Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Stephan Dollberg

Unleashing The Dragon

Implementation and validation of distributed route aggregation in the wild

Master Thesis 227-1501-00L
7.9.2015 to 7.3.2016

Supervisor: Prof. Laurent Vanbever

Abstract

The Border Gateway Protocol (BGP) is used as a routing protocol to exchange reachability information between Autonomous Systems. In recent years, it had to fight with operational problems, such as scalability. Several extensions and additions have been proposed to solve those problems, but at the same time stay compatible with the widely deployed current version of BGP. One of those proposals is Distributed Route Aggregation on the Global Network (DRAGON). It tries to solve BGP's problems by filtering routes that do not add any new reachability information and by combining multiple routes to a single route without losing information. As part of this thesis, we show that DRAGON can be implemented efficiently. Moreover, we depict some open questions that still have to be solved before DRAGON can be deployed into production.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

August 2014 was a rough month for several network operators and internet services such as Comcast, eBay or LastPass. At that point a small friction of the internet routers started to receive and maintain more than 512k prefixes. This number is the limit for many older devices. As a consequence, they could no longer function correctly and a part of the internet lost connectivity [1].

This incident showed an inherent problem in the design of the internet. The internet is a network of Autonomous Systems (AS). Yet, each AS freely decides on its routing process and connectivity. This includes how they buy address space, with whom to peer or which routes it announces to any of its peers. As a consequence, the address space of two peering ASes do not have to be related at all. This happens even more with the increasingly popular concept of *Multi-Homing*. In a multi homed setup a customer AS that has bought address space from another provider AS is not only connected to that provider but also to another unrelated AS to increase connectivity. This shows yet another problem. In a non-multi-homed system a provider AS can filter a customer sub prefix. This is not possible in a multi-homed setup. The reason is that packets are being routed based on longest-prefix matching. This would route packets to the customer AS not via the provider AS. However, the provider AS wants to route for its customers out of economical reasons. This lacking hierarchy leads to the huge amount of prefixes that have be announced.

To give counter examples of systems that provide a better hierarchy the telephone or road system can be mentioned. Telephone numbers have a country prefix based on which the country of origin of a telephone number can easily be located. The road system shows a similar aspect. If you are in Zurich and want to drive to a small suburb in Munich, the roadsigns in Zurich only show routes to Munich or bigger cities on the way to Munich but not directly the way to the small suburb.

The problem is further amplified by the architecture of routers. Routers do generally consist of two layers. First, the data plane which is responsible for forwarding actual data packets according to rule sets to certain network interfaces. Second, the control plane which consists of the logic that determines the rules for the data plane. The table that stores all the routes of a router is commonly called the routing table or Routing Information Base (RIB). On the other hand, the active forwarding rules are stored in the Forwarding Information Base (FIB).

The control plane is usually implemented using general purpose hardware for embedded systems. However, the data plane is implemented using specialized hardware to forward data packets as fast as possible. The FIB is stored in expensive memory that allows very fast access times. Due to the high costs of such memory its size is limited. This can be compared to how small caches are in normal CPUs. Thus, a router can only have a limited number of routes in its FIB. For some of the older routers that crashed in the August 2014 incident the size was 512k routes.

ASes use the Border Gateway Protocol (BGP) to exchange routing information. However, BGP does not make use of any hierarchy. While topics such as multi-homing make any advanced route aggregation harder there are still usecases for it. Even multi-homed ASes are mostly only multi-homed to ASes that are in close proximity to each other. Those ASes might have another provider in common. Thus, a certain degree of hierarchy is existing.

A route aggregation technique that tries to improve BGP and exploit any existing hierarchy is Distributed Route Aggregation on the Global Network (DRAGON). It works by locally filtering prefixes and aggregating others that can form a shorter one. In addition, DRAGON can be deployed locally. However, it provides incentives for ASes to adopt it. Simulations show that DRAGON can filter up to 80% of prefixes [2].

As DRAGON is designed to work on any prefix based routing system it is not specifically targeted at BGP. In this thesis we want to analyze the feasibility of implementing DRAGON in a real software router on top of BGP. Specifically, we are interested in possible problems that occur in relation to the BGP protocol and how router performance is affected.

Chapter 2

DRAGON

DRAGON is a technique that tries to solve the problem of the ever growing amount of routing prefixes. It does that by applying two algorithms.

First, the *filtering algorithm* which stops routes from being forwarded to neighbors that do not add any advantage from a routing perspective and are already covered by a shorter prefix. Second, the *aggregation algorithm* that combines routes for prefixes that can in combination form a shorter prefix. In the following we will explain both algorithms and their theoretical details [2].

2.1 Routing Model

The routing model, on which DRAGON is based, requires the ability to express arbitrary routing policies (algebraic theory of dynamic network routing [3]). One such model are the Gao-Rexford (GR) routing policies. Three types of relations exist between AS neighbors in the GR model - provider, peer and customer in the order of increasing preference. AS nodes forward incoming messages from providers or peers only to customers. Routes from customers are forwarded on all link types. These decisions are economically driven. In graphical figures the relationship is depicted using a top to bottom hierarchy. Providers are on top of customers and nodes on the same level are peers [4].

As an addressing schema IPv4 addresses are used. A subnet or prefix consists of a 32 bit IP prefix address together with a prefix length ranging from 0 to 32. Given two prefixes *p* and *q* with *q* having a longer prefix length than *p* and the prefix address of *q* being part of the prefix address of *p* then we say *p* is a *parent* prefix of *q* and vice versa *q* is a *child* of *p*. Looking at concrete examples, if *p* is $100.0.0.0/16$ and *q* is $100.0.0.0/24$ then *p* is a parent of *q*. On the other hand, if *p* is $200.0.0.0/16$ and *q* is unchanged then *p* is not a parent of *q*.

2.2 Filtering

2.2.1 Algorithm

For a node that receives a prefix *p* (parent prefix) and a longer more specific prefix *q* (child prefix) the filtering algorithm is defined by:

Algorithm (Filtering Algorithm). *"If the node is not the origin of p and the attribute of the elected q-route equals or is less preferred than the attribute of the elected p-route, then filter q. Otherwise, do not filter q." (Code CR) [2]*

Figure 2.1 depicts DRAGON in action. Node u_3 originates prefix *p* (e.g.: 200.0.0.0/16) while node u_4 originates prefix *q* (e.g.: 200.0.0.0/24). The other nodes might originate other prefixes

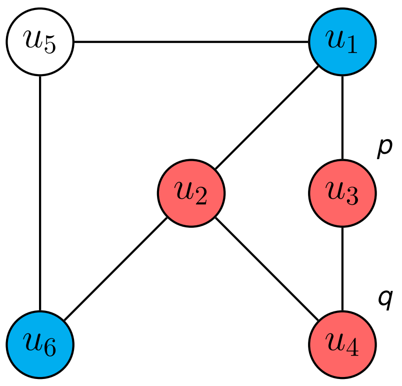

Figure 2.1: *Example topology showing how DRAGON is applied. Node u_3 originates parent prefix p and node u_4 originates the child prefix q. Red nodes install routes to both prefixes in their FIB. Blue nodes can filter q and do not install q routes in their FIB. However, they keep the routes in their RIB. White nodes are oblivious to q and do neither have a route in their FIB nor RIB. [2]*

though that is not important for this example. Node u_3 and u_4 are both originating nodes and such do not participate in any filtering code. Node u_1 however receives a customer *p* route from u_3 and a customer *q* route from u_3 and u_2. Thus, it can filter the *q* route meaning that it will not forward *q* to neither u_5 nor u_2. In addition, it neither puts it in its forwarding table. However, the router still keeps an entry in its routing table as it might need to announce the route once the *p* route gets withdrawn. Node u_2 then receives a *p* route from its provider u_1 and a customer *q* route from u_4. As customer routes can not be filtered by provider routes it forwards both *p* and *q* as in normal BGP. Analog to u_1, node u_6 receives only provider routes and can such filter *q*. Node u_5 will only receive *p* as *q* is being filtered by both u_1 and u_6.

This shows one of the benefits of DRAGON. Only the nodes that are adjacent to a child prefix have to be aware of the q route. Thus, neighbors of the adjacent nodes save routing and forwarding space.

In this context, we use GR link types as attributes. In reality and real BGP these attributes and GR routing policies have to be modeled. Possible attributes include the administrative preference, BGP local pref, AS-path lengths or BGP communities amongst others. We discuss practical considerations of those attributes in the implementation section.

2.2.2 Rule RA

There is another important restriction to the filtering algorithm called *Rule RA*:

Algorithm (Rule RA)**.** *"The origin of p announces p with a route whose attribute is equal or less preferred than the attribute of the elected q-route." (Rule RA) [2]*

This means that if a node originates a prefix p and receives a more specific prefix q it can only export the p route to neighbors to which it also exports the q route.

Figure 2.2 shows how without adhering to rule RA a blackhole would be created. Node u_2 receives a customer prefix *p* and a provider prefix *q*. As such, according to the filtering algorithm it is allowed to filter *q* and not forward the *q* route to node u_3. In addition, it only installs *p* in its forwarding table. Any packet coming from u_4 with destination in *q* will now be blackholed at u_3 which does not have any route leading to u_1. This is in contrast to normal BGP in which u_3 would have a route to *q* installed via u_2. Adhering to Rule RA forbids u_3 to announce *p* to u_2. Instead, the router has to deaggregate *p* into *p* - *q*. Those prefixes are then announced in place of *p* which is being withdrawn.

While the scenario in 2.2 is probably extremely rare in reality due to economic reasons such a case can be created in a different way as shown in the next subsection.

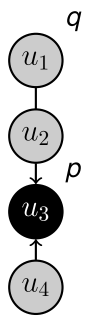

Figure 2.2: *Topology that shows how Rule RA is required to avoid the creation of blackholes. Node u_1 originates a child prefix q via a provider route to u_2. Node u_3 originates the parent prefix p via a customer route to u_2. Thus, u_2 can filter the q prefix. As a consequence, a blockhole is created at node u_3 as u_2 will forward packets with destination in q to node u_3. The arrows indicate how packets traveling to q are being routed. Enforcing Rule RA forbids u_3 from announcing p to u_2. [2]*

2.2.3 Link Failure

A consequence of rule RA is that special behavior is needed on link failure. If in Figure 2.1 the link between u_3 and u_4 fails then u_3 no longer receives the *q* route via a customer link (specifically, it does not receive it temporarily at all). This means that according to rule RA the node can no longer send *p* upstream to its provider and has to withdraw the *p* route. Given that this would not be route consistent with how normal BGP operates special action needs to be taken. As explained above, the node deaggregates *p* into the sub-aggregates that form the prefixes *p* - *q*. It then announces these routes instead of the single *p* route to its provider.

This case shows a disadvantage of DRAGON as on link failure we create additional prefixes that would normally not be announced. The amount is determined by the difference of the prefix lengths of the *p* and *q* prefix. However, we can avoid the propagation of those additional prefixes using another technique of DRAGON that we introduce in the next section.

2.3 Provider Independent Aggregation

The second technique that DRAGON uses is the *Provider Independent Prefix Aggregation (PIA)* algorithm.

Provider independent prefixes are prefixes that are not assigned by a provider. As those prefixes do not have any parent they can not be filtered by DRAGON. The PIA algorithm allows to aggregate several of those prefixes into a single parent prefix that includes all provider independent prefixes and does not introduce any new address space. At the same time, the attributes of the aggregated routes have to be merged in a way that makes the new route not better than any of the provider independent routes. A node that executes the PIA algorithm does not filter based on the aggregated prefix itself. In addition, it does not put the PIA route into its forwarding table. Both of these actions happen at a neighboring node as part of the *filtering algorithm* as described in the previous sections.

