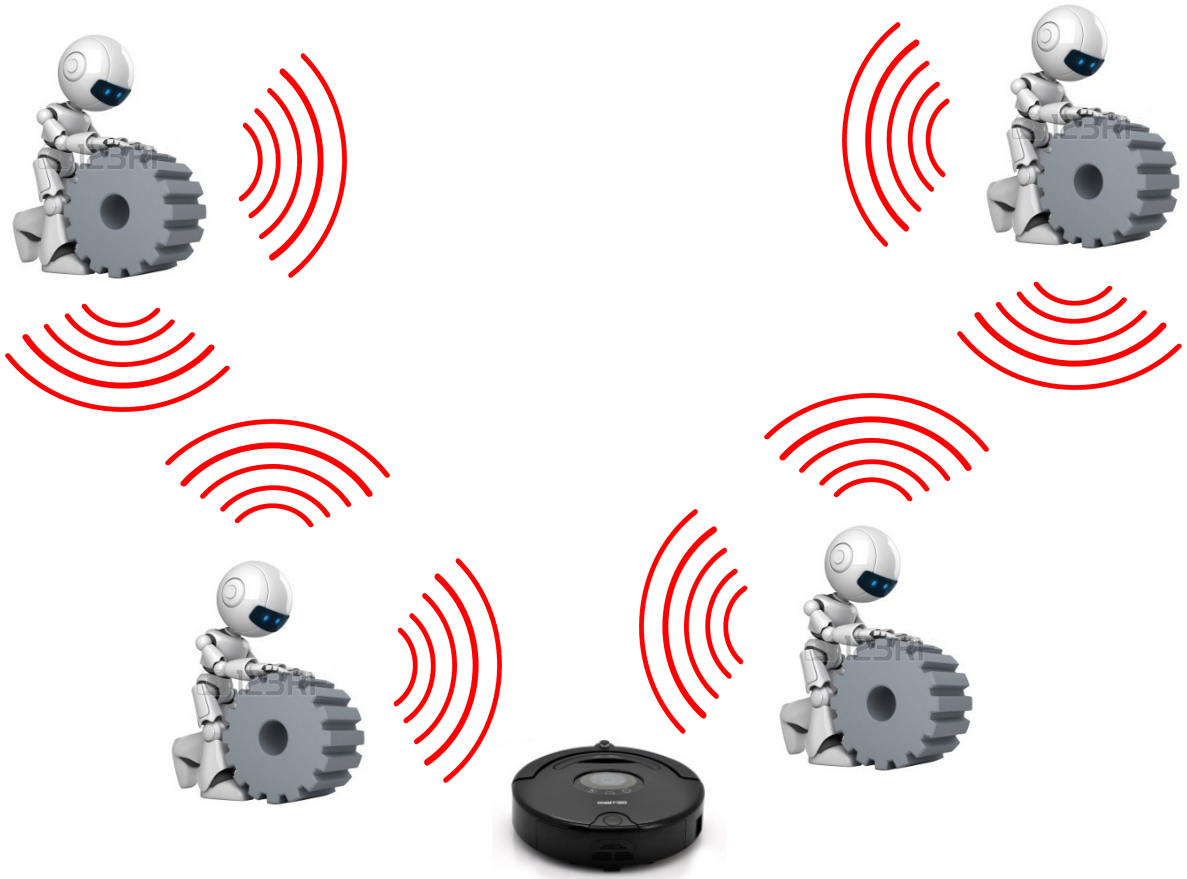
Synchronisation Distribuée
Exclusion mutuelle pour les environnements incertains

