Scaling Network Services Using Programmable Network Devices

The NEon system offers an integrated approach to architecting, operating, and managing network services. NEon uses policy rules defining the operation of individual network services and produces a unified set of rules that generic packet-processing engines enforce.

Christoph L. Schuba
Jason Goldschmidt
Michael F. Speer
Sun Microsystems Inc.

Mohamed Hefeeda
Simon Fraser University

Society increasingly relies on the Internet for communications, business transactions, information lookup, and entertainment, making it a critical part of our everyday life. The Internet's pervasiveness and its large-scale user base have prompted businesses and institutions to conduct many of their activities electronically and online, creating the need for efficient and reliable management of huge amounts of data.

A successful solution that has been adopted over the past several years is the concentration of critical computing resources in Internet data centers.[1] An IDC is a collection of computing resources typically housed in one physical location: a room, a floor in a building, or an entire building. Computing resources include Web, application, or database servers and network devices such as routers, firewalls, or load balancers. Large enterprises that rely heavily on the Internet and e-commerce applications typically operate their own IDCs, while smaller companies may lease computing resources within an IDC owned and operated by a service provider.

Computing resources in an IDC are typically organized into tiers. For instance, an IDC can dedicate one set of servers for Web access (Tier 1), a second set to run applications initiated by Web requests (Tier 2), and a third set to store data (Tier 3). Each tier is optimized for its own task: A Web server needs high-speed network access and the capacity to handle many concurrent connections, while a database server requires large storage capacity and fast I/O operations.

A tiered architecture allows incremental scaling of IDCs because the operator can independently upgrade each level. For example, if an IDC runs low on storage capacity, only the database server tier needs to be upgraded.

In addition to tiered architectures, IDCs employ other mechanisms to implement improved scalability and cost-effectiveness. One such mechanism is to offload especially expensive operations to special-purpose devices. For example, compute-intensive cryptographic engines often are used to protect client-server communications in financial transactions.

Instead of using expensive server cycles to perform cryptographic operations, highly optimized and less expensive devices can provide that functionality. These special-purpose Tier 0 devices, which precede the first server tier, are placed in the network before the end systems. Furthermore, the services that these devices provide are denoted as Tier 0 network services, or network services for short.

In addition to their use in Internet data centers, these network devices have been deployed in several other environments including at the edge of Internet service provider networks, in storage area networks, and between tiers in server farms.

Published by the IEEE Computer Society
0018-9162/05/$20.00 © 2005 IEEE

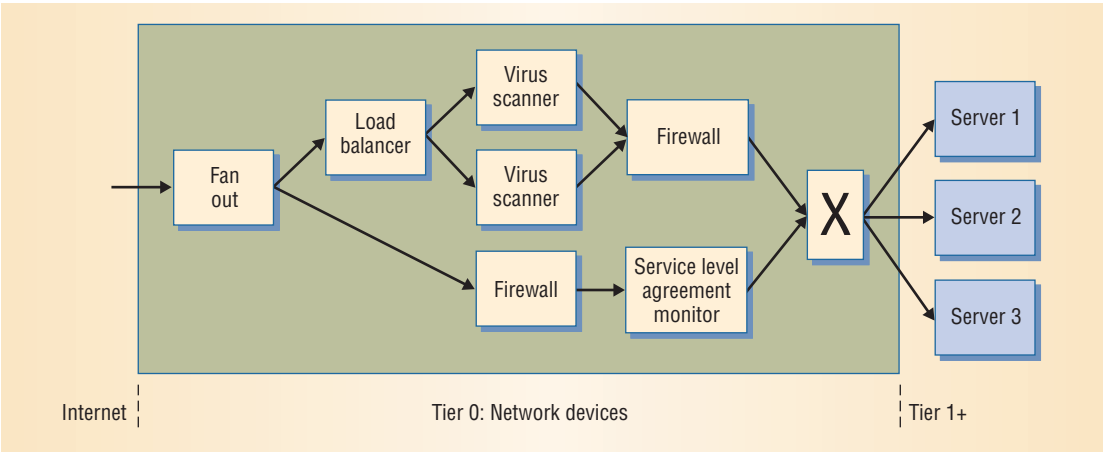

Figure 1. Network services deployment. The discrete approach implements network services as individual devices. As the number of network services increases, the discrete approach suffers from numerous scaling and manageability problems.

Network services are functions that network devices perform on packets before they reach their intended destination. These functions include firewalling, load balancing, intrusion detection, virus scanning, cryptographic acceleration, and service differentiation. Network devices are implemented using highly optimized software and custom hardware,[2] and they can be either standalone appliances or blades plugged into a blade chassis.

