CPS — Programmation par composants

© 2019- Jacques Malenfant

Master informatique, spécialité STL – UFR 919 Ingénierie
Sorbonne Université

Jacques.Malenfant@lip6.fr

Cours 9

Systèmes répartis à grande échelle

Les systèmes répartis à grande échelle I

- L'essor de l'internet et de la virtualisation a permis l'émergence des *systèmes répartis à grande échelle*.
  - Que ce soit pour les applications à taille mondiale ou pour le traitement de données massives, les systèmes répartis à grande échelle explosent en nombre et en taille.
  - Les prochaines séquences abordent quelques difficultés du passage à l'échelle, comme l'apparition de *goulots d'étranglement*, la *décentralisation* forte (pair-à-pair), l'*évolution dynamique* des architectures, les *délais* de diffusion de l'information, etc.
- Toutefois, ces dernières années et à moins d'un changement de base technologique à terme, une limite technologique freine cette évolution, la consommation d'***énergie*** :
  - La part de la consommation des équipements informatiques dans la production mondiale d'énergie électrique est passée de relativement négligeable avant 2000 à 5% environ en 2010 puis à plus de 13% en 2018[1] et poursuit une croissance potentiellement exponentielle.

Les systèmes répartis à grande échelle II

- Certains prédisent, de manière sérieuse et prudente, que ce serait 25% en 2025 voire 50% en 2030 compte tenu de l'engouement pour la 5G[2], la *blockchain*[3], les *big data* et l'intelligence artificielle (*deep learning* et LLM[4]).
- Sauf changement technologique majeur, après seulement 25 ans d'existence du web, nous touchons aux limites de sa croissance[5].

[1] Un seul centre de calcul situé en région parisienne consomme autant d'énergie électrique qu'une ville de 60.000 habitants, soit $1/1000^{\text{è}}$ de la consommation domestique totale d'électricité de la France, *un seul*.

[2] Avant même la 5G et la vidéo à très haut débit sur mobile, on estimait fin 2019 la consommation de bande passante de NetFlix à 25% et 33% de la bande passante totale installée respectivement en France et aux États-Unis aux heures de grande écoute.

[3] Au plus fort de l'engouement pour ce dernier, l'informatique soutenant le *bitcoin* consommait à elle seule autant d'énergie électrique que tout le Danemark. Des études (début 2020) estiment la consommation d'énergie du seul *bitcoin* à 0,4% de la production mondiale d'électricité. Compte tenu de l'énergie consommée par le système *bitcoin*, on peut se demander ce qui serait nécessaire pour passer le dollar américain ou l'euro en cryptomonnaie en utilisant la même technologie...

[4] Selon un article `lebigdata.fr` du 28/04/2023 consulté le 2/04/2024, ChatGPT-3 aurait consommé 1287 gigawatts heures pour son entraînement, ce à quoi il faut ajouter la consommation de son utilisation qui représenterait 60% de sa consommation totale et il consommerait autant d'eau qu'une centrale nucléaire. Selon le même article, en 2021, l'IA représentait 10 à 15% de la consommation d'électricité de Google, soit 2,3 térawatts heures anuelles, autant que la ville d'Atlanta (EU).

[5] À moins d'arriver à remplacer la technologie silicium actuelle par une nouvelle technologie permettant de produire une série de générations de processeurs offrant à nouveau, comme le silicium en son temps, une augmentation exponentielle de la puissance de calcul à consommation d'énergie à peu près constante. *Difficile à imaginer dans les prochains 10 ans...*

Plan

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre les phénomènes de variation de charge pouvant saturer les ressources d'un composant.
   - Aborder les principaux compromis de configuration d'un système gérant ses ressources et sa charge.
2. Compétences à acquérir
   - Savoir implanter des composants qui gèrent *explicitement* la quantité de parallélisme qu'ils offrent et leur charge de travail pour éviter l'écroulement par saturation de leurs ressources.
   - Savoir utiliser la personnalisation des groupes de *threads* Java pour gérer la mise en file d'attente des tâches et la politique de rejet des tâches lors de la saturation.

Saturation des groupes de *threads*

- Utiliser les groupes de *threads* à taille contrôlée évite la création de nouveaux *threads* en quantité non bornée.
- Mais pour augmenter la capacité de traitement et passer à l'échelle, le premier réflexe est d'augmenter le nombre de *threads* dans les groupes.
- Toutefois, peu importe ce nombre, une augmentation non bornée des requêtes par unité de temps finira nécessairement par saturer son groupe de *threads*.
  - À l'extrême, c'est le phénomène des *flash crowds*, c'est-à-dire l'arrivée massive et quasi-instantanée de tâches.
- Que constate-t-on alors ?
  - La taille de la file des tâches augmente de manière non bornée et la latence du service s'allonge indéfiniment.
- Conclusion : pour maîtriser la charge et la latence sur un groupe de *threads*, il faut maîtriser sa discipline de mise en file d'attente des tâches.

Personnalisation des instances de `ThreadPoolExecutor` I

- En Java, la classe `ThreadPoolExecutor` propose une implantation solide et personnalisable des groupes de *threads*.
- On peut créer un groupe avec des caractéristiques spécifiques pour adapter certains de ses comportements ; outre le nombre de *threads*, les plus utiles à adapter sont :
  - la file utilisée pour conserver les tâches en attente ;
  - la fabrique utilisée pour créer les *threads* du groupe ;
  - le gestionnaire (handler) de rejet, appelé lorsqu'une tâche soumise ne peut être ajoutée à la file de tâches
- Ces éléments peuvent être fixés par des paramètres passés au constructeur de `ThreadPoolExecutor`.
  - La file des tâches du groupe ne peut être fixée que par le constructeur car elle doit être disponible dès la création ; l'instruction suivante crée un groupe avec un nombre fixe de *threads* (`N`) et une file d'attente à taille fixe (`CAPACITY`) :