Laurent PHILIPPE

Master 2 Informatique
UFR des Sciences et Techniques

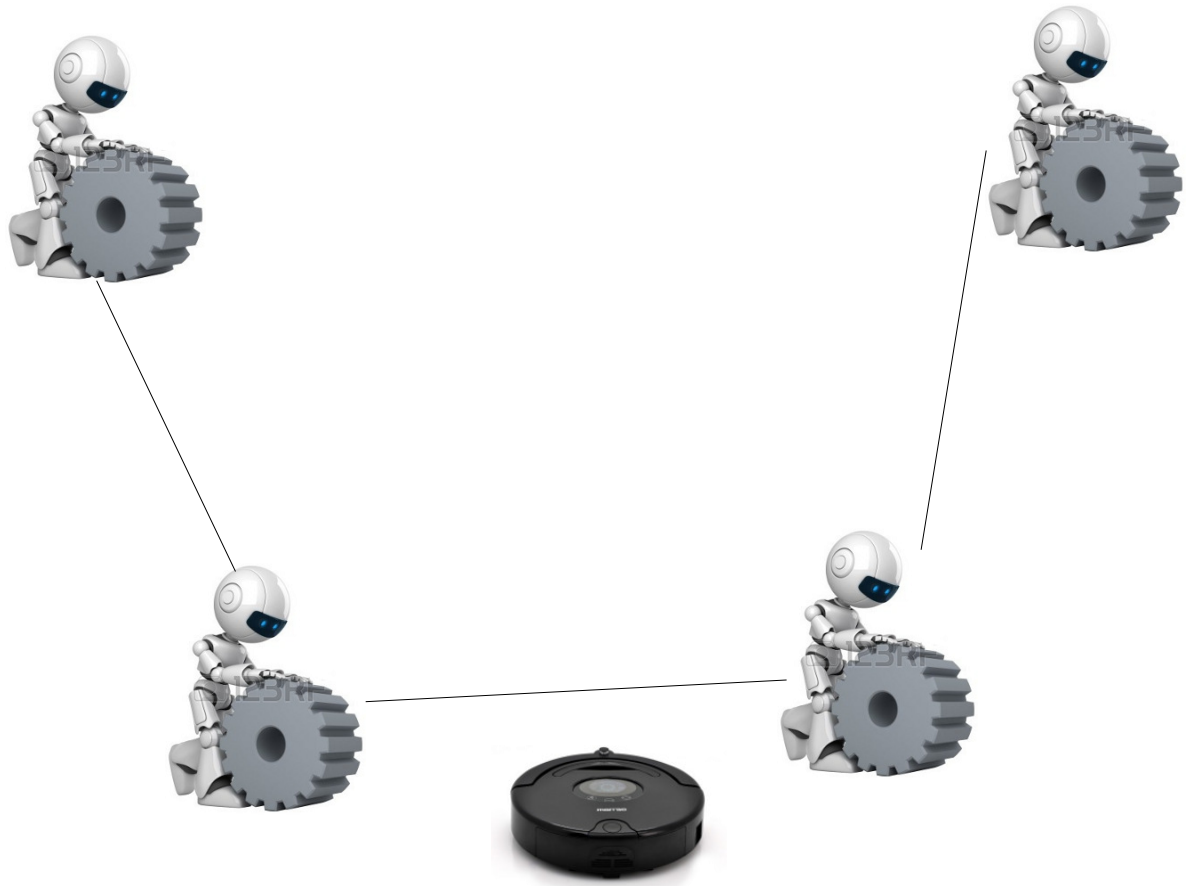
2022/2023

Communication point-à-point



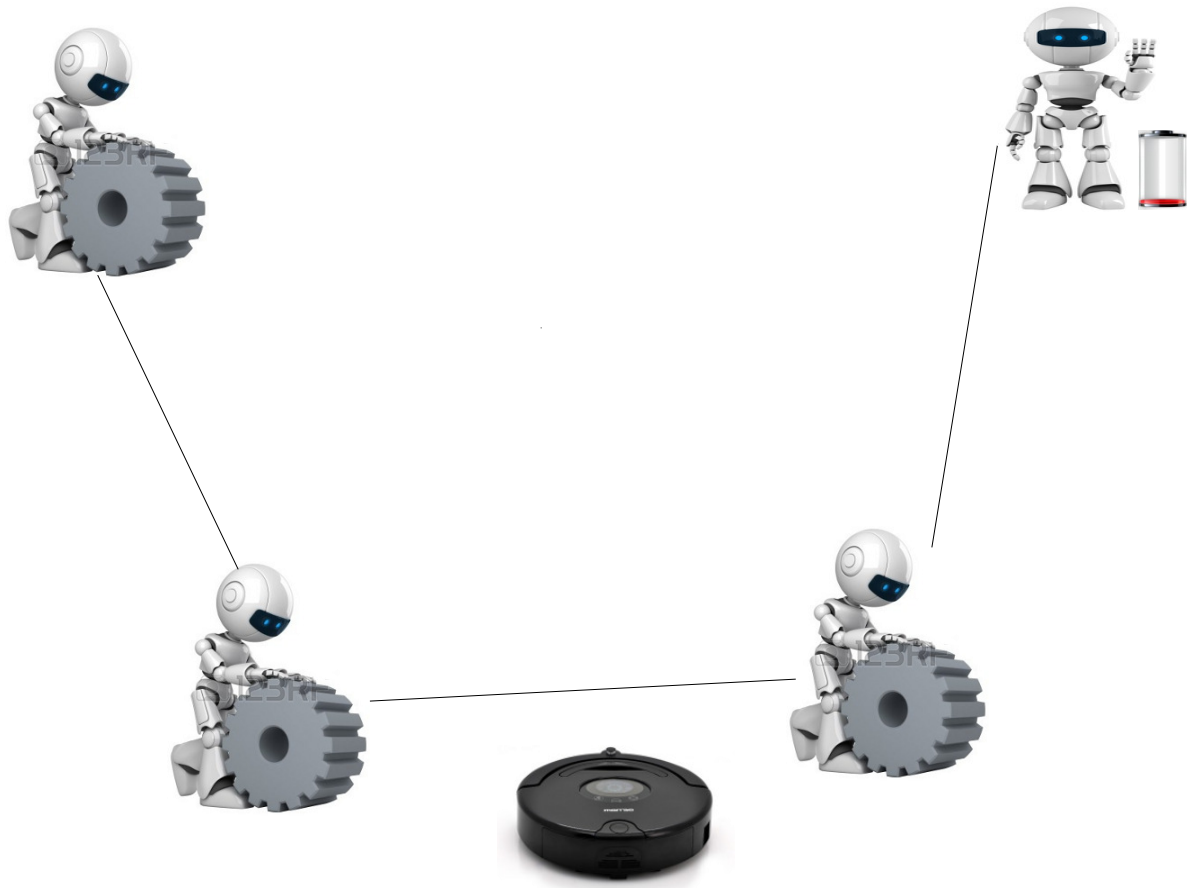
Les robots ne peuvent pas communiquer avec toute la flotte

