Designing Extensible IP Router Software

Mark Handley[†*] Eddie Kohler[‡*] Atanu Ghosh[*] Orion Hodson[*] Pavlin Radoslavov[*]
[*]*International Computer Science Institute* [†]*University College, London* [‡]*UCLA*
{*mjh, kohler, atanu, hodson, pavlin*}*@xorp.org*

ABSTRACT

Many problems with today's Internet routing infrastructure—slow BGP convergence times exacerbated by timer-based route scanners, the difficulty of evaluating new protocols—are not architectural or protocol problems, but *software* problems. Router software designers have tackled scaling challenges above all, treating extensibility and latency concerns as secondary. At this point in the Internet's evolution, however, further scaling and security issues require tackling latency and extensibility head-on.

We present the design and implementation of XORP, an IP routing software stack with strong emphases on latency, scaling, and extensibility. XORP is event-driven, and aims to respond to routing changes with minimal delay—an increasingly crucial requirement, given rising expectations for Internet reliability and convergence time. The XORP design consists of a composable framework of routing processes, each in turn composed of modular processing stages through which routes flow. Extensibility and latency concerns have influenced XORP throughout, from IPC mechanisms to process arrangements to intra-process software structure, and leading to novel designs. In this paper we discuss XORP's design and implementation, and evaluate the resulting software against our performance and extensibility goals.

1 INTRODUCTION

The Internet has been fabulously successful; previously unimagined applications frequently arise, and changing usage patterns have been accommodated with relative ease. But underneath this veneer, the low-level protocols that support the Internet have largely ossified, and stresses are beginning to show. Examples include security and convergence problems with BGP routing [18], deployment problems with multicast [10], QoS, and IPv6, and the lack of effective defense mechanisms against denial-of-service attacks. The blame for this ossification has been placed at various technical and non-technical points in the Internet architecture, from limits of layered protocol design [4] to the natural conservatism of commercial interests [9]; suggested solutions have included widespread overlay networks [23, 24] and active networking [6, 30]. But less attention has been paid to a simple, yet fundamental, underlying cause: the lack of extensible, robust, high-performance router software.

The router software market is closed: each vendor's routers will run only that vendor's software. This makes it almost impossible for researchers to experiment in real networks, or to develop proof-of-concept code that might convince network operators that there are alternatives to current practice. A lack of open router APIs additionally excludes startup companies as a channel for change.

The solution seems simple in principle: router software should have open APIs. (This somewhat resembles active networks, but we believe that a more conservative approach is more likely to see real-world deployment.) Unfortunately, extensibility can conflict with the other fundamental goals of performance and robustness, and with the sheer complexity presented by routing protocols like BGP. Relatively few software systems have robustness and security goals as stringent as those of routers, where localized instability or misconfiguration can ripple throughout the Internet [3]. Routers must also juggle hundreds of thousands of routes, which can be installed and withdrawn en masse as links go up and down. This limits the time and space available for extensions to run. Unsurprisingly, then, existing router software was not written with third-party extension in mind, so it doesn't generally include the right hooks, extension mechanisms and security boundaries.

We therefore saw the need for a new suite of router software: an integrated open-source software router platform running on commodity hardware, and viable both in research and production. The software architecture would have extensibility as a primary goal, permitting experimental protocol deployment with minimal risk to existing services. Internet researchers needing access to router software would share a common platform for experimentation, and get an obvious path to deployment for free. The loop between research and realistic real-world experimentation would eventually close, allowing innovation to take place much more freely. We have made significant progress towards building this system, which we call *XORP*, the eXtensible Open Router Platform [13].

This paper focuses on the XORP control plane: routing protocols, the Routing Information Base (RIB), net-

Orion Hodson is currently at Microsoft Research.

This material is based upon work supported by the National Science Foundation under Grant No. 0129541. Any opinions, findings, and conclusions or recommendations expressed in the material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

work management software, and related user-level programs that make up the vast majority of software on a router today. This contrasts with the forwarding plane, which processes every packet passing through the router. Prior work on component-based forwarding planes has simultaneously achieved extensibility and good performance [16, 26], but these designs, which are based on the flow of packets, don't apply directly to complex protocol processing and route wrangling. XORP's contributions, then, consist of the strategies we used to break the control plane, and individual routing protocols, into components that facilitate both extension and good performance.

For example, we treat both BGP and the RIB as networks of routing *stages*, through which routes flow. Particular stages within those networks can combine routes from different sources using various policies, or notify other processes when routes change. Router functionality is separated into many Unix processes for robustness. A flexible IPC mechanism lets modules communicate with each other independent of whether those modules are part of the same process, or even on the same machine; this allows untrusted processes to be run entirely sandboxed, or even on different machines from the forwarding engine. XORP processes are event-driven, avoiding the widely-varying delays characteristic of timer-based designs (such as those deployed in most Cisco routers). Although XORP is still young, these design choices are stable enough to have proven their worth, and to demonstrate that extensible, scalable, and robust router software is an achievable goal.

The rest of this paper is organized as follows. After discussing related work (Section 2), we describe a generic router control plane (Section 3) and an overview of XORP (Section 4). Sections 5 and 6 describe particularly relevant parts of the XORP design: the routing stages used to compose the RIB and routing protocols like BGP and our novel inter-process communication mechanism. The remaining sections discuss our security framework; present a preliminary evaluation, which shows that XORP's extensible design does not impact its performance on macro-benchmarks; and conclude.

2 RELATED WORK

Previous work discussed XORP's requirements and high-level design strategy [13]; this paper presents specific solutions we developed to achieve those requirements. We were inspired by prior work on extensible forwarding planes, and support Click [16], one such forwarding plane, already.

Individual open-source routing protocols have long been available, including *routed* [29] for RIP, *OSPFd* [20] for OSPF, and *pimd* [14] for PIM-SM multicast routing. However, interactions between protocols can be problematic unless carefully managed. *GateD* [21] is perhaps the best known integrated routing suite, although it began as an implementation of a single routing protocol. GateD is a single process within which all routing protocols run. Such monolithic designs are fundamentally at odds with the concept of differentiated trust, whereby more experimental code can be run alongside existing services without destabilizing the whole router. MRTD [28] and BIRD [2], two other open-source IP router stacks, also use a single-process architecture. In the commercial world, Cisco IOS [7] is also a monolithic architecture; experience has shown that this significantly inhibits network operators from experimenting with Cisco's new protocol implementations.

Systems that use a multi-process architecture, permitting greater robustness, include Juniper's JunOS [15] and Cisco's most recent operating system IOS XR [8]. Unfortunately, these vendors do not make their APIs accessible to third-party developers, so we have no idea if their internal structure is well suited to extensibility. The open-source Zebra [31] and Quagga [25] stacks use multiple processes as well, but their shared inter-process API is limited in capability and may deter innovation.

Another important distinguishing factor between implementations is whether a router is *event-driven* or uses a periodic *route scanner* to resolve dependencies between routes. The scanner-based approach is simpler, but has a rather high latency before a route change actually takes effect. Cisco IOS and Zebra both use route scanners, with (as we demonstrate) a significant latency cost; MRTD and BIRD are event-driven, but this is easier given a single monolithic process. In XORP, the decision that everything is event-driven is fundamental and has been reflected in the design and implementation of all protocols, and of the IPC mechanism.