Personnalisation des instances de `ThreadPoolExecutor` II

```
ThreadPoolExecutor executor =
  new ThreadPoolExecutor(N, N,  0L, TimeUnit.MILLISECONDS,
                new ArrayBlockingQueue<Runnable>(CAPACITY));
```

- Pour les autres caractéristiques, une approche plus simple est d'utiliser les méthodes `set` de `ThreadPoolExecutor` pour les modifier après création (avant la première soumission de tâches).
  - C'est le cas pour le gestionnaire de rejet qui peut être fourni après coup avec `setRejectedExecutionHandler`.
  - L'instruction suivante affecte au groupe la politique de rejet selon laquelle la tâche soumise sera exécutée mais par le *thread* appelant plutôt qu'un *thread* du groupe :

```
executor.setRejectedExecutionHandler(
                    new ThreadPoolExecutor.CallerRunsPolicy());
```

Politiques de gestion de la charge

- Politiques de rejet : appelées lors de la soumission d'une tâche et que la mise en file d'attente ne peut se faire ; elles sont définies par des classes internes à `ThreadPoolExecutor`.
  - `AbortPolicy` : rejet avec `RejectedExecutionException`.
  - `CallerRunsPolicy` : exécution par le *thread* appelant.
  - `DiscardOldestPolicy` : rejet de la plus ancienne dans la file et resoumission de la tâche courante.
  - `DiscardPolicy` : rejet silencieux.
  - Il est possible de créer de nouvelles politiques par une classe implantant `RejectedExecutionHandler`.
- Gestion par attente : l'utilisation d'autres types de files d'attente permet de gérer la charge sans rejet mais par attente.
  - `LinkedTransferQueue` : les ajouts et retraits attendent si la file a une capacité limitée et qu'elle est respectivement pleine ou vide.
  - `SynchronousQueue` : chaque insertion doit attendre un retrait et chaque retrait doit attendre une insertion (attention, cas très particulier, car sans stockage : ce n'est que la gestion de rendez-vous entre *threads* insérant et retirant).

Gestion de la charge et performance

- Obtenir la meilleure performance possible tout en gérant la charge est un problème de configuration délicat qui n'a pas de solution unique, mais des compromis adaptés à chaque situation.
  - Une file de taille importante couplée à un groupe avec peu de *threads* produit généralement une meilleure utilisation des ressources mais au prix d'un débit (*throughput*) plus faible.
- En BCM, grâce à l'abstraction du parallélisme et de la concurrence offerte par les composants, cette configuration passera par la configuration des composants.
  - Elle se fait par des paramètres formels, définis lors de la conception, qui vont exposer les options possibles pour la création des groupes de *threads* et la soumission des requêtes et des tâches.
  - Elle doit alors être parfaitement documentée de la manière la plus précise possible pour que les utilisateurs doivent pouvoir traduire leurs besoins en une configuration personnalisée efficace.

Personnalisation des groupes de *threads* en BCM

- Par défaut, BCM crée ses groupes de *threads* avec des choix standards ; toutefois, en passant par une création avec fabrique, il devient possible d'en exploiter toutes les possibilités.
- Pour forcer un composant BCM à créer un groupe à partir d'une classe donnée étendant `ThreadPoolExecutor`, `AbstractComponent` propose une méthode de création dédiée :

```
int createNewExecutorService(String uri, int nbThreads,
                             boolean schedulable, ExecutorServiceFactory f)
```

  - Similaire à celle déjà vue, elle prend en plus une fabrique qui implante l'interface fonctionnelle `AbstractComponent.ExecutorServiceFactory`.
  - La fabrique définit la méthode `createExecutorService` prenant en paramètre un nombre de *threads* et retournant l'instance personnalisée de `ThreadPoolExecutor`, ce qui peut se faire par une λ-expression.

Plan

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre les menaces et les freins potentiels à la sûreté de concurrence et à la performance.
   - Comprendre les compromis essentiels entre sûreté de concurrence et recherche de performance.
   - Comprendre les limites effectives à la parallélisation des programmes et au gain de performance espéré de l'augmentation du nombre de *threads*.
   - Comprendre les principales mesures de performance et leur caractère potentiellement ou effectivement contradictoires.
2. Compétences à acquérir
   - Savoir utiliser les conseils donnés dans la recherche d'une plus grande performance des applications BCM.
   - Connaître et savoir appliquer les principaux types de tests de programmes parallèles et concurrents.
   - Connaître et savoir éviter les principales sources de biais dans les tests et les mesures de performance.

Parallélisabilité des applications

- La finalité de l'introduction du parallélisme dans les applications est l'augmentation globale de la performance.
- Toutefois, le passage au *multithreading* a toujours un prix :
  - changements de contexte : déchargement/rechargement des processeurs avec les données des *threads* ;
  - synchronisation des accès mémoire : invalidation/vidage/remplissage des caches et barrières de synchronisation (`volatile`) ;
  - blocages et suspensions dûs aux synchronisations.
- Et toutes les applications ont des points de sérialisation forcée.
  - Dans les *pools* de *threads*, le retrait des tâches de la file d'attente ; la gestion du retour des résultats aussi, le plus souvent.
- La recherche de performance est donc nécessaire, mais toute amélioration apportée à des programmes parallèles est aussi :
  - une source de complexité supplémentaire (et de *bugs*)...
  - et une menace à la sûreté de concurrence.

Angle d'attaque sur la performance

- La principale cause de réduction de performance des programmes parallèles est le recours à l'exclusion mutuelle.
- Pour réduire l'impact de l'exclusion mutuelle, on a vu :
  - la réduction de la durée de détention des verrous ;
  - la réduction de la fréquence d'acquisition des verrous ;
  - la réduction de la portée des verrous et l'introduction de nouveaux verrous plus spécifiques (partition des invariants, ...) ;
  - le remplacement de l'exclusion mutuelle stricte par des mécanismes autorisant plus de parallélisme (classes concurrentes de Java, *read/write locks*, etc.).
- Impact visé : amélioration des principales mesures de performances (potentiellement contradictoires).
  - Durée : avec quelle rapidité un appel de service est-il servi ?
  - Débit (*throughput*) : quelle quantité d'appels peut-on traiter par unité de temps ?
  - Passage à l'échelle : quel gain proportionnel est obtenu par l'ajout de *threads* ?