Communication point-à-point



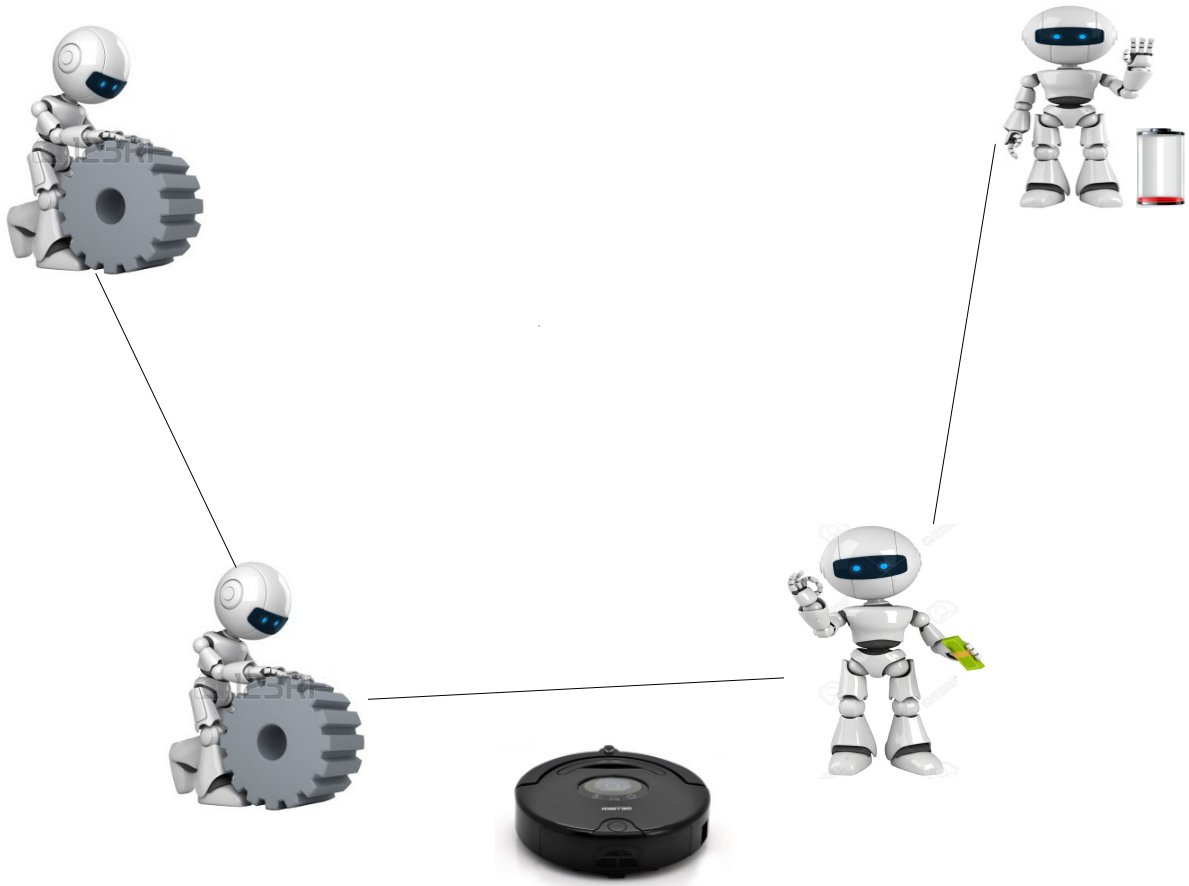
La portée du système de communication définit un graphe de communication

