RFC 7252 : Constrained Application Protocol (CoAP)

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 2 juillet 2014

Date de publication du RFC : Juin 2014

https://www.bortzmeyer.org/7252.html

Le protocole CoAP n'est pas destiné aux ordinateurs, ni même aux "*smartphones*" mais principalement aux petits engins, aux machines qui n'ont parfois qu'un microcontrôleur 8 bits pour tout processeur, très peu de mémoire (cf. RFC 6574 [1]) et qui, en prime, sont connectées par des liens radio lents et peu fiables (les « LowPAN » des RFC 4919 et RFC 4944), allant parfois à seulement quelques dizaines de kb/s. Pour de tels engins, les protocoles comme HTTP et TCP sont trop contraignants. CoAP est un « HTTP-like » (il reprend pas mal de termes et de concepts de HTTP, comme le modèle REST) mais spécialement conçu pour des applications M2M ("*machine-to-machine*") dans des environnements à fortes limitations matérielles.

Le succès de HTTP et notamment de REST a sérieusement influencé les concepteurs de CoAP. Le modèle de CoAP est celui de HTTP : protocole requête/réponse, verbes `GET`, `POST`, `DELETE`, etc, des URI et des types de media comme sur le Web (`application/json`...) Cette proximité facilitera notamment le développement de passerelles entre le monde des objets CoAP et le Web actuel (section 10). Une des pistes explorées pour l'Internet des Objets (terme essentiellement marketing mais répandu) avait été de prendre de l'HTTP normal et de comprimer pour gagner des ressources réseau mais CoAP suit une autre voie : définir un protocole qui est essentiellement un sous-ensemble de HTTP. Si vous connaissez HTTP, vous ne serez pas trop perdu avec CoAP.

CoAP tournera sur UDP (RFC 768), TCP étant trop consommateur de ressources, et, si on veut de la sécurité, on ajoutera DTLS (RFC 6347). CoAP fonctionne donc de manière asynchrone (on lance la requête et on espère une réponse un jour). On pourrait décrire CoAP comme composé de deux parties, un transport utilisant UDP avec des extensions optionnelles si on veut de la fiabilité (accusés de réception), et un langage des messages (requêtes et réponses en "*HTTP-like*").

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc6574.txt`

Et les messages ? L'en-tête CoAP (section 3 du RFC) a été conçu pour être facile à analyser par des programmes tournant sur de petites machines. Au lieu du texte lisible de HTTP, l'en-tête de CoAP commence par une partie fixe de quatre octets, qui comprend un "*Message ID*" de deux octets. Ce "*Message ID*" permet de détecter les duplicatas et d'associer un accusé de réception à un message précis. À noter qu'avec ces deux octets, on est limité à environ 250 messages par seconde (en raison du paramètre `EXCHANGE_LIFETIME` qui est à 247 secondes par défaut). Ce n'est pas une limite bien grave : les ressources (en énergie, en capacité réseau, etc) des machines CoAP ne leur permettront pas d'être trop bavardes, de toute façon.

Le message contient également un champ "*Token*" (utilisé pour mettre en correspondance les requêtes et les réponses, à un niveau supérieur à celui des messages, qui sont, eux, identifiés par le "*Message ID*") et un code sur huit bits. Ce code s'écrit sous forme d'un groupe de trois chiffres, le premier indiquant la classe (2 : succès, 4 : erreur du client, 5 : erreur du serveur, etc) et suivi d'un point, et les deux autres chiffres fournissant des détails. Comme vous connaissez HTTP, vous ne serez pas étonné d'apprendre que le code 4.03 indique un problème de permission... Un registre de ces codes `<https://www.iana.org/assignments/core-parameters/core-parameters.xhtml#response-codes>` existe.

Notez bien que l'encodage, lui, est radicalement différent de celui de HTTP. Il est en binaire et le code 4.03 sera transmis sous forme d'un octet (10000011, avec 100 pour la classe et 11 pour le détail), pas sous forme de la chaîne de caractères qu'utilise HTTP.