Figure 1 shows the discrete approach of deploying multiple network devices that each provide only one network service. However, as the number of network services increases, the discrete approach suffers from numerous scaling and manageability problems, including the following:

- *Fixed network service priorities*. Since network devices are physically cabled in a specific order, dynamically changing the flow processing order is difficult to accomplish. As network and business processing conditions change, dynamic alteration of priorities could provide new and valuable benefits in terms of enterprise security, competitiveness, and productivity.
- *Redundant packet classifications*. Each device performs packet classification and processing, essentially forcing a single flow to serially traverse processing stacks of the individual devices. Redundant processing is not only wasteful but also increases end-to-end latency, which has a negative impact on the user-perceived quality of service.
- *Multiple management consoles*. Each device requires a separate management console with its associated user interface and replicated administrative functions such as software updates and patching.
- *Lack of a feedback loop*. Applications running on servers may need to communicate with network devices that are processing pertinent packet flows. Such a feedback loop could significantly improve the performance of the network devices and the applications. However, it is difficult to establish this feedback loop in the discrete approach because an application would have to communicate with several heterogeneous devices, each with its own interface protocol.

The NEon architecture offers a novel approach for implementing network services. NEon is a paradigm shift away from special-purpose network devices, offering an integrated approach to architecting, operating, and managing network services. NEon employs new flow-handling mechanisms to integrate heterogeneous network services into one system.

NEON: AN INTEGRATED APPROACH

A NEon system accepts as input policy rules that define the operation of various network services and produces a unified set of rules that generic packet-processing engines can enforce. NEon uses rule unification (or crunching) to centralize the control of multiple network services. This centralization offers several advantages over the discrete approach for network services:

- *Flexible and dynamic network service priorities*. NEon merges network services rules together, with each rule possibly having a list of actions. These actions are ordered based on service priorities. Changing service priorities is a matter of changing the order of actions in the action list, which does not require recabling and can be done at runtime.
- *Single packet classification*. Each packet is classified only once before it is dispatched to the appropriate elements to perform the required actions, achieving a significant reduction in the packet processing delay.
- *Centralized management*. All supported network services are managed through one console through which the administrator inputs the rules and any configuration updates.
- *Single feedback point*. NEon servers tune the performance of network devices and applications at a single place. In contrast, in the discrete approach, applications are required to

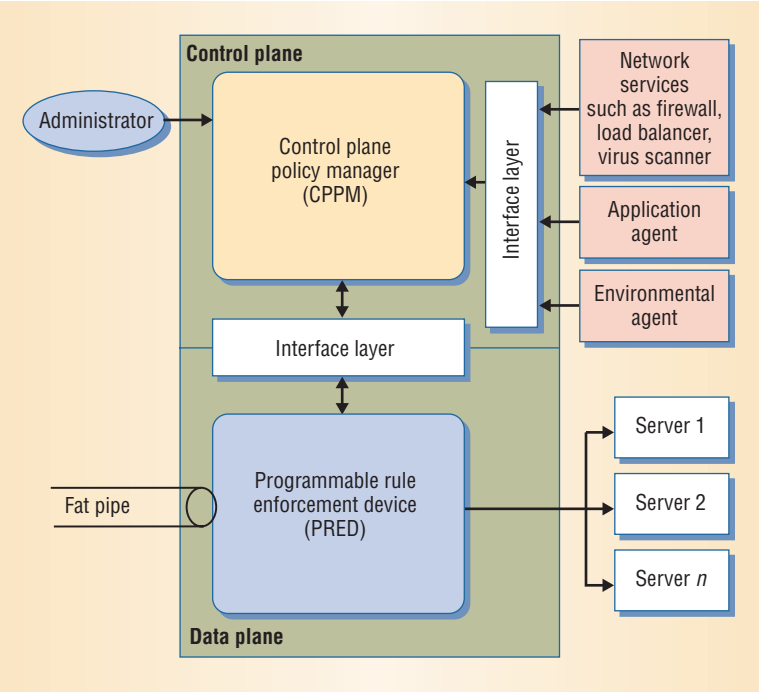

Figure 2. NEon architecture components. Components comprise a control plane and a data plane separated by standards-compliant interface layers.

interact with several devices with different interfaces and communication protocols.

As Figure 2 shows, the NEon architecture's components are divided between two planes: control and data, delineated by standards-compliant interface layers.

The *control plane policy manager* (CPPM) is concerned with network service policy rules and metadata. The CPPM receives policy rules of different network services from system administrators and from management applications representing these network services. It integrates these rules into a unified set of rules that collectively implements the network services.

This unification of policies provides a virtualization of network services while preserving the policy enforcement and mapping of service semantics to physical hardware semantics. Rule unification is the heart of the CPPM, and it is accomplished through a rule-crunching algorithm.

