Solution #1 : chacun leur tour

Principe :

Chaque élément système envoie à des moments précis, dans un ordre précis.

Exemple :

1. Remote envoie à chaque 30ms;
2. Tank Omega et Alpha reçoivent l’info de la Remote, et enclenchent un décompte;
3. Tank Omega fini son décompte après 10ms, et envoie;
4. Tank Alpha fini sont décompte après 20ms, et envoie.

Pours :

* Facile à implémenter;
* Efficace (pas tant de calcul que ça, pas d’envois inutiles, le bear minimum);

Contres :

* Remote est toujours le centre de la comm (ça donne des problèmes de communications similaires à la figure 1, alors que ces problèmes sont peut-être résolubles);

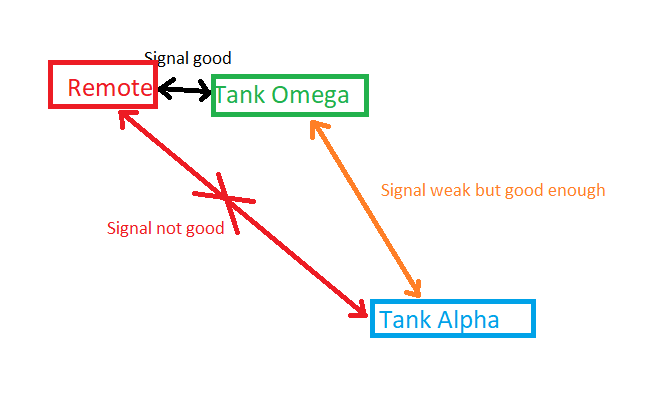


Figure 1 : deux comms good mais une bad

Solution #2 : détection de non-réponse symétrique

Principe :

C’est le même principe que celui de la solution #1, à la différence près que chaque système est capable de détecter si l’un des deux autres systèmes est absent dans la communication. Si la Remote, par exemple, détecte qu’elle ne reçoit plus d’information du Tank Alpha, elle envoie une commande qui indique au Tank Omega de transférer l’information au Tank Alpha.

Exemple :

1. Remote envoie à chaque 30ms;
2. Tank Omega reçoit l’info de Remote, et enclenche un décompte, mais Tank Alpha n’a rien reçu, donc lui, il n’enclenche pas son décompte;
3. Tank Omega fini son décompte après 10ms, et envoie son data;
4. Tank Alpha détecte que Remote est trop loin, car il n’a pas reçu son data en dedans de 40ms (la dernière commande reçue date de 30ms+20ms = 50ms), donc il envoie une commande spéciale qui indique à Tank Omega de parler à sa place à Remote;
5. Remote reçoit le data de Tank Alpha par le biais de Tank Omega;

La figure 2 montre bien le problème de communication que cette solution résout.

