Une modélisation python de l'algorithme Trickle

Cours de modélisation des systèmes temps réels

Quentin Rossettini, Ayaz Badouraly, Arnaud Convers

Mars - avril 2018

ARCHITECTURE GÉNÉRALE

Organisation des fichiers

Notre fichier de rendu est composé des fichiers suivants :

- rapport_simulation.pdf : ce fichier
- README.md : un fichier explicatif, condensé du rapport
- cattle.py : module contenant la classe “Cattle”
- node.py : module contenant la classe “Node”
- main.py : module contenant le script principal

Modélisations - grandes lignes

Un noeud, ainsi que son comportement local, est encapsulé dans la classe *Node*. L’ensemble des noeuds d’un réseau ainsi que le comportement global du réseau est encapsulé dans la classe *Cattle*. Une instance de *Cattle* aura pour mission de gérer le réseau de noeuds avec une horloge activant à chaque tour un noeud (nous discuterons de la façon dont le Cattle choisi un noeud à chaque tour).

Les choix de configurations sont réalisés directement dans le script main.py, en construisant différentes topologies (objet *cattle*).

Classes : attributs et méthodes

- *Classe Node - Attributs*

Nom	Type	Description
id	int	id du noeud
name	string	nom du noeud
neighbours	set<Node>	ensemble des autres noeuds présents dans le rayon du noeud

I	int	représente la durée de l'intervalle courant
Imin, Imax	int	bornes de la valeur possible pour I
tau	int	moment de la communication
c	int	compteur
t	int	temps (propre au noeud)
n	int	version actuelle du logiciel
inconsistent	bool	Détermine le comportement du nœud à la prochaine expiration de l'intervalle : si le paramètre est à False (informations de version consistantes avec l'état initial du nœud), alors on augmente l'intervalle ; dans le cas contraire, on remet le paramètre I à la valeur I min
buffer	set()	liste des messages reçus à considérer au prochain tick (non ordonné)
k	int	constance de redondance
number_of_code_sendings	int	nombre de communications du code

- *Classe Node - Méthodes*

Signature	Description

broadcast(bool)	Envoie un message (version + éventuellement code) à l'ensemble des voisins
receive(int, bool)	ajoute le message reçu au buffer du noeud
add_neighbour(node)	
remove_neigbour(node)	
update(int)	met à jour la version et le code
tick()	effectue actions (selon messages dans buffer et temps t)
reinit()	réinitialise le noeud

- *Classe Cattle - Attributs*

Nom	Type	Description
nodes	set<Node>	ensemble des noeuds du réseau décrit
Imin	int	taille minimale de l'intervalle d'écoute
max	int	paramètre utilisé pour le caclul de Imax
Imax	int	(propriété) taille maximale de l'intervalle d'écoute
k	int	constante de redondance pour paramétrer le comportement des noeuds
time	int	horloge du cattle (horloge globale)

connected_nodes	set<Nodes>	ensemble des noeuds par lesquels une nouvelle version du réseau peut être injectée
current_version	int	(propriété) Version la plus récente du logiciel
coverage	double	(propriété) pourcentage de couverture du réseau pour la version actuelle

- *Classe Cattle - Méthodes*

Signature	Description
new_node(string, int, bool=False)	ajoute un noeud, connecté ou non
remove_node(node, string)	retire un noeud
tick()	choisi un noeud dans la liste de noeuds et appelle la méthode tick() de ce même noeud
start()	commence la simulation
get_node_by_name(string)	retourne un noeud depuis le nom donné
get_versions()	retourne un dictionnaire donnant les versions de chaque noeud
get_number_of_code_sendings()	retourne le nombre total de message envoyés lors de la simulation
main()	itération de ticks : à chaque tick, le noeud va effectuer ses actions (recevoir, émettre, mettre à jour) ; une fois que toutes ses actions ont été effectuées, le cattle envoie le tick suivant

CHOIX DE CONCEPTION

Nous avons décidé de considérer deux points de vue.