Test efficace de programmes concurrents

- Propriétés recherchées :
  - Sûreté (*safety*) : rien de mauvais ne peut arriver.
  - Vivacité (*liveness*) : quelque chose de bon arrivera.
- Types de tests :
  1. Exactitude : absence d'erreurs de programmation, en se basant sur les spécifications pour produire les cas de tests.
  2. Blocages : se produisent-ils quand il le faut et uniquement là ?
  3. Sûreté : accès cohérents aux données protégées.
  4. Performance : mesures de réactivité/débit/passage à l'échelle.
- Deux difficultés induites par l'instrumentation du code pour produire des tests et des mesures de performance :
  1. Introduction de biais dans l'exécution et les mesures du fait de la présence du code d'instrumentation.
  2. L'exploration du plus grand nombre d'entrelacements possibles :
     - `Thread.yield()` : autorise l'ordonnanceur à passer la main à un autre *thread* (mais peut être ignoré par l'implantation).
     - `Thread.sleep(1L)` : sans trop perturber les mesures, amène plus sûrement l'ordonnanceur à passer à un autre *thread* si possible.

Biais dans les mesures de performance : conseils

- Sources de biais internes au programme et son instrumentation :
  - Imprécision de l'horloge : mesurer sur plusieurs répétitions.
  - Synchronisations par inadvertance (entrées/sorties, collections synchronisés, etc.) perturbant l'ordre d'exécution des *threads* : éviter les E/S et autant que possible le partage de données dans le code d'instrumentation exécuté au sein du code mesuré.
- Sources de biais dans la JVM et externes :
  - Autres programmes utilisant le même CPU.
    - Arrêter tous les autres programmes (autant que possible).
  - Gestion de la mémoire (GC) :
    - soit allouer suffisamment de mémoire pour l'éviter,
    - soit faire des mesures sur des durées suffisamment longues pour intégrer dans la mesure le coût moyen de GC.
  - Compilation dynamique : Java compile à la volée le *bytecode* JVM en langage machine, ce qui impacte la performance.
    - Comme Java utilise des caches de code compilé, il y a un phénomène de « chauffe » initiale et de taille de cache : commencer les mesures après avoir appelé toutes les méthodes à mesurer et augmenter la taille de la mémoire disponible pour la JVM.

Extension de `ThreadPoolExecutor` pour statistiques

- Java permet de créer des sous-classes de `ThreadPoolExecutor` pour adapter son comportement (cf. fin du cours 4).
- Cette possibilité donne un bon point d'entrée pour introduire des mesures de performance au niveau d'un *pool* de *threads*.
- Trois méthodes sont utiles à redéfinir dans ce cas :
  1. `beforeExecute` : est appelée au début de chaque exécution de tâche par un *thread* du *pool*.
  2. `afterExecute` : est appelée à la fin de chaque exécution de tâche par un *thread* du *pool*.
  3. `terminated` : est appelée lorsque le *pool* a terminé son exécution.
- Rappel : un composant BCM peut créer un *pool* à partir d'une classe donnée étendant `ThreadPoolExecutor` grâce à la méthode

```
int createNewExecutorService(String uri, int nbThreads,
                             boolean schedulable, ExecutorServiceFactory f)
```

prenant en dernier paramètre une fabrique qui implante l'interface `AbstractComponent.ExecutorServiceFactory`.

TimingThreadPool (adapté de Goetz et al., *op. cit.*, p. 180)

```
public class TimingThreadPool extends ThreadPoolExecutor {
  private final ThreadLocal<Long> startTime = new ThreadLocal<Long>(); // Un par thread.
  private final AtomicLong numTasks = new AtomicLong();
  private final AtomicLong totalTime = new AtomicLong();

  @Override
  protected void beforeExecute(Thread t, Runnable r) {
    super.beforeExecute(t, r);         // fait le travail standard d'un pool de thread.
    startTime.set(System.nanoTime()); // nanotime() n'a pas nécessairement une précision à la nanoseconde,
                                      // dépend de l'horloge matérielle et du système d'exploitation.
  }

  @Override
  protected void afterExecute(Runnable r, Throwable t) {
    try {
      long taskTime = System.nanoTime() - startTime.get();
      totalTime.addAndGet(tasktime);   // danger: synchronisations supplémentaires
      numTasks.incrementAndGet();      // mais après la mesure de fin des tâches !
    } finally {
      super.afterExecute(r, t);        // fait le travail standard d'un pool de thread.
    }
  }

  @Override
  protected void terminated() {
    try {      // À la fin, on calcule les statistiques.
      System.out.println(String.format("Thread %s: avg time=%dns", totalTime.get()/numTasks.get());
    } finally {
      super.terminated();
    }
  }
}
```

Plan

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre quelques unes des sources de difficultés dans la construction de systèmes répartis à grande échelle.
   - Comprendre les premières conséquences du passage à l'échelle dans les systèmes répartis.
   - Comprendre les raisons et les motivations du passage à de nouvelles architectures pour les systèmes répartis à grande échelle, comme les architectures pair-à-pair.
2. Compétences à acquérir
   - Savoir reconnaître dans un systèmes réparti les fonctions/nœuds/composants pouvant devenir des goulots d'étranglement lors du passage à l'échelle.
   - Savoir utiliser quelques solutions pour implanter des fonctions de systèmes répartis et les rendre robustes à l'utilisation d'informations incomplètes et relativement périmées.

