Supervision un peu plus avancée

Stéphane Bortzmeyer
AFNIC
1, rue Stephenson
78180 Montigny-le-Bretonneux

Résumé

La supervision automatique des services Internet est depuis longtemps un outil indispensable. Tout service sérieux est évidemment suivi par un des innombrables logiciels qui permettent de tester le bon fonctionnement de ces services. Pourtant, on voit encore des pannes n'être détectées que par un coup de téléphone d'un utilisateur mécontent ou, pire, par un tweet. Pourtant, tout le monde supervise, mais tout le monde ne supervise pas tout : il existe des trous dans la plupart des systèmes de supervision installés. Cela peut être dû à la difficulté de maintenir la liste des tests pour un système un peu complexe, ou bien au fait qu'on ne teste que les couches basses, s'assurant par exemple que le serveur HTTP accepte les connexions, mais pas qu'il donne la bonne réponse.

Un des trous les plus fréquents est celui de la supervision multi-points. L'Internet devient de plus en plus complexe, et il est fréquent qu'un test fonctionne depuis un point de mesure mais pas depuis les autres, par la faute du routage, ou de la résolution DNS ou, dans le cas d'un service anycasté, parce que certaines instances marchent et d'autres pas. Il est donc crucial de superviser depuis plusieurs points de mesure.

On présentera ici différentes techniques qui permettent d'améliorer la supervision, les exemples concrets se fondant sur le logiciel libre Icinga.

Mots-clefs

Supervision, Icinga, Nagios, RIPE Atlas

1 La supervision

On lit souvent que le but de la supervision est d'alerter l'administratrice ou l'administrateur système/réseau en cas de problème. C'est exact mais ça n'est qu'une partie de son rôle. Notamment, dans les nombreuses organisations qui ne peuvent pas se payer un NOC (Network Operations Center) actif 24x7, ou un service d'astreinte, les alarmes ne peuvent pas être l'unique moyen de gestion du réseau. Néanmoins, elles sont certainement utiles[6], bien que les cyniques diront peut-être qu'il suffit d'attendre que les gens râlent sur Twitter.

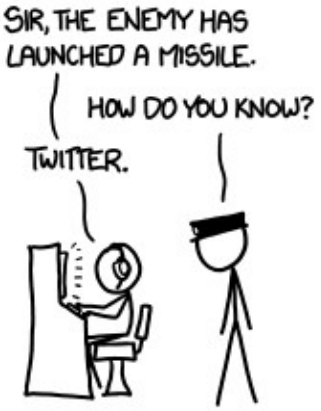

Figure 1 - XKCD (licence CC-BY-NC) https://what-if.xkcd.com/47/

En dehors des alertes, les deux raisons les plus importantes d'avoir un système de supervision sont la vérification que tout va bien et l'historique. La première permet, en un coup d'œil, de voir si tous les composants du système fonctionnent bien, par exemple après une mise à jour. Elle est l'équivalent des tests pour le développement logiciel : un moyen de savoir s'il n'y a pas eu de régression et si tout se comporte comme spécifié. Et la seconde permet, lorsqu'on arrive au bureau le lundi matin et qu'on vous dit que quelque chose était en panne dans la nuit de samedi à dimanche, de vérifier dans l'historique du système de supervision si c'était vrai, et quelle partie a effectivement eu des problèmes.

2 Le logiciel Icinga

Les exemples concrets présentés ici utiliseront le logiciel libre Icinga[7]. Icinga était à l'origine une scission de Nagios, mais la version 2 représente une réécriture complète. Notamment, les fichiers de configuration sont complètement différents.

Comme les autres logiciels de supervision, Icinga ne teste pas lui-même machines et services, il passe par des greffons (« plugins ») via l'API qui avait été initialement créée par Nagios[3]. Icinga peut donc utiliser les mêmes greffons de test que les autres logiciels de supervision, Nagios, Shinken, Centreon ou encore de nombreux autres.