The data-plane component, the *programmable rule enforcement device* (PRED), is a programmable classification and action engine that implements high-speed packet processing.[2] Packet processing is performed according to the rules that the CPPM prepares. Each rule consists of a filter and a list of actions.

The PRED checks data packets against the rule filters, and when a filter matches a packet, it applies the associated list of actions to that packet. PREDs use network processor technology to provide line-speed packet processing and the flexibility of programmable processors. Furthermore, network processors enable a PRED to support multiple network services concurrently.

The NEon architecture components communicate through interface layers that are designed to be compatible with open standards being developed by two industry-wide efforts: the Network Processing Forum (www.npforum.org) and the IETF Forwarding and Control Element Separation (ForCES) working group (www.ietf.org/html.charters/forces-charter.html). To further its efforts to accelerate the adoption of network processor technology in network devices, NPF publications identify the key elements in network devices and define standards for the hardware, software, and benchmarking aspects of building network devices. The ForCES working group defines a standard communication protocol between the control and data planes. The "Network Device Integration" sidebar describes other efforts to integrate the management of multiple network devices.

Standards-based separation of the NEon components offers a simplified management model and allows independent evolution of individual components. One interface layer resides between the CPPM and the PRED. This interface layer requires PREDs from different vendors to support a standard set of APIs that standards-compliant CPPMs will use. The other interface layer transforms input rules from various network services as well as application and environmental agents into the standard rule format that the CPPM supports. The NEon architecture uses application agents and environmental agents to enable dynamic adaptation and performance tuning of network devices.

Application agents form the feedback loop between applications running on the servers, in Tiers 1-3, and network devices. These agents run on servers and trap application-related events and forward them to the NEon CPPM. Typical examples of events that application agents gather are the number of connections opened, current CPU utilization, and memory usage.

Environmental agents provide input to the CPPM to adapt to environmental conditions such as server outages and link failures. Environmental agents allow NEon to dynamically steer the flow of packets to provide dynamic and highly available networked services.

NETWORK SERVICES INTEGRATION

The NEon approach integrates multiple network services and implements them in a single system.

The following are examples of features that different network services can have in common and of how their individual functions can be integrated.

- *Firewall*. A firewall checks individual data packets against preconfigured rule tables and either discards them or allows them to enter the network. An example of a rule in a firewall rule table looks like this: <TCP, 123.123.123.0/24, 0/0, 194.119.230.1/32, 80/16, ALLOW>, where the last field (ALLOW) represents the action that needs to be enforced on packets meeting the conditions specified in the preceding fields. That is, the rule allows the entry of TCP packets that come from the subnetwork 123.123.123.0/24 with any source port number and are destined to the HTTP server (port 80) running on the machine with IP address 194.119.230.1.
- *SLA monitor*. A service level agreement (SLA) monitor gathers statistics on the traffic flowing through a network. Its objective is, for example, to gather usage data for charging customers for the bandwidth used by their traffic. A configuration rule for an SLA monitor could look like this: <UDP, 123.123.123.0/24, 0/0, 0.0.0.0/0, 0/0, ACCT>, which means that statistics are to be collected on all UDP traffic originated from the subnetwork 123.123.123.0/24.
- *Load balancing*. A load balancer can be used in front of a number of servers to spread the load across them based on some criteria. The criteria could be the source address, destination address, protocol, or a combination thereof. A rule in the load balancer table may have the form: <TCP, 0.0.0.0/0, 0/0, 194.119.230.12/32, 23/16, LBGROUP1>, which means forward all TCP packets whose destination IP address is 194.119.230.12 and the destination port is 23 to a server that belongs to load balancing group LBGROUP1.

Rule virtualization

Systems administrators must encode the network service semantics in the form of rules to communicate them to the hardware. To virtualize the rules of different network services, their semantics can be mapped into a single instance of generic packet processing hardware.

Most network services perform their functions by enforcing a set of rules, with each rule taking the form <Filter, ActionList>, where Filter defines a class of packets on which operations that ActionList specifies are to be performed.

A Filter is composed of several fields, each with a Field Name and a Pattern. The Field Name identifies the header field in a protocol layer—for example, the source address in the IP layer. The Pattern is composed of a bit string that determines which packets match this field.

The Pattern also specifies which bits in the string should be considered in matching packets and which bits can be ignored. One way to do that is by using the *mask length notion* (/len), which is commonly used in IP routing tables. For example, a pattern for a 32-bit IP address could be 128.12.30.0/24, which means that the leftmost 24 bits (128.12.30) should match the corresponding bits in packets, while the rightmost 8 bits can be ignored.

Network Device Integration

Previous efforts to integrate the management of multiple network devices includes the Open Platform for Security (www.opsec.com), which provides a standard framework for managing several independent security devices such as firewalls, intrusion detection, and authorization. However, OPSEC integrates the devices only at the management level, whereas NEon integrates both the management and enforcement (hardware) levels.