Apparition de goulots d'étranglement

- Dans beaucoup de systèmes répartis, on impose à l'ensemble des nœuds d'utiliser certains nœuds particuliers, centralisant des fonctions partagées par l'ensemble de l'application.
  - Répartiteurs de tâches, outils de communication, gestionnaires de ressources, indexeurs d'informations, etc.
- Lors du passage à l'échelle, la croissance du nombre de nœuds de calcul augmente la charge de nœuds dédiés aux fonctions centralisées souvent bien plus que linéairement, parfois même exponentiellement.
  - La performance des nœuds dédiés étant physiquement limitée, il arrivera nécessairement une forte dégradation de la qualité de service offerte à chacun des autres nœuds.
- On parle alors de *goulots d'étranglement*, c'est-à-dire certains nœuds et la fonction partagée qu'ils réalisent dont la performance *bornée* va limiter la performance globale de l'application, même si on continue à ajouter des nœuds de calcul.

Nécessité d'une répartition à grande échelle

- Fondé sur le rejet des goulots d'étranglement, le passage à l'échelle nécessite des solutions à *décentralisation forte*.
- Cela peut mener à la répartition des fonctions partagées sur plusieurs des nœuds, allant jusqu'à des solutions totalement *pair-à-pair*.
  - Par exemple, une plate-forme de diffusion de contenu (musique) peut être construite autour d'un catalogue centralisé indiquant où trouver le contenu (fichier MP3) parmi un ensemble de serveurs.
  - Ceci implique que toutes les requêtes pour trouver du contenu passent par la fonction catalogue qui peut devenir un goulot d'étranglement.
  - Les premières applications grand public du pair-à-pair visaient à répartir le catalogue et les contenus sur l'ensemble des nœuds puis rechercher une entrée par échanges pair-à-pair entre les nœuds afin d'identifier un nœud permettant de récupérer le (ou une copie du) contenu.

Travailler avec de l'information incertaine et incomplète

- Les réseaux actuels impliquent une croissance des délais de transmission des informations en fonction de la taille du réseau.
  - Même à la vitesse de la lumière, l'éloignement physique des équipements imposent des délais significatifs par rapport à la vitesse de calcul des processeurs.
- Plus les délais de transmission deviennent importants par rapport à la vitesse des calculs, plus ces calculs doivent être robustes à l'utilisation d'informations immédiatement disponibles localement, sans attendre des données distantes fraîches.
  - Exemple : un répartiteur de tâches essaie d'affecter les nouvelles tâches aux nœuds (répartis) les moins chargés, mais dans un système à grande échelle, il sera impossible pour ce répartiteur de connaître la charge exacte de tous les nœuds à un instant précis.
- Plus généralement, la diffusion d'informations dans un système réparti à grande échelle implique qu'aucun nœud ne puisse à aucun moment avoir une image parfaitement à jour de l'ensemble des informations *produites par* ou *sur* le système.

Des pannes devenant une norme déstabilisante

- Pannes indépendantes et exécution permanente impliquent l'*évolution dynamique* de l'architecture du système soit parce que des nœuds tombent en panne ou quittent le système et que d'autres arrivent (en retour de panne ou s'y connectant).
- L'apparition et la disparition de nœuds imposent d'avoir de la *redondance* dans les rôles et dans le stockage des informations de manière à pouvoir les utiliser en tout temps.
- Elles génèrent un besoin d'*auto-adaptation dynamique* pour le système, par exemple par l'attribution et la réattribution des rôles (répartition des tâches, élection de nœuds maîtres, etc.).
- Ces perturbations entraînent une *instabilité* pendant laquelle les invariants locaux et globaux du système ne sont plus respectés.
- On recherche donc des algorithmes d'auto-adaptation qui vont garantir la (re)stabilisation à terme du système, ce qu'on appelle l'*auto-stabilisation* et des algorithmes *auto-stabilisants*.

Plan

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre les difficultés et les implications du partage de données dans les systèmes répartis à grande échelle.
   - Comprendre comment la réplication des données peut augmenter la performance des systèmes répartis, mais au prix d'un relâchement des contraintes de cohérences entre les copies.
   - Comprendre les principaux modèles de cohérence existant et leurs limitations par rapport aux types de systèmes répartis et à leur échelle..
2. Compétences à acquérir
   - Connaître quelques outils de partage de données avec réplication et cohérence.
   - Savoir utiliser des techniques de collecte asynchrone de données pour les systèmes répartis, y compris à grande échelle, et leur implantation-type en BCM.

État dans un système réparti

- En programmation répartie, *a fortiori* à grande échelle, établir quel est l'état courant du système devient un problème qui surprend bien des néophytes.
- Ex.: considérons un répartiteurs de tâches qui souhaite envoyer chaque tâche au replica dont la file de tâches à exécuter est actuellement la plus courte.
  - Supposons que le répartiteur interroge chaque serveur pour connaître la longueur de sa file d'attente.
  - Un appel à travers le réseau peut prendre quelques dizaines de millisecondes.
  - Interroger 100 serveurs *séquentiellement* peut prendre plusieurs secondes ; même en parallèle, le *délai* pour obtenir l'information demeure *long* par rapport à l'urgence de la décision.
  - *Et le temps de la synthétiser, elle ne sera déjà plus à jour...*
- Il est donc à la fois *contre-performant* et *illusoire* de penser pouvoir programmer un système réparti avec les mêmes principes qu'un système séquentiel centralisé.

Partage de données à grande échelle et cohérence

- Partager des données grâce à des dépôts *centralisés* est une option, mais qui peut vite devenir un goulot d'étranglement.
- Au passage à l'échelle, on opte pour des approches *décentralisées*, où les données sont *répliquées* et maintenues *cohérentes automatiquement*.
  - Répliquées : copies maintenues sur plusieurs nœuds pour améliorer la performance et augmenter le tolérance aux pannes.
  - Cohérence : lorsqu'une modification est faite sur une copie d'une données, elle doit être propagée aux autres copies.
- En pure abstraction fonctionnelle, un dépôt offre deux opérations, lecture et écriture, qui seront strictement cohérentes si toute lecture retourne toujours la dernière valeur écrite.
- Pour passer à l'échelle, il faut relâcher la sémantique de réplication en termes de temporalité des lectures par rapport écritures :
  - plus le nombre de nœuds augmente moins la cohérence peut être forte si on veut maintenir une bonne performance.

