement in terms of the amount of work required from each resource, and a single, global deadline (or rate) requirement. For example, for each packet of a flow, it may specify the CPU cycles required, the packet size in bytes, and a single deadline for the packet to get serviced. The application does not specify how deadlines should be allocated in the CPU and the link. This task is best done by the router OS that should figure out how to best deliver the overall deadline by allocating a per-resource deadline. We call this router OS function *integrated resource scheduling*.

We generalize integrated resource scheduling in terms of deadlines, since rate requirements can be mapped to deadlines. Thus, we assume that a flow asks for a deadline (d) for each of its packets, and specifies the amount of work required from the CPU (W_C) and the link (W_L). The goal of integrated resource scheduling is to split d into d_C, a deadline for the CPU, and d_L, a deadline for the link, according to an optimization criterion. We briefly describe a deadline allocation algorithm here. At any time, there are a set of requests admitted into the system, corresponding to a set of reservations in each resource. If a resource has capacity C and has admitted a set of requests where request i needs work W_i and has been allocated a deadline d_i, then the residual capacity of the resource is $R = C - \sum_i W_i/d_i$. When a new request comes for this resource, asking for an amount of work W, its minimal service time in this resource is $d_{min} = W/R$. If the sum of the quantity d_{min} for every resource is less than or equal to the global deadline of the task, then the request is admissible. However, if the sum is less than the global deadline, these deadlines can be *relaxed* such that each resource has some spare capacity left (Note that allocating a deadline of d_{min} in a resource corresponds to using up *all* the residual capacity of that resource). It is in this relaxation step that the system-wide optimization criterion comes in. For instance, if the optimization goal is to keep all resources equally utilized, so that the system keeps spare capacity uniformly available across all resources, then the deadline allocated in the heavily utilized resource would be relaxed more. In general, the relaxation algorithm tries to iteratively select a constant Y such that the residual capacity of resource i is $\beta_i Y$, where β_i encapsulates the optimization criterion. If uniform spare capacity is desired, β_i would be 1 for all i, otherwise it would reflect the ratio in which the resources are to be kept utilized. Such a mechanism should be an integral part of a router OS in order to achieve tight admission control for application flows.

6 Binding Resources to Flows

Typically router resources would be shared by a large number of application flows, which calls for appropriate resource arbitration. Moreover, many router applications would typically operate upon a *set* of flows belonging to a type of network application, as opposed to operating on single flows. In such cases, the router application would typically have an aggregate, as opposed to per-flow, performance requirement. This makes the task of accurately binding router resources to flows an important one. The *expressiveness* of the resource reservation interface determines how accurately router applications will be able to express their resource requirements. An inflexible interface may lead to coarse specifications, leading to underutilization of router resources. Likewise, an overly flexible interface may blow up the scheduling state in the system, while being a burden to a router application writer.

We believe that two key principles suffice to provide a simple and flexible interface.

1. Decouple execution contexts from scheduling contexts: This means that the interface should clearly distinguish between a thread of control associated with a flow, and the resource principal associated with it. Thus, an invocation made in the context of a flow should have two components : the identity of the flow, and the identity of the resource principal, which may be different in general.

2. Allow absolute as well as *symbolic* specification of resource reservations : A symbolic specification means a reference to another principal's resource reservation.

Thus, a flow f_1 may specify that it requires an overall rate of 100 packets/sec with each packet having 64 bytes (independent link reservation of 6400 bytes/sec), but shares the CPU with flow f_2 (symbolic CPU reservation).

These principles have two important implications. First, binding a flow to a resource principal now becomes an explicit operation. Second, resources can be shared on a per-resource basis, as opposed to an all-or-none basis (where either both CPU and link resources are shared, or none is shared). An example application where this is needed is a multicast application that transcodes incoming data on a link and distributes it over three output links. Each output flow requires its own context in its output link, and may even have distinct link rate requirements due to receiver heterogeneity. However, the transcoding operation is done once on a single copy of every packet, and hence the CPU reservation should be shared. This application can be implemented using three flows that share their CPU resource. One of the flows can make the absolute CPU reservation, and the other two can refer to this reservation symbolically. A cursory look at the example says that the same could be done by declaring one incoming flow that only reserves the CPU, and three outgoing flows reserving only the respective link rates. However, this would break the integrated CPU and link scheduling requirement described in section 5.