D'abord, du point de vue du noeud. Il est autonome au sens où il possède toute l'information (tous les paramètres de la modélisation) pour tourner seul, comme le ferait un petit objet IoT. Néanmoins, il ne s'active pas automatiquement, et expose une méthode publique pour calculer son état au temps suivant. Cette méthode lit les messages reçus par ses voisins, messages qui ont été stockés dans un buffer. Le noeud se met à jour et envoie un message dans le buffer des ses voisins si besoin.

Ensuite, du point de vue global. L'ensemble des noeuds sont considérés comme un troupeau (*cattle* en anglais.) C'est cet objet qui va appeler les méthodes d'activation de chaque noeud.

Pour simuler la notion de temps, nous avons implémenté deux heuristiques. La première consiste à choisir **au hasard** un noeud à activer. La seconde consiste à ordonner les noeuds du modèle dans une liste Python et à choisir les noeuds **successivement**. Cette seconde heuristique a l'avantage d'être déterministe.

LIMITATIONS

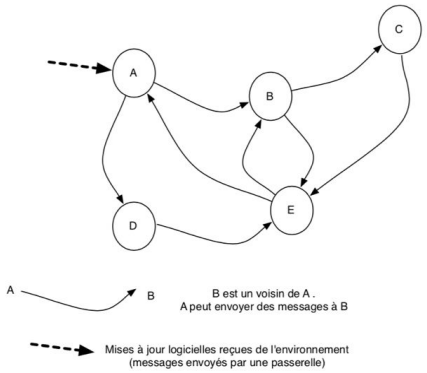
La principale limitation de notre modèle est de ne pas activer des noeuds en parallèle, mais bien les uns à la suite des autres. Une conséquence immédiate est l'impossibilité de modéliser un système réactif, c'est-à-dire modèle où chaque capteur exécuterait une routine dès la réception d'un message d'un de ses voisins.

Est-ce une limitation très forte ? Une étude plus longue serait à mener pour répondre précisément à la question. Néanmoins, une première piste est de remarquer que le temps de réaction d'un noeud (*ie.* le temps de calcul qu'il met à la réception du message) est intrinsèque à notre modélisation par buffer : le temps de calcul est simulé par le temps que le *cattle* met à appeler le noeud.

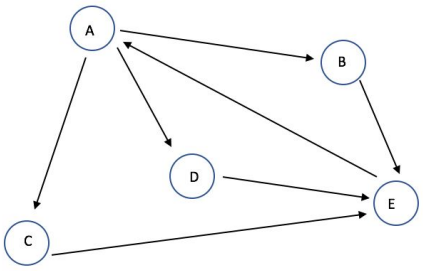
ANALYSE DE NOS RÉSULTATS

Nous avons testé plusieurs topologies :

- Topologie simple, présentée dans l'énoncé



- Topologie bloquante



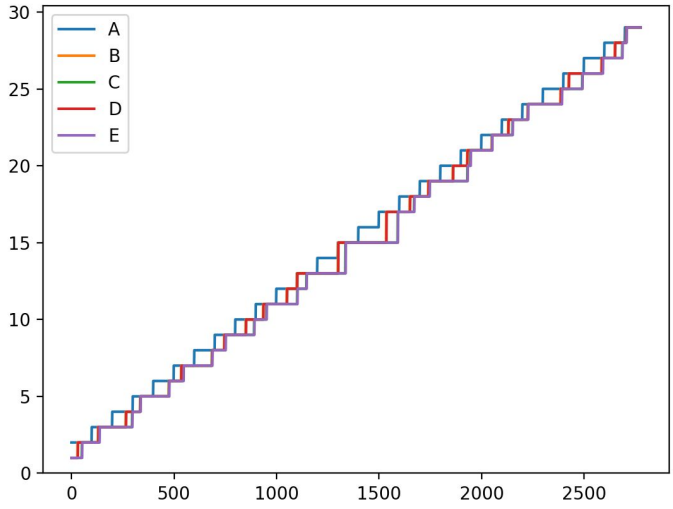
- Topologie aléatoire : étant donné un nombre de noeuds et un facteur de connexion (*ie* le nombre moyen de voisins par noeud), une topologie est générée