Figure 2.3 shows how PIA aggregates three distinct prefixes to a single one. Nodes t_1, t_2 and t_3 announce the shortened prefixes 100, 1010 and 1011. Once node u_2 receives these prefixes it aggregates them and creates a new additional prefix 10. As explained above, u_2 forwards all four prefixes to u_1. It will not add the 10 prefix to its own forwarding table. However, node u_1 can apply the filtering code and filter the aggregated prefixes. As such only the aggregated prefix 10 is forwarded to any potential neighbors. Note that u_2 might temporarily aggregate the prefixes from t_2 and t_3 to a 101 prefix which upon arrival of the 100 prefix from t_1 can then be further aggregated to 10.

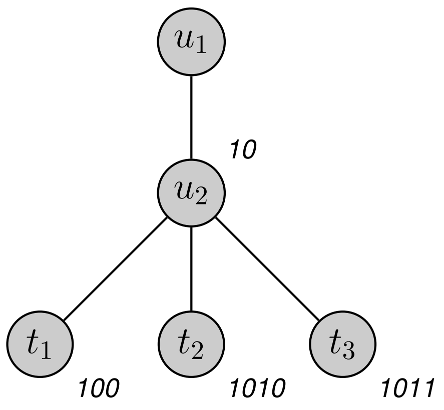

Figure 2.3: *Example topology that shows how prefixes are being aggregated using Provider Independent Aggregation (PIA). Nodes t_1, t_2 and t_3 announce prefixes 100, 1010 and 1011. Node u_2 can aggregate those three prefixes to the shorter prefix 10, which will then be announced in addition to the forming prefixes. Finally, u_1 can filter the longer prefixes based on the aggregated prefix 10 from u_2. [2]*

PIA allows DRAGON to further reduce the amount of prefixes in the internet. Sets of provider independent prefixes can be replaced by aggregation prefixes. However, PIA is currently not limited to real provider independent prefixes. This means that prefixes can also be aggregated if they do in fact have a provider (are provider dependent). Locally PIA causes an increase in the exchanged messages as the aggregation prefixes have to be announced or withdrawn. However, this effect is offset by the overall reduction in the network.

In the filtering code we explain the problem of having to split up a prefix in case of a Rule RA violation. These additional prefixes have to be announced across the internet which increases the global amount of messages that have to be exchanged. With PIA this problem is neglected. Node u_1 in Figure 2.1 receives prefixes $p - q$ from u_3 and prefix q from node u_2. This allows u_1 to aggregate *p* and subsequently u_5 can filter the sub-prefixes of *p*. This also shows how PIA applies to prefixes that are not really provider independent.

It shall be noted that several levels of aggregation prefixes can exist. Imagine t_1 being a provider link instead. In such a case, u_2 could create two aggregation prefixes.

First, the 10 as a provider prefix because it is combining two customer prefixes and one provider prefix. Second, an additional customer prefix for 101. This allows u_2 to forward the later prefix even to providers which it can not do with the 10 provider prefix.

Chapter 3

Design

In this chapter we show certain problems and solutions that originate from mapping the theoretical model of DRAGON on BGP. Finally, we will present an overview of our DRAGON implementation.

Border Gateway Protocol (BGP)

The Border Gateway Protocol is the de facto standard protocol in use to exchange reachability information messages between ASes. In addition, it is also used internally in an AS to exchange routes between the different routers in the AS. Used in this scenario (intra-AS) it is commonly referred to as internal or interior BGP (iBGP). Respectively, used between ASes (inter-AS) it is called exterior or external BGP (eBGP) [5].

Two main message types exist in BGP to exchange Network Layer Reachability Information (NLRI). Update messages are used to announce reachability of a certain network prefix while withdrawal messages are used to withdraw a previously announced NLRI.

BGP is a path vector protocol. This means that each BGP message contains the full path that a message has traversed. Specifically, this is a list of the AS numbers of each AS the message has been forwarded to. Using this path loops can be detected and avoided.

Each route has certain attributes attached to it. Such attributes can simply be the length of the path that the route has taken or specific preferences for different routes that can be configured by the operator. This allows operators to take influence on how packets are being routed. In the following section we will introduce the attributes that are important in the context of DRAGON.

Naturally, BGP routers can receive multiple different routes to a certain prefix. To elect the best path the BGP decision process is executed. This process compares the attributes of the different routes according to predefined procedure by the BGP standard. However, as mentioned above, the operator can influence the process by setting certain attributes. The most preferred route is elected and will be announced to the peers of the router. According to this principle NLRI is propagated through the network.

On a low level, BGP works by establishing a TCP connection between two communicating routers. Each router follows a certain state machine until the protocol has fully started up. At that point, NLRI is being exchanged via update and withdrawal messages. Routers periodically send keep-alive messages to each other to detect link failure. If a router disconnects, the routes of that peer are considered withdrawn and the BGP decision process will be run for the prefixes that changed.

Finally, BGP is a very old protocol with its first version appearing twenty years ago. It has been revised several times to fix problems that are related to it. Specifically, there are problems related security, stability and scalability (routing table growth). Several proposals have been suggested to solve those problems. DRAGON is one of those which especially targets the later two areas.

BIRD

As a platform of integration we choose the *Bird Internet Routing Daemon* (BIRD) [6]. BIRD is an open source routing daemon. It supports various protocols such as BGP, OSPF and RIP. In addition, it is commonly used at internet exchange points (IXP) as a route server. Using an existing route server allowed us to not having to re-implement BGP and we could directly focus on DRAGON.

BIRD offers the ability to configure different sets of instances of protocols. A *protocol instance* represents a connection to a peer via a certain protocol. Multiple protocol instances can be connected via global or per protocol instance routings tables. Protocol instances can use user-configurable import and export filters. A special protocol is the *kernel* protocol that connects to the hosts routing table (Linux). For our prototype we limited the scope to a single global routing table and only the use of BGP. Naturally, BIRD also allows to configure routes which it will originate.

Internally, BIRD has a modular architecture and is implemented in C. It consists of a core logic which is being executed for each NLRI update message. As part of this process, callback functions on different protocol instances are being called. For example, to compare different routes and to select the best one for a prefix the different BGP, OSPF or RIP best route election functions could be called. In addition, protocol instances carry a list of announce hooks. These hooks serve as an announcement channel to other protocol instances/peers.

This modularity can be used to make DRAGON a distinguished protocol next to BGP. Being able to configure both DRAGON and plain BGP protocol instances can make sense in scenarios with multiple independent routing tables. However, for simplicity we have integrated DRAGON directly into the BGP module in our prototype.

Our implementation is not intrusive in the sense that we did not modify any of BIRD's core state machine or event loop. We have only hooked ourself at positions at which it is required. Specifically, most of the DRAGON action happens after the standard route recalculations and announcements have happened. Naturally, some parts of DRAGON, for example the filtering part, need to happen before a route is announced.

3.1 DRAGON in BGP

In general, DRAGON can directly be translated onto BGP. However, there are small aspects that are not immediately obvious in regard to its implementation. The following sections will explain such cases and how we handle them.

Translating the GR Routing Model

First, we show how the GR routing model is generally implemented in BGP. Finally, we explain what the consequences of that are for DRAGON and how we modeled route attributes that allow a route to filter another one as part of the filtering algorithm.

As mentioned in the last chapter, the GR routing policies are an economically driven model. Such, a BGP update message that informs a neighbor about a new route does not carry any information such as coming from a customer, peer or provider. These attributes have to be configured by the operator in the router's import filters.

There are several possibilities to model GR routing policies in BGP. Generally, certain attributes are set on incoming routes in import filters which are then checked upon in export filters so that e.g.: provider routes are only exported to customers.

The first of those attributes is called the administrative preference. This is an attribute that is not related to BGP. However, it exists in BIRD and many other routers. It spans over different protocols and allows to order routes even across different protocols. A possible way to model GR relationships is to mark incoming routes from customers with a higher administrative preference. Thus, export filters to providers or peers can be configured to only allow routes that have an administrative preference which is at least on customer level.

The second is the BGP Local Preference (also just called BGP local pref). It is similar to the administrative preference but is limited to the scope of BGP. The local preference is a common choice to prefer certain routes over other ones. Especially, it is used intra-domain in an AS to prefer specific entry and exit points of an AS. Still, it can equally well be used to model the GR policies. The implementation looks similar to the one of the administrative preference. Routes from customers are tagged with a higher local pref than peers or providers. The peer over providers preference can be modeled by assigning a value that is above provider preference and below the customer level.

The third commonly used attribute are BGP communities. They are 16 bit numbers that can be attached to a route. BGP communities can be transitive meaning that they will be part of an update message and are being sent over the wire. The meaning of a certain community is context dependent. ISPs can publish the way they interpret certain communities so that customers can tag routes with them to enforce a certain policy at the provider. However, they can also be used locally in a router. To model the GR policies customer routes are tagged with a certain BGP community. Export filters can check whether this community is set and filter untagged routes. BGP communities also allow more complex rules than just the customer, peer and provider model.

For DRAGON the attributes are important as filtering of a longer prefix is only possible if the shorter prefix is at least as preferred as the longer one. As such, a production implementation should support the three possibilities of configuring the GR policies.

As part of our prototype we only use the administrative preference and BGP local pref. On the other hand, we want to test how far DRAGON can be implemented without imposing a certain configuration onto the operator. The subtle difference between the two is that we do not require any values to have a certain meaning. For example, a simplified implementation could assume that a BGP local pref above 200 means *customer*.

Our filtering code works as follows. Given a short prefix with elected route *x* and longer prefix covered by the short prefix with the elected route *y* the elected route for *y* can be filtered if *x* has a higher administrative preference than *y* or if the route *x* and *y* have the same administrative preference and *x* has a BGP local pref that is at least as high as *y*.

For the integration of routes with attached BGP communities into the filtering process we have two possible recommendations. The first possibility is to compare the communities in the filtering check. As there is no ordering imposed on the community list we recommend to first sort the list and then check whether the communities of route *y* are a subset of route *x* one's. Alternatively, one can argue that communities that are set internally as part of import filters will have a deterministic order. As a consequence, no sorting is needed. The second more extensive possibility is to run the export filter for every connection on both routes. If the parent route *x* can be exported to all neighbors to which *y* can also be sent then *y* can be filtered. This way is correct independent of which attributes are considered for filtering.

In addition, communities could also be used to define a strict definition of what customers, providers and peers are. Thus, filtering checks could be reduced to checking whether the specific communities are set. As explained above, this would lead to a loss of generality.

Finally, another attribute is interesting to consider for the question of filtering. As BGP is a vector routing protocol each update message carries the path that it has taken. During the BGP decision process the length of this path is taken into account to decide the best route between several routes to the same AS. As a matter of this it is possible to also take the AS path length into account for the DRAGON filtering decision. Directly comparing path lengths does neglect a lot of filtering opportunities. An example can be seen in figure 2.1. Taking the AS path length into account, node u_6 is not able to filter prefix *q* as the route to *q* via u_2 is shorter than the path to *p* which originates in u_3. However, path lengths alone might not always be the best indicator of the best route. Alternatively, a possible solution is to introduce a threshold below which the AS path length will not be taken into account. Increasing this threshold simulates the scenario without taking the AS path into account at all.

Implementation of Rule RA and Link Failure

In Figure 2.2 we showed why Rule RA is important to avoid the creation of black holes. However, we also explained that such a scenario can naturally not occur. As such, for our prototype we make the assumption that such situations do not appear. In addition, we assume that on startup all neighbor links are intact.

However, we still need to account for link failures. Link failures create the need to enforce Rule RA and deaggregation on nodes that originate a prefix. On links that are directly connected to a router it is easy to realize a link failure. BGP has built-in keep-alive messages that will fail if a link goes down. Once such an event is triggered we check whether deaggregation is necessary.

For each route that we lose via the link down event we do the following deaggregation checks. We iterate over all the parent prefixes and check whether we only have static prefixes. A static prefix is a prefix that the router itself is originating. If there is a single dynamic route (a route that has been learned via some other neighbor), we do not need to deaggregate. If this is not the case, we check whether rule RA is being violated. Here we have the problem that we have to compare the attributes of a static route with the attributes of a dynamic route. Static routes are not real BGP routes. Thus, they neither have a BGP local preference assigned nor BGP communities. This forces us to do the extensive attribute check that we described in the previous section. For each neighbor we run the export filter on the parent route. If the parent is being exported, but we have no other child route left or the elected child route is being filtered by the export filter, we are violating Rule RA. In the case of a violation the parent prefix is being deaggregated. This causes the withdrawal of the original *p* route and the announcement of the *p* - *q* prefixes. Note that it is important to exclude static routes in all those checks. The reason for this is that providers often statically configure routes to their customers.