Une bibliothèque de greffons de test existe, les « monitoring plugins[1] ». Elle rassemble des dizaines de programmes, ayant une interface cohérente. Ainsi, check_imap permet de tester un serveur IMAP (avec TLS, ce qu'indique l'option -S). Voici son utilisation « manuelle » (mais, normalement, il est lancé par le logiciel de supervision) :

```
% /usr/lib/nagios/plugins/check_imap -H imap.example.net \
     -p 993 -S
IMAP OK - 0.106 second response time on imap.example.net port 993
[* OK [CAPABILITY IMAP4rev1 LITERAL+ SASL-IR LOGIN-REFERRALS ID
ENABLE IDLE AUTH=PLAIN] Dovecot ready.]|
time=0.105788s;;;0.000000;10.000000
```

3 Factoriser les tests

L'administrateur ou l'administratrice système typique aura des dizaines, voire des centaines de machines à superviser, avec bien davantage de services. Il est donc nécessaire de factoriser la configuration au maximum, pour ne pas avoir à répéter les mêmes instructions. Icinga se configure via des fichiers, qu'on peut évidemment produire automatiquement depuis un programme, pour regrouper les instructions de configuration, mais cela n'est en général pas nécessaire, le langage de configuration fournit suffisamment de moyens pour la factorisation.

Ainsi, pour configurer le test de plusieurs serveurs IMAP et s'assurer que les paramètres sont les mêmes, on peut ne configurer qu'un seul **service** :

```
 apply Service "imap" {
  import "generic-service"
  check_command = "imap"
  assign where (host.address || host.address6) && host.vars.imap
  vars.imap_certificate_age = "4,2"
  vars.imap_port = 993
  vars.imap_expect = "Dovecot"
  vars.imap_ssl = true
}
```

Cela dit à Icinga qu'il doit tester IMAP avec TLS, sur le port standard 993, et s'attendre à voir « Dovecot » dans la réponse du serveur. Et le certificat doit avoir encore au moins quatre jours de validité[1]. Une fois ce service configuré, il sera appliqué (ligne « assign ») à toutes les machines qui ont une adresse IP, et la variable « imap » définie comme vraie, par exemple :

```
object Host "odin" {
  address6 = "odin.example.org"
  vars.imap = true
}
```

Un autre cas où la factorisation est utile est celui d'IPv4 et IPv6. L'incroyable lenteur avec laquelle les décideurs font migrer vers IPv6 fait que, pour encore de nombreuses années, la plupart des services devront être disponibles en IPv4 **et** IPv6. Comme il est parfaitement possible qu'un des deux protocoles fonctionne et pas l'autre, il faut donc tester les deux. La configuration par défaut d'Icinga inclut :

```
apply Service "ping4" {
  check_command = "ping4"
  assign where host.address
}
apply Service "ping6" {
```

1. L'oubli de ce test est très fréquent. Même quand le certificat est renouvellé automatiquement (cas de Let's Encrypt), il est important de la superviser ; le renouvellement automatique peut échouer.

```
  check_command = "ping6"
  assign where host.address6
}
```

Avec cela, un service ping4 est automatiquement créé pour les machines qui ont l'attribut `address` et un service ping6 pour celles qui ont l'attribut `address6`. Et les deux adresses, v4 et v6, sont donc supervisées.

Cela, c'est pour la connectivité, la couche 3. Mais plusieurs raisons[2] font qu'un service peut marcher en v4 mais pas en v6, ou bien le contraire, même si la connectivité est bonne dans les deux cas. Supervisons donc, par exemple, ssh avec les deux protocoles :

```
apply Service "ssh4" {
  check_command = "ssh"
  vars.ssh_use_ipv4 = true
  assign where host.address && host.vars.ssh_port
}
apply Service "ssh6" {
  check_command = "ssh"
  vars.ssh_use_ipv6 = true
  assign where host.address6 && host.vars.ssh_port
}
```

Contrairement au test avec ping, on utilise ici la même commande, mais avec une variable (`ssh_use_ipv{4,6}`) différente selon la famille d'adresses utilisée.

