Mes notes sur mon “simulateur réseau”

Emmanuel Chaput

5 février 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 ! Cela ne fait que copier une librairie (statique) dans un répertoire dédié dans l'arborescence.

On peut également compiler quelques programmes de tests de non régression :

```
$ make tests-bin
```

Et pourquoi pas les lancer :

```
$ make tests
```

Évidemment, les messages `OK` ne prouvent rien ! Seuls les messages d'erreur prouvent qu'il y a un problème, ... comme s'il y avait besoin de preuves.

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). Pour connaître la fonction, il faut la chercher dans la description de l'entité cible. Dans notre exemple, la fonction `PDUSink_processPDU` est décrite dans la section dédiée aux puits.

Plus de précisions sur cette histoire de fonctions dans la section 2 de description de l'architecture générale.

Comme nous voulons une file M/M/1, nous devons dire à notre serveur que son temps de traitement est exponentiel de paramètre `mu`. Et zou :

```
   float   mu = 10.0; // Paramètre du serveur

   ...

   /* Paramétrage du serveur */
   srvGen_setServiceTime(serveur, serviceTimeExp, mu);
```

Les serveurs génériques sont décrits plus précisément dans la section 10.

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);
```

Sans autre forme de procès, les files ne sont pas bornées. Elles sont décrites plus précisément dans la section 7. Les deux paramètres de la fonction de création ont le même rôle que ceux de la fonction de création du serveur.

1.3.5 Création de la source

Nous voilà à la source ! Nous allons utiliser un objet de NDES dont le rôle est de produire des PDUs. Mais avant cela, nous devons créer un autre objet qui lui indiquera les dates auxquelles les produire : il s'agit d'un "générateur de date", original, non ? Les générateurs de dates sont décrits dans la section **??**.

On veut une source poisonnienne, donc un générateur exponentiel :

```
#include <date-generator.h>

...

   struct dateGenerator_t * dateGenExp; // Un générateur de dates
   float   lambda = 5.0 ; // Intensité du processus d'arrivée

   ...

   /* Création d'un générateur de date */
   dateGenExp = dateGenerator_createExp(lambda);
```

Et maintenant nous pouvons donc créer notre source :

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

...

   struct PDUSource_t     * sourcePDU;  // Une source

   ...

   sourcePDU = PDUSource_create(dateGenExp,
filePDU,
(processPDU_t)filePDU_processPDU);
```

Le premier paramètre est donc l'objet qui lui permet de déterminer les dates d'envoi. Les deux suivants sont similaires à ceux passés lors de la cration de la file et du serveur.

1.3.6 Mise en place de sondes

Oui, je sais, le titre fait un peu peur, mais ça va bien se passer. Lorsqu'on veut lancer un simulateur, on espère en général obtenir des résulats. Dans NDES, ceux-ci seront collectés par un outil spécifique, la sonde.

Les différents types de sonde sont décrits dans la section 5. Nous utiliserons ici uniquement des sondes exhaustives. Pour chaque objet décrit, la liste des sondes disponibles est fournie.

Nous les déclarons et initialisons comme ça :

```
#include <probe.h>

...

   struct probe_t         * sejProbe, * iaProbe, * srvProbe; // Les sondes

...

   /* Une sonde sur les interarrivées */
   iaProbe = probe_createExhaustive();
   dateGenerator_setInterArrivalProbe(dateGenExp, iaProbe);

   /* Une sonde sur les temps de séjour */
   sejProbe = probe_createExhaustive();
   filePDU_addSejournProbe(filePDU, sejProbe);

   /* Une sonde sur les temps de service */
   srvProbe = probe_createExhaustive();
   srvGen_setServiceProbe(serveur, srvProbe);
```

1.3.7 Lancement de la simulation

Ça y est, nous y sommes enfin, voici comment démarrer la simulation. Nous devons activer les entités voulue au moment voulu. Ici, il n'y a que l'unique source à activer, et nous souhaitons le faire dès le début de la simulation :

```
   /* On active la source */
   PDUSource_start(sourcePDU);
```

Nous lançons maintenant la simulation. Nous allons la faire durer 100 secondes :

```
   /* C'est parti pour 100 000 millisecondes de temps simulé */
   motSim_runUntil(100000.0);
```

Nous pouvons ensuite afficher quelques paramètres internes du simulateur :

```
   motSim_printStatus();
```

Et voilà !

