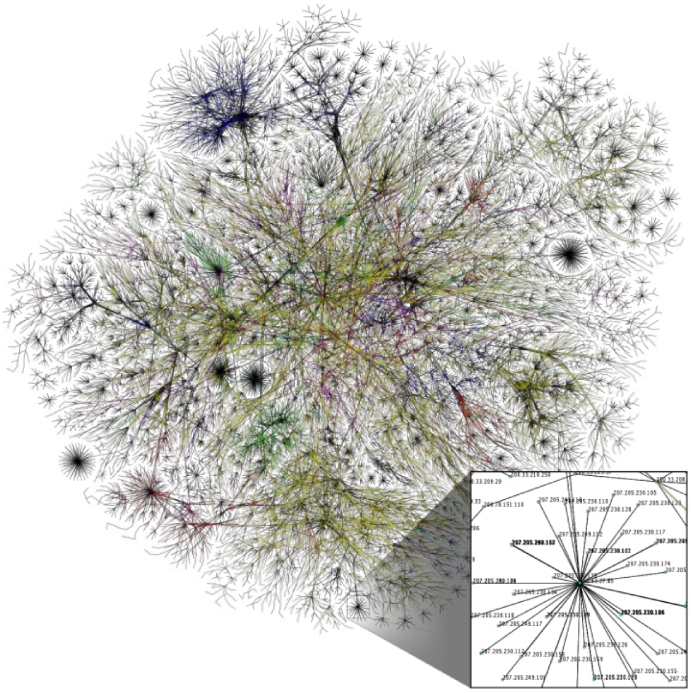
CHALMERS



A Distributed, Parallel and Fault Tolerant BGP Routing Daemon

Master of Science Thesis at the Department of Computer Science and Engineering

ERIK BERGSTRÖM
MARTIN MILLNERT

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2015

A Distributed, Parallel and Fault Tolerant BGP Routing Daemon

(In alphabetical order)
ERIK BERGSTRÖM
MARTIN MILLNERT

Examiner: Sally A. McKee

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
"Partial map of the Internet based on the January 15, 2005 data found on opte.org. Each line is drawn between two nodes, representing two IP addresses. The length of the lines are indicative of the delay between those two nodes. This graph represents less than 30% of the Class C networks reachable by the data collection program in early 2005." - https://commons.wikimedia.org/wiki/File:Internet_map_1024_-_transparent,_inverted.png

Department of Computer Science and Engineering
Göteborg, Sweden June 2015

Acknowledgements

We would like to extend our gratitude to Sally A. McKee, for her help in this work. We would also like to extend our gratitude to the great folks on the IRC chat rooms of #networker, as well as #erlang.
Martin would like to extend his greatest gratitude to Behnaz Pirzamanbein, for all her loving support and encouragement that eventually led to here. Erik would like to send a big thank you to his wife Yalda Bergström, for all her love and support.

Abstract

The protocol that has delivered the Internet to the world suffers from aged implementations in current core routers. The implementations lack proper fault tolerance and suffer from single-threaded processing, underutilizing multi-core processors. While a lack of multi-core processing capabilities was not a notable deficit 20 years ago, today a modular router is a small compute cluster in itself with multiple multi-core processors.
This thesis identifies first that in order to simplify, scale and improve fault tolerance in a BGP implementation, it can be decomposed into smaller parts. This thesis also shows that the processing of a BGP implementation's critical path can be parallelized down to a per-prefix level and distributed over multiple nodes in a cluster implementation of a distributed RIB. Using the ErlBGP architecture, greater stability could be achieved in the Internet.

Contents

Glossary **1**

1 Introduction **3**
1.1 Problem statement . . . 3
1.2 Research Questions . . . 4
1.3 Scope and Limitations for the prototype . . . 4

2 Background and literature study **5**
2.1 IP Router Building blocks . . . 5
2.1.1 Network interfaces . . . 6
2.1.2 Packet buffers . . . 6
2.1.3 Switch fabric . . . 7
2.1.4 Control processors . . . 8
2.1.5 Routing tables . . . 8
2.1.6 Packet forwarding engine . . . 8
2.2 Brief history of IP router's technology . . . 9
2.2.1 First generation routers . . . 9
2.2.2 Second generation routers . . . 10
2.2.3 Third generation routers . . . 11
2.2.4 Head-of-line blocking and Virtual Output Queues . . . 12
2.2.5 Recent developments . . . 12
2.2.5.1 Warehouse-Scale Computer's . . . 13
2.2.5.2 OCS Networking . . . 14
2.3 Border Gateway Protocol . . . 14
2.3.1 Introduction to BGP . . . 14
2.3.2 BGP Fundamentals . . . 17
2.3.2.1 BGP Message Types . . . 17
2.3.2.2 BGP Finite State Machine . . . 18
2.3.2.3 BGP Timers and Events . . . 19
2.3.2.4 BGP's Routing Information Bases . . . 20
2.3.2.5 BGP Decision Process and Update-Send Process . . . 21
2.3.3 BGP Extensions . . . 23
2.3.3.1 BGP Multiprotocol Extension, RFC 4760 . . . 23
2.3.3.2 4-byte ASN extension, RFC 4893 . . . 24
2.3.3.3 Graceful Restart Mechanism in BGP . . . 24
2.4 Routing plane software architecture evolution . . . 25
2.4.1 Early days monolithic software . . . 26

2.4.2 Data- and control plane separation 28
2.4.3 Decentralization of control plane components 28
2.4.4 Software Defined Networking 30
2.4.5 Typical Fault Tolerance in modern routing plane software 31
2.4.6 Typical concurrency in modern routing plane software 32
2.5 Distributed systems and the CAP theorem 32
2.5.1 Distributed storage of configuration state 33
2.5.2 Distributed storage of objects 34
2.6 Erlang 35
2.6.1 Erlang/OTP 36
2.6.2 Distributed Erlang 36

3 Methodology **36**
3.1 Method 36

4 Results **37**
4.1 Goal Design of the Total System 38
4.2 Distributed state storage systems 39
4.3 Parallelized and Distributed UPDATE message processing . . 40
4.3.1 Critical data path - BGP UPDATE message processing 40
4.3.2 Parallelization of Critical Path 42
4.3.3 Distributed processing of Critical Path 44
4.4 A Distributed RIB 45
4.4.1 Replication strategy overview 45
4.4.2 Distributed RIB and Routing Storage Daemons cluster 46
4.5 Decomposition of BGP with multi-process, distributed Erlang/OTP prototype 50
4.6 *ErlBGP* - multi-process, distributed Erlang/OTP prototype . 51
4.6.1 Decomposition into multiple processes 51
4.6.2 ErlBGP walkthrough 52
4.6.2.1 Distributed Erlang & Multiple Nodes 53
4.6.2.2 The ErlBGP application and a node's supervisor tree 54
4.6.2.3 The erlbgp_peerfsm and erlbgp_peerrecv relationship 54
4.6.2.4 Critical path and BGP UPDATE message processing 54
4.6.3 ErlBGP Usage Examples 56

5 Discussion **59**
5.1 On using the right tool for the job 59
5.2 Differences with (HIDELL) and (ONOS) 61
5.3 Future work 62
5.3.1 Improve Fault Tolerance 62
5.3.2 Implement Distributed RIB 62
5.3.3 Further Performance Optimizations 63
5.3.4 Look into TCP Socket Migration solutions 63
5.3.5 Configuration Management and API 63
5.3.6 Application of heavy BGP policy computations 64
5.3.7 Application of ErlBGP to a real-world router hardware platform 64

6 Conclusion **65**

7 Bibliography **67**

A Decision Process Pseudo Code **75**

B Gigabit Ethernet Ping-Pong Latency test **77**

C "sloccount" comparison **80**

Glossary

Here we present some terminology used in the report.

Adj-RIB-In Adjacency RIB, in. A RIB which stores the incoming route updates.

Adj-RIB-Out Adjacency RIB, out. A RIB which stores the information already sent or to be sent to a peer.

AS Autonomous System. An entity that controls a set of prefixes.

ASN Autonomous System Number. A usually globally unique and registered number connected to an AS.

AS path A list of ASN that describes a path between two networks.

BGP Border Gateway Protocol. A protocol for routers to exchange network reachability information.

BGP speaker A system that implements BGP.

EGP External Gateway Protocol. Routing protocols that are used externally, between AS'es. The only relevant protocol today is BGP.

FSM Finite State Machine. A model for computations in software that uses different states.

Less specific route A *less specific route*, R_1, relative another route, R_2, is a route to a shorter prefix, i.e. a shorter bitmask (less most significant ones). The prefix R_1 points to is a superset of the IP addresses contained within the prefix R_2 points to.

Loc-RIB Local Routing Information Base. Where the locally selected best routes are stored.

IGP Internal Gateway Protocol. Routing protocols that are used within an AS.

IP Internet Protocol. The most commonly used protocol to address network devices on the Internet.

IS-IS Intermediate System to Intermediate System. An IGP.

MAC address Media Access Control address. A unique address bound to a network interface.

More specific route A *more specific route*, R_1, relative another route, R_2, is a route to a longer prefix, i.e. a longer bitmask (more most significant ones). The prefix R_1 points to is a subset of the IP addresses contained within the prefix R_2 points to.

Next hop address A protocol address to send IP packets via.

NLRI Network Layer Reachability Information. Information containing prefixes.

OSPF Open Shortest Path First. An IGP.

RFC Request For Comments. A publication from the Internet Engineering Task Force (IETF) for standards.

RIB Routing Information Base. A table with routing information.

RIP Routing Information Protocol. An IGP.

Route A prefix with a next hop address.

Routing Table A datastructure organizing storage of routing information.

Path Here: another word for route.

Peer A BGP speaker, A, that is connected to another BGP speaker, B, is said to "peer" with B. B is thus also a peer of A.

Prefix A network address with an attached network bitmask, such as 192.0.2.0/24. 192.0.2.0 is the network address, and /24 means to apply a 24 bit long 1s bitmask for the most significant bits filled with 8 zeroes. Applying this bitmask on an address within the subnet specified (192.0.2.0 - 192.0.2.255) using a bitwise or operation will return the network address, 192.0.2.0.

TCP Transmission Control Protocol. A transport layer protocol on top of the Internet Protocol.

1 Introduction

Today, BGP version 4 is the single protocol responsible for the roughly 50 000 networks being connected [15] in the Internet.
BGP runs between routers sitting on the edges (hence "border") of IP networks, enabling communication between networks. The routing protocol exchanges *network reachability information* between networks which then propagates towards all connected networks. The implementations of BGP in routers today have to process a fair amount of information, which is only expected to increase [43].

1.1 Problem statement

While processing large amounts of information is not a challenge in itself to the protocol, all known existing implementations of it processes information completely in serial, in a single master process on each router.

This presents two challenges:

1. Fault tolerance and,
2. performance and scalability.

Some implementations addresses fault tolerance by having a single backup process running, and strategies have been proposed to address the performance and scalability challenge by partitioning the information to multiple serialized processes. The current answers to the fault tolerance problem do not address the performance and scalability issues, and, conversely, the current proposed answers to the performance and scalability challenge, we argue, do not properly address the fault tolerance challenge. We have found one proposal on partitioning that attempts to also address the fault tolerance challenge, however, this proposal, while on the right track, is high level and does not go deep enough to address the core of the problem: Decoupling the very internal functions of BGP such that they do not need to run on a single physical host with shared memory.
Lack of processing power to apply to this information reduces the amount of sanity checking that is possible to achieve, which reduces the accuracy of the routing information, negatively affecting the Internet with outages.
We research a design for a very granular decoupling of the internal functions of BGP, attempting to address both the fault tolerance and the performance and scalability challenges. In our experimental work we use the distributed programming toolbox of the Erlang programming language to show a proof

of concept. We identify the design's critical path and outline areas for further improvement.

1.2 Research Questions

We have defined the following Research Questions that we address in this work:

RQ.1 Is it feasible to implement a distributed BGP daemon that address the two challenges above?

RQ.2 Can the critical path of a BGP daemon be parallelized?

RQ.3 Is it feasible to partition a RIB over multiple nodes?

To accomplish this work, we had to first research how to build distributed programs efficiently, as well as research BGP and related published work thoroughly. We will answer RQ.1 by implementing a prototype using the functional programming languange Erlang.

1.3 Scope and Limitations for the prototype

Our prototype will show the core aspects, the decoupling of the internal functions, and that:

- the distributed system can connect to a peer
- a peer can connect to the distributed system
- we can maintain established peers
- our system exhibits some of the fault tolerance properties, and autonomously restarts dead processes

Specifically, we will not:

- implement route import / export policies
- implement a distributed, highly available, strongly consistent state storage
  - though we will research the requirements
- implement a distributed, highly available, strongly consistent RIB

  - though we will research the requirements
- make any attempts to optimize the critical path for speed
- implement any BGP extensions

2 Background and literature study

In this chapter we first give a brief history of the evolution of IP router technology, both hardware and software. We introduce the reader to BGP. We refer to relevant related research work where appropriate for further reading.

2.1 IP Router Building blocks

An IP router is a device that forwards, or routes, packets between different IP subnets towards their destination, according to its routing table. It may use dynamic routing protocols to exchange network reachability information with other routers, to populate its routing table. IP routers form routed IP networks, and different administrative domains operate distinct networks. Multiple distinct networks connected together form an internetwork and a very famous such internetwork is the Internet [10]

A network can internally use a dynamic routing protocol, called an Internal Gateway Protocol, to dynamically exchange network reachability information. When exchanging routes externally, conversely an External Gateway Protocol should be used. This has since the 90s been standardized to the "Border Gateway Protocol", from being on the border of a network, talking to other, foreign networks.

The major building blocks of an IP router are:

- Network interfaces
- Packet buffers
- Switch fabric
- Control processor
- Routing tables
- Packet Forwarding Engine

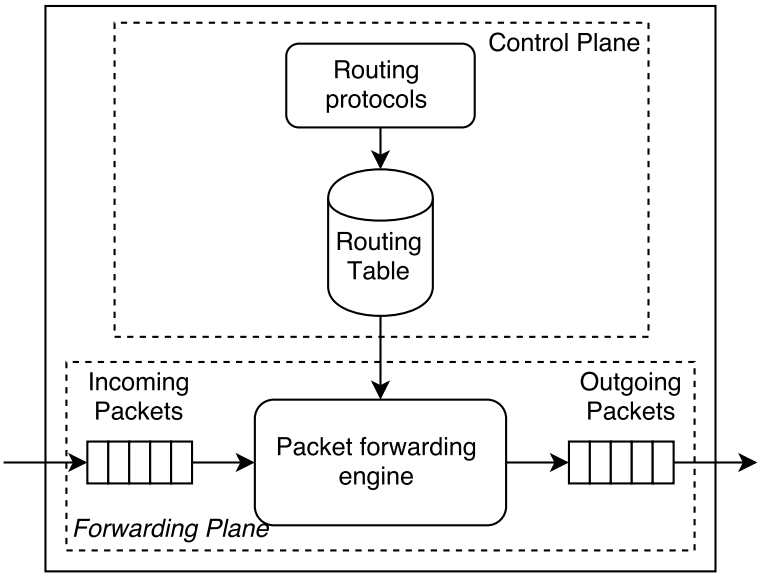


Figure 1: IP Router Building blocks

2.1.1 Network interfaces

Network interfaces are the translators between physical signals and the logical packetizations of these signals for a networked device, such as a computer, router or switch. Network interfaces are typically said to have a certain "bandwidth", and what is typically referred to is not the strictly physical signals bandwidth, e.g. frequency range of signaling, but the data rate.
A network interface that has a "bandwidth" of 10 gigabits per second, has actually a data rate of 10 gigabits per second. This means that on the physical layer, the interface will be transmitting one bit every 0.1 nanosecond. The device's physical apparatus, called "PHY", will, depending on modulation format of the signal and implementation, for example run at a clock rate of 10 GHz and decode 1 bit at a time when receiving a frame (packet). The device will stitch together a stream of bits to form the frame, typically with Ethernet framing, and make these frames available for the host machine.

2.1.2 Packet buffers

To illustrate the role of packet buffers in packet network devices, we will describe a device without any. For simplicity, there are more than two network interfaces, they are all of identical data rate and there is a single pipeline processing a stream of packets from input ports to their output ports.

Lacking any packet buffers, packets that arrived at any other port than the

one whose packet was picked up by the pipeline, are discarded. There is no possibility for either collisions or congestion within the device, but the aggregated throughput is also limited to the data rate of a single interface, despite having more than two ports.

If the data rate of a selected output port for a stream of packets is a tenth of the data rate of the packets' input port(s), it is only possible to forward a tenth of the received packets.

If, for every discrete time slot, the pipeline would pick up an input packet from more than one port, it could only dispatch them to non-identical output ports. From this it is apparent that with the introduction of a so-called *scheduler*, we could optimistically improve the aggregated throughput of the device in case the necessary output ports during a given time slot are more than one.

With the addition of packet buffers, either input buffers, output buffers or a combination of input and output buffers, it is possible to improve the aggregated throughput of a network device considerably. By increasing the probability that a received packet can be queued either at the ingress interface, the egress interface or a combination of them, the ability of the egress interface to be transmitting packets at every time slot increases. This reduces the packet loss rate.

2.1.3 Switch fabric

An efficient IP router contains a non-blocking switch, between the ingress interfaces and the egress interfaces, meaning it can send from any ingress port to any egress port without inherent contention in the switch fabric.

Any switch device can experience natural contention, when the total ingress bandwidth for a certain set of packet flows is larger than the available bandwidth on the flows' egress interfaces:

$$BW_{TotalIngressFlows} = \sum_{i=0}^{n} BW(i), i = flownumber \tag{1}$$

$$BW_{AvailableEgress} = \sum_{j=0}^{m} AvailableBW(j), j = egressinterface \tag{2}$$

$$BW_{TotalIngressFlows} > BW_{AvailableEgress} \tag{3}$$

Since network traffic arrives to a network device stochastically, packet buffers greatly reduces the packet loss probability. However with ideal traffic patterns and/or infinite buffers, there is a potential bottleneck in the interconnection between all ingress and egress ports. There are many designs for these interconnections, such as: shared bus, shared memory, distributed output buffered and crossbar.

2.1.4 Control processors

An IP router must manage several control functions, such as primarily the various routing protocols they run. These typically run on a dedicated control processor of a modern router, which is a generic CPU. These CPUs are basically provided in the form of a generic server and manage the router's various attached devices.