Communication point-à-point



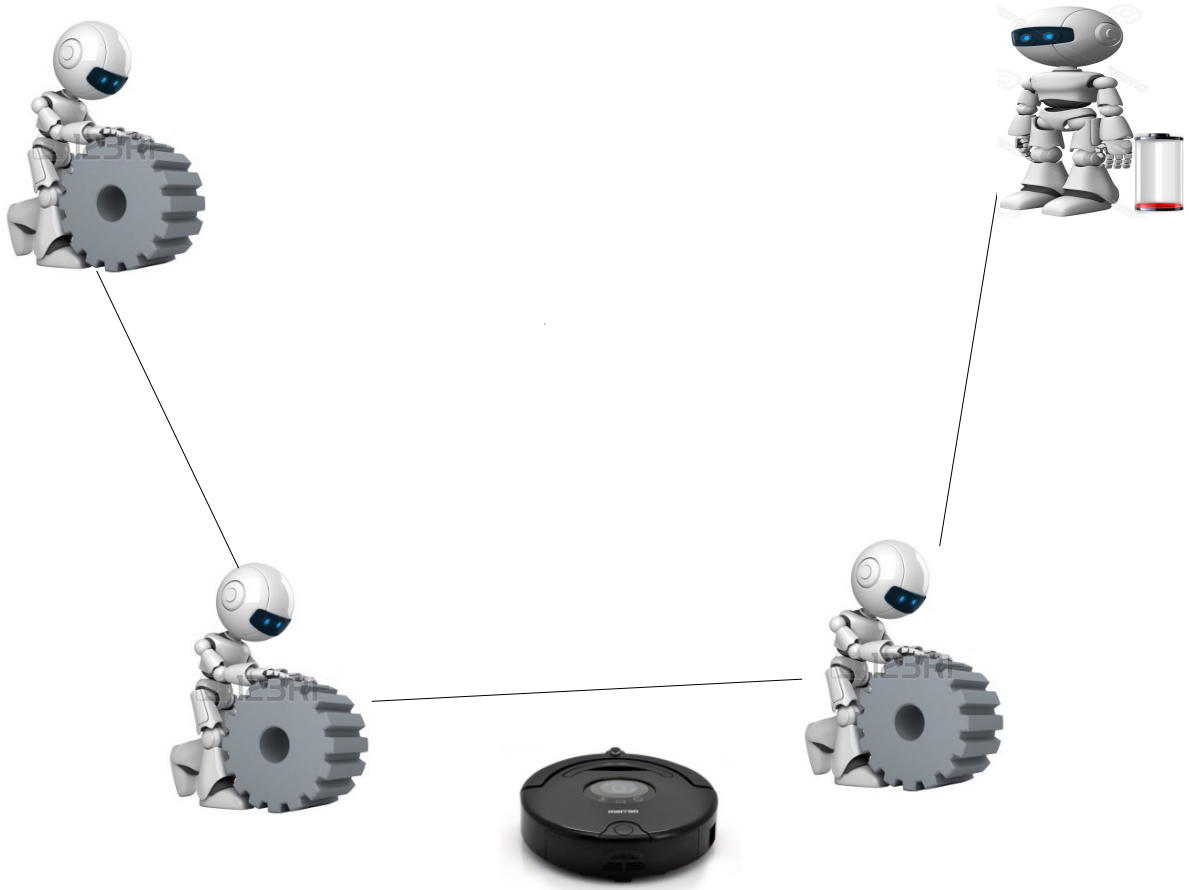
Si un robot demande l'accès à la base avec l'un des algorithmes présentés, il envoie son message à tous les robots qu'ils connaît.

Communication point-à-point



Seuls les robots qui reçoivent son message répondent

Communication point-à-point



Le robot attend l'acquittement des autres

Sommaire

Exclusion mutuelle sans réseau complet

Solutions

- Comment résoudre ce problème ?

Exclusion mutuelle sans réseau complet

Solutions

- Mettre en place un protocole de routage
- Mettre en place un anneau virtuel
- Utiliser les algorithmes vus précédemment

Exclusion mutuelle sans réseau complet

Exercice

Proposer un algorithme d'exclusion mutuelle sans réseau complet, avec les conditions suivantes :

- Les processus possèdent un identifiant unique
- Nombre de processus fixé et fixe (pas de panne de processus)
- Le réseau est connexe, pas de coupure
- Tous les processus ne peuvent pas communiquer directement
- Pas de perte de message, ni message faux

Algorithme de Maekawa

Principe (1)

- Les processus sont répartis en sous-ensembles
- Tous les processus appartiennent au moins à un sous-ensemble
- Les intersections de sous-ensemble sont non vides deux à deux : un sous-ensemble possède toujours en commun au moins un processus avec chacun des autres sous-ensembles
- Pour entrer en section critique le processus demande seulement à son sous-ensemble
- Les messages de requêtes sont estampillés : gestion de priorité entre les requêtes
- Les processus s'engagent (votent) un pour un processus
- Les processus ne peuvent voter que pour un processus

Algorithme de Maekawa

Principe (2)

- Demande de section critique : le processus envoie un message *REQUEST* aux processus de son sous-ensemble
- À la réception d'un message *REQUEST*, si le processus n'a pas déjà donné son vote (état *locked*), retourne un message *LOCKED*, sinon met la demande en attente.
- Si la requête courante est plus prioritaire, retourne un message *FAILED*
- Si la requête reçue est plus prioritaire, envoie un message *INQUIRE* au processus pour lequel il s'est engagé, sauf si déjà demandé
- À la réception d'un message *INQUIRE*, si le processus a reçu un message *FAILED* il rend le vote ; message *RELINQUISH* sinon met en attente.

Algorithme de Maekawa

Principes (3)

- Sortie de section critique : envoie un message *RELEASE* aux membres du sous-ensemble
- Si un processus reçoit un message *INQUIRE* après avoir envoyé un message *RELEASE*, le message est ignoré.
- À la réception d'un message *RELINQUISH*, le processus donne son vote à la requête la plus prioritaire : envoie un message *LOCKED* au processus correspondant.
- À la réception d'un message *RELEASE* : le processus donne son vote à la requête la plus prioritaire : envoie un message *LOCKED* au processus correspondant.
- Si la file est vide le processus se marque *unlocked*