1.3.8 Affichage des résultats

Maintenant que notre simulation est terminée, on a certainement envie d'en voir le résultat. On utilisera pour cela des fonctions founies par les sondes, par exemple :

```
   /* Affichage de quelques résultats scalaires */
   printf("%d paquets restant dans  la file\n",
  filePDU_length(filePDU));
   printf("Temps moyen de sejour dans la file = %f\n",
  probe_mean(sejProbe));
   printf("Interarive moyenne     = %f (1/lambda = %f)\n",
  probe_mean(iaProbe), 1.0/lambda);
   printf("Temps de service moyen = %f (1/mu     = %f)\n",
  probe_mean(srvProbe), 1.0/mu);
```

Génial, non ? Non ! Mais la suite est plus rigolote, ...

1.3.9 Tracé de courbes

Pour obtenir des résultats plus riches, nous allons utiliser (depuis le simulateur) un affichage *gnuplot*. Nous avons ici au moins deux courbes intéressantes à tracer, donc nous allons écrire une fonction pour cela :

```
/*
 * Affichage (via gnuplot) de la probre pr
 * elle sera affichée comme un graphbar de nbBar barres
 * avec le nom name
 */
void tracer(struct probe_t * pr, char * name, int nbBar)
{
   struct probe_t   * gb;
   struct gnuplot_t * gp;

   /* On crée une sonde de type GraphBar */
   gb = probe_createGraphBar(probe_min(pr), probe_max(pr), nbBar);

   /* On convertit la sonde passée en paramètre en GraphBar */
   probe_exhaustiveToGraphBar(pr, gb);

   /* On la baptise */
   probe_setName(gb, name);

   /* On initialise une section gnuplot */
   gp = gnuplot_create();
```

```
    /* On recadre les choses */
    gnuplot_setXRange(gp, probe_min(gb), probe_max(gb)/2.0);

    /* On affiche */
    gnuplot_displayProbe(gp, WITH_BOXES, gb);
}
```

L'utilisation de *gnuplot* est décrite dans la section 12.

Attention à ne pas oublier de mettre une petite pause à la fin de notre programme principal, sinon il s'arrête et il tue ses processus fils, et donc l'affichage gnuplot disparaît.

1.3.10 Utilisation de notre premier simulateur

Il ne nous reste plus qu'à compiler notre programme et à le lancer. Le répertoire `examples/tutorial-1` contient également un makefile que je te laisse observer, mais en gros, il faut aller chercher les includes et la librairie. En utilisant ce `Makefile`, on a donc :

```
 $ cd examples/tutorial-1
 $ make
 $ ./mm1
[MOTSI] Date = 99999.846555
[MOTSI] Events : 998243 created (3 m + 998240 r)/998242 freed
[MOTSI] Simulated events : 998243 in, 998242 out, 1 pr.
[MOTSI] PDU : 998242 created (27 m + 998215 r)/998242 freed
[MOTSI] Total malloc'ed memory : 25169976 bytes
[MOTSI] Realtime duration : 1 sec
0 paquets restant dans  la file
Temps moyen de sejour dans la file = 0.061008
Interarive moyenne     = 0.200352 (1/lambda = 0.200000)
Temps de service moyen = 0.100176 (1/mu     = 0.100000)
*** ^C pour finir ;-)
 $
```

1.4 Ma deuxiéme simulation : super, encore une file M/M/1 !

Le premier exemple est sympa, vas-tu me dire, mais ce serait mieux si le temps de traitement dépendait de la taille des clients (comme des paquets dont le temps d'émission dépend de la taille !)

Et bien soit ! C'est ce que nous allons faire dans ce deuxième tutoriel. Les fichiers se trouvent dans le répertoire `example/tutorial-2`.

Comme je n'ai pas que ça à faire, et toi non plus probablement, je vais juste commenter les différences par rapport au tutoriel précédent.

Une petite modification cosmétique est apportée dans la définition des paramètres de la simulation, que l'on va présenter avec un vocabulaire plus réseau et moins file d'attente :

```
float frequencePaquets = 5.0;      // Nombre moyen de pq/s
float tailleMoyenne    = 1000.0;   // Taille moyenne des pq
float debit            = 10000.0;  // En bit par seconde
```

Ces valeurs sont peu réalistes, mais choisies astucieusement pour avoir les mêmes résultats que le premier tutoriel.