2.1.5 Routing tables

The router exchange network reachability information, also known as routing information, with other routers and store this information in routing tables. Each network destination, called "prefix" for short, is kept at least once in these tables, together with information about which so called next hop IP address the router should use direct traffic towards the destination. In total, this information is called a "route". It also keeps a set of metadata together with each route that the router can use to make policy decisions about which one of multiple possible routes to select for a given prefix. To "select a route" means to use it for packet forwarding and potentially propagation to neighbor devices.

A router's protocols keep their own set of routing tables, in addition to the main routing table.

The routing tables in the control plane are called the Routing Information Base, or RIB.

2.1.6 Packet forwarding engine

In routers with a so called "control plane and forwarding plane separation", the main routing table is not used for the forwarding of packets by the forwarding plane, it is only used for forwarding of control plane traffic. Instead, the forwarding plane uses a packet forwarding engine, a set of hardware with an accelerated routing table which the control plane keeps in sync. The

routing tables in the forwarding plane are called the Forwarding Information Base, or FIB. The packet forwarding engine will for each received IP packet perform lookups of the IP packet's destination address against this accelerated routing table.

2.2 Brief history of IP router's technology

In this section a brief history of the evolution of IP router architecture is given, concluding with the latest designs that have a good fit with this research project.

A few core parameters of an IP router that limits its aggregated throughput:

- Cost
- Power consumption / heat dissipation
- Internal interconnection capacity
- Total interface capacity
- Forwarding route lookup performance
- Amount of packet buffer memory
- Degree of reliability and fault tolerance
- Packet handling flexibility in fast path

Constructing a high performance router could be seen as a linear equation system on these variables, where some are given, some should be minimized, others maximized.

An aspect that was alluded to in the introduction above that is very important, is the scheduling and contention resolution strategies of routers.

2.2.1 First generation routers

First generation routers were basically PCs with multiple network interfaces. The network interfaces were attached to a common bus with a fixed throughput shared between connected devices. Received packets were stored in the main memory of the router, and the regular CPU was used both for running the routing protocols and to perform the packet forwarding, including route lookups.

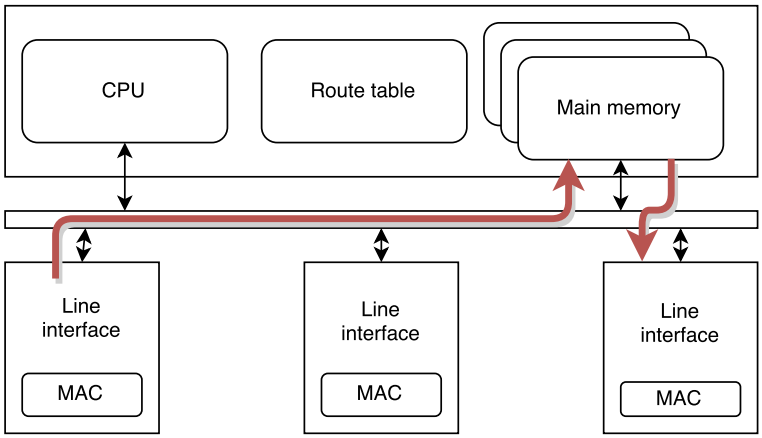


Figure 2: First generation routers

2.2.2 Second generation routers

When the utility of connecting networks together grew as more resources became networked, the requirement for increased aggregated throughput through the routers consequently increased as well.

It was immediately obvious that using the main memory as a packet buffer and the CPU as the forwarding engine constituted a single set of resource bottleneck, and especially with a shared bus, the IP packets are sent twice over it: to and from the packet buffer.

The first incremental improvements on this design constitute a second generation of routers.

In order to relieve the main memory and the CPU, both packet buffers and forwarding offloading was added to the network interfaces. Due to limited memory capacity, the forwarding offload could only handle a subset of all possible routes in the actual forwarding table, meaning a subset of the routes could be cached - forming a forwarding cache. In the event of a cache hit, packets could be sent directly from the incoming interface to the outgoing interface over the bus without going to main memory. This offloaded the contention between the main memory and the bus and allowed for greatly increased utilization of the shared bus and led to increased aggregated throughput of the routers.

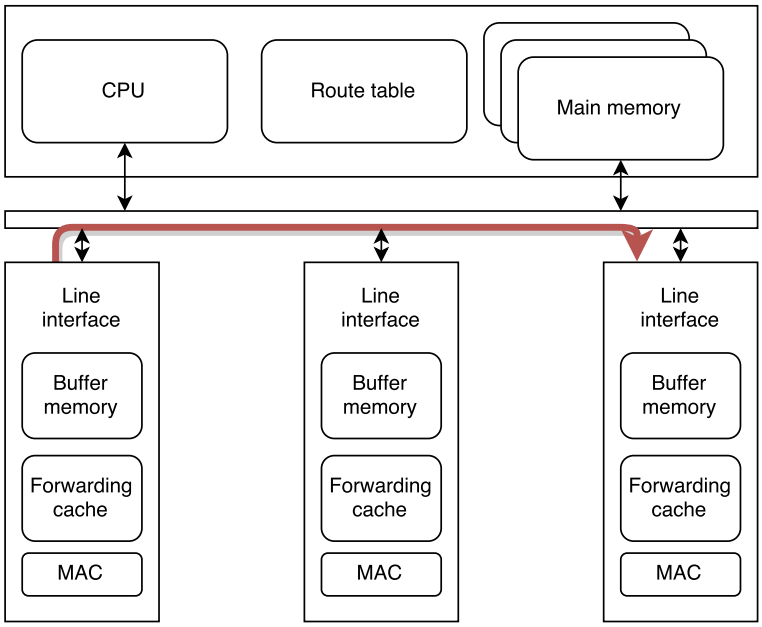


Figure 3: Second generation routers

2.2.3 Third generation routers

With the central packer buffer contention partially resolved, the next throughput limitation was the shared bus. A shared bus implies some form of Time-Division-Multiplexing (TDM) for allowing multiple connected interfaces to speak over the bus. While it is possible to run the bus at a proportionally higher bandwidth and clock rate than the connected interfaces to compensate for TDM effect, this uses more power and is more costly.

In order to further increase throughput, third generation routers improved on the second generation by addressing this bottleneck. The solution was to replace the shared bus with a crossbar switch. Into this crossbar switch, so called "Line Cards" were plugged. Line cards are improved network interfaces, basically small computers of their own, that may even have more than one network interface on them. The line card fits the entire forwarding table, has larger local buffers needed to support sending and receiving traffic over the crossbar to other line cards. [54]

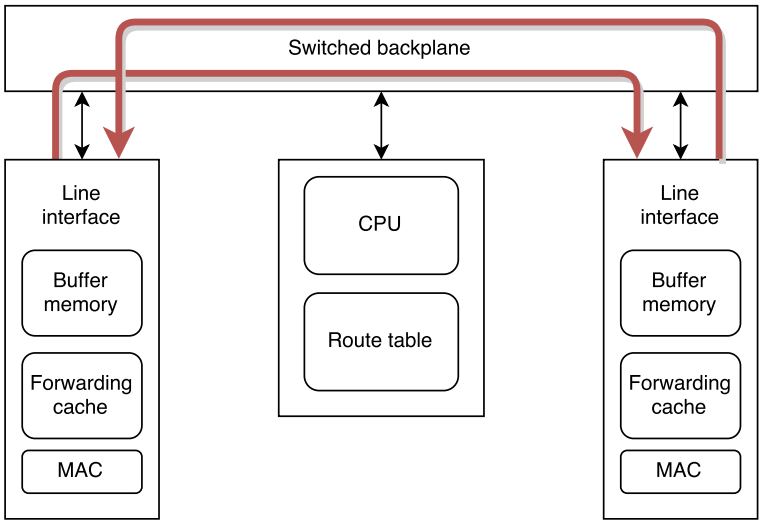


Figure 4: Third generation routers

2.2.4 Head-of-line blocking and Virtual Output Queues

The first incarnations of the crossbar switches however suffered from head-of-line blocking.

Head-of-line blocking is a situation where the input line card is unable to transmit the packet at the head of its crossbar output queue, due to the receiving line card on the crossbar's corresponding input queue being full of packets. There is a single output queue to the crossbar and it is sufficient for a single of all other line cards having its input queue full, to block any other line card.
This was addressed by the development of the Virtual Output Queues where each line card has one virtual output queue to the crossbar for every destination line card's input buffer. The transmission scheduler then can loop over the physical output queue and transmit not just the packet at the head of the queue, but the packet at the head of each Virtual Queue. [46]

2.2.5 Recent developments

In this section modern high-scale datacenter infrastructure is briefly reviewed as a lead-in to some very recent developments which are very relevant to the thesis research work.

2.2.5.1 Warehouse-Scale Computer's

To quote Barroso et al. 14 from Google:

> As computation continues to move into the cloud, the computing platform of interest no longer resembles a pizza box or a refrigerator, but a warehouse full of computers.

In these warehouse-scale computers, the networking fabric is an essential component since there can be 10s of thousands of servers in hundreds of racks. While a top-of-rack switch can provide non-blocking switching performance between the nodes within the same rack, providing non-blocking switching performance from any server to any other server in a data center in a single device is extremely costly. Instead, a large networking fabric is constructed with many smaller building blocks.

Citing Aweya [10]

> It is generally accepted that large router switch fabrics of 1 terabits per second (Tbps) throughput or more cannot be realized simply by scaling up a fabric design in size and speed. Instead, large fabrics must be constructed by interconnection of switch modules of limited throughput. The small modules may be designed following any approach, and there are various ways to interconnect them.

Designs of non-blocking switches has been known since at least Clos[23]. Clos networks, as the suggested method above, is a way to connect multiple smaller-capacity switches together in multi-stage levels to construct a larger switching fabric. This type of design is fundamentally how designs of large datacenter fabrics looks today, 2015[6][41][36]. Facebook, Microsoft and Google have all realized variations of IP-based Clos networks to create datacenter fabrics. The most common design of these IP-based Clos networks use E-BGP between the network devices, due to a few of BGP's properties: Many vendors have compatible implementations, operators know how to use it, and it limits the size of the broadcast domains.

Alternative design proposals have been published, such as the Portland design, which rather than via MAC-in-MAC encapsulation instead perform MAC address translation to form very large layer 2 networks. We have not been able to find any reference to an actual implementation of the envisioned 100,000 servers' large broadcast domain, whereas the IP E-BGP-based Clos designs have proven popular.[60]

2.2.5.2 OCS Networking

During 2015, Facebook has taken the Open Compute System[61] further forward by open sourcing the designs for their "Six-pack", a modular, simple, 8-slot 1.2 Tbps/slot data center switch.[11][13]

Each line card is based on the Facebook "Wedge" design[12], which hosts an OCS Micro-Server for control plane[63]. The "Six-pack" has a backplane speaking plain Ethernet allowing the line card to talk to each other using regular Ethernet frames.

The Open Compute System organisation have today a number of system designs pending review, primarily for 1RU top of rack switches. There is at the time of writing more competent hardware available. There is however a lack of software choices to manage the equipment.

Based on the available switch designs, a common specification for control CPU have circa 2GHz quad-core CPUs with 2x8 GB RAM - obvious multi-processor availability. In the case of the "Six-pack", there are a total of 8 line cards Micro Servers + 2 backplane switching boards with 2 Micro Servers each; a total of 12 CPUs which, using the commonly seen specifications could result in a total of 48 cores and 192GB RAM available totally in the system. Clearly, a distributed system could take advantage of such a design for high performance and fault tolerance.

In summary, the OCS Networking project is picking up speed when it comes to available hardware designs, but the software choices are lacking.

2.3 Border Gateway Protocol

In this section we give a brief introduction to the Border Gateway Protocol and review a couple of its key functions slightly more in-depth.

2.3.1 Introduction to BGP

BGP is the protocol for exchanging reachability information between different ISP's on the Internet. The protocol allows operators to define a so called Autonomous System (AS), a representation of an administrative domain such as a company, a service provider or a government agency.

To better understand where BGP fits in among the routing protocols, it

helps to briefly mention a couple other protocols.

Network operators typically use an Internal Gateway Protocol (IGP) for exchanging routes between routing elements within their network dynamically. Common ones used today include Routing Information Protocol (RIP, specifically version 2), Open Shortest Path First (OSPF, version 2 and 3) and Intermediate System to Intermediate System (IS-IS).
OSPF and IS-IS are both link state routing protocols – they advertise links, not routes, that each participating node can populate a reachability matrix that describes its reachability to the other nodes in the network. It then runs Dijkstra's Shortest Path First Algorithm to compute its best routes based on the information it received.
RIP is a distance vector routing protocol, meaning it works by advertising and re-advertising (distance, vector) per prefix, i.e. a metric for how far away it is to the originating network device of a certain prefix, and which next-hop interface that should be used.

BGP on the other hand, is a Path Vector Routing protocol (sometimes called Distance Vector). Path Vector means the protocol will, as operators announce network prefixes and they propagate through multiple networks, build a path list, called the AS PATH, of networks through which the prefix is reachable. That is, a participating network in an internetwork of AS's will receive reachability information containing information on over which AS PATH a specific prefix is reachable through.

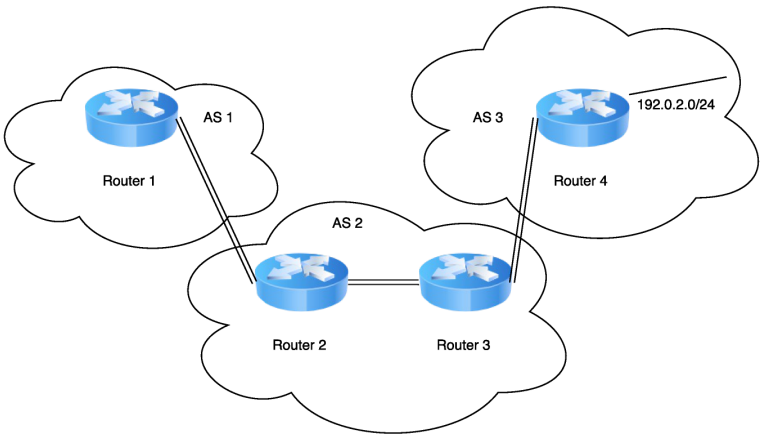


Figure 5: A diagram of three connected autonomous systems

An AS is, as the name suggests, autonomous. Its operators define for it a

certain routing policy, consisting of import and export routing policies. The policy defines what set of reachability information it will propagate to its peer networks, and possibly what, if any, local modifications it will apply to these informations.

When an AS has announced reachability information about a certain IP prefix to its peer networks in accordance with its export policy, this announcement will propagate through networks in accordance with those networks' respective policies. In the case of the Internet and an announcement of an allocated globally unique IP prefix registered with an address registration authority (RIR), eventually all properly connected networks will be able to receive this announcement.

If the only installed routes a network has in its BGP speaking routers, except directly connected routes from locally configured interfaces, are the ones it has dynamically learned using BGP, it will prior to receiving this reachability information not know where to send an IP packet with a destination IP address in said IP prefix. Routers configured in such a way, with only dynamically configured routes, especially lacking a so called route of last resort, are said to be "participating in the *Default-Free Zone*". Packets with a destination IP address for which the router lacks reachability information, will be discarded. E.g. just discard it. Depending on configuration, ICMP messages will be returned to the source IP address indicating the router does not know where to send the packet.

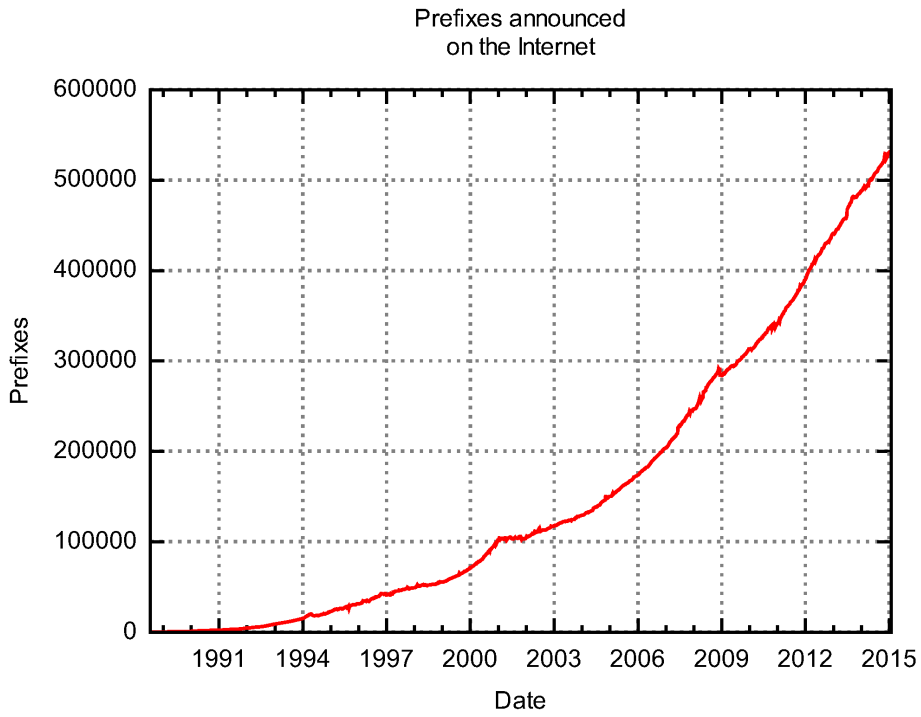


Figure 6: Number of prefixes announced to the Internet, e.g. the "Default-Free Zone" ((CC), Wikipedia user Mro)

However, once the announcement from the AS doing the original announcement has reached another network, this network will be able to send packets towards the origin network, passing through networks according to whichever resulting AS PATH. Thus, a network controls incoming traffic through which announcements it sends to peer network.

2.3.2 BGP Fundamentals

The BGP protocol uses TCP to transport messages between two so called BGP speakers. A BGP speaker must always be listening on an IP address on TCP port 179. Two BGP speakers that are configured to establish a connection between each other are said to be "peers" to one another.