However, there is a problem associated with that. Naturally, we do not have global knowledge over all link failure events that occur. The only events a router can see are the ones that appear on links that are connected to that router. This creates a problem in the scenario shown in figure 3.1.

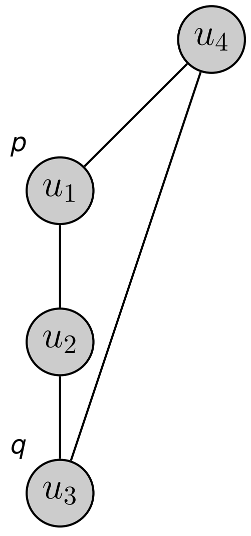

Figure 3.1: *Topology that shows the problem of differentiating a real withdrawal from link failure. If the link between u_2 and u_3 goes down u_2 will create a withdrawal message for q. The same happens if u_3 actively withdraws q. Node u_1 can not differentiate between those two events which has consequences for how to handle link failures. [7]*

In this scenario if the link from u_2 to u_3 goes down u_2 can no longer originate *q*. As such, it creates a withdrawal message which it sends to u_1. As explained above, the deaggregation logic is only triggered if a link down event is registered. However, such an event is not triggered in this case. Thus, u_4 still receives the full *p* prefix and *q* is being blackholed at u_1. A solution is to trigger the deaggregation logic whenever a withdrawal message is received for a child prefix. However, this creates unnecessary deaggregation events. Again, if u_3 withdraws *q* because it no longer wants to announce the prefix then u_2 will forward this withdrawal to u_1 which needlessly deaggregates. To remove this deaggregation a router restart is required which can be triggered after the operator learns on an out-of-band channel that the *q* AS actively withdrew its prefix. Another negative aspect is that in the case of a real withdrawal PIA aggregation can not happen at other neighbors because alternative routes to *q* are being withdrawn as well.

The root cause is that a router can not differentiate between real withdrawals and link down events on its neighbors. Solving this problem with an in-channel solution is difficult as by default there is no concept of attaching information to a BGP withdrawal message. Alternative solutions, such as sending a negative acknowledgment in the form of an update which is tagged with a certain community to indicate a link down event, might be possible.

For our prototype we have decided to assume that children networks are directly connected. As a result, we can differentiate between the two types of events. Note that this does not limit the prototype. Deaggregating in both cases only entails removing a certain check in the code.

To reaggregate a deaggregated prefix we perform a check once a child route of a deaggregated prefix is received. The check evaluates whether Rule RA is still being violated. This is done according to the same principle as in the deaggregation case. If there exists a peer to which the parent prefix can be exported but the child prefix can not be exported, Rule RA is still being violated. Otherwise, Rule RA is no longer violated and the parent route can be reaggregated.

PIA Route attributes

As part of the PIA algorithm we create new BGP routes. For those routes we want to pertain attributes of the routes that formed the prefix of the new route. This is important as the new routes shall follow the same policies as the old ones. We attach the administrative preference, the BGP local preference, BGP communities and the BGP AS path.

As mentioned above, the attributes of the new route shall not make the new route more preferred than any of the routes from the aggregate. We will call those routes the *forming routes*. In the following we will explain how we assemble the individual attributes.

The administrative preference is simply the minimum administrative preference of the forming routes.

$$AP_{PIA} = min([AP_{FR1}, \ \dots \ AP_{FRn}])$$

Similarly, the BGP local preference is the minimum BGP local preferences of the forming routes.

$$LocalPref_{PIA} = min([LocalPref_{FR1}, \ \dots \ LocalPref_{FRn}])$$

The question of how to form the AS path is more interesting. Naturally, the AS path indicates which path a route has taken (through which ASes). However, aggregate routes do not have a single origin but have a set of originating ASes. The concept of a sequential AS path does not fit the concept of such a set. However, there is an alternative to the sequential AS path. The AS path attribute in a BGP update message can be of two different types. While the classical sequential AS path is defined in BGP as a *AS_SEQUENCE* type there exists an alternative type called *AS_SET*. The *AS_SET* attribute indicates that the ASes in the AS path are an unordered set. This allows for usecases in scenarios as ours. Cisco routers that support manual route aggregation techniques use the *AS_SET* attribute. However, the *AS_SET* attribute is considered deprecated [8]. As such, we decided to not use the *AS_SET* attribute for PIA routes. Instead, we create a normal AS path of a certain length consisting only of our AS. The length of the path is determined by the maximum length of all the forming routes. This solution is not perfect either as we lose the information of which ASes are part of the route's path. Such information might be useful to some party. For example, operators might not want to route through a certain country or a competitor AS. Consequently, the optimal way to set the AS path attribute requires more research.

$$AS\ Path_{PIA} = [AS\ number, ..., AS\ number]$$

Finally, the correct BGP communities have to be attached to the aggregated route. Certain policies can be implemented based on whether a BGP community is set in an update message or not. Consequently, we can only attach a BGP community to the route if it exists in all of the forming routes. To tag the PIA route with the correct communities we build the intersection of the communities of all the forming routes. The intersection set is then attached.

$$BGP\ Communities_{PIA} = BGP\ Communities_{FR1} \cap \ldots \cap BGP\ Communities_{FRn}$$

Using those attributes the aggregate route is created and inserted into the routing table. It is then subject to the configured policies based on its attributes.

3.2 Implementation Details

Handling of Static Routes

As mentioned in the previous sections, a static route is a route that is created by the operator in the configuration of BIRD. This is used by operators to configure the prefixes which they originate. In the context of the DRAGON implementation certain edge-cases exist that are especially complex due to static routes.

Static routes do have a default administrative preference of 200 in BIRD. In contrast, the default administrative preference for routes learned via BGP is 100. Thus, statically configured routes are always preferred to learned ones.

In plain BGP routes are only compared in the context of a single prefix. Further, the BGP logic is mostly concerned with acting on changes of the elected best route. In DRAGON the routes of different prefixes are compared as well. In addition, in certain scenarios we might also have to take action if a route is being removed which is not the best route as elected by the BGP decision process. As a consequence, static routes introduce certain tricky edge-cases in the DRAGON implementation and are a common source of bugs.

We have already shown examples of such cases. For example, the way Rule RA is being checked is a consequence of this. Static routes do not share the BGP attributes and such can not directly be compared with a BGP route making simple Rule RA checks on link failure impossible.

A similar edge case is also related to link failure. Given are a provider AS *P* that originates a prefix and which has delegated a certain subnet of that prefix to a customer AS *C*. A common scenario is that operators of AS *P* statically configure routes to their customer AS *C* besides the ones to their own prefixes. This leads to the fact that the routers in AS *P* have two routes to the *C* prefix(es). First, the statically configured one. Second, the one which is learned from the customer AS. Given that static routes have a higher administrative preference by default they will become the elected routes. However, even in this scenario the routers still need to trigger the deaggregation behavior of the static parent prefixes on link failure. As a consequence, we have to act on link failures even if not the best route is going down. In addition, we have to temporarily filter the static route. As the list of routes for a certain prefix is not sorted (besides the elected one being in front) the Rule RA check also includes searching for the second best route.

Another question is whether static prefixes should be taken into account for the formation of PIA routes. Again, the question is how the attributes for a PIA route would be calculated if it is partly formed from a static route. While we initially started implementing DRAGON with static routes in mind for PIA, we recommend to ignore static routes for PIA aggregation. The reason is that too much special handling is required in attribute creation. In addition, static routes mostly appear in scenarios in which one would aggregate children prefixes or parent prefixes. Both cases are better covered by the next hops via filtering or aggregation.

Compact Prefix Tree

As explained in the introduction, BGP normally does not exploit any prefix or routing hierarchy. As a consequence, plain BGP routers do not need to store their prefixes and routes in an ordered or hierarchical data structure. This applies to the control plane. In the data plane some form of hierarchy is required to perform longest prefix matching. However, this is not of consideration for us as DRAGON happens in the control plane.

The same principles are true for BIRD. Prefixes and routes are stored in a hash table for fast constant time access. BIRD itself does not offer any data plane features. For packet forwarding the Linux kernel and its routing tables are used. Thus, BIRD offers no hierarchical data structures.

However, DRAGON operations heavily rely on relationship parent/child lookup operations. Filtering requires access to the children and parents of a prefix. PIA makes even more extensive use of such operations to determine the possibility for a route aggregation.

Simulating these operations on a hash table requires iterating through the whole table and comparing prefixes. This implies an $O(n)$ overhead per lookup operation with n being the amount of subnets or prefixes. Given that these lookup operations are required very frequently this imposes a huge overhead. In addition, this solution does not provide any way of iterating through a parent hierarchy.

As such, we use an additional data structure in the control plane. Specifically, we use a compact prefix tree. Using this kind of tree we can hierarchically walk children and parent prefixes as well as doing efficient $O(\log n)$ lookups.

In the prefix tree each node represents a subnet or prefix and has two children. The prefix length increases with each level in the tree. The single root node at level 0 has a prefix length of 0 (the $0.0.0.0/0$ prefix). Its two children differ in the bit of the prefix length' bit. This means that the two children of the root prefix are $0.0.0.0/1$ and $128.0.0.0/1$. Each with their prefix length or level increased by one. Further levels of the tree are then built recursively. In a prefix tree for IPv4 prefixes the maximum level or height of the tree is 32.

The plain implementation of the above tree would incur a huge memory overhead in the order of 2^{33} nodes. Thus, using a *compact* prefix tree allows to only store the prefixes that are in use (in addition to intermediate nodes) and such save memory. An instance of such a tree is depicted in Figure 3.2. Compact prefix trees can have intermediate tree nodes which are only required as a branching point. To differentiate between nodes that have been inserted and nodes that only serve as branching points we use a per-node pointer to the subnet structure in the main BIRD routing hash table. Branching nodes have this pointer set to null.

At each node we store a pointer to a subnet instance in the BIRD hash table, the prefix, the prefix length, a parent pointer and two children pointers. As explained in the last paragraph the pointer to the main subnet structure indicates whether a node really has been inserted into the tree. This backlink is also used for fast access to the routes of a subnet via lookups in the tree. As this pointer can be null we need to store the prefix and prefix length in two additional fields. The two children pointers are used for iteration to the left and right child. Using the parent pointer we can not only walk the tree downwards but also upwards. This is important for checks such as if a route can be filtered by a parent.

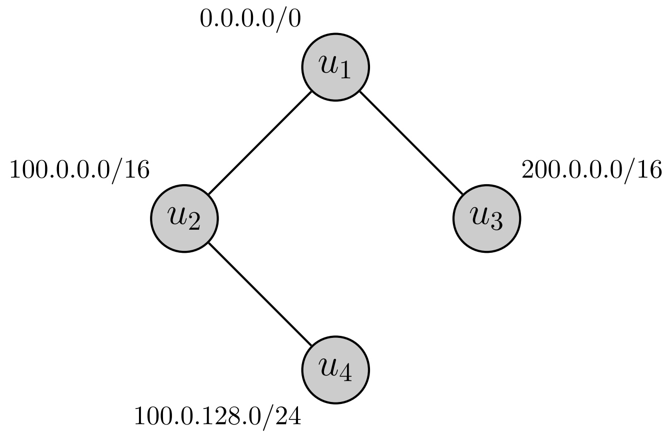

Figure 3.2: *Instantiation of a compact prefix tree. The root node has two children. Their prefix is different in the zeroth bit. Children with a zeroth bit of* 0 *go to the left while children with a zeroth bit of* 1 *go to the right. This logic is applied recursively. For example, the sixteenth bit of* $100.0.128.0/24$ *is* 1. *As such, it becomes the right child of the* $100.0.0.0/16$ *node. Being a compact tree no intermediate nodes are stored.*

Another often forgotten advantage of a parent pointer is that it allows tree traversal in constant space. Tree traversal without a parent pointer using a depth-first search (DFS) requires $O(h)$ additional stack space with h being the height of the tree.