7 *Srishti* and Aggregate TCP

The ideas presented above have been incorporated in *Srishti*, a substrate for writing applications in a router that uses the X86 architecture for application flow processing. Using the above design principles, Srishti allows composition of router applications through stateful functions and flows. The functions are untrusted, preemptible functions that can be efficiently co-located with core router functions in a single address space. Flows are execution contexts, bound explicitly to resource principals using Srishti's API. All control transfer is explicit and asynchronous, and functions are called through references. These references are obtained by a *naming* service that acts like a dynamic symbol table of loaded functions.

We briefly share our experience in writing a router application over Srishti to perform aggregated TCP congestion control. TCP does not provide mechanisms to allow a new connection to reuse congestion estimates gathered by other connections that have used the same path. This forces a new TCP connection to always start from a conservative estimate of available bandwidth, causing short connections to never reach the correct value of the available bandwidth. Short HTTP connections can perform at significantly sub-optimal performance levels due to this, if there is a lot of opportunity for temporal and spatial congestion state reuse. ATCP is a router mechanism that allows congestion state reuse between TCP connections that are expected to share bottleneck links in the network[3], without changing end-system TCP implementations. The details on how ATCP implements its functionality and its evaluation on a real-world HTTP trace are available in [4].

While composing ATCP using Srishti's API, the most interesting choice is in how resources are to be allocated, and the *scope* of congestion state sharing. The scenario we envision is that ATCP is deployed in a router that serves a certain number of busy TCP servers, say from N different organizations. In this case, the scope of congestion state sharing is all TCP flows originating at these N servers, and the resource allocation goal is to be max-min fair to these N organizations. Thus the ATCP implementation uses N resource principals, to which incoming TCP packets are bound. One can choose to implement ATCP as a monolithic function that holds per-flow state ; or as K modules where there are K congestion sharing groups, each of which holds per-flow state only for the flows in that group. We have currently implemented ATCP as a monolithic function. Only one execution context is used, since all session state is centralized in one function. Since there are no blocking calls in the application, there is no need for multiple execution contexts to hide blocking latency.

8 Evaluation

We have implemented Srishti on a 400 MHz Pentium and tested it as a router with Intel eepro100 network interfaces. While the implementation uses a Linux skeleton, it depends more directly on the X86 architecture rather than on Linux[4]. In this section, we provide some microbenchmarks on the system that give some insight into the design decisions laid out earlier.

We begin with some microbenchmarks related to protection. A null router application function co-located in a lesser privileged segment of the core router kernel incurs an overhead of 325 cycles for a call and return. When the function also makes a protected function call to a core router function before returning, the overhead becomes 814 cycles. This is more than twice of the single call, due to additional overheads of saving all general-purpose registers. To see the advantages of co-location, we ran a ping-pong test between two null functions in different address spaces, incurring on overhead of 1360 cycles per call. This overhead would be higher in general, due to the cost of re-populating

[3]ATCP approximates this by grouping together flows destined to the same subnet.

[4]Code for Srishti and ATCP is available via anonymous FTP from ftp.sequoia.cs.sunysb.edu/pub/srishti.

flushed TLB entries with every address space switch.

The next measurement shows the role of event-driven control flow in providing fine-grained prioritization to the router's generic flows. We modified the eepro100 driver to use polling instead of interrupts, as would be true of a high-performance implementation on a PC-like platform. Thus, interrupt context is not used to process generic flows. We tried three ways in which the scheduler could get control in order to serve generic flows. In the first case, the scheduler only gets control at system timer interrupts (10 msec). This simulates a time-slice based scheduler, and a system that uses synchronous function invocations. The second case gives control to the scheduler only when application functions return, simulating asynchronous control flow, but with no timers. In the last case, the scheduler gets control every time a function returns or the timer fires, representing the finest scheduling granularity. The system continuously runs invocations whose running time is uniformly distributed from 3 msec to 21 msec in increments of 3, centered roughly around 10 msec. The router is fed with a uniform stream of packets with varying inter-packet gap. The metric that reflects the "disturbance" introduced in the forwarding path of generic flows is the standard deviation of the inter-arrival time at the receiver. As shown in table 1, the event-driven approach leads to lesser perturbation than a constant time-slice based scheduler, and the scheduler that combines events with time slices performs the best.