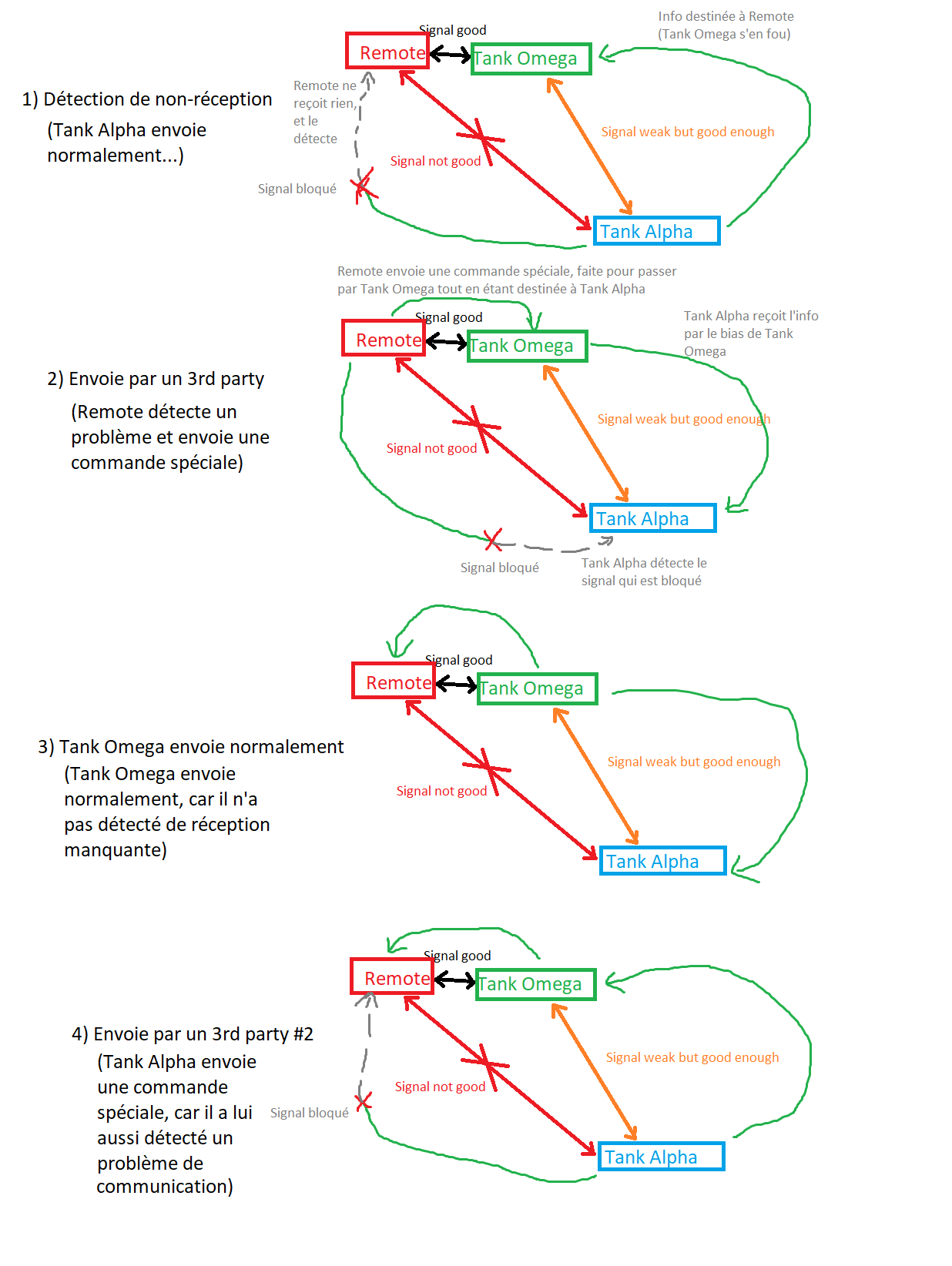


Figure 2 : communication par un « 3rd party »

Pours :

* Solution qui fonctionne pour un plus grand nombre de cas;
* Aucun système n’est le centre de la communication (ou plutôt, chaque système peut devenir le centre de la communication…);

Contres :

* Un peu plus compliqué à faire (il y a une complexité de plus avec les « commandes spéciales »);
* Certains « edgecases » ne sont toujours pas résolus. Par exemple, si la communication est bloquée dans une direction, mais pas dans l’autre, le premier système qui ne reçoit rien va envoyer une commande spéciale (jusque-là, tout va bien). Le deuxième système va recevoir le data, mais n’enverra pas de commande spéciale, même si le data n’aboutira pas, car lui, il reçoit bien et correctement le signal du premier. (Plot twist : la solution #2.5 va régler ce problème hehe). La figure 3 montre ce problème;

