NETTLE: A LANGUAGE FOR CONFIGURING BGP NETWORKS

Andreas Voellmy
OGST - April 7, 2009

Prof. Dr. Otto Wilhelm Thomé, Flora von Deutschland, Österreich und der Schweiz. 1885, Gera, Germany
BGP: FLEXIBLE & DANGEROUS!

• BGP is the Internet’s interdomain routing protocol; it is designed to be flexible and allow a variety of policies to be expressed by networks.

• This flexibility also makes BGP complex, and misconfiguration is common: Mahajan et al estimate that **50% of network outages are due to misconfigurations**;

• Furthermore, BGP routers typically play a crucial role in a network’s connectivity, and misconfiguration can have serious consequences.
Wow, AS7007!

- From: Stephen A Misel
- Date: Fri Apr 25 13:20:40 1997

I happened to be in one of our 7505 routers this afternoon when POP -- all of a sudden most of the internet disappeared! I immediately thought it was me, but looked around and saw this AS7007 broadcasting MY routes! It wasn't for all of our network space -- We have several /18's here, and it seemed only the first /24 of each CIDR was affected. When I found a workstation at the end of the /18, we got the whois info for 7007 -- Florida Internet Exchange, and called them.

They claimed to have a customer broadcasting some bad routing information and unplugged their router. A few moments later, the internet stabilized and I started seeing real routes.

Correct me if I'm wrong, but:

(1) We're going to read about this in EVERY computer magazine, newspaper and TV as "the end of the internet?"

(2) Access lists by backbone providers *should* have prevented this.

(3) Does or does not the RADB and other routing registries (MCI's, etc) prevent this?

I bet this hole will be patched up real soon!

Steve
- - - - - - - - - - - - - - - -
Our overall goal is to help operators configure BGP according to their intentions, reducing misconfigurations and improving productivity.

Domain-specific languages (DSL) help programmers construct correct programs by providing a language that matches the way domain experts think about their domain.

A domain-specific embedded language (DSEL) is a DSL embedded in a host language; this technique reduces the cost of implementation and allows the DSL to inherit the general features of the host language.

We have built Nettle, a DSEL in Haskell, in which BGP configurations for a whole network can be described, and a compiler which translates a Nettle program into router configuration files for the eXtensible Open Router Platform (XORP).
BGP KNOBS AND CONTROLS

• BGP provides lots of “controls” and “knobs”

• Nettle makes those “controls” available in Haskell

• We can now compose “controls” to make new “controls”
THIS TALK

• Intro to BGP: understanding BGP’s “controls”.

• Intro to Nettle: how we embed BGP’s controls in Haskell.

• Three examples: defining high-level controls.
A computer network consists of a set of nodes, each having an address, and a set of links connecting nodes.

- **Forwarding** is the process of sending packets to the next hop node.

- **Routing** is the process that establishes the paths along which forwarded packets flow. Routing results in each node having a forwarding table.

<table>
<thead>
<tr>
<th>Destination</th>
<th>Outgoing interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
</tbody>
</table>
FORWARDING ON THE INTERNET

- **IP addresses** are 32-bit values, typically written as 4 bytes, as in a.b.c.d,

- An **address prefix**, is written a.b.c.d/e, and denotes the subset of IP addresses.

- Forwarding is by the “longest match”, i.e. **most specific**

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Outgoing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0.0.0/0</td>
<td>1</td>
</tr>
<tr>
<td>1.1.0.0/16</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1.0/24</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Longest Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2.0.0</td>
<td>0.0.0.0/0/0</td>
</tr>
<tr>
<td>1.1.2.0</td>
<td>1.1.0.0/16</td>
</tr>
<tr>
<td>1.1.1.2</td>
<td>1.1.1.0/24</td>
</tr>
</tbody>
</table>
Internet Physical Infrastructure

Residential access
- Cable
- Fiber
- DSL
- Wireless

Campus access, e.g.,
- Ethernet
- Wireless

- The Internet is a network of networks
- Each individually administrated network is called an Autonomous System (AS)
BGP ROUTES