As mentioned above, the lookup operation in the tree hierarchy happens frequently in each update message. A way to speedup the initial lookup of a subnet in the tree would be to create a backlink from the main BIRD hash table into the prefix tree. This optimization would save the $O(\log n)$ tree walk at the expense of additional memory overhead. However, some parts of the DRAGON algorithm, such as the aggregation logic, still require $O(\log n)$ tree traversals.

MRAI Timer

The DRAGON algorithm creates many unnecessary update and withdrawal messages. The reason for this is that route updates appear in an non-deterministic order. We show two examples of such cases.

For the filtering algorithm the following scenario is very common. Given are two prefixes *p* ($100.0.0.0/16$) and *q* ($100.0.0.0/24$) with the same route attributes. If *p* arrives before *q*, *q* will be filtered on arrival and the only message that is sent to neighbors is the update message of *p*. On the other hand, if *q* arrives before *p* then three messages will be sent in total. On arrival of *q* a superfluous update message for *q* will be sent. Once, *p* is received *q* can be filtered. This results in a withdrawal of *q* and an update message for *p*.

In PIA a similar scenario is common. Given two prefixes that can form an aggregation *x*. If another route can later form a shorter aggregate with *x*, a withdrawal message for *x* is required and as a consequence two unnecessary messages have been created. A similar scenario that plays together with filtering is if *x* and its two forming prefixes can later be filtered. In this case not only the two forming prefixes have to be withdrawn but also the aggregate.

In BGP there is a parameter called *MinRouteAdvertisementIntervalTimer* (MRAI) [5]. This timer guarantees that update messages to a peer will only be sent in certain intervals. BIRD does not offer MRAI by default.

Consequently, we added it on top of constructs that BIRD already provides. In our prototype we make the setting optional with a configurable timeout. Our implementation works by exploiting the BIRD concept of *sending buckets*. Whenever an update or withdrawal message is supposed to be sent to a neighbor it is enqueued into a sending bucket. If a message for a certain prefix is to be sent and the sending bucket for a prefix already contains a message then that one will be replaced. As noted above, BIRD does not use timers to flush these buckets. Instead, BIRD is event based which means that the buckets are usually flushed relatively quickly after a batch of updates has been processed. To enable MRAI we disable the creation of this flushing signal. Rather, we register a timer that fires after the configured delay and which creates the sending event. As a result, unnecessary updates and withdraws that are created during the delay time will be evicted out of the sending buckets by the latest message for a specific prefix.

Higher timeouts avoid unnecessary update messages but slow down the convergence time of BGP across the whole network. However, as part of this thesis we are concerned with the local behavior of BIRD. Consequently, we do not perform any measurements or tuning for an optimal timeout value. In addition, there is no common consent on the best MRAI values in the scientific and operator communities [9].

In general, there are four different combinations of the order in which update and withdrawal messages of a prefix can be sent. Those are listed in Table 3.1. The combination U_1W_2 indicates that the message type of U is being sent followed by a message of type W. Type W indicates a withdrawal while U stands for an update message.

Input Msgs.	Output Msgs. without MRAI	Output Msgs. with MRAI
W_1W_2	W_1W_2	W_2
W_1U_2	W_1U_2	U_2
U_1U_2	U_1U_2	U_2
U_1W_2	U_1W_2	W_2

Table 3.1: *The table shows the output for different sequences of output messages with MRAI turned on and off. W indicates a withdrawal while U indicates an update message. The index signifies the order of arrival. We see that in all combinations MRAI can filter the first message. At same time it is clear that even with MRAI at least a single message has to be sent. An optimal smart algorithm could filter the withdrawal in the U_1W_1 case if no update has ever been sent.*

Table 3.1 shows which messages are filtered by MRAI. Three of the cases are of interest to us. First, the W_1U_2 case can happen if an aggregate is created based on two forming routes. This aggregate is later being withdrawn because a shorter aggregate prefix can be formed. However, at a later point in time an update for the previously aggregated prefix is received. This update message can then replace the withdrawal.

Second, U_1U_2 happens in a similar setup. If a prefix has been aggregated and later a route for this aggregate is received the update for the aggregate can be replaced, as well. While those two scenarios can develop they are of less importance.

The most interesting case is the third case. The U_1W_2 case describes the scenario that we showed in the previous paragraphs. Routes are being announced and later withdrawn. The table shows an interesting fact about this case. The update message is not being sent but the withdrawal is. This only saves half of the messages that we unnecessarily create. The reason why MRAI can not also filter the withdrawal message is that it can not know whether there has ever been an update sent for this prefix to this peer.

Consequently, even with MRAI we send a huge amount of unnecessary withdrawal messages. A smarter system would be needed to track whether the U_1W_2 case can be filtered completely by MRAI. Nevertheless, sending withdrawals is generally cheaper than updates as no temporary route attributes have to be created, which are attached to an update message.

Alternative Message Reduction Methods

The MRAI Timer is one approach to reduce the amount of unnecessary messages. There exist other methods that could be implemented.

First, a threshold for the amount of routes that have to be aggregated can be introduced. This means that an aggregate will only be created if it consists of at least at certain number of forming routes (e.g.: 4 or 8). For such a measure it might be interesting to analyze the distribution of how many routes a PIA route covers on a real route data set.

Second, DRAGON could be paced in general. For example, this can mean to not immediately send an update if the attributes for a PIA route can be improved. Both those measures are subject to fine-tuning and further research for their efficiency.

Filtering Algorithm Implementation

The DRAGON filtering algorithm requires to keep filtered routes. Especially, these routes are not being filtered by the import filters. The concept of keeping routes in the routing table but not exporting them might not be supported by all routers.

Luckily, BIRD supports a configuration option to keep filtered routes. One actual usecase of this feature could be to later analyze which routes have been filtered by the user filters. For the DRAGON filtering algorithm we use the implementation of this concept to not export routes but still keep them in case that they later have to be unfiltered and announced.

Specifically, BIRD implements this feature by setting an attribute on each route which is checked in the export filters. If the attribute is set, routes will not be exported. A more sophisticated implementation is required, if both features are of interest.

PIA Implementation

Originating PIA Routes

Besides the classical BGP attributes, the route data type in BIRD also carries a variety of other internal attributes. These attributes are important as otherwise BIRD would not treat the route correctly.

Most of those attributes are related to where a route originates from, specifically the route source and the hardware interface. We set the local loopback interface as destination interface. From a forwarding perspective this is irrelevant as PIA routes will not get installed into the FIB at the aggregating node. The question of the route source is more interesting. Initially, we used a static protocol instance to attach our routes to. Using this approach the problem of route attributes strikes again. While it is possible to attach BGP attributes to a static route the standard BIRD logic does ignore them. The alternative of attaching PIA routes to one of the forming protocol instances does not work either. To avoid loops, BIRD checks the source of a route before sending it to a peer. As such, PIA routes would not be sent to the sources of the forming routes. As a final solution we decided to create a dummy BGP protocol instance. This dummy instance does not register itself in any of the BIRD core code. Its only task is to serve as origin for PIA routes.

Handling PIA Routes

DRAGON can generate a lot of needless routes without regulation. Two ways have been proposed to reduce the amount of PIA messages that are created [7]:

- Only create a PIA route if all the forming routes are customers routes
- If a node receives a customer route for a prefix which the node itself aggregates it shall remove its aggregate

Again, the question is how these rules can be implemented. The first rule requires general knowledge of what a customer route is and what not. As we do not have such knowledge without further configuration we can not directly implement this rule.

Nonetheless, the second rule can be modeled. Independent of the attribute class, we remove an aggregate if we receive a route for an aggregate prefix with attributes that are better or equal to the ones of our route.

This leads to another question of how the attributes of PIA routes are actually compared. The answer to this question can also depend on the individual scenario. In the case described above it make sense to compare the routes by the normal BGP decision process.

However, there are several cases where the specifics are not so clear. Taking Figure 2.3 as an example, assume that t_2 and t_3 originate routes that have a certain community set which is not set in t_1. Depending on how routes are compared in the aggregation process the intermediate aggregation prefix 101 can be created in addition to 10 or not. If communities are a factor, the extra prefix will be created. Otherwise, it will be omitted. To understand why such an intermediate aggregation route can be important the next hop has to be considered. If the filtering process at u_1 takes communities into account and the intermediate route would not be created, then u_1 could only filter the 100 prefix from t_1. This neglects any benefits of DRAGON as no routes are saved. On the other hand, a tagged 101 prefix would allow u_1 to filter the routes from t_2 and t_3 as well.

In our prototype we include communities in comparisons between two PIA routes. Nevertheless, a concrete answer requires further research and might call for a more narrow definition of how GR relationships are modeled.

Algorithmic Overview

In the following we will give a rough overview of the pseudocode of some of the core PIA algorithms. All algorithms are based on operating on the compact prefix tree.

Whenever the elected route for a prefix changes Algorithm 1 is called. It consists of two parts that we will discuss in detail. First, it checks whether there already exists a parent PIA prefix that was previously aggregated (Algorithm 5). Second, if no parent PIA prefix was found it tries to create new ones by a call to Algorithm 2.

Algorithm 1 Basic logic for PIA

```
1: function CHECKPIA(prefix)
2:     $parentExists \leftarrow$ UPDATEPIA(prefix)
3:     if not $parentExists$ then
4:         CHECKPIA(prefix)
5:     end if
6: end function
```

The task of Algorithm 2 is to check whether aggregation is possible on the arrival of a new prefix. It starts by determining the attributes of the new prefix via a call to FINDBESTATTRIBUTES. This algorithm extracts either the attributes of the prefix itself or the attributes of its children if they can form better attributes. The following paragraph explains that algorithm in detail. Given the initial attributes Algorithm 2 iterates over all the existing siblings and grand-siblings (sibling of the parent node) of the new prefix. If there is no longer a parent or a sibling it stops as a further aggregation is not possible. If a sibling exists, its attributes or the ones of its children will be extracted. The most interesting part follows. If the attributes of the sibling are worse than those of the current node, an intermediate aggregate is created before walking further up the tree. Thus, all the possible aggregates are created. In such a case the current attributes are merged with the ones of the sibling. How the attributes are merged is explained in section 3.1. Once no more parent or sibling exists the iteration stops and the final aggregate is created.

As mentioned previously, Algorithm 3 extracts the best possible aggregation attributes of a subtree. It is defined recursively. If the given prefix does not have both a left and a right child, it can not be formed from its children with possible better attributes. As such, its attributes are returned directly. On the other hand, if it has both children, the attributes of those are being calculated recursively. Finally, the children attributes are merged and compared to the attributes of the prefix itself. The better ones of those two will be returned.