Le message contient ensuite des options, qui jouent le rôle des en-têtes de requête de HTTP (leur liste est également dans un registre `<https://www.iana.org/assignments/core-parameters/core-parameters.xhtml#option-numbers>`, on y trouve des trucs connus comme `If-Match` - section 5.10.8.1, `Accept` - section 5.10.4 - ou `ETag` - section 5.10.6). Pour économiser quelques bits, leur encodage est assez spécial. Le numéro de l'option, normalement sur 16 bits, n'est pas transmis tel quel mais sous forme d'une différence avec l'option précédente. Ainsi, si on utilise les options 4 (`ETag`) et 5 (`If-None-Match`), la première option indiquera bien 4 mais la seconde indiquera 1... De même, les valeurs de l'option sont violemment comprimées. Si on veut envoyer un entier, on ne doit transmettre que le nombre d'octets nécessaire (rien si l'entier vaut zéro, un seul octet s'il est inférieur à 255, etc).

Le transport utilisé par CoAP, UDP, ne fournit aucune garantie de remise : un paquet UDP peut être perdu, et UDP ne s'en occupera pas, ce sera à l'application de gérer ce cas. CoAP fournit un service pour faciliter la vie aux applications : on peut marquer un message CoAP comme nécessitant un accusé de réception (type `CON` pour "*Confirmable*"). Si ce type est présent dans l'en-tête CoAP, l'expéditeur recevra un accusé de réception (type `ACK`) portant le "*Message ID*" du message reçu. Le RFC fournit l'exemple d'un capteur de température interrogé en CoAP :

```
Client -> Serveur: CON [0xbc90] "GET /temperature"
Serveur -> Client: ACK [0xbc90] "2.05 Content" "22.5 °C"
```

Ici, le second message, la réponse du serveur, contient un accusé de réception pour 0xbc90 (le "*Message ID*"). La requête est un `GET` et la réponse a le code 2.05 (un succès). Notez que, contrairement à HTTP, les codes sont écrits avec un point entre le premier chiffre (la classe) et les deux autres (le détail, cf. section 5.9). Ainsi, 4.04 a la classe 4 (erreur du client) et le détail 04 (ressource non trouvée). Comme indiqué plus haut, les classes sont les mêmes qu'en HTTP : 2 pour un succès, 4 pour une erreur du client (qui devrait changer sa requête avant de réessayer) et 5 pour une erreur du serveur (réessayer à l'identique plus tard peut marcher).

Le "*Message ID*" et ces types `CON` et `NON` (son contraire, "*Non-confirmable*") permettent de mettre en œuvre des mécanismes simples de retransmission. L'émetteur envoie un message de type `CON`, s'il

ne reçoit pas de message de type ACK à temps, il réessaie, avec croissance exponentielle du délai. Le "*Message-ID*" permet aussi de détecter les éventuels duplicatas.

On n'est pas obligé d'utiliser le type CON. Si un message n'a pas besoin de fiabilité (par exemple les lectures périodiques qu'envoie un capteur, que quelqu'un écoute ou pas), il sera du type NON.

Le "*Message ID*" doit évidemment être unique, pendant la durée de vie possible des échanges (variable EXCHANGE_LIFETIME, quatre minutes par défaut). Il peut être généré de manière incrémentale (l'émetteur garde un compteur indiquant le dernier "*Message ID*" généré) mais, dans ce cas, le RFC recommande (section 4.4) que la valeur initiale soit aléatoire, pour limiter les possibilités d'attaque par un méchant non situé sur le chemin des données (c'est un problème de sécurité analogue à celui des identificateurs de fragments IP - voir l'option `--frag-id-policy` de l'outil frag6 <https://www.bortzmeyer.org/hacking-ipv6.html> - ou celui des numéros de séquences initiaux de TCP - voir le RFC 6528).

Autre problème pour l'émetteur CoAP : quelle taille de messages ? Les machines pour qui CoAP est conçu peuvent avoir du mal à gérer parfaitement la fragmentation. Et puis des gros paquets ont davantage de chances d'être corrompus que des petits, sur les liens radios typiques des machines CoAP. Le RFC recommande donc de ne pas dépasser la MTU et, si elle est inconnue, de ne pas dépasser 1 280 octets <https://www.bortzmeyer.org/fragmentation-ip-1280.html> en IPv6. (C'est plus compliqué pour IPv4 où le RFC recommande 576 octets et le bit DF - "*Don't Fragment*" - en notant que, pour les usages typiques de CoAP, le chemin sera simple, avec peu de routeurs et pas de liens à grande distance, donc peu de chances que la fragmentation soit nécessaire.)

