NS-2: Principes de conception et d'utilisation

par P. Anelli & E. Horlait

Version 1.3

Sommaire

Avant propos

NS est un outil logiciel de simulation de réseaux informatiques. Il est principalement bâti avec les idées de la conception par objets, de réutilisabilité du code et de modularité. Il est devenu aujourd'hui un standard de référence en ce domaine. C'est un logiciel dans le domaine public disponible sur l'Internet. Son utilisation est gratuite. Le logiciel est exécutable tant sous Unix que sous Windows.

Ce document ne vise pas à remplacer la documentation de NS mais plutôt de présenter cette application au lecteur avec un éclairage différent et plus général. Les explications s'appliquent à la version 2.1b4.

Conventions

nom{}	signifie une fonction ou méthode OTcl. Elle se nomme dans ce langage des "instance procédure".
nom()	signifie une fonction ou méthode C++.
"_"	Dans le code, ce caractère est appliqué comme suffixe au nom des variables; il indique une variable d'instance.

Avis aux lecteurs

Ce manuel a été réalisé avec soin pourtant, en utilisant NS, vous découvrirez peut-être des modifications, des erreurs qui se sont glissées dans nos descriptions, des phrases un peu obscures, des particularités auxquelles nous n'avons pas donné l'importance qu'elles méritent. Vous aurez probablement des suggestions à nous faire, des reproches à formuler. Faites nous part de vos expériences, de vos conseils pour que d'autres lecteurs ne commettent pas les mêmes erreurs. Aidez nous à mettre ce manuel à jour. Notre adresse:

UPMC - LIP 6 : Laboratoire d'Informatique de Paris VI
8, Rue du Capitaine Scott
75015 PARIS FRANCE
E-mail: `Pascal.Anelli@lip6.fr` URL : `http://www.lip6.fr/rp/~pan`

I. Présentation

1. Motivations

Dans [1], les auteurs donnent les principales raisons qui rendent la simulation de l'Internet difficile. Elles peuvent se résumer par la grande hétérogénéité et l'évolution rapide. L'hétérogénéité commence avec les liens, comprend les protocoles et finit avec les trafics d'applications diverses et variées. L'Internet évolue rapidement en terme de taille mais également en terme d'architecture et de topologie. C'est devant ce constat que des chercheurs ont essayé d'élever l'état de l'art de la simulation de l'Internet. Le projet VINT (`http://netweb.usc.edu/vint`) est une collaboration entre USC/ISI, Xerox parc, LBNL et UCB. Il a pour objectif principal de construire un simulateur multi-protocoles pour faciliter l'étude de l'interaction entre les protocoles et le comportement d'un réseau à différentes échelles. Le projet contient des bibliothèques pour la génération de topologies réseaux, des trafics ainsi que des outils de visualisation tel que l'animateur réseau nam (network animator). L'étude des comportements à des échelles différentes d'un réseau n'est pas obtenue par la simulation parallèle (qui peut être utile pour accélérer la simulation) mais par l'utilisation de techniques d'abstraction appliquées à différents éléments de la simulation. VINT est un projet en cours qui développe le simulateur NS.

Le simulateur NS actuel est particulièrement bien adapté aux réseaux à commutation de paquets et à la réalisation de simulations de petite taille. Il contient les fonctionnalités nécessaires à l'étude des algorithmes de routage unipoint ou multipoint, des protocoles de transport, de session, de réservation, des services intégrés, des protocoles d'application comme HTTP. De plus le simulateur possède déjà une palette de systèmes de transmission (couche 1 de l'architecture TCP/IP), d'ordonnanceurs et de politiques de gestion de files d'attente pour effectuer des études de contrôle de congestion. La liste des principaux composants actuellement disponible dans NS par catégorie est:

Application	Web, ftp, telnet, générateur de trafic (CBR, ...)
Transport	TCP, UDP, RTP, SRM
Routage	Statique, dynamique (vecteur distance) et routage multipoint (DVMRP, PIM)
Gestion de file d'attente	RED, DropTail, Token bucket
Discipline de service	CBQ, SFQ, DRR, Fair queueing
Système de transmission	CSMA/CD, CSMA/CA, lien point à point

Prises ensembles, ces capacités ouvrent le champ à l'étude de nouveaux mécanismes au niveau des différentes couches de l'architecture réseau. NS est devenu l'outil de référence pour les chercheurs du domaine. Ils peuvent ainsi partager leurs efforts et échanger leurs résultats de simulations. Cette façon de faire se concrétise aujourd'hui par l'envoi dans certaines listes de diffusion électronique de scripts de simulations NS pour illustrer les points de vue.

2. Documentation et sites essentiels

Le site de base de NS est maintenu par le projet VINT à l'URL `http://www-mash.CS.Berkeley.EDU/ns/`. On y trouve des liens et les informations principales sur NS. Les principales documentations sont:

- Le document [3] constitue la référence de NS qu'il est indispensable de posséder en complément à ce document. Il commente avec plus de détails les spécificités de NS.
- le tutoriel de Marc Greis. Il est accessible en ligne à l'URL `http://titan.cs.uni-bonn.de/~greis/ns/ns.html`. Il présente principalement les informations permettant de construire un modèle de simulation et montre comment utiliser l'interpréteur. Il n'y a que peu d'explications sur la création de nouveaux composants et sur le fonctionnement du simulateur. A noter que sur ce site, une page de liens est dédiée à la localisation de la documentation de NS, Tcl, C++.

