GossiPeer : vers l'uniformisation du développement d'algorithmes épidémiques

Vincent Gramoli[1 2], Erwan Le Merrer[3], Anne-Marie Kermarrec[3]

[1] *EPFL, Bât. INR, Station 14, CH-1015 Lausanne, Suisse.* vincent.gramoli@epfl.ch
[2] *Université de Neuchâtel, rue Emile-Argand 11, 2009 Neuchâtel, Suisse.*
[3] *INRIA-Rennes Bretagne Atlantique, Campus de Beaulieu 35042 Rennes, France.* {akermar,elemerre}@irisa.fr

Les méthodes ou algorithmes intégrant une communication épidémique (« gossip-based ») ont connu un développement majeur au cours de la dernière décennie, portés par leur potentiel de tolérance aux pannes et de passage à l'échelle. Alors que ces approches sont massivement étudiées, il n'existe pas d'outil modulaire permettant la réutilisation de code dans ces études. Nous présentons l'intergiciel GossiPeer, en motivant les modules majeurs de cet intergiciel par l'identification des comportements de base des algorithmes épidémiques.

Keywords: algorithmes épidémiques, pair-à-pair, intergiciel, java.

1 Introduction

1.1 Motivations

L'étude pratique des systémes à grande échelle constitue l'un des défis majeurs actuels. En effet, l'utilisation de plateformes d'exécution, telles que PlanetLab, EmuLab et Grid5000, connaît un succès grandissant. Parmi les applications distribuées exécutées sur de telles plateformes, certaines d'entre elles partagent de nombreuses similarités. Malgré ces similarités, leur développement reste toujours une tâche difficile.

Un paradigme de communication a été récemment adopté comme une dominante de recherche à part entière en systèmes distribués. Ce paradigme, souvent appelé *épidémique* (aussi appelé « gossip-based ») est caractérisé par des échanges de messages périodique entre les nœuds du système. Alors que ce paradigme a été observé dans des disciplines variées dépassant largement le cadre des réseaux pair-à-pair [CGJ+07], il apparaît crucial de pouvoir évaluer expérimentalement les algorithmes utilsant un tel mécanisme sur un réseau à grande échelle.

Les seuls intergiciels existant à l'heure actuelle, comme JXTA†, ne sont pas dédiés à un type d'application distribuée particulier. Par conséquent, le développement d'une nouvelle application épidémique nécessite à chaque fois de réinventer la roue. En dépit du temps nécessaire, ce travail répétitif est souvent sujet à des erreurs, le rendant donc difficile.

1.2 Contributions

Cet article propose un intergiciel, appelé *GossiPeer*, pour simplifier le développement d'applications épidémiques. GossiPeer fournit des briques de base réutilisables lors du développement d'applications épidémiques. Afin de tirer partie du chargement dynamique (durant l'exécution) de nouvelles applications épidémiques, GossiPeer est développé en Java. De fait, GossiPeer se présente comme une boîte à outil permettant l'implémentation d'algorithmes épidémiques de façon modulaire.

La Section 2 identifie les comportements de base des algorithmes épidémiques afin de motiver nos choix dans les modules les plus importants de GossiPeer. Ces modules font l'objet de la Section 3. La Section 4 explique l'implémentation d'un algorithme épidémique en utilisant GossiPeer et la Section 5 conclut cet article.

† JXTA est disponible en ligne à l'adresse suivante : https ://jxta.dev.java.net/

2 Unification des algorithmes épidémiques

Les algorithmes épidémiques [DGH+87] ont démontré leur performance à de nombreuses reprises dans le contexte des systèmes dynamiques à grande échelle. Costa et al. [CGJ+07] ont identifié cinq particularités partagées par de nombreux algorithmes empruntés à divers domaines tels que la biologie, la génétique, ou l'auto-stabilisation, et qui peuvent êtres charactérisés comme épidémiques. Ces particularités permettent de donner une définition informelle d'un algorithme épidémique :

1. **aléat :** une sélection aléatoire de nœud(s) pour une intéraction ;
2. **localité :** une communication locale s'effectuant entre un nœud et un sous-ensemble d'autres nœuds ;
3. **périodicité :** une activité (tâche computationnelle et de communication) effectuée périodiquement ;
4. **légèreté :** une quantité limitée d'information transmise et utilisée par itération du protocole ;
5. **unicité :** une exécution du même algorithme par tous les participants.

Par définition, un algorithme épidémique a pour caractéristique des échanges de messages répétés entre chacun des nœud *i* et un sous-ensemble de nœuds, appelés les *contacts* du nœud *i*. D'une part, afin de décrire son exécution périodique, un algorithme épidémique doit posséder un comportement actif dans lequel une exécution permet d'envoyer de façon périodique des messages, et son pendant passif permettant de prendre en compte leurs effets sur les nœuds contactés. D'autre part, afin de définir les contacts d'un nœud, il est important d'identifier la manière avec laquelle les nœuds sont mis en contact. Par exemple, un nœud peut être en contact avec un sous-ensemble de nœuds qui évolue régulièrement afin que les nœuds fautifs soient retirés au fur et à mesure des contacts de tout nœud. A l'extrême inverse, les contacts d'un nœud peuvent être un sous-ensemble de nœuds qui n'évolue qu'en cas de panne, à l'instar des réseaux structurés.