4 Combinaison de tests

Les tests donnent un résultat binaire : ce test a réussi, ou bien il a échoué. Mais, parfois, on veut prendre des décisions en tenant compte de plusieurs tests, surtout avant de déclencher une alarme qui va réveiller l'astreinte. Par exemple, on teste sa connectivité Internet en pinguant plusieurs machines externes[3], et on décide qu'il y a un problème à partir de N amers[4] injoignables. Ou bien on a un service qui peut être rendu par plusieurs machines (une zone DNS ayant plusieurs serveurs, par exemple) et, si on supervise chaque machine indépendamment, on veut aussi lever une alarme plus sérieuse si et seulement si au moins N des machines sont hors d'usage.

Il existe plusieurs solutions pour ce genre de tests, comme le greffon standard check_cluster. Je décris ici une alternative, malheureusement apparemment non maintenue, le greffon de test check_multi. Il se configure dans un fichier séparé. Ici, on teste cinq ancres[11] Atlas :

2. Un serveur qui ne se lie qu'aux adresses IPv4 de la machine (ou qu'aux adresses IPv6), des ACL configurées pour une seule famille, un pare-feu qui n'autorise ce port qu'avec une seule famille d'adresses, etc.

3. N'utilisez pas pour cela des machines prises un peu au hasard. Non seulement vous n'avez pas l'autorisation de les utiliser ainsi, mais elles peuvent arrêter de répondre aux requêtes ICMP Echo sans prévenir. Le mieux est sans doute d'utiliser les ancres Atlas[11], des machines prévues pour cela.

4. Un amer (terme venu de la marine) est la machine qu'on pingue pour voir l'état de la connectivité. L'ARCEP parle de « mire » pour ces machines de test.

```
command[ AFNIC ]                  =
/usr/lib/nagios/plugins/check_ping $MULTI_PARAMETERS$ -H fr-par-
as2486.anchors.atlas.ripe.net
command[ France-IX ]                 =
/usr/lib/nagios/plugins/check_ping $MULTI_PARAMETERS$  -H fr-par-
as57734.anchors.atlas.ripe.net
command[ SDV ]                  =
/usr/lib/nagios/plugins/check_ping $MULTI_PARAMETERS$  -H fr-sxb-
as8839.anchors.atlas.ripe.net
command[ Rezopole ]                  =
/usr/lib/nagios/plugins/check_ping $MULTI_PARAMETERS$  -H fr-lio-
as199422.anchors.atlas.ripe.net
command[ Univ-Erlangen-Nurnberg ] =
/usr/lib/nagios/plugins/check_ping $MULTI_PARAMETERS$  -H  de-
erl-as680.anchors.atlas.ripe.net

state   [ CRITICAL   ] = COUNT(CRITICAL) > 2
state   [ WARNING    ] = COUNT(WARNING) > 2 || COUNT(CRITICAL) >
1
state   [ UNKNOWN    ] = COUNT(UNKNOWN) > 1
```

Et on décide que l'état du test général est critique si au moins trois ancres sont dans un état critique. Ainsi, la panne ou l'injoignabilité d'une ou deux ancres ne va pas entrainer d'alerte inutile.

5 Écrire son greffon de test

Le logiciel de supervision ne fait pas en général lui-même les tests, il y aurait trop de possibilités. Au contraire, il se charge uniquement de l'ordonnancement des tests, et délègue le test à un greffon (« plugin »), un petit programme qui va recevoir comme paramètres les détails du système à tester, et indiquer le résultat du test. L'API la plus courante entre logiciel de supervision et greffon est celle qui a été popularisée par Nagios et se nomme donc souvent « API Nagios »[3]. Elle est d'une grande simplicité : le greffon de test reçoit les paramètres sur la ligne de commande, il écrit le résultat sur la sortie standard[5] et, surtout, il doit terminer son exécution avec un code de retour qui indique si tout s'est bien passé (code 0), s'il y a un avertissement (code 1) ou une erreur (code 2). On voit que c'est réalisable dans tous les langages de programmation.