Toutes les informations que l'on peut trouver sur NS sont en miroir sur le site `ftp://rp.lip6.fr/rp/pub/documentaire/Simulateur`.

Deux listes de diffusions traitent exclusivement de NS: `ns-users@mash.cs.berkeley.edu` et `ns-announce@mash.cs.berkeley.edu`. Pour s'inscrire, il suffit d'envoyer une demande à `majordomo@mash.cs.berkeley.edu`.

3. Notions pour l'utilisation de l'interpréteur

Dans ce paragraphe nous allons donner les bases du langage Tcl, les principes de l'OTcl et les explications sur le mécanisme qui permet à un programme C d'utiliser un interpréteur Tcl. La référence dans ce domaine est l'ouvrage [2] dont une deuxième édition est maintenant disponible. Pour OTcl, la page à consulter est localisée à l'URL `ftp://ftp.tns.lcs.mit.edu/pub/otcl/doc/tutorial.html`

3.1. Tcl

3.1.1. Présentation

Tcl est un langage de commande comme le shell UNIX mais qui sert à contrôler les applications. Son nom signifie *Tool Command Language*. Tcl offre des structures de programmation telles que les boucles, les procédures ou les notions de variables. Il y a deux principales façons de se servir de Tcl: comme un langage autonome interprété ou comme une interface applicative d'un programme classique écrit en C ou C++. En pratique, l'interpréteur Tcl se présente sous la forme d'une bibliothèque de procédures C qui peut être facilement incorporée dans une application. Cette application peut alors utiliser les fonctions standards du langage Tcl mais également ajouter des commandes à l'interpréteur.

3.1.2. Lancement

Toutes les applications qui utilisent Tcl créent et utilisent un interpréteur Tcl. Cet interpréteur est le point d'entrée standard de la bibliothèque. L'application `tclsh` constitue une application minimale ayant pour but de familiariser un utilisateur au langage Tcl. Elle ne comporte que l'interpréteur Tcl. On retrouve cet interpréteur dans l'application NS. Lorsque l'on tape ns, une série d'initialisations est faite puis l'application passe en mode interactif. On peut alors entrer les commandes Tcl.

3.1.3. Concepts de base

Chaque commande consiste en un ou plusieurs mots séparés par des espaces ou des tabulations. Le langage n'est pas typé. Tous les mots sont des chaînes de caractères. Le premier mot de la commande est le nom de la commande, les autres mots sont les arguments passés à la commande. Chaque commande Tcl retourne le résultat sous forme d'une chaîne de caractères. Le caractère de retour à la ligne termine une commande et lance son interprétation. Le caractère de séparation de commandes sur une même ligne est ";". Une fois la commande exécutée et terminée, l'application retourne en mode interactif c'est à dire que l'interpréteur est de nouveau prêt à recevoir une nouvelle commande.

Tcl n'est pas un langage compilé, c'est un langage interprété. Il n'y a pas de grammaire fixe qui traite l'ensemble du langage. Tcl est défini par un interpréteur qui identifie les commandes par des règles d'analyse syntaxique. Seules les règles d'analyse des commandes sont fixées. Tcl évalue une commande en deux étapes: analyse syntaxique et exécution. L'analyse syntaxique consiste à identifier les mots et effectuer les substitutions. Durant cette étape, l'interpréteur ne fait que des manipulations de chaînes. Il ne traite pas la signification des mots. Pendant la phase d'exécution, l'aspect sémantique des mots est traité comme par exemple déduire du premier mot le nom de la commande, vérifier si la commande existe et appeler la procédure de cette commande avec les arguments.

3.1.4. Substitutions

Substitution de variable : `$<variable>`
Le contenu d'une variable est obtenu en faisant précéder le nom de variable par le symbole $

```
>set a 20
>expr $a*2
< 40
```

Substitution de commande : `[<commande>]`
La substitution de commande permet de remplacer la commande par son résultat. Les caractères entre les crochets doivent constituer un script Tcl valide. Le script peut contenir un nombre quelconque de commandes séparées par des retour à la ligne et des ";".

Anti-slash substitution: "\"
Il est utilisé devant les caractères retour à la ligne, $, [pour indiquer de les interpréter comme des caractères ordinaires.

3.1.5. Inhibitions

Il existe plusieurs façons d'empêcher l'analyseur syntaxique de donner une interprétation spéciale aux caractères tel que $ ou ;. Ces techniques se nomment inhibitions. Tcl fournit deux formes d'inhibitions:

- double quote (") inhibe les séparations de mots et de commandes. Il reste les substitutions de variable et de commande. Les caractères espaces, tabs, retour à la ligne et point-virgule sont traités comme des caractères ordinaires.

```
>set msg " Aujourd hui: $date "
>set msg
<Aujourd hui: 22 Décembre
```

- accolade inhibe toutes les substitutions et tous les caractères spéciaux. Les caractères compris entre les crochet sont considérés comme un mot.

```
>set msg {Aujourd hui: $date}
>set msg
<Aujourd hui: $date
```

3.1.6. Conclusion

Dans cette section nous venons de donner les bases pour la lecture des scripts écrits en Tcl. Les différentes commandes de Tcl sont détaillées dans l'ouvrage [2] et dans la section 5 du manuel en ligne (man) d'UNIX.