Sender Inter Pkt Gap	Timer Interrupt	Function Return	Timer OR Fn. Return
1.0	1.746	1.821	1.573
4.0	4.383	3.783	3.069
7.0	4.312	3.832	3.082
12.0	3.881	3.658	2.844

Table 1. *Standard Deviation (in msec) of received inter-packet gap for three ways in which the scheduler can get control from router applications.*

The final measurement shows the impact of integrated resource scheduling in providing tighter admission control. We assume two resources, each with a capacity of 1000 units/sec. Flows request these resources by asking for a rate of 1 packet per 100 msec, requesting 1 unit of work from one resource and 10 units from the other. Figure 1 shows how resource usage for each resource changes as flows are added, for the case when deadline slack is allocated in a fixed manner (algorithm 1), and when it is allocated in a load-dependent manner (algorithm 2). Algorithm 1 allcoates half the global deadline to each resource, leading to skewed utilization and stops admission control sooner

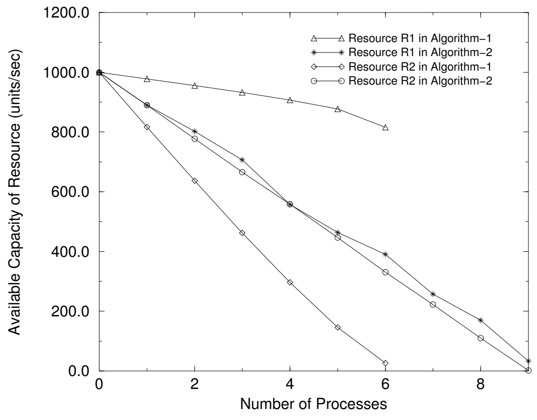

Figure 1. *Utilization of two resources v/s admitted flows for fixed allocation (algorithm 1) and load-dependent allocation (algorithm 2).*

than algorithm 2. Algorithm 2 tries to keep resource utilization balanced by allocating more slack to the more loaded resource.

9 Related Work

Recent interest in providing system support for router programmability has led to the specification of the NodeOS interface [10] which attempts to lay down implementation-independent primitives that a programmable router should provide. NodeOS *implementations* internally implement these primitives using substrates like language runtimes or specialized OSes [11], and expose the NodeOS interface to router applications. Placing our work in this context, the requirements we identify pertain to such NodeOS implementations. In other words, we expose some of the design decisions which are hidden beneath the NodeOS interface, but are important in making router programming a practical paradigm. Some of the requirements that we propose are generic, in the sense that they can be incorporated in existing implementations. For example, efficient memory protection primitives can be utilized to sandbox router plugins [12]. Similarly performance protection requirements for core router tasks, and integrated resource scheduling can be incorporated into any framework that supports scheduling. Stateful computation and event-driven control flow may not be possible in some systems, notably systems based upon functional languages. However fine-grained scheduling afforded by event-driven control flow can be supported in language runtimes by giving control to the language runtime at (a chosen set of) function calls.

References

[1] Wetherall D., Service Introduction in an Active Network, PhD thesis, MIT Laboratory for Computer Scince.

[2] Papadopoulos C., *et al*, An Error-Control Scheme for Large-Scale Reliable Multicast Applications, Proc. IEEE Infocom 1998.

[3] Amir E., *et al*, An Applical Level Video Gateway, Proc. ACM Multimedia 1995.

[4] Pradhan P., *et al*, Aggregate TCP Congestion Control Using Multiple Network Probing, Proc. ICDCS 2000.

[5] Apostolopoulos G., *et al*, Design, Implementation and Performance of a Content-Based Switch. Proc. IEEE Infocom 2000.

[6] Chiueh T., *et al*, Integrating Segmentation and Paging Protection for Safe, Transparent and Efficient Software Extensions, Proc. ACM SOSP 1999.

[7] Takahashi M., *et al*, Efficient Kernel Support for Fine-Grained Protection Domains for Mobile Code, Proc. ICDCS 1998.

[8] Kaashoek F., *et al*, Application Performance and Flexibility on Exokernel Systems, Proc. ACM SOSP 1997.

[9] Banga G., *et al*, Resource Containers : A New Facility for Resource Management in Server Clusters, Proc. USENIX 1999.

[10] Peterson L., *et al*, NodeOS Interface Specification.

[11] Peterson L., *et al*, A NodeOS Interface for Active Networks, In IEEE JSAC 2001.

[12] Decasper D.*et al*, Router Plugins : A Software Architecture for Next-Generation Routers, Proc. ACM SIGCOMM 1998.