1.4.1 Génération de la taille des paquets

Voilà la principale différence avec le premier tutoriel. On va utiliser un générateur de nombres aléatoires pour avoir des paquets de taille variable :

```
/* Création d'un générateur de taille (tailles non bornées) */
sizeGen = randomGenerator_createUInt();
randomGenerator_setDistributionExp(sizeGen, 1.0/tailleMoyenne);

/* Affectation à la source */
PDUSource_setPDUSizeGenerator(sourcePDU, sizeGen);
```

Les générateurs de nombres aléatoires sont décrits dans la section 8. Profitons en pour placer une sonde sur cette taille, afin de vérifier qu'elle a bien la bonne tête :

```
/* Une sonde sur les tailles */
szProbe = probe_createExhaustive();
randomGenerator_setValueProbe(sizeGen, szProbe);
```

Avec ça, on pourra tracer une jolie courbe de plus !

1.4.2 Prise en compte par le serveur

Ce que l'on vient de faire ne sert à rien si le serveur ne le prend pas en compte. Il nous faut donc préciser que ce dernier sert chaque client en un temps dépendant de la taille :

```
/* Paramétrage du serveur */
srvGen_setServiceTime(serveur, serviceTimeProp, 1.0/debit);
```

Cela signifie que le temps de service d'un client est proportionnel à sa taille avec comme coefficient l'inverse du débit.

2 L'architecture générale

Le but de cette section est de décrire la logique de NDES.

2.1 Type de simulation

NDES est une librairie de simulation de réseaux par événements discrets. Non, ce n'est pas original, mais c'est tout de même vachement important pour la suite. Tout le code de traitement d'un événement que l'on va écrire sera exécuté en une durée éventuelllement super longue, mais absolument pas comptabilisé dans le temps simulé. Et pire ! Pendant le traitement d'un événement, comme le temps simulé est figé, le système n'évolue pas (sauf au travers du code en question). C'est du connu, d'accord, mais c'est lourd de conséquences (musique stressante).

2.2 Schéma général des modèles

L'idée fondamentale est que le réseau à simuler peut être modélisé comme une suite d'entitées chaînées entre elles. Dans la version la plus simple, strictement linéaire, la première de ces entités va produire des messages, nommés PDUs, qu'elle fera suivre à l'entité suivante et ainsi de suite jusqu'à la dernière qui pourra sonserver les messages ou les détruire.

La file M/M/1 décrite dans le premier tutoriel est un parfait exemple de ce modèle. Nous connaissons donc déjà quatre types de noeuds : `sourcePDU`, `filePDU`, `serv_gen` et `PDUSink`.

La construction du réseau se fait en partant "de la fin" et en remontant vers la source, comme on peut le voir dans le tutoriel. Chaque fois que l'on crée une entité, on doit lui passer comme paramètres des informations sur l'entité en aval. Elle n'a, en revanche, aucun besoin de connaître la ou les entités amont. Oui, il peut y en avoir plusieurs.

Certaines entités peuvent également avoir plusieurs entités aval (qui seront fournies par une fonction spécifique), mais une PDU donnée n'est passée qu'à une seule d'entre elles.

La mauvaise nouvelle, c'est que que ce schéma va être à revoir tôt ou tard ! Il n'est pas génial pour modéliser des choses plus symétriques (une entité protocolaire qui émet et reçoit), et des choses sont à revoir comme le terme de PDU qui n'est pas du tout approprié. Ce ne sont pas que des PDU qui sont représentées par cette chose !

2.3 Fonctions d'échange des PDU

Du fait de l'architecture générale du réseau, pour toute entité susceptible de produire ou transférer des PDU doit exister une fonction de la forme

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

Le paramètre est un pointeur vers des "données privées" permettant d'identifier le noeud (typiquement un pointeur direct sur ce noeud).

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()`.