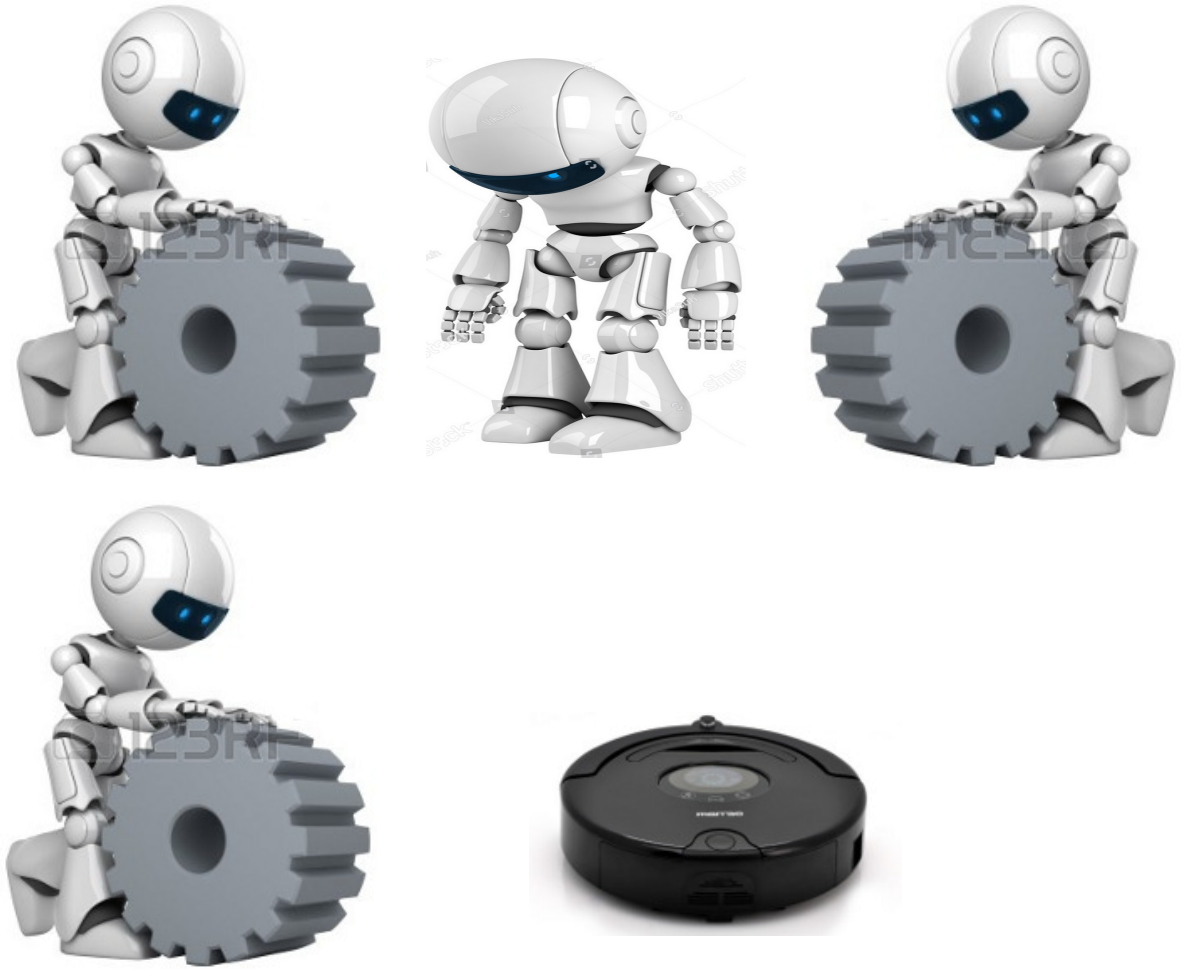
Algorithme de Maekawa

Difficultés

- Constitution des sous-ensembles
- Mettre en place un algorithme de constitution de sous-ensembles

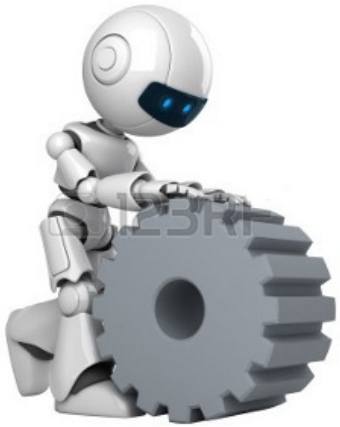
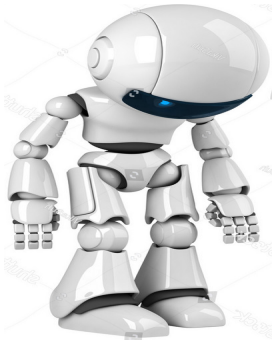
Sommaire