- BGP nodes announce routes to each other. These routes carry attributes, some of which are:
  - Address Prefix
  - Next hop address
  - AS Path, a sequence of AS numbers
  - Community attributes
EXAMPLE POLICIES

• Do not advertise routes heard from one provider to another -- no transit

• For a particular customer, only accept routes with the subnet they have been assigned -- no hijacks

• Prefer this route to my customer, but prepend my AS number several times when advertising it to others -- customer wants me to use this route, but to discourage others (but not prevent entirely) from using it.
BGP’S CONTROLS

- BGP nodes then repeat the following:
  - Collect announcements from neighbors;
  - **Choose some of the announced routes to consider** - *Use Filter*
  - **Assigns numeric preference to remaining routes** - *Preference Policy*
  - For each prefix announced, select the best one according to the decision process
  - Install best routes in the forwarding table of the router
  - **Choose some of the best routes to advertise** - *Ad Filter*
  - **Advertise these with (potentially) modified attributes** - *Ad Modifier*
BGP’S CONTROLS

- Two parts of BGP policy:
  - **Import policy**: which routes should we use?
  - **Export policy**: which routes should we offer to neighbors, and how good or bad should we make it look?

- useFilter : Neighbor × Route → Bool
- preference : Route → N
- adFilter : Neighbor × Route → Bool
- adModifier : Neighbor × Route → Route
BGP DECISION PROCESS

• The BGP decision process roughly implements shortest AS-path routing, while allowing networks to override this behavior by assigning non-default local preferences to routes.

• Of the routes known, a BGP node selects the best route for each prefix, by applying the BGP decision process, which is a lexicographic order on the following attributes:
  
  • highest local preference
  
  • shortest AS path length
  
  • ...
  
  • lowest router ID (used as a tie-breaker)
FORWARDING, REVISITED

• To forward a packet to an address outside of the network:

• Find the most specific BGP routes matching the address,

• Use the best one.

<table>
<thead>
<tr>
<th>Route</th>
<th>Prefix</th>
<th>NextHop</th>
<th>Pref</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0.0.0/0</td>
<td>10.10.10.10</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>1.1.0.0/16</td>
<td>20.20.20.20</td>
<td>80</td>
</tr>
<tr>
<td>3</td>
<td>1.1.0.0/16</td>
<td>30.30.30.30</td>
<td>90</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Route</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.1.1</td>
<td>3</td>
</tr>
<tr>
<td>1.2.0.0</td>
<td>1</td>
</tr>
</tbody>
</table>
Networks running BGP also run an internal routing protocol and these protocols interact by injecting routes into from one protocol to the other. This process is called route redistribution.

Some routes are known statically, and these may need to be injected into BGP. We view statically known routes as being computed by a static routing process.
NETTLE TUTORIAL

\[ \text{nettleProg} = \text{routingNetwork bgpNet staticNet redistPolicy} \]
BGP NETWORK

\[ \text{bgpNet} = \text{bgpNetwork myASNumber bgpConns prefs usefilter adfilter admodifier} \]
ROUTERS

\[ r1 = \text{router } r1\text{xorp} \]
\[ \text{where } r1\text{xorp} = \text{xorpRouter xorpBgpId xorpInterfaces} \]
\[ \text{xorpInterfaces} = [ \text{ifaceEth0} ] \]
\[ \text{ifaceEth0} = \text{xorpInterface "eth0" "data" [ virtualIfEth0Eth0 ]} \]
\[ \text{virtualIfEth0Eth0} = \text{let block} = \text{address 200 200 200 2 // 30} \]
\[ \quad \text{bcastAddr} = \text{address 200 200 200 3} \]
\[ \quad \text{in vif "eth0" (vifAddrs (vifAddr block bcastAddr))} \]
BGP CONNECTIONS

conn1 = externalConn r1 (address 100 100 1 0) (address 100 100 1 1) 3400
conn2 = internalConn r1 (address 130 0 1 4) r3 (address 130 0 1 6)
ROUTE PREDICATES