Algorithm 4 is a helper algorithm that is used in many parts of the PIA logic. Given a prefix it determines whether the prefix can be aggregated. It works recursively by checking whether

Algorithm 2 Aggregation check for a new prefixes

```
function CREATEAGGREGATE(newPrefix)
    $attrs \leftarrow$ FINDBESTATTRIBUTES($newPrefix$)
    $iter \leftarrow newPrefix$
    while $iter$ do
        $sibling \leftarrow SiblingNode(iter)$
        if not $sibling$ and not CANAGGREGATE($sibling$) then break
        end if
        $siblingAttrs \leftarrow$ FINDBESTATTRIBUTES($sibling$)
        if $attrs > siblingAttrs$ then
            AGGREGATESUBNET($iter$, $attrs$)
            $attrs \leftarrow$ MERGEATTRIBUTES($attrs$, $siblingAttrs$)
        end if
        $iter \leftarrow$ PARENT($iter$)
    end while
    if $iter \neq newPrefix$ then
        AGGREGATESUBNET($iter$, $attrs$)
    end if
end function
```

Algorithm 3 Extracts the best possible attributes for a given prefix

```
function FINDBESTATTRIBUTES(prefix)
    if not (LEFTCHILD($prefix$) and RIGHTCHILD($prefix$)) then
        return EXTRACTATTRIBUTES($prefix$)
    end if
    $leftChild \leftarrow$ FINDBESTATTRIBUTES(LEFTCHILD($prefix$))
    $rightChild \leftarrow$ FINDBESTATTRIBUTES(RIGHTCHILD($prefix$))
    $childrenAttrs \leftarrow$ MERGEATTRIBUTES($rightchild$, $leftchild$)
    $prefixAttrs \leftarrow$ EXTRACTATTRIBUTES(prefix)
    if $prefixAttrs >= childrenAttrs$ then
        return $prefixAttrs$
    end if
    return $childrenAttrs$
end function
```

its children exist and can be aggregated. It starts by checking whether the two children of the prefix exist. As we are using a compact prefix tree we have to check whether the children are indeed the direct children. If they do but are an intermediate node we further recurse down.

Algorithm 4 Checks whether a certain prefix can be aggregated

```
1: function CANAGGREGATE(prefix)
2:     $leftChild \leftarrow$ LEFTCHILD($prefix$)
3:     $rightChild \leftarrow$ RIGHTCHILD($prefix$)
4:     if $leftChild$ and $rightChild$ and PXLEN($leftChild$) $==$ PXLEN($prefix$) $+1$ and
       PXLEN($rightChild$) $==$ PXLEN($prefix$) $+1$ then
5:         if ISINTERMEDIATE($leftChild$) and not CANAGGREGATE($leftChild$) then
6:             return false
7:         end if
8:         if ISINTERMEDIATE($rightChild$) and not CANAGGREGATE($rightChild$) then
9:             return false
10:        end if
11:        return true
12:    end if
13:    return false
14: end function
```

Finally, algorithm 5 updates existing PIA prefixes if they exist. It works by iterating over all the parents of the given prefix. If a parent that is a PIA aggregate is found its attributes are extracted. In addition, the attributes of the parent's subtree are calculated. If those attributes are better, because they got improved by the new route, the parent's old attributes are being updated. In that case the algorithm returns *true* indicating that a parent has been found which means that no further aggregation logic needs to run. On the other hand, if no parent was found then *false* is returned to trigger the normal aggregation check.

Algorithm 5 Updates the attributes for a given PIA prefix

```
1: function UPDATEPIA(prefix)
2:     $parentExists \leftarrow false$
3:     $iter \leftarrow$ PARENT(prefix)
4:     while $iter$ do
5:         if ISPIAAGGREGATE($iter$) then
6:             $newAttrs \leftarrow$ FINDBESTATTRIBUTES($iter$)
7:             $oldAttrs \leftarrow$ EXTRACTATTRIBUTES($iter$)
8:             if $newAttrs > oldAttrs$ then
9:                 UPDATEANDANNOUNCEATTRS($iter$)
10:                return
11:            end if
12:        end if
13:        $iter \leftarrow$ PARENT($iter$)
14:        $parentExists \leftarrow true$
15:    end while
16:    return $parentExists$
17: end function
```

Last but not least, the PIA algorithm also needs to handle the case in which a child prefix from a PIA prefix is being withdrawn. We do not show our algorithms here for brevity. This part is the most complex one. It consists of many tricky edge cases which can be the source for bugs.

Two main cases exist. First is the scenario in which a route for a prefix is withdrawn for which no alternative route exists. In such a case, checks are needed whether the lost prefix can logically be replaced by further children. In addition, the question is whether the attributes can equally well be replaced. If that can not be done the prefix has to either be withdrawn in the former case or an update for the new attributes has to be sent in the later.

Second, the case in which an alternative route for the withdrawn prefix exists. Similarly to the previous case, it has to be guaranteed that the attributes of the remaining prefixes can form the attributes of the parent PIA prefix. An update to the parent prefix is needed if its attributes can not be maintained.

In both cases, if the attributes can not be maintained and the parent prefix has to be updated respectively withdrawn, possible sibling PIA prefixes have to be taken into account. The reason is that the sibling can possibly be aggregated with the former attributes of the parent.

Finally, it shall be noted that two possible ways exist to implement PIA. First, is the approach of doing a full check of a parent subtree on a route change. That includes calculating a PIA prefix candidate set which is then compared to the existing PIA prefixes.

The alternative solution that we decided to take for our prototype is the differential approach. In this way, on an update or withdraw message we try to analyze which change is required without calculating every single PIA prefix again. This has the advantage of offering better performance. On the other hand, it is more complex to implement. As mentioned before, this becomes especially tricky in the deaggregation part.

3.3 DRAGON in iBGP

Intra-AS DRAGON

So far we have treated an AS as a single router running BIRD. However, in reality an AS is a network itself and consists of many routers. The routers at the edge of the AS, the ones that interconnect multiple ASes with each other, are called *edge routers*.

The theory behind DRAGON covers this scenario as well. However, not much research has been done so far on practical consideration of DRAGON in iBGP. As part of this thesis we ran a first test based on our implementation. Nevertheless, this area needs to be a core aspect of further research as it is essential to the deployment of DRAGON.

The test setup is depicted in Figure 3.3. We modeled an AS based on three border routers. These three routers are connected to each other. The AS has three connecting links. First, a peer or provider link to another AS. Second, two customer links on which it learns the prefixes *p* and *q*. Prefix *p* is a parent prefix of *q*. The AS can be thought of as node u_1 in Figure 2.1.

Everything behaves as expected. The *q* prefix is filtered by the border routers that receive *q* (r_2, r_3) based on *p*, which is propagated to all routers in the AS. *q* reaches none of the FIBs. Traffic to *q* and *p* is being routed over router r_3.

We noticed one thing while executing the test. Similarly to previous matters, the way attributes are handled and propagated is important. We keep the internal links neutral. This means that we do not set any local pref or communities on the internal filters. Otherwise, routes that are coming from a provider border router might be treated as customers. On the other hand, it is important that the border routers set the attributes appropriately and that those get transported to the other border routers. However, the later part is not a problem as the BGP local pref will be attached to routes on iBGP connections (communities are attached either way).

Traffic engineering

Traffic Engineering is the discipline of routing traffic through a determined path to evenly distribute load. Thereby avoiding congestion and increasing performance.

Common scenarios are in the context of multi-homing. ASes that have more than one outgoing connection might want to spread their load across multiple exits.

BGP does not have any concept that would support such load-balancing scenarios, by default. There exists a concept called *Multipath BGP*. This feature is offered by several router

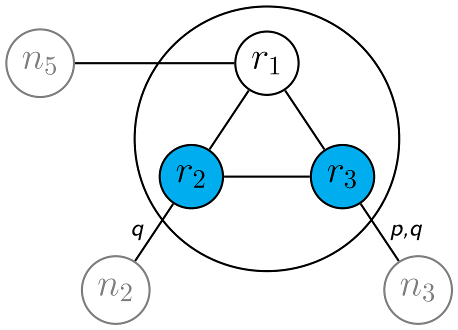

Figure 3.3: *Zoom into node u_1 from Figure 2.1. An AS represented by three borders routers each running DRAGON. Two prefixes p and q are being learned on customer links. p is a parent prefix of q. Blue routers can filter q and do not install routes in their FIB. However, they keep the routes in their RIB. White routers are oblivious to q and do neither have a route in their FIB nor RIB.*

vendors but is not part of any BGP standard. If multiple outgoing links exist, Multipath BGP will install multiple routes into the FIB. Thus, traffic can leave the router on several links.

While this load-balances outgoing traffic, incoming traffic is still not managed. To control how traffic reaches an AS a different technique can be used. Assuming an AS originates a $/16$ prefix and has two outgoing links. It can then split up this prefix into two longer $/17$ prefixes. It will then announce the $/16$ prefix on both links in addition to one of the $/17$ prefixes on each link.

As a consequence, traffic will be routed via two different paths according to longest prefix matching. Naturally, this does not automatically guarantee that the traffic on both links is totally balanced. In practice, operators might split up their prefixes according to load metrics.

However, this principle leads to a conflict with DRAGON. The conflict is already clear from a principle point of view. DRAGON tries to *reduce* the amount of prefixes. The described traffic engineering technique tries to balance load by *increasing* the number of routes and prefixes.

Imagine the topology from Figure 2.1. Assuming the scenario from above with u_4 originating a $/16$ prefix. In addition, it sends a $/17$ child prefix to u_2 and u_3 respectively. Once those arrive at node u_1 they will be filtered and only the shorter $/16$ prefix will be installed into the FIB. As a consequence, all traffic will take one of the two paths and no load balancing is happening.

There exists another traffic engineering related problem specifically related to iBGP. Assuming *q* is a high traffic prefix (think Netflix) in Figure 3.3. According to how DRAGON works, as explained in the previous section, all traffic directed to *q* will leave the AS via router r_3. This has a subtle consequence. All the traffic that arrives at router r_2 now traverses the AS. Given that such an AS could be represented by routers on the east and west coast in the US the issues are amplified. The result is a higher congestion in ones own network, which results in higher cost. In addition, latency is increased. Without DRAGON, the traffic would not enter the network and leave directly at router r_2.

Clearly, DRAGON would not be greeted in such a situation. Therefore, a solution might be to add some form of configuration option that signals that a route should only be filtered on the same router. Another way of doing this would be to differentiate between iBGP and eBGP origins. In such a case, if eBGP is preferred over iBGP, r_2 would not filter *q* as it receives *q* over an eBGP link while *p* is coming over iBGP.

Chapter 4

Results

The main task of this thesis is to evaluate the feasibility of a DRAGON implementation. Besides technical limitations related to the nature of BGP, the most important factor is the overhead that is incurred by the additional logic that needs to be executed as part of the DRAGON algorithm. The detailed additional steps have been shown in the previous sections. In this chapter we want to show how DRAGON affects the local performance of a router.

4.1 Simulation Results

Another interesting metric is the amount of additional BGP messages that DRAGON generates in comparison to a standard BGP router. While the amount of messages does have an impact on the local router performance it is more interesting to look at it at a global scale and compare the total message count in a network of several BGP routers.

This was recently tested in an article which simulated a DRAGON deployment in different scenarios in comparison to plain BGP [7]. Several interesting results were shown.

Specifically, the per-link message count does not increase significantly. In addition, the total number of messages for a DRAGON deployment in the whole network was decreased by an order of magnitude. The reason is that filtering confines how far children routes are being propagated. Further, PIA routes are not spread too far according to the same principle. The behavior on link failure was also specifically tested. It was shown that on link failure the number of messages can be reduced to up to a half. On the other hand, DRAGON creates more messages on link failures if the special logic to deaggregate a prefix is required which was the case for about five percent of the prefixes.

Overall, the global amount of messages is reduced. As a result, DRAGON can globally improve routing performance as less routes need to be processed per router even though DRAGON incurs a per-route overhead.

4.2 Experiments Overview

Our main focus for the experiments lies on measuring CPU and memory overhead.

We conduct several experiments to measure the performance in different scenarios. The scenarios are described on a per test basis. In general, we look at the point at which a router is coming up and is being fed a burst of prefix update messages. This point is the most interesting as the router is exposed to the most load.

In general, we compare three different BGP implementations. First, a plain unmodified version of BIRD (1.5). Second, our implementation of DRAGON on top of BIRD with only using the filtering algorithm to analyze the individual impact of the two parts of the DRAGON algorithm. Third, a full version of DRAGON combining both filtering and PIA.

Tests are divided in two categories. On the one side, we use an unmodified version of BIRD as a feeder instance that bursts a full routing table to our test instance. For some of the tests the instance under test is then connected to another unmodified BIRD instance. This is done in tests in which we want to examine the overhead of sending routes. On the other side, we use a modified version of the above setup to test certain individual parts of the system.