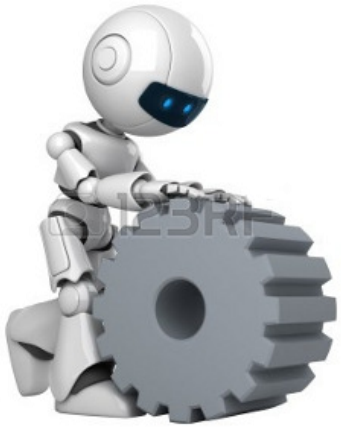
Exemple



Un robot tombe en panne

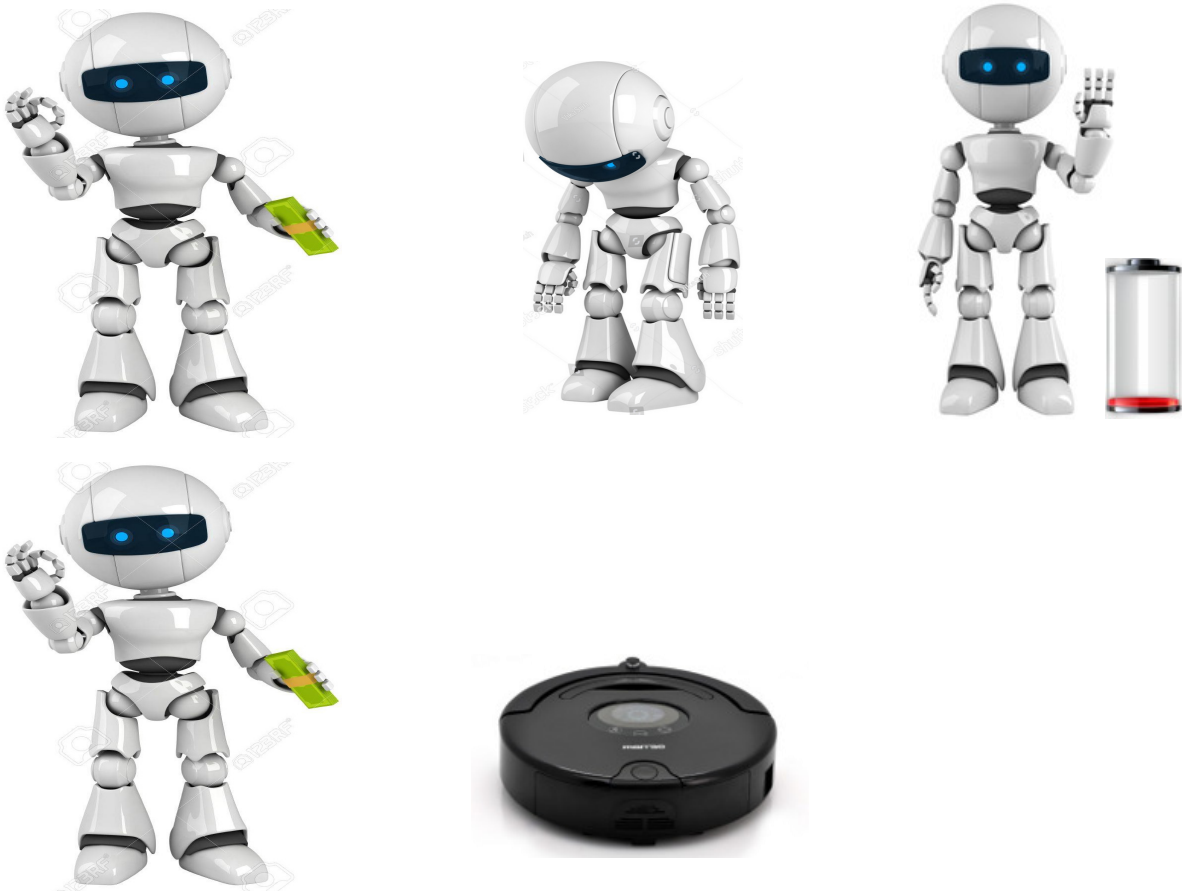
Exemple



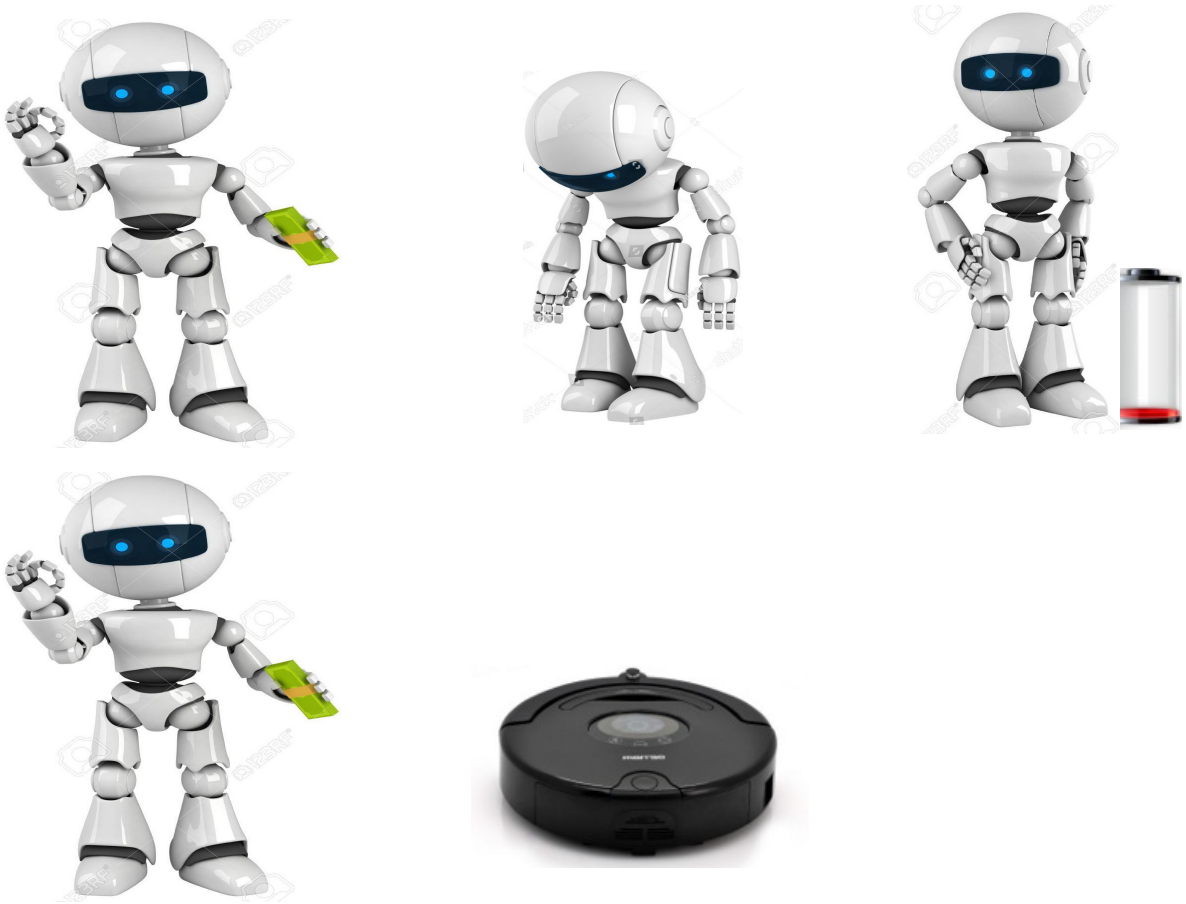

Le robot lance l'algorithme d'exclusion mutuelle

Exemple



Seuls les robots en état de marche donne leur accord

Exemple



Le demandeur n'a pas tous les accords, le système est bloqué

Exclusion mutuelle en cas de panne

Solutions

- Refaire l'anneau : perte du jeton ?
- Changer l'ensemble de diffusion
- Mettre à jour le routage : à quelle fréquence ?
- Comment détecter les pannes ?
- Adapter les algorithmes : NT, Lamport, etc.
- Proposer de nouveaux algorithmes

Tolérance aux pannes dans l'algorithme de Naimi-Tréhel

Sensibilité aux pannes

- Un processus tombe en panne dans la file des *owner*
- Un processus tombe en panne dans la file des *next*
- Le processus qui possède le jeton tombe en panne

Tolérance aux pannes dans l'algorithme de Naini-Tréhel

Exercice

- Proposer une adaptation de l'algorithme de Naimi-Tréhel qui supporte les pannes
- Hypothèse : pas de perte de message

Adaptation de l'algorithme de Naimi-Tréhel

Modifications

- Base reste la même :
- Détection des pannes de processus :
- Récupération après erreur :
- Perte du jeton :

Adaptation de l'algorithme de Naimi-Tréhel

Modifications

- Base reste la même :
  - ajout de nouveaux types de messages et nouvelles règles
  - états du processus : $state \in \{wait, consult, query, cand, obs\}$
- Détection des pannes de processus : diffusion de messages et timeouts