The rule-crunching algorithm has some similarities to high-speed packet-classification algorithms.[1] Packet classification means finding which rule matches a given packet. If multiple rules match, a conflict occurs and must be resolved so that only one rule applies. Rule conflicts can occur in NEon, but they are handled differently: Rules are merged or modified, and sometimes new rules are created.

An algorithm for detecting conflicts between two *k*-tuple filters creates a new filter when a conflict occurs; therefore, the total number of rules increases exponentially as the number of conflicts increases.[2] A more scalable conflict detection algorithm builds binary tries for each filter field; each level of the trie is one bit of the field.[3] The algorithm computes a bit vector from this trie to aid in conflict detection.

The growth in the total number of rules is a critical issue in PREDs with limited memory. Because our rule-crunching algorithm merges and prioritizes the actions of conflicting rules, it does not incur exponential growth. An efficient data structure for detecting rule conflicts that is based on rectangle geometry works only for two-dimensional classification.[4] However, because classifying based on two fields is not sufficient for many network services, our rule cruncher uses five fields. Finally, an algorithm for removing redundant rules can be used as a preprocessing step for the rule cruncher to eliminate unnecessary service rules.[5]

References

1. D.E. Taylor, *Survey and Taxonomy of Packet Classification Techniques*, tech. report WUCSE200424, Dept. Computer Science and Eng., Washington Univ., 2004.
2. A. Hari, S. Suri, and G. Parulkar, "Detecting and Resolving Packet Filter Conflicts," *Proc. IEEE Infocom 00*, IEEE Press, 2000, pp. 1203-1212.
3. F. Baboescu and G. Varghese, "Fast and Scalable Conflict Detection for Packet Classifiers," *Computer Networks*, Aug. 2003, pp. 717-735.
4. D. Eppstein and S. Muthukrishnan, "Internet Packet Filter Management and Rectangle Geometry," *Proc. 12th ACM SIAM Symp. Discrete Algorithms* (SODA 01), ACM Press, 2001, pp. 827-835.
5. A. Liu and M. Gouda, *Removing Redundancy from Packet Classifiers*, tech. report TR0426, Dept. Computer Sciences, Univ. Texas at Austin, 2004.

The NEon architecture can apply several rules from different services to the same packet.

The simple mask length approach applies only when bits that need to be matched are contiguous. To support more complex patterns such as those that match intermediate bits, the field pattern should specify the full bit mask (not only the length). To match a rule, a packet must match all fields of the rule's filter. To check a data packet against a specific field, the network device extracts the corresponding bits in the packet. If all bits in the field pattern match the extracted bits, a field is matched.

The ActionList is a series of processing actions that the network device performs on a packet. Actions are, for example, dropping a packet, gathering statistical information, controlling timer functions, modifying or marking a packet with metadata—such as inserting new fields or overwriting existing fields—or passing the packet on (doing nothing).

We can abstract the functioning of a network service as follows:

NetworkService	:=	< Rule > +
Rule	:=	< Filter, ActionList >
Filter	:=	< Field > +
Field	:=	< FieldName, Pattern >
ActionList	:=	< Action > +
Action	:=	Drop | Allow | Mark | Overwrite | ...

This abstraction allows combining rules from several network services into a unified set of rules. However, some network services such as stateful services do not readily lend themselves to using the algorithm for performing this unification.

Rule crunching

The rule-crunching algorithm uses service abstraction to take rules from multiple services and generates a consistent rule set that a PRED can apply. Two concepts underlie this algorithm: rule merging and rule enforcement order.

Rule merging. Because the NEon architecture integrates multiple network services in one device, it can apply several rules from different services to the same packet. Rule merging occurs when two or more rules would match the same packet. The rule-crunching algorithm merges rules based on the set of packets each rule influences.

Consider two rules r and r' that belong to two different services. We define S as the set of packets that match rule r. Similarly, S' is the set of packets that match r'. Five relationships are possible between S and S': EQUAL, DISJOINT, SUBSET, SUPERSET, and INTERSECT. Rule merging creates the following relationships:

