Efficient, Safe, Application-Specific Message Processing

Dawson R. Engler and Deborah A. Wallach*
{*engler, kerr*}@*lcs.mit.edu*
MIT Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139

Introduction

The complexity and ambition of applications increase commensurately with improvements in processing power and network performance. Unfortunately, although raw CPU and networking hardware speeds have increased, end-to-end network performance has lagged. This disparity is due both to the imbalance between memory subsystems and modern CPUs and to the inefficiency of current networking software. These limits on application performance will continue to increase [3, 4, 6]. This paper addresses the important problem of delivering network performance to applications by separating *message demultiplexing* (determining "who" a message is for), *message vectoring* (determining where to put the message), and *message handling* (running application-specific operations on the message). We introduce dynamic packet filters (DPF), which allow packet filters to be executed safely and efficiently in the kernel, thereby reducing the cost of demultiplexing messages (see Fig 1). In addition, DPF allows application-specific safe message handlers (ASHs) to be associated with filters. These handlers are made safe by controlling their operations and bounding their runtime.

Software Architecture

Message demultiplexing is the process of determining which application a message is for. Rapid demultiplexing is important in order to neither waste processing cycles nor add latency to end-to-end message time [2]. Application-specific demultiplexing is important so that applications can explore new protocols without kernel modification [5]. Previously, networking architects have had to trade flexibility for performance to demultiplex messages efficiently. DPF, our new packet filter system, achieves these twin goals of flexibility and performance by using a carefully-designed declarative packet-filter language that is amenable to aggressive dynamic code generation.

Packet filters are predicates written in a small safe language [5]. Logically, a packet filter examines all incoming network packets; those messages that satisfy its predicate are delivered to its associated application. This approach allows new protocols to be implemented outside of the kernel (and then downloaded into the packet filter driver), greatly increasing flexibility [5].

Currently, packet filters are interpreted, which entails a high computational cost. This cost is sufficiently high that, to the best of our knowledge, all high-performance networking systems except for the x-kernel project [2] avoid them completely. We show that applying dynamic code generation to packet filters can improve their performance to the point that they are equivalent in efficiency to (and in some cases faster than) hand-crafted in-kernel routines, while retaining all of the flexibility of the packet filter model.

The key in our approach to making filters run fast is dynamic code generation. Dynamic code generation is the creation of executable code at *run time*. Its most important property is that it allows the use of runtime information to improve code generation. Dynamic code generation (also known as *dynamic compilation*) is analogous to static compilation except that, since it occurs at runtime, it must be made much more efficient. Dynamic code generation can be used directly with packet filters by "compiling" a packet filter specification directly into actual machine code. While dynamic code generation has been considered a possible mechanism for improving the performance of packet filters since their inception [5], it has been deemed too "complicated" to implement. A contribution of our work is to show that dynamic code generation can be used in a straightforward manner.

Once a packet filter accepts a packet, the message must be delivered to the packet's associated application. There are a variety of actions that can be required in a networking system: message vectoring (e.g., copying a message into its intended slot in a matrix), message manipulations (e.g., checksums), message initiation (e.g., message reply), and control initiation (e.g., handler computation). An application-specific safe message handler (ASH) can perform all of these operations. ASHs are *untrusted* application-level message-handlers that are safely downloaded into the kernel, efficiently compiled into machine code, and executed upon message arrival. ASHs have a variety of mechanisms at their disposal to both direct and integrate data operations into message transfers and to initiate messages. ASHs are made safe by preventing exceptions and excessive runtime and by bounding control-flow and memory operations.

It is important to note the power of the ASH computational model. For example, it allows the vectoring process to be completely dynamic: the application does not have to pre-

*This paper describes joint work with Professor M. Frans Kaashoek. This work was supported in part by the Advanced Research Projects Agency under contract N00014-94-1-0985 and by a NSF National Young Investigator Award.