3 CONTROL PLANE FUNCTIONAL OVERVIEW

The vast majority of the software on a router is control-plane software: routing protocols, the Routing Information Base (RIB), firewall management, command-line interface, and network management—and, on modern routers, much else, including address management and "middlebox" functionality. Figure 1 shows a basic functional breakdown of the most common software on a router. The diagram's relationships correspond to those in XORP and, with small changes, those in any router. The rest of this section explores those relationships further.

The unicast routing protocols (BGP, RIP, OSPF, and IS-IS) are clearly functionally separate, and most routers only run a subset of these. However, as we will see later, the coupling between routing protocols is fairly complex. The arrows on the diagram illustrate the major flows of routing information, but other flows also exist.

The Routing Information Base (RIB) serves as the plumbing between routing protocols. Protocols such as

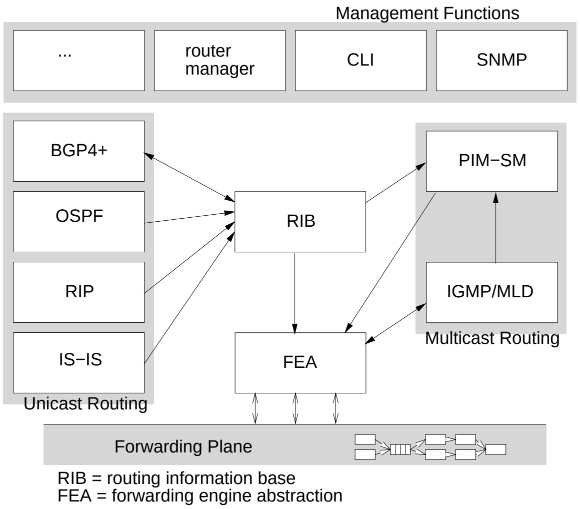


FIGURE 1—Typical router control plane functions

RIP and OSPF receive routing information from remote routers, process it to discover feasible routes, and send these routes to the RIB. As multiple protocols can supply different routes to the same destination subnet, the RIB must arbitrate between alternatives.

BGP has a more complex relationship with the RIB. Incoming IBGP routes normally indicate a *nexthop* router for a destination, rather than an immediate neighbor. If there are multiple IBGP routes to the same subnet, BGP will typically need to know the routing metrics for each choice so as to decide which route has the nearest exit (so-called "hot potato" routing). Thus, BGP must examine the routing information supplied to the RIB by other routing protocols to make its own routing decisions.

A key instrument of routing policy is the process of route redistribution, where routes from one routing protocol that match certain policy filters are redistributed into another routing protocol for advertisement to other routers. The RIB, as the one part of the system that sees everyone's routes, is central to this process.

The RIB is thus crucial to the correct functioning of a router, and should be extended only with care. Routing protocols may come and go, but the RIB should ideally be general enough to cope with them all; or failing that, it should support small, targeted extensions that are easily checked for correctness.

The Forwarding Engine Abstraction (FEA) provides a stable API for communicating with a forwarding engine or engines. In principle, its role is syntactic, and many single-platform routers leave it out, communicating with the forwarding plane directly.

PIM-SM (Protocol Independent Multicast—Sparse Mode [12]) and IGMP provide multicast routing functionality, with PIM performing the actual routing and IGMP informing PIM of the existence of local receivers. PIM contributes routes not to the RIB, but directly via the FEA to the forwarding engine. Thus, the FEA's interface is important for more than just the RIB. However, PIM does use the RIB's routing information to decide on the reverse path back to a multicast source.

The "Router Manager" holds the router configuration and starts, configures, and stops protocols and other router functionality. It hides the router's internal structure from the user, providing operators with unified management interfaces for examination and reconfiguration.

Our goal is a router control plane that provides all this functionality, including all the most widely used routing protocols, in a way that encourages extensibility. At this point, we do *not* automatically protect operators from malicious extensions or experimental code. Instead, our software architecture aims to *minimize extension footprint*, making it feasible for operators to check the code themselves. This requires a fundamental design shift from the monolithic, closely-coupled designs currently prevalent. In Section 7 we will discuss in more detail our current and future plans for XORP's security framework.

4 XORP OVERVIEW

The XORP control plane implements this functionality diagram as a set of communicating processes. Each routing protocol and management function is implemented by a separate process, as are the RIB and the FEA. Processes communicate with one another using an extensible IPC mechanism called XORP Resource Locators, or XRLs. This blurs the distinction between intra- and inter-process calls, and will even support transparent communication with non-XORP processes. The one important process not represented on the diagram is the *Finder*, which acts as a broker for IPC requests; see Section 6.2. (XORP 1.0 supports BGP and RIP; support for OSPF and IS-IS is under development.)

This multi-process design limits the coupling between components; misbehaving code, such as an experimental routing protocol, cannot directly corrupt the memory of another process. Performance is a potential downside, due to frequent IPCs; to address it, we implemented various ways to safely cache IPC results such as routes (Section 5.2.1). The multi-process approach also serves to decouple development for different functions, and encourages the development of stable APIs. Protocols such BGP and RIP are not special in the XORP design—they use APIs equally available to all. Thus, we have confidence that those APIs would prove sufficient, or nearly so, for most experimental routing protocols developed in the future.[1]

We chose to implement XORP primarily in C++, because of its object orientation and good performance. Realistic alternatives would have been C and Java. When we started implementing XORP, the choice was not completely clear cut, but we've become increasingly satisfied; for example, extensive use of C++ templates allows common source code to be used for both IPv4 and IPv6, with the compiler generating efficient implementations for both.

Each XORP process adopts a single-threaded event-driven programming model. An application such as a routing protocol, where events affecting common data come from many sources simultaneously, would likely have high locking overhead; but, more importantly, our experience is that it is very hard for new programmers to understand a multi-threaded design to the point of being able to extend it safely. Of course, threaded programs could integrate with XORP via IPC.

The core of XORP's event-driven programming model is a traditional `select`-based *event loop* based on the SFS toolkit [19]. Events are generated by timers and file descriptors; callbacks are dispatched whenever an event occurs. Callbacks are type-safe C++ functors, and allow for the currying of additional arguments at creation time.

When an event occurs, we attempt to process that event to completion, including figuring out all inter-process dependencies. For example, a RIP route may be used to resolve the nexthop in a BGP route; so a RIP route change must immediately notify BGP, which must then figure out all the BGP routes that might change as a result. Calculating these dependencies quickly and efficiently is difficult, introducing strong pressure toward a periodic route scanner design. Unfortunately, periodic scanning introduces variable latency and can lead to increased load bursts, which can affect forwarding performance. Since low-delay route convergence is becoming critical to ISPs, we believe that future routing implementations must be event-driven.

Even in an event-driven router, some tasks cannot be processed to completion in one step. For example, a router with a full BGP table may receive well over 100,000 routes from a single peer. If that peering goes down, all these routes need to be withdrawn from all other peers. This can't happen instantaneously, but a flapping peer should not prevent or unduly delay the processing of BGP updates from other peers. Therefore, XORP supports background tasks, implemented using our timer handler, which run only when no events are being processed. These background tasks are essentially cooperative threads: they divide processing up into small slices, and voluntarily return execution to the process's main event loop from time to time until they complete.

We intend for XORP to run on almost any modern operating system. We initially provide support, including FEA support, for FreeBSD and Linux, and for FreeBSD and Linux running Click as a forwarding path. Windows support is under development.