- $S = S'$ (EQUAL). This relationship indicates that r has the same filter as r', but they may have different actions. The result of merging is one rule denoted by cr. Rule cr will have the common filter and the concatenation (denoted by the pipe symbol) of the actions of r and r'. That is, $cr.filter = r.filter = r'.filter$, and $cr.ActionList = r.ActionList \mid r'.ActionList$.
- $S \cap S' = \phi$ (DISJOINT). This relationship means that the two rules r and r' are to be applied on different, nonintersecting sets of packets. In this case, no merging will happen, and the two rules r and r' are used in their original format.
- $S \subset S'$ (SUBSET). In this case, packets in the set S should be subjected to actions of rule r as well as rule r'. Moreover, packets in the set $S' - S$ should be subjected only to the action list of r'. Merging creates two rules cr_1 and cr_2, where (1) $cr_1.filter = r.filter$ and $cr_1.ActionList = r.ActionList \mid r'.ActionList$; and (2) $cr_2 = r'$. Note that the device stores cr_1 and cr_2 in a way that ensures that packets are checked against cr_1 before cr_2. Therefore, cr_2 will be applied only to packets that do not match cr_1 but match cr_2—that is, packets belonging to the set $S' - S$.
- $S \supset S'$ (SUPERSET). This case is equivalent to $S' \subset S$ (SUBSET) and handled accordingly.
- $S \cap S' \neq \phi$ (INTERSECT). Merging in this case results in three rules $cr1$, $cr2$, and $cr3$, where (1) $cr_1.filter = r.filter \cap r'.filter$ and $cr_1.ActionList = r.ActionList \mid r'.ActionList$; (2) $cr_2 = r$; and (3) $cr_3 = r'$. Again, cr_1 should be checked before both cr_2 and cr_3.

For each relationship, the algorithm creates equivalent crunched rules whose filters match the same set of packets as the original service rules. Moreover, the crunched rules perform the same actions as the original rules, and any existing ambiguity among them has been removed.

Rule enforcement order. A programmable rule enforcement device checks every packet flowing through it against the set of crunched rules stored in its table. The order of checking rules against packets is critical to the PRED's correct operation because a packet can match more than one crunched rule when only one rule should be applied. For example, a packet can match two rules, one of which contains more specific filters

because it matches source IP address and port number rather than just the source IP address. Clearly, the more specific rule—the first one—should be applied.

The algorithm uses two approaches to determine the order of checking rules against packets: *ordered precedence* and *longest prefix matching*. Ordered precedence matching places rules with more specific filters earlier in the rule table. Rules are considered one at a time in the order specified: The first matching rule fires, and its action list is executed. In longest prefix matching, the algorithm applies the rule that shares the longest prefix with the corresponding fields in the packet.

Typically, the algorithm stores rules in a data structure that facilitates longest prefix matching, such as binary tries. Longest prefix matching assumes matching contiguous bits.

Rule-crunching algorithm

The input to the rule-crunching algorithm is the *service rule database* (srdb), a list of rules of individual network services. The administrator assigns a unique priority to each network service. All rules of the same service get the same priority. The srdb is ordered based on this priority, which ensures that all rules of the same network service come after each other. We illustrate this algorithm using high-level pseudocode:

```
1. crdb ← r1; /* r1 is the first rule
                 in srdb. */
2. foreach r ∈ srdb − {r1} do
3.   foreach cr ∈ crdb do
4.     rel ← DetermineRel(r, cr);
5.     if rel == DISJOINT
6.       add r to crdb;
7.     else
8.       MergeRules(r, cr, rel, crdb);
9. return crdb;
```

The algorithm's output is a unified set of rules: the *crunched rule database* (crdb). The algorithm subsequently removes rules from the srdb and adds them to the crdb until there are no more rules to move. A rule *r* in srdb is compared with every rule *cr* in crdb for possible merging.

The *DetermineRel()* function invoked in line four determines the relationship, *rel,* between the two rules by comparing their corresponding field filters. If the two rules cannot be applied on any packet simultaneously—that is, their packet sets are DISJOINT—no rule merging is performed, and *r* is added to crdb. Otherwise, the *MergeRules()* function is invoked to merge the two rules based on their relationship, *rel*. For example, if *rel* = EQUAL, function *MergeRules()* adds the *r* action list to the *cr* action list and adjusts the priority of the modified *cr*. Note that rule *r* itself is not added to crdb, which reduces the total number of rules in crdb.

Algorithm analysis. As the discussion on rule merging indicates, the DISJOINT, SUBSET, and SUPERSET relationships do not add new rules to the crdb; they merely move the service rules from the srdb to the crdb and can modify the rule's filter or action lists.

The rule-crunching algorithm stores rules in a data structure that facilitates longest prefix or ordered precedence matching.

If the EQUAL relationship occurs between two rules, we add only one of them to the crdb—that is, the number of rules is reduced by one. If the INTERSECT relationship occurs between two rules, we add three rules to the crdb, which increases the total number of rules by one.

Our worst-case analysis assumes that no EQUAL relationships occur. Therefore, the crdb's final size is equal to the number of the original service rules (n) in addition to the maximum number of rules created from all possible INTERSECT relationships. To count the maximum number of INTERSECT relationships, the main factor in determining the relationship between two rules is their IP source and destination addresses because these fields can use address range wild-carding (for example, a.b.c.d/16 or a.b.c.d/24). Other fields will either match or will not.