2.3.2.1 BGP Message Types

BGP speakers communicate with each other using four well defined message types. These are:

- OPEN - Initial message sent by each speaker to its peer when attempting to establish a connection - contains session attributes

- KEEPALIVE - Message sent in a negotiated interval between peers to allow peers to continuously monitor the liveness of each other
- UPDATE - The message type carrying exchange of routing information, either new routes or withdrawn routes
- NOTIFICATION - message sent when a peer initiates the closing of a BGP connection - indicating why it was closed

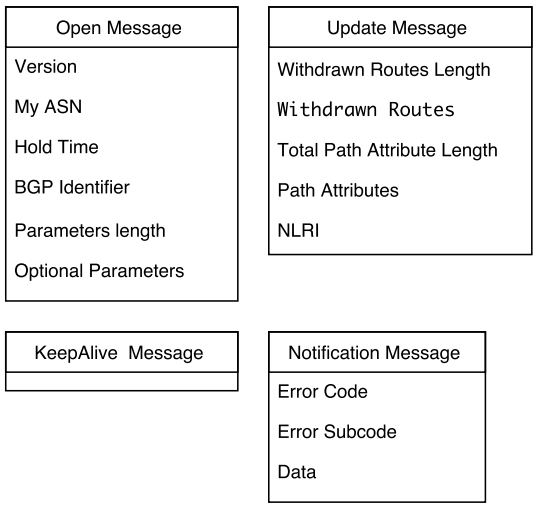


Figure 7: BGP Message Types

2.3.2.2 BGP Finite State Machine

In BGP, there is a well-defined finite state machine (FSM) that consists of the 6 states shown in figure 8 Idle, Connect, Active, OpenSent, OpenConfirm and Established.

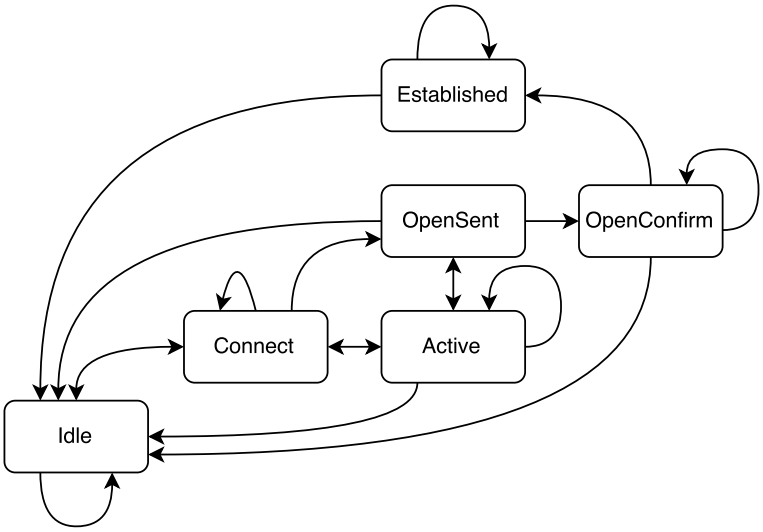


Figure 8: The six states in the FSM

These six states are a fusion of the state changes experienced by two different sides of a BGP connection, into one unified FSM:

- the state changes of a BGP speaker initiating a connection to the other (the active side of the connection), and,
- the state changes of the BGP speaker receiving an incoming connection (the passive side of the connection).

With the combined six states two BGP speakers may establish a peer relationship. There is a corner case where both peers have initiated a connection with each other and a so called connection collision occurs. The specification defines a method for the speakers to determine which of the connections should be kept and removed, respectively.

2.3.2.3 BGP Timers and Events

The transitions from a state, to either itself or a different state, is event-driven. There are over 20 defined events that can occur, that drive state changes. Some of these events are the result of a timer firing and there are a couple of especially important timers in the protocol, such as:

- KeepaliveTimer - When this timer fires, a keepalive message should be sent to the peer
- HoldTimer - When this timer fires, a BGP speaker hasn't heard from its peer for too long, and the connection is reset

- ConnectRetryTimer - This timer is used when the connection is being set up in the early states, for example between active and connect states.
- DelayOpenTimer - (Optional) This timer is used to delay the sending of an open message to the peer, to reduce the likelihood that both peers initiate one connection each
- IdleHoldTimer - (Optional) This timer is used to keep the FSM in the Idle state for a prolonged time to control peer oscillation - which is when a peer is oscillating between established and not established states continuously.

These timers are reset as a result of certain events happening in the system, such as for example when receiving a Keepalive message from the peer, the HoldTimer is reset.

While in the Established state of the FSM, a speaker sends UPDATE messages to its peers, to update them with what reachability information it has. These messages are the result of routing policies applied on the Routing Information Base, RIB, of the speaker.

2.3.2.4 BGP's Routing Information Bases

BGP contains the following three types of Routing Information Bases, RIBs:

- Adj-RIB-In - where received updates are processed and stored
- Loc-RIB - where the results of the BGP route selection, using the local import policy, applied to the routes from Adj-RIB-In, are stored
- Adj-RIB-Out - where the routes to keep peers in synch about are kept, resulting from the local export policy being applied to the Loc-RIB

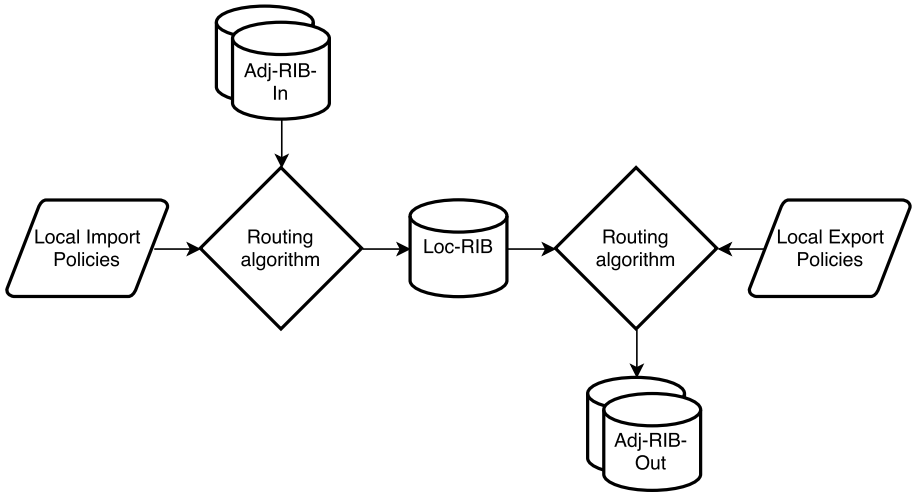


Figure 9: BGP: RIB

The BGP RFC defines these three distinct database types but does not mandate a specific implementation of them, only that the externally visible behavior is conformant with the specifications. Given that the information carried together with a route throughout the three RIBs changes only very little, there are good opportunities in an implementation to perform data reduction by storing the various information pieces only once.

A RIB implementation must support a few operations such as insert route, delete route, lookup route.

2.3.2.5 BGP Decision Process and Update-Send Process

The BGP Decision Process is the name of a set of procedures in BGP responsible for selecting one route per prefix as the active route, although there are later extensions to BGP to allow multiple paths to be selected[82]. It does this by processing received UPDATE messages' NLRI lists and Withdraw lists respectively.

When receiving a NLRI or Withdraw in an UPDATE message, a BGP speaker first process the information to the Adj-RIB-In according to:

- Any present routes in the Withdraw list, shall be removed (or marked for removal) from the Adj-RIB-In,
- If the NLRI of the new route is identical to one that is currently stored, the new route replaces the old, which implicitly withdraws the old one (i.e. deleted),

- If the NLRI of the new route has no current matching entry in the Adj-RIB-In, the new route will simply be stored in the Adj-RIB-In,

The decision process then takes over. It has three stages that corresponds to a path's progress through the Adj-RIB-In, the Loc-RIB and the Adj-RIB-Out tables.

- Phase 1: Calculate degree of preference for route; keep it in Adj-RIB-In.
- Phase 2: Route selection - considering local policy, calculate the best path for a prefix, among all paths to the prefix present in all Adj-RIB-Ins, to install to Loc-RIB,
- Phase 3: Route dissemination - per peering session, select paths from the Loc-RIB to the session's corresponding Adj-RIB-Out based on local policy

After the decision process, the Update-Send process takes place, which distributes any changes in any peer's Adj-RIB-Out to the peer.

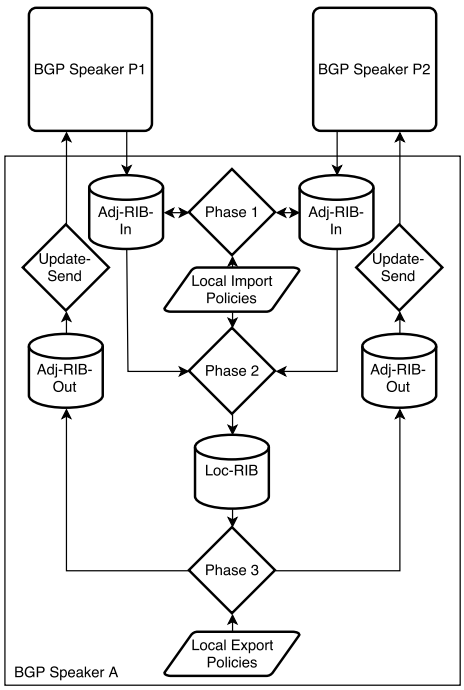


Figure 10: BGP Decision Process and UPDATE message processing

Phase 2 includes the often referenced best path selection algorithm, which basically selects the available path or paths to the prefix with the highest

preference. If more than one path is tied with the highest preference, it additionally performs a tie breaking algorithm to select just one of the available routes - the best path.
Only available paths are taken into consideration as the best path. A path to a prefix is said to be available when the router can determine an output interface on which to forward the traffic to the prefix over. Such a path is said to be *resolvable* by the router. *Route resolvability* then, is the ability of a router to, by performing a direct or recursive lookup on a next hop address, resolve an output interface for the forwarding.

By performing the route resolvability check on a path prior to selecting it as potentially the best path, the router can avoid installing bad forwarding information to the forwarding plane, which would result in so called "black holing" of traffic - i.e. sending traffic destined for a certain prefix into a black hole, where the packet are lost.

The Update-Send process is responsible for creating BGP UPDATE messages per Adj-RIB-Out. The process includes a construct of minimum delay between message sent, which allows routers to pack multiple NLRIs or Withdrawn routes per message. This batching of information per message is a processing versus latency trade off, in that it acts to amortize the processing overhead of a message over multiple carried information pieces.

2.3.3 BGP Extensions

BGP has a number of extensions defined over the base standard that are optional for an implementation to include. A few such extensions relevant to a production environment today includes:

- The BGP Multiprotocol Extension, RFC 4760[16]
- The 4-byte ASN extension, RFC 4893[80]
- The Graceful Restart Mechanism, RFC 4724[70]

2.3.3.1 BGP Multiprotocol Extension, RFC 4760

Recognizing that the NLRI defined in RFC 4271 only support IPv4 next hop addresses, aggregators and prefixes, the Multiprotocol Extension extends the support for multiple additional *Address Families*. It adds a new attribute called MP_REACH_NLRI that supports prefix and next hop addresses each up to 255 bytes long, as well as one called MP_UNREACH_NLRI, used to

signal the unreachability (withdrawal) of a route. BGP speakers advertise their capability to handle multiprotocol sessions by setting a *Capabilities Optional Parameter* in their OPEN messages, one parameter per address family supported. The extension specifies that the addition of this capability is additive, i.e. many distinct Address Families can be supported in a single session.

2.3.3.2 4-byte ASN extension, RFC 4893

The 4-byte ASN extension extends the range of possible ASN, from the earlier fixed-length 16-bit ASN field, via a new capability, providing a new 32-bit ASN field. To maintain backwards compatibility with routers that do not support the extension, a special reserved ASN, 23456, is encoded in the regular AS_PATH attribute. A new optional and transitive attribute, AS4_PATH, carries 4-byte as well as 2-byte ASNs. A BGP speaker with a 32-bit ASN cannot peer with a BGP speaker that is incapable of understanding 32-bit ASNs.

2.3.3.3 Graceful Restart Mechanism in BGP

A relevant BGP extension applicable for fault tolerance designs is the Graceful Restart Mechanism. The basics of the mechanism is that speakers when establishing a peer advertises a capability of being able to perform graceful restart. When later one of the BGP speakers, called *the restarting speaker*, terminates the session, the other speaker, called *the receiving speaker*, will maintain the routes advertised from the restarting speaker for some time. Depending on what capability flags the restarting speaker advertised on session initiation, the receiving speaker can keep forwarding to next hops associated with the restarting speaker during the restart. Likewise, the restarting speaker can keep forwarding according to its FIB as it existed before restarting the BGP processes.

There are mechanics in the restarting procedures to reduce the stress on recalculating best route selections, both in the receiving and the restarting speakers. These are assisted by the addition of a so called End-of-RIB marker - a special NLRI UPDATE message that is sent as the last update message when bringing up a session and advertising the Adj-RIB-Out to a peer, such that the peer can learn that the entire initial RIB has been transferred. This is useful in BGP implementations to hold off any BGP route selection computations until the entire Adj-RIB-Out has been transferred.

It is up to an implementation to decide whether to support continued forwarding while BGP is being restarted.

Advantages of Graceful Restart includes reduced computational load on routers and thus reduced cascading BGP UPDATE messaging, by removing cycles of withdrawal of routes followed by the re-announcement of largely the same routes. This reduces churn among the connected networks considerably.

Disadvantages of Graceful Restart include the risk of a receiving speaker to be forwarding traffic to a restarting speaker that does not properly forward the received traffic - there is e.g. a risk of black holing traffic when using the feature.

2.4 Routing plane software architecture evolution

In this section the evolution of IP router software architecture[38][37][27][21][58] is briefly reviewed, without which it is difficult to fully understand and appreciate the position from where this thesis work is building upon.

From first to third generation IP routers, the software went from controlling all forwarding to managing forwarding offload components, resulting in modularization of the software as a whole. The control processor became a control plane where forwarding eventually could continue even if the control plane restarts.

At this point, around 2003, standardization of control and forwarding plane separation became a hot research topic. An IETF working group, "Forwarding and Control Element Separation" (ForCES), was formed, that worked on developing standardized protocols for ways for Control Elements to talk to Forwarding Elements.

This description is supported by "The Road to SDN"[31] that describes the evolutionary path to today's SDN going through three phases:

1. Active networking, mid-1990s to early 2000s, introducing programmable functions to the network,

2. Separating Control and Data planes, 2001-2007, developing open interfaces between control and data plane (ForCES)

3. the OpenFlow API and network operating systems

There is an old argument in research papers over many years[42] for breaking up the "holistic" routing software implementations seen today on commercial routing platforms into modularized sub-systems. Such a modularization would allow for specialized focused development on subsystems creating a place for integrators to put these standardized subsystem components together instead.

From 2010 and forward this started to become reality, in large due to extremely large IT companies such as Google and Facebook increasingly buying so called white label network devices and putting their own software on top. Some companies specialize in providing a Hardware Abstraction Layer, HAL, for these companies internal development to interface with.[32][59][56][33][39][53]

Throughout all these developments in router software architecture, very little has happened with BGP software implementations. With system throughput in the largest routing platforms with multiple line cards at multiple terabits per second today, the exchange between internal and external routing information using the BGP4 protocol is typically always handled by a single BGP process on the system. There are regularly bugs detected in the various implementations[89][62][67][55][88][65].

For example a simple BGP message parser bug triggered on one of several hundred BGP peering sessions crashes the entire BGP process, disrupting network forwarding.

The latest developments in this field relates to past years efforts around Software Defined Networking (SDN), where a controller function manages multiple network devices in some form. A granular method for SDN is the OpenFlow standards where the controller can program the actual forwarding tuples on the network devices. The ideas in SDN and OpenFlow are recognizable from earlier ForCES and modularization efforts.

2.4.1 Early days monolithic software

Cisco's first router operating system[18], IOS, wasn't originally envisioned or intended to become what it became – it was just the OS that ran a Cisco router. As demand for IP routers increased, so did the demand for adding more features to the routers, which Cisco did by adding piece after piece onto IOS.

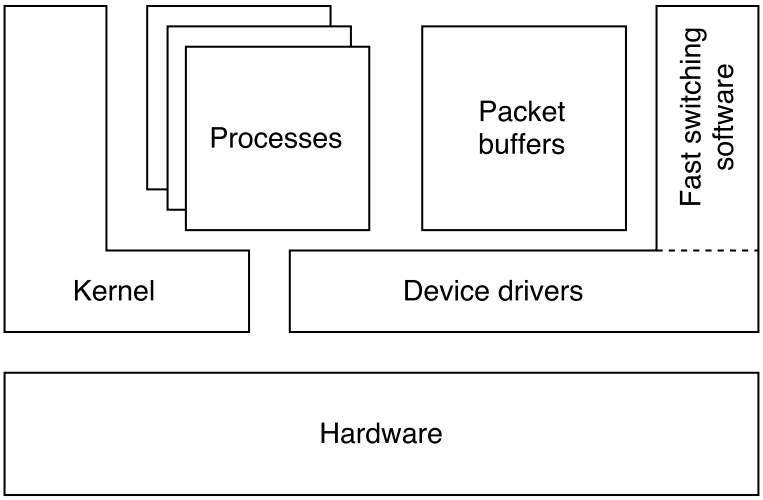


Figure 11: Cisco IOS system architecture overview

As IOS was originally built for embedded system it lacks some safeguards such as inter-thread memory protection. This was a conscious decision at Cisco since the CPU was a very limited resource and maximum cycles were required for packet processing.
Having no inter-process or inter-thread memory protection, all threads can access each other's memory regions. IOS is like a large monolithic single process system, where the processes within the operating systems are more similar to threads within the OS process. An IOS process has precisely one thread.
For process scheduling, IOS uses a run-to-completion scheduling mechanism with 4 priorities: Critical, High, Medium and Low.

The IOS kernel runs alongside the rest of IOS, all in user space with full system access. The kernel handles the process priority scheduler which has six process queues: Idle, Dead and one Ready queue per process priority.

Processes communicate with each other by writing binary blobs directly to each other's memory regions. IOS only exists for 32-bit MIPS CPU architectures.

IOS was first released in the mid-1980s and it is still being maintained for a range of Cisco products. As it was designed for performance at the expense of fault tolerance and security there was room to improve upon.