\[\text{nextHopEq}\ (\text{address}\ 128\ 32\ 60\ 1) \lor \text{taggedWith}\ (5000 :: 120)\]

\[\text{prefixInSet}\ [\text{address}\ 128\ 32\ 60\ 0\ //\ 24,\ \text{address}\ 63\ 100\ 0\ 0\ //\ 16] \land \text{taggedWithAtLeastOneOf}\ [5000 :: 120, 7500 :: 101]\]

\[\text{asSeqIn}\ (\text{repeat}\ (i\ 7000) \triangleright \text{repeat}\ \text{any} \triangleright (i\ 3370 ||| i\ 4010) \triangleright \text{repeat}\ (i\ 6500))\]
NEW PREDICATES

Subsets of Routes

At the core of Nettle lies a small language of route predicates for specifying sets of routes. This language is used to apply different policies to different sets of routes. The language, which is designed with an eye towards implementation on available router platforms, consists of a set of atomic predicates, a conjunction operator $\wedge$ denoting the intersection of two subsets of routes, and a disjunction operator $\vee$ denoting the union of two subsets of routes. For example,

\[
\text{nextHopEq(address opv qp tn og) } \vee \text{taggedWith(fsnnn xxx opng)}
\]

denotes the set of routes whose next hop routers have address opvlqpltnlo or those which are tagged with community attribute snnnxopnl.

Another example is

\[
\text{destInSet(address opv qp tn n) } \wedge \text{taggedWithAnyOf([snnn xxx opn, usnn xxx ono])}
\]

which denotes the set of routes for destination prefixes opvlqpltnlnmpr or tqlonnlnlnmot and which are marked with one or more of community attributes snnnxopn or usnnxonol.

The `asSeqIn` predicate allows users to specify complex conditions on BGP routes, as in the following example which requires that routes originate at AS unnn, pass through any number of networks, and finally pass through either AS qqun or rmon once followed by AS tsnnx:

\[
\text{asSeqIn repeat f unnn } \kappa \text{repeat any } \kappa \text{f } \text{qqun } ||\text{f rmon } \kappa \text{repeat f tsnnx}
\]

The full collection of predicates is shown in Figure ol.

Using Haskell's standard abstraction mechanisms, this predicate language allows users to write new predicates. For example, we can define a predicate that matches a path exactly while allowing for prepending, as follows:

\[
\text{pathIs xs = asSeqIn } \kappa \text{foldr (\lambda a r } \rightarrow \text{repeat } (i \ a) \triangleright r \text{) empty xs}
\]
USE & AD FILTERS

\[
\text{reject \ (prefixEq \ (address \ 128 \ 32 \ 0 \ 0 \ // \ 16))}
\]

usefilter \( c = \)

\[
\begin{align*}
\text{if } & c \equiv c1 \\
\text{then } & \text{reject \ ((prefixEq \ (address \ 128 \ 32 \ 0 \ 0 \ // \ 16) \ \& \ \text{taggedWith} \ (5000 ::: 120))} \\
& \quad \lor \text{taggedWith} \ (12345 ::: 100)) \\
\text{else } & \text{reject \ (asSeqIn \ (repeat any \ \triangleright \ \text{repeat} \ (i \ 7000) \ \triangleright \ \text{repeat} \ \text{any}))}
\end{align*}
\]
ROUTE PREFERENCES

\[
\text{cond} \ (\text{taggedWith} \ (5000 ::: 120)) \ 120
\]

\[
\$ \ \text{cond} \ (\text{taggedWith} \ (5000 ::: 100)) \ 100
\]

\[
\$ \ \text{cond} \ (\text{taggedWith} \ (5000 ::: 80)) \ 80
\]

\[
\$ \ \text{always} \ 100
\]
ROUTE MODIFIERS

\[ \text{tag (5000 ::: 130)} \]

\[ \text{prepend 65000} \]

\[ \text{tag (5000 ::: 130)} \triangleright \text{prepend 65000} \]

\[ \text{cond (taggedWith (5000 ::: 120)) (prepend 65000) (always (tag 1000 ::: 99))} \]

\[ \text{adMod c | c \equiv c1 = always (prepend 65000)} \]
\[ | c \equiv c2 = always (prepends 2 65000 \triangleright \text{tag (65000 ::: 120)}) \]
\[ | \text{otherwise = always ident} \]
COMPILING

```
compile nettleProg r1
```
3 EXAMPLES

