Mes notes sur mon “simulateur réseau”

Emmanuel Chaput

18 janvier 2013

Table des matières

1 Tutoriel

1.1 Introduction

Félicitations ! Tu t'apprêtes à entrer dans le monde fabuleux de NDES, ...

Il s'agit en gros d'une sorte de librairie astucieusement agrémentée de nombreux bugs dont le but et de t'aider (ou pas) à faire ton propre simulateur réseau ! Si tu es moi, tout ça est à peu près vrai ; pour les autres, il y a mieux ailleurs.

L'implantation d'un simulateur passera donc par l'écriture d'un programme C utilisant cette librairie.

1.2 Installation

```
$ git clone https://github.com/Manu-31/ndes.git
$ cd ndes
$ make
$ make install
```

Pas de panique, ça n'installe rien pour le moment !

1.3 Ma première simulation : chouette, une file M/M/1 !

Le fichier source (et son Makefile, parce qu'on ne se moque pas du client) se trouve dans le répertoire `example/tutorial-1`.

Dans NDES, le système va être modélisé par une source, suivie d'une file d'attente, en aval de laquelle se trouve un serveur suivi par un puits. Je te laisse faire un dessin et je publierai ici le plus joli !

1.3.1 Création du simulateur

Avant toute manipulation, on crée le simulateur de la façon suivante

```
#include <motsim.h>

...
```

```
/* Creation du simulateur */
motSim_create();
```

1.3.2 Création du puits

Un puits sera un objet qui reçoit sans rechigner des messages et qui les détruit instantanément. On le crée très simplement de la façon suivante

```
#include <pdu-sink.h>

...

   struct PDUSink_t       * sink; // Déclaration d'un puits

   ...

   /* Crétion du puits */
   sink = PDUSink_create();
```

1.3.3 Création du serveur

Dans la mesure où on ne souhaite rien faire de bien intelligent avec notre serveur, nous allons utiliser un serveur générique, offert par la maison, qui ne recule devant aucun sacrifice !

Il s'utilise comme ça

```
#include <srv-gen.h>

...

   struct srvGen_t        * serveur; // Déclaration d'un serveur générique

...

   /* Création du serveur */
   serveur = srvGen_create(sink, (processPDU_t)PDUSink_processPDU);
```

La création du serveur est un peu plus compliquée que celle du puits. La raison est que le serveur doit savoir à qui envoyer les clients (des PDUs pour NDES) après les avoir servis. Il faut donc lui dire quelle fonction leur appliquer (ici `PDUSink_processPDU` un fonction spécifiques aux puits) et à quelle entité cette fonction s'applique (notre unique puits, ici).

Les serveurs génériques sont décrits plus précisément dans la section 3.

1.3.4 Création de la file

Une file permet de stocker des PDU en transit. On la construit ainsi

```
struct filePDU_t       * filePDU; // Déclaration de notre file

 ...

/* Création de la file */
filePDU = filePDU_create(serveur, (processPDU_t)srvGen_processPDU);
```

Les files sont décrites plus précisément dans la section 4.

2 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.

Les sondes peuvent être placées en divers points des outils de simulation via des fonctions de la forme <type>_add<point>Probe(...). Plusieurs sondes peuvent être ainsi "chaînées" sur un même point de mesure. En revanche, la même sonde **ne peut pas** être chaînée plusieurs fois.

2.1 Les méthodes de base

2.1.1 L'échantillonage

La méthode suivante permet d'échantilloner une valeur dans une sonde

```
/*
 * Echantillonage d'une valeur
 */
void probe_sample(struct probe_t * probe, double value);
```

Il est également possible d'échantilloner une date, sans valeur associée, de la façon suivante

```
/*
 * Echantillonage de la date d'occurence d'un evenement
 */
void probe_sampleEvent(struct probe_t * probe);
```

2.1.2 Consultation d'un échantillon

2.1.3 La moyenne

2.1.4 Les valeurs extrèmes

2.1.5 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);`

2.1.6 La mesure du débit

Les valeurs échantillonées pourront souvent être des tailles de messages, émis ou reçus. Dans ce cas, il peut être intéressant d'utiliser la fonction suivante qui fournit une mesure du "débit instantané".

```
double probe_throughput(struct probe_t * p);
```

Bien sûr, le mode d'estimation de cette valeur est dépendant de la nature de la sonde.

2.2 Les différents types

2.2.1 Les "méta-sondes"

À chaque échantillon, certaines sondes mettent à jours des informations que l'on peut souhaiter conserver (par exemple la moyenne mobile) afin de tracer leur évolution dans le temps.

Pour cela, une sonde périodique P peut être placée sur une sonde observée O. La sonde P permettra ainsi d'observer une propriété de la sondeO sur la base d'un échantillonage à une fréquence caractérisant la sonde P.

Cette technique permet également de collecter dans une sonde unique G des échantillons prélevés dans plusieurs sondes différentes S1, S2, ldots Pour cela, G sera ajoutée comme sonde sur les échantillons de S1, S2, ldots

2.2.2 La sonde exhaustive

2.2.3 L'histogramme

2.2.4 La fenêtre glissante

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

```
struct probe_t * probe_slidingWindowCreate(int windowLength);
```

2.2.5 La sonde périodique

Le but d'une sonde périodique est de prélever un échantillon toutes les τ unitées de temps. On se fonde pour cela sur l'idée que la valeur mesurée n'est modifier qu'au cours du traitement d'un événement.

2.2.6 La moyenne mobile

Une moyenne du type EMAProbeType conserve à tout moment une moyenne mobile calculée à chaque nouvel échantillon e de la façon suivante $m < -\alpha.m + (1 - \alpha).e$.

2.2.7 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);
```

3 Le serveur générique

Le serveur générique permet de modéliser à moindre frais un serveur. Il est définit dans le fichier `srv-gen.h`.

4 Les files

C'est l'outil de base pour stocker des objets selon une stratégie FIFO. Elles sont définies dans le fichier `file_pdu.h`.

4.1 Création

Une file est créée de la façon suivante

```
struct filePDU_t * filePDU_create(void * destination,
    processPDU_t destProcessPDU);
```

Le paramètre destination est un pointeur sur l'objet vers lequel sont transmis les objets présents dans la file. Dés qu'un objet est inséré dans la file, si la destination est disponible, il lui est envoyé.

4.2 Gestion de la taille

4.3 Les sondes

Les files sont dotées des sondes suivantes

`InsertSize` pour mesurer la taille des paquets insérés dans la file. A chaque insertion d'une PDU, la taille de cette dernière est échantillonée avec la date d'insertion.

`ExtractSize` pour mesurer la taille des paquets extraits de la file. A chaque extraction d'une PDU, la taille de cette dernière est échantillonée avec la date d'extraction.

`Sejourn`

5 L'affichage par GnuPlot

6 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.

7 Des exemples

7.1 Utilisation des sondes

7.1.1 Mesurer un débit

Considérons le cas simple d'une source, dont nous voulons mesurer le débit de sortie. Pour cela, nous allons insérer une sonde sur la taille des paquets transmis grâce à la méthode `PDUSource_setPDUGenerationSizeProbe`.

Le type de sonde dépendra de la mesure souhaitée.

Débit moyen Supposons que nous voulons simplement connaître le débit moyen sur toute la transmission, alors une sonde mesurant la moyenne sera parfaitement suffisante :

Débit "instantanté"

8 Vrac

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.