3 Structure de GossiPeer

Chaque nœud participant exécute une instance de GossiPeer contenant les algorithmes épidémiques choisis. Le *nœud courant* fait référence, dans la suite de cet article, au nœud sur lequel s'exécute l'instance de GossiPeer décrite.

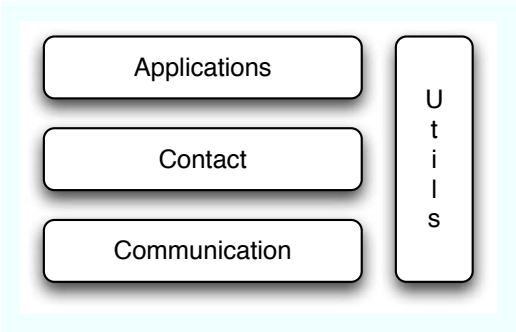

FIG. 1: Les modules de l'intergiciel GossiPeer.

Les propriétés énnoncées dans la Section 2 se divisent en deux catégories : *(i)* les propriétés liées à la computation (unicité et periodicité), *(ii)* celles liées à la communication (aléat, localité et légerté). Ces deux catégories correspondent aux deux modules *Applications* et *Contact* de GossiPeer décrits par les interfaces respectives `Applicable` et `Contactable`, et représentés sur la Figure 1.

Les deux autres modules représentés sur la figure sont entièrement implémentés par GossiPeer et sont complètement invisibles aux yeux du programmeur. Le module `Communication` regroupe l'ensemble des classes nécessaires à la gestion des communications réseaux et le module `Utils` contient les classes secondaires servant par exemple au chargement des informations de configuration ou à l'enregistrement des statistiques d'exécution.

3.1 Le module Applications

Le module *Applications* est le module de plus haut niveau d'abstraction. C'est à ce niveau que sont définis les protocoles épidémiques qui ne nécessitent pas de gestion de voisinage spécifique. De tels algorithmes, s'exécutent au moyen de deux procédures majeures : une procédure active s'exécutant périodiquement et une procédure passive déclenchée à la suite d'un événement extérieur (e.g. réception d'un message). C'est pourquoi nombre d'algorithmes épidémiques sont représentés en pseudocode au moyen de ces deux procédures [JKvS03, VGvS05, JVG+07, GVB+07]. GossiPeer décrit les spécificités que doit remplir une application épidémique, à l'aide de l'interface `Applicable` contenant ces méthodes et décrite Figure 2.

```
public interface Applicable {
  public abstract void activeExec();
  public abstract void passiveExec(Message msg, String sender, String hostname);
}
```

FIG. 2: Interface du module *Applications*.

Threads actif et passif. Ces deux types de comportement actif et passif sont exécutés en parallèle et doivent faire l'objet d'une exécution multi-threadée. Ainsi la procédure active s'exécute périodiquement et doit être implémentée par un thread qui s'endort régulierement et la procédure passive crée un nouveau thread qui partage l'état du nœud courant. GossiPeer effectue la création et la gestion des threads de façon transparente pour l'utilisateur, ainsi le développement d'une application est simplifié et seulement la spécification abstraite du code exécuté par chaque thread doit être décrite au niveau de l'application.

Autres méthode et constructeur. L'état du du nœud courant est initialisé simplement par un appel au constructeur de la classe du protocole en question. Une méthode permettant d'écrire dans un fichier de journalisation l'état courant permet ensuite d'étudier les performances de l'application. Cependant cette dernière méthode n'est pas nécessaire et ne fait donc pas partie de l'interface `Applicable`.

3.2 Le module Contact

Le module *Contact* est chargé de fournir au module *Applications* les nœuds en contact avec le nœud courant, typiquement leur identité (leur nom ou leur adresse IP). Certains algorithmes épidémiques tels que Cyclon [VGvS05], Peer-Sampling [JVG+07] ou Newscast [JKvS03] ont pour but le maintien des contacts de chaque nœud : deux nœuds exécutant de tels algorithmes utilisent une communication épidémique pour échanger l'ensemble des contacts qu'ils ont. Ces algorithmes, dits de *gestion de voisinage*, peuvent également être utilisés par des algorithmes du module *Application* puisqu'ils renseignent sur les contacts du nœud courant. En fait, tout algorithme implémentant l'interface `Contactable` permet d'être réutilisé par un algorithme de plus haut niveau. Cette interface est décrite Figure 3.

```
public interface Contactable {
  public String getPeer();
  public Vector<String> getView();
}
```

FIG. 3: Interface du module *Contact*.

