Decoupling Policy from Protocols:
Implementation Issues in Extensible IP Router Software

Andrea Bittau, Mark Handley
University College London

Abstract

Policy is a crucial component of today's Internet, allowing ISPs to tune the default behavior of routing protocols to match economic and administrative needs. However, providing powerful and expressive policy controls in router software is not easy and only a few router implementations do this well.

In this paper we describe the implementation of an *extensible* policy framework that can serve both current and future routing protocols. Designing for extensibility makes this hard because policy is both tightly coupled with the details of each specific routing protocol and can also describe subtle interactions *between* different protocols. Such a policy engine must have no in-built protocol-specific knowledge. However, it must also be capable of satisfying the needs of each specific protocol and cope with filters that span more than one protocol.

We present the design of a single generic solution to this problem. Our highly extensible policy framework can support most, if not all, routing protocols with very little work needed on the part of the protocol implementor. We will demonstrate that all this flexibility and generality can be achieved with sufficient performance for even large complex backbone routers.

1 Introduction

Routing policy is of critical importance for anyone running a non-trivial sized network. Policy is the means by which ISPs determine the large-scale flow of traffic and ensure that the traffic roughly follows the flow of money. Without effective policy mechanisms, the Internet is neither technically nor financially viable. However, whereas routing protocols are described in detail in the literature and there is some work on policy specification [1, 3], very little is written on how to *implement* policy.

The context for this paper is the implementation of an extensible Internet router. The goal of the *eXtensible Open Router Platform* (XORP [4]) is to build a complete IP routing suite and management framework, with particular emphasis on design for future extension. For example, it should be simple to write a new routing protocol, and seamlessly integrate it into the existing router framework without modifying any of the existing router code. This is needed so extensions from multiple vendors can co-exist gracefully on the same box.

For extensible routers in general, and for XORP in particular, policy is somewhat problematic. There are three main reasons for this:

- A single policy rule can span multiple routing protocols, even though those protocols have no inbuilt knowledge of each other.
- Within a routing protocol, policy filters are complex and hard to debug, and need to integrate closely with the router's management framework. The learning curve for developers is steep.
- Any policy configuration framework will need to evolve over the years. This is made much harder if it necessitates updating every supported or experimental protocol from every vendor and research lab.

Our goal then is to design and implement a policy routing framework that avoids these problems. It must be generic enough to satisfy all, isolate functionality sufficiently that each protocol can be implemented with no knowledge of others, be easy for developers to understand and use, and be fast enough to work with complex policies in large backbone routers. We believe we have succeeded in building just such a framework. While the details are specific to XORP, we believe the architecture and ideas also have wider applicability.

1.1 What is Policy?

To provide some background, we start with some examples of very simple policies:

- *Do not advertise routes heard from upstream provider X to upstream provider Y.* Do not pay someone money to carry their transit traffic.
- *From customer C, only accept route prefix 1.2.3.0/24.* This prevents a customer's misconfiguration diverting traffic bound for elsewhere.
- *Prepend our AS number 3 times to BGP routes received over customer C's backup connection.* This makes the route less preferred.

A full routing policy is of course more complex than this, but the general idea should be clear. Although the richest policies tend to involve BGP [11] other protocols also need policy. For example, route redistribution between routing protocols is a form of routing policy.

The details of any policy are closely coupled to routing protocols such as BGP, and need to be implemented within those protocols. However, a complete routing policy is broader than this and must be globally coordinated, perhaps spanning multiple different protocols[1]. To understand why this is the case, consider the following (somewhat contrived) policy:

> *Take all routes from OSPF area 10.0.0.1 and redistribute them to BGP, setting a MED of 3 and an Origin of "IGP" on these routes.*

The router operator wants to be able to express such a policy in a single expression—he should be unaware of the internal architecture of the software.

To implement this policy requires knowledge of OSPF [8] (the *area* attribute is OSPF-specific), it requires the ability to re-distribute routes from one protocol to another, and it requires knowledge of BGP (the *MED* and *Origin* attributes are BGP-specific). However, we want BGP and OSPF to know nothing of each other. Furthermore, we want none of the router's core components such as the *Routing Information Base* (RIB) or *Command Line Interface* (CLI) to have inbuilt knowledge of either protocol or any new protocol anyone devises in future.

Thus, routing policy is one place where the goal of extensibility might come directly into conflict with the expressiveness of functionality we wish to support.

2 The Framework

Routing protocols are quite varied in design, so it is a challenge to design a single framework that can support all policies in all protocols without becoming excessively complex. In this section, we will illustrate what a policy does at the most abstract level in an attempt to discover common policy requirements. We will then look at how we can implement these requirements in an *extensible* manner, while also making it as simple as possible for new routing protocols to use this framework.

2.1 Abstract Policy Primitives

Routing policy is actually quite tightly constrained by the routing protocols themselves, so we will consider policy primarily from a mechanistic point of view.
Broadly speaking, the primitives of routing policies are:

- Match certain properties of a route.
- Drop matching routes.
- Modify certain properties of matching routes.
- Redistribute matching routes from one protocol to another.

These primitives can be applied in multiple places in the route processing pipeline, especially when route redistribution is concerned, and so the policies that can be built up are quite complex. There are, however, some limitations:

- Policies are only applied to individual routes. No commercial router performs actions on one route based on the properties of another route.
- Policies are not Turing-complete. No commercial router supports full iterative or recursive filters.

These limitations are probably a good thing, as they prevent the interactions between policies configured on different routers being even more subtle or unpredictable than they are today. We take these limitations as given, although our architecture could in fact support iteration if the benefits turn out to outweigh the dangers in future.

2.2 Import and Export Filters

Using the basic primitives described, a policy defines which routes to filter and how to filter them. However, it is also important *where* filtering is performed. In general, there are two main types of route policy:

1. Import Policies. Filtering will occur on the path into the router from a neighbor.
2. Export Policies. Filtering will occur on the path out of the router to a neighbor.

[1] In XORP, each protocol is in a separate process.

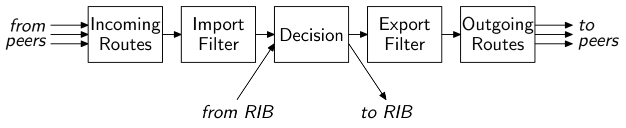

Figure 1: Import/export filters in a vector protocol.