2.4.2 Data- and control plane separation

Due to the complexities and reliability concerns of a monolithic routing system[31] such as the Cisco IOS, where a crash in one part of the control system would also stop forwarding, there was a push to separate the control functions from the forwarding functions. As seen by the IP router technology evolution (chapter 2.2), there was a basic need to scale performance upwards which drove more intelligence out to the network interfaces, now line cards. Having the network interfaces manage local copies of the forwarding tables, were done in two ways. Cisco modified IOS to manage sending routes to the forwarding hardware on these line cards. The other approach was to rewrite the network router operating system from scratch and much more clearly than before separate the control plane from the forwarding plane. The latter approach was taken by Juniper Networks, who began in 1996 to build their new operating system called JunOS, which was released 1998 together with their first router.

Academia discussed data- and control plane separation and researchers wanted access to plug in to the router architectures[26][48] on both the data plane and the forwarding plane. "Router plugins" demonstrated a modular data plane where per-flow plugins could be installed to handle distinct flows differently. The "Click" modular forwarding design was a novel design at its time, modeling a pluggable forwarding architecture as a (Directed Acyclic Graph) DAG where each simple IP forwarding function was modeled as a small function with n inputs and moutputs. These micro functions, that the authors call *elements*, connect to each other to form a DAG from initial packet inputs to terminal packet outputs. This is very similar to the designs of ASICs, etc.

The IETF listened to the discussions and ideas of the industry and researchers alike and chartered the Forwarding and Control Element Separation working group in July 2001[30]. The idea was to formalize and standardize the various protocols and nomenclature involved in separating the control plane from the forwarding plane of a router. The working group was concluded March 2015. Unfortunately, there was never much traction for ForCES outside the standardization body and research world, in the scope of a single router.

2.4.3 Decentralization of control plane components

Plain separation of forwarding and control planes happened naturally to a limited degree by vendors' need to control dedicated forwarding hardware

from the control plane software, though no commercial implementation was open for others to plugin to. The commercial vendors are protective of this ability, as suggested by ForCES not being very successful, and one can only draw more or less informed conclusions as to why they didn't implement any such features to any reasonable standard this far.

Markus Hidell's doctoral thesis is titled
"Decentralized Modular Router Architectures"[42]. It is a work that is closely related to the work we've performed, that we were unaware of at the beginning of our own work.

Hidell argues in the closing remarks:

> We believe that there is a gain in allowing components to evolve independently and that modularization may be a way to provide such a scenario. Such a modularization could, if it proves successful, have impact on the router industry. A modular and decentralized approach to building routers would ultimately open up for sub-system vendors to develop specific parts of a router and for system vendors to focus on integration of modules, choosing whatever modules may best fit their needs.

This modularization of components *have* taken place within some very large IT companies.

Some of Hidell's main findings are:

- System designs and a prototype implementation of a decentralized router framework
- That a decentralized router framework in the ForCES model can be built using standard parts such as a PC computer
- That a previous monolithic system, the Zebra open source routing suite, can be decomposed into pieces
- That processing times can be greatly reduced by distributing the workload on multiple processing units
- That the penalties of using TCP-based multicast distribution protocols are increasing as the number of receivers scales up compared with e.g. NORM, NACK Oriented Reliable Multicast (RFC5740[4])

Hidell's findings include noting future areas of interest to research as:

- A continuation of the BGP-specific work
- Decompose additional protocols in addition to BGP
- The internal data network between the distributed component

Hidell's work regarding BGP is based on taking an open source routing daemon suite, Zebra, and decomposing the BGP implementation, "bgpd", into a frontend and a backend process. This is explained in Hidell's chapter 4.2.1. Hidell names the backend process "service process" and the frontend process "session manager". The frontend process keeps the responsibility for running the FSM with the peers, and processes incoming BGP updates. It then appears to be peering with one or many backend processes, by means of almost standard BGP. A couple of methods of distributing prefixes over multiple backends for computation speedup are attempted, and it is noted that the prefix distribution in the DFZ today on the Internet is uneven both in terms of number of prefixes per prefix length, and in terms of number of prefixes per region of the IPv4 address space. Hidell's figure 4.15 and 4.16 shows this distribution.

In "Distributed implementation of control protocols in routers and switches."[87] the authors claim a distributed control plane but not decentralized. The claim relates to the existence of a single control point with multiple controlled points, e.g. forwarding functions. This is the basic concept that appeared in the second phase of the router technology evolution.

Other work[20] discusses a solution to an iBGP route scalability issue where a Routing Control Platform would control the BGP route selection for an entire network. The authors stretched the work as far as including the effects of distributing and replicating the state of the RCP function (chapter 3.2), and showed that it would still produce consistency in the steady state. Distributing and replicating the function for fault tolerance is absolutely vital, for a network-wide controller function.

2.4.4 Software Defined Networking

Software Defined Networking[73][66] is a relative new term for the networking industry that has gone through a recent hype cycle and confusion about what it would mean for network operators[78]. In "Road to SDN"[31], SDN is defined as:

1. "An SDN separates the control plane (which decides how to handle

the traffic) from the data plane (which forwards traffic according to decisions that the control plane makes)"

2. "An SDN consolidates the control plane, so that a single software control program controls multiple data plane elements,"

According to this definition of SDN, this thesis work arguably qualifies as an SDN.

The SDN phase involves:

1. The vision of a network operating system

2. Distributed state management techniques

The authors of RCP have continued their work and published "the Intelligent Route Service Control Point", which externalises path computation from BGP itself. [Onix; 46] published the first concept of a Network Information Base, that the network operating system can use for storing its network state. It is important to point out that this thesis' work is not aimed towards a Network Operating System, but specifically at a, preferably modular, single-device control plane architecture architecture.

2.4.5 Typical Fault Tolerance in modern routing plane software

Traditional fault tolerance constructs of BGP build on the Graceful Restart (GR) feature explained in 2.3.3.3. A BGP speaker implementing GR can either support continued forwarding while restarting, or not.

Commercial vendors have commonly implemented Fault Tolerance based on[22]:

- Graceful Restart, a BGP speaker restarts sessions gracefully, and either supports continued forwarding or not while restarting,

- Non Stop Routing, state synchronization from primary to backup node, that coupled with Graceful Restart allows either planned or unplanned restarts without causing best path selection churn which risks programming cycles on the FIB,

Based on these features, vendors have commonly implemented features for upgrading the software on a router while in service.

Both Juniper and Cisco utilize a primary to backup state synchronization replication scheme. Arista in EOS have separated state storage from the individual processes which simplifies the restarting of agents[8], however, Arista has not developed any replication features for Sysdb so it is a very vulnerable centralized database of each router.

In "Routers for the Cloud", hidden BGP migration is researched using TCP socket migration, which would allow a BGP speaker to hide its BGP peer process / TCP socket migrations from its peers, which further reduces failure exposure to peers[5].

2.4.6 Typical concurrency in modern routing plane software

As seen in section 2.2, early router software architectures were single-threaded as a whole, whereas second generation architectures had multiple processes running. In the case of JunOS however, virtually all processing related to routing was still handled by a single process called rpd, the Routing Protocol Process[75].
IOS XR implements the different routing protocols (BGP, OSPF, OSPv3) and RIBv4, RIBv6 as independent processes[76]. This is an incremental improvement on JunOS, which is natural given that the architecture of IOS XR is younger than JunOS.

JunOS was built by modifying a regular FreeBSD kernel, IOS XR is built on top of a QNX kernel and Arista's EOS is built on top of a regular Linux Kernel. Arista's EOS is an incremental evolution on IOS XR, that simplifies the multi process state management by implementing an in-memory database called Sysdb[8].

None of these approaches address the multiprocessing opportunity within each process, such as BGP, which this report covers in chapter 4.3.

2.5 Distributed systems and the CAP theorem

Brewer[19] defined a theorem on consistency (all the nodes have the same data), availability (the system will always answer) and partitioning tolerance (nodes still work after separation), which is known as the CAP-theorem. It states that a distributed system must choose between two out of these three.

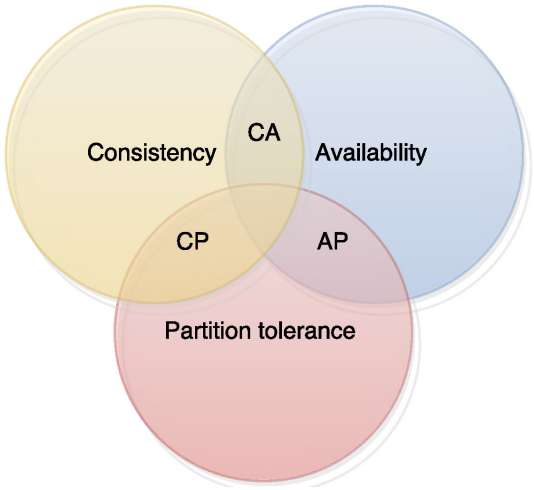


Figure 12: Consistency, availability and partitioning tolerance

We identify two distinct needs for distributed state storage in our thesis work:

- Distributed, highly available, strongly consistent e.g. configuration state storage
- Distributed, highly available, strongly consistent RIB

The state storage problem is old, by now. Brewer identified it as future work year 2000. Today, 2015, there has been work done even in the field of distributed network operating systems, in for example Onyx[49] and ONOS[50].

Both define two types of state storage, a type of strongly consistent state storage and a more loosely consistent one-hop DHT for faster access. ONOS pick off the shelf open source software components for both tasks; Apache Cassandra for the latter and Apache Zookeeper for the former.

2.5.1 Distributed storage of configuration state

When building a typical distributed systems there are a couple of commonly recurring challenges, such as naming, distributing synchronization and group communication. Several network controllers have chosen to use Apache ZooKeeper for this purpose[74].

xThe Apache ZooKeeper project[7] describes itself:

> ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and

> providing group services. All of these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable. Because of the difficulty of implementing these kinds of services, applications initially usually skimp on them, which make them brittle in the presence of change and difficult to manage. Even when done correctly, different implementations of these services lead to management complexity when the applications are deployed.

This readily available component satisfies the needs of this thesis work.

2.5.2 Distributed storage of objects

Whereas Onix and ONOS use eventually consistent storage consistency for the Network Information Base of a off-the-shelf open source component, this thesis work cannot use the same consistency model for a distributed RIB implementation. The increased risk of state loss due to failure exposure while only being stored at one node, is too severe for the level of fault tolerant this work is targeting. Therefore this work requires a novel approach to a specific distributed state storage application, specifically designed for the RIB use case, where there exists no previous work to the knowledge of the authors.

If we reduce the distributed RIB problem to a strongly consistent object storage problem, there is previous work. Weil et al. [84] developed a pseudo-random data distribution algorithm called Controlled Replication Under Scalable Hashing and using this developed the storage system Ceph[83], which achieves strongly consistent object storage in a cluster without a central point of failure.

Ceph distributes data to its RADOS cluster[85] using the CRUSH hash function.

RADOS and CRUSH, simplified, specify a hierarchy of cluster placement information:

- Cluster contains object storage daemons (OSDs)
- Cluster contains a set of pools
- A pool contain a set of Placement Groups (PGs)
- PGs have a dynamic map of OSDs

- Replication levels and placement strategy is maintained on the pool abstraction level
- The placement function, CRUSH, controls the selection of PGs for a write of an object in a pool

The benefits are a strongly consistent storage system with single network hop access to read data. Writes are directed at a primary which then replicates, resulting in a two-hop latency for writes.

This thesis work adapts this work for the specific use in a distributed RIB in section 4.4.

2.6 Erlang

Erlang was developed by Ericsson and first appeared in 1986. Erlang is a functional programming language for developing concurrent, distributed and fault-tolerant applications.

> Functional programming forbids code with side effects. Side effects and concurrency don't mix. You can have sequential code with side effects, or you can have code and concurrency that is free from side effects. You have to choose. There is no middle way.
>
> Erlang is a language where concurrency belongs to the programming language and not the operating system. Erlang makes parallel programming easy by modeling the world as sets of parallel processes that can interact only by exchanging messages. In the Erlang world, there are parallel processes but no locks, no synchronized methods, and no possibility of shared memory corruption, since there is no shared memory.
>
> Erlang programs can be made from thousands to millions of extremely lightweight processes that can run on a single processor, can run on a multicore processor, or can run on a network of processors.

Joe Armstrong, Programming Erlang - Software for a Concurrent world, 2007[9]

2.6.1 Erlang/OTP

Erlang/OTP is a framework to help build Erlang programs. It contains some modules, behaviors, libraries and a general design of how to write the program.

The most common behaviors are:

- gen_server - A framework for building a simple server application.
- gen_fsm - A framework for building a finite state machine.
- supervisor - A framework for building a supervisor that is responsible for monitoring its child processes.

2.6.2 Distributed Erlang

Erlang is distributed by design. When communicating between processes in Erlang it does not matter if the process is local or on another node/host.

3 Methodology

In this chapter we explain how we organized our project work and which tools and methods we use when we researched and developed our application.

3.1 Method

First of all it is important to mention that we have several years of work experience with networks and BGP. Even so, we started out with advancing our knowledge on routers and BGP by further research, with an evolutionary angle. We mainly did this by studying the BGP RFCs over and over again, and reading research articles.

We then learned the programming language Erlang using an online tutorial[40] written by Fred "MononcQc" Hébert and Programming Erlang: Software for a Concurrent world[45], by Joe Armstrong.

Our next step was setting up a test environment per author, consisting of two BGP speakers: JunOS (Olive) and ExaBGP[52]. JunOS is a FreeBSD-based network operating system from Juniper Networks that can be run in an unsupported lab environment directly on a PC or VM, called "Olive". We established an Ethernet VPN tunnel between our two lab locations over

which we let our respective Olive's peer. We connected one of the locations to an ISP to receive a live full table BGP feed to support our development. By working directly on the live data feeds we are able to avoid making incorrect assumptions about either performance of our application or the distribution of prefix lengths etc. Additionally, we're able to expose our software to the full nature of all transitive BGP attributes that exist in the wild today.

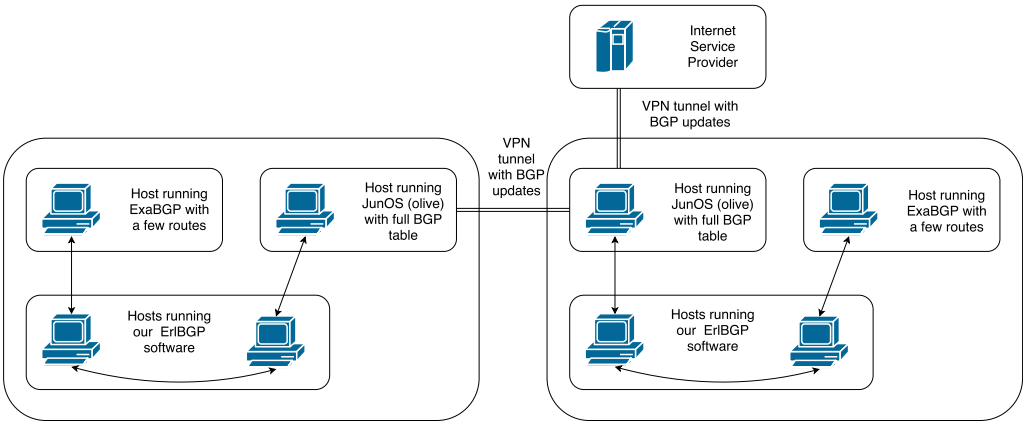


Figure 13: The lab setup

We have been using a number of distributed collaboration tools to support our work, such as Google Docs[35] for distributed report writing, Kanbanflow[24] for "To-do list management", the Git[79] protocol for source code control, Github[34] for Git repository hosting, appear.in[77] video chat for telepresence. In addition to these distributed tools, we program in Emacs and built the final report in LaTeX.

4 Results

In this chapter we present our findings including our proof-of-concept.
It is structured as follows: First we present our resulting blueprint for a complete implementation in section 4.1. Then we present the characteristics of distributed state storage of our system in more depth, in section 4.2. Following that we present our findings on parallelizing and distributing the critical path of a BGP program, e.g. the BGP UPDATE message processing in section 4.3. In section 4.4 we propose a design for implementing a distributed RIB, which builds on the learnings from section 4.2 and 4.3. The next section is 4.5 - where we present our findings on decomposition of BGP components. In the final section of the chapter, 4.6, we describe our prototype implementation, written in distributed Erlang/OTP.

4.1 Goal Design of the Total System

In this section we give a high level view of our so called Goal Design of the Total System, which encompasses pieces we have spent research time on developing prototype specifications for, as well as pieces we have prototyped in software. We have also discovered areas of future work that are of interest but further from the core functionality we're trying to achieve, that are none the less part of the Goal Design as an option.

During our research and prior to starting the experimental work, we have noted a number of weaknesses or deficits in current and popular BGP implementations such as: Run-to-completion scheduling, single-threaded implementations, lack of scale-out performance and complex perhaps overly optimized tightly coupled implementations that results in fault tolerance challenges with master/slave redundancy strategies. Additionally, from work experience, and what is essentially an established truth in the operator community, we have learned that so called Non-Stop Service Upgrades, e.g. in-flight software upgrades, essentially does not work in practice.

We are designing a novel BGP implementation to address the shortcomings we listed in the previous paragraph: Parallel (multiple layers) & distributed execution, scale-out performance, Fault Tolerance.

By using knowledge from the fields of networking, distributed systems, parallel processing and message based programming, we have arrived at a goal design of our total system.

Parallel & distributed execution:

- We decompose the internal functions of BGP into multiple smaller processes responsible for specific sub-parts of the whole
- We employ message passing for communication between these processes

Scale-out performance. By -

- identifying the critical path of a BGP speaker, and,
- utilizing per-prefix level granularity of work partitioning, and,
- partitioning work over many participating nodes,

we achieve scale-out performance characteristics that unlock substantially more processing power to be employed to process each route than previously

available in implementations.

Improved Fault Tolerance model:

- Distributed, replicated and fault-tolerant state storage
- Distributed, replicated and fault-tolerant specific RIB storage
- independent, supervised *PeerFSM* processes, 1 per peering session, with minimalistic internal state, suitable for replication

We define fault tolerance as being able to hide or repair errors such that peers do not learn an error did in fact occur.

We develop a prototype, proof-of-concept implementation of a decomposed BGP speaker implementation in a feasibility study. Figure 14 shows how our prototype can fit in on a real-world system.

