WHITE PAPER

NETWORK OPERATING SYSTEM EVOLUTION

Juniper Networks JUNOS Software: Architectural Choices at the Forefront of Networking

Table of Contents

Table of Figures

Executive Summary

This paper discusses the requirements and challenges inherent in the design of a carrier-class network operating system (OS). Key facets of Juniper Networks® JUNOS® Software, Juniper's network operating system, are used to illustrate the evolution of OS design and underscore the relationship between functionality and architectural decisions.

The challenge of designing a contemporary network operating system is examined from different angles, including flexibility, ability to power a wide range of platforms, nonstop operation, and parallelism. Architectural challenges, trade-offs and opportunities are identified, as well as some of the best practices in building state-of-the-art network operating systems.

Introduction

Modern network devices are complex entities composed of both silicon and software. Thus, designing an efficient hardware platform is not, by itself, sufficient to achieve an effective, cost-efficient and operationally tenable product. The control plane plays a critical role in the development of features and in ensuring device usability.

Although progress from the development of faster CPU boards and forwarding planes is visible, structural changes made in software are usually hidden, and while vendor collateral often offers a list of features in a carrier-class package, operational experiences may vary considerably.

Products that have been through several generations of software releases provide the best examples of the difference made by the choice of OS. It is still not uncommon to find routers or switches that started life under older, monolithic software and later migrated to more contemporary designs. The positive effect on stability and operational efficiency is easy to notice and appreciate.

However, migration from one network operating system to another can pose challenges from nonoverlapping feature sets, noncontiguous operational experiences and inconsistent software quality. These potential challenges make it is very desirable to build a control plane that can power the hardware products and features supported in both current and future markets.

Developing a flexible, long-lasting and high-quality network OS provides a foundation that can gracefully evolve to support new needs in its height for up and down scaling, width for adoption across many platforms, and depth for rich integration of new features and functions. It takes time, significant investment and in-depth expertise.

Most of the engineers writing the early releases of JUNOS Software came from other companies where they had previously built network software. They had firsthand knowledge of what worked well, and what could be improved. These engineers found new ways to solve the limitations that they'd experienced in building the older operating systems. Resulting innovations in JUNOS are significant and rooted in its earliest design stages. Still, to ensure that our products anticipate and fulfill the next generation of market requirements, JUNOS is periodically reevaluated to determine whether any changes are needed to ensure that it continues to provide the reliability, performance and resilience for which it is known.

Origin and Evolution of Network Operating Systems

Contemporary network operating systems are mostly advanced and specialized branches of POSIX-compliant software platforms and are rarely developed from scratch. The main reason for this situation is the high cost of developing a world-class operating system all the way from concept to finished product. By adopting a general-purpose OS architecture, network vendors can focus on routing-specific code, decrease time to market, and benefit from years of technology and research that went into the design of the original (donor) products.

For example, consider Table 1, which lists some operating systems for routers and their respective origins (the Generation column is explained in the following sections).

Table 1: Router Operating System Origins

VENDOR	ROUTER OS	DONOR OS	DONOR OS OWNER	GENERATION	DONOR OS WEB SITE
Juniper Networks	JUNOS Software	FreeBSD	Internet community	2, 3	www.freebsd.org
Cisco	IOS-XR	QNX	QNX Software Systems	2	www.qnx.com
Cisco	IOS	n/a	Proprietary	1	n/a
Cisco	IOS-XE	IOS, Linux	Hybrid (ASR)	1.5	n/a
Cisco	Modular IOS	IOS, QNX	Hybrid (6500)	1.5	n/a
Alcatel-Lucent	SR OS	VxWorks	WindRiver	2	www.windriver.com
Redback	SEOS	NetBSD	Internet community	2	www.netbsd.org
Force10	FTOS	NetBSD	Internet community	2	www.netbsd.org
Extreme	ExtremeWARE	Linux	Internet community	2	www.xoslinux.org

Generally speaking, network operating systems in routers can be traced to three generations of development, each with distinctively different architectural and design goals.

First-Generation OS: Monolithic Architecture

Typically, first-generation network operating systems for routers and switches were proprietary images running in a flat memory space, often directly from flash memory or ROM. While supporting multiple processes for protocols, packet handling and management, they operated using a cooperative, multitasking model in which each process would run to completion or until it voluntarily relinquished the CPU.

All first-generation network operating systems shared one trait: They eliminated the risks of running full-size commercial operating systems on embedded hardware. Memory management, protection and context switching were either rudimentary or nonexistent, with the primary goals being a small footprint and speed of operation. Nevertheless, first-generation network operating systems made networking commercially viable and were deployed on a wide range of products. The downside was that these systems were plagued with a host of problems associated with resource management and fault isolation; a single runaway process could easily consume the processor or cause the entire system to fail. Such failures were not uncommon in the data networks controlled by older software and could be triggered by software errors, rogue traffic and operator errors.

Legacy platforms of the first generation are still seen in networks worldwide, although they are gradually being pushed into the lowest end of the telecom product lines.

Second-Generation OS: Control Plane Modularity

The mid-1990s were marked by a significant increase in the use of data networks worldwide, which quickly challenged the capacity of existing networks and routers. By this time, it had become evident that embedded platforms could run full-size commercial operating systems, at least on high-end hardware, but with one catch: They could not sustain packet forwarding with satisfactory data rates. A breakthrough solution was needed. It came in the concept of a hard separation between the control and forwarding plane—an approach that became widely accepted after the success of the industry's first application-specific integrated circuit (ASIC)-driven routing platform, the Juniper Networks M40. Forwarding packets entirely in silicon was proven to be viable, clearing the path for next-generation network operating systems, led by Juniper with its JUNOS Software.

Today, the original M40 routers are mostly retired, but their legacy lives in many similar designs, and their blueprints are widely recognized in the industry as the second-generation reference architecture.

Second-generation network operating systems are free from packet switching and thus are focused on control plane functions. Unlike its first-generation counterparts, a second-generation OS can fully use the potential of multitasking, multithreading, memory management and context manipulation, all making systemwide failures less common. Most core and edge routers installed in the past few years are running second-generation operating systems, and it is these systems that are currently responsible for moving the bulk of traffic on the Internet and in corporate networks.

However, the lack of a software data plane in second-generation operating systems prevents them from powering low-end devices without a separate (hardware) forwarding plane. Also, some customers cannot migrate from their older software easily because of compatibility issues and legacy features still in use.

These restrictions led to the rise of transitional (generation 1.5) OS designs, in which a first-generation monolithic image would run as a process on top of the second-generation scheduler and kernel, thus bridging legacy features with newer software concepts. The idea behind "generation 1.5" was to introduce some headroom and gradually move the functionality into the new code, while retaining feature parity with the original code base. Although interesting engineering exercises, such designs were not as feature-rich as their predecessors, nor as effective as their successors, making them of questionable value in the long term.

Third-Generation OS: Flexibility, Scalability and Continuous Operation

Although second-generation designs were very successful, the past 10 years have brought new challenges. Increased competition led to the need to lower operating expenses and a coherent case for network software flexible enough to be redeployed in network devices across the larger part of the end-to-end packet path. From multiple-terabit routers to Layer 2 switches and security appliances, the "best-in-class" catchphrase can no longer justify a splintered operational experience—true "network" operating systems are clearly needed. Such systems must also achieve continuous operation, so that software failures in the routing code, as well as system upgrades, do not affect the state of the network. Meeting this challenge requires availability and convergence characteristics that go far beyond the hardware redundancy available in second-generation routers.

Another key goal of third-generation operating systems is the capability to run with zero downtime (planned and unplanned). Drawing on the lesson learned from previous designs regarding the difficulty of moving from one OS to another, third-generation operating systems also should make the migration path completely transparent to customers. They must offer an evolutionary, rather than revolutionary, upgrade experience typical to the retirement process of legacy software designs.

Basic OS Design Considerations

Choosing the right foundation (prototype) for an operating system is very important, as it has significant implications for the overall software design process and final product quality and serviceability. This importance is why OEM vendors sometimes migrate from one prototype platform to another midway through the development process, seeking a better fit. Generally, the most common transitions are from a proprietary to a commercial code base and from a commercial code base to an open-source software foundation.

Regardless of the initial choice, as networking vendors develop their own code, they get further and further away from the original port, not only in protocol-specific applications but also in the system area. Extensions such as control plane redundancy, in-service software upgrades and multichassis operation require significant changes on all levels of the original design. However, it is highly desirable to continue borrowing content from the donor OS in areas that are not normally the primary focus of networking vendors, such as improvements in memory management, scheduling, multicore and symmetric multiprocessing (SMP) support, and host hardware drivers. With proper engineering discipline in place, the more active and peer-reviewed the donor OS is, the more quickly related network products can benefit from new code and technology.