Symétriquement, tout module susceptible de recevoir des PDU doit fournir une fonction de la forme :

```
int 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 à 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().

La valeur de retour de cette fonction est nulle en cas d'échec, et non nulle en cas de succès.

Cette fonction peut être invoquée avec les deux derniers paramètres `NULL`. Dans ce cas, l'objectif est uniquement de déterminer si l'entité `rec` est disposée à traiter une nouvelle PDU. Elle doit donc renvoyer 0 ou 1 si elle est occupée ou libre, respectivement.

Cette propriété doit être implantée dans tous les cas et pourra être utilisée par exemple par une entité amont qui peut conserver les PDUs (une file par exemple) afin d'éviter une perte dûe à une entité aval qui ne pourrait pas recevoir une nouvelle PDU (un lien en cours de transmission par exemple).

Un exemple d'utilisation est donné dans le premier " *Tutoriel du programmeur*" en sous-section 3.1.

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

Alors là tu vas me dire "qu'est-ce que c'est que ce chantier ? !". En fait, voilà comment c'est censé fonctionner (et ça a l'air de marcher !). Imaginons deux entités `A` et `B` de notre réseau, `B` étant en aval de `A` comme dans la figure **??**.

2.3.1 Transmission d'une PDU

Si, lors du traitement d'un événement, l'entité `A` doit faire suivre une PDU à `B`, elle le fera en invoquant la fonction processPDU associée à `B`. À partir de ce moment, la PDU est sous la responsabilité de `B`. Deux cas sont envisageables :

- Soit `B` est prête à traiter la `pdu` (c'est une file non pleine, ou un serveur inactif par exemple), alors il invoque immédiatement (dans le code de son processPDU) la fonction getPDU de `A` et tout va bien.
- Soit `B` n'est pas prête. Dans le code de son processPDU, il n'y aura donc pas d'invocation du getPDU de `A`, mais il devra y avoir une action permettant

de le faire plus tard [1] (dans le temps simulé !), par exemple positioner un booléen quelconque. Mais d'ici à ce que cette invocation arrive, peut-être `A` aura détruit la PDU (imagine que `A` modélise une couche physique, elle ne va pas retenir une PDU, enfin, sois raisonable !

2.3.2 Arrivée d'une PDU

D'un autre côté, si, lors du traitement d'un événement, l'entité `B` est prête à consommer une PDU, elle le fera en invoquant la fonction getPDU associée à `A`.

Là aussi, deux cas sont envisagables

- Soit `A` dispose effectivement d'une PDU à fournir à `B`, et dans ce cas là tout roule !
- Soit `A` n'a pas de PDU. Dans ce cas, la fonction getPDU va renvoyer un pointeur NULL qu'il vaut donc être prèt à traiter.

L'invocatio de la fonction getPDU de l'entité amont peut donc être déclanchée par un événement qui n'a rien a voir avec elle, c'est comme cela que l'on va aller chercher des PDUs sans y avoir été invité, d'où la note précédente, et d'où (entre autres) le risque du pointeur NULL.

2.3.3 Chronologie des événements

Tout cela peut paraître un peu vicieux, et tu dois avoir du mal, cher public, à voir quelle fonction utiliser quand ! Déjà, rassure-toi, tout ça n'a besoin d'être maîtrisé que par celui qui veut créer de nouveaux types d'entités pour le simulateur.

D'autre part, c'est en fait très simple, il suffit de suivre la logique des événements. Imaginons par exemple que tu veuilles modéliser un ordonnanceur *round-robin*. On va supposer qu'il est en aval d'un certain nombre de files, et en amont d'un serveur qui modélise un lien de communication. Voir la figure **??**.

schedRRProcessPDU À quoi va ressembler le code de la fonction de traitement d'une PDU de cet ordonnanceur ?

Cette fonction sera invoquée lorsque l'une des files amont aura une PDU à fournir à l'ordonnanceur. Mais cette PDU ne doit être ordonnancée que si les deux conditions suivantes sont vérifiées :

- le support (aval) est libre ;
- c'est au tour de cette file d'être servie ou les autres sont vides [2].

La fonction schedRRProcessPDU va donc devoir tester ces conditions et, si elles sont vraies, récupérer effectivement la PDU (avec le getPDU de la file) puis l'envoyer sur le lien (avec le processPDU du serveur).

Si les conditions ne sont pas vérifiées, alors elle laisse la PDU où elle est.

1. En fait, ce n'est pas tout à fait vrai ! Il faut être certain que `B` ira récupérer cette `pdu`, mais on peut s'arranger autrement comme on va le voir avec l'arrivée d'une PDU
2. Sinon il est non work conserving

schedRRGetPDU Et maintenant à quoi ressemble la fonction d'obtention d'une PDU de notre ordonnanceur ?

Elle est invoquée par le support de communication (aval) lorsqu'il est libre (un événement de fin de transmission de la précédente par exemple).

L'ordonnanceur ne dispose pas de PDU lui même. Il doit donc aller chercher dans les files amont la prochaine à servir et lui prendre une PDU par le biais de son getPDU. S'il ne trouve rien, il retourne un pointeur NULL à l'entité aval (le lien) qui ne fait donc rien.

Dans cette situation le prochain événement sera l'arrivée d'une PDU dans une file par l'invocation de son processPDU qui invoquera celui de l'ordonnanceur, ...

2.4 Plusieurs émetteurs vers le même destinataire

Comment faire lorsque plusieurs entités sont en amont d'une même entitée aval ? Par exemple plusieurs sources qui envoient des clients vers un unique serveur.

En fait, ce n'est pas fondamentalement un problème en soit, dans la mesure ou l'entité aval n'a en général pas à connaître l'entité amont. Du coup, s'il y en a deux ou plus, elle s'en moque. Oui, sauf que non ! Imagine une entité qui n'est pas toujours disposée à traiter une PDU entrante. En général, elle va stocker quelque part une information qui lui permettra de venir chercher la PDU quand elle le voudra bien. Si ce phénomène se produit plusieurs fois avant que l'entité soit disposée à enfin traiter les PDUs en attente, seule la dernière risque d'avoir été mémorisée !

Du coup, on pourrait être tenté de faire une liste chaînée de ces événements, de sorte à n'en rater aucun. Malheureusement, cela met nécessairement en place une politique FCFS, alors qu'on peut souhaiter implanter autre chose.

3 Extension du simulateur

3.1 Réalisation d'un ordonnanceur

À titre d'exemple, nous allons implanter un ordonnanceur *Round Robin*. Les fichiers source et `Makefile` sont dans le répertoire `tuto-prog-1`.

3.1.1 Caractéristiques de l'ordonnanceur

Passons sur les `include` et regardons comment caratériser un ordonnanceur :