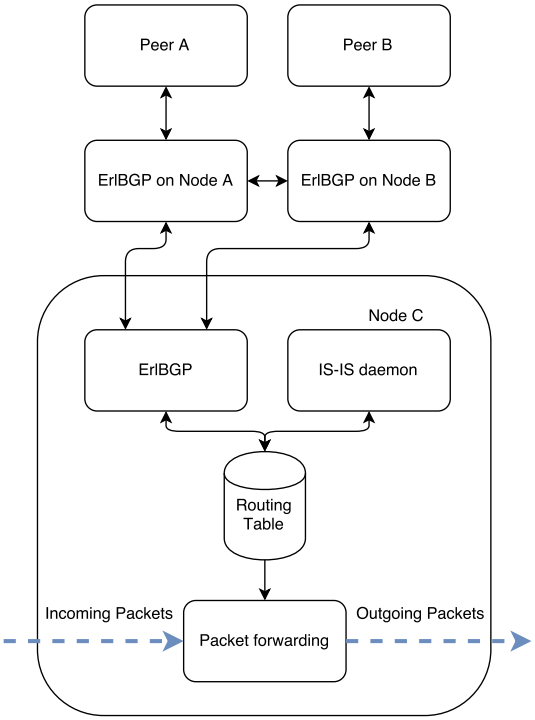


Figure 14: Prototype

4.2 Distributed state storage systems

Our system design requires two distinct distributed state storage systems:

- Distributed, highly available, strongly consistent state storage - i.e. configuration state and similar
- Distributed, highly available, strongly consistent RIB

Highly available: Replication such that failures of individual member nodes does not make the service unavailable.
Strongly consistent: All "up" nodes in the system agree on the state. I.e. once the cluster has ACK:ed a write to a storage client, it has reached its desired replication factor. In the case of state storage, the simplest way to orchestrate writes while delivering this availability-guarantee is to block them until they have achieved the desired replication factor.

The state storage could be served by a more generic key/value type storage mechanism, such as the Zookeeper library, whereas the RIB has additional requirements, for example scaling efficiently and high in both storage capacity (bytes of memory per route stored) and performance (updates per second).

In both cases, consumers of the storage systems must see consistent data. I.e. a read operation must always return the latest available version of an object that has been written anywhere to the cluster. Thus the requirement of strong consistency.

The RIB however, is on the critical data path, and there is a much higher requirement for high throughput. Here there are gains to be made with asynchronous messaging, which requires the client to keep a buffer with sent messages until they've been ACK:ed. In addition to asynchronous messaging, to achieve speedup in performance we need to make sure clients can parallelize writes to the storage system.

4.3 Parallelized and Distributed UPDATE message processing

In this section we begin with describing what the (performance) critical path of a BGP speaker program is. Then we show that this path is fully parallelizable and distributable and what consequences such a design would have on e.g. the RIB.

4.3.1 Critical data path - BGP UPDATE message processing

The critical path in a BGP speaker is the processing that occurs between having received an UPDATE message from a peer, until the BGP speaker

has possibly disseminated the NLRI's to other peers.

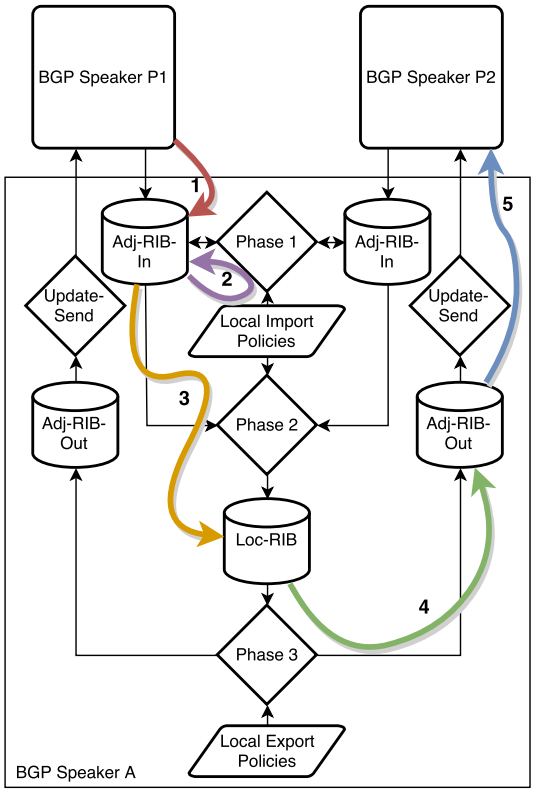


Figure 15: BGP UPDATE message processing

It involves the following steps:

1. *BGP Speaker P1* sends an UPDATE message to *BGP Speaker A*.

2. *BGP Speaker A* parses the UPDATE message and stores its NLRI's in *P1's* corresponding Adj-RIB-In, and marks any routes in the Withdrawal section of the UPDATE message for removal.

3. The *BGP Decision Process* runs its first phase - and computes preferences for newly added routes to Adj-RIB-In.

4. Directly upon completion of phase 1, phase 2 - best path selection - commences. This phase validates new routes' validity and uses local policy to determine which routes to install into Loc-RIB.

5. Phase 3 commences upon completion of Phase 2 and decides, using protocol specification and local policy, which routes to export to the

various peer speakers' respective Adj-RIB-Out's. This includes routes that are no longer to be announced - e.g. withdrawn.

6. Finally, the Update-Send process creates outgoing UPDATE message for each peer speaker that so requires, i.e. having NLRI's or Withdrawn routes to communicate.

The RFC in chapter 9 gives a high level model of a reference implementation of the BGP Decision and Update-Send Process. Per usual, other implementations are also allowed, as long as they exhibit the same externally visible behavior.

4.3.2 Parallelization of Critical Path

In this section we address our RQ.2, "Can the critical path of a BGP daemon be parallelized?" by showing the feasibility of parallelization of the critical path execution. We show that equivalent behavior of the RFC's description can be achieved without the use of locking.

The reference high-level implementation mentions, for the various phases, among other:

- Lock the Adj-RIB-In's while phase 1 is running
- Upon completion of phase 1, immediately start phase 2
- Lock the Loc-RIB while phase 2 is running
- When running phase 2, prohibit phase 3 from running
- When finishing phase 2, start phase 3
- When running phase 3, prohibit phase 2 from running

The RFC suggests using locking in phase 2 and 3.

We define five key properties of the NLRI / Withdraw processing:

1. The main computations on a prefix are done without consideration for any other prefix
2. Work units can be partitioned down to the unit of a single prefix
3. NLRI's and Withdraws from a peer concerning a single prefix must be applied in order

4. Phase 2 of the Decision Process performs a route resolvability check
5. BGP additionally specifies a route aggregation functionality

These properties are derived from the RFC's 9th chapter.

From property 2 we learn that we cannot for example run two best path selection computations for two routes to one common prefix in parallel - computations pertaining a common prefix must be ordered (serialized). From property 1 we learn that we need not take into account other prefixes or routes when computing on a prefix. From property 3 we note there are no strict ordering requirements on UPDATEs from different peers, and, for UPDATE's from a specific peer it can only be inferred from the locking that there are serial ordering requirements on a per prefix basis.

Based on property 1, 2 and 3, alone, it is clear updates pertaining to distinct prefixes are independent from each other. This means parallelization, this far, is possible.

Property 4 simply means that a route lookup or equivalent is required on the next hop address in order to determine the validity of the route to a specific prefix. This is a read-only requirement on the RIB, or parts of the RIB depending on implementation. No locking is required when performing route lookups.
Property 5 relates to a feature where a BGP speaker in phase 3 of the Decision Process may choose to aggregate neighboring routes, that share certain information, to a combined less specific route, before they are placed in the Adj-RIB-Outs.

When a BGP speaker is configured, e.g. by policy, to aggregate routes in certain prefixes or from certain peers, it will generate less specific routes that covers the more specific member routes. For example, if a BGP speaker is configured to aggregate routes under 192.0.2.0/24 and it has received only 192.0.2.0/25, no aggregate route should be generated, since it would provide false reachability information as the BGP speaker lacks reachability information regarding 192.0.2.128/25. If and when reachability information regarding 192.0.2.128/25 arrives to the speaker, it can attempt to generate the aggregating 192.0.2.0/24 route, since the entire prefix is covered by contributing member routes.
The validation steps are read-only, but the generated route needs to be written to the relevant Adj-RIB-Out's.

Considering these five properties overall, it is in fact possible to perform update processing in parallel. As long as addition and removal of routes to a specific prefix are done in per sending peer order, there is no need for an implementation to lock the RIB for writes as this processing is performed. A simple FIFO queue fulfills the ordering requirements as there is no need in BGP to guarantee ordering between updates to different prefixes. It is possible to consider an UPDATE message's contents' traversal through the Critical Path as "atomic" units of work - there is no hard requirement to perform batch processing the way described by the RFC.

4.3.3 Distributed processing of Critical Path

In this section we validate that UPDATE's message processing can in addition to be parallelized, also be distributed over multiple hosts, i.e. if implementing a distributed RIB where route information is partitioned over multiple hosts with no shared memory.

The difference between simple multi-threaded execution on one host and execution on multiple hosts is primarily the lack of shared memory in the latter case. The largest consequence of distributed computation is therefore the added latency of network communication, which makes synchronization events, including locking, very expensive in terms of performance. In order to reduce round trip latency over the network, which synchronization events result in, it is necessary to not rely on synchronized state over multiple distributed hosts in the Decision Process computations. The first property of the previous section supports this. E.g. only the route resolvability and aggregation functions require consideration taken of other prefixes, and primarily it is reads that are required.

A BGP speakers' main routing table is responsible for mapping routes' next hop addresses to interfaces, whereas the BGP RIB only has to validate that the next hop addresses actually resolves to interfaces - physical interface information is not a direct concern of the BGP speaker.

When the state of a next hop address changes, for example due to the interface's link going down, the main routing table must inform the BGP about this fact. In this event, but also in the case of an indirect next hop address pointing to a route that is withdrawn, the Decision Process has to re-run for all affected routes. It is up to an implementation to make the necessary computation time-, memory- and latency-tradeoffs in implementing this.

For good performance of a distributed system such as a distributed RIB, it is important to consider what state is useful to keep at each participating node to avoid network communication, while considering what added state synchronization complexity doing so leads to. A good candidate of such state is state that is changed infrequently but used often in the Decision Process. The policy information is a top candidate, as well as local system routing table data, i.e. locally connected interfaces, their prefixes and state.

4.4 A Distributed RIB

In this section we present a novel draft scheme for implementing a Distributed RIB, based on the findings in the previous section and those used in an earlier object data storage system.

To achieve our fault tolerant requirement we are required to replicate the state contained in the system, including the RIB contents, such that node failure or failures - depending on configured redundancy - does not lead to data loss.

4.4.1 Replication strategy overview

The replication requirement means more than 1 node must keep replicas of the written information. However, if all nodes are keeping all replicas, in memory, we have reduced the possible storage capacity of the system to that of the node of the participating nodes with the least amount of available memory, which is contrary to our capacity scalability requirement.
Instead, we could map each RIB onto a subset of all available nodes. Since the storage system must be able to heal when a node fails, the map must be dynamic such that a failed node can be replaced by another node. However, this still maps the entirety of a single RIB onto a set of nodes, fully replicated to each node, which does not allow us to scale capacity beyond the limitations of a single participating node. Additionally, if multiple RIBs are mapped to a single node, without care for the amount of available memory, this becomes a problem. A more scalable approach, then, is to make the unit of replication more granular, for example each individual prefix.

What is required then is a function that maps a prefix to a set of nodes that it should be replicated to:

$$f(prefix) = \{n_1, n_2, ..., n_i\} \tag{4}$$

What are the objectives with such a mapping function?

We claim it should:

- Provide a balanced and adaptive replication to all cluster nodes
- Enable the user to control the amount of replication
- Enable the storage system to dynamically maintain the replication status, per prefix stored

CRUSH have achieved a simplification of the above general problem by splitting it into two functions. One that maps a so called *placement group* to a deterministically priority ordered list of nodes:

$$f(pg_i) = \{n_{i1}, n_{i2}, ..., n_{ij}\} \tag{5}$$

And another that maps an object to a *placement group*, rather than to nodes directly:

$$f(object) = pg_i \tag{6}$$

This approach decouples the objects from the replication infrastructure. There are many (>million) objects and relatively few PGs (<million). This reduces both memory and computation requirements for the replication infrastructure significantly. If the user defines a replication level of 3, CRUSH returns $\{n_{i1}, n_{i2}, ..., n_{ij}\}$ with more than 3 OSDs in a pseudo-random manner that additionally adheres to the placement requirements of the CRUSH map. It then selects the first 3 available nodes to store data on. The first available node is selected as the *primary* node, and all read/write operations goes through it, providing a serialization point which fulfills the per-prefix ordering requirements. An effect of this design is that when nodes become unavailable, CRUSH instructs the system to begin replicating to the next available nodes according to the list of nodes.

4.4.2 Distributed RIB and Routing Storage Daemons cluster

By adopting a similar design as CRUSH and RADOS, we arrive at a logical structure as follows:

- The widest construct is the *cluster*
- A *cluster* consists of a number of *Routing Storage Daemons (RSDs)* and a number of *monitor daemons*
  - *RSDs* have a defined capacity in terms of available memory

- A *cluster* defines a number of *RIB instances (RI)*
- An *RI* defines a number of *placement groups (PGs)*
- Each *RI's PGs* are mapped to a deterministic priority-ordered list of *RSDs*
- Each *RI* consists of a dynamic set of *Adj-RIB-Ins*, *Adj-RIB-Outs* and a *Loc-RIB*.
  - This the conceptual representation - for efficiency the RIB data structure will maintain them within the nodes of a single n-way Trie, or similar.
- Routes are stored to *RIs* using *cluster client code* that writes directly to the *primary* (i.e. first available) *RSD*, which acts as a serialization point.

Similarly to the monitors in RADOS, the cluster's monitor daemons keeps track of the state of the cluster at all times, including the state of the RSDs, the RIs as well as the state of the RI's PGs.

The critical path for an UPDATE message through our prototype design of a Distributed RIB (D-RIB) becomes similar to that shown in Figure 16.

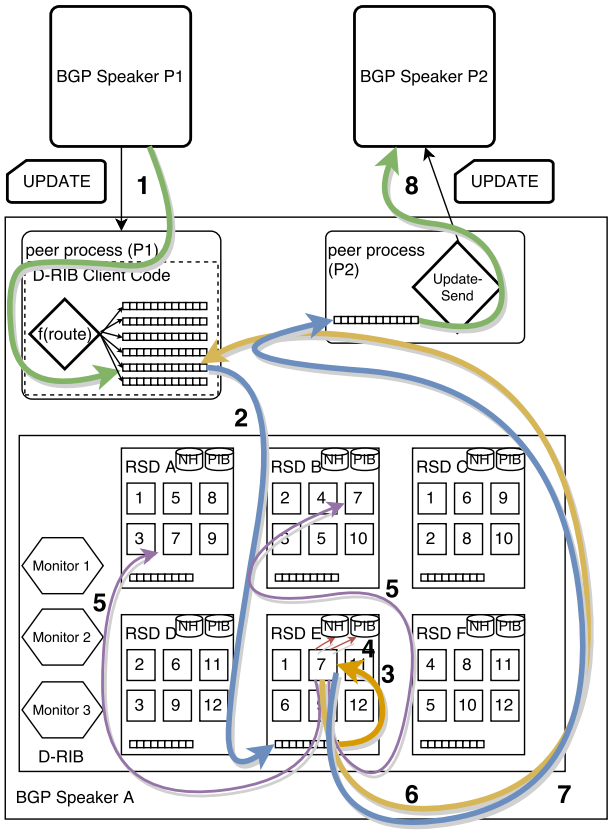


Figure 16: Distributed RIB critical path operation

An example processing flow as per figure 16 contains the steps:

1. *BGP Speaker A* receives an UPDATE message from *BGP Speaker P1* in its corresponding *peer process*, parses an NLRI from it, runs *"write(NLRI=<nlri>)"*, which calls the hashing function to learn that the *placement group* for this NLRI is *7*, then runs the local lookup function to see that the *primary RSD* for *PG 7* is *RSD E* and thus puts it in the output buffer towards *RSD E*.

2. *BGP Speaker A's* peer process asynchronously transmits the NLRI to *RSD E* as a write operation, where the command and payload is parsed and it is placed in the input write queue,

3. The write processor on *RSD E* sees that the write operation was to *PG 7*, and directs it there,

4. When the operation reaches *PG 7*, phase 1, 2 and 3 of the Decision Process is run, where the local policy database is consulted for import

policy. Route resolvability checks are also done, and in the example above there's a hit on a next hop table cache local to the node, which avoids network communication for the check,

5. *RSD E* then sends the *result* of the resulting operation on its underlying data structure, after the decision process communication phase 1 and 2, to the replica PG's, in this case housed on *RSD A* and *RSD B*, respectively

6. *BGP Speaker A*, where asynchronous communication is in use, does not remove the operation from its *P1 peer process* output queue until it receives an acknowledgement, which *RSD E* now transmits to peer *P1's peer process*,

7. RSD E has finished phase 3 and the resulting change to the Adj-RIB-Out is moved to an output queue in the peer process for P2,

8. The *peer process* for P2 performs the Update-Send process and constructs UPDATE messages using information reduction techniques, etc., based on its input queue,

Using this type of design, in the example we have 12 unique PG's in 6 RSD's. As only a PG's primary node performs the Decision Process that means potentially 6 RSD processes and 12 PG threads that can process the incoming routes, from multiple peers, achieving fairly good load balancing.

The end-to-end latency is bounded by step 2 (one-way) + 5 (round trip) + 7 (one way), in cases of ideal TCP communication, except for cases with indirect next hops where a network route resolvability check is necessary in phase 2. Asynchronous communication where appropriate can hide the latency by filling a pipeline of requests, increasing throughput when processing a large number of messages.

From a network latency experiment shown in Appendix B, we note an average ping-pong round trip latency over TCP of 74 microseconds for a 100 byte message over GbE networking. Thus, in the end-to-end example in the previous paragraph, we could expect a lower bound application-to-application communication latency of $74 \cdot (1/2 + 1 + 1/2) = 148$ microseconds. It might not seem like much, but during this time a regular 3.0 GHz CPU will have passed 444 000 cycles.

While end-to-end UPDATE processing latency, from received UPDATE to

resulting sent UPDATE, will inherently always be higher for a distributed RIB implementation than a fast regular single process implementation, provided asynchronous messaging is used successfully in the implementation, the communication *processing* overhead will be more than compensated by increased availability of processing power. Conversely, since the network communication processing overhead is a fair amount of cycles per route processed, it is in relative terms inexpensive to increase the amount of policy processing per route.