This relationship generally explains another market trend evident in Table 1—only two out of five network operating systems that emerged in the routing markets over the past 10 years used a commercial OS as a foundation.

Juniper's main operating system, JUNOS Software, is an excellent illustration of this industry trend. The basis of the JUNOS kernel comes from the FreeBSD UNIX OS, an open-source software system. The JUNOS kernel and infrastructure have since been heavily modified to accommodate advanced and unique features such as state replication, nonstop active routing and in-service software upgrades, all of which do not exist in the donor operating system. Nevertheless, the JUNOS Software tree can still be synchronized with the FreeBSD repository to pick the latest in system code, device drivers and development tool chains, which allows Juniper Networks engineers to concentrate on network-specific development.

Commercial Versus Open-Source Donor OS

The advantage of a more active and popular donor OS is not limited to just minor improvements—the cutting edge of technology creates new dimensions of product flexibility and usability. Not being locked into a single-vendor framework and roadmap enables greater control of product evolution as well as the potential to gain from progress made by independent developers.

This benefit is evident in JUNOS Software, which became a first commercial product to offer hard resource separation of the control plane and a real-time software data plane. Juniper-specific extension of the original BSD system architecture relies on multicore CPUs and makes JUNOS the only operating system that powers both low-end software-only systems and high-end multiple-terabit hardware platforms with images built from the same code tree. This technology and experience could not be created without support from the entire Internet-driven community. The powerful collaboration between leading individuals, universities and commercial organizations helps JUNOS stay on the very edge of operating system development. Further, this collaboration works both ways: Juniper donates to the free software movement, one example being the Juniper Networks FreeBSD/MIPS port.

Functional Separation and Process Scheduling

Multiprocessing, functional separation and scheduling are fundamental for almost any software design, including network software. Because CPU and memory are shared resources, all running threads and processes have to access them in a serial and controlled fashion. Many design choices are available to achieve this goal, but the two most important are the memory model and the scheduling discipline. The next section briefly explains the intricate relation between memory, CPU cycles, system performance and stability.

Memory Model

The memory model defines whether processes (threads) run in a common memory space. If they do, the overhead for switching the threads is minimal, and the code in different threads can share data via direct memory pointers. The downside is that a runaway process can cause damage in memory that does not belong to it.

In a more complex memory model, threads can run in their own virtual machines, and the operating system switches the context every time the next thread needs to run. Because of this context switching, direct communication between threads is no longer possible and requires special interprocess communication (IPC) structures such as pipes, files and shared memory pools.

Scheduling Discipline

Scheduling choices are primarily between cooperative and preemptive models, which define whether thread switching happens voluntarily (Figure 1). A cooperative multitasking model allows the thread to run to completion, and a preemptive design ensures that every thread gets access to the CPU regardless of the state of other threads.

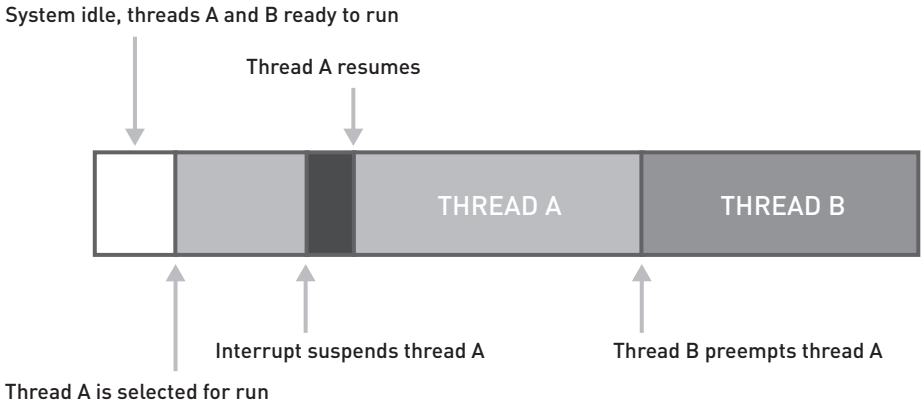


Figure 1: Typical preemptive scheduling sequence

Virtual Memory/Preemptive Scheduling Programming Model

Virtual memory with preemptive scheduling is a great design choice for properly constructed functional blocks, where interaction between different modules is limited and well defined. This technique is one of the main benefits of the second-generation OS designs and underpins the stability and robustness of contemporary network operating systems. However, it has its own drawbacks.

Notwithstanding the overhead associated with context switching, consider the interaction between two threads (Figure 2), A and B, both relying on the common resource R. Because threads do not detect their relative scheduling in the preemptive model, they can actually access R in a different order and with varying intensity. For example, R can be accessed by A, then B, then A, then A and then B again. If thread B modifies resource R, thread A may get different results at different times—and without any predictability. For instance, if R is an interior gateway protocol (IGP) next

hop, B is an IGP process, and A is a BGP process, then BGP route installation may fail because the underlying next hop was modified midway through routing table modification. This scenario would never happen in the cooperative multitasking model, because the IGP process would release the CPU only after it finishes the next-hop maintenance.

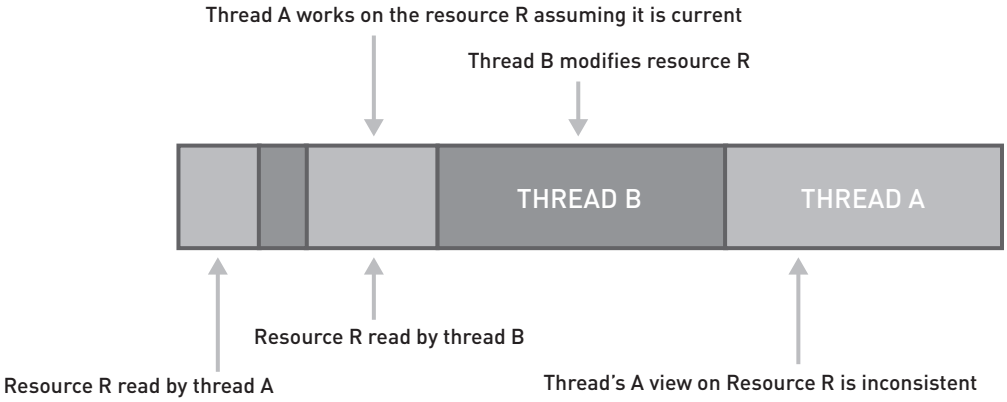


Figure 2: Resource management conflicts in preemptive scheduling

This problem is well researched and understood within software design theory, and special solutions such as resource locks and synchronization primitives are easily available in nearly every operating system. However, the effectiveness of IPC depends greatly on the number of interactions between different processes. As the number of interacting processes increases, so does the number of IPC operations. In a carefully designed system, the number of IPC operations is proportional to the number of processes (N). In a system with extensive IPC activity, this number can be proportional to N^2.

Exponential growth of an IPC map is a negative trend not only because of the associated overhead, but because of the increasing number of unexpected process interactions that may escape the attention of software engineers.

In practice, overgrown IPC maps result in systemwide "IPC meltdowns" when major events trigger intensive interactions. For instance, pulling a line card would normally affect interface management, IGP, exterior gateway protocol and traffic engineering processes, among others. When interprocess interactions are not well contained, this event may result in locks and tight loops, with multiple threads waiting on each other and vital system operations such as routing table maintenance and IGP computations temporarily suspended. Such defects are signatures of improper modularization, where similar or heavily interacting functional parts do not run as one process or one thread.

The right question to ask is, "Can a system be too modular?" The conventional wisdom says, "Yes."

Excessive modularity can bring long-term problems, with code complexity, mutual locks and unnecessary process interdependencies. Although none of these may be severe enough to halt development, feature velocity and scaling parameters can be affected. Complex process interactions make programming for such a network OS an increasingly difficult task.

On the other hand, the cooperative multitasking, shared memory paradigm becomes clearly suboptimal if unrelated processes are influencing each other via the shared memory pool and collective restartability. A classic problem of first-generation operating systems was systemwide failure due to a minor bug in a nonvital process such as SNMP or network statistics. Should such an error occur in a protected and independently restartable section of system code, the defect could easily be contained within its respective code section.

This brings us to an important conclusion.

No fixed principle in software design fits all possible situations.

Ideally, code design should follow the most efficient paradigm and apply different strategies in different parts of the network OS to achieve the best marriage of architecture and function. This approach is evident in JUNOS Software, where functional separation is maintained so that cooperative multitasking and preemptive scheduling can both be used effectively, depending on the degree of IPC containment between functional modules.