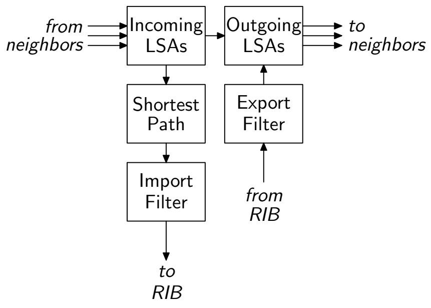

Figure 2: Import/export filters in a link state protocol.

Filtering is used differently with vector protocols (BGP, RIP) than with link-state protocols (OSPF, IS-IS [9]).

For vector protocols (Figure 1) import filtering occurs before the decision of best route is made, as a policy typically forces one route to be preferred over another. For example, an operator might wish all BGP routes learnt from a particular peering to be prioritized, irrespective of the normal BGP cost weighting. Conversely, export filtering occurs after the best route has been decided, as a router advertises the best version of a route to its peers.

For link-state protocols (Figure 2) filtering is more restricted because of the need for all routers to perform the same shortest-path calculation based on the same link-state database. An import filter for a link-state protocol occurs after the shortest-path computation has been performed, but before export to the *Routing Information Base* (RIB). Thus, a link-state import filter does not affect what a router tells its neighbors, but it does change the routes the routing protocol tells the RIB. The main use of such a filter is to weight a route from one routing protocol over one from another protocol. For example, we may prefer an OSPF route to an EBGP route, contrary to normal default priorities. An export filter in a link-state protocol occurs on the output path from the RIB to the protocol's neighbors, affecting only routes the RIB re-distributed to the routing protocol.

Thus, although their purpose may vary, all protocols which support policy will normally have both an import filter bank and an export filter bank.

2.3 Route Re-distribution

Route re-distribution occurs when a route learnt by one protocol is re-advertised via a different protocol. Such re-distribution is intimately tied to routing policy. Consider the following policy:

Take all routes from RIP with a metric of less than 3 and redistribute them to EBGP peer 192.168.1.2 setting a MED of 4.

This becomes a BGP export policy. In XORP syntax:

```
policy redist {
    from {
        protocol: rip
        metric < 3
    }
    to {
        neighbor: 192.168.1.2
    }
    then {
        MED: 4
        accept
    }
}

bgp export redist
```

As this is an export policy, matching and filtering must occur just before BGP advertises the route. Therefore, the neighbor will be matched just as the route is leaving BGP and the MED will then be set.

However, the from clause must be matched in the RIP protocol because the metric attribute in this context is RIP-specific. Within RIP, the matching must occur *post-decision* because only the best RIP routes should be re-advertised, and it must occur prior to the point where routes are sent to the RIB. Thus, in addition to the basic *import* and *export* filter banks shown in Figures 1 and 2, an additional filter bank is needed to select routes for later action.

After a route is matched in its source protocol, it is sent to the RIB where it is compared against all the other routes to the same destination subnet that have been received by other routing protocols. Only the best route is sent to the forwarding engine and similarly only the best route may be re-distributed to other routing protocols.

The RIB does not perform its own matching to decide whether a route should be redistributed, but instead it relies on the matching performed in the original routing protocol. To do this though, a final additional filter bank must exist in the RIB itself which selects these pre-chosen routes for re-distribution and forwards them to the correct protocol.

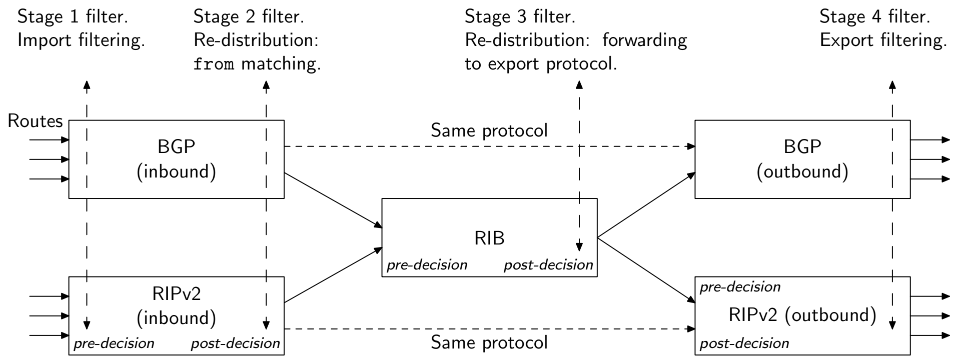

Figure 3: Logical placement of policy filters.

Figure 3 shows the complete policy filtering framework for two vector-based routing protocols. The split of a routing protocol into inbound and outbound is only conceptual—in practice they are in the same process. The only difference for a link-state protocol is that the Stage 1 filter occurs later, just before the Stage 2 filter.

Import policies are implemented entirely in the Stage 1 filter bank. Export filters require a combination of Stages 2, 3 and 4. It should be clear that these export filters do not operate in isolation and that some communication is needed between the filters. We address this next.

2.3.1 Policy Tags

In XORP and similar modern routing implementations, protocols within a router do not share memory. Thus, some other mechanism is needed in order to pass partial-match information from the source protocol to the RIB and on to the destination protocol. To do this, we add meta-data called *policy tags* to routes.

Each policy that performs route re-distribution is assigned a unique policy tag. If a route matches the from clause in the Stage 2 filter of its source protocol, the tag allocated for this policy is assigned to the route. A single route may have multiple tags indicating it matched multiple from clauses in different route re-distribution policies. These policy tags are propagated together with the route everywhere within the router.

After a route is chosen in its source protocol as the best route, it is sent to the RIB. If the RIB also chooses it as the best route over routes from other protocols, and if it matches the Stage 3 filter, then it must be forwarded to the protocol performing the export. A map[2] is held in the RIB which maps policy tags directly to destination protocols. Therefore, this Stage 3 filter is simple and fast.

[2] A C++ STL *map* is a data-structure that does fast key-value lookup.

When matching the to clause in the exporting protocol, the policy tags also are checked against the tag allocated for this policy. This ensures that the route did match the from clause in its source protocol. Finally if all matches succeed, the appropriate actions take place.