Cohérence : principaux modèles (sémantiques) I

Source Wikipédia, « Cohérence (données) », définition accédée le 19/03/2020.

- Modèles faisant abstraction de la synchronisation (vue « idéale » des données) :
  - Stricte (atomique) : une lecture renvoie toujours la dernière valeur écrite, peu importe où ces écritures se font.
  - Forte : à tout instant, toutes les copies d'une même donnée sont identiques (très coûteux).
  - Séquentielle : le résultat de toute exécution est équivalent à celui d'*une* exécution séquentialisée de tous les processus.
  - Immédiate : l'opération d'écriture est finalisée seulement lorsque tous les processus ont été synchronisés pour voir la nouvelle valeur au même moment.
  - Causale : affaiblissement de la cohérence séquentielle ne considérant que les lectures et écritures en *relation causale*.
  - À terme (*eventual consistency*) : toute écriture faite par un processus est inéluctablement vue par les autres et lorsque les écritures s'arrêtent, tous les processus voient la même valeur.

Cohérence : principaux modèles (sémantiques) II

Ce modèle permet les écritures concurrentes, mais doit résoudre les conflits :

1. Réconciliation à la lecture : résolution lors d'une lecture constatant un conflit entre copies (ralenti la lecture).
2. Réconciliation à l'écriture : résolution lors d'une écriture constatant un conflit entre copies (ralenti l'écriture).
3. Réconciliation asynchrone : faite par un processus interne réconciliant les copies lorsqu'il constate un conflit.

- Modèles avec synchronisation (force la cohérence lorsque nécessaire, par des synchronisations souvent de type barrière) :
  - Faible : pas de contraintes sur les opérations non synchronisées mais pour les autres, toutes les écritures précédant la synchronisation doivent avoir été propagées à tous avant de procéder.
  - Delta : à intervalle régulier, l'ensemble des processus sont synchronisés et les opérations propagées avant de procéder ; la durée pendant laquelle des incohérences peuvent se produire est partiellement maîtrisable car elle dépend de l'intervalle choisi et du moment où l'écriture est faite pendant cet intervalle.

Mécanismes et options pour assurer la cohérence

- Réplication des données individuelles :
  - Maître unique : une copie en écriture, les autres en lecture seule.
  - Maîtres multiples : jusqu'à toutes les copies en lecture/écriture.
- Gestion des réplicas :
  - Totale : tous les nœuds ont leur copie de toutes les données.
  - Partielle : une partie des données sur chaque nœud.
- Propagation des modifications :
  - Avide (*eager*) : immédiatement après l'écriture.
  - Paresseuse (*lazy*) : en parallèle avec l'exécution des processus. Peut produire des conflits d'écriture à réconcilier avec une estampille de temps ou en fonction de la sémantique de la donnée partagée et des actions faites, ...
- Mode de communication :
  - classique, point-à-point : diffusées aux nœuds directement.
  - pair-à-pair : diffusées aux voisins, qui rediffusent aux leurs, etc.

Outils de partage de données réparties

- Les développeurs n'en sont que des utilisateurs :
  - Ils évitent de développer eux-mêmes des solutions de partage de leurs données propres, sauf quand la performance l'exige.
  - Ils laissent cela aux spécialistes, autant que possible.
- Bien que la recherche dans le domaine soit encore active, les premiers travaux remontent à plus de 30 ans.
- Il existe ainsi sur le marché plusieurs outils matures proposés par des grands éditeurs ; ces outils se partageant entre :
  - les outils persistents, s'appuyant sur des bases de données et
  - les outils non persistents où les données sont conservées en mémoire sur les nœuds.
- Il existe des normes spécifiant des services de distribution de données comme le *Data Distribution Service* (DDS) de l'OMG et même une déclinaison pour la programmation temps réel.

Inversion de contrôle, asynchronisme et conséquences

- Le maintien de la cohérence de données partagées répliquées requiert de plus en plus de ressources et impose des délais de plus en plus grands quand le système croît en taille.
- Pour dépasser ces limites, il faut passer à l'échange asynchrone de données pour synthétiser localement un état du système ou échanger des données et résultats des calculs.
  - On inverse le contrôle en demandant aux entités (par ex., serveurs) d'envoyer leur état au nœud synthétiseur (répartiteur) de manière asynchrone par rapport au traitement que ce dernier réalise (répartion des tâches).
  - Le nœud synthétiseur/requérant met à jour sa vue locale et prend ses décisions par rapport à cet état local, sans attendre.
- L'état synthétisé (longueurs des files d'attente) ne sera pas nécessairement à jour ; l'idée est qu'il le soit « suffisamment » pour les besoins de l'application (répartir la charge pour obtenir le meilleur débit possible).

Exemple de collecte asynchrone : répartiteur

- Exemple des cours précédents mais avec un répartiteur qui applique la politique de la plus courte file d'attente et ce, en se basant sur une remontée asynchrone des données.
- `StateBasedDispatcher` implante le composant répartiteur.
  - La méthode `acceptNewServerState` reçoit les mises à jour des serveurs et recalcule le serveur ayant la file la plus courte selon les données reçues.
  - La méthode `call` envoie la requête à ce serveur.
  - Pour permettre de comparer, la classe permet aussi une exécution avec la politique de répartition *round-robin*.
- `MonitoredServer` utilise la personnalisation du *pool* de *threads* pour exécuter les requêtes de telle manière à accéder à la taille de la file de tâches.
  - La méthode `call` exécute simplement la requête.
  - La méthode `notifyState` est appelée répétitivement pour envoyer au répartiteur la taille courante de la file d'attente de ce *pool*.
- Un document fourni explique l'exemple dans le détail.

Plan

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre le besoin d'une interconnexion logique des nœuds d'une application répartie.
   - Comprendre la distinction entre nœud logique et ordinateur physique d'une part et réseau logique et réseau physique d'autre part.
   - Comprendre l'organisation d'un réseau logique et ses principales fonctions de communication.
2. Compétences à acquérir
   - Savoir créer un réseau logique simple.
   - Savoir utiliser la communication dans un réseau logique.
   - Approfondir ses compétences par l'examen d'un premier exemple complet : un anneau logique.