Generic Kernel Design

Kernels normally do not provide any immediately perceived or revenue-generating functionality. Instead, they perform housekeeping activities such as memory allocation and hardware management and other system-level tasks. Kernel threads are likely the most often run tasks in the entire system. Consequently, they have to be robust and run with minimal impact on other processes.

In the past, kernel architecture largely defined the operating structure of the entire system with respect to memory management and process scheduling. Hence, kernels were considered important differentiators among competing designs.

Historically, the disputes between the proponents and opponents of lightweight versus complex kernel architectures came to a practical end when most operating systems became functionally decoupled from their respective kernels. Once software distributions became available with alternate kernel configurations, researchers and commercial developers were free to experiment with different designs.

For example, the original Carnegie-Mellon **Mach microkernel** was originally intended to be a drop-in replacement for the kernel in BSD UNIX and was later used in various operating systems, including mkLinux and **GNU FSF** projects. Similarly, some software projects that started life as purely microkernel-based systems later adopted portions of monolithic designs.

Over time, the radical approach of having a small kernel and moving system functions into the user-space processes did not prevail. A key reason for this was the overhead associated with extra context switches between frequently executed system tasks running in separate memory spaces. Furthermore, the benefits associated with restartability of essentially all system processes proved to be of limited value, especially in embedded systems. With the system code being very well tested and limited to scheduling, memory management and a handful of device drivers, the potential errors in kernel subsystems are more likely to be related to hardware failures than to software bugs. This means, for example, that simply restarting a faulty disk driver is unlikely to help the routing engine stay up and running, as the problem with storage is likely related to a hardware failure (for example, uncorrectable fault in a mass storage device or system memory bank).

Another interesting point is that although both monolithic and lightweight kernels were widely studied by almost all operating system vendors, few have settled on purist implementations.

For example, Apple's Mac OS X was originally based on microkernel architecture, but now runs system processes, drivers and the operating environment in **BSD-like subsystems**. Microsoft NT and derivative operating systems also went through multiple changes, moving critical performance components such as graphical and I/O subsystems in and out of the system kernel to find the right balance of stability, performance and predictability. These changes make NT a **hybrid operating system**. On the other hand, freeware development communities such as FSF, FreeBSD and NetBSD have mostly adopted monolithic designs (for example, **Linux kernel**) and have gradually introduced modularity into selected kernel sections (for example, device drivers).

So what difference does kernel architecture make to routing and control?

Monolithic Versus Microkernel Network Operating System Designs

In the network world, both monolithic and microkernel designs can be used with success.

However, the ever-growing requirements for a system kernel quickly turn any classic implementation into a compromise. Most notably, the capability to support a real-time forwarding plane along with stateful and stateless forwarding models and extensive state replication requires a mix of features not available from any existing monolithic or microkernel OS implementation.

This lack can be overcome in two ways.

First, a network OS can be constrained to a limited class of products by design. For instance, if the OS is not intended for mid- to low-level routing platforms, some requirements can be lifted. The same can be done for flow-based forwarding devices, such as security appliances. This artificial restriction allows the network operating systems to stay closer to their general-purpose siblings—at the cost of fracturing the product lineup. Different network element classes will now have to maintain their own operating systems, along with unique code bases and protocol stacks, which may negatively affect code maturity and customer experience.

Second, the network OS can evolve into a specialized design that combines the architecture and advantages of multiple classic implementations.

This custom kernel architecture is a more ambitious development goal because the network OS gets further away from the donor OS, but the end result can offer the benefits of feature consistency, code maturity, and operating experience. This is the design path that Juniper selected for JUNOS.

JUNOS Software Kernel

According to the formal criteria, the JUNOS kernel is fully customizable (Figure 3). At the very top is a portion of code that can be considered a microkernel. It is responsible for real-time packet operations and memory management, as well as interrupts and CPU resources. One level below it is a more conventional kernel that contains a scheduler, memory manager and device drivers in a package that looks more like a monolithic design. Finally, there are user-level (POSIX) processes that actually serve the kernel and implement functions normally residing inside the kernels of classic monolithic router operating systems. Some of these processes can be compound or run on external CPUs (or packet forwarding engines). In JUNOS Software, examples include periodic hello management, kernel state replication, and protected system domains (PSDs).

The entire structure is strictly hierarchical, with no underlying layers dependent on the operations of the top layers. This high degree of virtualization allows the JUNOS kernel to be both fast and flexible.

However, even the most advanced kernel structure is not a revenue-generating asset of the network element. Uptime is the only measurable metric of system stability and quality. This is why the fundamental difference between the JUNOS kernel and competing designs lies in the focus on reliability.

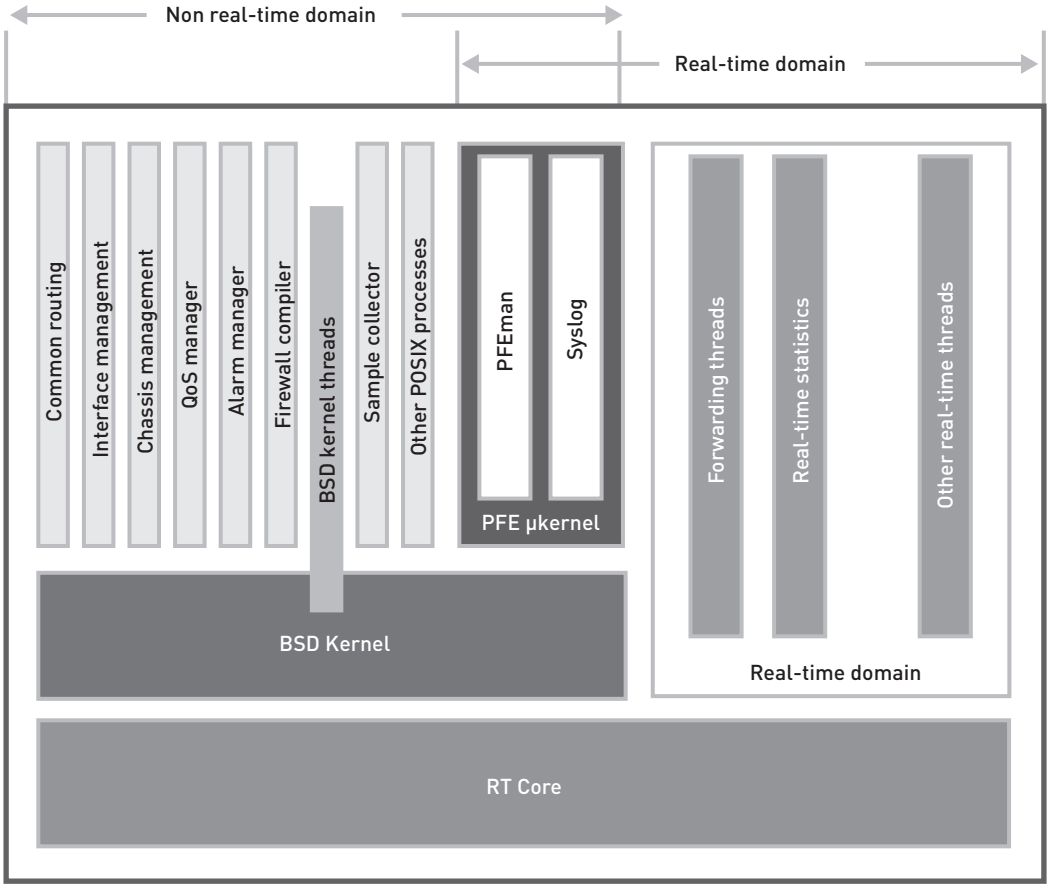


Figure 3: Generic JUNOS Software 9.0 architectural structure

Coupled with Juniper's industry-leading nonstop active routing and system upgrade implementation, kernel state replication acts as the cornerstone for continuous operation. In fact, the JUNOS redundancy scheme is designed to protect data plane stability and routing protocol adjacencies at the same time. With in-service software upgrade, networks powered by JUNOS are becoming immune to the downtime related to the introduction of new features or bug fixes, enabling them to approach true continuous operation. Continuous operation demands

that the integrity of the control and forwarding planes remains intact in the event of failover or system upgrades, including minor and major release changes. Devices running JUNOS Software will not miss or delay any routing updates when either a failure or a planned upgrade event occurs.

This goal of continuous operation under all circumstances and during maintenance tasks is ambitious, and it reflects Juniper's innovation and network expertise, which is unique among network vendors.

Process Scheduling in JUNOS Software

Innovation in JUNOS Software does not stop at the kernel level; rather, it extends to all aspects of system operation.