Notre code permet plusieurs utilisations :

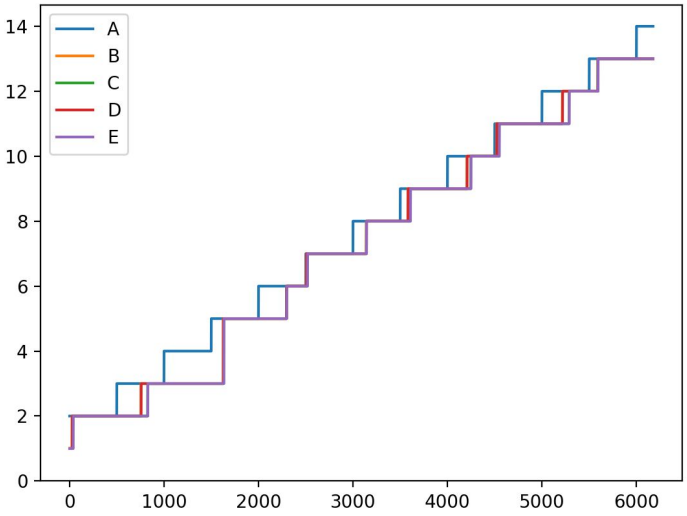
- Génération d'une trace d'exécution
- Visualisation d'un graphique d'évolution des versions des noeuds
- Calcul du temps moyen de propagation d'une nouvelle version
- Détection des topologies bloquantes

1) Graphique d'évolution des versions

On peut tracer l'évolution des versions pour chaque noeud du réseau. Par exemple pour la topologie simple avec une nouvelle version poussée sur le noeud A tous les 100 pas de temps (en abscisse le temps, en ordonnée le numéro de version) :

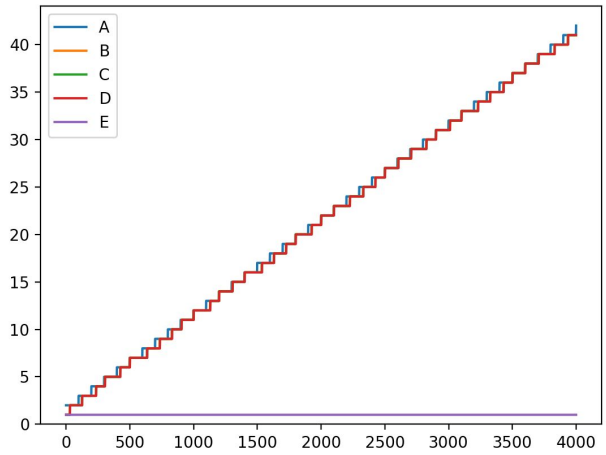


Pour la même topologie mais avec une nouvelle version tous les 500 pas de temps :



On observe dans les deux cas qu'il existe des versions qui ne sont jamais transmises à tous les noeuds (certains noeuds "sautent" des versions).

Avec la topologie bloquante :



On voit ici que cette topologie est “bloquante”, dans le sens où le noeud E ne reçoit jamais de nouvelle version.

2) Temps moyen de propagation d’une nouvelle version

On se place dans le cas où aucune nouvelle version n’est poussée de l’extérieur. On s’intéresse au temps nécessaire pour que la version 2, initialement présente uniquement sur le premier noeud, se propage à tous les noeuds. Ce temps est infini pour une topologie bloquante.

Pour la topologie simple, ce temps moyen est d’environ **255** pas de temps, et environ **45** messages sont échangés en moyenne.

