Mes notes sur mon “simulateur réseau”

Emmanuel Chaput

13 octobre 2011

Table des matières

Début de doc

o Modèle de transmission des PDU

- Tout module susceptible de produire ou transférer des PDU doit fournir une fonction de la forme

`struct PDU_t * getPDU(void * source);`

Le paramètre est un pointeur vers des "données privées" permettant d'identifier l'instance du module (typiquement un pointeur direct sur cette instance).

Le pointeur retourné est celui d'une PDU qui n'est plus prise en compte par la source. Elle doit donc absolument être gérée (ou, au moins, détruite) par l'utilisateur de cette fonction. En cas d'indisponibilité de PDU, la valeur NULL est retournée.

Cette fonction et le pointeur associé doivent être fournis à l'entité destinataire, s'il en existe une !

Si le nom du module est toto, la fonction sera nommée par exemple `toto_getPDU()`.

- Tout module susceptible de recevoir des PDU doit fournir une fonction de la forme :

`void processPDU(void * rec, getPDU_t getPDU, void * source);` C'est cette fonction qu'invoquera une source pour lui notifier la disponibilité d'une PDU. Cette fonction aura donc la responsabilité d'aller récupérer la PDU (grâce la la fonction getPDU et à la source fournies) et de la traiter. La récupération et le traitement pourront être remis à plus tard (en cas d'indisponibilité) mais au risque d'avoir un pointeur NULL retourné par getPDU().

Si le nom du module est `toto`, la fonction sera nommée par exemple `toto_processPDU()`.

Le problème ici est que ce modèle, censé permettre plus simplement d'avoir plusieurs producteurs vers un même consommateur (grâce à la fonction getPDU et au pointeur source passés en paramètres plutôt que stockés comme attributs) rend délicat le "report" de l'invocation du getPDU. Comment, dans un serveur par exemple, assurer que cette invocation se fera dans le même traitement que la fin de traitement ? Cela dit, est-ce vraiment nécessaire ? Pour le moment, cela implique un événement pour la fin du traitement et un événement pour chaque getPDU.

La difficulté est d'assurer un traitement dans le bon ordre de ces événements qui ont la même date ... J'ai pourtant dans l'idée que c'est la bonne solution. Plusieurs pistes pour rêgler ce problème des événements "simultanés" mais devant être traités dans un ordre donné, sachant que le problème des événements simultanés est complexe :

. A date égale, les événements sont insérés dans l'ordre de création et exécutés dans l'ordre d'insertion. Bof, notemment dans la mesure où la date est un réel. Avantage : c'est simple ! . Gérer des chaines d'événements ?

Bon, en fait, ici, le mieux est peut-être d'abandonner cette idée. Elle n'est pas logique. Si plusieurs sources signalent des paquets disponibles, il n'y a aucune raison que ça provoque des événements simultanés ! Il faut donc stocker le getPDU et le pointeur associé. Eventuellement dans une liste (ordonnée) pour en permettre plusieurs. Le traitement n'est pas alors différé à un nouvel événement mais reporté à la fin du service en cours.

1 Les sondes

Les sondes sont implantées dans le modules `probe`. Les sondes permettent d'enregistrer des mesures de paramètres scalaires. Ce sont elles qui permettront ensuite d'analyser le résultat de la simulation, d'évaluer les performances du système, de tracer des courbes, ...

Chaque échantillon d'une mesure prélevé par une sonde est daté. Il est égalament possible de n'enregistrer que la date d'un événement, sans aucune mesure associée.

1.1 Les méthodes

1.1.1 L'échantillonage

1.1.2 Consultation d'un échantillon

1.1.3 La moyenne

1.1.4 Les valeurs extrèmes

1.1.5 La sauvegarde

Une sonde peut être dumpée dans un fichier ouvert grâce à la fonction
`void probe_graphBarDumpFd(struct probe_t * probe, int fd, int format);`

1.2 Les différents types

1.2.1 L'histogramme

1.2.2 La fenêtre glissante

Ces sondes conservent tous les échantillons sur une fenêtre glissante dont la taille est fournie en paramêtre du constructeur :

```
// Conserve des échantillons sur une fenêtre
struct probe\_t * probe\_slidingWindowCreate(int windowLength);
```

1.2.3 La moyenne

1.2.4 La moyenne par tranches temporelles

Cette sonde conserve une moyenne pour chaque tranche temporelle de durée `t`, passée en paramètre du constructeur (les fenêtres temporelles sont sautantes donc disjointes) :

```
struct probe_t * probe\_createTimeAverage(double t);
```

2 L'affichage par GnuPlot

3 Notion de simulation et de campagne

Une simulation est une instance unique d'exécution d'une séquence d'événements suite à l'initialisation du modèle. Une campagne est une suite de simulations sur un même modèle avec une ré-initialisation des variables entre deus simulations.

Les sondes liées à la simulation sont ré-initialisées à la fin de la simulation. Des sondes peuvent être liées à la campagne ; elles ne seront réinitialisées qu'à la fin de cette dernière et peuvent permettre ainsi à établir des valeurs inter-simulation, par exemple des intervalles de confiance.