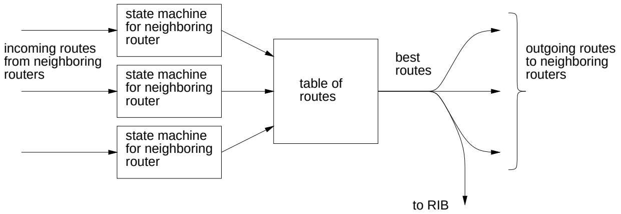

state machine for neighboring router
incoming routes from neighboring routers
state machine for neighboring router
state machine for neighboring router
table of routes
best routes
outgoing routes to neighboring routers
to RIB

FIGURE 2—Abstract routing protocol

5 ROUTING TABLE STAGES

From the general process structure of the XORP control plane, we now turn to modularity and extensibility *within* single processes, and particularly to the ways we divide routing table processing into *stages* in BGP and the RIB. This modularization makes route dataflow transparent, simplifies the implementation of individual stages, clarifies overall organization and protocol interdependencies, and facilitates extension.

At a very high level, the abstract model in Figure 2 can represent routing protocols such as RIP or BGP. (Link-state protocols differ slightly since they distribute *all* routing information to their neighbors, rather than just the best routes.) Note that packet formats and state machines are largely separate from route processing, and that all the real magic—route selection, policy filtering, and so forth—happens within the table of routes. Thus, from a software structuring point of view, the interesting part is the table of routes.

Unfortunately, BGP and other modern routing protocols are big and complicated, with many extensions and features, and it is very hard to understand all the interactions, timing relationships, locking, and interdependencies that they impose on the route table. For instance, as we mentioned, BGP relies on information from intra-domain routing protocols (IGPs) to decide whether the nexthop in a BGP route is actually reachable and what the metric is to that nexthop router. Despite these dependencies, BGP must scale well to large numbers of routes and large numbers of peers. Thus, typical router implementations put all routes in the same memory space as BGP, so that BGP can directly see all the information relevant to it. BGP then periodically walks this jumbo routing table to figure out which routes win, based on IGP routing information. This structure is illustrated in Figure 3. While we don't *know* how Cisco implements BGP, we can infer from clues from Cisco's command line interface and manuals that it probably works something like this.

Unfortunately, this structure makes it very hard to separate functionality in such a way that future programmers can see how the pieces interact or where it is safe to make changes. Without good structure we believe that it will be impossible for future programmers to extend our

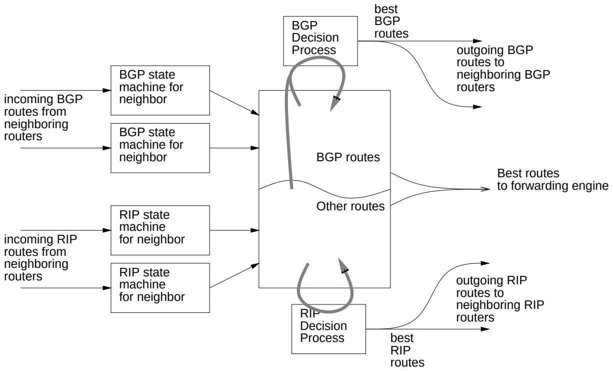

BGP Decision Process
best BGP routes
outgoing BGP routes to neighboring BGP routers
incoming BGP routes from neighboring routers
BGP state machine for neighbor
BGP state machine for neighbor
BGP routes
Other routes
Best routes to forwarding engine
incoming RIP routes from neighboring routers
RIP state machine for neighbor
RIP state machine for neighbor
RIP Decision Process
outgoing RIP routes to neighboring RIP routers
best RIP routes

FIGURE 3—Closely-coupled routing architecture

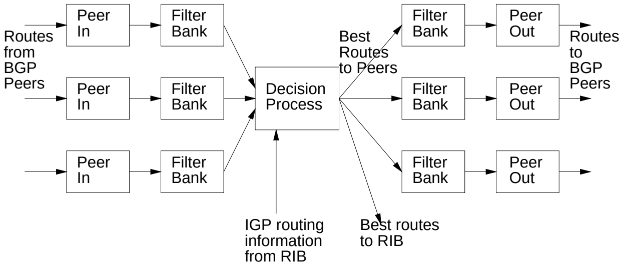


FIGURE 4—Staged BGP architecture

software without compromising its stability.

Our challenge is to implement BGP and the RIB in a more decoupled manner that clarifies the interactions between modules.

5.1 BGP Stages

The mechanism we chose is the clear one of data flow. Rather than a single, shared, passive table that stores information and annotations, we implement routing tables as dynamic processes through which routes flow. There is no single routing table object, but rather a network of pluggable routing *stages*, each implementing the same interface. Together, the network stages combine to implement a routing table abstraction. Although unusual—to our knowledge, XORP is the only router using this design—stages turn out to be a natural model for routing tables. They clarify protocol interactions, simplify the movement of large numbers of routes, allow extension, ease unit testing, and localize complex data structure manipulations to a few objects (namely trees and iterators; see Section 5.3). The cost is a small performance penalty and slightly greater memory usage, due to some duplication between stages. To quantify this, a XORP router holding a full backbone routing table of about 150,000 routes requires about 120 MB for BGP and 60 MB for the RIB, which is simply not a problem on any recent hardware. The rest of this section develops this stage design much as we developed it in practice.

To a first approximation, BGP can be modeled as the pipeline architecture, shown in Figure 4. Routes come in from a specific BGP peer and progress through an incoming filter bank into the decision process. The best routes then proceed down additional pipelines, one for each peering, through an outgoing filter bank and then on to the relevant peer router. Each stage in the pipeline receives routes from upstream and passes them downstream, sometimes modifying or filtering them along the way. Thus, stages have essentially the same API, and are indifferent to their surroundings: new stages can be added to the pipeline without disturbing their neighbors, and their interactions with the rest of BGP are constrained by the stage API.

The next issue to resolve is where the routes are actually stored. When a new route to a destination arrives, BGP must compare it against *all* alternative routes to that destination (not just the previous winner), which dictates that all alternative routes need to be stored. The natural place might seem to be the Decision Process stage; but this would complicate the implementation of filter banks: Filters can be changed by the user, after which we need to re-run the filters and re-evaluate which route won. Thus, we only store the original versions of routes, in the Peer In stages. This in turn means that the Decision Process must be able to look up alternative routes via calls *upstream* through the pipeline.

The basic interface for a stage is therefore:

- add_route: A preceding stage is sending a new route to this stage. Typically the route will be dropped, modified, or passed downstream to the next stage unchanged.
- delete_route: A preceding stage is sending a delete message for an old route to this stage. The deletion should be dropped, modified, or passed downstream to the next stage unchanged.
- lookup_route: A later stage is asking this stage to look up a route for a destination subnet. If the stage cannot answer the request itself, it should pass the request upstream to the preceding stage.

These messages can pass up and down the pipeline, with the constraint that messages must be consistent. There are two consistency rules: (1) Any delete_route message must correspond to a previous add_route message; and (2) the result of a lookup_route should be consistent with previous add_route and delete_route messages sent downstream. These rules lessen the stage implementation burden. A stage can assume that upstream stages are consistent, and need only *preserve* consistency for downstream stages.

For extra protection, a BGP pipeline could include stages that *enforced* consistency around possibly-erroneous experimental extensions, but so far we have not needed to do this. Instead, we have developed an extra consistency checking stage for debugging purposes. This cache stage, just after the outgoing filter bank in the output pipeline to each peer, has helped us discover many subtle bugs that would otherwise have gone undetected. While not intended for normal production use, this stage could aid with debugging if a consistency error is suspected.