3.2. OTcl

La documentation est accessible à l'URL `ftp://ftp.tns.lcs.mit.edu/pub/otcl/`. OTcl est une extension orientée objet de Tcl. Les commandes Tcl sont appelées pour un objet. En OTcl, les classes sont également des objets avec des possibilités d'héritage. Les analogies et les différences avec C++ sont:

- C++ a une unique déclaration de classe. En OTcl, les méthodes sont attachées à un objet ou à une classe.
- Les méthodes OTcl sont toujours appelées avec l'objet en préfixe
- L'équivalent du constructeur et destructeur C++ en OTcl sont les méthodes `init{}`/`destroy{}`
- L'identification de l'objet lui-même: `this` (C++), `$self` (OTcl). `$self` s'utilise à l'intérieur d'une méthode pour référencer l'objet lui-même. A la différence de C++, il faut toujours utiliser `$self` pour appeler une autre méthode sur le même objet. C'est à dire "`$self xyz 5`" serait "`this->xyz(5)`" ou juste "`xyz(5)`" en C++.
- Les méthodes OTcl sont tout le temps "virtual". A savoir, la détermination de la méthode à appeler est effectuée à l'exécution.
- En C++ les méthodes non définies dans la classe sont appelées explicitement avec l'opérateur de portée "::". En OTcl, les méthodes sont implicitement appelées dans l'arbre d'héritage par "`$self next`". Cette commande appelle la méthode de même nom de la classe parent.
- L'héritage multiple est possible dans les deux langages.

La définition d'une classe commence par la directive `Class`. Les fonctions et les attributs d'une classe s'installent par la suite par les commandes `instvar` et `insproc`. L'utilisation `instproc` définit les méthodes de la classe de manière assez semblable à C++. Par exemple écrire en C++ "`void XTP::send(char* data)`" s'écrit en OTcl "`XTP instproc send {data}`". Lors d'une instanciation en OTcl, le constructeur de la classe appelle `init{}` pour les initialisations et les traitements propres à l'objet. La méthode `init{}` est optionnelle, c'est à dire que si l'instanciation ne nécessite aucune initialisation il n'est pas nécessaire de la définir.
Un attribut est une variable qui sera instanciée autant de fois que la classe comportera d'instances. On peut l'appeler également variable d'instance. La valeur de l'attribut appartient à l'objet. La commande `instvar` est utilisée dans une méthode pour mettre en correspondance une variable d'instance avec une variable locale à la méthode. Il n'est pas nécessaire de faire une déclaration préalable au niveau de la classe. Une déclaration d'une variable d'instance se fait à l'intérieur d'une méthode membre. Par exemple, pour déclarer la variable d'instance, il suffit d'écrire "`$self instvar x`". Le schéma général de déclaration d'une classe est le suivant:

```
Class <Nom de classe>
<Nom de classe> instproc init {args} {
# constructeur
...
}

<Nom de classe> instproc methode1 {args} {
$self instvar variable1                # variable membre de la classe
...
}
etc.
```

En NS, les objets sont créés par la commande `new{}` qui retourne une référence sur l'objet créé. Par exemple, la création d'une instance de Simulator est effectuée par la commande suivante:

```
set ns [new Simulator]
```

Les objets topologiques (noeud et lien) ont une syntaxe de création quelque peu différente. On n'utilise pas la commande `new{}` mais une procédure de la classe Simulator. Dans les deux cas, la classe est uniquement OTcl et la référence des instances de ces classes est mémorisée dans un tableau de la classe Simulator. Par exemple la création d'un noeud est effectuée avec la syntaxe suivante:

```
$ns node
```

Une fois créés, les objets sont manipulables avec les méthodes dont ils disposent. Ces méthodes proviennent de leur classe et de leurs super-classes. L'appel d'une méthode est effectué via l'objet. Le nom de la méthode est le premier argument qui suit la référence de l'objet comme c'est le cas pour créer un noeud (cf exemple précédent). La méthode appelée est la première trouvée dans l'arbre d'héritage de l'objet. Les affectations sur les variables membres se font par la commande `set{}` et suivent le format général des commandes OTcl:

```
<référence instance> set <instance variable> <valeur>
```

L'héritage est indiqué par l'option `-superclass <nom de la super-classe>` après la directive Class. Par exemple la class1 hérite de la class2 et class3 en utilisant la syntaxe suivante:

```
Class class1 -superclass {class2 class3}
```

Lors des créations d'objets en NS, la première ligne est souvent `eval $self next $args`. La méthode `next` est utilisée pour appeler la méthode de même nom dans la hiérarchie de classe. Elle sert à effectuer une agrégation de méthode. Si aucune autre méthode de même nom n'est trouvée, une chaîne nulle est retournée sinon les arguments passés avec `next` sont passés à la méthode. Par la suite, nous reviendrons sur la création des objets.