For example, the export policy in Section 2.3 would be split into two parts. The first policy fragment would be executed by the RIP protocol's Stage 2 filter:

```
from {
    metric < 3
}
then {
    policytags: 100
}
```

This will perform the matching of the metric and assign the policy tag of 100 (a unique number per policy) to the route. In the RIB, the Stage 3 filter would simply re-distribute all routes with a policy tag of 100 to BGP.

BGP's Stage 4 export filter would then execute the second policy fragment:

```
from {
    policytags: 100
}
to {
    neighbor: 192.168.2.1
}
then {
    MED: 4
    accept
}
```

This will ensure that the from clause matching occurred in the source protocol by matching the policy tag of 100, which is unique to this policy. The filter will then perform the rest of the matching and, if a match is found, the "then" actions will be performed.

2.4 Policy Manager

Figure 3 summarizes the conceptual model of all filters described so far. Import filters (Stage 1) are present in the protocol's incoming path and export filters (Stage 4) in the protocol's outgoing path. To enable route re-distribution, a filter that performs the source match (Stage 2) is available post-decision in protocols and a map is held in the RIB (Stage 3) which relates policy tags to export protocols.

While all these filter stages are evident to the developer, the implementation details need to be hidden from the operator of a router so that the configuration can be expressed in terms of *what is desired*, rather than how to implement it.

An entity called the *Policy Manager* is responsible for taking the operator's configuration and setting up the relevant filters. For an extensible router, the Policy Manager must have no embedded knowledge of the protocols being controlled. However, it must be able to break the policy into the correct components for each filter stage, and it must be able to check both the syntax and semantics of the policy so that no run-time error can occur in any policy filter stage.

Splitting the policy into components is relatively simple and there are really three different cases to consider:

- Import policies are mapped directly to Stage 1 filters so no split is required.
- A simple export policy for a protocol has a from clause that specifies the same protocol (or does not specify it, and allows it to default). Such filters are mapped directly to the Stage 4 filter bank in the relevant protocol. Any attributes specified in the from clause are matched directly in the protocol's Stage 2 filter.
- A re-distribution export policy for a protocol specifies a different protocol in the from clause. This requires the policy to be split by the Policy Manager as mentioned earlier. First, the from clause must be mapped into a Stage 2 filter in the source protocol and the Policy Manager allocates a new policy tag to identify routes that matched this filter. Second, the to and then parts must be mapped into a Stage 4 filter in the destination protocol, with a match clause added for the policy tag. Finally the RIB Stage 3 map is updated to include the newly allocated policy tag.

Once the split has been done and before actually configuring the filters, the policy manager must check the filter parameters. To do this, it needs information about the route attributes supported by each protocol. We will return to this point after we have discussed the policy filters themselves.

2.5 Policy Filters

All routing protocols are different. They each have their own attributes associated with routes and their own internal representations of routes. We wish to have a single generic policy framework that can handle all routing protocols and we wish to make it as simple as possible for a developer writing a new routing protocol to integrate this policy framework with their routing protocol. To make matters more complex, the filters must be fast, especially for BGP where backbone routers must handle receiving hundreds of thousands of routes from each of dozens of peers, within the space of a few minutes.

The first observation is that in each protocol we only require three filtering stages (stages 1, 2 and 4) which are programmable to satisfy the needs of all current routing protocols. The protocol designer can place these filters in different places depending on the protocol design, so long as they perform the well-defined *import*, *from-match* and *export* roles identified earlier.

The second observation is that the API by which all these filter stages are configured should be uniform in syntax across all routing protocols. If this were not the case, then the Policy Manager would need special knowledge of a routing protocol to be able to configure it, which would conflict with our policy goals.

The final observation is that the internal filtering engine needs to have the same programming model (its programming language) across all routing protocols. Again this minimizes the knowledge the Policy Manager needs to have.

Based on these observations, the design we derived uses a single *generic policy filter engine* for all filters in all routing protocols. From the point of view of the protocol designer, this generic filter is a black-box which will match a route, modify it if necessary and return whether or not the route has been accepted.

Internally, the generic policy filter comprises a simple stack machine, together with its programming API and instruction set. The stack machine needs to be able to execute instructions, obtain attributes from a route, delete attributes from routes, and write modified or additional attributes back to the route. When the stack machine program terminates, it must also return whether the route is to be accepted or rejected.

The remaining problem to be solved is how to provide

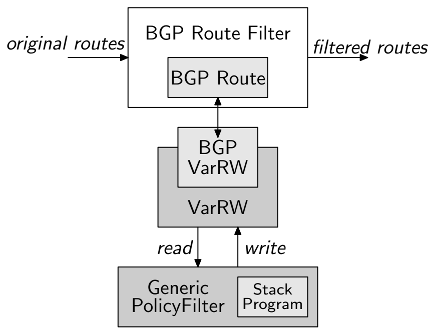

Figure 4: Conceptual model of the generic policy filter.

access to protocol-specific attributes in a policy filter that has no knowledge of such attributes. Our solution is to have the protocol developer write a protocol-specific instantiation of a generic *reader* and *writer* for route attributes. This generic reader/writer is known as *VarRW*.

When a route arrives in the protocol-specific part of a filter bank, the routing protocol invokes the generic filter by calling its `run_filter()` method. It passes in a protocol-specific instance of VarRW which has been bound to the route to be filtered. The filter program in the generic filter will then run, using the VarRW interface to obtain the attributes it needs and to modify the route as necessary. Finally, `run_filter()` returns *true* if the route was accepted, or *false* if it was rejected and should be dropped by the protocol. The conceptual architecture of the generic filter and its interaction with the route via a VarRW instance is shown in Figure 4.

From the point of view of the protocol designer, integrating the policy framework is extremely simple. All the programmer needs to do is write a protocol-specific subclass of VarRW that, on request, can provide attributes specific to the new protocol. These attributes are identified by attribute name and can take one of a limited set of data types. Similarly, it must be able to write these attributes back to the route. This is a very small amount of code: for example, the RIP VarRW implementation is 120 lines of C++ to support both IPv4 and IPv6. For this effort, the programmer immediately gains access to the whole policy framework, its Policy Manager, generic filters, and the expressiveness of the stack machine.