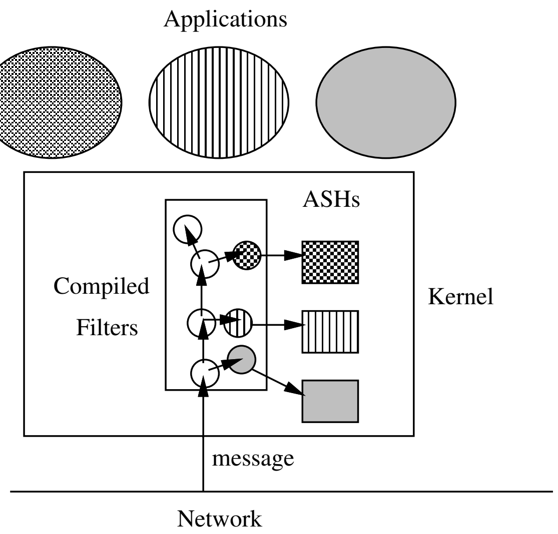

Figure 1: DPF compiles and merges filters into a parse tree. Filters determine how to parse messages. When a filter accepts a message, the corresponding application-specific safe handler (ASH) is invoked. The ASH performs computation and may access application data and send replies on the network.

specify that it is waiting for a particular message, nor does it have to pre-bind locations to put the message. Instead, it can defer these decisions until message reception, when it can then use the user-level data structures, system state, and/or the message itself to determine where to place the message. Capturing the same expressiveness in terms of a static protocol definition is difficult.

To address the problems of message copying and traversal, we designed and implemented a simple extension language, called *vcode*, that allows untrusted applications to safely integrate specialized message vectoring functions into the kernel; we call these handlers application-specific safe handlers (ASH). A novel feature of ASHs is that they are made safe by controlling their operations and bounding their runtime. An ASH is invoked when its associated packet filter accepts, and can exploit application-specific knowledge to determine where to place the message data in memory, thus avoiding memory copies. Our system also provides a general, modular framework for the integration of computation into this data movement, much like [1]. For example, applications can easily integrate encryption, checksumming and data conversion in a single data transfer operation, eliminating duplicate data traversals. In our current system, such integration can reduce message handling costs by a factor of two: this improvement translates directly into a doubling of peak network throughput.

ASHs also provide enough expressive power to implement application-specific *message handling*. Using ASHs, applications can express a broad set of application-specific low-latency operations. The latency of a system determines its performance and scalability; low latency is especially important for tightly-coupled distributed systems. For example, the single most important determinant of parallel program scalability is the latency of communication; special-purpose communication models, such as Active Messages [7], have been developed to achieve this goal. In the context of a client/server system, the faster the server can process messages, the less load it has (and, therefore, the more clients it can support) and the faster response time observed by clients. ASHs can perform a rich set of computations and be used to provide a lightweight, efficient mechanism with which to transfer control. Even recently, low-overhead control transfer had been considered impossible [6].

Conclusions

We have designed, implemented, and tested DPF and a prototype ASH system. We have performed an initial experimental evaluation in user-space, in order to fairly perform compare DPF against other packet filter systems, irrespective of their kernel environment or networking hardware. Results from experiments with standard filters from the literature and several ASHs indicate that substantial performance improvements can be obtained.

DPF filters run 10–40 times faster than PATHFINDER filters, the fastest numbers reported in the literature [2]. DPF's performance is either equivalent to or, when it can exploit runtime information, superior to hand-coded demultiplexors. Results from an implementation of ASH demonstrate a 1.3–2 times improvement in message vectoring and 10–20 improvement in message handling. An in-kernel implementation of DPF and ASH is in progress.

References

[1] M. B. Abbott and L. L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, October 1993.

[2] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkar. PATHFINDER: A pattern-based packet classifier. In *Proceedings of the First Symposium on OSDI*, pages 115–123, 1994.

[3] D.D. Clark and D.L. Tennenhouse. Architectural considerations for a new generation of protocols. *ACM SIGCOMM*, pages 200–208, September 1990.

[4] P. Druschel, M.B. Abbott, M.A. Pagels, and L.L. Peterson. Network subsystem design. *IEEE Network*, 7(4):8–17, July 1993.

[5] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of 11th SOSP*, pages 39–51, Austin, TX, November 1987.

[6] C. A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating data and control transfer in distributed operating systems. In *ASPLOS-VI*, pages 2–11, 1994.

[7] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–267, May 1992.