Les détails les plus "croustillants" à propos de OTcl sont dans sa documentation. La maîtrise d'OTcl comme celle de C++ demande d'abord une bonne connaissance des concepts objets plutôt que celle de la syntaxe elle-même qui reste d'ailleurs très simple.

3.3. Liens C++ et Tcl

[2] explique en détail dans les chapitres 28 à 31 le fonctionnement de l'ajout d'une commande à Tcl. NS utilise la même mécanique. Pour faciliter les interactions entre le code interprété et C++, NS fournit la classe `Tcl`. La documentation de l'API C de OTcl se situe à l'URL `ftp://ftp.tns.lcs.mit.edu/pub/otcl/doc/capi.html` et sert surtout à comprendre comment le simulateur interagit avec l'interpréteur.

Construire une application avec un interpréteur Tcl revient à inclure une bibliothèque Tcl qui définit les commandes de bases de Tcl dans l'application. Comme nous l'avons dit, l'interpréteur effectue l'analyse syntaxique et appelle la fonction C correspondant à la commande Tcl. Ajouter une commande Tcl consiste à établir un lien entre un mot et une fonction C. Le mot sera le nom de la commande Tcl. La fonction C est définie dans le code source de l'application. Au démarrage, l'application procède dans son `main()` aux initialisations nécessaires et passe la main à l'interpréteur. L'application passe en mode interactif: à chaque commande tapée par l'utilisateur, la fonction C correspondante est appelée afin de réaliser la commande demandée.

II. Utilisation

NS est un outil de simulation de réseaux de données. Il est bâti autour d'un langage de programmation appelé Tcl dont il est une extension. Du point de vue de l'utilisateur, la mise en œuvre de ce simulateur se fait via une étape de programmation qui décrit la topologie du réseau et le comportement de ses composants, puis vient l'étape de simulation proprement dite et enfin l'interprétation des résultats. Cette dernière étape peut être prise en charge par un outil annexe, appelé nam qui permet une visualisation et une analyse des éléments simulés. Dans cette section, nous allons montrer comment utiliser le simulateur avec les composants disponibles dans la distribution.

1. Utilisation de composants standards

Nous allons commencer par un exemple simple qui illustre assez bien le comportement de ce système et permet d'en apprendre la terminologie élémentaire. Cet exemple est une simple simulation d'un réseau minimal constitué de deux stations communiquant l'une avec l'autre via une liaison spécialisée.

Dans la terminologie NS, ce que nous appelons machine s'appelle un nœud. Un nœud peut contenir des agents qui représentent des comportements, par exemple des applications. Une bibliothèque assez complète de composants existe de façon standard. Une propriété intéressante de ce système est son extensibilité. En effet, il est assez facile d'étendre la bibliothèque des comportements, des types de liens, ou de tout autre élément du système en programmant ses propres extensions qui deviennent alors intégrées au système.

Pour traiter ce premier cas, nous allons donc écrire un programme simple. En voici le contenu. Les commentaires (introduits par le caractère #) expliquent le rôle de chaque instruction ou partie de programme.

```
# création d'un simulateur
set ns [new Simulator]

# création du fichier de trace utilisé par le visualisateur
# et indication à ns de l'utiliser
set nf [open out.nam w]
$ns namtrace-all $nf

# lorsque la simulation sera terminée, cette procédure est appelée
# pour lancer automatiquement le visualisateur
proc finish {} {
        global ns nf
        $ns flush-trace
        close $nf
        exec nam out.nam &
        exit 0
}

# création de deux noeuds
```

```
set n0 [$ns node]
set n1 [$ns node]

# création d'une ligne de communication full duplex
# entre les noeuds n0 et n1
$ns duplex-link $n0 $n1 1Mb 10ms DropTail

# création d'un agent générateur de paquets à vitesse constante
# paquets de 500 octets, générés toutes les 5 ms
# implantation de cet agent dans le noeud n0
set cbr0 [new Agent/CBR]
$ns attach-agent $n0 $cbr0
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005


# création d'un agent vide, destiné à recevoir les paquets
# il est implanté dans n1
set null0 [new Agent/Null]
$ns attach-agent $n1 $null0

# le trafic issus de l'agent cbr0 est envoyé
# vers null0
$ns connect $cbr0 $null0

# scénario de début et de fin de génération des paquets par cbr0
$ns at 0.5 "$cbr0 start"
$ns at 4.5 "$cbr0 stop"

# la simulation va durer 5 secondes de temps simulé
$ns at 5.0 "finish"

# début de la simulation
$ns run
```

Lors de cette simulation, les résultats sont consignés dans un fichier de trace que l'outil de visualisation nam va permettre de traiter. Dans notre programme, l'outil de visualisation est appelé directement à la fin de la simulation. Deux éléments intéressants sont proposés à la visualisation : un dessin de la topologie du réseau étudié, et une visualisation dynamique du déroulement du programme dans le temps.

Nous verrons plus loin comment accéder à d'autres informations à partir de cette interface.