5.1.1 Decomposing the Decision Process

The Decision Process in this pipeline is rather complex: in addition to deciding which route wins, it must get nexthop resolvability and metric information from the RIB, and fan out routing information to the output peer pipeline branches and to the RIB. This coupling of functionality is undesirable both because it complicates the stage, and because there are no obvious extension points within such a macro-stage. XORP thus further decomposes the Decision Process into Nexthop Resolvers, a simple Decision Process, and a Fanout Queue, as shown in Figure 5.

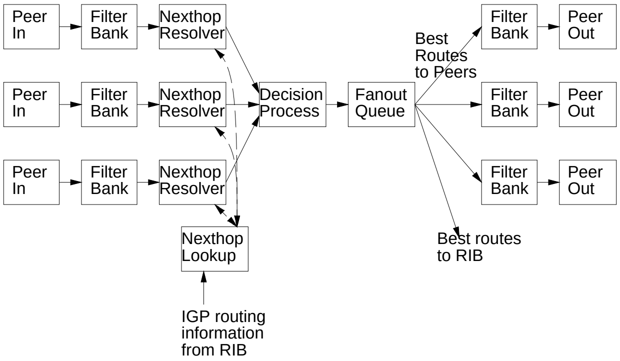


FIGURE 5—Revised staged BGP architecture

The Fanout Queue, which duplicates routes for each peer and for the RIB, is in practice complicated by the need to send routes to slow peers. Routes can be received from one peer faster than we can transit them via BGP to other peers. If we queued updates in the *n* Peer Out stages, we could potentially require a large amount of memory for all *n* queues. Since the outgoing filter banks modify routes in different ways for different peers, the best place to queue changes is in the fanout stage, after the routes have been chosen but before they have been specialized. The Fanout Queue module then maintains a single route change queue, with *n* readers (one for each peer) referencing it.

The Nexthop Resolver stages talk asynchronously to the RIB to discover metrics to the nexthops in BGP's routes. As replies arrive, it annotates routes in add_route and lookup_route messages with the relevant IGP metrics. Routes are held in a queue until the relevant nexthop metrics are received; this avoids the need for the Decision Process to wait on asynchronous operations.

5.1.2 Dynamic Stages

The BGP process's stages are *dynamic*, not static; new stages can be added and removed as the router runs. We made use of this capability in a surprising way when we needed to deal with route deletions due to peer failure. When a peering goes down, all the routes received by this peer must be deleted. However, the deletion of more than 100,000 routes takes too long to be done in a single event handler. This needs to be divided up into slices of work, and handled as a background task. But this leads to a further problem: a peering can come up and go down in rapid succession, before the previous background task has completed.

To solve this problem, when a peering goes down we create a new dynamic *deletion stage*, and plumb it in directly after the Peer In stage (Figure 6).

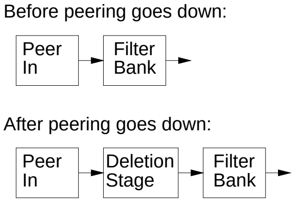


FIGURE 6—Dynamic deletion stages in BGP

The route table from the Peer In is handed to the deletion stage, and a new, empty route table is created in the Peer In. The deletion stage ensures consistency while gradually deleting all the old routes in the background; simultaneously, the Peer In—and thus BGP as a whole—is immediately ready for the peering to come back up. The Peer In doesn't know or care if background deletion is taking place downstream. Of course, the deletion stage must still ensure consistency, so if it receives an add_route message from the Peer In that refers to a prefix that it holds but has not yet got around to deleting, then first it sends a delete_route downstream for the old route, and then it sends the add_route for the new route. This has the nice side effect of ensuring that if the peering flaps many times in rapid succession, each route is held in at most one deletion stage. Similarly, routes not yet deleted will still be returned by lookup_route until after the deletion stage has sent a delete_route message downstream. In this way none of the downstream stages even know that a background deletion process is occurring—all they see are consistent messages. Even the deletion stage has no knowledge of other deletion stages; if the peering bounces multiple times, multiple dynamic deletion stages will be added, one for each time the peering goes down. They will unplumb and delete themselves when their tasks are complete.

We use the ability to add dynamic stages for many background tasks, such as when routing policy filters are changed by the operator and many routes need to be refiltered and reevaluated. The staged routing table design supported late addition of this kind of complex functionality with minimal impact on other code.

5.2 RIB Stages

Other XORP routing processes also use variants of this staged design. For example, Figure 7 shows the basic structure of the XORP RIB process. Routes come into the RIB from multiple routing protocols, which play a similar role to BGP's peers. When multiple routes are available to the same destination from different protocols, the RIB must decide which one to use for forwarding. As with BGP, routes are stored only in the origin stages, and similar add_route, delete_route and lookup_route messages traverse between the stages.

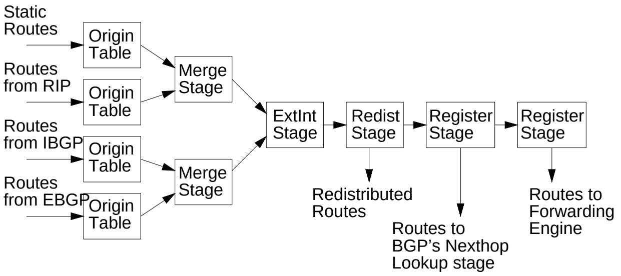


FIGURE 7—Staged RIB architecture

Unlike with BGP, the decision process in the RIB is distributed as pairwise decisions between Merge Stages, which combine route tables with conflicts based on a preference order, and an ExtInt Stage, which composes a set of external routes with a set of internal routes. In BGP, the decision stage needs to see all possible alternatives to make its choice; the RIB, in contrast, makes its decision purely on the basis of a single *administrative distance* metric. This single metric allows more distributed decision-making, which we prefer, since it better supports future extensions.

Dynamic stages are inserted as different watchers register themselves with the RIB. These include Redist Stages, which contain programmable policy filters to redistribute a route subset to a routing protocol, and Register Stages, which redistribute routes depending on prefix matches. This latter process, however, is slightly more complex than it might first appear.

5.2.1 *Registering Interest in RIB Routes*

A number of core XORP processes need to be able to track changes in routing in the RIB as they occur. For example, BGP needs to monitor routing changes that affect IP addresses listed as the nexthop router in BGP routes, and PIM-SM needs to monitor routing changes that affect routes to multicast source addresses and PIM Rendezvous-Point routers. We expect the same to be true of future extensions. This volume of registrations puts pressure on the Register Stage interface used to register and call callbacks on the RIB. In monolithic or shared-memory designs centered around a single routing table structure, a router could efficiently monitor the structure for changes, but such a design cannot be used by XORP. We need to share the minimum amount of information between the RIB and its clients, while simultaneously minimizing the number of requests handled by the RIB.

What BGP and PIM want to know about is the routing for specific IP addresses. But this list of addresses may be moderately large, and many addresses may be routed as part of the same subnet. Thus when BGP asks the RIB about a specific address, the RIB informs BGP about the address range for which the same answer applies.