We obtain redundancy and therefore fault tolerance of the data stored in the RIB, and we have decoupled the heavy processing parts of a BGP program from the rest of the program. This means that under stress, i.e. resource (CPU) starvation, we can prioritize e.g. keepalive sending and processing as more important than for example route processing, which will avoid at least one type of common problem in traditional BGP implementations that leads to peer sessions going down, causing even more CPU load due to the increase of required processing.

Implementing this distributed RIB remains a future work effort. We implement a mock-up RIB in our prototype.

4.5 Decomposition of BGP with multi-process, distributed Erlang/OTP prototype

In this section we present our findings on identifying relatively distinct components within a BGP speaker, suitable for breaking up into components with distinct roles.

We have identified that a BGP speaker consists of the following components:

- Finite State Machine with timers and events
- Socket IO processing and BGP message parsing
- RIBs
- BGP UPDATE message processing with locally configured policy
- Communication with system main routing table
- Configuration state storage
- Configuration management

- User interfaces / API:s

These distinct roles in most cases have only very loose coupling with each other, often with fairly clear and clean interfaces in between each other. In other words they lend themselves to be implemented as separate modules or components. In our choice of language, Erlang/OTP, with its light-weight processes, it is straight forward to implement these components as distinct processes. An immediate consequence of doing this is the loss of shared memory between the different processes, which mandates the need of defining clear API's between processes, and furthermore assists in understanding which pieces of the actual code belongs where.

4.6 *ErlBGP* - multi-process, distributed Erlang/OTP prototype

In this section we present our prototype implementation using Erlang/OTP applications and supervisor trees. We present our decomposition and motivate our choices. We also describe which parts, in accordance with our scope and limitations that remain to be implemented.

4.6.1 Decomposition into multiple processes

We have chosen to implement our distributed BGP implementation, *ErlBGP*, by decomposing it into the following processes:

- erlbgp - in which we have implemented:
  - User interfaces / API:s
  - Configuration management
- erlbgp_peerrecv - in which we have implemented:
  - Socket IO processing and BGP message parsing
- erlbgp_peerfsm - in which we have implemented:
  - Finite State Machine with timers and events
- erlbgp_worker (RIB) - in which we have implemented:
  - RIBs
  - BGP UPDATE message processing with locally configured policy

- erlbgp_tcpserv - in which we have implemented:
  - Socket IO

We have chosen to keep the Finite State Machine as simple as possible and perform socket IO, BGP message processing and UPDATE message handling outside of this process. We, by following the critical path of the BGP UPDATE message, have made a decomposition that scales out to many processes as number of peers increase.
We have *not* implemented the Distributed RIB described in section 4.4, instead our RIB as implemented now is a rudimental implementation for development and testing purposes, that neither performs any advanced local import/export policy processing nor have we fully implemented the BGP Decision Process. It is left as future work to implement the Distributed RIB.

These processes are managed using an Erlang/OTP supervisor tree:

- erlbgp_supersup
  - erlbgp_peersup
  - erlbgp_tcplistensup

The following components are not implemented:

- Communication with system main routing table
- Configuration state storage

As described in section 4.2, the state storage in a distributed application can for example be implemented using an appropriate library such as Apache Zookeeper, discussed in section 2.5. We need communication with the system's main routing table in order to perform the route resolvability check of phase 2 of the decision process.

4.6.2 ErlBGP walkthrough

In this section we perform a walkthrough of the components in ErlBGP and how they interact with each other. We also show how the processing of the critical path is performed, i.e. BGP UPDATE message processing.

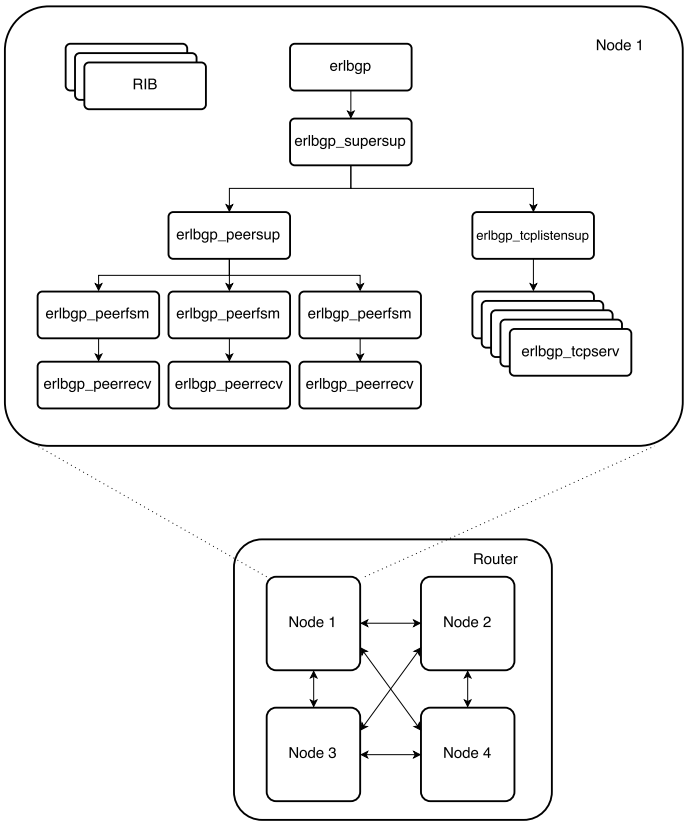


Figure 17: ErlBGP as implemented

Erlang/OTP programs with supervisor trees tend to look similar to *Node 1* in the picture above. We will now walk through the various components and describe them.

4.6.2.1 Distributed Erlang & Multiple Nodes

Using Distributed Erlang we support running multiple nodes with one instance such as shown by *Node 1* in each of them. Each node carries its very own supervisor tree, else we would still be suffering from *single point of failure* risks, if we instead used a global supervisor process that manages other nodes. Our code is written such that the processes can communicate with each other on any node, and Erlang takes care of the Inter-Process Communication (IPC) messaging.

There are two caveats to this:

1. We currently only globally register the first mockup RIB to start, and

the distributed application will therefore always refer to just this node - there is no replication or similar, and

2. we have not yet implemented distributed configuration management.

4.6.2.2 The ErlBGP application and a node's supervisor tree

On each node, there is a top process, *erlbgp*. *erlbgp* contains some API calls to manage the program, and on first instantiation on a node, creates the *erlbgp_supersup*, a supervisor of supervisors. At this point it manages two supervisors, the peer supervisor, *erlbgp_peersup*, and a TCP listener supervisor, *erlbgp_tcplistensup*, for every local IP address that has a peer configured.

erlbgp_tcplistensup keeps a number of child processes ready ("pre-forked"), waiting for an incoming TCP connection from a peer. When receiving an incoming connection, one of the *erlbgp_tcpserv's* are unblocked and creates a new *erlbgp_peerfsm* via a call to the peer FSM supervisor, *erlbgp_peersup*.

4.6.2.3 The erlbgp_peerfsm and erlbgp_peerrecv relationship

When a FSM is created, it also creates a linked child process, an *erlbgp_peerrecv*, a process in charge of the socket communication including outbound message formatting and inbound message parsing. In the case of an incoming connection, the *erlbgp_tcpserv* will transfer socket ownership of the accepted socket to the correct *erlbgp_peerrecv* process. A TCP listener supervisor can of course be involved in setting up multiple peer connections.

When configuring a peer, ErlBGP will in addition to enabling a TCP listener supervisor with its child processes, if the peer is configured as *active*, instantiate a local *erlbgp_peerfsm* and initialize it.

The *erlbgp_peerrecv* process will parse incoming UPDATE messages and forward the contained data to the RIB using Erlang terms. This assures that any parse errors will at worst only crash the victim *erlbgp_peerrecv* process.

4.6.2.4 Critical path and BGP UPDATE message processing

Our BGP UPDATE message processing path looks as follows:

1. *Peer process A* receives an UPDATE message from *Peer A*. *peerrecv* will process the message and extracts any contained path attributes, any contained NLRI and any contained Withdrawn route.

2. *peerrecv* generates BGP event #27, *UpdateMsg*, and sends to *peerfsm*, which restarts the hold timer in the FSM,

3. *peerrecv* sends the NLRI and their attributes and withdrawn routes to the worker process, which adds to or removes them from the Adj-RIB-In belonging to *Peer A*. It then performs part of the Decision Process and possibly installs them into the Loc-RIB, and also the Adj-RIB-Out of *Peer B*,

4. The worker process notifies the *peerrecv* of *Peer process B* about the new information,

5. *peerrecv* sends the new information to *Peer B*

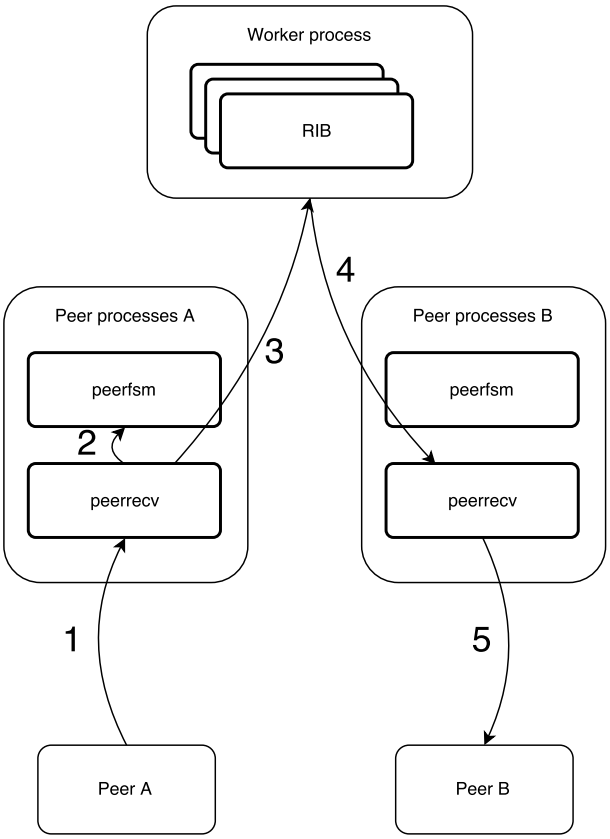


Figure 18: ErlBGP: Update path

This is largely according to our goal design, with the exception of the lack of the Distributed RIB and the main system routing table communication. It works for us for the moment but there are some immediate changes we

need to perform. For instance, when receiving a full BGP table that is to be propagated to *Peer B*, step 4 leads to the sending of approximately 550 000 messages to the Erlang inbox of *peerrecv*. This only takes a couple of seconds.
Then however, step 5, when generating the UPDATE messages to send to *Peer B*, ExaBGP in our test setup, takes longer than 90 seconds. We haven't yet determined the cause for this long processing time. We currently suspect that our Update-Send implementation in *peerrecv* has poor performance.

Additionally, currently, in step 2, *peerrecv* will generate one *UpdateMsg* events for every UPDATE message *Peer A* sends to our ErlBGP. Our plan is to add a dampening effect that reduces the amount of *UpdateMsg* events sent - it is not strictly necessary to inform the FSM thousands of times per second.

4.6.3 ErlBGP Usage Examples

In this section we show a couple of sample screenshots from the application. We show that the distributed system can connect to a peer and conversely, that a peer can connect to the distributed system. Additionally, we show that we can maintain established peers over time and that our system exhibits some of the fault tolerance properties, and autonomously restarts dead processes.

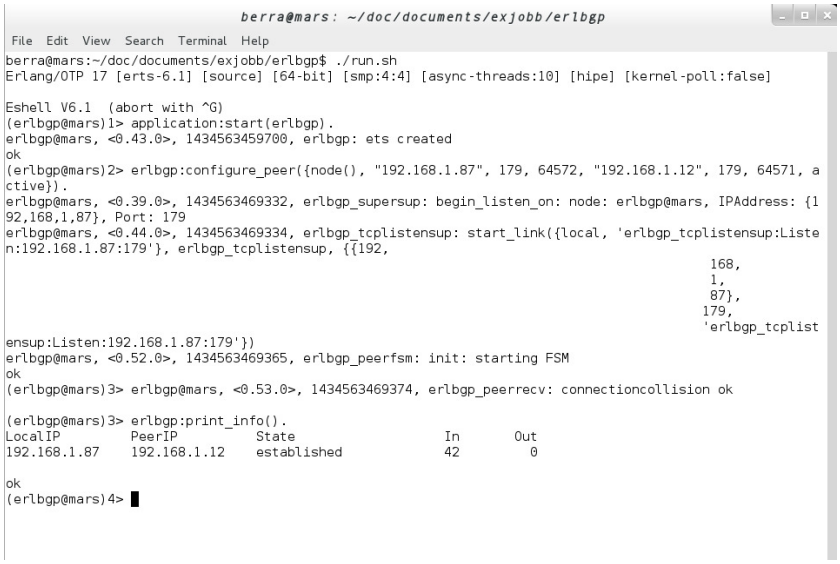


Figure 19: Screenshot of an ErlBGP node with one peer

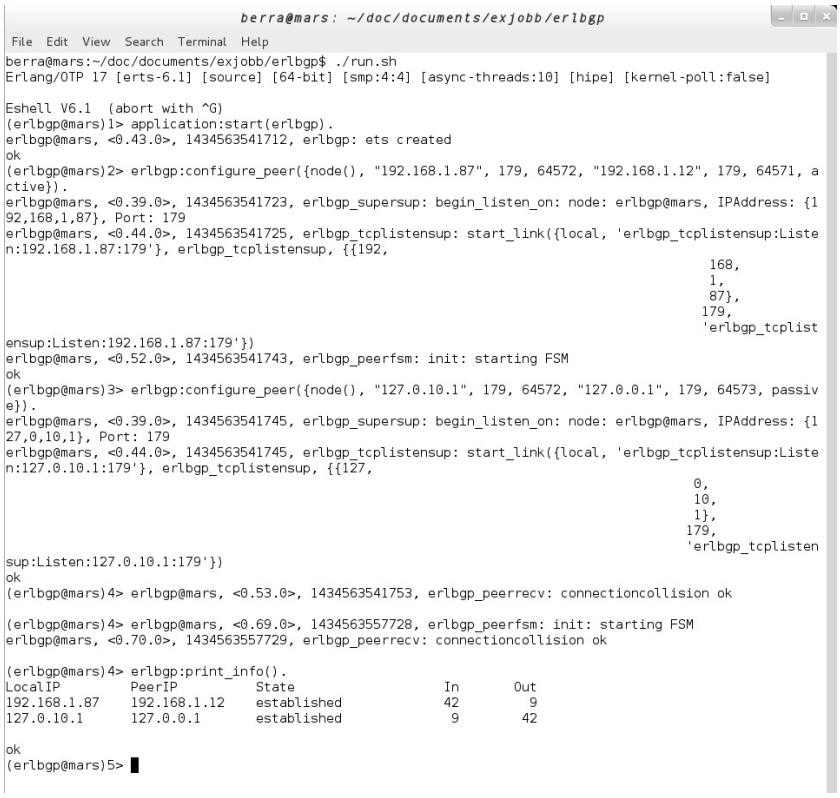


Figure 20: Screenshot of an ErlBGP node with two peers

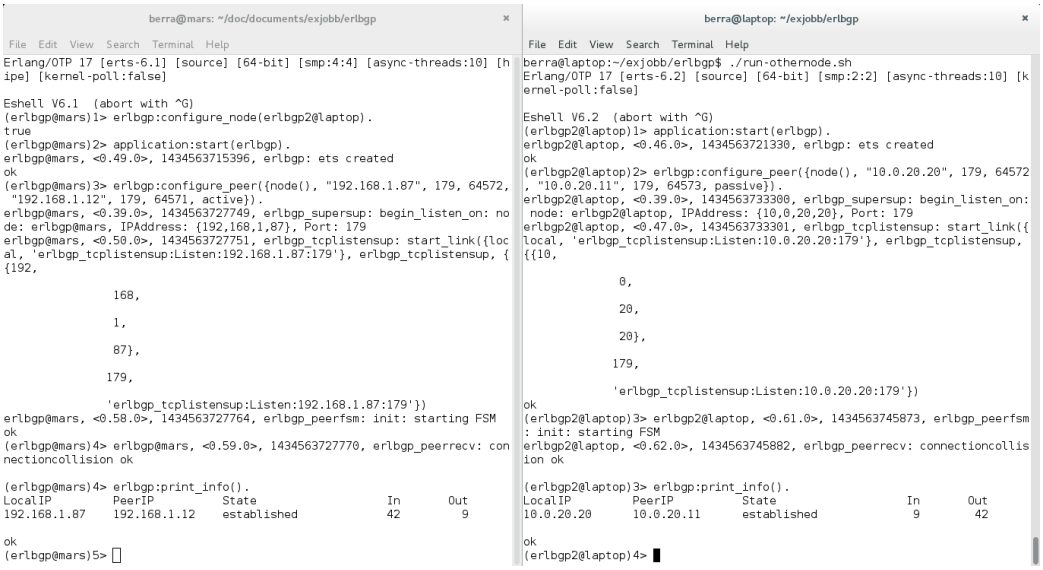


Figure 21: Screenshot of two nodes of ErlBGP with one peer each. We haven't implemented a unified multi-node output view from one command yet.