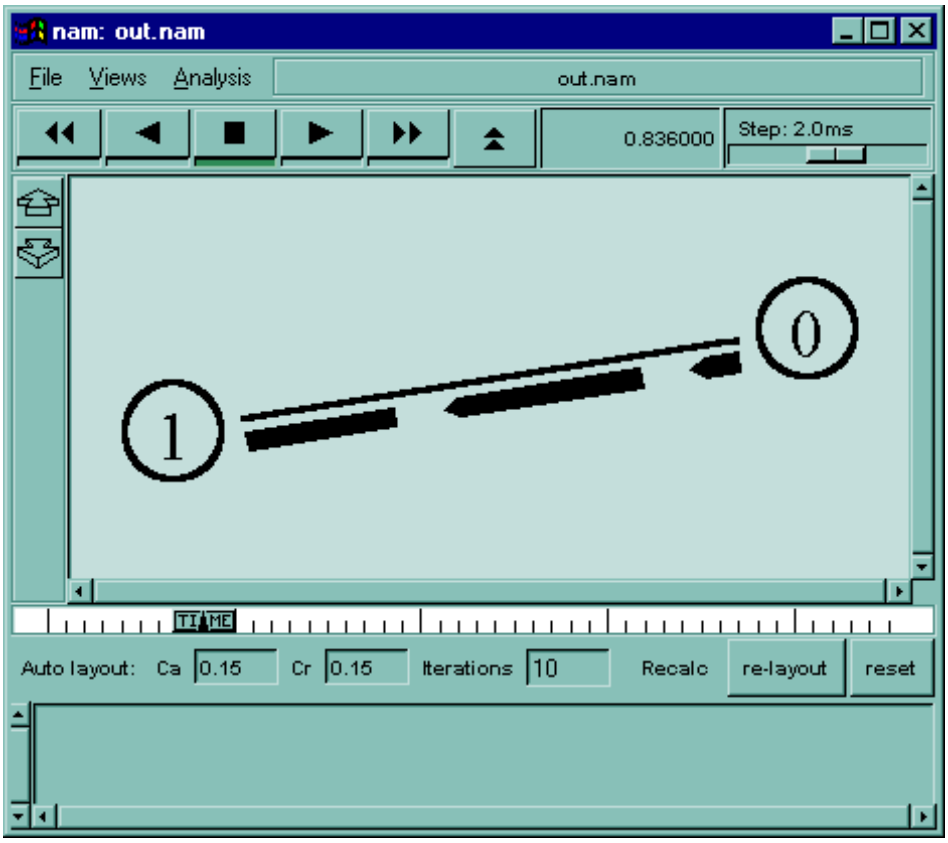


Figure 1: Exécution du premier exemple

Un deuxième exemple va montrer maintenant comment complexifier un peu la topologie du réseau obtenu et rendre plus visible les comportements en construisant un réseau de quatre nœuds avec deux sources de paquets. Lors de la visualisation, on souhaite que les paquets d'une source apparaissent en bleu, les autres en rouge. Voici le texte de ce nouveau programme et la visualisation qui en est obtenue. On remarque la visualisation de la file d'attente de messages qui se forme au nœud 2, puisque les trafics en entrée sont assez proches de la saturation.

```
# création d'un simulateur
set ns [new Simulator]

# création du fichier de trace utilisé par le visualisateur
# et indication à ns de l'utiliser
set nf [open out.nam w]
$ns namtrace-all $nf

# lorsque la simulation sera terminée, cette procédure est appelée
# pour lancer automatiquement le visualisateur
proc finish {} {
        global ns nf
        $ns flush-trace
        close $nf
        exec nam out.nam &
        exit 0
}

# création de quatrenoeuds
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
```

```
# création de lignes de communication full duplex
# entre les noeuds
$ns duplex-link $n0 $n2 1Mb 10ms DropTail
$ns duplex-link $n1 $n2 1Mb 10ms DropTail
$ns duplex-link $n3 $n2 1Mb 10ms DropTail

# création d'agents générateurs de paquets à vitesse constante
# paquets de 500 octets, générés toutes les 5 ms
# implantation de cet agent dans le noeud n0
set cbr0 [new Agent/CBR]
$ns attach-agent $n0 $cbr0
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 set fid_ 1
set cbr1 [new Agent/CBR]
$ns attach-agent $n1 $cbr1
$cbr1 set packetSize_ 500
$cbr1 set interval_ 0.005
$cbr1 set fid_ 2

# création d'un agent vide, destiné à recevoir les paquets
# il est implanté dans n1
set null0 [new Agent/Null]
$ns attach-agent $n3 $null0

# routage des trafics
$ns connect $cbr0 $null0
$ns connect $cbr1 $null0

# surveillance d'une file d'attente du noeud 2 vers le noeud 3
$ns duplex-link-op $n2 $n3 queuePos 0.5

# scénario de début et de fin de génération des paquets par cbr0
$ns at 0.5 "$cbr0 start"
$ns at 1.0 "$cbr1 start"
$ns at 4.0 "$cbr1 stop"
$ns at 4.5 "$cbr0 stop"

# paramétrage des couleurs d'affichage
$ns color 1 Blue
$ns color 2 Red
# la simulation va durer 5 secondes de temps simulé
$ns at 5.0 "finish"

# début de la simulation
$ns run
```

En exécutant ce programme (voir figure, on verra que la gestion de la file d'attente à l'entrée du lien 2 vers 3 n'est pas très équitable. Pour modifier cela, il suffit d'indiquer une gestion politique différente à cette file en modifiant la déclaration du lien en question par :

```
$ns duplex-link $n3 $n2 1Mb 10ms SFQ
```

La politique SFQ (Stochastic Fair Queuing) a pour conséquence de mieux équilibrer les pertes de paquets entre les différents flots de données.