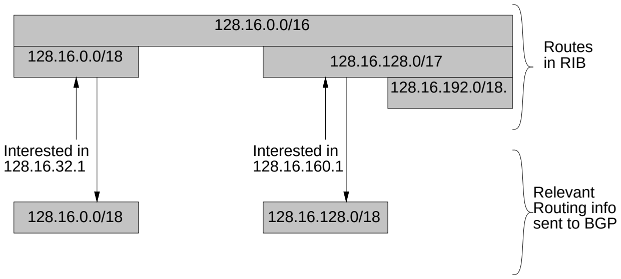


FIGURE 8—RIB interest registration

Figure 8 illustrates this process. The RIB holds routes for 128.16.0.0/16, 128.16.0.0/18, 128.16.128.0/17 and 128.16.192.0/18. If BGP asks the RIB about address 128. 16.32.1, the RIB tells BGP that the matching route is 128. 16.0.0/18, together with the relevant metric and nexthop router information. This address also matched 128.16.0. 0/16, but only the more specific route is reported. If BGP later becomes interested in address 128.16.32.7, it does not need to ask the RIB because it already knows this address is also covered by 128.16.0.0/18.

However, if BGP asks the RIB about address 128.16. 160.1, the answer is more complicated. The most specific matching route is 128.16.128.0/17, and indeed the RIB tells BGP this. But 128.16.128.0/17 is overlayed by 128.16.192.0/18, so if BGP only knew about 128.16. 128.0/17 and later became interested in 128.16.192.1, it would erroneously conclude that this is also covered by 128.16.128.0/17. Instead, the RIB computes the largest enclosing subnet that is not overlayed by a more specific route (in this case 128.16.128.0/18) and tells BGP that its answer is valid for this subset of addresses only. Should the situation change at any later stage, the RIB will send a "cache invalidated" message for the relevant subnet, and BGP can re-query the RIB to update the relevant part of its cache.

Since no largest enclosing subnet ever overlaps any other in the cached data, RIB clients like BGP can use balanced trees for fast route lookup, with attendant performance advantages.

5.3 Safe Route Iterators

Each background stage responsible for processing a large routing table, such as a BGP deletion stage, must remember its location in the relevant routing table so that it can make forward progress on each rescheduling. The XORP library includes route table *iterator* data structures that implement this functionality (as well as a Patricia Tree implementation for the routing tables themselves). Unfortunately, a route change may occur while a background task is paused, resulting in the tree node pointed to by an iterator being deleted. This would cause the iterator to hold invalid state. To avoid this problem, we use some spare bits in each route tree node to hold a reference count of the number of iterators currently pointing at this tree node. If the route tree receives a request to delete a node, the node's data is invalidated, but the node itself is not removed immediately unless the reference count is zero. It is the responsibility of the last iterator leaving a previously-deleted node to actually perform the deletion.

The internals of the implementation of route trees and iterators are not visible to the programmer using them. All the programmer needs to know is that the iterator will never become invalid while the background task is paused, reducing the feature interaction problem between background tasks and event handling tasks.

6 INTER-PROCESS COMMUNICATION

Using multiple processes provides a solid basis for resource management and fault isolation, but requires the use of an inter-process communication (IPC) mechanism. Our IPC requirements were:

- to allow communication both between XORP processes and with routing applications not built using the XORP framework;
- to use multiple transports transparently, including intra-process calls, host-local IPC, and networked communication, to allow a range of tradeoffs between flexibility and performance;
- to support component namespaces for extensibility and component location for flexibility, and to provide security through per-method access control on components;
- to support asynchronous messaging, as this is a natural fit for an event-driven system; and
- to be portable, unencumbered, and lightweight.

During development we discovered an additional requirement, *scriptability*, and added it as a feature. Being able to script IPC calls is an invaluable asset during development and for regression testing. Existing messaging frameworks, such as CORBA [22] and DCOM [5], provided the concepts of components, component addressing and location, and varying degrees of support for alternative transports, but fell short elsewhere.

We therefore developed our own *XORP IPC* mechanism. The *Finder* process locates components and their methods; communication proceeds via a naturally scriptable base called *XORP Resource Locators*, or *XRLs*.

6.1 XORP Resource Locators

An XRL is essentially a method supported by a component. (Because of code reuse and modularity, most processes contain more than one component, and some components may be common to more than one process; so the unit of IPC addressing is the *component instance* rather than the process.) Each component implements an XRL *interface*, or group of related methods. When one component wishes to communicate with another, it composes an XRL and *dispatches* it. Initially a component knows only the generic component name, such as "bgp", with which it wishes to communicate. The Finder must resolve such generic XRLs into a form that specifies precisely how communication should occur. The resulting resolved XRL specifies the transport *protocol family* to be used, such as TCP, and any parameters needed for communication, such as hostname and port.

The canonical form of an XRL is textual and human-readable, and closely resembles Uniform Resource Locators (URLs [1]) from the Web. Internally XRLs are encoded more efficiently, but the textual form permits XRLs to be called from any scripting language via a simple *call_xrl* program. This is put to frequent use in all our scripts for automated testing. In textual form, a generic XRL might look like:

```
finder://bgp/bgp/1.0/set_local_as?as:u32=1777
```

And after Finder resolution:

```
stcp://192.1.2.3:16878/bgp/1.0/set_local_as?as:u32=1777
```

XRL arguments (such as "as" above, which is an Autonomous System number) are restricted to a set of core types used throughout XORP, including network addresses, numbers, strings, booleans, binary arrays, and lists of these primitives. Perhaps because our application domain is highly specialized, we have not yet needed support for more structured arguments.

As with many other IPC mechanisms, we have an interface definition language (IDL) that supports interface specification, automatic stub code generation, and basic error checking.

6.2 Components and the Finder

When a component is created within a process, it instantiates a receiving point for the relevant XRL protocol families, and then registers this with the Finder. The registration includes a component class, such as “bgp”; a unique component instance name; and whether or not the caller expects to be the sole instance of a particular component class. Also registered are each interface’s supported methods and each method’s supported protocol families. This allows for specialization; for example, one protocol family may be particularly optimal for a particular method.

When a component wants to dispatch an XRL, it consults the Finder for the resolved form of the XRL. In reply, it receives the resolved method name together with a list of the available protocol families and arguments to bind the protocol family to the receiver. For a networked protocol family, these would typically include the hostname, receiving port, and potentially a key. Once resolved, the dispatcher is able to instantiate a sender for the XRL and request its dispatch. XRL resolution results are cached, and these caches are updated by the Finder when entries become invalidated.

In addition to providing resolution services, the Finder also provides a component lifetime notification service. Components can request to be notified when another component class or instance starts or stops. This mechanism is used to detect component failures and component restarts.

6.3 Protocol Families

Protocol families are the mechanisms by which XRLs are transported from one component to another. Each protocol family is responsible for providing argument marshaling and unmarshaling facilities as well as the IPC mechanism itself.

Protocol family programming interfaces are small and simple to implement. In the present system, there are three protocol families for communicating between XORP components: *TCP*, *UDP*, and *intra-process*, which is for calls between components in the same process. There is also a special *Finder protocol family* permitting the Finder to be addressable through XRLs, just as any other XORP component. Finally, there exists a *kill* protocol family, which is capable of sending just one message type—a UNIX signal—to components within a host. We expect to write further specialized protocol families for communicating with non-XORP components. These will effectively act as proxies between XORP and unmodified XORP processes.

7 SECURITY FRAMEWORK

Security is a critical aspect of building a viable extensible platform. Ideally, an experimental protocol running on a XORP router could do no damage to that router, whether through poor coding or malice. We have not yet reached this ideal; this section describes how close we are.