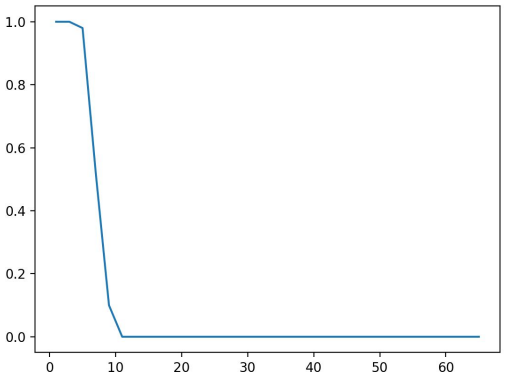
Pour la topologie bloquante, ce temps est **infini** par définition.

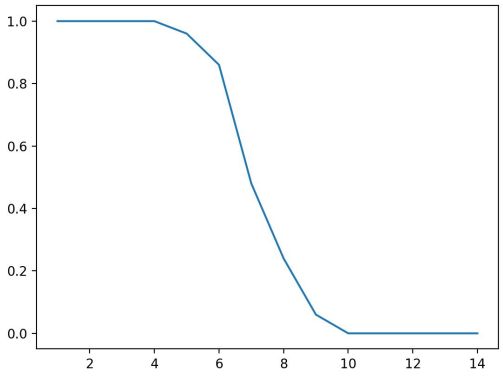
Pour les topologies aléatoires, le but est de faire une moyenne sur *n* topologies aléatoirement générées. Le problème est que certaines topologies générées sont bloquantes, et qu’il suffit qu’une seule soit bloquante pour que le temps moyen soit infini.

3) Détection des topologies bloquantes

La détection de topologies bloquantes se fait de façon empirique : si aucun noeud ne change de version pendant un laps de temps donné (100 pas de temps), alors on considère que la topologie est bloquante. Ce n’est pas une méthode parfaite, car une topologie peut être classée comme bloquante alors qu’elle ne l’est pas. Par contre, si une topologie est classée non bloquante, alors on est sûr qu’elle l’est.

Nous avons pu tracer un graphique de l’évolution de la proportion de topologies bloquantes en fonction du nombre moyen de voisins dans les topologies aléatoires à 100 noeuds (le graphique de droite est zoomé sur la partie intéressante) :

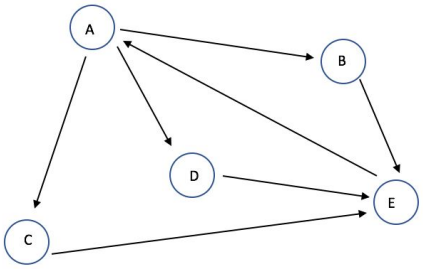




On voit donc qu'il y a un seuil à passer, situé environ à 10 voisins, au-delà duquel une topologie sera presque sûrement non-bloquante.

VARIANTE

Nous avons remarqué qu'avec l'algorithme proposé, nous faisions face à de nombreux cas de topologies bloquantes, y compris dans des graphes où le nombre de voisins par noeud n'est pas spécialement faible. Reprenons la topologie bloquante suivante :

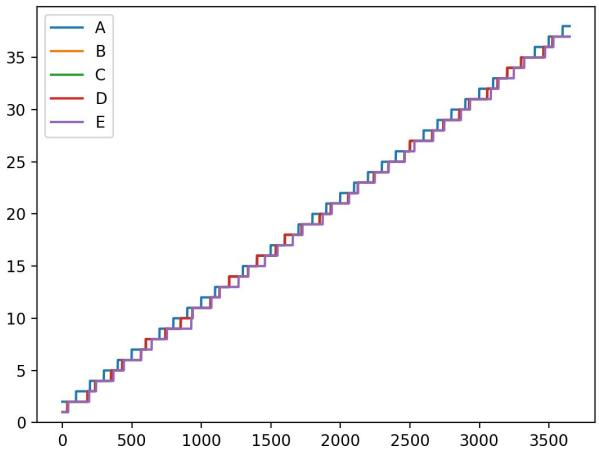


Nous avons observé que le noeud E ne recevait jamais de nouvelle version. En effet, il ne peut en recevoir que si un de ses prédécesseurs (B, C ou D) reçoit un numéro de version inférieur strictement au sien. Or ces noeuds ne reçoivent d'informations que du noeud A, qui est lui-même à jour, et donc qui n'enverra jamais un numéro strictement inférieur à ceux de l'un de ses fils.