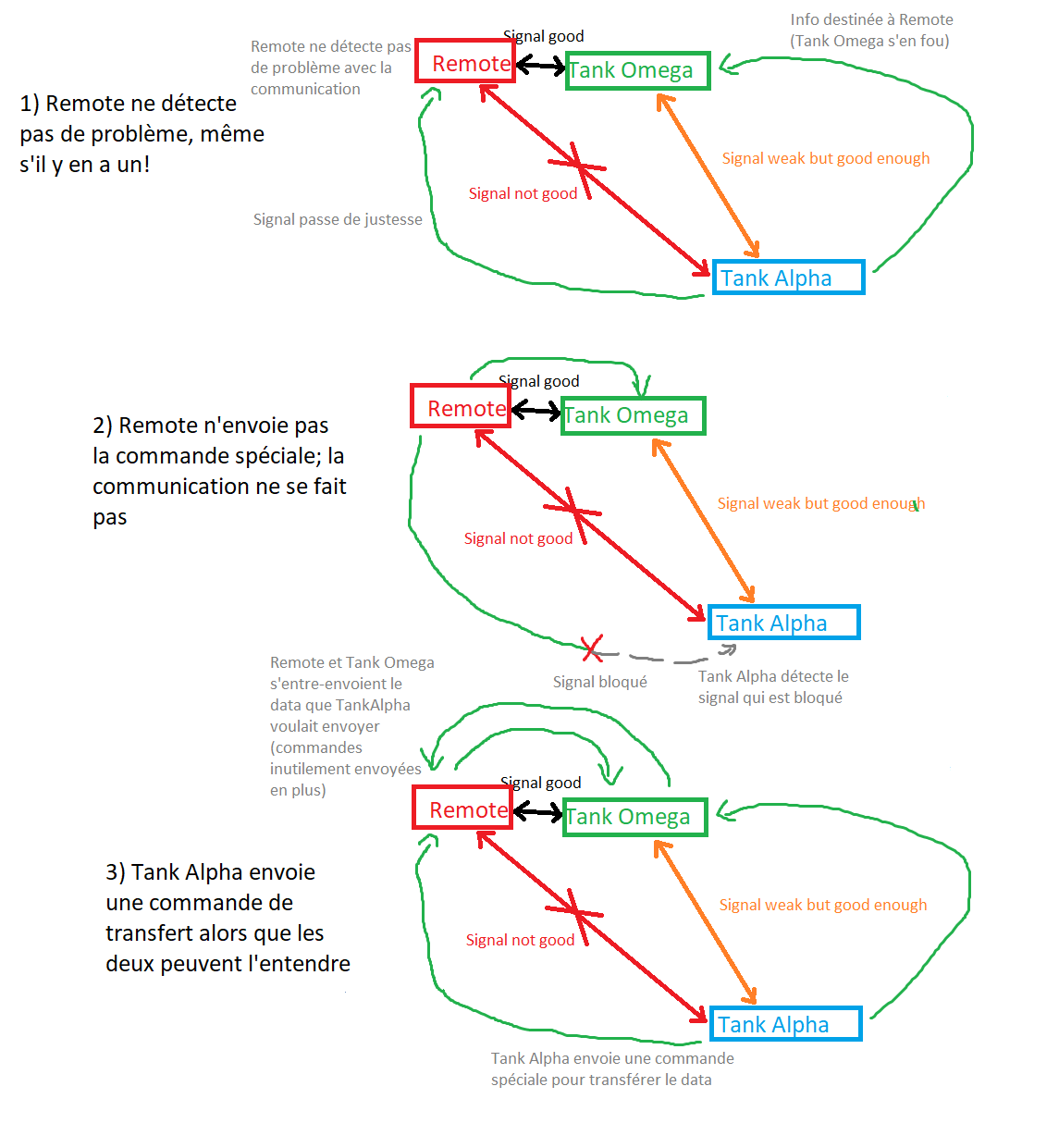


Figure 3 : one way comm brings back the problem we had

Solution #2.5 : détection de non-réponse asymétrique

Principe :

Le principe est le même que celui de la solution #2, à la différence près que les systèmes peuvent détecter les réceptions « doubles ».

Pour que cette solution marche, on a besoin de deux « types » de commandes, si on veut :

* Le type « normal », qui sert à transmettre du data une seule fois. Le data reçu par les autres systèmes n’est pas retransmit. Generate, consume, the end of transmission;
* Le type « transférant », qui sert à transporter du data d’un système à un autre, en passant par un système intermédiaire. Avec cette commande, il faut spécifier un destinataire à qui la commande s’addresse. Le data contenu dans ce type de commande ne sert pas strictement au système qui le reçoit. Celui qui reçoit ce type de commande doit nécessairement transformer le type de la commande en type « transféré » (on va le voir juste après, c’est le 3e type!), et il doit la renvoyer immédiatement aux autres systèmes, à l’exception près que le destinataire à qui il doit transférer la commande soit lui-même. Dans le cas où il est son propre destinataire d’une commande de type « transférant », il doit s’envoyer à lui-même la commande transformée (et donc pas l’envoyer du tout (c’est important, on va voir pourquoi), et juste setter sa variable de réception de commande comme s’il avait vraiment reçu une commande par RF);

Exemple :