La liste des greffons (« plugins ») de test sur le site des « monitoring plugins »[1] est impressionnante, et on se dit qu'elle couvre vraiment tous les cas. Mais ce n'est pas forcément vrai, par exemple pour les nouveaux protocoles comme DoT (DNS sur TLS) et DoH (DNS sur HTTPS). Sans parler des protocoles applicatifs spécifiques à votre organisation. Il est donc souvent nécessaire d'écrire ses propres greffons de test. C'est très facile et je vous encourage à le faire.

5. Et jamais sur l'erreur standard, ce qui peut dérouter le programmeur Unix.

Notez que, de nos jours où tant de protocoles applicatifs s'appuient sur HTTPS, le greffon générique check_http convient souvent. Ainsi, pour superviser un service RDAP[6], qui renvoie du JSON sur HTTPS, on peut se contenter de :

```
vars.http_vhosts["https"] = {
  http_uri = "/domain/quimper.bzh"
  http_vhost = "rdap.nic.bzh"
  http_string = "\"eventDate\" : \"2014-07-11T08:52:15Z"
}

vars.http_vhosts["https-notexist"] = {
  http_uri = "/domain/iknowitdoesnotexist.bzh"
  http_vhost = "rdap.nic.bzh"
  http_expect = 404
}
```

On fait une requête HTTP normale, l'une testant un nom de domaine existant et vérifiant qu'on obtient sa date de création, l'autre un nom qui n'existe pas vérifiant qu'on obtient bien le 404 attendu.

En outre, souvent, le greffon qu'on écrit est un simple emballage d'un greffon existant. Ainsi, pour superviser des serveurs .onion[7], accessibles uniquement via Tor, que le greffon standard check_http ne connaît pas, on peut « emballer » l'appel à check_http dans un script shell qui appelle torsocks, programme qui redirige les appels réseau via socks avant de les envoyer à Tor.

Voyons d'abord comment le faire en shell[8], pour superviser un service whois[9] (les variables W_HOST, D_LOOK et S_MATCH sont remplies à partir de la ligne de commande) :

```
R_OUT=$(whois -h ${W_HOST} ${D_LOOK} 2>/dev/null)

if [ $? -ne 0 ];
then
    echo "WHOIS server '${W_HOST}' did not respond"
    exit 2
fi
```

On appelle la commande whois du shell (on aurait pu utiliser netcat, le protocole whois étant simple). Si elle échoue, on renvoie tout de suite le code de retour 2, erreur critique.

```
L_CNT=$(echo ${R_OUT} | grep "${S_MATCH}" 2>/dev/null | wc -l)
if [ ${L_CNT} -eq 0 ];
```

6. Protocole concurrent de whois, permettant d'obtenir des informations sur un objet enregistré, par exemple un nom de domaine, ou un préfixe IP, dans un registre.

7. Comme le note Rayna Stamboliyska, l'auteure de « La face cachée d'Internet », « heureusement que le darknet n'a pas de SLA, vu le niveau moyen de disponibilité des .onion ».

8. Le script est disponible sur mon site.

9. Au moment de l'écriture de cet article, le protocole RDAP essaie de remplacer whois, mais ce n'est pas encore fait.

```
then
    echo \
   "WHOIS server '${W_HOST}' did not return a match on ${D_LOOK}"
    exit 1
else
    echo "OK - Found ${L_CNT} match"
    exit 0
fi
```

On teste ensuite si la chaîne de caractères S_MATCH est bien dans le résultat[10]. Son absence produit un avertissement. Autrement, tout va bien et on renvoie zéro.