For two rules to intersect, their IP source (or destination) addresses must share a common prefix, while their other fields can differ. Since IP addresses have a fixed number of bits k (k = 32 for IPv4), the common prefix can range from 1 bit to a maximum of k bits. In addition, crdb prefixes are unique because if two rules have the same prefix, they must have been merged in a previous iteration.

When the rule crunching algorithm compares one rule from the srdb against all entries in the crdb, there can be at most $(k + k) \times 2 \times c = O(1)$ INTERSECT relationships, where c is the number of fields in the filter other than the IP addresses (c = 3 in the 5-tuple rules). Because c fields may or may not match in the INTERSECT relationship, we must include the factor $2 \times c$. The factor 2 comes from the fact that there are two IP addresses: source and destination. Therefore, each iteration of the algorithm's outer loop (line 2) will add a maximum of $O(1)$ rules to the crdb, which results in a space complexity of $n \times O(1) + n = O(n)$.

The same arguments apply for determining the algorithm's time complexity. This is because the

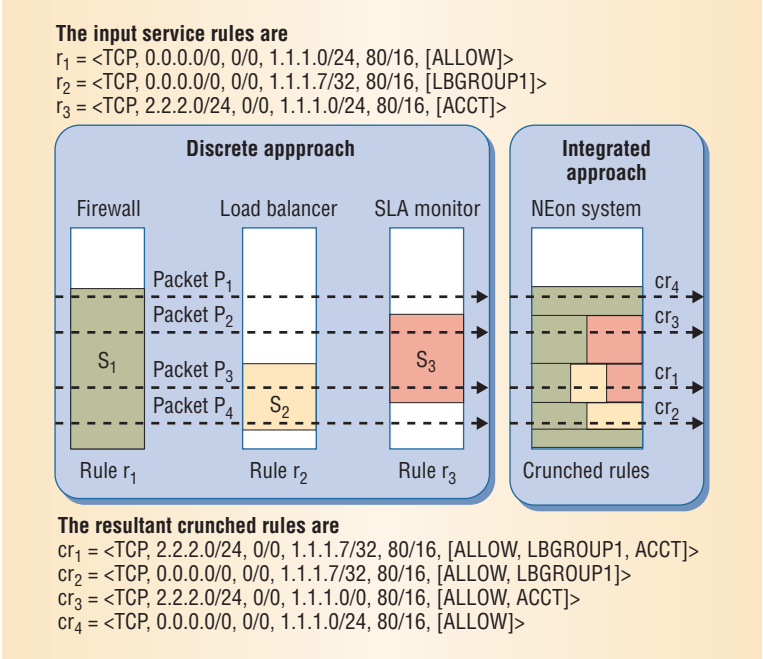

Figure 3. Rule crunching in NEon. The algorithm merges the input service rules for firewalling, load balancing, and service level agreement monitoring into a set of four crunched rules allowing different actions on incoming packets.

inner loop is invoked at most n times, the algorithm's overall time complexity is $O(n^2)$.

The rule-crunching algorithm is a part of the CPPM, which operates in the control plane, not in the performance-critical, high-speed data plane. Moreover, the CPPM does not necessarily run on the network device's hardware. The CPPM can run on a monitoring or management server that controls several network devices.

The management server provides a single point to feed and update rules for various network services. It also performs the rule crunching once for all attached network devices and then pushes the crunched rules to each network device. Furthermore, because it has the crunched rules for all devices, the management server can perform some optimizations such as removing redundant rules.[3]

Example. Figure 3 illustrates the crunching of three rules r_1, r_2, and r_3 belonging to three different services: firewalling, load balancing, and SLA monitoring, respectively. The color of each area in the figure represents the set of packets that matches a rule whose action is represented by the same color. For instance, the green area represents the set of packets on which the firewall performs the action ALLOW. The word ALLOW in the action list of the rules is also colored in green. Let S_1, S_2, and S_3 represent packet sets that match rules r_1, r_2, and r_3, respectively. The figure shows various relationships between the packet sets. For example, S_1 is a SUPERSET of S_2 and S_3, while the relationship between S_2 and S_3 is INTERSECT.

The rules r_1, r_2, and r_3 are initially stored in the srdb, and the crdb is empty. Line 1 of the algorithm moves r_1 to the crdb. Then, it merges r_2 with r_1. Since $S_2 \subset S_1$, r_2 is added to the crdb after concatenating the action list of r_1 to its action list, r_2 will have <ALLOW, LBGROUP1> as its action list and it will be inserted before r_1 in the crdb. Then, r_3 will be merged with the modified r_2. This is an INTERSECT relationship.