1. Remote envoie à chaque 30 ms;
2. Tank Omega reçoit le data de Remote et enclenche son décompte interne de 10ms, mais Tank Alpha ne reçoit pas la commande;
3. Tank Omega transmet son data, tout va bien jusque-là;
4. Tank Alpha a détecté que Remote n’a pas réussit à lui transmettre son data. Il décide donc d’envoyer une commande de type « transférant » à Remote par le biais de Tank Omega (destinataire = Remote);
5. Tank Omega reçoit le data à transférer, transforme la commande en type « transféré », et la renvoie par RF;
6. Remote reçoit le data de type « transférant » AUSSI (car il a accès au data que Tank Alpha envoie (c’est un cas de communication à sens unique de Tank Alpha vers Remote)), mais il voit que le destinataire, c’est lui-même. Il ne transfert pas le data aux autres;
7. Remote reçoit le data de type « transféré » de la part de Tank Alpha et transféré par Tank Omega;
8. Remote se rend compte qu’il a reçu du data de type « transférant » et de type « transféré » en même temps au moment où Tank Alpha devait transférer son data. Il comprend donc que Tank Alpha ne l’entend pas;
9. Quand c’est à nouveau son tour de transmettre de l’information à Tank Alpha, il envoie une commande de type « transférant » à Tank Alpha par le biais de Tank Omega (car il sait que Tank Alpha ne le voit pas);

Pours :

* On gère tous les cas frauduleux! Avec cette technique, on n’a plus de problème de communication dû à une centralisation d’une des stations (enfin!);

Contres :

* Dès qu’une des stations se met à transmettre du data de type « transférant », il n’y a plus moyen de lui faire envoyer du data de type « normal » à nouveau! Si jamais la communication redevenait disponible pour tout le monde, Tank Alpha recevrait maintenant du data de type « transférant » et « transféré », ce qui lui ferait transmettre du data de type « transférant », ce qui va faire en sorte que Remote va recevoir du data de type « transférant » et « transféré », ce qui va faire en sorte que Remote va encore envoyer du data de type « transférant »… vous voyez le genre? (Plot twist #2 : en réalité, on a pas besoin de type « normal »! Ça, c’est la solution #3 :D);
* Commence à être considérablement compliqué à gérer;

Solution #3 : transfert avec destinateur et destinataire

Principe :

Le principe est le même que celui de la solution #3, à la différence près qu’il n’y a pas de data de type « normal ». À chaque fois qu’on envoie du data, c’est toujours, TOUJOURS de type « transférant ». Les trames contiennent toujours l’ID du destinateur (le transmetteur) et du destinataire (le récepteur). Si une station reçoit une trame, et qu’il est le destinataire, alors il consomme la trame, et ne la renvoie pas (la trame à aboutie). S’il est le destinateur (ce qui implique qu’il a reçu la même trame qu’il a envoyé), il ne la renvoie pas non-plus, mais il ne la consomme pas. S’il n’est ni le destinateur, ni le destinataire (il est donc un 3rd party), il ne fait que la renvoyer, sans la consommer.

Exemple :

1. Remote envoie à chaque 30ms. La trame envoyée contient deux commandes : une pour Tank Alpha, et une pour Tank Omega;
2. Tank Omega reçoit et commence à compter 10ms, mais Tank Alpha ne reçoit pas;
3. Aussitôt que Tank Omega a fini de recevoir la trame de Remote, il la décode et voit qu’il est le destinataire d’une des deux commandes, mais qu’il n’est ni le destinataire, ni le destinateur de l’autre. Il consomme donc la première, et renvoie la deuxième sans la consommer;
4. Remote reçoit la trame de Tank Omega (celle qu’il vient d’envoyer as a matter of fact), et il voit qu’il est le destinateur dans la trame. Il décide donc de ne pas la renvoyer;
5. Tank Alpha reçoit la trame de Tank Omega, il voit qu’il est le destinataire, et que Remote est le destinateur. Il la consomme donc et ne la renvoie pas;
6. 10ms plus tard, c’est au tour de Tank Omega d’envoyer son data, ce qu’il fait;
7. Tank Alpha reçoit le data, et le renvoie à Remote sans le consommer, car il n’est pas le destinataire ni le destinateur;
8. Remote reçoit presque simultanément une trame en provenance, d’abord de Tank Omega, et ensuite de Tank Alpha. Il voit qu’il est le destinataire des deux trames, et il les consomme donc les deux, sans les renvoyer;
9. 20ms plus tard, Tank Alpha envoie sa trame, etc.

Pours :

* Fait la même job que la solution #2.5, mais est plus efficace à parser (un type de commande de moins) et plus simple à implémenter;
* Marche pour tous les edgecases (décentralisation de Remote);

Contres :

* Ça arrive souvent qu’on envoie une trame qui a déjà été reçue (pas un problème en tant que tel, genre ça va marcher pareille, mais c’est pas efficace de consommer deux fois le même data de suite pour rien (on peut palier à ça avec des command IDs par exemple, genre on numérote les commandes. Si on reçoit un numéro plus petit ou égal au numéro de la dernière commande reçu, on assume qu’on l’a déjà consommé/renvoyé));