Ces greffons de test ne font rien qui soit spécifique d'un langage de programmation particulier, donc on peut les écrire dans le langage de son choix. Ainsi, pour superviser les serveurs DoH (DNS sur HTTPS), j'utilise un programme en Python[11], reposant sur la bibliothèque curl :

```
c.setopt(c.URL, url)
…
c.perform()
rcode = c.getinfo(pycurl.RESPONSE_CODE)
if rcode == 200:
    body = buffer.getvalue()
    try:
       response = dns.message.from_wire(body)
    except dns.name.BadLabelType as e:
       print("%s ERROR - %s" % (url, "Not a DNS reply, is it a
DoH server? \"%s\"" % e))
       sys.exit(STATE_CRITICAL)
    print("%s OK - %s" % (url, "No error for %s/%s, %i bytes
received" % (lookup, ltype, sys.getsizeof(body))))
   sys.exit(STATE_OK)
else :
    body =  buffer.getvalue()
    print("%s HTTP error - %i: %s" % (url, rcode, body.decode()))
    sys.exit(STATE_CRITICAL)
```

On se connecte au serveur DoH (c.perform) à qui on envoie la requête DNS et on regarde son code de retour HTTP pour définir le code de retour de sortie du greffon.

Autre possibilité, en Go, ce qui est le cas d'un greffon de test d'une zone DNS (il vérifie que tous les serveurs répondent et lève une alarme si un nombre de serveurs supérieur à une valeur prédéfinie est en panne[12]) :

```
if brokenServers < *criticalThreshold {
```

10. Un des oublis les plus fréquents, lorsqu'on met en place la supervision, est de ne tester que le fait que le service répond, pas qu'il répond **correctement**. Ainsi, en testant un site Web, il ne faut pas seulement regarder le code de retour HTTP, il vaut aussi vérifier le corps de la réponse (certains CMS Web répondent « 200 OK » alors que la réponse est « Cannot connect to the database »).

11. Disponible sur mon site.

12. Ce greffon est disponible sur Github.

```
          nagios.ExitStatus(nagios.WARNING, errorMessages[0],
                    errorMessages[1:errors], false)
    } else {
          nagios.ExitStatus(nagios.CRITICAL, errorMessages[0],
                    errorMessages[1:errors], false)
}
```

Et enfin bien sûr, ces greffons peuvent être écrits en C, notamment si on a besoin de bibliothèques C. C'est le cas par exemple du greffon de supervision[13] de DoT (DNS sur TLS), dont une version améliorée est désormais dans la bibliothèque getdns.

6 Plusieurs sondes coordonnées

Parfois, un test ne peut pas être fait à distance, et nécessite un processus tournant sur la machine supervisée. D'autre part, le test d'un service réseau a souvent intérêt à être fait depuis plusieurs machines, tout en restant piloté depuis un point unique. Nagios, et la première version d'Icinga, faisaient cela via le protocole NRPE. Peu sécurisé, et n'ayant pas évolué, NRPE a été abandonné par Icinga au profit d'un système de maître et de satellites. Dans ce système, une machine, le maître, coordonne les tests, qui sont effectués par des satellites. On peut ainsi tester ce qui n'est pas accessible à distance[14], et surtout avoir un système réparti de test. Cela nécessite l'installation d'Icinga sur chaque satellite mais cela vaut la peine.

Le moyen le plus simple de configurer maître et satellites est via la commande `icinga2 node wizard`. Des certificats auto-signés seront alors générés, puisque la communication entre maître et satellites est sécurisée par TLS. Mais on peut aussi le faire à la main, notamment si on veut utiliser des certificats faits par une PKI existante[15]. Comme c'est toujours le maître qui se connecte aux satellites, il faut un certificat pour chaque satellite.