As mentioned before, there are two tiers of schedulers in JUNOS, the topmost becoming active in systems with a software data plane to ensure the real-time handling of incoming packets. It operates in real time and ensures that quality of service (QoS) requirements are met in the forwarding path.

The second-tier (non-real-time) scheduler resides in the base JUNOS kernel and is similar to its FreeBSD counterpart. It is responsible for scheduling system and user processes in a system to enable preemptive multitasking.

In addition, a third-tier scheduler exists within some multithreaded user-level processes, where threads operate in a cooperative, multitasking model. When a compound process gets the CPU share, it may treat it like a virtual CPU, with threads taking and leaving the processor according to their execution flow and the sequence of atomic operations. This approach allows closely coupled threads to run in a cooperatively multitasking environment and avoid being entangled in extensive IPC and resource- locking activities (Figure 4).

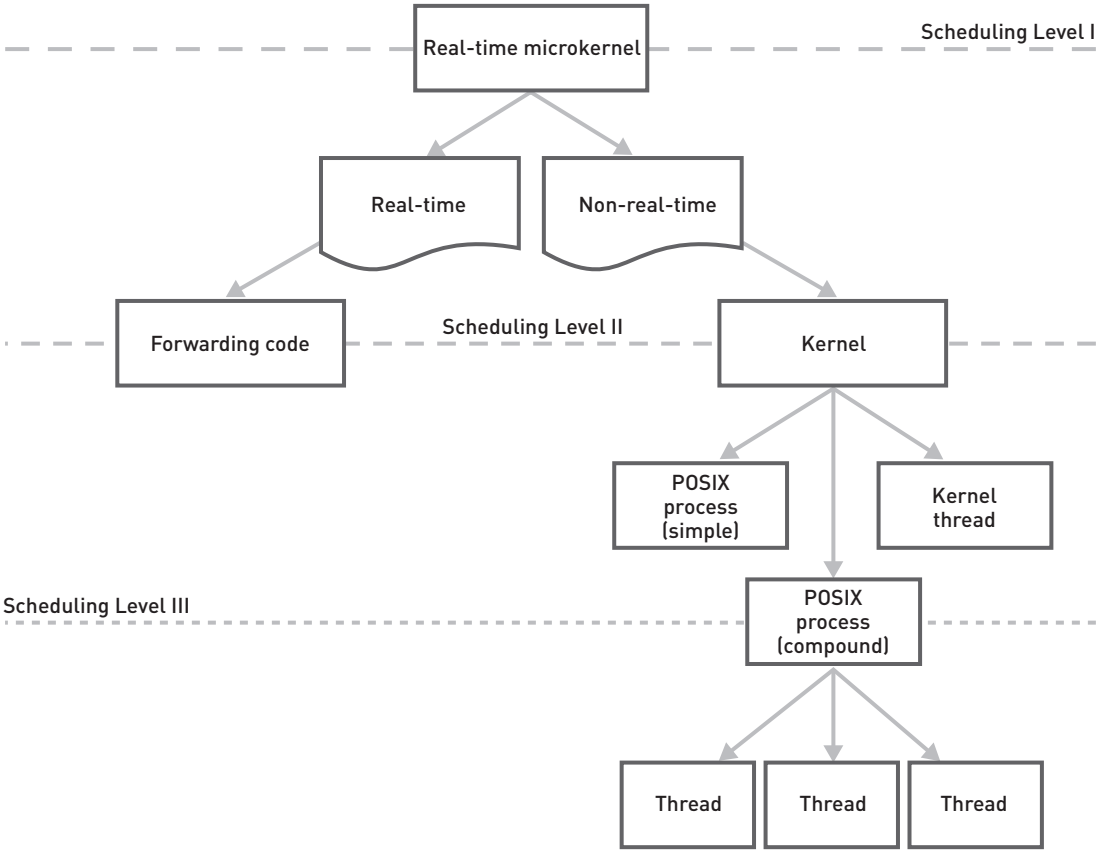


Figure 4: Multilevel CPU scheduling in JUNOS Software

Another interesting aspect of multi-tiered scheduling is resource separation. Unlike first-generation designs, JUNOS systems with a software forwarding plane cannot freeze when overloaded with data packets, as the first-level scheduler will continue granting CPU cycles to the control plane.

JUNOS Software Routing Protocol Process

The routing protocol process daemon (RPD) is the most complex process in a JUNOS system. It not only contains much of the actual code for routing protocols, but also has its own scheduler and memory manager. The scheduler within RPD implements a cooperative multitasking model, in which each thread is responsible for releasing the CPU after an atomic operation has been completed. This design allows several closely related threads to coexist without the overhead of IPC and to scale without risk of unwanted interactions and mutual locks.

The threads within RPD are highly modular and may also run externally as standalone POSIX processes—this is, for example, how many periodic protocol operations are performed. In the early days of RPD, each protocol was responsible for its own adjacency management and control. Now, most keepalive processing resides outside RPD, in the Bidirectional Forwarding Detection protocol (BFD) daemon and periodic packet management process daemon (PPMD), which are, in turn, distributed between the routing engine and the line cards. The unique capability of RPD to combine preemptive and cooperative multitasking powers the most scalable routing stack in the market.

Compound processes similar to RPD are known to be very effective but sometimes are criticized for the lack of protection between components. It has been said that a failing thread will cause the entire protocol stack to restart. Although this is a valid point, it is easy to compare the impact of this error against the performance of the alternative structure, where every routing protocol runs in a dedicated memory space.

Assume that the router serves business VPN customers, and the ultimate revenue-generating product is continuous reachability between remote sites. At the very top is a BGP process responsible for creating forwarding table entries. Those entries are ultimately programmed into a packet path ASIC for the actual header lookup and forwarding. If the BGP process hits a bug and restarts, forwarding table entries may become stale and would have to be flushed, thus disrupting customer traffic. But BGP relies on lower protocols in the stack for traffic engineering and topology information, and it will not be able to create the forwarding table without OSPF or RSVP. If any of these processes are restarted, BGP will also be affected (Figure 5). This case supports the benefits of running BGP, OSPF and RSVP in shared memory space, where the protocols can access common data without IPC overhead.

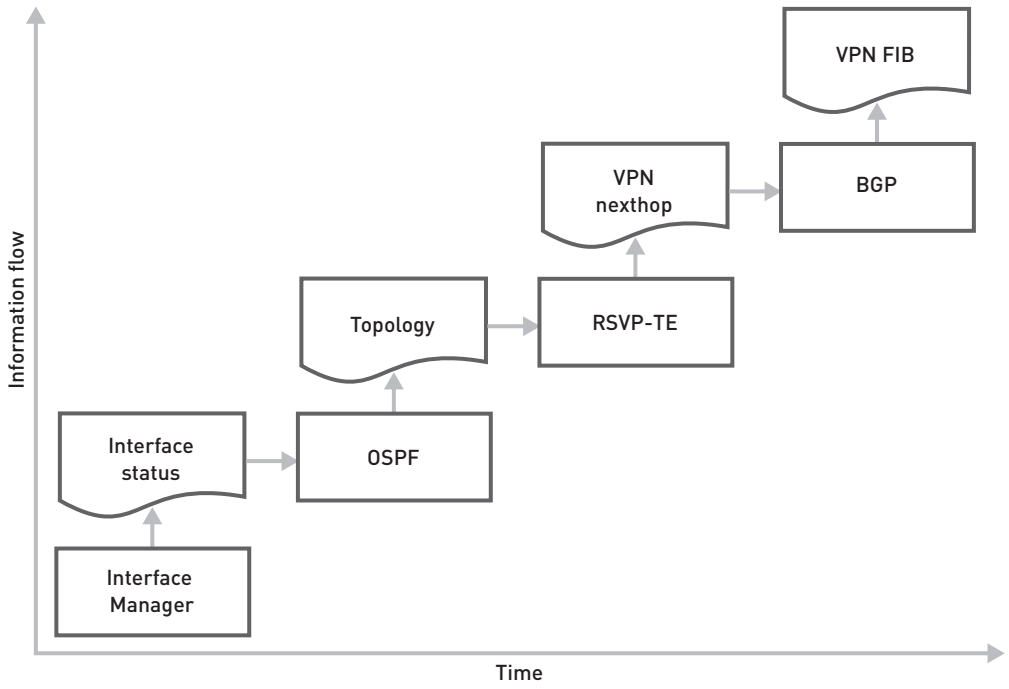


Figure 5: Hierarchical protocol stack operation

In a reverse example, several routing protocols legitimately operate at the same level and do not depend on each other. One case would be unicast family BGP and Protocol Independent Multicast (PIM). Although both depend on reachability information about connected and IGP known networks, failures in one protocol can be safely ignored in the other. For instance, unicast forwarding to remote BGP known networks can continue even if multicast forwarding is disrupted by PIM failure. In this case, the multicast and unicast portions of the routing code are better off stored in different protected domains so they do not affect each other.