2.6 Route Re-filtering

Production routers operate for long periods of time and it must be possible to reconfigure them on the fly, without the need to reboot or restart peerings. Such reconfiguration includes policy filters. The desired effect from the operator's point of view of changing a policy is that all routes are re-filtered and the new policy takes effect. In some circumstances, the policy framework will need some aid from the developer of a protocol in order to perform this correctly.

If a routing protocol is stateless and routes are refreshed periodically, all routes will be re-filtered automatically upon a refresh. This occurs in RIP [5]. In RIP, routes are re-advertised approximately every 30 seconds. If a policy changes, after 30 seconds all RIP routes will be re-filtered as they re-enter the protocol. No extra effort is needed by the developer in this case.

If a protocol is stateful, as with BGP, an explicit re-filtering of all the current routes will be required. This will cause all the routes stored in a protocol to pass through the policy filters once more. The protocol developer will therefore have to implement this mechanism himself and it will be triggered by the policy framework when necessary. Depending on the protocol's implementation this may be a trivial or demanding task. Care must be taken to propagate the *original* route through the filter and not a instance of the route modified by the previous version of the filter.

3 Implementation

An implementation of this policy framework has been developed for XORP [6]. The main component to develop was the generic *PolicyFilter* which lies at the heart of the framework. A decision had to be made on what language this filter would understand and execute.

One choice would be to use the same syntax of the user configuration as the policy language. The drawback was that the back-end policy filters (the ones in the protocols themselves) would have to perform complex parsing.

The solution was to make the Policy Manager "compile" the user policy into a stack-based language. This language is much more generic than any front-end syntax that users might interact with. Should the user syntax change in the future, only the Policy Manager will need to be updated. Also, if the back-end policy language is powerful enough, it could potentially express any policy and the filters would never have to be changed—all the work would be done in the front-end Policy Manager. The back-end filters are therefore simple stack machines and their language will be presented next.

3.1 Back-end Policy Language

Consider the following policy:

```
from {
    metric > 4
}
then {
    metric = metric * 2
    accept
}
```

This policy will double the metric on all routes with a metric greater than four. Such a policy could be used in either an import or an export filter.

The Policy Manager would compile this policy into the following filter code:

```
PUSH u32 4
LOAD metric
>
ON_FALSE_EXIT
PUSH u32 2
LOAD metric
*
STORE metric
ACCEPT
```

The first `PUSH` instruction will place an unsigned integer with the value of 4 on the stack. The language is typed and each `PUSH` operation needs an explicit type identifier as the type of a literal value is not always obvious from lexical analysis. For example, 4 could have also been a signed integer instead of unsigned.

The `LOAD` instruction will cause the VarRW to read a route attribute which will be placed on the stack. It does not require an explicit type (like with `PUSH`) as the VarRW implementation will return an object of the correct concrete type (see Section 3.4.1).

The next line contains the operator '>'. In this case, it will perform a mathematical greater than operation and return the result on the stack. Each operator is overloaded based on the concrete type of the arguments it is supplied with. The problem is that the concrete type of the elements on the stack is unknown as they are stored according to their base type. The solution is implemented with the use of *multi-methods* [7]. When an operator is encountered, its *arity* (the number of arguments it requires) is checked. If n arguments are required, a hash is computed based on the operator type and the concrete type of the n top-most elements on the stack.[3] This hash is used to find the appropriate method to call for this permutation of operator and argument types (essentially a function signature). All of this is encapsulated in a dispatcher which takes the base type of an operator and its arguments, and performs the magic. The dispatcher is very fast if the hash functions are implemented efficiently.

[3]It is possible to make the hash reflect the concrete type of an object by invoking a pure virtual hash function on the base class object (C++).

After the '>' operator returns its result, the `ON_FALSE_EXIT` instruction will be run. This will terminate the policy execution if a boolean value of *false* is on the top of the stack. In this case, if the match condition was false, the then block would not be run.

If the route did match, the metric is doubled via the '*' operator. The `STORE` instruction will write the value on the top of the stack via the VarRW interface into the route. Finally, the policy filter will accept (not drop) the route upon executing the `ACCEPT` instruction.

Even a complicated front-end syntax may be easily translated into this language. If the language supported `JUMP` instructions, it would be possible to compute virtually any imaginable policy—however we have so far refrained from this to avoid loops in policy programs.

Also, if the concept of what a policy is and how it should be expressed changes in the future, it is possible that our original generic back-end filters will still be appropriate. The back-end language may be thought of as an assembly like language which provides immense flexibility. For example, although software programming has evolved from imperative to object-oriented, the same assembly language is still able to express the result of both programming approaches.

3.2 Types and Operators

Our policy engine is limited by the types and operators it supports. So far, it appears that a small set of operators and types can handle all the policies supported by commercial router vendors, for all the main routing protocols. While we do not expect it to be a frequent requirement, it is possible that some future protocol might need to define a very sophisticated type or operation which is not yet supported. It might in principle be possible to devise a way to extend the operator set at run-time, by using a low-level "microcode" approach, but we took the view that there is little to be gained by this and it would likely hinder performance.

Adding a new type and operator in the base framework is in fact very simple and requires very little code. The only requirement for an operator is to have a string representation. This will be used in order to compute the hash mentioned earlier for dispatching operators. It will also be used for representing the operator in the code produced. Thus, all a developer needs to do is to create a new operator class with a unique string representation and register it with the dispatcher.

New types are also added quite easily. They require a unique hash (used in the dispatcher) and must have a string representation for their value. This representation is again required for the code generation as the code is textual. Furthermore, types need to be constructible via a string. This is required, for example, by the `PUSH` instruction as the actual value will be passed as a string. Finally, the type needs to be registered with a *factory* in order for it to be created when required. Types are created, for example, during semantic checking while policies are being validated.

After a new type or operator has been added, it is available for *all* protocols. It is therefore desirable to create operators and types which may be re-used and that perform simple operations. It is possible to create *macros* which use existing operations to perform more complex ones. We discuss this next.

3.3 Combining Existing Operators

A goal of XORP is that new protocols can be added at run-time without the need to modify existing code. To integrate processes into the management framework and CLI, each new process comes with a configuration template file which specifies the valid syntax and parameters for configuring the process, together with ASCII templates of the inter-process communication calls that the *Router Manager* will use to configure it.[4]

When the Router Manager configures a process, it performs a textual substitution of the parameters in each template IPC call and fires off the request. From a policy point of view, this provides a useful hook. We use this re-writing to implement front-end macros, so that the router operator can express one syntax in the configuration files and CLI, which can then be translated via template substitution into the more generic language sent to the policy manager. The goal here is to support a user-friendly configuration syntax, which is often protocol-specific, without burdening the policy manager with needing to understand such protocol-specific syntax.