Les satellites sont regroupés en **zones**, chaque zone comportant plusieurs nœuds (« endpoints »). Ici, on configure, sur le maître, une zone avec un seul nœud :

```
object Endpoint "loki.example.net" {
  host = "loki.example.net"
}
object Zone "loki.example.net" {
  endpoints = [ "loki.example.net" ]
  parent = ZoneName // La zone du maître
}
```

On configure ensuite, sur le maître, les tests qui doivent être faits par les satellites, et on a alors sur le maître le résultat des tests distants[16]. Si les satellites sont extérieurs à votre réseau, ce qui est très recommandé, on peut ainsi avoir une vision externe de ses

13. Disponible sur GitHub.

14. Par exemple un serveur local, qui n'écoute que sur localhost, pour des raisons de sécurité.

15. J'utilise l'AC CAcert. Let's Encrypt n'est pas forcément utilisable, les machines pouvant ne pas être accessibles de l'extérieur.

16. Le système de tests répartis d'Icinga est bien plus riche que cela, on n'en a donné qu'un bref aperçu.

services, évitant de répondre « mais non, tout marche » à un utilisateur qui signale un problème qui ne se produit pas depuis le réseau interne.

7 Les sondes RIPE Atlas

On a dit que beaucoup de problèmes dans le réseau dépendaient du point de mesure. Certaines pannes sont binaires (si un service tourne sur une seule machine physique, et que cette machine s'arrête, il n'y a plus rien) mais d'autres sont plus complexes à analyser, dépendant du réseau où on se trouve[17]. Un problème de routage ne va frapper que certains AS (systèmes autonomes). Un « trou noir » annoncé à tort ne va capter qu'une partie du trafic. L'anycast et/ou un répartiteur de charge va vous diriger vers un serveur différent (et potentiellement à problèmes), selon votre point de départ. D'où l'importance de regarder depuis plusieurs points de l'Internet. Une façon de faire cela est de demander à des copains et copines qui sont connectés via d'autres réseaux de tester. Mais ce n'est pas très pratique, et, souvent, les informations obtenues sont vagues et incomplètes. On peut aussi louer des serveurs en plusieurs endroits (comme dans le cas des satellites cités plus tôt), mais cela coûte de l'argent et du temps, pour l'installation et la maintenance.

Une meilleure solution est donc parfois d'établir un réseau de sondes largement réparties, et qui seront utilisables par tous les participants au projet. C'est le cas par exemple du RING[4]. Mais je vais me focaliser sur un autre réseau, celui des sondes RIPE Atlas[5]. Ce projet est géré par le RIPE-NCC. Les sondes Atlas sont de petits boitiers, avec de jolies couleurs, que des volontaires installent en plein d'endroits de l'Internet. Elles reçoivent ensuite des ordres de leur contrôleur, leur demandant d'effectuer des mesures actives, en utilisant notamment les protocoles ICMP Echo, NTP et DNS. L'intérêt des sondes Atlas est que tout le monde peut y avoir accès, et demander des mesures de ce qui l'intéresse.

Les sondes Atlas peuvent être commandées par l'interface Web ou par l'API d'Atlas. Pour utiliser cette API, il existe un client officiel[9]. Voici un exemple de son utilisation :

```
% ripe-atlas measure ping --target 194.57.3.23
…
  https://atlas.ripe.net/measurements/23019928/
...
48 bytes from probe #51988 89.248.190.34   to 194.57.3.23
(194.57.3.23): ttl=242 times:49.669,  48.475,  48.558,
48 bytes from probe #33888 81.250.155.13   to 194.57.3.23
(194.57.3.23): ttl=243 times:15.826,  16.105,  15.844,
48 bytes from probe #32297 185.125.112.206 to 194.57.3.23
(194.57.3.23): ttl=240 times:77.303,  76.567,  76.673,
…
```

On voit dans cet exemple plusieurs sondes répondre qu'elles ont pingué avec succès cet objectif, et afficher la latence mesurée. Le client officiel fait de gros efforts pour que les

17. Et, en général, pas de la localisation géographique. C'est pour cela qu'il est en général absurde, lors d'une discussion sur une panne, d'indiquer la ville où on se trouve. Le numéro d'AS de l'opérateur serait une information plus utile.

résultats affichés soient très proches de ce qu'affichent les commandes traditionnelles de déboguage, comme ping ou dig.