La méthode `getPeer()` assure la propriété d'aléat dans le choix du nœud avec lequel le nœud courant peut interagir. La méthode `getView()` quant à elle renvoie l'ensemble des contacts du nœud courant. Le module `Contact` est chargé de maintenir le voisinage de chaque nœud.

4 Comment étudier un nouveau protocole épidémique ?

GossiPeer utilise la « reflection » Java pour permettre l'instanciation de nouveaux objets à l'exécution. Ainsi, GossiPeer est paramétré pour qu'il sache où trouver les nouvelles classes de protocoles à instancier lors de son exécution. Ces informations sont renseignées dans le fichier de configuration de GossiPeer. Le fichier de configuration contient les noms des classes représentant les protocoles à exécuter au niveau de la couche application, `APPROT`, et de la couche contact, `CTPROT`, ainsi que des paramètres propres à GossiPeer tels que `LAUNCH_FREQUENCY` qui détermine la périodicité des applications épidémiques.

Afin de définir un nouveau protocole, le programmeur doit simplement en écrire la classe et en renseigner le nom dans le fichier de configuration. En effet, pour une nouvelle application, le programmeur doit écrire une classe spécifique `MonProtocoleApplication.java` du package `applications` en respectant les contraintes données par l'interface `Applicable`, puis il doit renseigner le champs `APPROT` ← `MonProtocoleApplication.java` dans le fichier de configuration. Nous encourageons vivement les programmeurs à enrichir GossiPeer en y développant également leurs propres protocoles de contact d'une façon identique : en écrivant `MonProtocoleContact.java` tel qu'il respecte l'inter-

face `Contactable` et renseigner le champs `CTPROT` ← `MonProtocoleContact.java` dans le fichier de configuration.

Au sein de GossiPeer, le programmeur bénéficie d'un langage d'un haut niveau d'abstraction pour écrire la spécification de ses protocoles `MonProtocoleApplication` et `MonProtocoleContact`. Par conséquent, non seulement le temps de codage est réduit mais la lecture des protocoles en est grandement simplifiée. En effet, GossiPeer permet au programmeur d'ignorer certaines complications communes à toute application épidémique, notamment :

1. **le multi-threading** : la gestion de l'accès concurrent aux variables partagées entre les threads actifs et ceux de gestion de voisinage—par exemple, le programmeur spécifiant un protocole Applicable doit simplement remplir le constructeur et les méthodes `activeExec` et `passiveExec` et ne crée à aucun moment de thread ;
2. **la tolérance aux fautes** : lorsqu'un nœud contact potentiel tombe en panne, il peut être immédiatement remplacé par un autre nœud actif aléatoirement choisi, et ce de façon transparente.
3. **la communication réseau** : la gestion des ouvertures et fermetures des communications TCP/UDP ainsi que l'écriture et la lecture sur les cannaux de communication appropriés—par exemple, le programmeur crée un message en utilisant `Message msgToSend = new Message()`, le remplit au moyen de `msgToSend.ht.put("key", "value")` et l'envoie en utilisant `CommunicationTCP.sendToANeighbor(msgToSend)`.

5 Conclusion

GossiPeer facilite le développement d'applications épidémiques en factorisant le code commun à ce type d'application. GossiPeer a déjà permis l'étude d'une application de dénombrement de participants et d'une application de morcellement distribué (« distributed slicing ») [GVB+07] avec succès et son code est disponible sur `http ://gossipeer.gforge.inria.fr`. Nous projetons d'utiliser GossiPeer pour l'évaluation d'une application épidémique de mesure de va-et-vient.

Le but de GossiPeer, développé sous licence française de logiciel libre (licence CeCILL), est d'être adopté par les chercheurs du domaine, pour que le catalogue de techniques immédiatement disponibles pour expérimentation soit le plus riche possible. L'unification résultante permet à ces approches d'être employées conjointement pour la construction de blocs logiciels de plus haut niveau, ou à l'inverse permet une comparaison et une mise à l'épreuve des différentes techniques au sein du même environnement d'expérimentation.

Références

[CGJ+07] Paolo Costa, Vincent Gramoli, Márk Jelasity, Gian Paolo Jesi, Erwan Le Merrer, Alberto Montresor, and Leonardo Querzoni. Exploring the interdisciplinary connections of gossip-based systems. *SIGOPS Oper. Syst. Rev.*, 41(5) :51–60, 2007.

[DGH+87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles Of Distributed Computing*, pages 1–12, New York, NY, USA, 1987. ACM.

[GVB+07] Vincent Gramoli, Ymir Vigfusson, Ken Birman, Robbert van Renesse, and Anne-Marie Kermarrec. Fast distributed slicing without requiring uniformity. Technical report, Cornell University, 2007. http ://www.cs.cornell.edu/projects/quicksilver/pubs.html.

[JKvS03] Márk Jelasity, Wojtek Kowalczyk, and Maarten van Steen. Newscast computing. Technical Report IR-CS-006, Vrije Universiteit Amsterdam, November 2003.

[JVG+07] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3) :8, 2007.

[VGvS05] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. Cyclon : Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2) :197–217, 2005.