Figure 5 summarizes XORP's configuration architecture and the role of template files. The Router Manager first syntax-checks the user configuration against the template files. Configuration directives are then sent to the routing protocols, and policy primitives are sent to the Policy Manager which will ensure that the configured policy is sensible. Next, appropriate code will be generated for the policy. Finally this code is sent to the relevant back-end filters.

To illustrate how this isolates the Policy Manager from protocol-specific configuration syntax, consider the implementation of policy for BGP's AS-path. Using template substitution, we can implement this with only basic types. In our first implementation, the AS-path was be treated as a string and AS-path operations were easily implemented via string manipulation. For example, a common BGP policy command is *AS-path prepending*, where an AS number is added to the front of the path:

```
as-path-prepend: u32
    ..."aspath = ($ARG + ',') + aspath"...
```

The AS-path is a string of comma separated integers, since this is how the user matches regular expressions against it. The policy engine will simply expand `$ARG` to the integer value supplied to the `as-path-prepend` directive, concatenate the integer with a comma, and concatenate the result with the existing AS-path. This will then be stored as the new value of the AS-path.

In this case, the overloaded definition of '+' with an integer and a string as arguments to mean concatenation already existed. However, the policy language also supports casts to aid writing powerful and complex macros.

We have provided a set of basic types and operators to the policy framework so that a developer of a future protocol could easily add complex operations via macros if required. We do expect this set to increase over time, but the hope is that extensions will be re-usable by more than one protocol.

In the end, we implemented a specialist AS-Path type, as BGP performance is all-critical, and string manipulation is relatively slow. This seems to be the right balance between flexibility (we could and did implement BGP without this) and pragmatism (BGP is not just another routing protocol).

3.4 Policy Manager

Until now, details of the back-end filter have mainly been presented. The coordinator of these filters, the Policy Manager, will now be investigated more deeply.

The Policy Manager has more to do than simply compiling programs and sending them off to the relevant filter. The two main problems which the Policy Manager needs to solve are checking for errors in policies and keeping track of "code fragments".

3.4.1 Error Checking

It should be clear now that the actual policy stack programs perform no run-time error checking. This is a deliberate decision as it is essential that the filters perform

[4]For more details of XORP's IPC calls and template files, see [4].

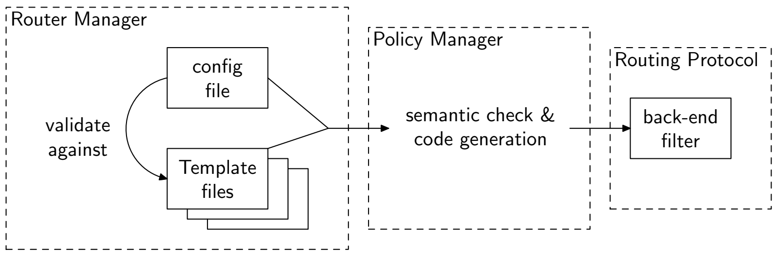

Figure 5: Policy configuration within XORP.

well. This puts the onus on the Policy Manager to perform all the error checking in advance, before compiling the filter programs. Several things need to be checked:

- That the attribute names are those of valid attributes for the protocol in question.
- That the types of attributes match the types of operators to be used and that they match the possible types of literals specified by the user.
- That only attributes permitted to be modified by the protocol are in fact modified by the filter.

The Policy Manager is capable of detecting all run-time errors during semantic checking. The *halting problem* is not an issue since the language does not include jump statements. Similarly, division by zero cannot occur because of the absence of a division operator. If division were required, options would be to allow only literal denominators that can be statically checked, or to require a policy macro to define the value to be returned in the event of divide-by-zero.

The information needed for policy checking is supplied by the policy *VarMap*. This is a configuration file—it is not hard-coded into the policy framework, making it easy for new protocols to add entries. Each new protocol supplies its own VarMap file, which is read at run-time by the Policy Manager. VarMap entries take the form:

```
bgp network4  r
bgp MED       rw
```

This example states that the BGP protocol supports the reading of the "network4" and "MED" variables. Furthermore it also supports the writing of the "MED".

3.4.2 Code Fragments

Policy filters are configured atomically. Code cannot be added or removed from the back-end filters—the whole lot must be reconfigured. This design decision was made in order to keep the back-end filters as simple as possible, but this pushes complexity into the Policy Manager.

A protocol may have several policies attached to it and they will all be executed in the order specified. Suppose that an operator configured one BGP policy and later adds another. The Policy Manager will need to create code for both policies and combine them. Each time the Policy Manager creates code for a policy, it stores it as a *code fragment*. When a new policy is attached to BGP, it is combined with all the fragments needed for this particular BGP filter (import/export). The same occurs when a policy is deleted or modified. The particular fragment is located and is modified or removed. The fragments are then combined and the filter is reconfigured.

3.5 Route Re-filtering

A final issue which deserves further examination is implementing route re-filtering in protocols. In some circumstances, its implementation may be somewhat subtle. In this section we will describe a generic mechanism which may be adopted in order to perform route re-filtering. This mechanism is currently used in BGP's route re-filtering implementation.

Depending on the requirements of an implementation, route re-filtering may be quite difficult to implement correctly. XORP's philosophy is to maintain route consistency no matter what the price is [4]. This means that if route x is added, it is illegal to re-add route x without first deleting it. For this requirement to be met, care must be taken when route re-filtering is being performed.

Suppose that route x has been dropped by a filter. If the policies are now reconfigured, route re-filtering will occur. If route x is now accepted by the new filter, the route needs to be added. It must not be replaced (deleted and added) since, due to the original filter, it was not inserted in the first place. Thus we must maintain an additional bit of state in each route indicating whether has been dropped by a policy filter. This allows the appropriate action to be taken upon route re-filtering.