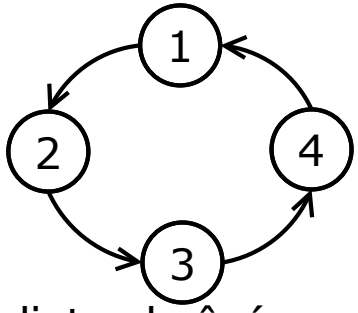
Algorithmique répartie pair-à-pair et réseau

- L'un des fondements des systèmes répartis à grande échelle est l'algorithmique pair-à-pair.
- Comme nous l'avons vu, il s'agit de proposer des algorithmes et des implantations pour les fonctions qui :
  - sont réparties entre (tous) les nœuds,
  - où tous les nœuds jouent le même rôle *i.e.*, sont des pairs équivalents.
- Pour déployer ces algorithmes le premier besoin est de construire un réseau reliant tous les composants pour leur permettre de s'échanger en pair-à-pair des messages les uns avec les autres.
  - Les systèmes répartis utilisent des réseaux physiques (ethernet, internet) qui connectent les ordinateurs entre eux.
  - Ce qui est recherché ici est une notion de réseau abstrait entre les nœuds/composants, qui sera au réseau physique ce qu'un processus est au processeur physique.

Notion de réseau logique (*overlay network*)

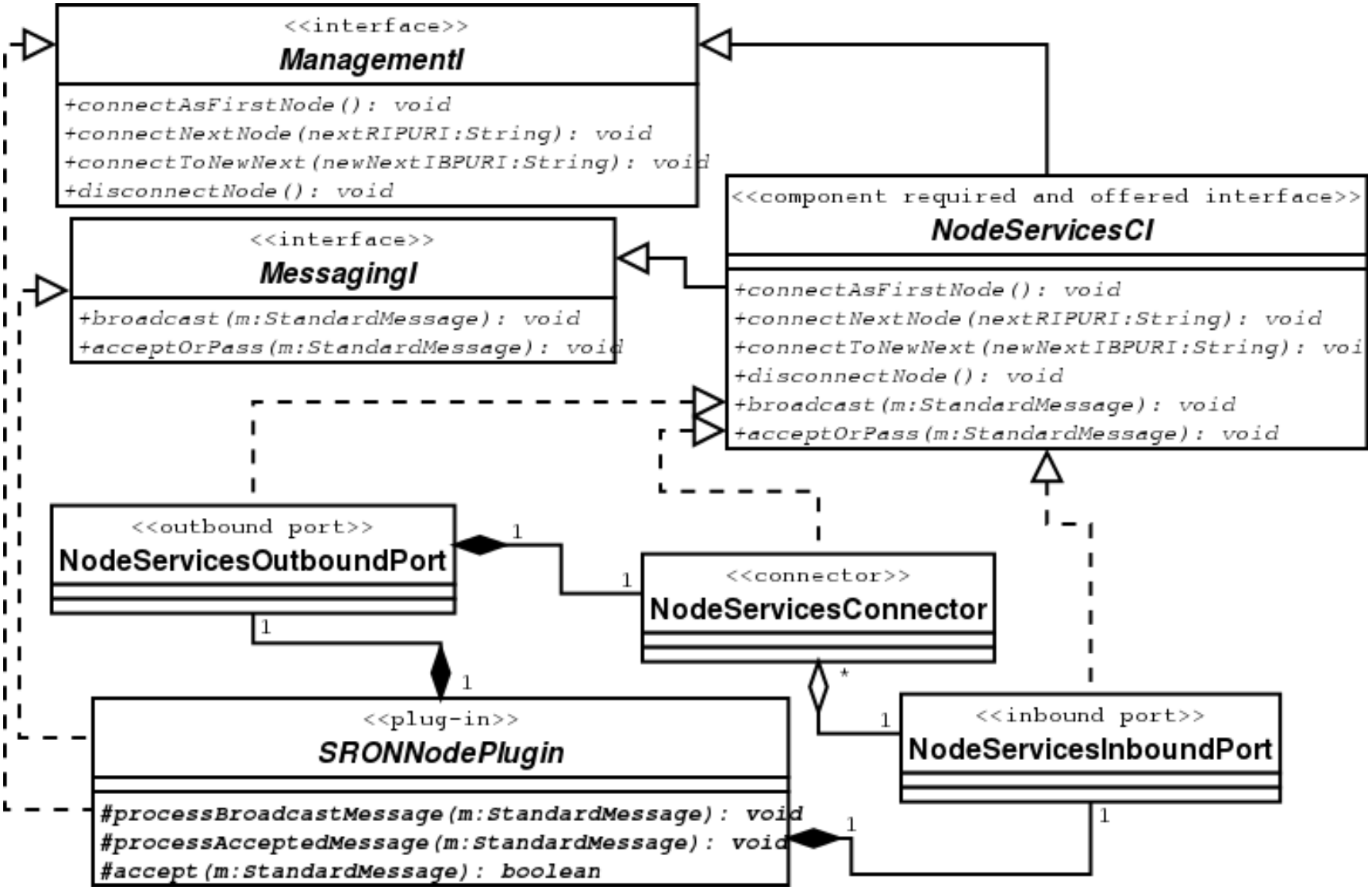
- Un *réseau logique* interconnecte les nœuds logiques en un graphe leur permettant de communiquer entre eux.
- Pour passer à l'échelle et être totalement dynamique, la communication dans un réseau logique se fait d'abord par *diffusion* : un message n'est pas routé en fonction de son destinataire mais diffusé sur l'ensemble du réseau jusqu'à ce que le destinataire le reçoive.
- Le réseau logique échange les messages tantôt par mémoire physique partagée (pour les nœuds colocalisés), tantôt par le réseau physique.
- Que ce soit en diffusion ou en point-à-point, les messages sont physiquement échangés uniquement entre nœuds *adjacents* ; pour arriver à destination, ils sont propagés dans le réseau de nœuds en nœuds.