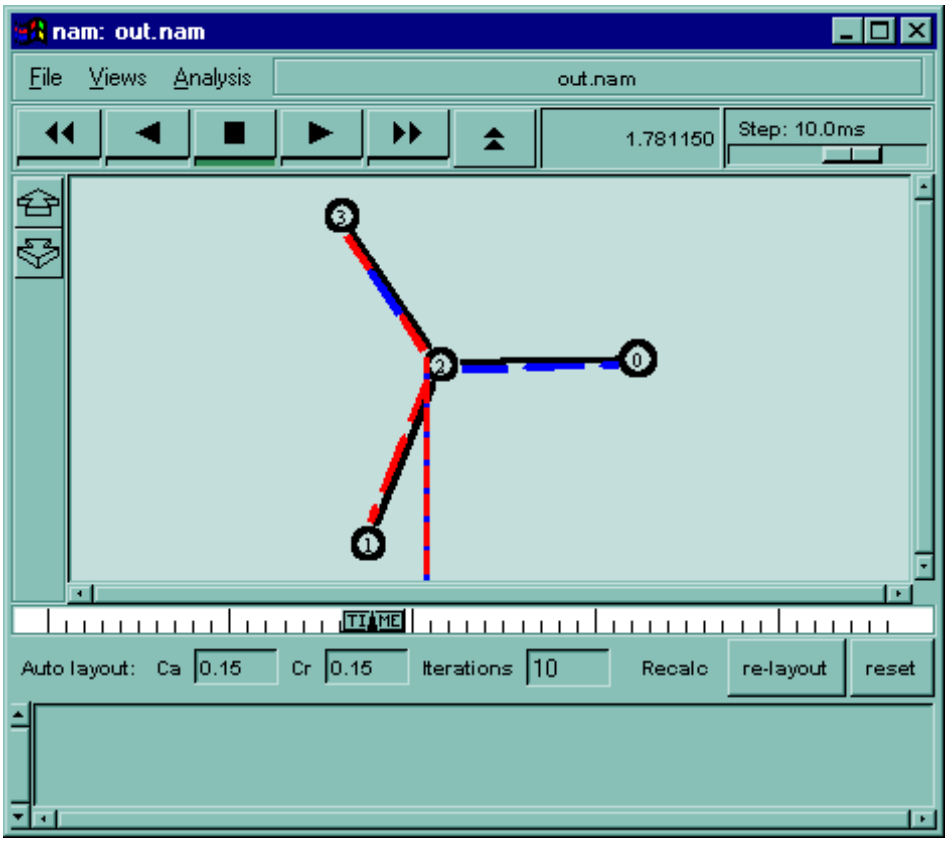


Figure 2: Exécution avec politique DropTail

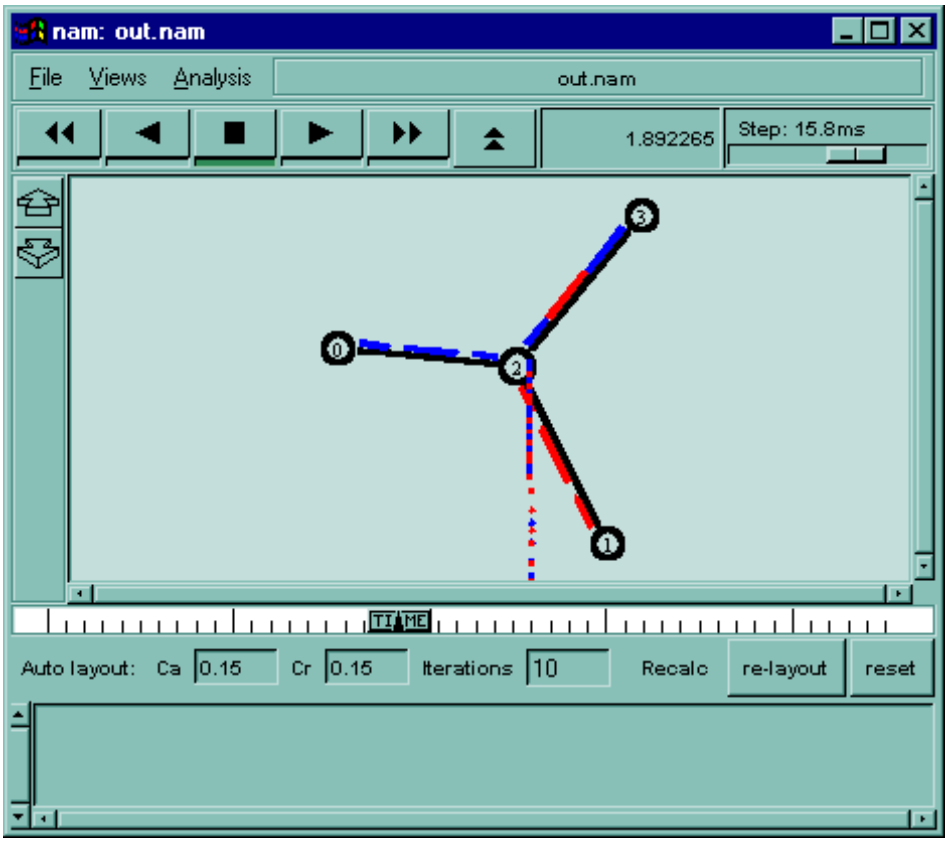


Figure 3: Exécution avec politique SFQ

Montrons maintenant comment créer des systèmes un peu plus complexes. Ce troisième exemple crée une topologie circulaire de réseau, génère un trafic et fait apparaître des pannes sur l'un des liens.

```
# création d'un simulateur
set ns [new Simulator]

# création du fichier de trace utilisé par le visualisateur
# et indication à ns de l'utiliser
set nf [open out.nam w]
$ns namtrace-all $nf

# lorsque la simulation sera terminée, cette procédure est appelée
# pour lancer automatiquement le visualisateur
proc finish {} {
        global ns nf
        $ns flush-trace
        close $nf
        exec nam out.nam &
        exit 0
}

# modèle de routage
$ns rtproto DV

# création de sept noeuds à l'aide d'une boucle
# les descripteurs de noeuds sont stockés dans un tableau
for {set i 0} {$i < 7} {incr i} {
  set n($i) [$ns node]
}

# création de lignes de communication full duplex
# entre les noeuds
for {set i 0} {$i < 7} {incr i} {
  $ns duplex-link $n($i) $n([expr ($i+1)%7]) 1Mb 10ms DropTail
}