Mais je préfère me servir en général du programme Blaeu[8], plus pratique pour avoir des résultats agrégés. Voici un exemple :

```
% blaeu-reach 194.57.3.23
5 probes reported
Test #23019858 done at 2019-10-03T14:11:38Z
Tests: 15 successful tests (100.0 %), 0 errors (0.0 %), 0
timeouts (0.0 %), average RTT: 30 ms
```

Ici, on demande de tester la connectivité de 194.57.3.23 (avec ICMP Echo). Par défaut, cinq sondes sont utilisées, chacune faisant trois tests, et tous les tests ont été réussis, avec une latence aller-retour moyenne de trente milli-secondes. Un des intérêts des sondes Atlas est qu'on peut sélectionner les sondes selon de nombreux autres critères, comme le numéro d'AS ou le pays[18]. Ici, je demande des sondes au Chili, et je teste un objectif nettement moins bien connecté que le précédent :

```
% blaeu-reach --requested 100 --country CL 194.133.122.42
29 probes reported
Test #23020017 done at 2019-10-03T14:37:44Z
Tests: 86 successful tests (98.9 %), 0 errors (0.0 %), 1 timeouts
(1.1 %), average RTT: 417 ms
```

Si la latence est bien plus élevée, on note que, là encore, aucun paquet n'a été perdu. Avec un réseau mal configuré (problème hélas fréquent en IPv6, où les opérateurs réseau sont bien plus négligents), on voit des pertes :

```
% blaeu-reach --requested 100 --by_probe 2001:4268:200::1
99 probes reported
Test #23022835 done at 2019-10-04T06:37:43Z
Tests: 82 successful probes (82.8 %), 17 failed (17.2 %), average
RTT: 184 ms
```

Les sondes RIPE Atlas savent également faire des tests DNS :

```
% blaeu-resolve --type A www.jres.org
[194.57.3.23] : 5 occurrences
Test #23020036 done at 2019-10-03T14:42:36Z
```

C'est utile dans le cas où la résolution dépend du point de mesure. Ainsi, en avril 2016, un problème de routage rendait les serveurs de noms du domaine impots.gouv.fr injoignables depuis certains opérateurs, comme Orange. Pendant que les réseaux sociaux et les forums entamaient un dialogue de sourd (« c'est en panne », « non, ça marche »), les mesures montraient bien que les résolveurs DNS de certaines sondes

18. Ce dernier point est utile pour tester les cas de censure, celle-ci étant en général nationale.

RIPE Atlas réussissaient à résoudre ce nom, et d'autres pas (SERVFAIL = « Server Failure ») :

```
% blaeu-resolve --requested 500 --country FR --type A \
      impots.gouv.fr
[ERROR: SERVFAIL] : 177 occurrences
[145.242.11.48] : 216 occurrences
Test #3645793 done at 2016-04-05T10:01:16Z
```

