Griffon Anthony
Sauray Antoine
Leblanc Jimmy
Gauton Thibaud

Protocole : AJA

Problématique : permettre à une machine A d'envoyer à une machine B des données sur le réseau.

Disposition du réseau :

- 1 paquet sur le réseau
- heure identique sur tous les ordinateurs(inutile si on utilise pas une date pour le TTL)
- identifiant unique

Codage du paquet

Protocol	type	@ dest	@ src	TTL	size	check sum	payload	check sum
8 bits	8 bits	8 bits	8 bits	8 bits	16 bits	16 bits	n bits	16 bits

Protocole

Il est codé sur 8 bits. Il permet la gestion d'au plus 255 protocoles sur le réseau.

Type

Notre réseau gère au plus 255 types de paquets différents. Plusieurs types sont détaillés dans la partie "Type de paquets".

@dest

L'adresse de destination est codée sur 8 bits. Cela permet de faire cohabiter 255 hôtes sur notre réseau. Il est envisageable de modifier cette valeur dans une autre version du protocole. Par exemple, passer à 2^{16}=65536 hôtes possibles (au détriment de la performance).

@src

Identique à l'adresse destination. Le nombre de bits alloués devra suivre le nombre de bits alloués à la partie @dest.

TTL

Le TTL (ou Time To Live) détermine la durée de vie d’un paquet sur le réseau. La valeur initiale du TTL d'un paquet est de 255 et est décrémenté par la suite (voir plus bas), on lui alloue donc 8 bits.

Size

Il détermine la taille du payload. Il est codé sur 2^{16}. Il permet de coder la taille du payload jusqu’à 65536 octets.

checksum (1 / 2)

Il permet de vérifier l’intégrité du header du paquet. Il est codé sur 16 bits.

Payload

Il contient l’ensemble des données transmises. Son codage dépend de la valeur de Size. Sa taille peut varier de 0 à 65536 octets.

checksum (2 / 2)

Il permet de vérifier l’intégrité du payload. Il est codé sur 16 bits.

Types de paquets

token (type particulier)

Le token (ou jeton) autorise une machine à envoyer des paquets sur le réseau. Sans possession du token, seuls les paquets de type accusés de réception sont autorisés.

Protocol	type	@ dest	@ src	TTL	size	check sum	payload	check sum
AJA	token	@dest	my_id					

accusé de réception

Permet de transmettre un accusé de réception à une machine (acknoledgement) afin de confirmer la bonne réception d'un paquet.

Protocol	type	@ dest	@ src	TTL	size	check sum	payload	check sum
AJA	ack	@dest	my_id					

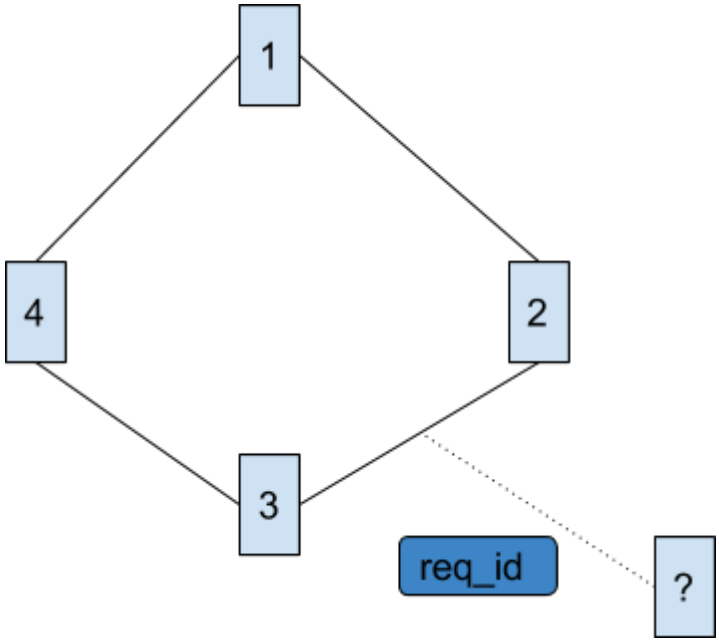
données

Permet de transmettre des données à un hôte.

