# RFC 8522 : Looking Glass Command Set

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 6 février 2019

Date de publication du RFC : Février 2019

https://www.bortzmeyer.org/8522.html

---

Avec plusieurs systèmes de routage, et notamment avec le protocole standard de l'Internet, BGP, un routeur donné n'a qu'une vue partielle du réseau. Ce que voit votre routeur n'est pas forcément ce que verront les autres routeurs. Pour déboguer les problèmes de routage, il est donc souvent utile de disposer d'une vue sur les routeurs des autres acteurs. C'est fait par le biais de "*looking glasses*" qui sont des mécanismes permettant de voir l'état d'un routeur distant, même géré par un autre acteur. Il n'y a aucun standard pour ces mécanismes, il faut donc tout réapprendre pour chaque "*looking glass*", et il est difficile d'automatiser la collecte d'informations. Ce RFC propose une solution (surtout théorique aujourd'hui) : une norme des commandes à exécuter par un "*looking glass*", utilisant les concepts REST.

Voyons d'abord deux exemples de "*looking glass*". Le premier, au France-IX, utilise une interface Web :

Le second, chez Route Views, utilise telnet :

```
% telnet route-views3.routeviews.org
...
route-views3.routeviews.org> show bgp 2001:678:c::1
BGP routing table entry for 2001:678:c::/48
Paths: (8 available, best #4, table Default-IP-Routing-Table)
  Not advertised to any peer
  39351 2484
    2a03:1b20:1:ff01::5 from 2a03:1b20:1:ff01::5 (193.138.216.164)
      Origin IGP, localpref 100, valid, external
      AddPath ID: RX 0, TX 179257267
      Last update: Mon Jan 28 15:22:29 2019

  46450 6939 2484
    2606:3580:fc00:102::1 from 2606:3580:fc00:102::1 (158.106.197.135)
      Origin IGP, localpref 100, valid, external
      AddPath ID: RX 0, TX 173243076
      Last update: Sat Jan 26 08:26:39 2019
...
```

Notez que le premier "*looking glass*" n'affichait que les routes locales au point d'échange alors que le second voit toute la DFZ. Les "*looking glasses*" jouent un rôle essentiel dans l'Internet, en permettant d'analyser les problèmes réseau chez un autre opérateur. (On peut avoir une liste, très partielle et pas à jour, des "*looking glasses*" existants en .)

La section 2 de notre RFC décrit le fonctionnement de ce "*looking glass*" normalisé. Il y a trois acteurs, le client, le serveur et le routeur. Le client est le logiciel utilisé par l'administrateur réseaux qui veut des informations (par exemple curl), le serveur est le "*looking glass*", le routeur est la machine qui possède les informations de routage. (Le RFC l'appelle « routeur » mais ce n'est pas forcément un routeur, cela peut être une machine spécialisée qui récolte les informations en parlant BGP avec les routeurs.) Entre le client et le serveur, on parle HTTP ou, plus exactement, HTTPS. Le client ne parle pas directement au routeur. La communication entre le serveur et le routeur se fait avec le protocole de leur choix, il n'est pas normalisé (cela peut être SSH, NETCONF, etc).

La requête se fera toujours avec la méthode HTTP `GET`, puisqu'on ne modifie rien sur le serveur. Si le serveur est `lg.op.example`, la requête sera vers l'URL `https://lg.op.example/.well-known/looking-gla` (le préfixe bien connu est décrit dans le RFC 8615[1], et ce `looking-glass` figure désormais dans le registre IANA <https://www.iana.org/assignments/well-known-uris/well-known-uris.xml>). L'URL est complété avec le nom de la commande effectuée sur le routeur. Évidemment, on ne peut pas exécuter de commande arbitraire sur le routeur, on est limité au jeu défini dans ce RFC, où on trouve les grands classiques comme ping ou bien l'affichage de la table de routage. La commande peut être suivie de détails, comme l'adresse IP visée, et de paramètres. Parmi ces paramètres :

- `protocol` qui indique notamment si on va utiliser IPv4 ou IPv6,
- `router` qui va indiquer l'identité du routeur qui exécutera la commande, pour le cas, fréquent, où un même serveur "*looking glass*" donne accès à plusieurs routeurs.
- `vrf` ("*Virtual Routing and Forwarding*"), une table de routage spécifique, pour le cas où le routeur en ait plusieurs,
- `format`, qui indique le format de sortie souhaité sous forme d'un type MIME ; par défaut, c'est `text/plain`. Attention, c'est le format des données envoyées par le routeur, pas le format de l'ensemble de la réponse, qui est forcément en JSON.