```
#define SCHED_RR_NB_INPUT_MAX 8

struct rrSched_t {
   // La destination (typiquement un lien)
   void         * destination;
   processPDU_t   destProcessPDU;
```

```
   // Les sources (files d'entrée)
   int        nbSources;
   void     * sources[SCHED_RR_NB_INPUT_MAX];
   getPDU_t   srcGetPDU[SCHED_RR_NB_INPUT_MAX];

   // La dernière source servie par le tourniquet
   int lastServed;
};
```

Il nous faut connaître les entités amont, puisque nous voulons les traiter séparément !

La fonction de création est assez simple :

```
struct rrSched_t * rrSched_create(void * destination,
  processPDU_t destProcessPDU)
{
   struct rrSched_t * result = (struct rrSched_t * )sim_malloc(sizeof(struct rrSched_t));

   // Gestion de la destination
   result->destination = destination;
   result->destProcessPDU = destProcessPDU;

   // Pas de source définie
   result->nbSources = 0;

   // On commence quelquepart ...
   result->lastServed = 0;

   return result;
}
```

On gère la destination, comme toute entité qui peut fournir des PSUs doit le faire, et on s'occupe de ses spécificités.

Une source sera par exemple attribuée de la façon suivante :

```
void rrSched_addSource(struct rrSched_t * sched,
       void * source,
       getPDU_t getPDU)
{
   assert(sched->nbSources < SCHED_RR_NB_INPUT_MAX);

   sched->sources[sched->nbSources] = source;
   sched->srcGetPDU[sched->nbSources++] = getPDU;
}
```

Attention, ce n'est pas très robuste ! Mais ce n'est pas l'onjectif ici, ...

3.1.2 La fonction `rrSched_getPDU`

C'est cette fonction qui est invoquée par l'entité aval pour obtenir une PDU. C'est dans cette fonction que nous allons donc implanter l'algorithme d'ordonnancement.

En voici le code

```
struct PDU_t * rrSched_getPDU(void * s)
{
   struct rrSched_t * sched = (struct rrSched_t * )s;
   struct PDU_t * result = NULL;

   assert(sched->nbSources > 0);

   int next = sched->lastServed;