- Récupération après erreur : diffusion de messages et timeouts
- Perte du jeton : élection

Adaptation de l'algorithme de Naimi-Tréhel

Principe (1)

- Détection des pannes : délais T_{wait}, basé sur le temps de communication et le temps passé en section critique
- Envoi de $REQ(P_i)$: P_i arme T_{wait}
- Timeout T_{wait}(suspicion de panne) : P_i diffuse le message *CONSULT* associé au timeout T_{elec} (délais max de diffusion/réponse)
- Réception *CONSULT* : P_j répond avec le message *QUIET* si $next = P_i$
- Timeout T_{elec} + P_i n'a pas reçu de message (panne avérée) : réactive T_{elec} et diffuse le message *FAILURE*

Adaptation de l'algorithme de Naimi-Tréhel

Principe (2)

- Réception *FAILURE* : P_j répond *PRESENT* à P_i s'il a le jeton
- Problème jeton en transit : P_j mémorise la demande et envoie *PRESENT* à P_i si reçoit le jeton plus tard.
- Réception *PRESENT* : P_i renouvelle sa demande de section critique à P_j
- Timeout T_{elec} + P_i n'a pas reçu de message (jeton perdu) : P_i réactive T_{elec} et diffuse $ELECTION(i)$

Adaptation de l'algorithme de Naimi-Tréhel

Principe (3)

- Réception *ELECTION*(j) : P_i ne répond que si $i > j$
- Messages *ELECTION* concurrents : processus avec le plus petit identificateur est élu
- Timeout $T_{elec} + P_i$ n'a pas reçu *ELECTION* plus petit que le sien : P_i est élu, régénère le jeton et diffuse un message *ELECTED*
- Réception *ELECTED* : les processus réinitialisent leurs données.