Looking deeper into the realm of exceptions, we find that they occur due to software and hardware failures alike. A faulty memory bank may yield the same effect as software that references a corrupt pointer—in both cases, the process will most likely be restarted by a system.

In general, the challenge in ensuring continuous operation is fourfold:

- First, the existing forwarding entries should not be affected. Restart of a process should not affect the traffic flowing through the router.
- Second, the existing forwarding entries should not become stale. Routers should not misdirect traffic in the event of a topology change (or lack thereof).
- Third, protocol operation should have low overhead and be well contained. Excessive CPU utilization and deadlocks are not allowed as they negatively affect node stability.
- Fourth, the routing protocol peers should not be affected. The network should remain stable.

Once again, we see that few software challenges can be met by structuring in one specific way.

Routing threads may operate using a cooperative, preemptive or hybrid task model, but failure recovery still calls for state restoration using external checkpoint facilities. If vital routing information were duplicated elsewhere and could be recovered promptly, the failure would be transparent to user traffic and protocol peers alike. Transparency through prompt recovery is the principal concept underlying any NSR design and the main idea behind the contemporary Juniper Networks RPD implementation.

Instead of focusing on one technology or structure, Juniper Networks engineers evolve the JUNOS Software protocol stack according to a "survival of the fittest" principle, toward the goal of true nonstop operation, reliability and usability. State replication, checkpointing and IPC are all used to reduce the impact of software and hardware failures. The JUNOS control plane is designed to maintain speed, uptime and full state under the most unfavorable network situations.

Adapting to ever-changing real-world conditions and practical applications, the JUNOS routing architecture will continue to evolve to become even more advanced, with threads added or removed as dictated by the needs of best-in-class software design. Juniper Networks software is constantly adapted to the operating environment, and as you read this paragraph, new ideas and concepts are being integrated into JUNOS Software. Stay tuned.

Scalability

JUNOS can scale up and down to platforms of different sizes. This capability is paramount to the concept of "network OS" that can power a diverse range of network elements. The next section highlights the challenges and opportunities seen in this aspect of networking.

Scaling Down

Scaling down is the capacity of a network operating system to run on low-end hardware, thus creating a consistent user experience and ensuring the same level of equipment resilience and reliability across the entire network, from high-end to low-end routing platforms.

Achieving this goal involves multiple challenges for a system designer. Not only does the code have to be efficient on different hardware architectures, but low-end systems bring their own unique requirements, such as resource constraints, cost, and unique security and operations models. In addition, many low-end routers, firewalls and switches require at least some CPU assistance for packet forwarding or services, thus creating the need for a software forwarding path.

Taking an arbitrary second-generation router OS and executing it in a low-end system can be a challenging task, evidenced by the fact that no vendor except Juniper actually ships low-end and high-end systems running the same OS based on second-generation design principles or better.

But bringing a carrier-sized OS all the way down to the enterprise is also rewarding.

It brings immediate advantages to customers in the form of uniform management, compatibility and OPEX savings across the entire network. It also improves the original OS design. During the "fasting" exercise needed to fit the OS into low-end devices, the code is extensively reviewed, and code structure is optimized. Noncritical portions of code are removed or redesigned.

What's more, the unique requirements of variable markets (for example, security, Ethernet and enterprise) help stress-test the software in a wide range of situations, thus hardening the overall design. Scaling limits are pushed across many boundaries when the software adopts new roles and applications.

Finally, low-end systems typically ship in much larger quantities than high-end systems. The increased number of systems in the field proportionally amplifies the chances of finding nonobvious bugs and decreases the average impact of a given bug on the installed base worldwide.[1] All these factors together translate into a better product, for both carriers and enterprises.

It can be rightfully said that the requirement for scaling down has been a major source of inspiration for JUNOS developers since introduction of the Juniper Networks J Series Services Routers. The quest for efficient infrastructure has helped with such innovative projects as JUNOS Software SDK, and ultimately paved the way to the concept of one OS powering the entire network—the task that has never been achieved in history of networking before.

Scaling Up

Empowerment of a multichassis, multiple-terabit router is associated with words such as upscale and high end, all of which apply to JUNOS. However, it is mostly the control plane capacity that challenges the limits of software in modern routers with all-silicon forwarding planes. For example, a 1.6-terabit router with 80 x 10 Gigabit Ethernet core-facing interfaces may place less stress on its control plane than a 320-megabit router with 8,000 slow-speed links and a large number of IGP and BGP adjacencies behind them.

Scaling is dependent on many factors. One of the most important is proper level of modularity. As discussed in the previous sections, poor containment of intermodule interactions can cause exponential growth in supplementary operations and bring a system to gridlock.

Another factor is design goal and associated architectural decisions and degree of complexity. For instance, if a router was never intended to support 5,000 MPLS LSP circuits, this number may still be configurable, but will not operate reliably and predictably. The infrastructure changes required to fix this issue can be quite significant.

Realistic, multidimensional scaling is an equivalent of the Dhrystone[2] benchmark. This scaling is how a routing system proves itself to be commercially attractive to customers. Whenever discussing scaling, it is always good to ask vendors to stand behind their announced scaling limits. For example, the capability to configure 100,000 logical interfaces on a router does not necessarily mean that such a configuration is viable, as issues may arise on different fronts—slow responses to user commands, software timeouts and protocol adjacency loss. Vendor commitment means that the advertised limits are routinely checked and tested and new feature development occurs according to relevant expectations.

Scaling is where JUNOS Software delivers.

Some of the biggest networks in the world are built around JUNOS scaling capacities, supporting thousands of BGP and IGP peers on the same device. Likewise, core routers powered by JUNOS can support tens of thousands of transit MPLS label-switched paths (LSPs). With its industry-leading slot density on the Juniper Networks T Series Core Routers, JUNOS has proven to be one of the most scalable network operating systems in existence.

[1]This statement assumes a good systems test methodology, where toxic defects are never shipped to customers and chances for the widely experienced software problems are relatively small.

[2]A short synthetic benchmark program by Reinhold Weicker, intended to be representative of system (integer) programming.

Architecture and Infrastructure

This section addresses architecture and infrastructure concerns related to parallelism, flexibility and portability, and open architecture.

Parallelism

Advances in multicore CPU development and the capability to run several routing processors in a system constitute the basis for increased efficiency in a router control plane. However, finding the right balance of price and performance can also be very difficult.

Unlike the data mining and computational tasks of supercomputers, processing of network updates is not a static job. A block of topology changes cannot be prescheduled and then sliced across multiple CPUs. In routers and switches, network state changes asynchronously (as events happen), thus rendering time-based load sharing irrelevant.

Sometimes vendors try to solve this dilemma by statically sharing the load in functional, rather than temporal, domains. In other words, they claim that if the router OS can use separate routing processors for different tasks (for example, OSPF or BGP), it can also distribute the bulk of data processing across multiple CPUs.

To understand whether this is a valid assumption, let's consider a typical CPU utilization capture (Figure 6). What is interesting here is that the different processes are not computationally active at the same time—OSPF and BGP do not compete for CPU cycles. Unless the router runs multiple same-level protocols simultaneously, the well-designed network protocol stack stays fairly orthogonal. Different protocols serve different needs and seldom converge at the same time.

```
>show processes cpu seconds unicast
Date       Average RIP OSPF BGP RIPng OSPF6 BGP4+ RA ISIS
01/22 15:48:19  3   0 0  0 0  0  0  0  0
01:22 15:48:20  3   0 1  0 0  0  0  0  0
01/22 15:49:18  3   0 0  1 0  0  0  0  0
```

Figure 6: Typical CPU times capture (from NEC 8800 product documentation)

For instance, an IGP topology change may trigger a Dijkstra algorithm computation; until it is complete, BGP next-hop updates do not make much sense. At the same time, all protected MPLS LSPs should fall on precomputed alternate paths and not cause major RSVP activities.

Thus, the gain from placing different processes of a single control plane onto physically separate CPUs may be limited, while the loss from the overhead functions such as synchronization and distributed memory unification may be significant.

Does this mean that the concept of parallelism is not applicable to the routing processors? Not at all.

Good coding practice and modern compilers can make excellent use of multicore and SMP hardware, while clustered routing engines are indispensable when building multichassis (single control and data plane spanning multiple chassis) or segmented (multiple control and data planes within a single physical chassis) network devices. Furthermore, high-end designs may allow for independent scaling of control and forwarding planes, as implemented in the highly acclaimed Juniper Networks JCS1200 Control System.