• Multi-homed “stub” network
• Hierarchical BGP
• Provider policies allowing customer-controlled policy
EXAMPLE: MULTI-HOMED NETWORK

- Example from Zhang and Bartell, “BGP Design and Implementation”

- Objectives:
  - Balance traffic over high and low bandwidth links.
  - Use multiple links to provide robustness under link failures.
  - Default routing only is too coarse.
  - Request “default and partial” routes from providers.
EXAMPLE: MULTI-HOMED NETWORK

- Routes are of these types:
  - Customer of 100
  - Customer of 200
  - Customer of both 100 & 200
  - Neither customer of 100 nor customer of 200.

- To balance traffic we want
  - most traffic to flow over high, including non customers and customers of 100
  - traffic to customers of 200 (and not of 100) to flow over low-2.
EXAMPLE: MULTI-HOMED NETWORK

- Plan for link failures
  - When one link fails:
    - If low-2 fails, use high
    - If low-1 fails, traffic is unaffected
    - If high fails, balance over low-1 and low-2
  - When two links fail, use the remaining one.
EXAMPLE: MULTI-HOMED NETWORK

We can write the preference policy simply...

cond (nextHopEq (peerAddr connHigh)) 120 (always 100)

but this obscures the intentions of the policy. We would like to express this more directly.
• Balance on the most direct links available having the highest bandwidth

\[
\text{balanceByBandwidth} :: [[\text{BGPConnection}]] \rightarrow \text{Cond BGPT Preference}
\]

```haskell
import Nettle.MultiHomed
prefs = balanceByBandwidth [[connHigh], [connLow100, connLow200]]
```
EXAMPLE 2: HIERARCHICAL BGP

- Gao and Rexford found that business relationships between ASes falls into two types:
  - customer-provider
  - peer-peer
EXAMPLE 2: HIERARCHICAL BGP

- These two types strongly constrain BGP policy, both usage and advertising:
  - route preferences: customer > peer > provider
  - customer routes are advertised to all neighbors, whereas peer and provider routes may only be shared with customers
- Writing the routing configurations to satisfy these constraints is tedious and error-prone.
- We can generate these policies using Nettle!
HBGP IN NETTLE

\textbf{data} PeerType = Customer \mid Peer \mid Provider

\begin{verbatim}
hbgpAdFilter :: [(ASNumber, PeerType)]
   \rightarrow BGPCconnection r
   \rightarrow Filter BGPT
hbgpPrefs    :: [(ASNumber, PeerType)]
   \rightarrow PartialOrder ASNumber
   \rightarrow Cond BGPT Preference
\end{verbatim}
**HBGP IN NETTLE**

\[
\begin{align*}
\text{home} &= 100; \text{cust}1 = 200; \text{cust}2 = 300; \text{cust}3 = 400 \\
\text{cust}4 &= 500; \text{peer}1 = 600; \text{peer}2 = 700; \text{prov} = 800 \\
\text{peerTyping} &= [(\text{cust}1, \text{Customer}), (\text{cust}2, \text{Customer}), (\text{cust}3, \text{Customer}), \\
&\quad (\text{cust}4, \text{Customer}), (\text{peer}1, \text{Peer}), (\text{peer}2, \text{Peer}), (\text{prov}, \text{Provider})] \\
\text{prefs} &= \text{hbgpPrefs} \text{ peerTyping basicPrefs} \\
&\quad \textbf{where} \text{ basicPrefs} = [(\text{cust}1, \text{cust}2), (\text{cust}1, \text{cust}4), (\text{cust}3, \text{cust}4), \\
&\quad\quad (\text{peer}2, \text{peer}1), (\text{prov}, \text{prov})] \\
\text{adFilter} &= \text{hbgpAdFilter} \text{ peerTyping}
\end{align*}
\]
COMPILING HBGP

term impterm13 {
  from {
    as-path: "\^200\(200\ [0-9]+\)\*\(\ [0-9]*\)\)$"
  }
  then {
    localpref: 105
  }
}
...