Protocol	type	@ dest	@ src	TTL	size	check sum	payload	check sum
AJA	data	@dest	my_id					

req_id

“Demande d’un identifiant”
Permet à une machine ne possédant aucun identifiant, de faire une demande pour s’en voir attribuer un. Typiquement cet appel est réalisé après le branchement d’une nouvelle machine.



Protocol	type	@ dest	@ src	TTL	size	check sum	payload	check sum
AJA	count_ host	0	0					

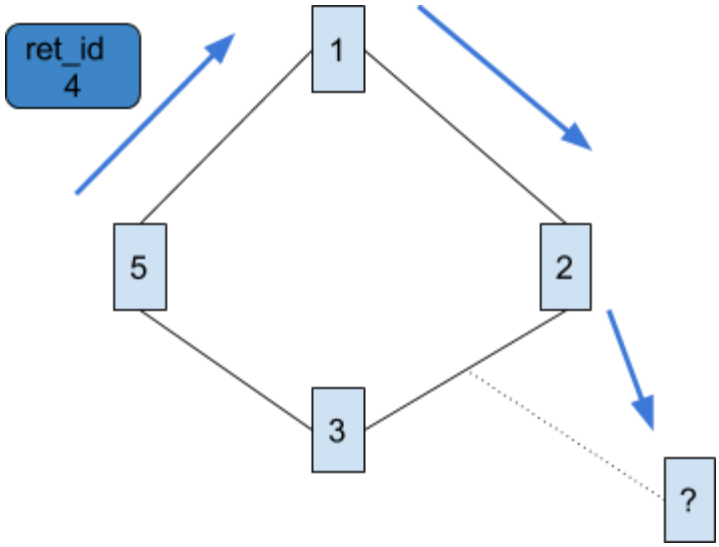
Ce paquet transite sur le réseau en broadcast. Il s’arrête à la première machine rencontrée sans id (id=0). Pour des raisons de convenance, il est conseillé de brancher les machines une par une sur le réseau, ceci afin d’éviter des interceptions de paquets.
Attention toutefois, une interception de paquet ne représente aucun problème en soi (si ce n’est une perte de temps). La machine ayant émit la demande d’id s'apercevra qu’elle n’a pas reçu d’id et réalisera une nouvelle demande.

Algorithme

```
if (paquet [type] == req_id){
        // si le paquet est de type req_id
        if ( paquet [@dest] != paquet [@src] ){
                // si l'adresse de destination n'est pas la source
                // le paquet n'a pas encore fait le tour
                // On va traiter le paquet
                if( paquet[payload] != my_id ){
                        // Si le payload ne contient pas mon propre id
                        // Alors il existe un vide entre l'id précédent et le mien
                        // ex : 1 -  3
                        // On va donc renvoyer l'id trouvé.
                        // On a trouvé un id.
                        id = paquet [payload] - 1;
                        // On va stopper la boucle ! on envoit l'id sur le réseau
                        setId(id);
                }
                else{
                        if(my_id != 0 ){
                                // Notre machine a un identifiant
                                // on fait suivre avec l'id suivant (en théorie notre id + 1)
                                reqId (my_id ++ );
                        }
                        else{
                                // Notre machine n'a pas d'identifiant !
                                // On fait suivre sans changer le paquet
                                reqId (paquet [payload] );
                                // On fait suivre avec l'id déjà présent
                        }
                }
        }
        else{
                // si jamais le paquet est revenu a son point de départ
                if ( paquet[payload] != 255){
                        // si le paquet n'est pas égal au max
                        // On s'attribue l'id
                }
                else{
                        // Sinon échec ! Trop de machines sur le réseau
                }
        }
}
```

ret_id

“Retourne un identifiant”
Permet d’indiquer à une machine le nouvel identifiant qu’elle doit porter. Ce paquet permet en outre d’intégrer une machine sur le réseau ou de modifier un identifiant de machine sur le réseau.