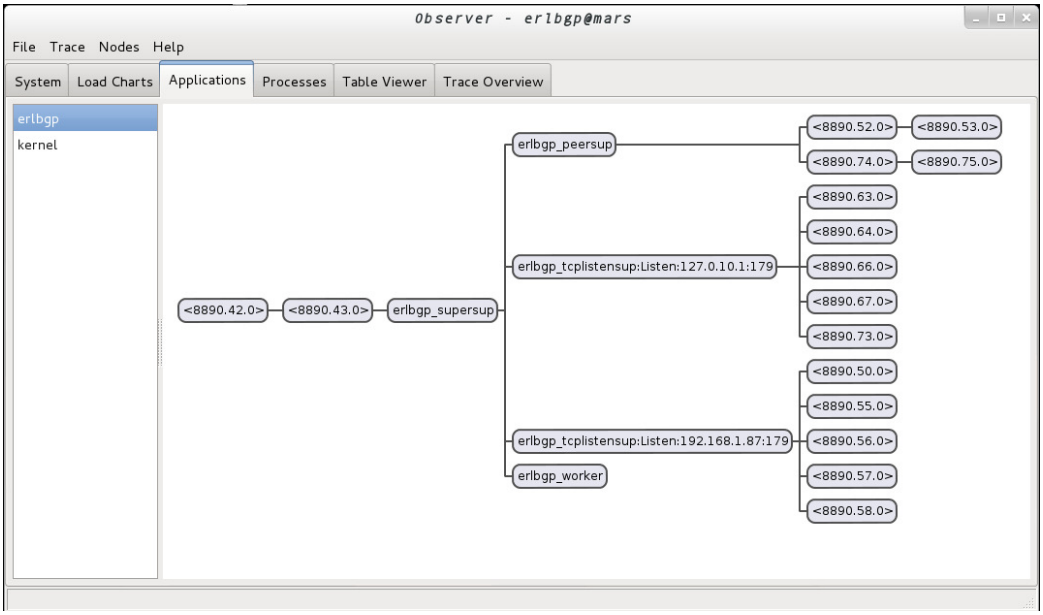


Figure 22: Screenshot of Erlang Observer

Screenshot22 of the Erlang Observer connected to the host "mars", showing the ErlBGP application and its various processes. It has two configured

peers, seen as the two processes *<8890.52.0>* and *<8890.74.0>* respectively. These are *peerfsm* processes. They have each one *peerrecv* process, *<8890.53.0>* and *<8890.75.0>*, respectively. Also worth noting is that ErlBGP has created two *erlbgp_tcplistensup* processes, one per configured peer as they are peering using different local IP addresses. They each use a pool of 5 *erlbgp_tcpserv* processes to handle incoming connections.

5 Discussion

This thesis has presented an approach for achieving both fault tolerance and performance scalability of a BGP routing daemon, by decomposing the critical path at a granular level. The result is a partition of work that enables parallel processing on each route processed in a distributed RIB implementation.

The prototype implementation, ErlBGP, is written in a high-level language which is on the order of 10-20 times more time efficient to develop in than a regular low-level language such as C. Section 5.1 drills down in this aspect.

The thesis has accomplished something to the authors knowledge not attempted before, in making a completely novel implementation of BGP in order to support a granular level of decomposition and distributed processing. The thesis is demonstrably distinct from the two perhaps most closely related works, Hidell[42] and ONOS[50]. See section 5.2 for more details.

The risk of this approach is that the effort is not continued, but there are a couple of strategies available to mitigate that risk, including Open Sourcing the platform. It has not been decided to Open Source the software at this point.

This work opens up for a number of continuation efforts that are outlined in section 5.3.

5.1 On using the right tool for the job

We believe a routing protocol suite implemented in a concurrent, message passing language suitable for distributed operations offers the foundation for realizing this proof of concept. We try to map out the pieces of the puzzle required to accomplish this new architecture.

We pick a certain relatively self-contained and clear piece of the puzzle, BGP, and show that it is possible, with very limited resources, to come pretty far towards the objective, by just using the right tools for the job.

In Appendix C we have attached a *sloccount* comparison between our program and Quagga's *bgpd*. It reports *bgpd* consists of 46000 lines of 100% ANSI C, whereas our program consists of 3500 lines of 100% Erlang. Our implementation is not complete, but it is not unlikely that we will only grow the program 5x before we have feature parity with *bgpd*. And *bgpd* does not contain all relevant code for actually running Quagga with *bgpd* — for example Zebra is required as well, for communicating with the main routing table, managing configuration, etc. Zebra adds 21000 lines of 100% ANSI C code additionally.

People unaware of the benefits of programming in a language such as Erlang, often criticize it for being slow. The fact is that the language does run in a VM, like Java, but it is not inherently slow if used correctly. For most programs, the majority of the code is not involved in a program's critical data path. Similarly, a program's top "hot spots", e.g. parts in a program where for example simply the ability to do very tight loops on a CPU is a great contributing factor to the performance of the program, are often solvable in Erlang as well with so called Native Implemented Functions - C handoffs.

Programs written in low level languages, such as C, are easy to make relatively fast as single process programs, but also easy to make unsafe and buggy. Programmers spend a considerable amount of time getting memory management right, even though there are good tools to guide them. Writing concurrent programs in C is challenging. Programmers instead tend to write structured and modularized source code, but often it will still end up a relative large binary responsible for, for example, the entire BGP protocol, for all peers, including all communication with them.
This leads to the so called "blast zone" of failures within the program to be essentially the entire program. A bug in a section of the program may cause unexpected side-effects on the remainder of the program, or even crash the program.
With Erlang, such programs can be broken apart into multiple smaller processes, each responsible for only a small part of the whole. What's more, it is easy to achieve this using Erlang.

In the context of BGP, there are only a few such pieces, primarily the different parts of processing UPDATE messages and applying route policy to

the RIBs. These parts of a BGP implementation are mostly stressed when a peering session goes up or down, since it provokes a long queue of messages to be handled. A typical C implementation would simply queue them in a single-threaded program and process one route at a time. With a concurrent and distributed Erlang implementation, the RIBs would be accessible and replicated according to some strategy to multiple processing elements, and there are inherent abilities that makes it easy to let a number of workers work in parallel to consume these queues.

The lack of multithreading or multiprocessing in typical C implementations of BGP are fitting to the earlier years when resources in general in a router were really scarce, processors were rather slow and most of all, always single-core.

Using a functional language such as Erlang, it is within time scope of a Master's Thesis to write a BGP router daemon from scratch in completely new ways compared to before. Free of the burden of having written a BGP router daemon before, we can enable ourselves to solve the typical problems in ways made possible by this toolbox for concurrent software developing, Erlang.

5.2 Differences with (HIDELL) and (ONOS)

The method Hidell employed to control route dissemination to the various backends are all based on regular "bgpd" route filters in the frontend process, a method that scales quite poorly and is fairly imprecise. Hidell concludes the route filter implementation scales poorly. The reason for this is simply that Hidell relies on the BGP Decision Process to perform load sharing to the actual BGP backend process, and the policy implementation in Quagga, or any other BGP daemon, aren't designed for this scale.

The main differentiation between our work and that of Hidell can be summarized as:

- We implement BGP from scratch with full control over all aspects of the software - Hidell makes patches on top of "bgpd" and is thus constrained by it
- We do not follow ForCES - Hidell does
- Our equivalent to Hidell's frontend process is made using two processes - we have a dedicated process for socket ownership, message parsing and

message formatting - Hidell have essentially full BGP implementation running in the frontend process, yet still propagate the BGP UPDATEs as-is to the service processes. Our method breaks apart the UPDATE messages very early.

- The communication between our peerrecv process and our state storage is not based on a regular BGP peer with BGP UPDATE messages - Hidell's is.

In (ONOS) a distributed Network Controller is demonstrated. ErlBGP has some similarities with ONOS in its highly distributed nature of, but the use case is different - ErlBGP is not a network controller, it's a BGP routing daemon.

5.3 Future work

ErlBGP, being a prototype, requires further work in several areas, some of which are discussed here. In order to become a fully functional production capable distributed routing daemon, a number of areas need further work:

- Distributed policy processing,
- A number of BGP extensions adopted for distributed operation, Multiprotocol, 4-byte ASN, VPN, MPLS
- Main routing table communication process

A prototype of the main routing table communication process can easily be developed using off the shelf open source projects[86][72].

5.3.1 Improve Fault Tolerance

ErlBGP needs further work for fully deliver on the proposed fault tolerance. There are two areas of work, which regards to the state storage and distributed RIB, in addition to BGP Graceful Restart. Once those are addressed, the work can continue towards TCP Socket Migration architectures, which is described in 5.3.4.

5.3.2 Implement Distributed RIB

This thesis describes a novel approach to a distributed RIB implementation. It has to be developed to deliver on the total system design.

5.3.3 Further Performance Optimizations

The critical data path needs optimizations in memory, latency and processing. Erlang supports using what is called Native Interface Functions, NIFs, that are C interfaces. These should be deployed in processing hot spots or to represent efficient, pointer based, storage data structures.

When developing a routing daemon, care must be taken to prioritise tasks and processes against each other, else there is a risk that less time critical bulk computations such as BGP UPDATE message handling starves out more important keepalive messages, and similar. The Erlang VM supports scheduling priorities between tasks.

5.3.4 Look into TCP Socket Migration solutions

The BGP Graceful Restart mechanism allows for BGP speakers to implement various methods of performing in-service upgrades and in general reduce the network churn when faults happen. Thus it improves fault tolerance. It is however not transparent to the receiving speaker. In a truly fault tolerant implementation of BGP, a single BGP speaker can hide or mask faults from peers entirely, including at the TCP layer.

To mask failures at the TCP endpoint on a host, or in our case on a set of hosts, the hosts' TCP implementation need to support TCP Socket Migration.

This is useful both for planned maintenance of sockets when for example a node will restart and also for crashes. The feature is much assisted by an ability to also direct incoming IP packets to the correct host, before and after a socket relocation. This would be quite straightforward to accomplish if a system allows granular control over how to move IP addresses around between nodes, or, perhaps even better, steer incoming packets based on a TCP 4-tuple: IP source address, IP destination address, TCP source port, TCP destination port[71][51][25][5].

5.3.5 Configuration Management and API

If the project were to be moved forward, there would be a real need for proper configuration management and APIs to manage configuration. These are today standardized into the Network Configuration Protocol[28] (NETCONF), which would be an obvious protocol to implement, of which there are more

than one Erlang implementations already[1][29][64]. Network operators today expect configuration management to provide them with versioning, candidate configurations with commit/rollback functions[81].
Fortunately, a configuration data modeling language, called YANG, have been standardized that fits with NETCONF[17]. Using YANG and NETCONF together today increases the chances that existing management tools can work with a system. There are Erlang implementations of YANG available as well[3].

5.3.6 Application of heavy BGP policy computations

Using ErlBGP opens up for more heavy policy processing per prefix, which can benefit Internet stability, especially for core Internet routers that have tremendous packet throughput but still haven't seen improvements in policy processing capabilities.

The lack of CPU resources for policy processing in the core of the Internet affects its stability severely. Existing methods to address this include using Internet Routing Registry data, (a set of databases where networks register their prefixes and peering relationships), to compute valid registered paths of announcements. This method is completely dependent on the quality of the data and very good heuristics need to be available before an operator can decide to reject a received route as malicious.

Malicious routes are addressed by the PATHSEC work of the IETF[47], which adds cryptographic computations to each UPDATE and NLRI. Both the Internet Registry Data approach and the PATHSEC approach require good computational power attached to routers, and the closer to the core of the Internet a router is the more important is it to handle high BGP UPDATE message volumes quickly. The two approaches have in common that it is intractable to pre-compute all possible available paths due to extremely large storage requirements.

Both approaches need on-demand computation in a router. Integrating this into ErlBGP is an interesting task.

5.3.7 Application of ErlBGP to a real-world router hardware platform

In practical terms, a real-world hardware routing platform needs many more functions than a BGP daemon, such as chassis inventory and management

functions, internal routing protocols, main routing table.

ErlBGP could become a piece in such a puzzle. This requires additional extensions of the main routing table communication software.

6 Conclusion

In this thesis a design has been defined and tested that address two core challenges with previous BGP implementations:

1. Fault tolerance, and
2. Performance and scalability

The very core challenge of satisfying both problems simultaneously were early identified as a problem of decoupling the internal functions of BGP. Three Research Questions were defined:

RQ.1 Is it feasible to implement a distributed BGP daemon that address the two challenges above?

RQ.2 Can the critical path of a BGP daemon be parallelized?

RQ.3 Is it feasible to partition a RIB over multiple nodes?

Answers to the questions above:

RQ.1 As evident by this thesis work, it is feasible with a very limited amount of resources, as well as technically feasible.

RQ.2 Yes — the critical path of a BGP daemon can definitely be parallelized, in fact it even lends itself well to the point.

RQ.3 Yes, this thesis extends earlier work which has shown it feasible but not optimal, to show that it is promising in terms of performance by using asynchronous unicast communication in the cluster.

The major findings of the reports are the answers to the research questions above, shown via both a prototype ErlBGP implementation as well as a novel design for a distributed RIB, supporting granular decomposition of route updates to per-prefix levels.

In addition to these research questions the following six criteria were defined at the very beginning of the thesis work:

1. Without crashing any other process, crash and restart the policy process.
2. Without crashing any other process, crash and restart the Peer A process.
3. Without crashing any other process, crash and restart the Peer B process.
4. Be able to run the processes on separate hosts.
5. Establish peers with Peer A and Peer B.
6. Receive, process and propagate BGP UPDATE messages between these two peers.

The ErlBGP prototype meets each of these criteria.

All known implementations of BGP processes the protocol's critical path serially in a single process with poor fault tolerance implementation. Lack of processing capabilities results in much simpler sanity checking of route announcements than otherwise possible. This, due to the trusting nature of BGP, in turn leads to instability in the Internet with outages as a result, and not just in the edge of the Internet, but in its core as well.
The ErlBGP architecture provides better fault tolerance and scale-out processing capabilities than existing implementations of BGP. Using the ErlBGP architecture would allow for an important improvement in the stability of the Internet as a whole.

7 Bibliography

[1] Netconf implementation using Erlang. URL `https://code.google.com/p/netconf-erlang/`.

[2] Network benchmarking utility. URL `https://github.com/Mellanox/sockperf/`.

[3] Yang parser and validator. URL `https://github.com/travelping/yang`.

[4] B. Adamson, C. Bormann, M. Handley, and J. Macker. Nack-oriented reliable multicast (norm) transport protocol. Technical report, IETF RFC5740, 2009.

[5] A. Agapi, K. Birman, R. M. Broberg, C. Cotton, T. Kielmann, M. Millnert, R. Payne, R. Surton, and R. van Renesse. Routers for the cloud. *Internet Computing*, 15(5), 2011.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review*, 38(4):63–74, 2008.

[7] Apache Software Foundation. Apache ZooKeeper. URL `https://zookeeper.apache.org/`.

[8] Arista Networks, Inc. EOS: The next generation extensible operating system. URL `https://www.arista.com/assets/data/pdf/EOSWhitepaper.pdf`.

[9] J. Armstrong. Programming Erlang: Software for a concurrent world. Pragmatic Bookshelf, 2007.

[10] J. Aweya. On the design of IP routers part 1: Router architectures. *Journal of Systems Architecture*, 46(6):483–511, 2000.

[11] Y. Bachar. Introducing 6-pack: the first open hardware modular switch. URL `https://code.facebook.com/posts/717010588413497/introducing-6-pack-the-first-open-hardware-modular-switch/`.

[12] Y. Bachar. Facebook wedge. URL `http://files.opencompute.org/oc/public.php?service=files&t=cbc56082857b154a157aa46cdcb4b9b1`.

[13] Y. Bachar. Introducing 6-pack, the first open hardware modular switch. URL https://www.youtube.com/watch?v=uiiLMtO9nW8.

[14] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[15] T. Bates, P. Smith, and G. Huston. CIDR report. URL http://www.cidr-report.org/as2.0/#General_Status.

[16] T. Bates, R. Chandra, D. Katz, and Y. Rekhter. Multiprotocol extensions for BGP-4. Technical report, IETF RFC4760, 2007.

[17] M. Bjorklund. Yang — a data modeling language for the network configuration protocol (netconf). Technical report, IETF RFC6020, October, 2010.

[18] V. Bollapragada, C. Murphy, and R. White. *Inside Cisco IOS software architecture*. Cisco Press, 2000.

[19] E. A. Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[20] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 15–28. USENIX Association, 2005.

[21] S. Calo, E. M. Nahum, and D. Verma. Software implementation of network switch/router, Apr. 12 2013. US Patent App. 13/861,850.

[22] Cisco Systems, Inc. Graceful restart, non stop routing and IGP routing protocol timer manipulation solution overview. URL http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/high-availability/solution_overview_c22-487228.html.

[23] C. Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[24] CodeKick AB. Kanbanflow. URL https://kanbanflow.com/.

[25] A. Cohen, S. Rangarajan, and H. Slye. On the performance of TCP splicing for URL-aware redirection. In *Proceedings of the 2nd conference on USENIX Symposium on Internet Technologies and Systems-Volume 2*, pages 11–11. USENIX Association, 1999.

[26] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router plugins: A software architecture for next generation routers. In *ACM SIGCOMM Computer Communication Review*, volume 28, pages 229–240. ACM, 1998.

[27] A. Elangovan, A. S. Agilandan, and A. Lakkshmanan. Building a robust software based router. In *Advanced Computing, Networking and Informatics-Volume 2*, pages 181–188. Springer, 2014.

[28] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. Network configuration protocol (netconf). Technical report, IETF RFC6241, 2011.

[29] Ericsson AB. Erlang manual: ct_netconfc. URL `http://www.erlang.org/doc/man/ct_netconfc.html`.

[30] A. Farrel. Forwarding and control element separation Working Group. URL `https://datatracker.ietf.org/doc/charter-ietf-forces/`.

[31] N. Feamster, J. Rexford, and E. Zegura. The road to SDN: An intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review 44.2 (2014)*

[32] S. Garrison. White box switch OS. URL `http://www.pica8.com/white-box-switches/white-box-switch-os.php`.

[33] S. Garrison and S. Crehan. The rise of white-box switches. URL `http://www.infoworld.com/article/2609937/sdn/the-rise-of-white-box-switches.html`.

[34] GitHub, Inc. Github. URL `https://github.com/`.

[35] Google. Google docs. URL `https://docs.google.com/`.

[36] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM computer communication review*, volume 39, pages 51–62. ACM, 2009.

[37] M. Handley, O. Hodson, and E. Kohler. Xorp: An open platform for network research. *ACM SIGCOMM Computer Communication Review*, 33(1):53–57, 2003.

[38] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov. Designing extensible IP router software. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 189–202. USENIX Association, 2005.

[39] D. Harley. Data center: Battleground for traditional vs white box ethernet switch vendors. URL `http://tinyurl.com/nkk8wk3`.

[40] F. Hebert. Learn you some Erlang. URL `http://learnyousomeerlang.com/`.

[41] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. Elastictree: Saving energy in data center networks. In *NSDI*, volume 10, pages 249–264, 2010.

[42] M. Hidell. Decentralized modular router architectures. Diss. KTH Royal Institute of Technology, 2006.