   // Quelle est la prochaine source à servir ?
   do {
      // On cherche depuis la prochaine la première source qui a des
      // choses à nous donner
      next = (next + 1)%sched->nbSources;
      result = sched->srcGetPDU[next](sched->sources[next]);
   } while ((result == NULL) && (next != sched->lastServed));

   if (result)
     sched->lastServed = next;
   return result;
}
```

En fait, il n'y a pas grand-chose à en dire ! On applique l'algorithme d'ordonnancement, et on fournit une PDU, éventuellement `NULL` s'il n'y a rien à ordonnancer.

3.1.3 La fonction `rrSched_processPDU`

Plus rigolo, passons à la fonction `rrSched_processPDU` qui sera invoquée par une source qui dispose d'une PDU et qui souhaite la faire passer à l'ordonnanceur.

Celui-ci devra la traiter s'il le peut, mais ne pas la prendre s'il ne peut pas la traiter. Voici le début de la fonction

```
int rrSched_processPDU(void *s,
       getPDU_t getPDU,
       void * source)
{
   int result;
   struct rrSched_t * sched = (struct rrSched_t *)s;

   printf_debug(DEBUG_SCHED, "in\n");
```

J'ai utiliser la fonction de débogage à titre d'exemple. Il y en a d'autres dans le fichier source. On se contente ici d'un *cast* pour respecter le prototype des fonctions d'échange.

La première chose à faire est de vérifier la disponibilité de l'entité aval :

```
// La destination est-elle prete ?
int destDispo = sched->destProcessPDU(sched->destination, NULL, NULL);
```

Maintenant nous allons traiter le cas du test (paramètres `NULL` pour tester notre disponibilité) :

```
// Si c'est un test de dispo, je dépend de l'aval
if ((getPDU == NULL) || (source == NULL)) {
   result = destDispo;
```

Et maintenant le cœur de la meule ! Si l'entité aval est prête, on lui dit de venir chercher une PDU (celle-là ou une autre, c'est l'algo qui le dira, mais on ne voit pas comment ça pourrait en être une autre !). Sinon, on laisse tomber, ...

```
} else {
   if (destDispo) {
      // Si l'aval est dispo, on lui dit de venir chercher une PDU, ce
      // qui déclanchera l'ordonnancement
      result = sched->destProcessPDU(sched->destination, rrSched_getPDU, sched);
   } else {
      // On ne fait rien si l'aval (un support a priori) n'est pas pret
      result = 0;
   }
}
```

On oublie pas de renvoyer le résultat !

```
   printf_debug(DEBUG_SCHED, "out\n");
   return result;
}
```

3.1.4 Utilisation

Ben ça c'est pas m'échant, je te laisse lire les sources !

4 Le moteur de simulation

4.1 Gestion mémoire

L'allocation de mémoire peut s'avérer problématique, aussi la fonction suivante a été ajoutée

```
void * sim_malloc(int l);
```

Elle présente l'avantage de vérifier le résultat de `malloc` (si les assertions sont activées) et de comptabiliser les appels afins de détecter une fuite mémoire.

4.2 Message de débogage

La fonction suivante est proposée

```
void printf_debug(int lvl, char * format, ...);
```

Elle s'utilise comme `printf` avec en plus un niveau de débogage (le premier paramètre). Un certain nombre de valeurs sont définies dans `motsim.h` parmi lesquelles

```
#define DEBUG_ALWAYS   0xFFFFFFFF
```

qui permet d'assurer que le message sera toujours affiché.

Les mécanismes de débogage sont activés par la définition de la macro `DEBUG_NDES`.

4.3 Gestion des événements

Elle est décrite dans `event.h` et codée dans `event.c`.

On peut créer un événement avec la fonction suivante

```
struct event_t * event_create(void (*run)(void *data),
      void * data,
      double date);
```

Attention, l'événement ne sera exécuté que si il est placé dans la file d'attente du simulateur.

On peut créer un événement et l'insérer immédiatement dans cette file grâce à la fonction

```
void event_add(void (*run)(void *data),
       void * data,
       double date);
```

5 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 également 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.

Pour chacun des objets décrit dans ce fabuleux document, je tâcherai de lister (dans une section"Les sondes") les différents points de mesure disponibles.

5.1 Les méthodes de base

5.1.1 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);
```

5.1.2 Consultation d'un échantillon

5.1.3 La moyenne

5.1.4 Les valeurs extrèmes

5.1.5 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);`

5.1.6 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.