Finally, we use RIPE raw routing table dumps as test data to simulate a realistic load [10]. Again, for some of the more specialized experiments we crafted a special set of routes. The following section will describe each scenario individually and explain the results.

As a test platform we use physical off-the-shelf servers. Specifically, the servers are equipped with Intel Ivy Bridge CPUs [11]. All test instances run on a different host and are connected via a commodity switch. The links have a bandwidth of $1Gbps$.

4.3 Results

In the following sections we will present the results of the tests that we performed. Our prototype is not optimized. Consequently, an optimized implementation could further reduce processing overhead.

It shall be mentioned that we test the scenario in which a DRAGON router has non-DRAGON neighbor(s) and receives a full routing table. However, a high deployment rate of DRAGON also implicitly improves local performance as less prefixes and routes reach each individual router.

4.3.1 Processing Overhead

To test the overhead that is caused by the additional processing which is required by DRAGON we feed our test instance with 470k routes and wait until they have been processed. In this test, we are only interested in the pure processing overhead and as such the routes will not be forwarded to another router.

Figure 4.1 shows the results of this benchmark. We take the unmodified BIRD version as a baseline and compare it with filtering only and full DRAGON. We see that with only filtering enabled we take a bit more than 80% of the time of an unmodified version. Full DRAGON increases that to about 90%.

Analyzing this we see that DRAGON does in fact grant a performance improvement. This might be counter-intuitive at first. One might expect that the additional processing incurs an overhead. As expected the PIA adds additional overhead to the filtering. To better understand those results we look into further benchmarks.

4.3.2 Filter Rate Scaling

Profiling shows that most of the time is spent in functions that are related to updating the Linux FIB.

To confirm this we run a modified version of the benchmark. We compare three variations with differing percentages of routes that are being filtered. Specifically, we test ratios of 20, 50 and 80 percent of the routes being filtered. To test those ratios we create a custom set of routes for each test that results in the required percentages. We only run with filtering enabled to get a clearer picture. Results are shown in figure 4.2.

We see that the higher the filter rate the faster DRAGON performs in comparison to the plain BGP implementation. Note that the underlying benchmark for all three cases is different so one can not directly speak of scaling characteristics.

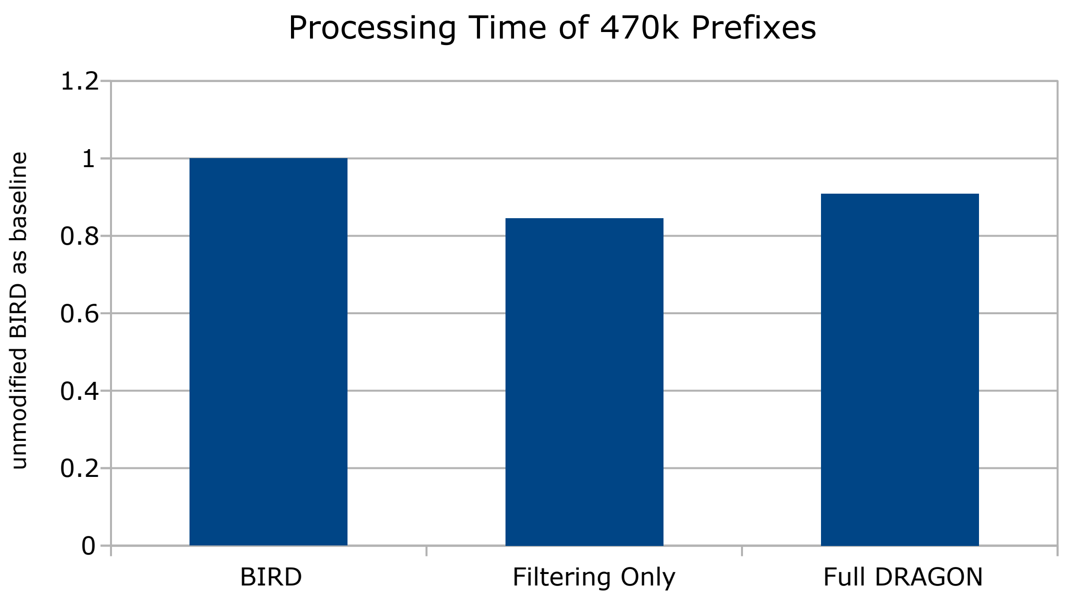

Figure 4.1: *Comparison of the processing time of 470k routes of an unmodified BIRD instance with a filtering-only DRAGON implementation and a full DRAGON version. We see that DRAGON is on a level with a plain BGP implementation. The reason is that due to the filtering in DRAGON less routes have to be pushed to the FIB. The RIB to FIB path is a common bottleneck in routers. This advantage makes up for the additional processing that is required.*

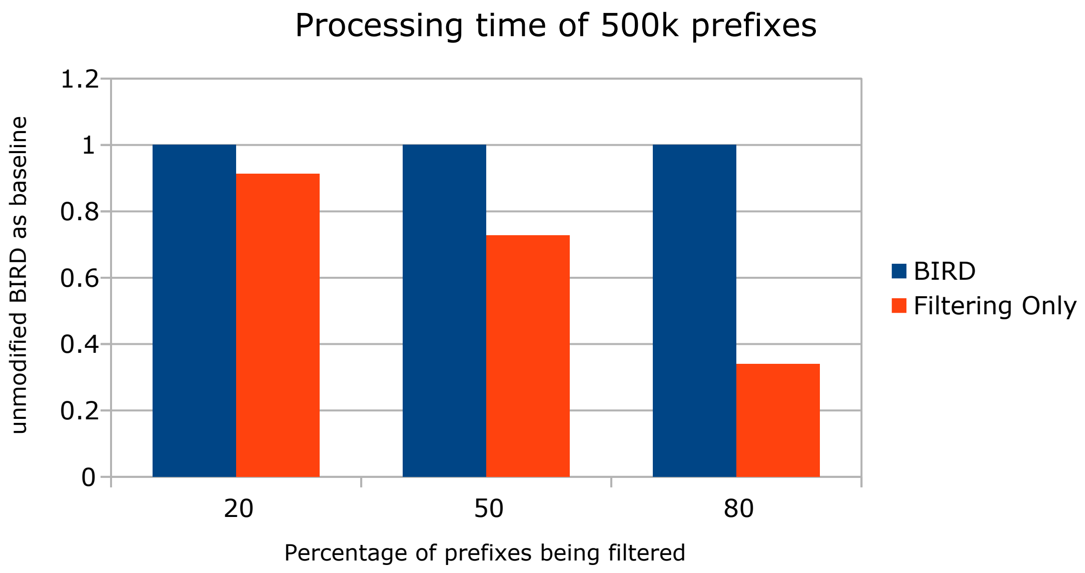

Figure 4.2: *We compare the processing time of 500k prefixes of an unmodified BIRD instance with our DRAGON implementation (filtering-only) with varying degrees of the filtering rate. We see that the more prefixes that can be filtered the faster DRAGON performs. This confirms the results that the RIB to FIB pipeline is one of the major bottlenecks.*

This shows that the bottleneck in the system is indeed the pipe to the FIB and that DRAGON can reduce the strain on this pipe. While we are testing on an off-the-shelf commodity server other papers report that for hardware routers the connection between the data and the routing plane is a general bottleneck as well [12].

Comparing Figure 4.1 to 4.2 we see that the test on real data falls somewhere between the

20 and 50 percent mark. However, on the real data a bit more than 50% of the routes are being filtered which should actually fall into the 50 to 80 percent range in figure 4.2. The reason for that is that in the real system routes are being announced in a non deterministic order. This is in contrast to the benchmark in 4.2 in which we specifically crafted the arrival of the routes. Thus, cases exist in which routes, that are actually being filtered, are first announced and later have to be withdrawn. These operations cause writes to the FIB which degrades performance.

4.3.3 Per Prefix Overhead

Seeing that the speedup in DRAGON comes from less updates to the kernel FIB, it is also interesting to see from which parts slowdowns originate. To analyze the per prefix overhead that DRAGON incurs we micro-benchmark the time it takes to process a route for a new prefix and which part of that is due to DRAGON processing.

We measure the average prefix processing time in steps of 50k prefixes. This allows us to see how the processing overhead grows with an increasing number of prefix in our additional datastructures.

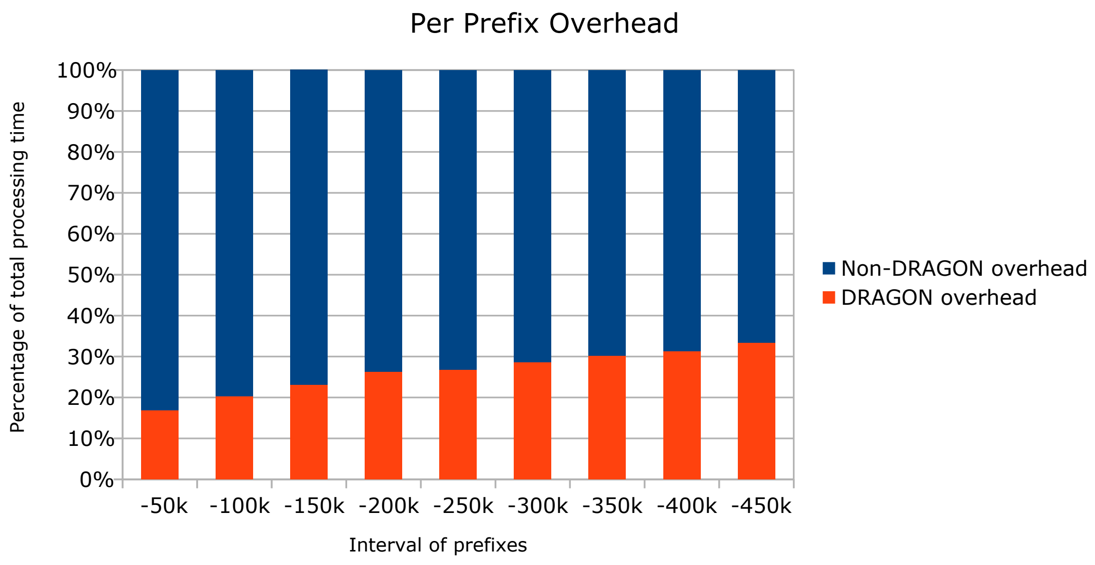

Figure 4.3: *The contribution of DRAGON and non-DRAGON related processing to the average per prefix processing time in different intervals. We measure the average processing time in intervals of 50k prefixes of a total of 450k prefixes. We see that with an increasing number of prefixes in the RIB the overhead that is caused by DRAGON grows. The reason is that the compact prefix tree grows and such makes tree walk operations more expensive.*

Figure 4.3 shows the result of this benchmark. We see the expected results. The per-prefix overhead increases with the amount of prefixes that are already in our prefix tree. Inserting the last 50k of prefixes incurs double the overhead as the first 50k routes. Note, that this shows the ratios in the full DRAGON version meaning that the ratio taken by DRAGON processing in an unmodified version of BIRD would be lower.

Combining the results of the previous sections we can see that DRAGON induces a per prefix processing overhead but overall DRAGON can still outperform a normal BGP implementation on real data due to the filtering of prefixes.

4.3.4 Processing/Sending Overhead

In addition to analyzing the overhead of processing DRAGON logic, we also want to show the overhead that is created by the additional routes that are announced or withdrawn.

As described in the overview, we add another unmodified BIRD instance at the end of the pipeline. This instance will serve as a receiver node which will receive the routes that our test instance generates. From that point on we run the same scenario as in the processing test. The pipeline then looks as follows. The feeder instance sends 470k prefixes of real route data to the instance under test which will execute the plain BGP algorithm, only the filtering code or full DRAGON and send the generated routes further to the receiver instance. Results are shown in Figure 4.4.