[43] G. Huston. BGP in 2014. URL `https://labs.apnic.net/?p=568`.

[44] K. Ishiguro. Quagga routing suite. URL `http://www.nongnu.org/quagga/`.

[45] A. Joe. Programming Erlang: Software for a concurrent world. *Pragmatic Bookshelf*, 2007.

[46] M. J. Karol, M. G. Hluchyj, and S. P. Morgan. Input versus output queueing on a space-division packet switch. *Communications, IEEE Transactions on*, 35(12):1347–1356, 1987.

[47] S. Kent and A. Chi. Threat model for BGP path security. Technical report, IETF RFC7132, 2014.

[48] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[49] T. Koponen, M. Casado, N. Gude, and J. Stribling. Distributed control platform for large-scale production networks, Sept. 9 2014. US Patent 8,830,823.

[50] U. Krishnaswamy. Open network operating system. URL `http://www.slideshare.net/umeshkrishnaswamy/open-network-operating-system`.

[51] B. Kuntz and K. Rajan. MIGSOCK: Migratable TCP socket in Linux. Carnegie Mellon University, Information Networking Institute, 2002.

[52] T. Mangin. ExaBGP. URL `https://github.com/Exa-Networks/exabgp`.

[53] C. Matsumoto. Hyperscale + white box switches: Prepare to be shocked. URL `https://www.sdxcentral.com/articles/news/hyperscale-white-box-switches-prepare-to-be-shocked/2015/04/`.

[54] N. McKeown, A. Mekkittikul, V. Anantharam, and J. Walrand. Achieving 100% throughput in an input-queued switch. *Communications, IEEE Transactions on*, 47(8):1260–1267, 1999.

[55] R. McMillan. Cisco patches bug that crashed 1% of Internet. URL `http://www.computerworld.com/article/2515200/networking/cisco-patches-bug-that-crashed-1--of-internet.html`.

[56] C. Metz. You cant have Google's Pluto switch, but you can have this. URL `http://www.wired.com/2013/03/big-switch-indigo-switch_light/`.

[57] D. Mills. Exterior gateway protocol formal specification, DARPA network working group report. Technical report, IETF RFC904, M/A-COM Linkabit, 1984.

[58] J. C. Mogul, P. Yalagandula, J. Tourrilhes, R. McGeer, S. Banerjee, T. Connors, and P. Sharma. API design challenges for open router platforms on proprietary hardware. *ACM HotNets-VII*, 2008.

[59] D. Munjal. So google is building a 10GbE switch? URL `http://blogs.cisco.com/datacenter/so_google_is_building_a_10gbe_switch`.

[60] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 39–50. ACM, 2009.

[61] Open Compute Project Foundation. Open compute project. URL `http://www.opencompute.org/`.

[62] I. Pepelnjak. Oversized as paths: Cisco IOS bug. URL `http://blog.ipspace.net/2009/02/oversized-as-paths-cisco-ios-bug.html`.

[63] C. Petersen. Micro-server card. URL http://www.opencompute.org/wiki/Motherboard/SpecsAndDesigns#System_on_Chip_.28SoC.29_Servers.

[64] R. Pienaar. Netconf library for Erlang. URL https://github.com/FlowForwarding/enetconf.

[65] M. Prince. Today's outage post mortem. URL https://blog.cloudflare.com/todays-outage-post-mortem-82515/.

[66] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker. Software-defined internet architecture: Decoupling architecture from infrastructure. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 43–48. ACM, 2012.

[67] F. Y. Rashid. Bug in Juniper router firmware update causes massive internet outage. URL http://tinyurl.com/7279kjx.

[68] Y. Rekhter and T. Li. A Border Gateway Protocol 4 Technical report, IETF RFC1771, 1995.

[69] Y. Rekhter, T. Li, and S. Hares. Border gateway protocol 4. Technical report, IETF RFC4271, 2006.

[70] S. Sangli, E. Chen, R. Fernando, J. Scudder, and Y. Rekhter. Graceful restart mechanism for BGP. Technical report, IETF RFC4724, 2007.

[71] M. Schlager. The remote socket architecture: A proxy based solution for TCP over wireless. Diss. Berlin Institute of Technology, 2004.

[72] A. Schultz. gen_netlink. URL https://github.com/travelping/gen_netlink.

[73] S. Sezer, S. Scott-Hayward, P.-K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao. Are we ready for SDN? implementation challenges for software-defined networks. *Communications Magazine, IEEE*, 51(7):36–43, 2013.

[74] A. Singla and B. Rijsman. OpenContrail architecture document. URL http://www.opencontrail.org/opencontrail-architecture-documentation/.

[75] J. Sonderegger, O. Blomberg, K. Milne, and S. Palislamovic. *Junos High Availability: Best Practices for High Network Uptime.* " O'Reilly Media, Inc.", 2009.

[76] M. Tahir, M. Ghattas, D. Birhanu, and S. N. Nawaz. *Cisco IOS XR Fundamentals*. Pearson Education, 2009.

[77] Telenor Digital. appear.in. URL `https://appear.in/`.

[78] M. Terpstra. SDN and gartner's hype cycle. URL `http://www.plexxi.com/2015/01/sdn-gartners-hype-cycle/`.

[79] L. Torvalds. Git. URL `https://git-scm.com/`.

[80] Q. Vohra and E. Chen. BGP support for four-octet AS number space. Technical report, IETF RFC4893, 2007.

[81] S. Wallin and C. Wikstrom. Automating network and service configuration using netconf and yang. URL `http://www.tail-f.com/wordpress/wp-content/uploads/2013/02/Tail-f-Presentation-Netconf-Yang.pdf`.

[82] D. Walton, A. Retana, E. Chen, and J. Scudder. Advertisement of multiple paths in BGP. URL `https://datatracker.ietf.org/doc/draft-ietf-idr-add-paths/`.

[83] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[84] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 122. ACM, 2006.

[85] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*, pages 35–44. ACM, 2007.

[86] U. Wiger. libnl binding. URL `https://github.com/Feuerlabs/netlink`.

[87] R. Yavatkar and S. Bakshi. Distributed implementation of control protocols in routers and switches, Jan. 4 2002. US Patent App. 10/039,279.

[88] E. Zmijewski. Longer is not always better. URL `http://research.dyn.com/2009/02/longer-is-not-better/`.

[89] E. Zmijewski. Reckless driving on the Internet. URL http://research.dyn.com/2009/02/the-flap-heard-around-the-world/.

A Decision Process Pseudo Code

```
1  phase_1_decision_process(rib adj_rib_in, list (route)new_routes,
       preference LOCAL_PREFERENCE, policy bgp_policy):
2
3  lock(adj_rib_in)
4
5  for each route in new_routes:
6    if is_internal(route):
7      if preference(bgp_policy(route)) != LOCAL_PREFERENCE:
8        adj_rib_in.setpreference(route, bgp_policy(route))
9      else:
10       adj_rib_in.setpreference(route, LOCAL_PREFERENCE)
11   else: // route is external
12     if preference(bgp_policy(route)) == INELIGBLE_ROUTE:
13       adj_rib_in.setineligible(route)
14     else:
15       adj_rib_in.setpreference(bgp_policy(route))
16
17 unlock(adj_rib_in)
```

Listing 1: Pseudo-code representation of the RFC4271 Decision Process Phase 1 (chapter 9.1.1).

```
1  phase_2_decision_process(rib loc_rib, list (rib)adj_ribs_in,
     preference LOCAL_PREF, policy bgp_policy, asn LOCAL_AS):
2
3  for each rib in adj_ribs_in:
4    lock(adj_rib_in)
5
6  set_mutex(phase_2_or_3)
7
8  for adj_rib_in in adj_ribs_in:
9    for route in adj_rib_in.eligible_routes:
10     if unresolveable(route.nexthop) or
          would_become_unresolveable(route.nexthop):
11       set_unresolveable(route)
12       continue
13     if contains(route.as_path, asn):
14       continue
15     if has_highest_preference(route, adj_ribs_in) // being the
          only route is
16       and                                          // having
             highest preference
17       is_tie_breaking_winner(route):
18       loc_rib.install(route)
19
20 for each rib in adj_ribs_in:
```

```
21    unlock(adj_rib_in)
22
23    clear_mutex(phase_2_not_3)
```

Listing 2: Pseudo-code representation of the RFC4271 Decision Process Phase 2 (chapter 9.1.2).

```
1  phase_3_decision_process(list (rib) adj_ribs_out, rib loc_rib,
       policy bgp_policy):
2
3  set_mutex(phase_2_or_3)
4
5  for route in loc_rib:
6    if unresolvable(route):
7      continue
8    for adj_rib_out in adj_ribs_out:
9      case pass_policy(bgp-policy, route, adj_rib_out) of
10       reject:
11         adj_rib_out.withdraw_if_exist(route)
12       accept:
13         adj_rib_out.install(route)
14
15 clear_mutex(phase_2_or_3)
```

Listing 3: Pseudo-code representation of the RFC4271 Decision Process Phase 3 (chapter 9.1.3).

B Gigabit Ethernet Ping-Pong Latency test

Using Mellanox' sockperf [2], we performed a simple ping-pong latency test over TCP, where a 100 byte message is sent from a test client to a test server, and then sent back by the test server.

Test equipment per the below data:

Server host: Lenovo X1 Carbon with i7-4600U CPU @ 2.10GHz, Intel I218-LM Network Controller,
Server kernel: Linux kepler 3.16.0-4-amd64 #1 SMP Debian 3.16.7-ckt9-3 deb8u1 (2015-04-24) x86_64 GNU/Linux
Client host: Asus P9X79 WS with i7-3820 CPU @ 3.60GHz, Intel 82574L Network Controller
Client kernel: Linux pishuli 3.2.0-4-amd64 #1 SMP Debian 3.2.65-1+deb7u2 x86_64 GNU/Linux
Switch: Netgear GS116E

```
1 anticimex@kepler /usr/local/src/sockperf/src [sockperf_v2] $ ./
    sockperf server --tcp -i 192.168.120.128 -p 5000
2 sockperf: == version #2.5. ==
3 sockperf: [SERVER] listen on:
4 [ 0] IP = 192.168.120.128 PORT =  5000 # TCP
5 sockperf: Warmup stage (sending a few dummy messages)...
6 sockperf: [tid 16109] using recvfrom() to block on socket(s)
```

Listing 4: Server output

```
1 anticimex@pishuli /usr/local/src/sockperf/src [
    sockperf_v2] $ ./sockperf pp --tcp -i
    192.168.120.128 -p 5000 -m 100 -t 10s
2 sockperf: == version #2.5.exported ==
3 sockperf[CLIENT] send on:sockperf: using recvfrom() to
    block on socket(s)
4
5 [ 0] IP = 192.168.120.128 PORT =  5000 # TCP
6 sockperf: Warmup stage (sending a few dummy messages)
    ...
7 sockperf: Starting test...
8 sockperf: Test end (interrupted by timer)
9 sockperf: Test ended
```

```
10 sockperf: [Total Run] RunTime=10.100 sec; SentMessages
   =61539; ReceivedMessages=61538
11 sockperf: ========== Printing statistics for Server No:
   0
12 sockperf: [Valid Duration] RunTime=10.000 sec;
   SentMessages=60938; ReceivedMessages=60892
13 sockperf: ====> avg-lat=463.248 (std-dev=17175.763)
14 sockperf: # dropped messages = 0; # duplicated messages
    = 0; # out-of-order messages = 38
15 sockperf: Summary: Latency is 463.248 usec
16 sockperf: Total 60892 observations; each percentile
   contains 608.92 observations
17 sockperf: ---> <MAX> observation = 774111.815
18 sockperf: ---> percentile  99.99 = 774080.229
19 sockperf: ---> percentile  99.90 =  149.773
20 sockperf: ---> percentile  99.50 =  104.138
21 sockperf: ---> percentile  99.00 =   98.445
22 sockperf: ---> percentile  95.00 =   89.097
23 sockperf: ---> percentile  90.00 =   85.468
24 sockperf: ---> percentile  75.00 =   83.422
25 sockperf: ---> percentile  50.00 =   81.392
26 sockperf: ---> percentile  25.00 =   79.310
27 sockperf: ---> <MIN> observation =   64.171
```

Listing 5: Client output, using blocking sockets

```
 1 anticimex@pishuli /usr/local/src/sockperf/src [sockperf_v2] $ ./
   sockperf pp --tcp -i 192.168.120.128 -p 5000 -m 100 -t 10s --
   nonblocked
 2 sockperf: == version #2.5.exported ==
 3 sockperf[CLIENT] send on:sockperf: using recvfrom() to block on
   socket(s)
 4
 5 [ 0] IP = 192.168.120.128 PORT =  5000 # TCP
 6 sockperf: Warmup stage (sending a few dummy messages)...
 7 sockperf: Starting test...
 8 sockperf: Test end (interrupted by timer)
 9 sockperf: Test ended
10 sockperf: [Total Run] RunTime=10.100 sec; SentMessages=68534;
   ReceivedMessages=68533
11 sockperf: ========= Printing statistics for Server No: 0
12 sockperf: [Valid Duration] RunTime=10.000 sec; SentMessages
   =67898; ReceivedMessages=67898
13 sockperf: ====> avg-lat= 73.612 (std-dev=5.491)
14 sockperf: # dropped messages = 0; # duplicated messages = 0; #
```

```
      out-of-order messages = 0
15 sockperf: Summary: Latency is 73.612 usec
16 sockperf: Total 67898 observations; each percentile contains
      678.98 observations
17 sockperf: ---> <MAX> observation =  528.548
18 sockperf: ---> percentile  99.99 =  234.570
19 sockperf: ---> percentile  99.90 =  112.286
20 sockperf: ---> percentile  99.50 =   95.101
21 sockperf: ---> percentile  99.00 =   89.179
22 sockperf: ---> percentile  95.00 =   80.247
23 sockperf: ---> percentile  90.00 =   77.279
24 sockperf: ---> percentile  75.00 =   75.222
25 sockperf: ---> percentile  50.00 =   73.113
26 sockperf: ---> percentile  25.00 =   70.800
27 sockperf: ---> <MIN> observation =   48.711
```

Listing 6: Client output, using non-blocking sockets

C "sloccount" comparison

```
1  anticimex@kepler /usr/local/src/quagga [master] $ sloccount bgpd
       /
2  Creating filelist for bgpd
3  Categorizing files.
4  Finding a working MD5 command....
5  Found a working MD5 command.
6  Computing results.
7
8  SLOCDirectorySLOC-by-Language (Sorted)
9  45949    bgpd              ansic=45949
10
11 Totals grouped by language (dominant language first):
12 ansic:          45949 (100.00%)
13
14 Total Physical Source Lines of Code (SLOC)                 =
       45,949
15 Development Effort Estimate, Person-Years (Person-Months) =
       11.13 (133.54)
16  (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
17 Schedule Estimate, Years (Months)                          = 1.34
       (16.06)
18  (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
19 Estimated Average Number of Developers (Effort/Schedule)  = 8.32
20 Total Estimated Cost to Develop                            = $
       1,503,251
21  (average salary = $56,286/year, overhead = 2.40).
22 SLOCCount, Copyright (C) 2001-2004 David A. Wheeler
23 SLOCCount is Open Source Software/Free Software, licensed under
       the GNU GPL.
24 SLOCCount comes with ABSOLUTELY NO WARRANTY, and you are welcome
        to
25 redistribute it under certain conditions as specified by the GNU
        GPL license;
26 see the documentation for details.
27 Please credit this data as ``generated using David A. Wheeler's
       'SLOCCount'.''
```

Listing 7: *sloccount* of Quagga's bgpd [44]

```
1  anticimex@kepler /usr/local/src/quagga [master] $ sloccount
       zebra/
2  Creating filelist for zebra
3  Categorizing files.
4  Finding a working MD5 command....
5  Found a working MD5 command.
6  Computing results.
```

```
 7
 8  SLOCDirectorySLOC-by-Language (Sorted)
 9  21396    zebra            ansic=21396
10
11  Totals grouped by language (dominant language first):
12  ansic:         21396 (100.00%)
13
14  Total Physical Source Lines of Code (SLOC)                =
        21,396
15  Development Effort Estimate, Person-Years (Person-Months) = 4.99
        (59.85)
16   (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
17  Schedule Estimate, Years (Months)                         = 0.99
        (11.84)
18   (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
19  Estimated Average Number of Developers (Effort/Schedule)  = 5.06
20  Total Estimated Cost to Develop                           = $
        673,738
21   (average salary = $56,286/year, overhead = 2.40).
22  SLOCCount, Copyright (C) 2001-2004 David A. Wheeler
23  SLOCCount is Open Source Software/Free Software, licensed under
        the GNU GPL.
24  SLOCCount comes with ABSOLUTELY NO WARRANTY, and you are welcome
        to
25  redistribute it under certain conditions as specified by the GNU
        GPL license;
26  see the documentation for details.
27  Please credit this data as ``generated using David A. Wheeler's
        'SLOCCount'.''
```

Listing 8: *sloccount* of Quagga's zebra

```
 1  anticimex@kepler ~/code/erlang/erlbgp [master] $ sloccount src/
 2  Creating filelist for src
 3  Categorizing files.
 4  Finding a working MD5 command....
 5  Found a working MD5 command.
 6  Computing results.
 7
 8  SLOCDirectorySLOC-by-Language (Sorted)
 9  3473     src              erlang=3473
10
11  Totals grouped by language (dominant language first):
12  erlang:         3473 (100.00%)
13
14  Total Physical Source Lines of Code (SLOC)                =
        3,473
15  Development Effort Estimate, Person-Years (Person-Months) = 0.74
        (8.87)
```

```
16   (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
17  Schedule Estimate, Years (Months)                    = 0.48
        (5.73)
18   (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
19  Estimated Average Number of Developers (Effort/Schedule)  = 1.55
20  Total Estimated Cost to Develop                      = $
        99,858
21   (average salary = $56,286/year, overhead = 2.40).
22  SLOCCount, Copyright (C) 2001-2004 David A. Wheeler
23  SLOCCount is Open Source Software/Free Software, licensed under
        the GNU GPL.
24  SLOCCount comes with ABSOLUTELY NO WARRANTY, and you are welcome
        to
25  redistribute it under certain conditions as specified by the GNU
        GPL license;
26  see the documentation for details.
27  Please credit this data as ``generated using David A. Wheeler's
        'SLOCCount'.''
```

Listing 9: *sloccount* of ErlBGP