5.2 Les différents types

5.2.1 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

5.2.2 La sonde exhaustive

5.2.3 L'histogramme

5.2.4 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);
```

5.2.5 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.

5.2.6 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 $m < -\alpha.m + (1 - \alpha).e$.

5.2.7 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);
```

6 Les PDUs

La PDU est l'objet de base échangé entre les entités du simulateur. Si on voit considère deux entités homologues qui dialoguent, il est effectivement légitime d'utiliser ce terme. Cependant, dans une simulation réseau faite avec NDES, la notion d'entité est bien plus vaste que cela si bien que ce nom est finalement mal choisi ! N'empèche que pour le moment, je vais le conserver !

Les PDUs sont définies dans le fichier `pdu.h` et implantées dans le fichier `pdu.c`. Une PDU est une structure extrèmement légère, dotée de quelques caractéristiques de base : un identifiant, une taille, une date de création (dans le temps simulé), et des données privées (un pointeur).

6.1 Création/destruction

On la crée avec la fonction

```
struct PDU_t * PDU_create(int size, void * private);
```

Le paramètre `size` est évidemment la taille en octets, et `private` peut être un pointeur vers des données privées associées à cette PDU (il peut être `NULL`, personnellement, je m'en fiche).

6.2 Les caractéristiques de base

La taille d'une PDU est donnée par

```
int PDU_size(struct PDU_t * PDU);
```

6.3 Les données privées

7 Les files de PDU

C'est l'outil de base pour stocker des objets selon une stratégie FIFO. Elles sont définies dans le fichier `file_pdu.h` et implantées dans le fichier `file_pdu.c`. Une file ne contient que des PDUs mais, comme on l'a vu, les PDUs peuvent être utilisées pour véhiculer tout et n'importe quoi grâce à leurs données privées. Il est donc simple de construire des files de n'importe quoi !

7.1 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 au travers de la fonction destProcessPDU passée en paramètre.

7.2 Insertion

```
void filePDU_insert(struct filePDU_t * file,
    struct PDU_t * PDU);
```

7.3 Extraction

```
struct PDU_t * filePDU_extract(struct filePDU_t * file);
```

7.4 Gestion de la taille

```
int filePDU_length(struct filePDU_t * file);
```

7.5 Les sondes

Les files sont dotées des points de mesure suivants

`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.

`DropSize`

`Sejourn`

On peut ajouter une sonde sur chacun de ces points de mesure avec une fonction de la forme

```
void filePDU_add<measure>Probe(struct filePDU_t * file,
                               struct probe_t   * probe);
```

8 Les générateurs aléatoires

Dans un simulateur, la génération de nombres aléatoires est un élément important. Dans NDES, on prend ces choses-là très au sérieux. Du coup, la gestion des nombres aléatoires est une horreur sans nom ! J'avoue avoir moi-même du mal à comprendre. La bonne nouvelle c'est que du coup le résultat est vraiment ... aléatoire.

8.1 Caractéristiques d'un générateur aléatoire

Un générateur aléatoire est caractérisé par trois composantes fondamentales

Le type des données générées peut être réel, entier, discret, ...

La loi peut être uniforme, exponentielle, ...

La source permet de déterminer la qualité de l'aléa, et par exemple de le rendre déterministe (afin d'obtenir la même séquence sur plusieurs simulation).

Attention, ces trois composantes sont éventuellement constitutées d'un jeu de paramètres.

Le principe général de génération d'une valeur est le suivant.

- Un nombre aléatoire est fourni par la source. Ce sera un entier entre deux valeurs extrèmes, par exemple, en fonction de la nature de la source.
- Une transformation est appliquée afin de respecter la densité de probabilité de la loi.
- Une seconde transformation est appliquée pour obtenir une valeur du type voulu.

Bref, tout est fait pour laisser sa place au hasard ...

8.2 Utilisation

Le schéma général d'utilisation est simple : on crée un générateur, on l'initialise avec le type de données voulu, on lui associe une distribution, éventuellement on peut changer la source d'aléa sur laquelle il se fonde, puis on peut lui extirper des valeurs aléatoires et enfin on le détruit sans un mot de remerciement.

Observons en détail les fonctions utiles à ce programme alléchant.