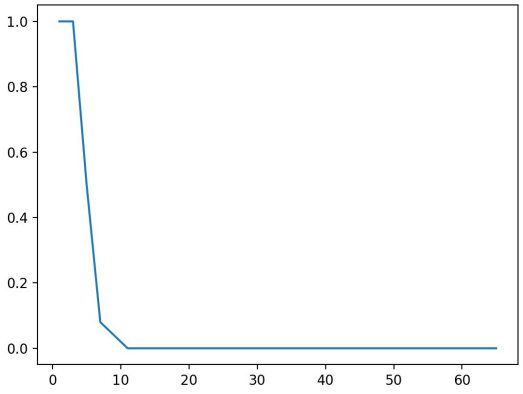
Cette topologie est donc bloquante, alors que visuellement, elle semble suffisamment connectée. En d'autres termes, on s'attendrait, dans une telle topologie, à ce que l'algorithme "trickle" fonctionne bien.

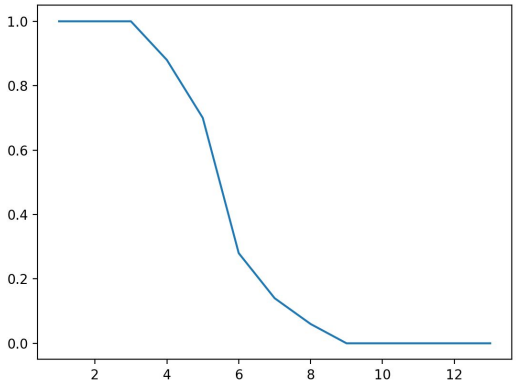
Après quelques recherches, nous avons trouvé une nouvelle version de "trickle", dans laquelle quand un noeud arrive à la fin d'un intervalle, si son compteur est strictement inférieur à sa constante *k*, alors il envoie son numéro de version ET son code à ses voisins. Dans l'algorithme proposé dans le sujet, le noeud n'envoie que son numéro de version.

Avec cette variante, voici le graphe d'évolution des versions de chaque noeud dans la topologie ci-dessus :



On voit que la topologie n'est plus bloquante avec cette variante. Si on retrace le graphique d'évolution de la proportion de topologies bloquantes pour les topologies aléatoires de 100 noeuds en fonction du nombre de voisins, on obtient ceci (le graphique de droite est zoomé sur la partie intéressante) :





On voit, en comparant avec le graphique précédent, que le seuil du nombre de voisins critique est légèrement inférieur à celui avec l'ancienne version de l'algorithme. Cela nous conforte donc dans l'idée que cette variante permet d'éviter certaines topologies bloquantes.

UTILISATION DU CODE

Notre code peut s'utiliser de différentes façons. Il suffit pour cela de modifier les valeurs des variables CONFIG, TOPOLOGY, FREQ_NEW_VERSION, NB_NODES, AVG_NB_NEIGHBOURS et ALGORITHM_VERSION dans le fichier main.py (lignes 28 à 40). C'est ce fichier qui doit être exécuté pour lancer la simulation (quelle que soient les valeurs des variables). Voici les utilisations possibles du code :

1) Graphique d'évolution des versions

Pour obtenir un graphique d'évolution des versions des différents noeuds, voici les valeurs à donner aux variables :