Comme CoAP repose sur UDP, il n'y a pas de mécanisme de contrôle de la congestion par défaut. CoAP doit donc s'auto-limiter : réémissions avec croissance exponentielle des délais, et au maximum une requête en attente pour chaque machine avec qui on communique (paramètre NSTART). Ces règles sont pour le client. Si tous les clients les respectent, tout ira bien. Mais un serveur CoAP prudent doit considérer que certains clients seront bogués ou malveillants. Les serveurs ont donc tout intérêt à avoir une forme ou une autre de limitation de trafic.

La sémantique des requêtes et réponses CoAP figure en section 5. On l'a dit, elle ressemble à celle de HTTP, un client qui envoie une requête et un serveur qui transmet une réponse. L'une des plus grosses différences avec HTTP est le caractère asynchrone de l'échange.

Les requêtes sont les classiques (section 5.8) GET, POST, PUT et DELETE, agissant sur une ressource donnée (identifiée par un URI), comportant des données optionnelles (surtout utilisées pour POST et PUT) et utilisant les types de média. CoAP exige que GET, PUT et DELETE soient idempotents (une bonne pratique mais pas toujours respectée sur le Web). Une requête non reconnue et la réponse sera un 4.05. La liste des requêtes possibles est dans un registre IANA <https://www.iana.org/assignments/core-parameters/core-parameters.xhtml#method-codes>.

Du fait du caractère asynchrone de l'échange, mettre en correspondance une requête et une réponse est un peu plus compliqué qu'en HTTP, où toutes les deux empruntaient la même connexion TCP. Ici, la section 5.3.2 résume les règles de correspondance : une réponse correspond à une requête si l'adresse IP source de la réponse était l'adresse IP de destination de la requête **et** que le "*Message ID*" et le "*Token*" correspondent.

Comme HTTP, CoAP permet d'utiliser des relais. C'est d'autant plus important que les machines CoAP sont souvent indisponibles (hibernation pour économiser la batterie, par exemple) et un relais pourra donc répondre à la place d'une machine injoignable. Les relais sont traités plus en détail en

section 5.7. Comme en HTTP, on pourra avoir des "*forward proxies*" sélectionnés par les clients ou des "*reverse proxies*" placés devant un serveur.

On l'a vu, CoAP, comme HTTP, utilise des URI, détaillés en section 6. Ils commencent par les plans `coap:` ou `coaps:`. On verra ainsi, par exemple, `coap://example.com/ sensors/temp.xml` ou `coaps://[2001:db8::2:1]/measure/temperature/`. Attention, tout le monde croit bien connaître les URI car ils sont massivement utilisés pour le Web mais le RFC met en garde les programmeurs : il y a plein de pièges dans le traitement des URI, notamment dans le décodage des valeurs encodées en pour-cent.

Si le port n'est pas indiqué dans l'URI (ce qui est le cas des deux exemples ci-dessus), le port par défaut est 5683 pour `coap:` et 5684 pour `coaps:`.

CoAP utilise, comme HTTP, la convention du `/.well-known/` (RFC 8615) pour placer des ressources « bien connues », facilitant ainsi leur découverte.

Justement, à propos de découverte, la section 7 détaille comment un client CoAP va pouvoir trouver les URI des ressources intéressantes. CoAP est conçu pour des communications de machine à machine, où aucun humain n'est présent dans la boucle pour deviner, faire une recherche Google, utiliser son intuition, etc. La méthode privilégiée dans CoAP est celle du RFC 6690 : un fichier au format normalisé situé en `/.well-known/core`.

Les section 9 et 11 sont consacrées à la sécurité de CoAP. Évidemment, vu les ressources très limitées des machines CoAP typiques, il ne faut pas s'attendre à une grande sécurité. Par exemple, une protection par DTLS (RFC 6347) est possible mais pas obligatoire, juste conseillée. En outre, les ressources matérielles limitées des machines CoAP compliquent la tâche du programmeur : des processeurs très lents qui facilitent les attaques par mesure du temps écoulé, et pas de bon générateur aléatoire. En l'absence d'un tel générateur, on ne peut donc pas générer des clés cryptographiques localement, il faut le faire en usine et les enregistrer sur la machine. Et pas question de mettre les mêmes clés à tout le monde : comme les machines CoAP seront souvent dispersées dans des endroits ayant peu de sécurité physique (capteurs placés un peu partout), et n'auront en général pas de protection contre les manipulations physiques, pour des raisons de coût, un attaquant pourra facilement ouvrir une machine et en extraire ses clés.

La complexité est l'une des ennemies principales de la sécurité. Le RFC rappelle que le traitement des URI (bien plus compliqué qu'il n'en a l'air) impose des analyseurs relativements complexes et donc augmente la probabilité de bogues, pouvant mener à des failles de sécurité. Même chose pour le format des liens du RFC 6690.