Le code de retour est un code HTTP classique (RFC 7231, section 6), la réponse est de type `application/json`, suivant le profil JSend <https://github.com/omniti-labs/jsend>, rappelé dans l'annexe A. Ce profil impose la présence d'un champ `success`, ainsi que d'un champ `data` en cas de succès. Voici un exemple où tout s'est bien passé :

```
HTTP/1.1 200 OK
Content-Type: application/json
{
     "status" : "success",
     "data" : {
       "router" : "route-server.lg.op.example"
       "performed_at" : "2019-01-29T17:13:11Z",
       "runtime" : 2.63,
       "output" : [
         "Neighbor              V         AS MsgRcvd MsgSent   TblVer  InQ OutQ  Up/Down State/PfxRcd
          2001:67c:16c8:18d1::1 4     206479   80966   79935        0    0    0 01w2d14h            2
          2401:da80::2          4      63927 12113237   79918        0    0    0 07w6d13h        62878
          2405:3200:0:23::      4      17639       0       0        0    0    0    never      Connect"
       ],
       "format" : "text/plain"
     }
}
```

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc8615.txt`

JSend impose un champ `status` qui, ici, indique le succès. Le champ `data` contient la réponse. `output` est la sortie littérale du routeur, non structurée (le type est `text/plain`). `router` est le nom du routeur utilisé (rappelez-vous qu'un serveur "*looking glass*" peut donner accès à plusieurs routeurs).

Ici, par contre, on a eu une erreur (une erreur sur le routeur, le serveur, lui, a bien fonctionné, d'où le code de retour 200, cf. section 4 sur les conséquences de ce choix) :

```
HTTP/2.0 200 OK
Content-Type: application/json
{
     "status" : "fail",
     "data" : {
       "performed_at" : "2019-01-29T17:14:51Z",
       "runtime" : 10.37,
       "output" : [
         "Sending 5, 100-byte ICMP Echos to 2001:db8:fa3::fb56:18e",
         ".....",
         "Success rate is 0 percent (0/5)"
       ],
       "format" : "text/plain",
       "router" : "route-server.lg.op.example"
     }
}
```

Le `fail` indique que la commande exécutée sur le routeur a donné un résultat négatif, mais, autrement, tout allait bien. Si le serveur "*looking glass*" ne peut même pas faire exécuter la commande par le routeur, il faut utiliser `error` et mettre un code de retour HTTP approprié :

```
HTTP/1.1 400 Bad Request
{
     "status" : "error",
     "message" : "Unrecognized host or address."
}
```

La section 3 du RFC donne la liste des commandes possibles (un "*looking glass*" peut toujours en ajouter d'autres). La syntaxe suivie pour les présenter est celle des gabarits d'URL du RFC 6570. La première est évidemment ce brave `ping`, avec le gabarit `https://lg.op.example/.well-known/looking-glass/v1/ping/`

```
GET /.well-known/looking-glass/v1/ping/2001:db8::35?protocol=2
Host: lg.op.example

HTTP/1.1 200 OK
{
     "status" : "success",
     "data" : {
       "min" : 40,
       "avg" : 41,
       "max" : 44,
       "rate" : 100,
       "output" : [
         "Sending 5, 100-byte ICMP Echos to 2001:db8::35",
         "!!!!!",
         "Success rate is 100 percent (5/5)"
       ],
       "format" : "text/plain",
       "performed_at" : "2019-01-29T17:28:02Z",
       "runtime" : 0.77,
       "router" : "c2951.lab.lg.op.example"
     }
 }
```

Notez que le RFC ne décrit pas les champs possibles (comme `min` ou `avg`), ni les unités utilisées (probablement la milliseconde).

Autre commande traditionnelle, `traceroute` :

```
GET /.well-known/looking-glass/v1/traceroute/192.0.2.8
Host: lg.op.example

HTTP/1.1 200 OK
{
     "status": "success",
     "data": {
       "output": [
         "Tracing the route to 192.0.2.8",
         "",
         "  1 198.51.100.77 28 msec 28 msec 20 msec",
         "  2 203.0.113.130 52 msec 40 msec 40 msec",
         "  3 192.0.2.8 72 msec 76 msec 68 msec"
       ],
       "format": "text/plain",
       "performed_at": "2018-06-10T12:09:31Z",
       "runtime": 4.21,
       "router": "c7206.lab.lg.op.example"
     }
}
```

Notez une des conséquences du format non structuré de `output` : ce sera au client de l'analyser. Le RFC permet d'utiliser des formats structurés (par exemple, pour traceroute, on peut penser à celui du RFC 5388) mais ce n'est pas obligatoire car on ne peut pas forcément exiger du serveur qu'il sache traiter les formats de sortie de tous les routeurs qu'il permet d'utiliser. L'analyse automatique des résultats reste donc difficile.