- CONFIG = CHART_VERSIONS_EVOLUTION
- TOPOLOGY = SIMPLE_TOPOLOGY, BROKEN_TOPOLOGY ou RANDOM_TOPOLOGY (selon la topologie souhaitée)
- FREQ_NEW_VERSION = valeur au choix (entre 100 et 500 pour avoir des résultats observables)
- NB_NODES : inutile sauf si TOPOLOGY == RANDOM_TOPOLOGY, dans ce cas, choisir le nombre de noeuds dans les topologies aléatoires souhaitées
- AVG_NB_NEIGHBOURS : inutile sauf si TOPOLOGY == RANDOM_TOPOLOGY, dans ce cas, choisir le nombre moyen de voisins par noeud dans les topologies aléatoires souhaitées
- ALGORITHM_VERSION = BASIC ou VARIANT, selon la version de l'algorithme souhaitée

Pour obtenir le graphique, lancer la simulation, puis une fenêtre contenant le graphique s'ouvrira après un court instant. Cette simulation génère également une trace dans le fichier network.log.

2) Temps moyen de propagation d'une nouvelle version

Pour obtenir un graphique d'évolution des versions des différents noeuds, voici les valeurs à donner aux variables :

- CONFIG = AVERAGE
- TOPOLOGY = SIMPLE_TOPOLOGY, BROKEN_TOPOLOGY ou RANDOM_TOPOLOGY

(selon la topologie souhaitée)
- FREQ_NEW_VERSION : la valeur n'est pas utilisée dans ce cas
- NB_NODES : inutile sauf si TOPOLOGY == RANDOM_TOPOLOGY, dans ce cas, choisir le nombre de noeuds dans les topologies aléatoires souhaitées
- AVG_NB_NEIGHBOURS : inutile sauf si TOPOLOGY == RANDOM_TOPOLOGY, dans ce cas, choisir le nombre moyen de voisins par noeud dans les topologies aléatoires souhaitées
- ALGORITHM_VERSION = BASIC ou VARIANT, selon la version de l'algorithme souhaitée

Lancer la simulation, puis après un court instant, le temps moyen de propagation de la version ainsi que le nombre moyen de messages échangés s'affiche, ainsi que la proportion de topologies bloquantes (0 si SIMPLE_TOPOLOGY, 1 si BROKEN_TOPOLOGY, peut prendre d'autres valeurs dans le cas de topologies aléatoires).

3) Proportion de topologies bloquantes dans les topologies aléatoires

Pour obtenir un graphique d'évolution des versions des différents noeuds, voici les valeurs à donner aux variables :

- CONFIG = CHART_RANDOM
- TOPOLOGY : la valeur n'est pas utilisée dans ce cas
- FREQ_NEW_VERSION : la valeur n'est pas utilisée dans ce cas
- NB_NODES : choisir le nombre de noeuds dans les topologies aléatoires souhaitées
- AVG_NB_NEIGHBOURS : choisir le nombre moyen de voisins par noeud dans les topologies aléatoire souhaitées
- ALGORITHM_VERSION = BASIC ou VARIANT, selon la version de l'algorithme souhaitée

Cette simulation prend plusieurs minutes, afin d'avoir une moyenne significative. Quand elle se termine, le graphe s'affiche dans une fenêtre séparée.

CONCLUSION

Nous présentons ici une modélisation possible d'un ensemble de capteurs qui communiquent leur firmware en utilisant l'algorithme Trickle. En se fixant une heuristique d'évolution du temps, nous pouvons étudier des comportements macroscopiques. Nous pouvons suivre la propagation d'une mise à jour sur le réseau et mesurer son temps moyen en multipliant les exécutions de la simulation. Notamment, nous pouvons mettre en avant des topologies bloquantes.

Toutefois, il ne s'agit là que d'une simulation, dont les résultats ne sont pas **prouvés**. C'est un outil qui permet d'émettre des hypothèses qui peuvent ensuite être vérifiées grâce à d'autres outils spécialisés, tels que UPAAL ou Diversity.

Il serait également intéressant de pousser l'étude en modifiant la modélisation et notamment l'implémentation du temps dans notre modèle.