Memory protection is of course the first step, and XORP’s multi-process architecture provides this. The next step is to allow processes to be sandboxed, so they cannot access important parts of the router filesystem. XORP centralizes all configuration information in the Router Manager, so no XORP process needs to access the filesystem to load or save its configuration.

Sandboxing has limited use if a process needs to have root access to perform privileged network operations. To avoid this need for root access, the FEA is used as a relay for all network access. For example, rather than sending UDP packets directly, RIP sends and receives packets using XRL calls to the FEA. This adds a small cost to networked communication, but as routing protocols are rarely high-bandwidth, this is not a problem in practice.

This leaves XRLs as the remaining vector for damage. If a process could call any other XRL on any other process, this would be a serious problem. By default we don’t accept XRLs remotely. To prevent local circumvention, at component registration time the Finder includes a 16-byte random key in the registered method name of all resolved XRLs. This prevents a process bypassing the use of the Finder for the initial XRL resolution phase, because the receiving process will reject XRLs that don’t match the registered method name.

We have several plans for extending XORP’s security. First, the Router Manager will pass a unique secret to each process it starts. The process will then use this secret when it resolves an XRL with the Finder. The Finder is configured with a set of XRLs that each process is allowed to call, and a set of targets that each process is allowed to communicate with. Only these permitted XRLs will be resolved; the random XRL key prevents bypassing the Finder. Thus, the damage that can be done by an errant process is limited to what can be done through its normal XRL calls. We can envisage taking this approach even further, and restricting the range of arguments that a process can use for a particular XRL method. This would require an XRL intermediary, but the flexibility of our XRL resolution mechanism makes installing such an XRL proxy rather simple. Finally, we are investigating the possibility of running different routing processes in different virtual machines under the Xen [11] virtual machine monitor, which would provide even better isolation and allow us to control even the CPU utilization of an errant process.

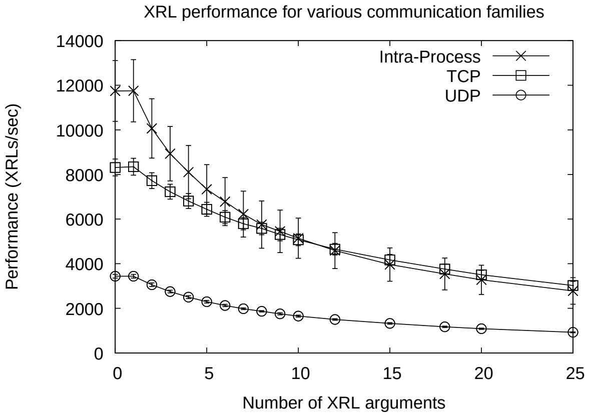


FIGURE 9—XRL performance results

8 EVALUATION

As we have discussed, the XORP design is modular, robust and extensible, but these properties will come at some cost in performance compared to more tightly coupled designs. The obvious concern is that XORP might not perform well enough for real-world use. On previous generations of hardware, this might have been true, but we will show below that it is no longer the case.

The measurements are performed on a relatively low-end PC (AMD Athlon 1100MHz) running FreeBSD 4.10. At this stage of development we have put very little effort into optimizing the code for performance, but we have paid close attention to the computation complexity of our algorithms. Nevertheless, as we show below, even without optimization the results clearly demonstrate good performance, and the advantage of our event-driven design.

8.1 XRL Performance Evaluation

One concern is that the XRL IPC mechanism might become a bottleneck in the system. To verify that it is not, the metric we are interested in is the throughput we can achieve in terms of number of XRL calls per second.

To measure the XRL rate, we send a transaction of 10000 XRLs using a pipeline size of 100 XRLs. Initially, the sender sends 100 XRLs back-to-back, and then for every XRL response received it sends a new request. The receiver measures the time between the beginning and the end of a transaction. We evaluate three communication transport mechanisms: TCP, UDP and Intra-Process direct calling where the XRL library invokes direct method calls between a sender and receiver inside the same process.[1]

In Figure 9 we show the average XRL rate and its standard deviation for TCP, UDP and Intra-Process transport mechanisms when we vary the number of arguments to the XRL. These results show that our IPC mechanism can easily sustain several thousands of XRLs per second on a relatively low-end PC. Not surprisingly, for a small number of XRL arguments, the Intra-Process performance is best (almost 12000 XRLs/second), but for a larger number of arguments the difference between Intra-Process and TCP disappears. It is clear from these results that our argument marshalling and unmarshalling is not terribly optimal, but despite this the results are quite respectable. In practice, most commonly used XRLs have few arguments. This result is very encouraging, because it demonstrates that typically the bottleneck in the system will be elsewhere.

The UDP performance is significantly worse because UDP was our first prototype XRL implementation, and does not pipeline requests. For normal usage, XORP currently uses TCP and does pipeline requests. UDP is included here primarily to illustrate the effect of request pipelining, even when operating locally.

8.2 Event-Driven Design Evaluation

To demonstrate the scaling properties of our event-driven design, we present some BGP-related measurements. Routing processes not under test such as PIM-SM and RIP were also running during the measurements, so the measurements represent a fairly typical real-world configuration.

First, we perform some measurements with an empty routing table, and then with a routing table containing a full Internet backbone routing feed consisting of 146515 routes. The key metric we care about is how long it takes for a route newly received by BGP to be installed into the forwarding engine.

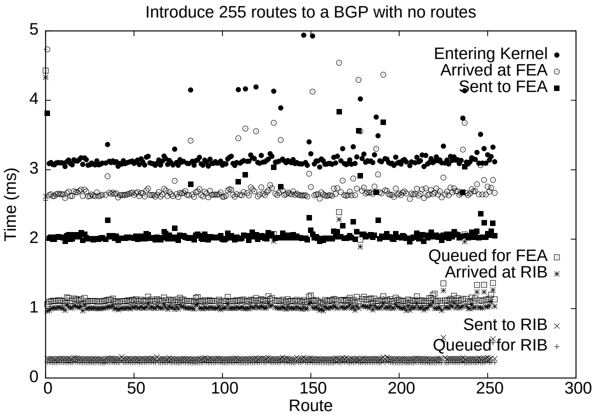
XORP contains a simple profiling mechanism which permits the insertion of profiling points anywhere in the code. Each profiling point is associated with a profiling variable, and these variables are configured by an external program *xorp_profiler* using XRLs. Enabling a profiling point causes a time stamped record to be stored, such as:

```
route_ribin 1097173928 664085 add 10.0.1.0/24
```

In this example we have recorded the time in seconds and microseconds at which the route “10.0.1.0/24” has been added. When this particular profiling variable is enabled, all routes that pass this point in the pipeline are logged.

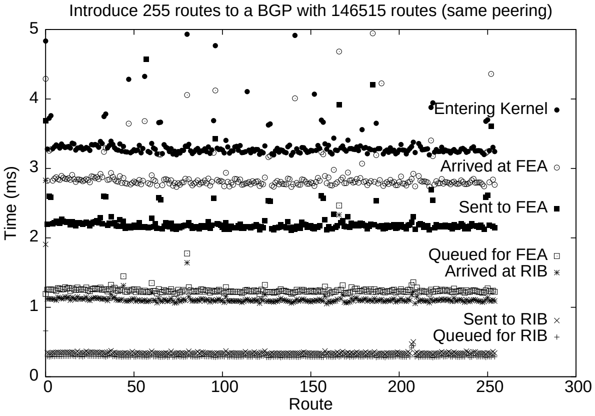
If a route received by BGP wins the decision process, it will be sent to its peers and to the RIB (see Figure 1). When the route reaches the RIB, if it wins against routes from other protocols, then it is sent to the FEA. Finally,

[1] To allow direct comparison of Intra-Process against TCP and UDP, both sender and receiver are running within the same process. When we run the sender and receiver on two separate processes on the same host, the performance is very slightly worse.