The merging produces a new rule with the filter <TCP, 2.2.2.0/24,0/0, 1.1.1.7/32, 80/16>, which is the intersection of the r_2 and r_3 filters. The new rule has the action list <ALLOW, LBGROUP1, ACCT>. In the final step, r_3 is merged with r_1, which modifies r_3 by concatenating the action list of r_1 to r_3's before adding r_3 to the crdb.

The resultant crunched rules are shown in the bottom part of Figure 3. Figure 3 also demonstrates the flow of sample packets P_1 to P_4 through the discrete network devices and the NEon system. The NEon system performs exactly the same actions on each packet that the discrete network devices perform. For example, packet P_3 matches the three rules r_1, r_2, and r_3 of the discrete devices and at the same time matches cr_1, which has the same actions as r_1, r_2, and r_3.

PROTOTYPE SYSTEM

To validate the NEon concept, we developed a complete prototype system. The prototype has been tested with commercial hardware devices using synthetic data traffic as well as configuration files from operational network devices.

CPPM code

The CPPM code is implemented in Java and has two main parts: a *service listener* and a *rule cruncher*.

The service listener receives policy rules of individual network services from the administrator and from the software agents representing network services. It is implemented as a pool of threads, one for each active network service. The service listener stores the received rules and rule updates in the srdb. Rules in the srdb are ordered based on their defined network service priorities.

The rule cruncher applies the rule-crunching algorithm to the srdb to create the crunched rule database crdb. The rule cruncher can be invoked periodically (every few minutes or seconds), upon a rule update (insert, delete, or modify a rule), or explicitly by the administrator. The automatic and periodic invocation of the rule cruncher allows fast propagation of updated configurations to the enforcement devices.

PRED

Because there are currently no commercially available generic PREDs that can apply rules from multiple network services, for our prototype we

modified two different hardware products to serve as PREDs.

We have tested our prototype on PolicyEdge, a network processor chip emulator from FastChip, and on the Sun Fire Content Load Balancer Blade (B10n) from Sun Microsystems. Successfully modifying these devices to run the NEon prototype demonstrates that the integrated approach for network services is both feasible and viable.

The B10n is a networking product that provides content load balancing for blade servers and horizontally scaled systems. The B10n operates at the data center edge, uses user-specified rules to classify client-side inbound traffic at wire speed, and applies load balancing actions on the data traffic.

The B10n was designed to provide only content load balancing—that is, only one action is associated with each rule. We have augmented the firmware and the data structures to support multiple actions for each rule.

The B10n's rule matching technique is based on two fields: IP source address and source port. In NEon, we use 5-tuple rules to represent a wider range of network services. We have changed the rule structure to support five layer-4 fields: source IP, destination IP, source port, destination port, and protocol. In addition, any field can be either fully specified or wild-carded.

Results

Several NEon system parameters were tested including the rule-crunching algorithm's runtime, the relationship between the number of crunched rules versus the number of raw service rules, and the number and type of merging relationships occurring among rules.

To perform the experiments, we used configuration files from deployed network services, handcrafted scenarios, and generated data. The configuration files had a small number of unique service rules—63 on average, which is typical for many Internet data centers. The crunching algorithm took about 6 milliseconds on a Sun Fire Ultrasparc 240 server-class machine and produced 148 crunched rules on average.

We also verified that the INTERSECT relationship, which increases the size of the crdb, does not occur frequently between rules: only 9.9 percent of the merging relationships were INTERSECT. More than 87 percent of the relationships were DISJOINT, with the small remaining percentage distributed among EQUAL, SUBSET, and SUPERSET relationships.

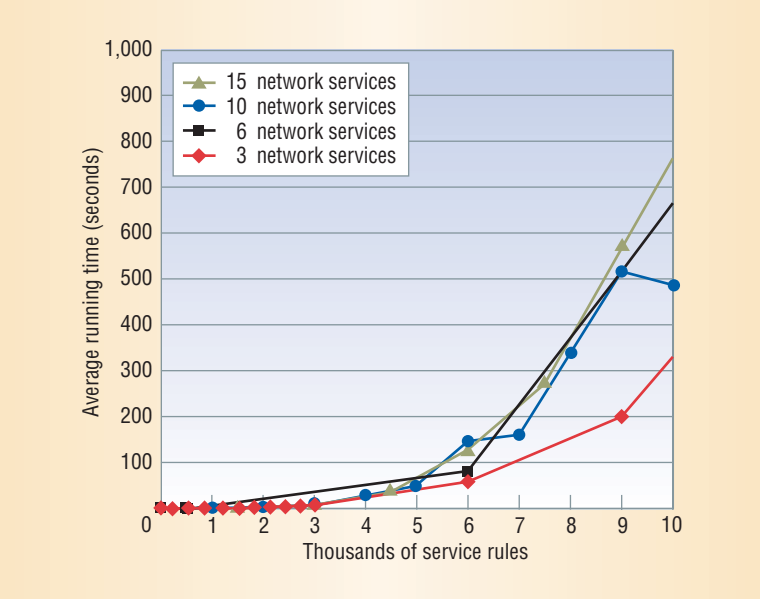

Figure 4.Rule cruncher runtime. The rule cruncher algorithm terminates in less than one minute for up to approximately 4,000 rules.