Adaptation de l'algorithme de Naimi-Tréhel

Sensibilité aux pannes

- Définition du timeout T_{wait} : temps de passage de tous les processus en section critique + temps de communication = peut être long (temps de recharge des robots par exemple)
- Un processus tombe en panne dans la file des *owner*
- Un processus tombe en panne dans la file des *next*
- Le processus qui possède le jeton tombe en panne

Algorithme de Naimi-Tréhel, adaptation Sopena

Modifications

- Adaptation de l'algorithme de Naimi-Tréhel, même partie originale
- Détection plus tôt de la panne :
  1. timeout T_{msg} ne dépend pas du temps en section critique, uniquement des temps de messages, temps max pour parcourir l'arbre
  2. ajout d'un message *COMMIT* par le premier prédécesseur (dernier *next*), en réponse à une demande de jeton
  3. demandeur vérifie régulièrement l'état de son prédécesseur
- Maintient le plus possible la queue des *next* en incluant la file des prédécesseurs dans *COMMIT*

Algorithme de Naimi-Tréhel, adaptation Sopena

Demande de section critique

- Envoi de $REQ(i)$, P_i arme timeout T_{msg}
- P_j qui enregistre P_i comme son *next* lui envoie un message $COMMIT(file, pos)$; *file* : k prédécesseurs (*next*) de P_j ; *pos* : position dans la file (position de $P_j + 1$)
- Réception $COMMIT(file, pos)$: P_i mémorise l'info
- P_i envoi ARE_YOU_ALIVE régulièrement à P_j, avec timeout
- Réception ARE_YOU_ALIVE : P_j répond I_AM_ALIVE

Algorithme de Naimi-Tréhel, adaptation Sopena

Récupération sur faute

3 cas de panne possibles :

1. Moins de k processus en panne dans la file *next*
2. Plus de k processus en panne dans la file *next*
3. P_i ne reçoit pas *COMMIT*

Algorithme de Naimi-Tréhel, adaptation Sopena

Récupération sur faute

Cas 1 : moins de k processus en panne dans la file, un des prédécesseur peut répondre à P_i

- P_i détecte une panne de son prédécesseur (pas de réponse à ARE_YOU_ALIVE) : envoie ARE_YOU_ALIVE à tous ses prédécesseurs (k), un par un, avec le timeout.
- Réception de ARE_YOU_ALIVE : prend P_i comme son nouveau *next* et envoie I_AM_ALIVE
- P_i s'arrête quand un prédécesseur lui répond
- La file des *owner* n'est pas affectée
- L'algorithme de base reprend

Algorithme de Naimi-Tréhel, adaptation Sopena

Récupération sur faute

Cas 2 : plus de k processus en panne dans la file (prédécesseurs de P_i tous en faute, pas réponse I_AM_ALIVE)

- P_i arme T_{msg} et diffuse un message $SEARCH\_PREV$ (tentative de reconnection à l'arbre des *next*)
- Réception $SEARCH\_PREV$ par P_j : si $j < i$ répond par $I\_AM\_ALIVE(pos)$
- Réception I_AM_ALIVE : P_i mémorise pos
- Timeout T_{msg} : P_i choisit le processus avec pos max et envoie $CONNECTION$ (reconnecte à l'arbre).
- P_i n'a pas de réponse (jeton perdu) : P_i régénère le jeton et met $pos = 0$.

Algorithme de Naimi-Tréhel, adaptation Sopena

Récupération sur faute

Cas 3 : P_i ne reçoit pas de *COMMIT* (ne sait pas où se reconnecter dans la file *next*) : plusieurs cas possibles

1. Seul P_i a détecté la panne
2. Plusieurs processus ont détecté la panne

Algorithme de Naimi-Tréhel, adaptation Sopena

Récupération sur faute

Cas 3 + seul P_i a détecté la panne

- P_i arme (T_{msg}) et diffuse $SEARCH\_QUEUE$
- Réception $SEARCH\_QUEUE$ + P_j dans la file : répond $ACK\_SEARCH\_QUEUE(pos, aNext)$; pos : position dans la file ; $aNext$ booléen s'il a un $next$
- Timeout T_{msg} : P_i choisit le processus avec pos max
  là encore trois cas possibles :
  1. $aNext = false$, P_i renvoie $REQ(i)$ à P_j. L'arbre $owner$ n'est pas modifié
  2. $aNext = true$ (processus $next$ est en panne) : P_i envoie $CONNECTION$ à P_j
  3. Pas reçu de réponse (jeton perdu) : P_i régénère le jeton et met $pos = 0$.

Algorithme de Naimi-Tréhel, adaptation Sopena

Récupération sur faute

Cas 3 + plusieurs processus ont détecté la panne.

- P_i reçoit un, ou plusieurs, message(s) $SEARCH\_QUEUE$ de P_j pendant délais T_{msg}
- Mécanisme d'élection : si $j < i$ il perd l'élection et envoie un message REQ au plus petit P_j
- Si le processus P_j perd l'élection il fait comme P_i
- Si le processus P_j gagne toutes les élections, il peut continuer comme pour le cas d'un seul qui a détecté

Gestion de l'arbre *owner*

- À la réception d'un message $SEARCH\_QUEUE$, les processus mettent à jour leur *owner* avec le processus émetteur