term impterm13 {
  from {
    as-path: "\^300\(300\ [0-9]+\)\*\(\ [0-9]*\)\)$"
  }
  then {
    localpref: 104
  }
}
...

as well as export filters implementing the HBGP scope preferences, such as:
term expterm7 {
  to {
    neighbor: 130.6.1.1
    as-path: "\^700\(700\ [0-9]+\)\*\(\ [0-9]*\)\)$"
  }
  then {
    reject
  }
}

term expterm8 {
  to {
    neighbor: 130.6.1.1
    as-path: "\^800\(800\ [0-9]+\)\*\(\ [0-9]*\)\)$"
  }
  then {
    reject
  }
}
COMPILING HBGP

term impterm13 {
  from {
    as-path: o^200|{200 [0-9][0-9]}{ [0-9][0-9]}|}|ro
  }
  then {
    localpref: 105
  }
}
...
term impterm13 {
  from {
    as-path: o^300|{300 [0-9][0-9]}{ [0-9][0-9]}|}|ro
  }
  then {
    localpref: 104
  }
}
...
as well as export filters implementing the HBGP scope preferences, such as:
term expterm7 {
  to {
    neighbor: 130.6.1.1
    as-path: "^700| (700 [0-9][0-9])*( [0-9][0-9])*$"
  }
  then {
    reject
  }
}
term expterm8 {
  to {
    neighbor: 130.6.1.1
    as-path: "^800| (800 [0-9][0-9])*( [0-9][0-9])*$"
  }
  then {
    reject
  }
}
EXAMPLE 3: CUSTOMER-ADJUSTABLE POLICY

- Allow customers to adjust preference of announced routes.

<table>
<thead>
<tr>
<th>Community</th>
<th>Preference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000:80</td>
<td>80</td>
</tr>
<tr>
<td>1000:100</td>
<td>100</td>
</tr>
<tr>
<td>1000:120</td>
<td>120</td>
</tr>
</tbody>
</table>

```
cond (taggedWith 1000 ::: 80) 80
$ cond (taggedWith 1000 ::: 100) 100
$ cond (taggedWith 1000 ::: 120) 120
$ always 100
```
CUSTOMER-ADJUSTABLE POLICY

- Allow customers to control advertising:
  - Suppress by AS number - community tag indicates which peers not to advertise to based on their AS Numbers.

<table>
<thead>
<tr>
<th>Community</th>
<th>Suppress when advertising to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000:5500</td>
<td>5500</td>
</tr>
<tr>
<td>1000:5600</td>
<td>5600</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

reject (\lambda connection \rightarrow taggedWith 1000 ::: (asNumber connection))
CUSTOMER-ADJUSTABLE POLICY

• Prepending by AS Number - community tag indicates how many times to prepend when advertising a route to a specific neighbor

<table>
<thead>
<tr>
<th>Community</th>
<th>AS</th>
<th>Times to Prepend</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001:5500</td>
<td>5500</td>
<td>1</td>
</tr>
<tr>
<td>1002:5500</td>
<td>5500</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001:5600</td>
<td>5600</td>
<td>1</td>
</tr>
</tbody>
</table>
CUSTOMER-ADJUSTABLE POLICY

• We can write functions these policies in Nettle, and then apply them by applying these functions:

  adjustablePrefs homeASNum [80, 100, 120] (always 100)

  adjustablePrepending homeASNum [1, 2, 3, 4, 5] [5500, 5600] (always ident)