Profile Point	Avg	SD	Min	Max
Entering BGP	-	-	-	-
Queued for transmission to the RIB	0.226	0.026	0.214	0.525
Sent to RIB	0.280	0.026	0.267	0.582
Arriving at the RIB	1.036	0.132	0.964	2.285
Queued for transmission to the FEA	1.139	0.132	1.066	2.389
Sent to the FEA	2.234	1.903	1.965	31.663
Arriving at FEA	2.876	1.918	2.580	32.443
Entering kernel	3.374	1.980	3.038	33.576

FIGURE 10—Route propagation latency (in ms), no initial routes



Profile Point	Avg	SD	Min	Max
Entering BGP	-	-	-	-
Queued for transmission to the RIB	0.289	0.015	0.275	0.447
Sent to RIB	0.344	0.015	0.329	0.502
Arriving at the RIB	1.112	0.088	1.052	2.331
Queued for transmission to the FEA	1.247	0.088	1.187	2.465
Sent to the FEA	2.362	1.571	2.108	25.910
Arriving at FEA	3.019	1.590	2.734	26.689
Entering kernel	3.632	3.394	3.191	56.422

FIGURE 11—Route propagation latency (in ms), 146515 initial routes and same peering

the FEA will unconditionally install the route in the kernel or the forwarding engine.

The following profiling points were used to measure the flow of routes:

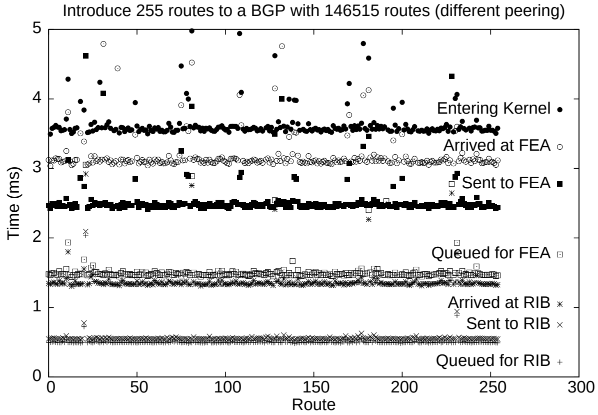
1. Entering BGP
2. Queued for transmission to the RIB
3. Sent to the RIB
4. Arriving at the RIB
5. Queued for transmission to the FEA
6. Sent to the FEA
7. Arriving at the FEA
8. Entering the kernel

One of the goals of this experiment is to demonstrate that routes introduced into a system with an empty routing table perform similarly to a system with a full BGP backbone feed of 146515 routes. In each test we introduce a new route every two seconds, wait a second, and then remove the route. The BGP protocol requires that the next hop is resolvable for a route to be used. BGP discovers if a next hop is resolvable by registering interest with the RIB. To avoid unfairly penalizing the empty routing table tests, we keep one route installed during the test to prevent additional interactions with the RIB that typically would not happen with the full routing table.

The results are shown in Figures 10–12. In the first experiment (Figure 10) BGP contained no routes other than the test route being added and deleted. In the second experiment (Figure 11) BGP contained 146515 routes and the test routes were introduced on the same peering from which the other routes were received. In the third experiment (Figure 12) BGP contained 146515 routes and the test routes were introduced on a different peering from which the other routes were received, which exercises different code-paths from the second experiment.

All the graphs have been cropped to show the most interesting region. At the tables indicate, one or two routes took as much as 90ms to reach the kernel. This appears to be due to scheduling artifacts, as FreeBSD is not a realtime operating system.

The conclusion to be drawn from these graphs is that



Profile Point	Avg	SD	Min	Max
Entering BGP	-	-	-	-
Queued for transmission to the RIB	0.508	0.100	0.483	2.039
Sent to RIB	0.563	0.101	0.536	2.094
Arriving at the RIB	1.377	0.182	1.306	2.920
Queued for transmission to the FEA	1.517	0.193	1.438	3.053
Sent to the FEA	3.149	5.943	2.419	92.391
Arriving at FEA	3.822	5.964	3.037	93.179
Entering kernel	4.417	6.278	3.494	93.662

FIGURE 12—Route propagation latency (in ms), 146515 initial routes and different peering

routing events progress to the kernel very quickly (typically within 4ms of receipt by BGP). Perhaps as importantly, the data structures we use have good performance under heavy load, therefore the latency does not significantly degrade when the router has a full routing table. The latency is mostly dominated by the delays inherent in the context switch that is necessitated by inter-process communication. We should emphasize that the XRL interface is pipelined, so performance is still good when many routes change in a short time interval.

We have argued that an event driven route processing model leads to faster convergence than the traditional route scanning approach. To verify this assertion we performed a simple experiment, shown in Figure 13. We introduced 255 routes from one BGP peer at one second intervals and recorded the time that the route appeared at another BGP peer. The experiment was performed on XORP, Cisco-4500 (IOS Version 12.1), Quagga-0.96.5, and MRTD-2.2.2a routers. It should be noted that the granularity of the measurement timer was one second.

This experiment clearly shows the consistent behavior achieved by XORP, where the delay never exceeds one second. MRTD's behavior is very similar, which is important because it illustrates that the multi-process architecture used by XORP delivers similar performance to a closely-coupled single-process architecture. The Cisco and Quagga routers exhibit the obvious symptoms of a 30-second route scanner, where all the routes received in the previous 30 seconds are processed in one batch. Fast convergence is simply not possible with such a scanner-based approach.

Teixeira et al demonstrate [27] that even route changes within an AS can be adversely affected by the delay introduced by BGP route scanners. In real ISP networks, the found delays of one to two *minutes* were common between an IGP route to a domain border router changing, and the inter-domain traffic flowing out of a domain changing its exit router. During this delay, they show that transient forwarding loops can exist, or traffic may be blackholed, both of which may have significant impact on customers. Thus fast convergence is clearly of high importance to providers, and can only become more so with the increase in prominence of real-time traffic.

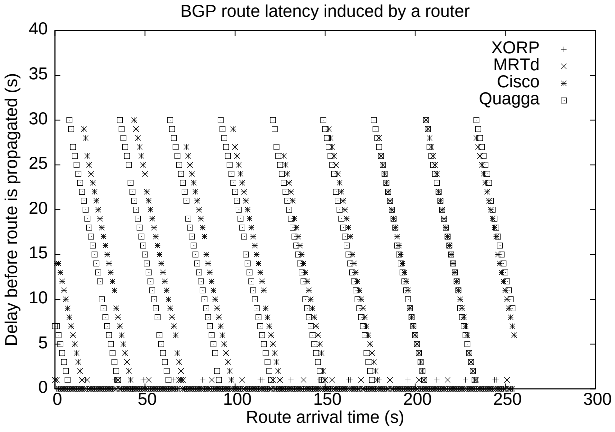


FIGURE 13—BGP route flow

8.3 Extensibility Evaluation

The hardest part of our design to properly evaluate is its extensibility. Only time will tell if we really have the right modularity, flexibility, and APIs. However, we can offer a number of examples to date where extensibility has been tested.