With immediate access to state-of-the art processor technology, Juniper Networks engineers heavily employ parallelism in the JUNOS control plane design, targeting both elegance and functionality.

A functional solution is the one that speeds up the control plane without unwanted side effects such as limitations in forwarding capacity. When deployed in a JCS1200, JUNOS can power multiple control plane instances (system domains) at the same time without consuming revenue-generating slots in the router chassis. Moreover, the JUNOS architecture can run multiple routing systems (including third-party code) from a single rack of routing engines, allowing an arbitrary mix-and-match of control plane and data plane resources within a point of presence (POP). These unique capabilities translate into immediate CAPEX savings, because a massively parallel control plane can be built independent of the forwarding plane and will never confront a limited common resource (such as the number of physical routers or a number of slots in each chassis).

Elegance means the design should also bring other technical advantages: for instance, bypassing space and power requirements associated with the embedded chassis and thus enabling use of faster silicon and speeding up the control plane. Higher CPU speed and memory limits can substantially improve the convergence and scaling characteristics of the entire routing domain.

The goal of Juniper design philosophy is tangible benefits to our customers—without cutting corners.

Flexibility and Portability

A sign of a good OS design is the capability to adapt the common software platform to various needs. In the network world, this equates to the adoption of new hardware and markets under the same operating system.

The capability to extend the common operating system over several products brings the following important benefits to customers:

- Reduced OPEX from consistent UI experience and common management interface
- Same code for all protocols; no unique defects and interoperability issues
- Common schedule for software releases; a unified feature set in the control plane
- Accelerated technology introduction; once developed, the feature ships on many platforms

Technology companies are in constant search of innovation both internally and externally. New network products can be developed in-house or within partnerships or acquired. Ideally, a modern network OS should be able to absorb domestic (internal) hardware platforms as well as foreign (acquired) products, with the latter being gradually folded into the mainstream software line (Figure 7).

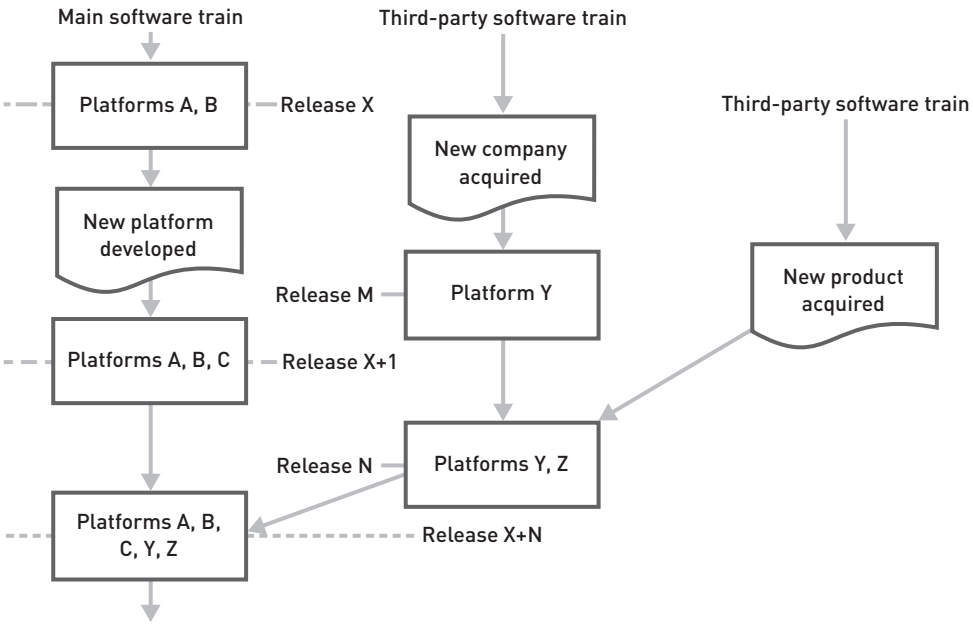


Figure 7: Product consolidation under a common operating system

The capability to absorb in-house and foreign innovations in this way is a function of both software engineering discipline and a flexible, well-designed OS that can be adapted to a wide range of applications.

On the contrary, the continuous emergence of internally developed platforms from the same vendor featuring different software trains and versions can signify the lack of a flexible and stable software foundation.

For example, when the same company develops a core router with one OS, an Ethernet switch with another, and a data center switch with a third, this likely means that in-house R&D groups considered and rejected readily available OS designs as impractical or unfit. Although partial integration may still exist through a unified command-line interface (CLI) and shared code and features, the main message is that the existing software designs were not flexible enough to be easily adapted to new markets and possibilities. As a result, customers end up with a fractured software lineup, having to learn and maintain loosely related or completely unrelated software trains and develop expertise in all of them—an operationally suboptimal approach.

In contrast to this trend, Juniper has never used a multitrain approach with JUNOS Software and has never initiated multiple operating system projects. Since its inception in 1996, JUNOS has been successfully ported to a number of Intel, MIPS, and PowerPC architectures and currently powers a broad spectrum of routing products ranging from the world's fastest Juniper Networks T1600 Core Router to low-end routing devices, Ethernet switches, and security appliances. Juniper's clear goal is to keep all products (both internally developed and acquired together with industry-leading companies and talent) under the same JUNOS Software umbrella.

Degrees of Modularity

Software modularity, as previously described, has focused on the case where tasks are split into multiple loosely coupled modules. This type of modularity is called "horizontal," as it aims at limiting dependency and mutual impact between processes operating at the same peer level. Another interesting degree of modularity is known as "vertical modularity," where modular layers are defined between parts of the operating system in the vertical direction.

Without vertical modularity, a network OS remains built for a specific hardware and services layout. When porting to a new target, much of this infrastructure has to be rewritten. For example, both software- and hardware-based routers can provide a stateful firewall service, but they require dramatically different implementations. Without a proper vertical modularity in place, these service implementations will not have much in common, which will ultimately translate into an inconsistent user experience.

Vertical modularity solves this problem, because most OS functions become abstracted from lower-level architecture and hardware capabilities. Interaction between upper and lower OS levels happens via well-known subroutine calls. Although vertical modularity itself is almost invisible to the end user, it eliminates much of the inconsistency between various OS implementations. This can be readily appreciated by network operations center (NOC) personnel who no longer deal with platform-specific singularities and code defects. Vertical modularity is an ongoing project, and the JUNOS Software team has always been very innovative in this area.

Open Architecture

An interesting implication of vertical modularity is the capability to structure code well enough to document appropriate software interfaces and allow external pluggable code. While a high degree of modularity within a system allows easy porting to different and diverse hardware architectures, a well-defined and documented application programming interface (API) can be made available to third parties for development of their own applications.

In JUNOS, the high degree of modularity and documentation eventually took the form of the Partner Solution Development Platform (PSDP), which opened the API and tool chain specific to Juniper to customers and integrators worldwide. PSDP allows these customers and integrators to co-design the operating system, fitting it precisely to their needs, especially in support of advanced and confidential applications. The degree of development may vary from minor changes to software appearance to full-scale custom packet processing tailored to specific needs.

The Juniper Networks Software Developer's Kit (SDK) highlights the achievements of JUNOS in network code design and engineering and reflects the innovation that is integral to Juniper's corporate culture. This high level of synergy between original equipment manufacturer (OEM) vendors and operators promises to enable creation of new services and competitive business differentiators, thus removing the barriers to network transformation. Just as the open-source FreeBSD was the donor OS for JUNOS Software, with the Juniper Networks SDK, JUNOS is now a platform open to all independent developers.

Product Maintenance

Another important characteristic of products is maintainability. It covers the process of dealing with software defects and new features, abilities to improve existing code, and the introduction of new services and capabilities. It also makes a big difference in the number and quality of NOC personnel that is required to run a network. Maintainability is where a large portion of OPEX resides.

Self-Healing

Routers are complex devices that depend on thousands of electronic components and millions of code lines to operate. This is why some portion of the router installed base will almost inevitably experience software or hardware defects over the product life span.

So far, we have been describing the recovery process, in which state replication and process restarts are the basis of continuous operation. In most cases, JUNOS will recover so efficiently that customers never notice the problem, unless they closely monitor the system logs. A failing process may restart instantly with all the correct state information, and the router operation will not be affected.

But even the best recovery process does not provide healing; software or hardware component remains defective and may cause repeated failures if it experiences the same condition again. The root cause for the failure needs to be tracked and eliminated, either through a software fix or a hardware replacement.

Traditionally, this bug investigation begins with a technical assistance center (TAC) ticket opened by a customer and requires intensive interaction between the customer and vendor engineers. Once identified, the problem is usually resolved through a work-around, software upgrade or hardware replacement, all of which must be performed manually.