Protocol	type	@ dest	@ src	TTL	size	check sum	payload	check sum
AJA	ret_id	0	0				id	

Algorithme

```
if (paquet [type] == ret_id){
        // si le paquet est de type ret_id
        if ( my_id = 0 ){
                // Si je n’ai pas d’id j’intercepte
                my_id = paquet [payload];
        }
        else {
                // sinon je fais suivre
                retId();
        }
}
```

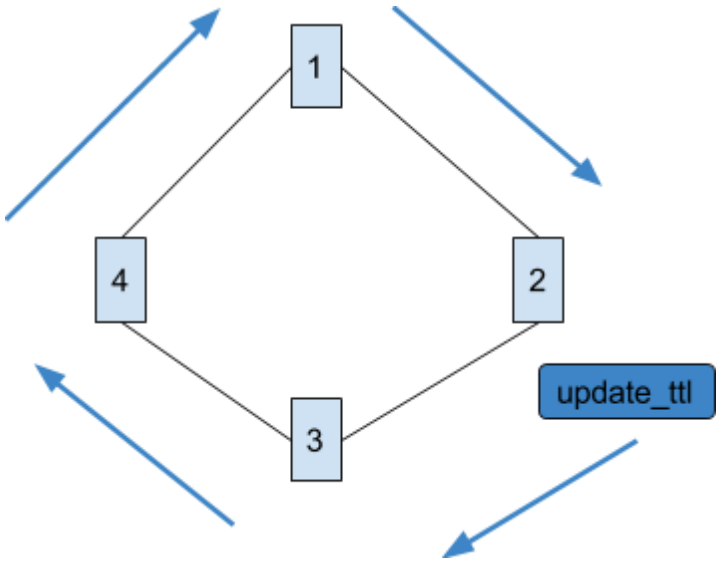
TTL

- Le TTL : Time To Live est la durée de vie d'un paquet.
- On l'exprime par la date à laquelle le paquet doit mourir.
- Le TTL est défini fixement suivant le nombre de machines maximales potentielles du réseau. En effet, si un hôte envoie un paquet a l'hôte situé juste avant lui sur le réseau, il devra faire tout le tour du réseau avant d'atteindre sa cible.
- La valeur initiale du TTL d'un paquet est donc de 255, elle est ensuite décrémentée de 1 chaque fois que le paquet passe par un nouvel hôte. Quand la valeur de 1 est atteinte, le paquet est détruit.

Cependant, on peut imaginer un TTL variable suivant le nombre de machine actuelles du réseau, que nous ne souhaitons pas implémenter car pour tout mettre à jour il faut faire transiter beaucoup de paquets, sacrifice que nous ne pouvons faire à cause de la condition d'un paquet sur le réseau. Si on veut le faire, voir fonction suivante : update_ttl

update_ttl (utile si TTL dynamique)

Indique à chaque machine recevant le paquet de mettre à jour son TTL avec le count_host fourni. Permet d'équilibrer les performances du réseau.



Protocol	type	@ dest	@ src	TTL	size	check sum	payload	check sum

AJA	update_ttl	0	my_id	max_ttl			new_ttl	

Cas typiques d'utilisation:

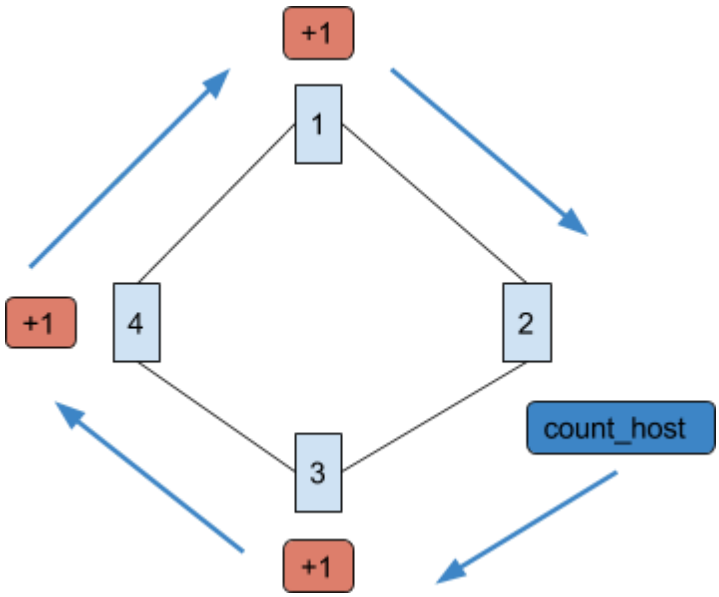
- Perte de paquets trop importante (après 5 échecs consécutifs)
- Sinon Envoi quotidien pour optimiser le réseau (en heure creuse 4h du matin)

Algorithme

```
if (paquet [type] == update_ttl ){
        my_ttl = paquet [ payload ];
        if( paquet [ @dest ] == my_id){
                updateTtl();
        }
}
```

count_host (utile si TTL dynamique)

Retourne le nombre d'hôtes sur le réseaux. C'est un paquet utile pour calculer le TTL dynamique.. Cette requête est souvent associée à une requête du type time_req.