Adding Policy to BGP

We implemented the core BGP and RIB functionality first, and only then thought about how to configure policy, which is a large part of any router functionality. Our policy framework consists of three new BGP stages and two new RIB stages, each of which supports a common simple stack language for operating on routes. The details are too lengthy for this paper, but we believe this framework allows us to implement almost the full range of policies available on commercial routers.

The only change required to pre-existing code was the addition of a tag list to routes passed from BGP to the RIB and vice versa. Thus, our staged architecture appears to have greatly eased the addition of code that is notoriously complex in commercial vendors' products.

What we got wrong was the syntax of the command line interface (CLI) template files, described in [13], used to dynamically extend the CLI configuration language. Our original syntax was not flexible enough to allow user-friendly specification of the range of policies that we need to support. This is currently being extended.

Adding Route Flap Damping to BGP

Route flap damping was also not a part of our original BGP design. We are currently adding this functionality (ISPs demand it, even though it's a flawed mechanism), and can do so efficiently and simply by adding another stage to the BGP pipeline. The code does not impact other stages, which need not be aware that damping is occurring.

Adding a New Routing Protocol

XORP has now been used as the basis for routing research in a number of labs. One university unrelated to our group used XORP to implement an ad-hoc wireless routing protocol. In practice XORP probably did not help this team greatly, as they didn't need any of our existing routing protocols, but they did successfully implement their protocol. Their implementation required a single change to our internal APIs to allow a route to be specified by interface rather than by nexthop router, as there is no IP subnetting in an ad-hoc network.

9 Conclusions

We believe that innovation in the core protocols supporting the Internet is being seriously inhibited by the nature of the router software market. Furthermore, little long term research is being done, in part because researchers perceive insurmountable obstacles to experimentation and deployment of their ideas.

In an attempt to change the router software landscape, we have built an extensible open router software platform. We have a stable core base running, consisting of around 500,000 lines of C++ code. XORP is event-driven, giving fast routing convergence, and incorporates a multi-process design and novel inter-process communication mechanisms that aid extensibility, and allow experimental software to run alongside production software.

In this paper we have presented a range of innovative features, including a novel staged design for core protocols, and a strong internal security architecture geared around the sandboxing of untrusted components. We also presented preliminary evaluation results that confirm that our design scales well to large routing tables while maintaining low routing latency.

In the next phase we need to involve the academic community, both as early adopters, and to flesh out the long list of desirable functionality that we do not yet support. If we are successful, XORP will become a true *production-quality* platform. The road ahead will not be easy, but unless this or some other approach to enable Internet innovation is successful, the long-run consequences of ossification will be serious indeed.

Acknowledgments

We thank the other members of the XORP project, including Fred Bauer (Linux), Andrea Bittau (routing policy), Javier Cardona (SNMP support), Adam Greenhalgh (initial BGP implementation), Luigi Rizzo (fast forwarding), Bruce M. Simpson (Windows), and Marko Zec (Click integration and fast lookups). We gratefully acknowledge those who financially supported XORP, including the ICSI Center for Internet Research, Intel Corporation, the National Science Foundation (ANI-0129541), and Microsoft Corporation. We also sincerely thank Scott Shenker for believing in the work against all odds.

Notes

[1] We have some confirmation of this: a group implementing an ad-hoc routing protocol found that XORP's RIB supported their application with just one trivial interface change [17].

References

[1] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators. RFC 1738, Internet Engineering Task Force, December 1994.

[2] Bird project. The BIRD Internet Routing Daemon (Web site). http://bird.network.cz/.

[3] V. J. Bono. 7007 explanation and apology, 1997. http://www.merit.edu/mail.archives/nanog/1997-04/msg00444.html.

[4] Robert Braden, Ted Faber, and Mark Handley. From protocol stack to protocol heap—Role-based architecture. In *Proc. 1st Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[5] Nat Brown and Charlie Kindel. Distributed Component Object Model Protocol – DCOM 1.0. Online, November 1998. Expired IETF draft draft-brown-dcom-v1-spec-02.txt.

[6] Kenneth L. Calvert, James Griffioen, and Su Wen. Lightweight network support for scalable end-to-end services. In *Proc. ACM SIGCOMM 2002 Conference*, pages 265–278, August 2002.

[7] Cisco Systems. Cisco IOS software. http://www.cisco.com/public/sw-center/sw-ios.shtml.

[8] Cisco Systems. Cisco IOS XR software. http://www.cisco.com/en/US/products/ps5845/index.html.

[9] National Research Council. *Looking Over the Fence at Networks*. National Academy Press, 2001.

[10] Christophe Diot, Brian Neil Levine, Bryan Lyles, Hassan Kassem, and Doug Balensiefen. Deployment issues for the ip multicast service and architecture. *IEEE Network*, January/February 2000.

[11] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proc. 18th ACM Symposium on Operating Systems Principles*, October 2003.

[12] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol Independent Multicast—Sparse Mode (PIM-SM): Protocol specification. RFC 2362, Internet Engineering Task Force, June 1998. ftp://ftp.ietf.org/rfc/rfc2362.txt.

[13] M. Handley, O. Hodson, and E. Kohler. XORP: An open platform for network research. In *Proc. 1st Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[14] A. Helmy. Protocol indendent multicast-sparse mode (pim-sm) implementation document, 1996. http://netweb.usc.edu/pim/pimd/docs/.

[15] Juniper Networks. JunOS software. http://www.juniper.net/products/junos/.

[16] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Trans. on Computer Systems*, 18(3):263–297, August 2000.

[17] Thomas Kunz. Implementing bcast (implementation report). http://www.sce.carleton.ca/wmc/code.html, March 2004.

[18] Zhuoqing Morley Mao, Ramesh Govindan, George Varghese, and Randy H. Katz. Route flap damping exacerbates Internet routing convergence. In *Proc. ACM SIGCOMM 2002 Conference*, August 2002.

[19] David Mazières. A toolkit for user-level file systems. In *Proc. USENIX 2001 Annual Technical Conference*, pages 261–274, June 2001.

[20] John Moy. *OSPF Complete Implementation*. Addison-Wesley, December 2000.

[21] NextHop Technologies. GateD releases (Web site). http://www.gated.org/.

[22] Object Management Group. Common Object Request Broker Architecture Specification 3.0.3, March 2004. http://www.omg.org/cgi-bin/doc?formal/04-03-12.

[23] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. 1st Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[24] Larry Peterson, Scott Shenker, and Jonathan Turner. Overcoming the internet impasse through virtualization. In *Proc. 3rd Workshop on Hot Topics in Networks (HotNets-III)*, November 2004.

[25] Quagga project. Quagga Routing Suite (Web site). http://www.quagga.net/.

[26] Tammo Spalink, Scott Karlin, Larry Peterson, and Yitzchak Gottlieb. Building a robust software-based router using network processors. In *Proc. 18th ACM Symposium on Operating Systems Principles*, pages 216–229, October 2001.

[27] R. Teixeira, A. Shaikh, T. Griffin, and J. Rexford. Dynamics of hot-potato routing in ip networks. In *Proc. 2004 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 2004.

[28] University of Michigan and Merit Network. MRT: Multi-threaded Routing Toolkit (Web site). http://www.mrtd.net/.

[29] Unix Manual Pages. routed - network RIP and router discovery routing daemon.

[30] David Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 64–79, Kiawah Island, South Carolina, December 1999.

[31] Zebra project. GNU Zebra routing software (Web site). http://www.zebra.org/.