Un premier réseau logique : l'anneau logique



- Le réseau logique le plus simple est l'anneau :
  - les nœuds sont liés entre eux sous la forme d'une liste chaînée dont le dernier nœud est lié au premier pour fermer l'anneau.
- La communication dans un anneau est très simple :
  - un message peut entrer par n'importe quel nœud ;
  - pour une diffusion, le message passe de nœud en nœud jusqu'à ce qu'il revienne au nœud de départ où il est éliminé ;
  - en « point-à-point », le message est diffusé jusqu'à ce qu'un nœud l'accepte et alors la diffusion s'arrête immédiatement.
- La construction du réseau logique se fait de la manière suivante :
  - il débute par un nœud unique,
  - puis chaque nœud peut recevoir une requête pour ajouter un nœud le suivant immédiatement dans l'anneau.

Conception d'un greffon anneau logique simple pour BCM



Présentation de l'exemple

- Fourni dans l'archive sous `SimpleRingOverlayNetwork-CM9`.
- La classe `SRONNodePlugin` implante le greffon qui permet à un composant d'adopter les fonctionnalités d'un nœud dans un réseau logique sous la forme d'un anneau.
  - Elle permet l'ajout dynamique de nœuds dans l'anneau et utilise un verrou de type `ReentrantReadWriteLock` pour gérer la concurrence entre la transmission des messages et la modification de l'anneau.
  - Elle utilise la classe `StandardMessage` qui implante un message standard échangé sur le réseau logique.
- Un petit exemple est fourni, avec une classe `DiffusionNode` qui implante un composant très simple à mettre en réseau logique, et la classe `DynamicAssembler` qui crée un anneau de 5 composants puis diffuse un message (à tous les nœuds donc) et ensuite envoie un message destiné au nœud 3.

Échange de messages sur l'anneau logique

- Diffusion d'un message m :
  - le message est d'abord reçu par un nœud via la méthode `broadcast` qui le marque de son identité (un message ne peut être marqué qu'une fois) ;
  - ce nœud le traite par la méthode `processBroadcastMessage` puis le diffuse au nœud suivant par la méthode `broadcast` et ce dernier recommence ;
  - lorsque le message revient au nœud de départ, ce dernier reconnaît l'avoir marqué et arrête la diffusion.
- Envoi d'un message m en « point-à-point » :
  - on procède comme en diffusion sauf que m est reçu et passé par la méthode `acceptOrPass` ;
  - sur réception de m, chaque nœud appelle sa méthode `accept` qui retourne vrai si le nœud accepte le message et alors il le traite par la méthode `processAcceptedMessage`, sinon il est envoyé au nœud suivant par `acceptOrPass`.

Plan

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre la notion d'algorithme réparti pair-à-pair.
   - Comprendre les principales hypothèses sous-jacentes aux algorithmes répartis pair-à-pair et leurs conséquences sur les limites de ces algorithmes.
2. Compétences à acquérir
   - Apprendre la programmation d'algorithmes répartis pair-à-pair sur une application structurée autour de nœuds organisés en réseau logique.
   - Approfondir ces compétences par l'examen d'un premier exemple complet : l'algorithme d'élection sur un anneau logique de Chang et Roberts.

Introduction à l'algorithmique répartie

- L'algorithmique répartie classique propose de nombreux algorithmes pair-à-pair.
  - Exemple : élection d'un chef parmi les nœuds.
  - Autres algorithmes : création de réseau logique, propagation d'informations par vagues, détection de terminaison, détection de pannes, synchronisation des horloges locales, etc.
- Ces algorithmes sont généralement fondés sur l'existence de liens de communication entre les nœuds (le réseau logique) et sur des protocoles anonymes d'échanges de messages.
- Ils s'appuient cependant la plupart du temps sur l'une ou l'autre des hypothèses plus ou moins fortes suivantes :
  - absence de pannes de liens et/ou de nœuds,
  - topologie statique pendant leur exécution,
  - délai de transmission point-à-point borné,
  - attribution d'identifiants uniques aux nœuds,
  - capacité à découvrir (forger) l'adresse des nœuds pour entrer en contact et communiquer avec eux, etc.