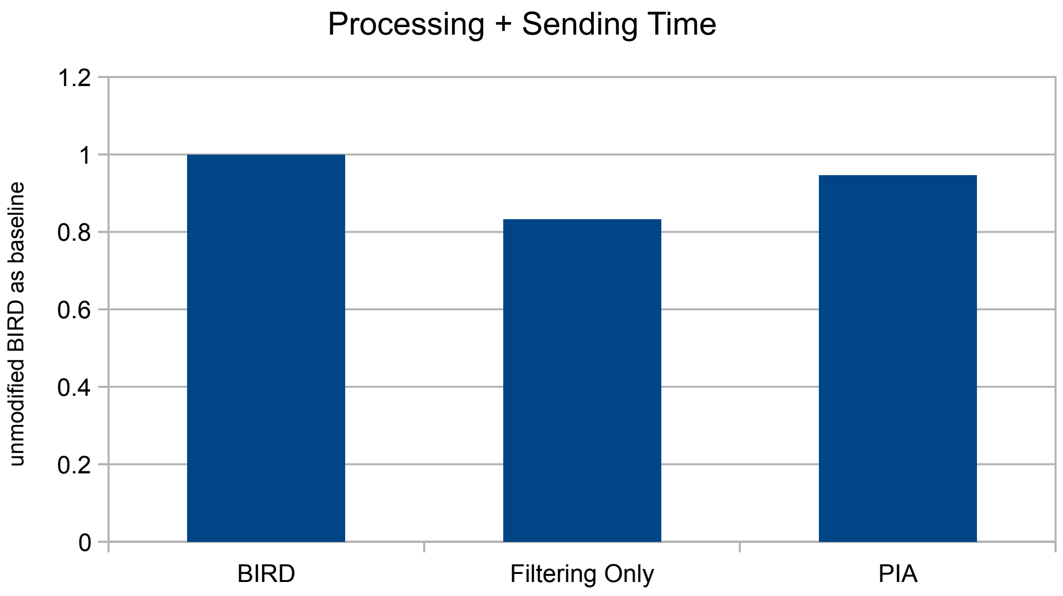

Figure 4.4: *Comparison of the processing and sending time of 470k routes of an unmodified BIRD instance with a filtering-only DRAGON implementation and a full DRAGON version. We extend the test in Figure 4.1 by sending the prefixes that we process to another peer. DRAGON is still on a level with a plain BGP implementation. The difference to Figure 4.1 is only minimal as the overhead of sending messages is very low on modern servers.*

Results are similar to the processing-only tests (Figure 4.1). Filtering-only gained a little speed compared to unmodified BIRD while full DRAGON lost a bit. Looking at the raw data in Table A.4 this can be explained. We see that the overall execution time has increased slightly. This indicates that sending routes has only a minimal overhead. Filtering wins a little due to not having to send as many updates as unmodified BIRD. However, it shall be noted that it has to send withdrawals that mostly neglect any win from sending less updates. In addition, DRAGON sends updates for prefixes that it might later withdraw. This is the same pattern as explained in the processing section. Full DRAGON's speedup declined in comparison to 4.1 as the PIA algorithm sends additional updates and withdrawals for aggregated routes.

While we see that sending routes has no big overhead in BIRD this might not be true for hardware routers. BIRD benefits from fast CPUs and NICs on modern servers. In addition, the TCP implementation in Linux can be considered state of the art. Hardware routers might play conservative here and use older implementations of TCP that have proven to work reliable but can not offer the performance that Linux provides. Analysis of such TCP implementations has been done in the past [13]. In such cases the additional routes that DRAGON creates, especially aggregation updates and withdrawals from PIA, can have a bigger negative impact than we see in the BIRD implementation.

To evaluate high latency and low bandwidth scenarios we do specific tests in which we reduce the bandwidth of the link on which we send out the route updates. In addition, we introduce latency to model longer links.

To introduce these handicaps we use a tool called *traffic control* (tc). tc is part of a collection of Linux command line utilities (*iproute2*) which manage the Linux network stack. It allows to apply traffic shaping to network interfaces. While its primary usecase is to improve network performance by tuning certain parameters (queuing policies), that might improve the performance of an application, it can also be used to degrade performance [14].

We test the impact of latency and bandwidth in two stages. First, we introduce a bidirectional delay of $100ms$. This means that each direction has an additional $100ms$ latency. As a consequence, the total round-trip time (RTT) equals $200ms$. We choose this value as it is typical for transcontinental connections (e.g: Europe to North America).

In a second stage, we reduce the bandwidth of the link to one Megabit per second ($1Mbps$). This is three orders of magnitude less than the bandwidth of the unmodified interface. As such, it is a very pessimistic assumption of the bandwidth.

We test in the exact same way as we do in the non-handicap version. Results of those the tests are depicted in Figure 4.5 and Figure 4.6.

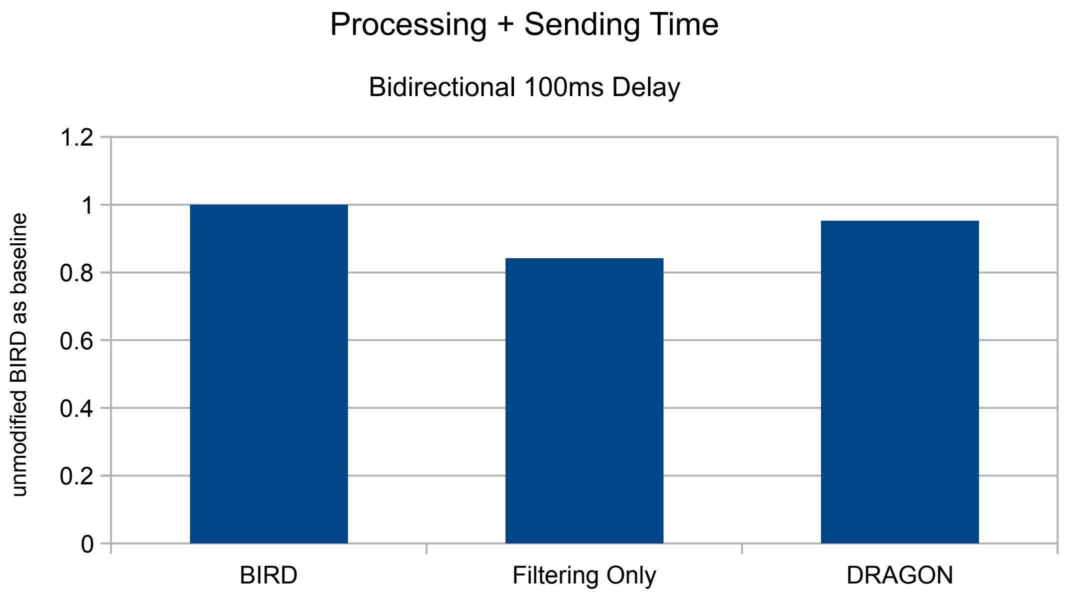

Figure 4.5: *Same test as in Figure 4.4 with a $100ms$ delay added on both the receiver and sender side resulting in a $200ms$ RTT. The results are only minimally impacted as the sending pattern is mostly one directional and once TCP has slow-started it can send at full speed.*

Comparing Figure 4.5 with the non-handicapped results in Figure 4.4 we see that the difference is only minimal. Compared to plain BIRD the full DRAGON implementation loses a bit. The reason is again the same that DRAGON sends a higher amount of messages. Filtering-only, which sends less messages than plain BIRD, gains a small speedup.

All in all, the reason that the results are not that much different to 4.4 is that TCP can easily cope with such a delay as the receiving side is mostly only sending acknowledgments. Specifically, once the instance under test (sending side) has enlarged its TCP sending window it can sustain the bandwidth which is required to avoid making the network a limiting factor.

Looking at the results in Figure 4.6 shows that the gap between filtering-only and full DRAGON widens. At the same time, full DRAGON is now slightly slower than plain BIRD. A glance at the raw data in Table A.6 depicts that the absolute execution time is about fifty percent slower than without the reduced bandwidth.

The two tests showed that DRAGON is relatively not too much impacted by the lower bandwidth and higher latency. However, it is not a perfect simulation of the TCP implementation of

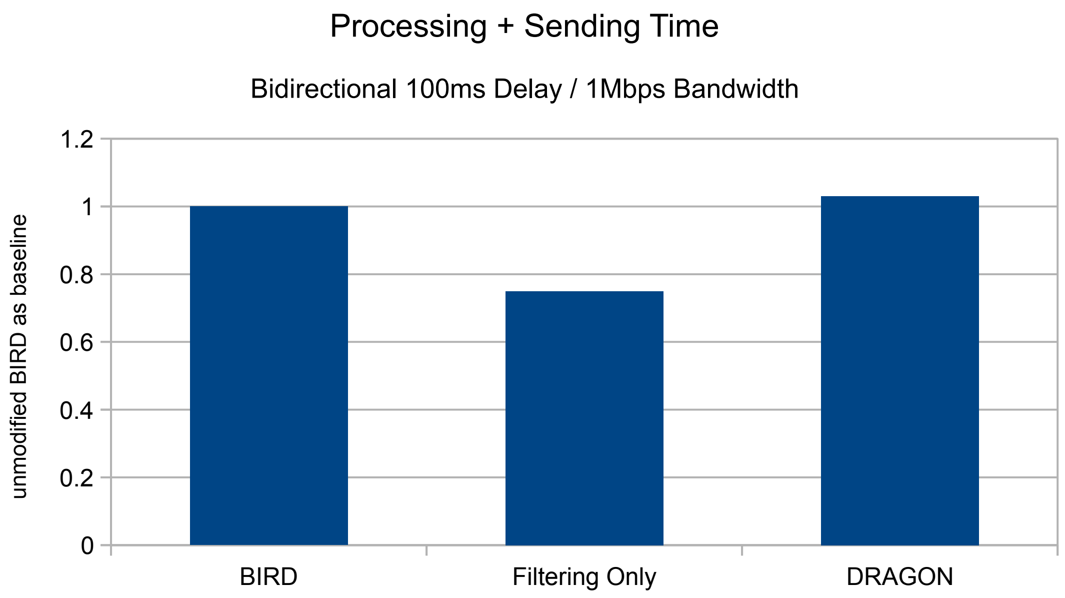

Figure 4.6: *Same test as in Figure 4.4 with a $100ms$ added on both the receiver and sender side resulting in a $200ms$ RTT. In addition, the bandwidth on both ends is limited to $1Mbps$. Full DRAGON is impacted slightly more than plain BIRD and loses its advantage.*

slower routers. While those implementations might even have higher bandwidth than what we tested they might differ in details such as TCP slow start. These specifics could have a higher impact but can only be precisely estimated by a real implementation.

4.3.5 Additional Update/Withdraw Count

To get a feeling for what the maximum additional overhead from a real route burst could be we look at the count of additional updates and withdrawals that are being created.

We order the prefixes in the dataset by prefix length in decreasing order. This means that prefixes with a longer prefix length will be announced first. In consequence, many unnecessary updates and withdrawals for prefixes that can later be filtered by a shorter prefix are created.

As depicted in Table A.7 filtering-only creates an additional 225k withdrawals. Full DRAGON creates 186k additional updates and 340k withdrawals. Adding up updates and withdrawals shows that in the worst case DRAGON doubles the message count.

Unfortunately, we can not undertake any performance tests about this setup. The reason is that to the best of our knowledge there is no tool that can create BGP update messages at a rate that saturates BIRD on a modern server. To do the message count tests we use a tool called *bgpsimple* [15]. It is a script written in Perl that takes a list of routes and feeds those to a BGP peer. However, it is very slow at doing that. Such, we can not saturate BIRD and use it for performance testing.

4.3.6 Memory Usage

Finally, we also want to know the memory overhead that our DRAGON implementation incurs. Again, we use the real data set of 470k prefixes and fill both an unmodified BIRD instance in addition to our DRAGON implementation. In addition, we measure the memory usage with a different number of routes per prefix.