# création d'agents générateurs de paquets à vitesse constante
# paquets de 500 octets, générés toutes les 5 ms
# implantation de cet agent dans le noeud n0
set cbr0 [new Agent/CBR]
$ns attach-agent $n(0) $cbr0
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005

# création d'un agent vide, destiné à recevoir les paquets
# il est implanté dans n1
set null0 [new Agent/Null]
$ns attach-agent $n(3) $null0

# routage des trafics
$ns connect $cbr0 $null0

# scénario de début et de fin de génération des paquets par cbr0
$ns at 0.5 "$cbr0 start"
$ns at 4.5 "$cbr0 stop"

# panne d'un lien
$ns rtmodel-at 1.0 down $n(1) $n(2)
$ns rtmodel-at 2.0 up $n(1) $n(2)

# la simulation va durer 5 secondes de temps simulé
$ns at 5.0 "finish"

# début de la simulation
$ns run
```

Nous avons introduit ici un protocole de routage explicite dans le système. Il s'agit simplement d'une méthode de type vecteur de distance qui modélise un protocole à la RIP. Le résultat obtenu graphiquement est donné à la Figure 4. La déclaration du type de routage est faite par l'instruction :

```
$ns rtproto DV
```

Les pannes qui entraînent un recalcul des routages sont, elles, déclenchées par les instructions :

```
$ns rtmodel-at 1.0 down $n(1) $n(2)
$ns rtmodel-at 2.0 up $n(1) $n(2)
```