Since the early days of JUNOS, Juniper Networks routers were designed to include the built-in instrumentation needed to diagnose and remedy problems quickly. Reflecting Juniper's origins as a carrier-class routing company, every JUNOS system in existence comes with an extensive array of software and hardware gear dedicated to device monitoring and analysis. Juniper has been a pioneer in the industry with innovations such as persistent logging, automatic core file creation and development tools (such as GDB) embedded in JUNOS Software, all facilitating fast defect tracing and decision making). In the traditional support model, customers and Juniper Networks TAC (JTAC) engineers jointly use those tools to zero in on a possible issue and resolve it via configuration change or software fix.

In many cases, this is enough to resolve a case in real time, as soon as the defect traces are made available to Juniper Networks Customer Support.

However, Juniper would never have become a market leader without a passion for innovation. We see routing systems with embedded intelligence and self-healing capabilities as the tools for ensuring survivability and improving the user experience. Going far beyond the automated hardware self-checking normally available from many vendors, JUNOS can not only collect data and analyze its own health, but can also report this state back to the customer and to the JTAC with the patent-pending Advanced Insight Service (AIS) technology. As a result, the router that experiences problems can get immediate vendor attention around the clock and without involving NOC personnel. A support case can be automatically created and resolved before operators are aware of the issue. If a code change is needed, it will go into the next maintenance or major JUNOS release and will be available through a seamless upgrade on the router. This cycle is the basis of self-healing JUNOS Software operation and paves the way to dramatic OPEX savings for existing networks.

The main difference between AIS and similar call-home systems is the degree of embedded intelligence.

AIS-enabled JUNOS Software both monitors itself for apparent failures such as a process crash or laser malfunction and proactively waits for early signs of problems such as degrading storage performance or increasing number of unresolved packets in the forwarding path. Triggered and periodic health checks are continuously improved based on actual field cases encountered and resolved by the JTAC, thus integrating the collective human expertise into the running of JUNOS systems. Further, AIS is fully programmable with new health metrics and triggers that customers can add. Better yet, in its basic form, AIS comes with every JUNOS system—for free.

Troubleshooting

An often forgotten but very important aspect of functional separation is the capability to troubleshoot and analyze a production system. As the amount of code that constitutes a network operating system is often measured in hundreds of megabytes, software errors are bound to occur in even the most robust and well-regressed designs. Some errors may be discovered only after a huge number of protocol transactions have accumulated on a system with many years of continuous operation. Defects of this nature can rarely be predicted or uncovered even with extensive system testing.

After the error is triggered and the damage is contained by means of automatic software recovery, the next step is to collect the information necessary to find the problem and to fix it in the production code base. The speed and effectiveness of this process can be critical to the success of the entire network installation because most unresolved code defects are absolutely not acceptable in production networks.

This is where proper functional separation comes into major play. When a software defect is seen, it is likely to become visible via an error message or a faulty process restart (if a process can no longer continue). Because uptime is paramount to the business of networking, routers are designed to restart the failing subsystem as quickly as possible, typically in a matter of milliseconds.

When this happens, the original error state is lost, and software engineers will not be able to poke around a live system for possible root causes of the glitch. Unless the defect is trivial and easily understood, code designers may take some time to recreate and understand the issue. Offsite reproduction can be challenging, because replicating the exact network conditions and sequence of events can be difficult, and sometimes impossible. In this case, the post-mortem memory image **(core dump**) of the failing process is indispensable because it contains the state of data structures and variables, which can be examined for integrity. It is not uncommon for JUNOS engineers to resolve a defect just by analyzing the process core dump.

The catch here is that in tightly coupled processes, the failure of one process may actually be triggered by an error in another process. For example, RSVP may accept a "poisoned" traffic engineering database from a link-state IGP process and subsequently fail. If the processes run in different memory spaces, RSVP will dump the core, and IGP will continue running with a faulty state. This situation not only hampers troubleshooting, but also potentially brings more damage to the system because the state remains inconsistent.

The issue of proper functional separation also has implications for software engineering managers. It is a common practice to organize development groups according to code structure, and interprocess software defects can become difficult to troubleshoot because of organizational boundaries. Improper code structure can easily translate into a TAC nightmare, where a defect is regularly seen in the customer network, but cannot be reliably reproduced in the lab or even assigned to the right software engineering group.

In JUNOS Software, the balance between the amount of restartable code and the core dump is tuned to improve troubleshooting and ensure quick problem resolution. JUNOS is intended to be a robust operating system and to deliver the maximum amount of information to engineering should an error occur. This design helps ensure that most software defects resulting in code restarts are resolved within a short time period, often as soon as the core file is delivered.

Quality and Reliability

System integrity is vital, and numerous engineering processes are devoted to ensuring it. The following section touches on the practice of quality software products design.

System Integrity

If you were curious enough to read this paper up to this point, you should know that a great deal of work goes into the design of a modern operating system. Constant feature development and infrastructural changes mean that each new release has a significant amount of new code.

Now you might ask if the active development process can negatively affect system stability.

With any legacy software design process, the answer would be definite: Yes.

The phenomenon known to programmers as "feature bloating" is generally responsible for degrading code structure and clarity over time. As new code and bug fixes are introduced, the original design goals are lost, testing becomes too expensive, and the release process produces more and more "toxic builds" or otherwise unusable software with major problems.

This issue was recognized very early in the JUNOS development planning stage.

Back in 1996, automated system tests were not widely used, and most router vendors crafted their release methodology based on the number of changes they expected to make in the code. Typically, every new software release would come in mainstream and technology versions, with the former being a primary target for bug fixes, and the latter receiving new features. Defects were caught mainly in production networks after attempts to deploy new software, which resulted in a high number of bug fixes and occasional release deferrals.

To satisfy the needs of customers looking for a stable operational environment, "general deployment" status was used to mark safe-harbor software trains. It was typically awarded to mainstream code branches after they had run for a long enough time in early adopters' networks.

As a general rule, customers had to choose between features and stability. Technology and early deployment releases were notoriously problematic and full of errors, and the network upgrade process was a trial-and-error operation in search for the code train with a "right" combination of features and bugs.

This approach allowed router vendors to avoid building extensive test organizations, but generally led to low overall product quality. General deployment software trains lingered for years with almost no new features, while technology builds could barely be deployed in production because of reliability problems. Multiple attempts to find the balance between the two made the situation even worse due to introduction of even more software trains with different stability and feature levels.

This practice was identified as improper in the fledgling JUNOS design process. Instead, a state-of-the-art test process and pioneering release methodology were born.

Each JUNOS Software build is gated by a full regression run that is fully automated and executes for several days on hundreds of test systems simulating thousands of test cases. These test cases check for feature functionality, scaling limits, previously known defects and resilience to negative input (such as faulty routing protocol neighbors). If a failure occurs in a critical test, the final product will not be shipped until the problem is fixed. This process allows JUNOS releases to occur on a predictable, periodic basis. In fact, many customers trust JUNOS to the point that they run the very first build of each version in production. Still, every JUNOS version is entitled to the so-called regression run (if requested by customers). A regressed release is a fully tested original build with all latest bug fixes applied.

The JUNOS shipping process is based on several guiding principles:

- Every JUNOS release is gated by a systems test, and no releases with service-affecting issues are cleared for shipment.
- Regressed (maintenance) releases, by rule, deliver no new features. For example, no features were introduced between JUNOS Software 8.5R1 and 8.5R2.
- As a general rule, feature development happens only at the head of the JUNOS Software train. Experimental (engineering) branches may exist, but they are not intended for production.
- No feature backports are allowed (that is, features developed for rev 9.2 are not retrofitted into rev 8.5)
- No special or customer-specific builds are allowed. This restriction means JUNOS never receives modifications that are not applicable to the main code base or cannot pass the system test. Every change and feature request is carefully evaluated according to its value and impact worldwide; the collective expertise of all Juniper Networks customers benefits every JUNOS product.

This release process ensures the exceptional product quality customers have come to expect from Juniper over the years. Although initially met with reluctance by some customers accustomed to the randomly spaced, untested and special builds produced by other vendors, our release policy ensures that no production system receives unproven software. Customers have come to appreciate the stability in OS releases that Juniper's approach provides.

With its controlled release paradigm, Juniper has set new standards for the entire networking industry, The same approach was used later by many other design organizations.

However, the JUNOS Software design and build structure remains largely unmatched.

Unlike competitors' build processes, our build process occurs simultaneously for all Juniper Networks platforms and uses the same software repository for all products. Each code module has exactly one implementation, in both shared (common) and private (platform-specific) cases. Platform-specific and shared features are merged during the build in a well-controlled and modular fashion, thus providing a continuous array of functionality, quality and experience across all JUNOS software routing, switching and security products.