8.3 Création et destruction

Il existe au moins une fonction de création pour chaque type de données géré (il manquerait plus que ça !). Mais il existe également parfois des fonctions permettant de spécifier en même temps la distribution à utiliser. Voyons ça type par type.

8.3.1 Les entiers non signés

La fonction de création de base est

```
struct randomGenerator_t * randomGenerator_createUInt();
```

8.3.2 Les entiers non signés entre `min` et `max` inclus

Ça c'est sympa pour jouer aux dés. On les crée avec

```
struct randomGenerator_t * randomGenerator_createUIntRange(unsigned int min,
      unsigned int max);
```

8.3.3 Une liste d'entiers non signés

Pratique pour tirer au hasard des tailles de paquets !

```
struct randomGenerator_t * randomGenerator_createUIntDiscrete(int nbValues,
      unsigned int * values);
```

Le premier paramètre donne le nombre de valeurs, et le second est un tableau qui contient (au moins) ces valeurs. Son contenu sera recopié, donc si tu veux le détruire/modifier ensuite, vis ta vie !

Comme on se doute bien que dans ce genre de situations on va vouloir associer une probabilité à chaque valeur, on peut utiliser la version suivante :

```
struct randomGenerator_t * randomGenerator_createUIntDiscreteProba(
int nbValues,
unsigned int * values,
double * proba);
```

8.3.4 Des réels double précision

On crée un tel générateur avec la fonction suivante

```
struct randomGenerator_t * randomGenerator_createDouble();
```

On peut également en créer un fondé sur une distribution exponentielle de paramètre `lambda` :

```
struct randomGenerator_t * randomGenerator_createDoubleExp(double lambda);
```

8.3.5 Des réels double précision entre `min` et `max`

8.3.6 Une liste de réels double précision

Pour générer des nombres aléatoires choisis dans une liste fournie en paramètre :

```
struct randomGenerator_t * randomGenerator_createDoubleDiscrete(
                                    int nbValues,
                                    double * values);
```

ou, en fournissant directement les probabilités :

```
struct randomGenerator_t * randomGenerator_createDoubleDiscreteProba(
                                    int nbValues,
                                    double * values,
                                    double * proba);
```

Les paramètres sont analogues à la version fondée sur des entiers non signés, bref, voir ci-dessus.

8.3.7 Destruction

On détruit un générateur grâce à la fonction

```
void randomGenerator_delete(struct randomGenerator_t * rg);
```

8.4 Choix de la loi

Avant de pouvoir être utilisé, un générateur aléatoire doit être caractérisé par la loi qui le gouverne. Il existe des fonctions spécifiques à cet objectif. Certaines fonctions de création font appel à l'une ou l'autre de ces fonctions, mais pas toute ! Attention donc à s'assurer qu'une distribution est associée à un générateur avant de l'utiliser.

8.4.1 Loi uniforme

On spécifie une loi uniforme grâce à la fonction suivante

```
void randomGenerator_setDistributionUniform(struct randomGenerator_t * rg);
```

Attention, le type des données peut être un intervalle borné "continu" ou un ensemble discret, mais s'il s'agit d'un intervalle non borné, le résultat est ... aléatoire.

8.4.2 Loi explicite

Je ne sais pas trop comment l'appeler celle-là ! L'idée est qu'on fournit explicitement toutes les probabilités. Elle est spécifiée par la fonction suivante :

```
void randomGenerator_setDistributionDiscrete(struct randomGenerator_t * rg,
     int nb,
                                                  double * proba);
```

Attention, elle ne s'applique de toute évidence qu'à des données discrètes !

Les probabilités sont copiées par la fonctions donc le pointeur `proba` peut être libéré ensuite.

8.5 Choix de la source

8.6 Génération d'une valeur

Une nouvelle valeur est obtenue à chaque appel d'une des fonctions suivantes (à choisir en fonction du type attendu)

```
unsigned int randomGenerator_getNextUInt(struct randomGenerator_t * rg);
double randomGenerator_getNextDouble(struct randomGenerator_t * rg);
```

8.7 Les sondes

9 Les générateurs de dates

10 Le serveur générique

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

11 Le puits

12 L'affichage par GnuPlot

13 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.

14 Des exemples

14.1 Utilisation des sondes

14.1.1 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"

15 La librairie

15.1 Un lien unidirectionnel

15.2 Un lien DVB-S2

15.3 Un ordonnanceur ACM

16 Index des fonctions

Index