To test the scalability of the approach, we simulated various combinations of network services with a large number of rules. The rule filters were generated randomly within the appropriate ranges. For example, the transport protocol field was chosen randomly from either TCP or UDP. Random generation of rules stresses the rule-crunching algorithm because it produces more INTERSECT relationships than in typical configuration files; therefore, it pushes the running time and the size of the crunched rule database toward their worst cases.

Figure 4 shows the rule cruncher's average running time as the number of service rules increases from 0 to 10,000. The total number of service rules was divided among the simulated number of network services. For example, if we simulate 8,000 rules and 10 network services, each network service will have, on average, 800 rules. The algorithm terminates in less than one minute for up to approximately 4,000 rules. For larger numbers of rules, the running time is still on the order of minutes. This is acceptable given that the rule cruncher is not invoked frequently: It is needed initially and when rules or rule sets are modified.

The NEon integrated approach offers numerous advantages over the current practice of implementing network services as discrete devices, including

- flexibility of dynamically changing service priorities;
- single packet classification, which leads to shorter end-to-end delay;
- centralized management; and
- a single point for applications to establish a feedback loop with network devices.

Testing a NEon prototype architecture with commercial hardware devices and data collected from operational network devices demonstrates that this system offers a feasible and viable approach for implementing sophisticated network services.

This work can be extended in several directions. The rule-crunching algorithm reconstructs the entire crunched rule database upon the modification of any service rule. Currently, this is not a major concern because rule updates are infrequent and occur on a time scale of hours or days. However, in the future, more dynamic environments in which service rules can be updated on a shorter time scale will pose a challenge for the rule cruncher. One possible solution is to design a differential rule cruncher, which performs the minimum amount of work to adjust the crunched-rule database to reflect changes in service rules.

Another future direction that would broaden the scope and applicability of the NEon approach is to extend the rule unification mechanism to accommodate more network services such as stateful services, services that access data beyond packet headers (application data), and services such as the network address translator that can change a packet's identity.

A complete test bed is needed to thoroughly compare NEon with the discrete approach. Such a test bed would contain several network devices connected in the traditional discrete way and an equivalent NEon system. It would be interesting to measure performance parameters such as average packet processing time, the time needed to reconfigure the network devices, and how quickly the system can propagate rule updates to the rule-enforcing hardware. ■

Acknowledgments

We appreciate the insightful discussions, suggestions, and support from Robert Bressler, Danny Cohen, Cindy Dones, Kevin Kalajan, Lisa Pavey, and Raphael Rom. We also thank Ahsan Habib, Nan Jin, Sumantra Kundu, Santashil PalChaudhuri, George Porter, Ioan Raicu, and Aaron Striegel for their contributions to the NEon project during their summer internships at Sun Microsystems, Inc.

References

1. M. Arregoces and M. Portolani, *Data Center Fundamentals*, Cisco Press, 2003.
2. D. Comer, *Network Systems Design Using Network Processors*, Prentice Hall, 2004.
3. A. Liu and M. Gouda, *Removing Redundancy from Packet Classifiers*, tech. report TR0426, Dept. Computer Sciences, Univ. Texas at Austin, 2004.

Christoph L. Schuba *is a senior staff engineer in the Security Program Office at Sun Microsystems Inc. His research interests include network and computer system security, high-speed networking, and distributed systems. Schuba received a PhD in computer science from Purdue University. He is a member of the Internet Society and Usenix. Contact him at christoph.schuba@sun.com.*

Mohamed Hefeeda *is an assistant professor in the School of Computing Science at Simon Fraser University, Canada. His research interests include computer networks, multimedia networking, peer-to-peer systems, and network security. Hefeeda received a PhD in computer science from Purdue University. He is a member of the IEEE and the ACM Special Interest Group on Data Communications. Contact him at mhefeeda@cs.sfu.ca.*

Jason Goldschmidt *is a technical staff engineer, Network Systems Group, Sun Microsystems Inc. His research interests include network processing, protocols, and scalable architectures. Goldschmidt received a BS in computer science and engineering from Bucknell University. Contact him at jason.goldschmidt@sun.com.*

Michael F. Speer *is a senior staff engineer, Netra Systems and Networking, Sun Microsystems Inc. His research interests include scalable network systems and services architectures for network deployments. He received a BS in computer engineering from the University of California, San Diego. He is a member of the IEEE Computer Society, the ACM, and Usenix. Contact him at michael.speer@sun.com.*