Question réseau, CoAP souffrira sans doute d'attaques rendues possibles par l'usurpation d'adresses IP. Comme UDP ne nécessite pas d'échange préalable, il est trivial d'envoyer un paquet depuis une adresse usurpée et de le voir accepté. Cela permet des tas d'attaques comme :

— Envoyer un `RST` ("*Reset*") aux demandes de connexion vers un serveur, rendant ainsi ce serveur inutilisable (déni de service),
— Envoyer une fausse réponse à une requête `GET`.

Pour cette deuxième attaque, il faut pouvoir deviner le "*Token*" utilisé. Si l'attaquant est aveugle (parce que situé en dehors du chemin des paquets), sa tâche sera difficile si le "*Token*" est bien aléatoire (mais ces machines contraintes en ressources n'ont souvent pas de vrai générateur aléatoire, au sens du RFC 4086).

En pratique, il est probable que les déploiements réels de CoAP aient lieu dans des réseaux fermés, ne contenant que des machines de confiance, et où l'accès à l'extérieur passera par une passerelle qui s'occupera de sécurité. Autrement, un simple ping à destination d'une machine CoAP pourrait suffir à vider sa batterie.

CoAP introduit aussi une autre menace, celle d'attaques par **amplification** (section 11.3). Comme SNMP et le DNS (tous les deux déjà utilisés pour ces attaques <https://www.bortzmeyer.org/attaques-reflexion.html>), CoAP a des réponses plus grandes que les questions (permettant l'amplification) et repose sur UDP et non pas TCP (permettant l'usurpation d'adresse IP).

La seule limite à ces attaques semble devoir être les faibles ressources des machines CoAP, qui devraient les rendre relativement peu intéressantes comme amplificateurs contre des serveurs Internet. En revanche, à l'intérieur d'un réseau de machines CoAP, le risque demeure élevé et le RFC ne propose aucune vraie solution.

CoAP est, d'une certaine façon, un vieux protocole (voir l'article « "*CoAP : An Application Protocol for Billions of Tiny Internet Nodes*" » de Carsten Bormann, Angelo Paolo Castellani et Zach Shelby publié en 2012 dans dans "*Internet Computing, IEEE (Volume :16 , Issue : 2)*" et malheureusement pas disponible en ligne, il faut demander une copie à l'auteur). Il existe déjà plusieurs mise en œuvre, qui ont été souvent testées dans des essais d'interopérabilité formels. Un exemple de mise en œuvre est celle du système Contiki.

Des exemples de requêtes CoAP décrites en détail figurent dans l'annexe A du RFC. Grâce à Renzo Navas, voici quelques traces CoAP au format pcap :

- Un simple GET (en ligne sur `https://www.bortzmeyer.org/files/01-coap-getwellknown.pcap`) du fichier des services, `/.well-known/core`,
- Un autre GET (en ligne sur `https://www.bortzmeyer.org/files/02-getHelloWorldANDtoUppercaseRE`
`pcap`),
- Un GET avec réponse (en ligne sur `https://www.bortzmeyer.org/files/03-separateResponse.pcap`),
- POST, PUT et DELETE (en ligne sur `https://www.bortzmeyer.org/files/04-post-put-get-delete.pcap`).

Si tcpdump ne connait pas encore CoAP, Wireshark, par contre, sait bien l'analyser. Ici, une la première requête `GET` du premier fichier pcap et sa réponse :

```
Constrained Application Protocol, TID: 27328, Length: 21
    01.. .... = Version: 1
    ..00 .... = Type: Confirmable (0)
    .... 0000 = Option Count: 0
    Code: GET (1)
    Transaction ID: 27328
    Payload Content-Type: text/plain (default), Length: 17, offset: 4
        Line-based text data: text/plain
            \273.well-known\004core

Constrained Application Protocol, TID: 27328, Length: 521
    01.. .... = Version: 1
    ..10 .... = Type: Acknowledgement (2)
    .... 0000 = Option Count: 0
    Code: 2.05 Content (69)
    Transaction ID: 27328
    Payload Content-Type: text/plain (default), Length: 517, offset: 4
        Line-based text data: text/plain
            \301(\261\r
            [truncated] \377</careless>;rt="SepararateResponseTester";title="This resource will ACK anything, bu
```

Merci à Laurent Toutain pour sa relecture attentive.