Figure 4.7 shows the results of the tests. We see that the memory usage grew with DRAGON by about eighty percent if a single route per prefix is used. This is expected due to the additional

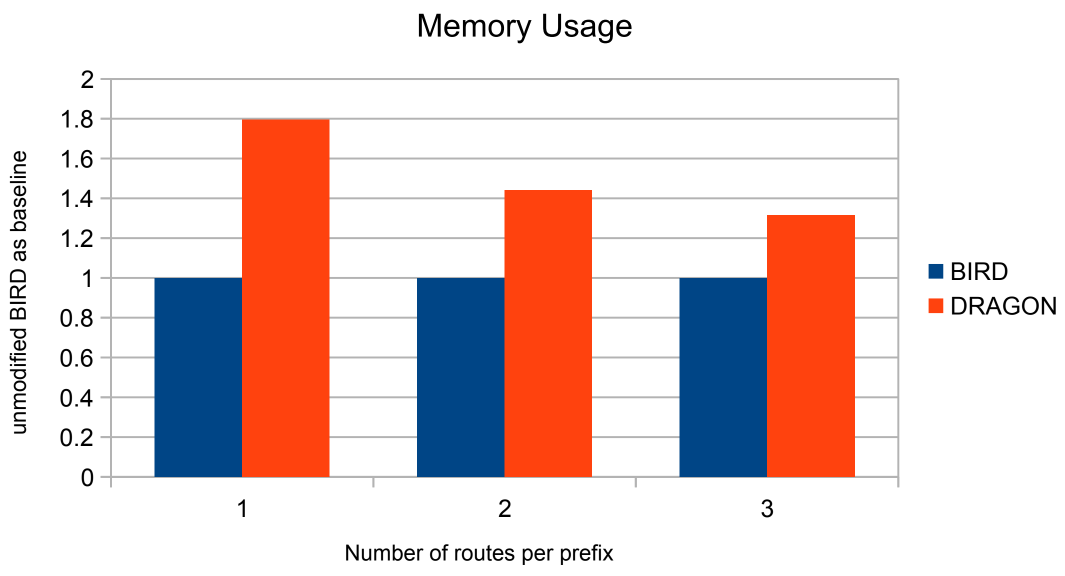

Figure 4.7: *Comparison of an unmodified BIRD version with a full DRAGON implementation. We feed 470k prefixes with a single route to both test instances. We see that DRAGON almost doubles the memory usage for a single route per prefix. The reason is that every prefix creates at least one additional node (and maybe more intermediate nodes) in our compact prefix tree. With an increasing count of routes per-prefix the DRAGON overhead becomes negligible.*

data structure that we use. Every node in our compact prefix tree takes a certain amount of memory. In addition, we also need to store intermediate nodes which increase the memory footprint.

Moreover, we see that the relative overhead of DRAGON becomes less relevant the more routes that exist per prefix in the RIB. This is due to DRAGON scaling with the number of prefixes and not with the number of routes. The reason is that the nodes in our compact prefix tree link back to the subnet structure in the main BIRD RIB.

Looking at the raw data in Table A.8 we see that the memory usage grew from around 100MB to 180MB. Both of these numbers are hardly a problem for modern servers. In addition, we see that the per prefix overhead is of the same magnitude as adding an additional route for a prefix.

Further, if the optimization of adding a backlink from the main BIRD RIB to our prefix tree was used that would also only scale with the number of prefixes. However, the order of additional memory overhead would be less than what we see here as it would only require a single additional pointer per prefix and no pointers to the intermediate nodes in the prefix tree are needed.

It shall be noted that there is no real differentiation between RIB and FIB memory for BIRD. However, it can be said that the increase in memory happens in the RIB space as that is where the prefix tree resides. The whole goal of DRAGON is to reduce the amount of required FIB memory (i.e., reducing the amount of prefixes in the FIB).

Chapter 5

Summary and Further Work

Implementing the DRAGON prototype we showed that DRAGON can definitely work in real systems without a show-stopping overhead. At the same time, we present some open problems associated with DRAGON and implementation difficulties.

Measuring and analyzing the performance of the implementation we see interesting results. On off-the-shelf servers our implementation does not incur any big net overhead compared to a plain BGP implementation. Two components contribute to the processing time of a prefix update. First, is the processing that is required by the BGP decision process and in our case because of DRAGON. Second, the time it takes to install a prefix or route from the RIB into the FIB. The later generally takes the bigger chunk of time. Because of the filtering algorithm the amount of times a route is being installed in the FIB is reduced. This performance improvement makes a filtering-only DRAGON implementation even faster than a plain BGP implementation and covers for the additional overhead that we induce with DRAGON.

Another pattern that we showed is that the order of arrival of prefixes is a key aspect in how much overhead DRAGON incurs. In addition, given that our prototype is unoptimized there is space to further reduce the processing overhead or to implement additional features without degrading the performance.

An aspect that we did not mention, yet, is the question of implementation complexity. While DRAGON comes over as very simple and straight forward in the beginning there are several edge cases that make the implementation quite tricky. An example of this is the implementation of Rule RA and the deaggregation that follows it. Further, DRAGON introduces the new paradigm of hierarchy which traditional BGP does not feature. This requires a new data structure and makes different prefixes interact with each other. In plain BGP this is not the case.

Another interesting aspect is how the GR routing model is transfered to BGP. We showed possible ways of how to implement the policies and problems that arise with them. Specifically, problems related to static routes in combination with Rule RA. In addition, we explain how attributes of PIA routes are formed. However, there still exist open problems such as the differentiation between link down events and route withdrawals on neighbors that are not directly connected. Another open problem is how the AS path of PIA routes can be better modeled.

In addition, a standard is required that defines BGP specific operations, such as which attributes are used for filtering or for creating and comparing PIA attributes. Consolidating these aspects guarantees that implementations of different vendors play well together. Otherwise, DRAGON can have a negative impact on the network.

A goal of our initial prototype is to implement DRAGON in BIRD on top of standard BGP without introducing any kind of special configuration. We want to test whether DRAGON can be implemented without enforcing any rules on route attributes or other factors. A question for further implementations is whether this requirement can be lifted. For example, requiring a low administrative preference for locally configured static routes that refer to routes of ones' neighbors could simplify problems related to static routes. The general pattern is that the more restrictions on attributes and configuration scenarios are enforced the more edge cases can possibly be removed and such permit easier and less error-prone implementations.

A further question is how security aspects are incorporated into DRAGON. *Resource Public Key Infrastructure* allows ASes to authenticate the prefixes and routes they announce [16, 17, 18]. In this regard the question is how PIA aggregation can work in such a scenario.

Finally, another question is how operators will like the concept of PIA aggregation. What PIA aggregation is doing at its core is to hijack the IP space of other ASes. Not everybody might welcome this fact.

All in all, we see that DRAGON can be considered a possible candidate for future experimental router features. At the same time, there are still unsolved problems and implementation hurdles for production implementations.

Appendix A

Experiment Data

Processing Overhead: Figure 4.1

Test	Mean	stdev
BIRD	6.8	0.25
Filtering Only	5.75	0.16
DRAGON	6.18	0.17

Table A.1: *Times in seconds.*

Filter Rate Scaling: Figure 4.2

Percentage Filtered	BIRD Mean	BIRD stdev	FILTER Mean	FILTER stdev
20	5.8	0.1	5.3	0.13
50	5.5	0.15	4	0.1
80	5.3	0.1	1.8	0.1

Table A.2: *Times in seconds.*

Per Prefix Overhead: Figure 4.3

Prefix Interval	BIRD Mean	BIRD stdev	DRAGON Mean	DRAGON stdev
-50k	13380	899	2268	140
-100k	13465	609	2737	340
-150k	13456	455	3104	409
-200k	12669	488	3339	395
-250k	12664	471	3391	276
-300k	12211	831	3493	168
-350k	12030	414	3636	156
-400k	11794	399	3689	258
-450k	12120	637	4054	158

Table A.3: *Times in nanoseconds*

Processing/Sending Overhead: Figure 4.4

Test	Mean	stdev
BIRD	7.4	0.1
Filtering Only	6.16	0.1
DRAGON	7	0.1

Table A.4: *Times in seconds*

Processing/Sending Overhead with 100ms Latency: Figure 4.5

Test	Mean	stdev
BIRD	7.6	0.2
Filtering Only	6.4	0.2
DRAGON	7.2	0.2

Table A.5: *Times in seconds*

Processing/Sending Overhead with 100ms Latency and 1Mbps Bandwidth: Figure 4.6

Test	Mean	stdev
BIRD	10	0.3
Filtering Only	7.5	0.3
DRAGON	10.3	0.2

Table A.6: *Times in seconds*

Additional Update/Withdraw Count

Test	Routes Created	Routes Withdrawn
Filtering Only	0	225000
DRAGON	186051	340191

Table A.7: *Message Count*

Memory Usage: Figure 4.7

Routes per Prefix	BIRD	DRAGON
1	102	183
2	184	265
3	264	347

Table A.8: *Memory usage in Megabytes*

Bibliography

[1] R. Lemos. Internet routers hitting 512k limit, some become unreliable. `http://arstechnica.com/security/2014/08/internet-routers-hitting-512k-limit-some-become-unreliable/`, 2014. Accessed: 16.2.2016.

[2] J. L. Sobrinho, L. Vanbever, F. Le, and J. Rexford. Distributed route aggregation on the global network. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 161–172, New York, NY, USA, 2014. ACM.

[3] J. L. Sobrinho. An algebraic theory of dynamic network routing. *IEEE/ACM Trans. Netw.*, 13(5):1160–1173, October 2005.

[4] L. Gao and J. Rexford. Stable internet routing without global coordination. *IEEE/ACM Trans. Netw.*, 9(6):681–692, December 2001.

[5] Y. Rekhter, T. Li, and S. Hares. A border gateway protocol 4 (bgp-4). `https://tools.ietf.org/html/rfc4271`, 2006. Accessed: 18.2.2016.

[6] O. Filip, M. Mares, and O. Zajicek. The bird internet routing daemon. `http://bird.network.cz/`, 2015. Accessed: 16.2.2016.

[7] A. Sousa and J. L. Sobrinho. Dynamic performance of distributed route aggregation for the internet, 2015.

[8] W. Kumari and K. Sriram. Deprecation of as_set and as_confed_set in bgp. `https://tools.ietf.org/html/rfc6472`, 2011. Accessed: 8.2.2016.

[9] P. Jakma. Revisions to the bgp minimum route advertisement interval. `https://tools.ietf.org/html/draft-ietf-idr-mrai-dep-04`, 2011. Accessed: 22.2.2016.

[10] Ripe NCC. Ris raw data. `https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data`, 2015. Accessed: 15.2.2016.

[11] Intel. Intel® xeon® processor e5-1620 v2. `http://ark.intel.com/products/75779/Intel-Xeon-Processor-E5-1620-v2-10M-Cache-3_70-GHz`, 2013. Accessed: 25.2.2016.

[12] M. Alan Chang, T. Holterbach, M. Happe, and L. Vanbever. Supercharge me: Boost router convergence with sdn. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 341–342, New York, NY, USA, 2015. ACM.

[13] R. Bush, M. Allman, K. Patel, B. V. Venkatachalapathy, M. Pandey, V. Paxson, C. Pelsser, and E. Kern. Tcp behavior of bgp. `https://archive.psg.com/121009.nag-bgp-tcp.pdf`, 2012. Accessed: 16.2.2016.

[14] The Linux Foundation. Iproute2 is a collection of utilities for controlling tcp / ip networking and traffic control in linux. `http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2`, 2015. Accessed: 24.2.2016.

[15] S. McIntyre. simple bgp peering and route injection script. https://code.google.com/archive/p/bgpsimple/, 2011. Accessed: 15.2.2016.

[16] M. Lepinski. Bgpsec protocol specification. https://tools.ietf.org/html/draft-ietf-sidr-bgpsec-protocol-14, 2015. Accessed: 15.2.2016.

[17] M. Lepinski and S. Kent. An infrastructure to support secure internet routing. https://tools.ietf.org/html/rfc6480, 2012. Accessed: 15.2.2016.

[18] R. Bush, R. Austein, K. Patel, H Gredler, and M. Waehlisch. Resource public key infrastructure (rpki) router implementation report. https://tools.ietf.org/html/rfc6472, 2011. Accessed: 15.2.2016.