Les sondes Atlas peuvent également effectuer des traceroutes, avec les protocoles ICMP, UDP ou TCP, ce qui évite de demander à des ami·e·s « tu pourrais faire un traceroute depuis chez toi ? ». Voici un exemple depuis le Pérou :

```
% blaeu-traceroute --format --country PE --requested 3 \
     194.57.3.23
Test #23019978 done at 2019-10-03T14:32:27Z
From:  179.6.203.204    12252    America Movil Peru S.A.C., PE
Source address:  192.168.0.12
Probe ID:  18371
1    192.168.0.1    NA    NA    [0.651, 0.527, 0.479]
2    10.172.59.1    NA    NA    [7.189, 10.203, 27.49]
3    10.150.144.209    NA    NA    [7.972, 9.801, 8.552]
4    10.95.156.34    NA    NA    [11.033, 9.592, 10.236]
5    ['*', '*', '*']
6    ['*', '*', '*']
7    10.95.156.46    NA    NA    [13.852, 8.614, 8.877]
8    173.205.43.241    3257    GTT-BACKBONE GTT, DE    [87.969,
96.463, 87.457]
9    89.149.185.133    3257    GTT-BACKBONE GTT, DE    [190.802,
190.735, 189.739]
10    77.67.123.206    3257    GTT-BACKBONE GTT, DE    [192.98,
191.3, 187.991]
11    193.51.177.27    2200    FR-RENATER Reseau National de
telecommunications pour la Technologie, FR    [189.311, 189.804,
190.008]
12    193.51.181.117    2200    FR-RENATER Reseau National de
telecommunications pour la Technologie, FR    [189.713, 189.751,
190.216]
13    ['*', '*', '!', '!', '!', '!']
14    194.57.3.23    2200    FR-RENATER Reseau National de
telecommunications pour la Technologie, FR    ['!', '!', 127.953]
15    194.57.3.23    2200    FR-RENATER Reseau National de
telecommunications pour la Technologie, FR    [22.666, 16.854,
130.512]
…
```

Notez qu'on peut combiner les tests Atlas et Icinga[10]. Ainsi, une fois qu'une mesure Atlas périodique est lancée pour une machine (regardez la mesure #2060427), on configure Icinga pour voir ces résultats :

```
vars.http_vhosts["atlas"] = {
    http_address = "atlas.ripe.net"
    http_uri = "/api/v2/measurements/2060427/status-check ?
permitted_total_alerts=3"
    http_string = "global_alert\":false"
    }
```

Si les sondes Atlas de la mesure #2060427 ne peuvent joindre l'objectif, on n'aura plus « global_alert:false » dans le résultat, et Icinga lèvera une alarme.

8 Et l'avenir ?

La supervision, comme l'Internet en général, évolue sans cesse. Par exemple, la classique séparation entre « machines » et « services », qu'on trouve dans beaucoup de logiciels de supervision, où le service dépend de la machine sur laquelle il tourne, perd de son sens avec des systèmes comme les orchestrateurs, qui créent et détruisent machines virtuelles et containers à la demande, pour s'adapter à l'activité. Dans un tel environnement, l'arrêt d'une machine n'est plus forcément un problème. De même, les organisations qui utilisent des systèmes comme les « singes du chaos », par exemple Netflix, ne souhaitent pas forcément lever une alarme chaque fois qu'un primate a arrêté un serveur.

Mais quelles que soient les évolutions, la supervision sera de plus en plus nécessaire, vu le nombre de services critiques qui sont accessibles par l'Internet. Il n'est plus acceptable aujourd'hui d'avoir un système qui ne soit pas complètement supervisé.

Bibliographie

[1] Site officiel des « monitoring plugins ».

[2] Documentation officielle d'Icinga.

[3] La documentation officielle de l'API Nagios et la documentation pour écrire un greffon de test.

[4] Le RING, un réseau de machines fournissant des comptes « shell » aux membres, pour tester le réseau.

[5] RIPE Atlas, le plus grand réseau de mesure du monde.

[6] Alertes envoyées vers le fédivers, ou « on peut vraiment notifier avec n'importe quoi ».

[7] Site officiel Icinga.

[8] Site officiel de Blaeu et un article décrivant comment l'utiliser pour déboguer des problèmes réseau.

[9] Client officiel de l'API des sondes RIPE Atlas.

[10] Les « status checks » d'Atlas.

[11] Les ancres Atlas, des machines plus grosses que les sondes traditionnelles, et qui peuvent notamment servir d'amers pour les mesures.