Algorithme d'élection de Chang et Roberts (1979) I

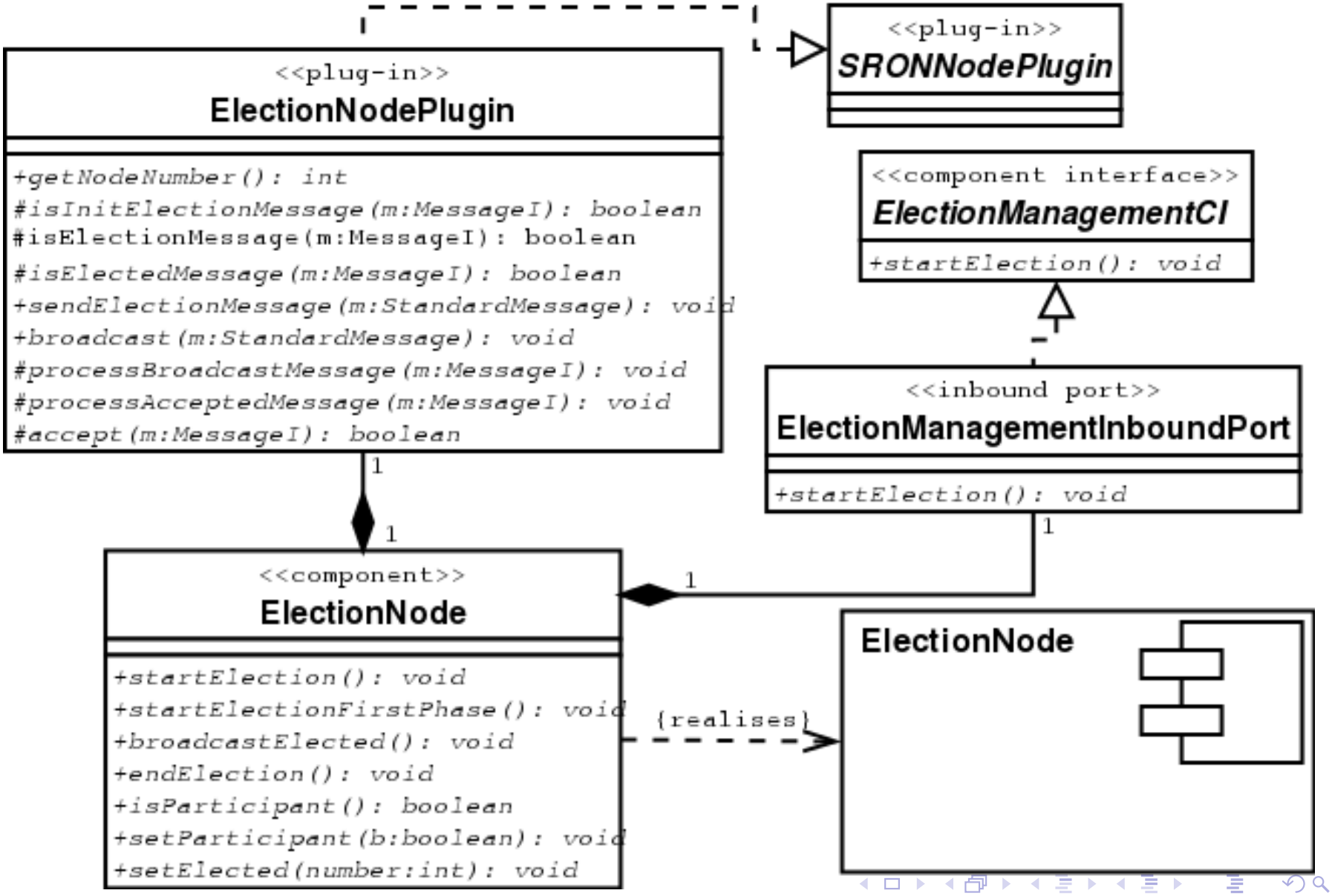
- Source : Wikipedia, *Chang and Roberts algorithm*, 12/01/2015.
- Objectif : élire un maître dans un *anneau* (réseau logique).
- Hypothèses de départ :
  1. tous les nœuds $i, 1 \leq i \leq N$ disposent d'un identifiant unique $u_i \in UID$ muni d'une relation d'ordre ($\prec$), ce qui permet d'élire le nœud ayant le plus grand identifiant parmi les présents, et
  2. les nœuds sont en mesure de s'organiser en anneau de communication unidirectionnel allant de chaque nœud à son voisin dans le sens de l'horloge.
- Algorithme en deux étapes, la première étape est :
  1. Initialement, chaque nœud est marqué *non-participant*.
  2. Observant l'absence de coordinateur (panne), un/des nœud/s lance/nt l'élection en créant (chacun) un message d'élection contenant son UID et en l'envoyant à son voisin.
  3. À chaque fois qu'un nœud envoie ou retransmet un message d'élection, il se marque comme *participant* à cette élection.

Algorithme d'élection de Chang et Roberts (1979) II

4. Lorsqu'un nœud i reçoit un message d'élection m, il compare son UID u_i à l'UID présent dans le message u_m et :
   - si $u_i \prec u_m$, i retransmets inconditionnellement m à son voisin ;
   - si $u_m \prec u_i$ et i marqué *non-participant*, i remplace u_m par u_i et envoie ce message modifié à son voisin ;
   - si $u_m \prec u_i$ et i marqué *participant*, i détruit simplement le message ;
   - si $u_m = u_i$, i se déclare *élu* et débute la seconde étape.

- Deuxième étape :
  1. L'élu se marque comme non-participant et envoie le message *élu* avec son UID à son voisin.
  2. Lorsqu'un nœud reçoit un message *élu*, il se marque comme *non-participant*, mémorise l'UID de l'élu et retransmet le message *élu* à son voisin.
  3. Lorsque le message *élu* atteint l'élu, l'élu détruit le message et l'élection est terminée.
- L'algorithme se termine pour tout N (s'il n'y a pas de fautes) ; il est sûr, vivace et prend $\mathcal{O}(3N-1)$ messages quand il y a un seul initiateur.

Conception d'un greffon élection pour BCM



Présentation de l'exemple

- L'exemple est fourni dans la même archive que celle contenant le greffon réseau logique en anneau.
- Il reprend essentiellement la même architecture que l'exemple précédent, la classe `ElectionNodePlugin` définissant le comportement d'élection et `ElectionNode` implantant le composant mis en réseau logique et utilisant la fonctionnalité d'élection.
- L'interface de composant `ElectionManagementCI` et sa méthode `startElection` permettent à un composant tiers de lancer une ékection à partir d'un nœud quelconque du réseau logique. Elle est appelée par la méthode `execute` du composant `Dynamic-Assembler` réalisant un déploiement dynamique d'un réseau logique à 5 nœuds.

Avantages et inconvénients

- Le principal avantage des algorithmes répartis classiques réside dans leurs propriétés (complexité, vivacité, tolérance aux fautes, auto-organisation, auto-stabilisation, etc.) étudiées et démontrées formellement.
- Ils ont souvent des implantations largement diffusées dont on peut s'inspirer.
- Leur principal inconvénient réside dans leur durée d'exécution souvent longue et surtout non-bornable.
- Cet inconvénient est exacerbé lorsqu'on augmente le nombre d'entités en passant à l'échelle.
- Assez souvent, les hypothèses de départ des algorithmes sont difficiles à assurer en pratique.

  Pour l'algorithme de Chang et Roberts, l'absence de fautes est une hypothèse bien restrictive. De manière générale, les démonstrations de propriétés utiles en pratique (comme la complexité) pour les systèmes où les nœuds *et* les liens peuvent tomber en panne sont malheureusement rares.

Activités à réaliser avant le prochain TME

1. Lire le code des exemples fournis.