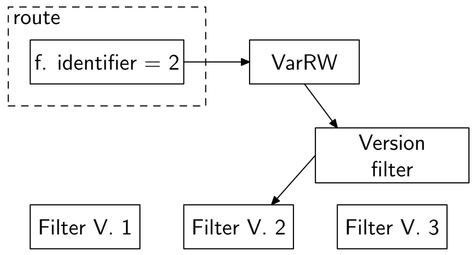

Figure 6: Operation of a version filter.

3.5.1 Version Filters

XORP's BGP implementation has even more stringent requirements for re-filtering. When a route is replaced as a result of a policy change, both the old and new versions of the route are required to ensure consistency. To allow re-filtering at all, the original unmodified route must be kept. If we also stored the modified version of each route, this would increase memory consumption to an intolerable degree, especially for export filters which can modify a route in a different way for each peer.

Our solution to this problem involves trading CPU cycles for memory, and re-running the old filter to regenerate the old version of the route. We do this by storing previous filter programs. Each time the policy filter is configured, it is assigned a unique program version number. All routes have an indicator of which filter they need to be run against. A special indicator (0) is used in order to mark new and unfiltered routes, which will cause them to be run against the current version of the filter.

Figure 6 depicts the operation of a version filter. When a route needs to be evaluated, the version filter will first obtain a pointer to the correct filter program which needs to be run by requesting this as an attribute from the route via the normal VarRW mechanism.

Initially, all routes will have their filter identifier set to 0. As they are filtered, their filter identifier will be set to the current filter version, perhaps 1. Suppose that the filter is now re-configured and obtains a version of 2. Version 1 of the filter is still retained in memory. When route re-filtering occurs, the version 1 filter (as indicated in the route) will be run first. Next, the route will be evaluated against the version 2 filter. This way, both the previous and current version of the filtered route are obtained. It is now possible to conclude what action should be taken. For example if the previous route was accepted and modified whereas the current one is dropped, a delete of the previous route is propagated. It is no longer necessary to store copies of modified routes in memory. All that is required is that each route has extra space for a filter version id—a minimal overhead.

Using this technique it is also possible to spot invariants. If the old filter and the current filter yield the same route, there is no need to delete and re-add the route, reducing routing protocol chatter and churn.

Finally, route filter programs are reference counted. When no route remains that references an old filter program, the program is automatically deleted.

These version filters may be used in any protocol which have design requirements similar to XORP's BGP: route consistency and no copy of previously filtered routes. The version filter will read and write the filter version identifier in the route via the VarRW interface. In order to use them, a developer simply needs to add the filter identifier to the routes and extend the VarRW to support the reading and writing of this identifier.

4 Evaluation

In most protocols, policy should not be the bottleneck of computation. Therefore, policy filters need to have a low overhead and must be able to process many routes in a very short period of time. The first metric used in evaluating our policy filters was indeed routes filtered per second. All the experiments which follow have been performed on a Pentium IV 2.4GHz laptop with 512MB of RAM. The results are from a test harness that uses the normal XORP BGP policy filters, but does not run the rest of BGP. This allows us to separate the performance of policy filters from that of BGP's decision process.

Consider the following BGP import policy:

```
from {
  network4 != 10.0.0.0/24
  nexthop4 != 10.0.66.1
}
then {
  AS-path-prepend: 6234
  MED += 1
}
```

This policy states the following:

To all incoming BGP routes which do not have a network prefix of 10.0.0.0/24 and do not have a next hop of 10.0.66.1, prepend 6234 to the AS-path and increment the MED by one.

Although this policy is not too useful, it captures many aspects of common policies. Firstly, it is attribute intensive. It will read and write many route attributes and will either compare them with user values or assign values to

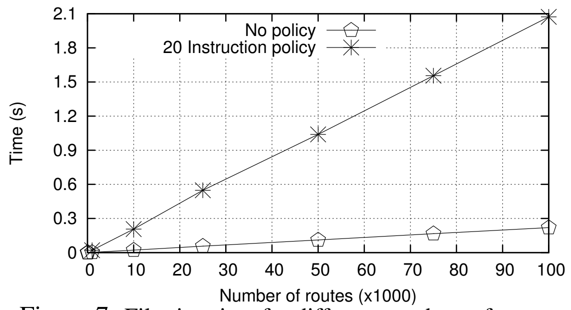

Figure 7: Filtering time for different numbers of routes.

them. This will stress the VarRW interface which potentially is the most difficult to optimize—it relies on the BGP protocol using computationally cheap data structures for attributes. Secondly, the attributes being considered are perhaps the most frequently used in BGP. Certainly, most policies will tend to match network prefixes or next hops. Similarly, AS-paths and MEDs are the attributes which commonly require tuning. The first results we will present use a filter configured with this policy.

Figure 7 shows the time spent filtering against the number of routes processed using the policy above. As expected, the graph is a straight line, indicating later routes take the same time to process as earlier routes. The plot also shows the time taken when no policy is present. This curve represents the fixed overhead of adding a route, passing it through an empty filter and detecting the route as it exits the system. It serves as a baseline when calculating the effective overhead of filtering (the difference between the two curves). On average, approximately 51,000 routes per second are filtered.

To put this figure in perspective, consider a backbone router that reboots and wishes to receive and process a full routing table from its ten BGP peers in five minutes. If the processing time spent for policy is limited to 25% of the total processing time, then the filters must take no more than 75 seconds. With 150,000 routes in a full BGP backbone routing table, the router will receive 1,500,000 routes from its ten peers. The policy filter will need to process at least 20,000 routes per second—our implementation will handle this.

In most cases policies are much more complex than the one used in this example. We examine this next.

4.1 Increasingly Complex Policies

The policy in the previous section expands to 20 total lines of stack language instructions (including some "meta" instructions such as `POLICY_END`). We will now evaluate how our filters deal with more complex policies—ones with more instructions. The metric used will be instructions executed per second.

In this experiment, the number of routes is kept constant (10,000) whilst the number policy instructions is increased. The same policy as before is used although it is inflated as necessary by padding it with numerical addition and assignment instructions. Specifically, the following is performed repeatedly:

```
PUSH "u32" 1
PUSH "u32" 1
+
STORE MED
```

Intuitively, these instructions will perform $1 + 1$ and store the value of 2 in the MED attribute. In this case, the dispatcher (the core of the policy filter) will be stressed. Note that although these are trivial operations, many tasks are being performed in the background:

- Two unsigned integer *elements* with the value of 1 are created. They are both pushed on the stack.
- The arity n of the + operator is checked and n elements are retrieved from the stack.
- The dispatcher hashes the operator + and the n elements producing a key.
- The function pointer for the implementation of this particular operator/argument permutation is retrieved by indexing a hash table using the key.
- The implementation is executed and the result will create a new unsigned integer element with the value of 2. It is pushed on the stack.
- An element is popped from the stack and is written to the MED (or cached).