Protocol	type	@ dest	@ src	TTL	size	check sum	payload	check sum
AJA	count_host	my_id	my_id				nb_host	

Algorithme

```
if (paquet [type] == count_host){
	if( paquet [ @dest ] == my_id){
		nb_host = paquet [ payload ];
	}
	else{
		countHosts(payload++);
	}
}
```

time_req (utile si TTL dynamique temporel utilisé : premier prototype)

Retourne le temps maximum d'échange entre deux machines. Le paquet transite sur le réseau en simulant le cas le plus défavorable.

Protocol	type	@ dest	@ src	TTL	size	check sum	payload	check sum
AJA	set_id	0	0					

Algorithme

```
if (paquet [type] == time_req){
	if( paquet [ @dest ] == my_id){
		max_time = paquet [ payload ];
		// max_time contient le temps total des machines sur le réseau
		// a diviser par le nombre de machines sur le réseau pour obtenir le temps
		// moyen haut pour des très longtemps traitements
	}
	else{
		sleep ( 200 ); // Simulation d'un traitement long de 200ms
		timeReq();
	}
}
```

Multiplexage

Fonctionnement :

Même fonctionnement qu'avant mais ici on "encapsule" les paquets.

Le paquet qui transit va être une sorte de "train" où l'on va concaténer tous les paquets que les hôtes veulent envoyer, comme ceci :

Paquet : ([Train | NbMessages, [type, ID_S, ID_D, Message1], [type, ID_S, ID_D, Message2], ...])

L'hôte qui a le token (arbitraire au début, disons la première machine ajoutée) créé donc un paquet contenant le nombre de messages actuels dans le train, et tous les messages qu'il doit envoyer (on fixe un nombre arbitraire de 10 pour l'instant, afin d'avoir un multiplexage assez élevé, mais pouvoir quand même être rapide). Chaque message est un "paquet type" de l'exercice 1 :

Protocol	type	@ dest	@ src	TTL	size	check sum	payload	check sum

Le paquet parcours tout le réseau jusqu'à revenir à la source.

Chaque hôte recréé un paquet avec pour nombre de messages 0. Puis il analyse le paquet en suivant le modèle de l'exercice 1, en regardant le nombre de messages, puis le champ destination de chaque message.

- Si le message lui est destiné, il ajoute dans le paquet qu'il a créé un accusé de réception sans incrémenter le nombre de messages. Il retire aussi l'ancien message qui lui était destiné
- Si le message ne lui est pas destiné, il ajoute le message dans le paquet qu'il a créé(sans modifier la source) et incrémente le nombre de message de 1.

Le paquet contiendra alors les messages non lus et les accusés de réception de chaque hôte ainsi que les autres paquets de l'exercice 1 si besoin (req_id, set_id…).

Lorsque le paquet a fait le tour du réseau et arrive donc à la source, on peut observer 3 cas :

- Le paquet est vide: On suppose que les messages et accusé de receptions sont tous arrivé, on fait passation du "token" ou du rôle de maitre au prochain avec un paquet de l'exercice 1: "passage token".

- Le paquet n'est pas vide: On le refait passer.

Remarque: On peut (pour faire passer le token ou rôle maître) faire en sorte que au bout de 5 tours ou plus, quand le train passe et n'est pas plein, on rajoute un paquet "passage token", ce qui va passer le token ou rôle maître au prochain hôte sur le réseau.

début 3ème partie:

Il s’agit ici d’établir un “handshake” avec un autre hôte. Pour cela nous allons implémenter trois nouveaux types dans notre protocole (Handshake_début, handshake_fin, handshake_ok).

Un hote va envoyer à un autre hote la demande de handshake, une fois établis, le premier va envoyer des trains de messages à l’autre hote et l’autre lui enverra ses messages avec les accusés de receptions des messages reçus jusqu’à la demande de fermeture de connexion d’un des deux hotes.