D'autres commandes permettent d'explorer la table de routage. Ainsi, `show route` affiche le route vers un préfixe donné (ici, le routeur est un Cisco et affiche donc les données au format Cisco) :

```
GET /.well-known/looking-glass/v1/show/route/2001:db8::/48?protocol=2,1
Host: lg.op.example

HTTP/1.1 200 OK
{
     "status": "success",
     "data": {
       "output": [
         "S   2001:DB8::/48 [1/0]",
         "     via FE80::C007:CFF:FED9:17, FastEthernet0/0"
       ],
       "format": "text/plain",
       "performed_at": "2018-06-11T17:13:39Z",
       "runtime": 1.39,
       "router": "c2951.lab.lg.op.example"
     }
   }
```

Une autre commande, `show bgp`, affiche les informations BGP :

```
GET /.well-known/looking-glass/v1/show/bgp/192.0.2.0/24
Host: lg.op.example

HTTP/1.1 200 OK
{
     "status": "success",
     "data": {
       "output": [
         "BGP routing table entry for 192.0.2.0/24, version 2",
         "Paths: (2 available, best #2, table default)",
         "  Advertised to update-groups:",
         "     1",
         "  Refresh Epoch 1",
         "  Local",
         "    192.0.2.226 from 192.0.2.226 (192.0.2.226)",
         "      Origin IGP, metric 0, localpref 100, valid, internal",
         "[...]"
       ],
       "format": "text/plain",
       "performed_at": "2018-06-11T21:47:17Z",
       "runtime": 2.03,
       "router": "c2951.lab.lg.op.example"
     }
}
```

Les deux précédentes commandes avaient un argument, le préfixe IP sur lequel on cherche des informations (avec la syntaxe des gabarits, cela s'écrit `https://lg.op.example/.well-known/looking-glass/v1/sho`
Mais certaines commandes n'ont pas d'argument. Par exemple, `show bgp summary` affiche la liste des pairs BGP :

```
GET /.well-known/looking-glass/v1/show/bgp/summary?protocol=2&routerindex=3
Host: lg.op.example

HTTP/1.1 200 OK
{
     "status": "success",
     "data": {
       "output": [
         "BGP router identifier 192.0.2.18, local AS number 64501",
         "BGP table version is 85298, main routing table version 85298",
         "50440 network entries using 867568 bytes of memory",
         "[...]",
         "Neighbor        V       AS MsgRcvd MsgSent   TblVer  Up/Down",
         "2001:DB8:91::24 4    64500  481098  919095   85298   41w5d"
       ],
       "format": "text/plain",
       "performed_at": "2018-06-11T21:59:21Z",
       "runtime": 1.91,
       "router": "c2951.lab.lg.op.example"
     }
}
```

D'autres commandes (« organisationnelles ») sont destinées à aider le client à formuler sa question. C'est le cas de `show router list`, qui affiche la liste des routeurs auquel ce serveur *"looking glass"* donne accés :

```
GET /.well-known/looking-glass/v1/routers
```

```
...
{
     "status" : "success",
     "data" : {
       "routers" : [
         "route-server.lg.op.example",
         "customer-edge.lg.op.example",
         "provider-edge.lg.op.example"
       ],
       "performed_at" : "2018-10-19T12:07:23Z",
       "runtime" : 0.73
     }
   }
```

Autre exemple de commande organisationnelle, `cmd`, qui permet d'obtenir la liste des commandes acceptées par ce serveur (curieusement, ici, la syntaxe des gabarits du RFC 6570 n'est plus utilisée intégralement) :

```
GET /.well-known/looking-glass/v1/cmd
...
{
     "status" : "success",
     "data" : {
       "commands" : [
         {
           "href" : "https://lg.op.example/.well-known/looking-glass/v1/show/route",
           "arguments" : "{addr}",
           "description" : "Print records from IP routing table",
           "command" : "show route"
         },
         {
           "href" : "https://lg.op.example/.well-known/looking-glass/v1/traceroute",
           "arguments" : "{addr}",
           "description" : "Trace route to destination host",
           "command" : "traceroute"
         }
       ]
     }
   }
```

La liste des commandes données dans ce RFC n'est pas exhaustive, un "*looking glass*" peut toujours en ajouter (par exemple pour donner de l'information sur les routes OSPF en plus des BGP).

La section 6 du RFC décrit les problèmes de sécurité envisageables. Par exemple, un client malveillant ou maladroit peut abuser de ce service et il est donc prudent de prévoir une limitation de trafic. D'autre part, les informations distribuées par un "*looking glass*" peuvent être trop détaillées, révéler trop de détail sur le réseau et ce qu'en connaissent les routeurs, et le RFC note qu'un serveur peut donc limiter les informations données.

Et question mises en œuvre de ce RFC ? Il existe Beagle <https://github.com/lamehost/beagle> mais qui, à l'heure où ce RFC est publié, ne semble pas à jour et concerne encore une version antérieure de l'API. Peut-être RANCID va t-il fournir une API compatible pour son "*looking glass*". Periscope <http://www.caida.org/tools/utilities/looking-glass-api>, lui, est un projet différent, avec une autre API.