The VarRW will not impact on performance because of variable caching—the MED will be written only once upon the termination of the policy. However, the implementation which performs the caching will be stressed, although this is under the control of the generic framework and may be optimized by us.

Figure 8 depicts the filtering time as the number of policy instructions increase. The graph would look different if diverse instructions have been used for inflating the policy. However, the goal is not to measure the complexity of individual instructions but rather to evaluate the overhead of the *core* policy engine—namely the stack machine and the dispatcher. For example, a lot of memory management is involved, especially with this particular policy. Each time the result of the '+' operator is

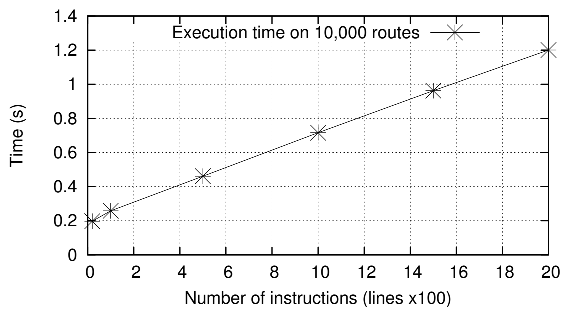

Figure 8: Time to filter 10,000 routes for different policy lengths.

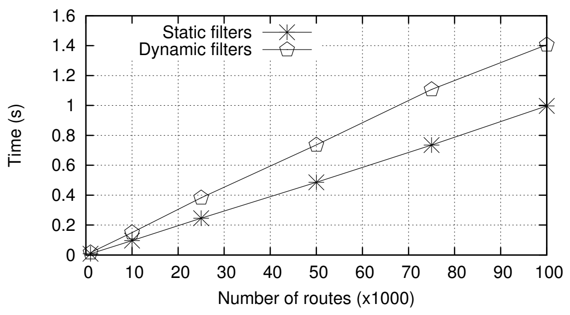

Figure 9: Comparison of static versus dynamic filters for a default EBGP import filter.

created, it may not be deleted immediately because of write caching, since the write could be performed at the *end* of the policy execution. Nevertheless, the complexity remains linear even in such cases. On average, about 16.5 million instructions per second are executed.

For a better understanding of this figure, consider the following illustration. In most cases, a single user configuration directive (i.e. the high level policy representation) will yield ≈ 4 instructions. For example, many directives (such as AS-path-prepend) require:

1. Reading an attribute.
2. Pushing a user argument.
3. Executing an operator.
4. Storing the attribute.

Other directives normally require only 3 instructions—one for the attribute, argument and operator. Therefore, the current implementation will execute about 4 million user directives per second. If a *user* policy contains 200 lines, we can filter about 20,000 routes per second.

Note the mismatch with the previous figure where we claimed $\approx 51,000$ r/s were filtered for a 20 line stack language policy. According to the results of this experiment $\approx 830,000$ r/s should be filtered with the 20 line policy. The difference is mainly due to the fact that the policy in this experiment is stack machine intensive rather than VarRW intensive. The bottleneck of BGP's policy is actually BGP's implementation (how attributes are stored/modified) rather than a under-performance of the generic policy framework.

In summary, all these figures should be taken indicatively as they highly depend on the specific VarRW implementation of protocols and on the complexity of the policy operations. They do however give a general idea of the performance bounds of our implementation.

4.2 Comparison with Static Filters

Just how well should we expect a policy filter to perform? The previous experiments give no idea how close we are to a reasonable lower bound on execution time. Would any good implementation need to take this much time to filter, or are we wasting time using fancy dispatching mechanisms and figuring out how to execute operators instead of performing them directly? To try and answer this question we can compare running the same policy using dynamic filters and hard-coded static filters.

XORP has a set of basic static filters which were needed for standards compliance and pre-date the policy-filter implementation. These are implemented directly in C++. The two static import filters are:

1. Loop detection. If the AS-path in the route contains the AS of this router, drop the route.
2. Default local-pref. Set the local-pref to 100.

The equivalent policy for the dynamic filters has been written and requires 13 lines of stack language.

Figure 9 shows the execution time against number of routes processed, comparing the static version of this filter against the dynamic policy-filter version. The complexity of the dynamic filter is obviously much greater than that of the static version, but the time overhead is only $\approx 40\%$. Dynamic filters are an essential component of any real router, but the important point to note is that in implementing a *generic* framework for policy filters that can be used in all routing protocols, we have not sacrificed performance in any significant way.

This result also confirms that the stack machine itself is not the real overhead. The VarRW implementation of BGP seems to be the most time consuming component. Both the static and dynamic filters need to write BGP attributes. However, only the dynamic filters have the overhead of the stack machine. Since the difference of the two plotted curves is low, the overhead of the stack machine itself is quite minimal when taking into account its complexity.

4.3 Optimization Considerations

All the previous experiments were performed on an optimized version of the policy engine. We are convinced that there is much room for optimization in this design and that the architecture does *not* limit performance in any way. To justify this we will briefly discuss some optimizations we performed and further ones which could be implemented.

The first component to optimize was the VarRW. Instead of requesting reads and writes using a string identifier, such as `STORE metric`, a numeric identifier is assigned to each variable. This can easily be achieved by adding the numerical ID of the variable to the VarMap. This way, code generated will resemble `STORE 1` instead of `STORE metric`. The VarRW implementation may simply use this number to index an array and follow a function pointer which will read or write the variable—achievable in constant time. String comparisons are no longer needed.

The next set of optimizations involved similar changes to the dispatcher. A numeric 5-bit index is assigned to each operator and element type. This will allow for a maximum of 32 types and 32 operators. Since current operators are at most binary, it is safe to compute a 15-bit key by appending, in order, the index of the arguments to the one of the operator. This will produce a 15-bit key which may be used to directly index an array in order to retrieve the function pointer of the specific implementation for this particular operator/type permutation. It will allow dispatching, and thus operator execution, to occur in constant time. However, the memory overhead required is holding an array which can accommodate 32,768 pointers. On a 32-bit machine, this normally means a 128KB overhead. It is a good compromise as the speed improvements are highly notable.