Release Process Summary

Even the best intentions for any software development are inadequate unless they can prove themselves through meaningful and repeatable results. At Juniper, we firmly believe in a strong link between software quality and release discipline, which is why we have developed criteria for meeting—or failing—our own targets.

Here is a set of metrics for judging the quality of release discipline:

- Documented design process: The Juniper Networks software design process has met the stringent TL9000 certifications requirements.
- Release schedule: JUNOS releases have been predictable and have generally occurred every three months. An inconsistent, unpredictable or repeatedly slipping release process generally indicates problems in a software organization.
- Code branching: This is a trend where a single source tree branches out to support either multiple platforms or alternative builds on the same platform with unique software features and release schedules. Branching degrades system integrity and quality because the same functionality (for example, routing) is being independently maintained and developed in different software trains. Branching is often related to poor modularity and can also be linked to poor code quality. In an attempt to satisfy a product schedule and customer requirements, software engineers use branching to avoid features (and related defects) that are not critical to their main target or customer. As a result, the field ends up with several implementations of the same functionality on similar or even identical hardware platforms.

  Although JUNOS powers many platforms with vastly different capabilities, it is always built from one source tree with core and platform-specific sections. The interface between the two parts is highly modular and well documented, with no overlap in functionality. There is no branching in JUNOS Software code.
- Code patching: To speed defect resolution, some vendors provide code patching or point bug-fix capability, so that selected defects can be patched on a running operating system. Although technically very easy to do, code patching significantly degrades production software with uncontrolled and unregressed code infusions. Production systems with code patches become unique in their software state, which makes them expensive to control and maintain.

  After some early experiments with code patching, JUNOS ceased this process in favor of a more comprehensive and coherent in-service software upgrade (ISSU) and nonstop routing implementation.
- Customer-specific builds: The use of custom builds is typically the result of failures in a software design methodology and constitutes a form of code branching. If a feature or specific bug fix is of interest to a particular customer, it should be ported to the main development tree instead of accommodated through a separate build. Code branching almost inevitably has major implications for a product such as insufficient test coverage, feature inconsistency and delays. JUNOS is not delivered in customer-specific build forms.
- Features in minor (regressed) releases: Under Juniper's release methodology, which has been adopted by many other companies, minor software releases are regressed builds that almost exclusively contain bug fixes. Sometimes the bug fix may also enable functionality that existed but was not made public in the original feature release. However, this should not be a common case. If a vendor consistently delivers new functionality along with bug fixes, this negatively affects the entire release process and methodology because subsequent regressed releases may have new caveats based on the new feature code they have received.

Final Product Quality and Stability

Good code quality in a network operating system means that the OS runs and delivers functionality without problems and caveats—that is, it provides revenue-generating functionality right out of the box with no supervision. Customers often measure software quality by the number of defects they experience in production per month or per year. In the most severe cases, they also record the downtime associated with software defects.

Generally, all software problems experienced by a router can be divided into three major categories:

- Regression defects are those introduced by the new code; a regression defect indicates that something is broken that was working before.
- Existing software defects are those previously present in the code that were either unnoticed or (up to a certain point) harmless until they significantly affected router operation.
- New feature fallouts are caveats in new code.

Juniper's software release methodology was created to greatly reduce the number of software defects of all types, providing the foundation for the high quality of JUNOS. Regression defects are mostly caught very early in their lifetime at the forefront of the code development.

Existing software defects represent a more challenging case. JTAC personnel, SE community or customers can report them.

Some defects are, in fact, uncovered years after the original design. The verity that they were not found by the system test or by customers typically means that they are not severe or that they occur in rare circumstances, thus mitigating their possible impact. For instance, the wrong integer type (signed versus unsigned) may affect a 32-bit counter only when it crosses the 2G boundary. Years of uptime may be needed to reveal this defect, and most customers will never see it.

In any case, once a new defect class is found, it is scripted and added to the systest library of test cases. This guarantees that the same defect will not leak to the field again, as it will be filtered out early in the build process. This systest library, along with the JUNOS code itself, is among the "crown jewels" of Juniper intellectual property in the area of networking.

As a result, although any significant feature may take several years of development, Juniper has an excellent track record for making sure things work right at the very first release, a record that is unmatched in the networking industry.

Conclusion

Designing a modern operating system is a difficult task that challenges developers with complex problems and choices. Any specific feature implementation is rarely perfect and often strikes a subtle balance among a broad range of reliability, performance and scaling metrics.

This balance is something that JUNOS developers work hard to deliver every day.

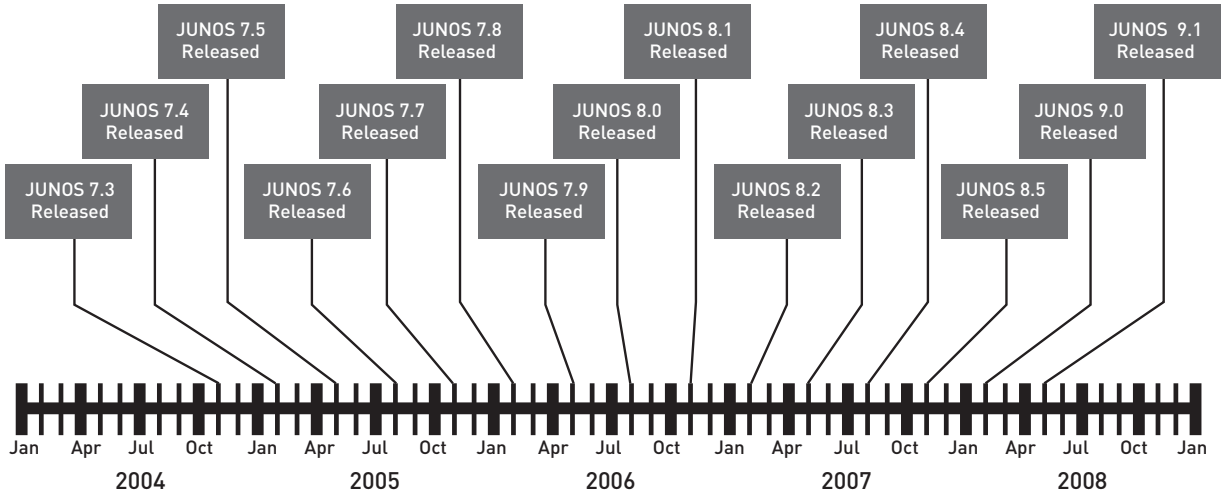
The best way to appreciate JUNOS Software features and quality is to start using JUNOS in production, alongside any other product in a similar deployment scenario. At Juniper Networks, we go beyond what others consider the norm to ensure that our software leads the industry in performance, resilience and reliability.

What Makes JUNOS Software Different

Features

FEATURE	BENEFITS TO CUSTOMER
JUNOS Software development efforts 1996–2008	Unmatched experience and expertise
Open-software policy	SDK, custom programming and scripting
Self-healing	Automatic core processing and Advanced Insight Solution feature
Scaling down	Low-end systems with JUNOS software onboard
Scaling up	Juniper Networks TX Matrix Core Router, 32,000 logical interfaces, 4000 BGP peers, and 4000 VPN routing and forwarding (VRF) tables
Nonstop operation	Unified ISSU and nonstop routing for all code revisions
Intelligent thread scheduling	No process deadlocks under all circumstances
Strict engineering and release discipline	Code quality, no regressions, and features on time

Predictability: JUNOS Software Releases 2004–2008



About Juniper Networks

Juniper Networks, Inc. is the leader in high-performance networking. Juniper offers a high-performance network infrastructure that creates a responsive and trusted environment for accelerating the deployment of services and applications over a single network. This fuels high-performance businesses. Additional information can be found at **www.juniper.net.**

Corporate And Sales Headquarters
Juniper Networks, Inc.
1194 North Mathilda Avenue
Sunnyvale, CA 94089 USA
Phone: 888.JUNIPER
(888.586.4737)
or 408.745.2000
Fax: 408.745.2100

APAC Headquarters
Juniper Networks (Hong Kong)
26/F, Cityplaza One
1111 King's Road
Taikoo Shing, Hong Kong
Phone: 852.2332.3636
Fax: 852.2574.7803

EMEA Headquarters
Juniper Networks Ireland
Airside Business Park
Swords, County Dublin,
Ireland
Phone: 35.31.8903.600
Fax: 35.31.8903.601

To purchase Juniper Networks solutions, please contact your Juniper Networks representative at **1-866-298-6428** or authorized reseller.

2000264-001-EN Feb 2009

Printed on recycled paper.