To illustrate the flexibility of this policy architecture and to convince the reader that high speeds may really be achieved, the following optimization may be considered. Instead of "compiling" the high-level user representation of policies into our stack language, it might be possible to produce native machine code by run-time compilation of the stack language. This would run at the speed of static filters but would provide all the benefits of dynamic filters. At the present time though, our performance appears to be adequate and so we do not plan to investigate this approach further.

5 Related Work

As very little has been written in this area, it is hard to compare against other systems. We simply do not know how commercial implementations tackle this problem. However, we do know that they do not have the same extensibility requirements, as all their protocol development takes place in-house. Any attempt to change this business model to support third-party routing software on a common platform will run up against the same sort of problems we have faced.

However, we have looked at how other open-source routers and protocol implementations handled the development of policy in order to see where our work stands.

Zebra and Quagga: The first router we studied was GNU Zebra [13]. It is closely related to Quagga [10] (they share code) so the discussion will apply to both. Firstly, these routers lack flexibility in the policy specification. For example, matches virtually always involve equalities such as *metric = 3*. Specifying a match condition like *metric* < 3 is impossible.

Secondly, route redistribution does not seem very advanced. There is no way to specify match conditions in the source protocol of a redistribution. Instead, all routes from a protocol need to be sent to its destination. For example, it is impossible to redistribute RIP routes learnt from only a particular interface into BGP.

Finally, the code complexity is much higher. For example, BGP's filter implementation is over 3,500 lines whereas in XORP it is under 1,000 lines. The main reason being that the protocol is responsible for parsing and executing policy. The RIP and OSPF implementations also deal with this complexity. This does not occur in our architecture, where instead all parsing and execution is performed by the generic policy engine, and not by protocol-specific code.

MRT: Another open-source router is MRT [12]. Its policy support is very limited and it suffers from the issues discussed on Zebra and Quagga. Furthermore, both RIP and OSPF seem to lack dynamic policy.

The policy implementation of MRT's BGP suggests that supporting policy in a clean way could be problematic. Upon filtering, about 15 "if" statements are used to determine which policy directives the user has specified.

Bird: The policy framework of the Bird [2] router reflects our work most closely. It uses a single filter which supports a mini C-style language with conditional statements and function calls. Our policy language would be equivalent in functionality if we were to implement

`JUMP` statements, but to date we have chosen not to.

Bird's architecture is not distributed. It is a single process router and all protocols share memory. It is therefore much simpler to write a policy framework which interacts amongst all protocols, since all the necessary information is present in one place.

However, Bird's implementation highlights the difficulty of implementing a rich policy language. The core of Bird's filter is implemented in a similar way to MRT's. It contains a large "switch" statement ($\approx$ 400 lines) which determines the operator being executed. A further switch is required in each "case" for determining the types of the arguments. In comparison, the advantage of using multi-methods and our dispatcher should be clear.

Finally, Bird's filter is coupled with its protocols. If new attributes or protocols are added, the filter itself must be modified. In contrast to our VarMap mechanism, Bird does not allow run-time extension.

6 Conclusions and Future Work

Implementing policy in routers in an extensible and generic manner is a surprisingly demanding task. Routing protocols differ significantly in their attributes, requirements, and architectures. We need one framework to manage them all to provide a unified management interface for operators. Furthermore, we want to strongly encourage developers to use the full power of that framework for all protocols because the need for conditional route redistribution creates an inherent coupling between them. To do this, the framework needs to be simple to re-use, no matter what the protocol design is.

In this paper we presented a single solution which will accommodate routing policy in all of today's most used protocols. It is a highly extensible and yet performant framework which has no in-built knowledge of specific protocols. Incorporating policy into a new routing protocol primarily involves implementing an interface which allows our generic filter to read and write specific route attributes.

We have successfully implemented this policy framework in XORP. Policy has been added to BGP and RIP with relatively little effort, and is currently being added to OSPF. While the details in this paper are unashamedly linked to XORP, we believe that the problem breakdown, ideas, and architecture are more widely applicable.

To date we have concentrated on providing the sort of router user interfaces supported on commercial routers. In fact our policy framework consists of three levels: user interface, policy manager interface, and 4-stage filter stack-machine interface. Looking into the future we can see that the biggest benefit would be if we could configure and reason about *network-level* policy rather than *router-level* policy. We do not yet know whether revealing these internal interfaces to an external policy engine could permit this, but it certainly seems that this would provide much more expressiveness than can currently be configured through any commercial router CLI.

Acknowledgments

We would like to thank Atanu Ghosh and Orion Hodson for their feedback on earlier versions of this paper. Many thanks also to Pavlin Radoslavov, Marko Zec, Atanu Ghosh and Bruce Simpson for their invaluable assistance with integrating policy into XORP.

References

[1] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, and M. Terpstra. Routing Policy Specification Language (RPSL). RFC 2622, IETF, June 1999.

[2] Bird project. The BIRD Internet Routing Daemon. `http://bird.network.cz/`.

[3] L. Blunk, J. Damas, F. Parent, and A. Robachevsky. Routing Policy Specification Language next generation (RPSLng). RFC 4012, IETF, March 2005.

[4] Mark Handley, Eddie Kohler, Atanu Ghosh, Orion Hodson, and Pavlin Radoslavov. Designing Extensible IP Router Software. In *Proc. 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, USA, May 2005.

[5] C. Hendrick. Routing Information Protocol. RFC 1058, IETF, June 1988.

[6] ICIR. eXtensible Open Router Platform. `http://www.xorp.org`.

[7] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[8] J. Moy. OSPF Version 2. RFC 1583, IETF, March 1994.

[9] D. Oran. OSI IS-IS intra-domain routing protocol. RFC 1142, IETF, February 1990.

[10] Quagga project. Quagga Routing Suite. `http://www.quagga.net/`.

[11] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). RFC 1771, IETF, March 1995.

[12] University of Michigan and Merit Network. MRT: Multi-threaded Routing Toolkit. `http://www.mrtd.net/`.

[13] Zebra project. GNU Zebra routing software. `http://www.zebra.org/`.