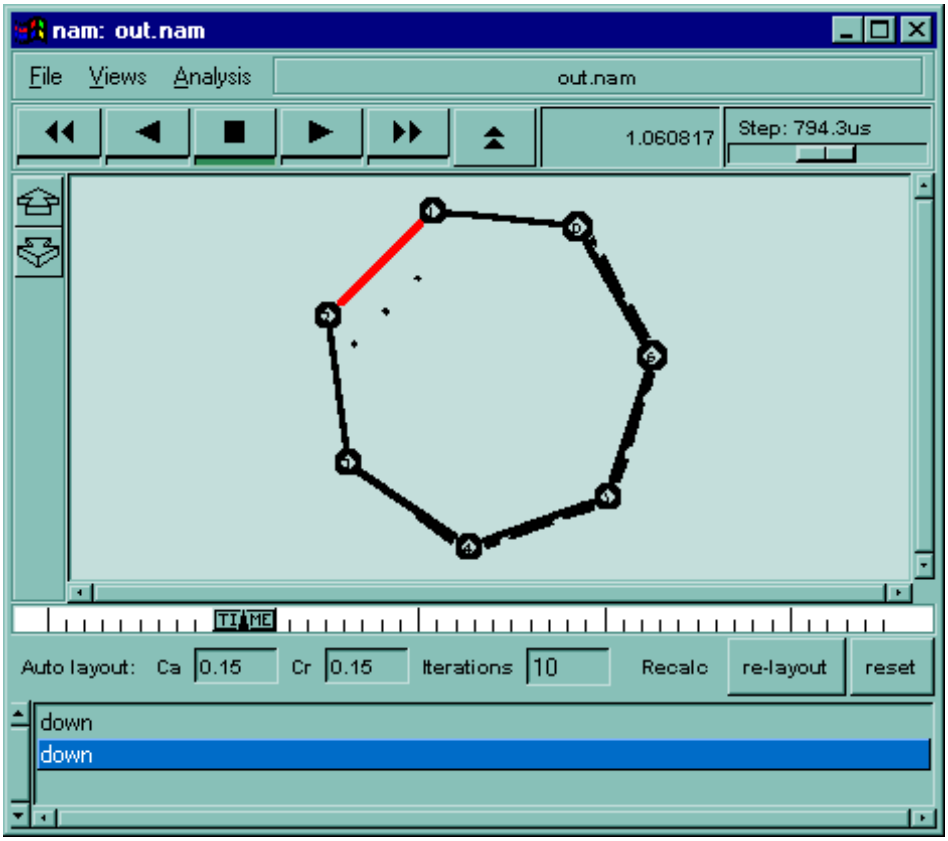


Figure 4: Panne et re-routage

2. Modification de composants existants

NS est en réalité un programme relativement complexe écrit en C++ et interfacé via Tcl. Pour modifier le comportement d'objets existants, il est donc nécessaire de modifier le code C++ qui en réalise l'implantation.

Pour illustrer cette façon de travailler, nous allons traiter un exemple simple. Nous nous proposons de faire des statistiques de trafic sur un échange de données utilisant TCP. Dans NS, une connexion TCP est modélisée par une source de paquets et un puits de données responsable, entre autres choses, de la génération des acquittements. La méthode choisie est la suivante :

1. Nous utiliserons le puits TCP pour faire notre modification. Pour cela, nous ajouterons dans chaque puits, une variable qui comptabilise le nombre d'octets reçus par le puits.
2. Ensuite, nous rendrons cette variable accessible via l'interface Tcl.

3. Enfin, lors de l'utilisation, une tâche périodique, écrite en Tcl, sera activée pour venir lire le contenu de cette variable et préparer le fichier de données correspondant.

2.1. Ajout d'une variable à la classe TcpSink

La comptabilisation des octets reçus se fera dans une variable entière de la classe TcpSink ; la déclaration de cette classe est faite dans le fichier TcpSink.h. La modification apportée est très simple : elle est présentée en grisé sur l'extrait de code ci-dessous.

```
class TcpSink : public Agent {
public:
     TcpSink(Acker*);
     void recv(Packet* pkt, Handler*);
     void reset();
     int command(int argc, const char*const* argv);
     TracedInt& maxsackblocks() { return max_sack_blocks_; }
protected:
     void ack(Packet*);
     virtual void add_to_ack(Packet* pkt);
     Acker* acker_;
     int ts_echo_bugfix_;
     /* for byte counting */
     int bytes_;

     friend void Sacker::configure(TcpSink*);
     TracedInt max_sack_blocks_; /* used only by sack sinks */

};
```

2.2. Utilisation de la nouvelle variable dans le simulateur

La classe TcpSink est maintenant munie d'une nouvelle variable que nous allons pouvoir utiliser. Deux choses principales restent à réaliser : l'initialisation correcte de cette variable dans le constructeur de la classe TcpSink et la mise à jour en cours de simulation de cette variable. Dans les deux cas, les modifications interviennent ici dans le code C++ implantant la classe TcpSink, c'est à dire dans le fichier TcpSink.cc.

L'initialisation de cette variable, `bytes_`, est faite dans le constructeur de TcpSink, comme le montre l'extrait de code suivant :

```
TcpSink::TcpSink(Acker* acker) : Agent(PT_ACK), acker_(acker)
{
     bind("packetSize_", &size_);
     bind("bytes_", &bytes_); // for statistics
     bytes_ = 0; // init
     bind("maxSackBlocks_", &max_sack_blocks_); // used only by sack
     bind_bool("ts_echo_bugfix_", &ts_echo_bugfix_);
}
```

La fonction bind permet de faire le lien entre une variable C++ et une variable (de même nom ici) accessible par l'interface Tcl. Lors de la construction d'un objet TcpSink, la variable est initialisée à 0.

La deuxième phase de la modification dans ce même fichier consiste à mettre à jour le contenu de cette variable en fonction du trafic reçu. Ce travail peut être facilement implanté dans la procédure de réception d'un paquet par un object TcpSink. La méthode à modifier est appelée `recv()`. L'extrait de code ci-dessous montre la modification réalisée.

```
void TcpSink::recv(Packet* pkt, Handler*)
{
     int numToDeliver;
     int prebytes; // tempo
     int numBytes = ((hdr_cmn*)pkt->access(off_cmn_))->size();
     hdr_tcp *th = hdr_tcp::access(pkt);
     acker_->update_ts(th->seqno(),th->ts());
          numToDeliver = acker_->update(th->seqno(), numBytes);
     if (numToDeliver)
          recvBytes(numToDeliver);
          ack(pkt);
     // statistics
     prebytes = th->seqno() + numBytes -1;
     if (prebytes > bytes_)
          bytes_ = prebytes;
     Packet::free(pkt);
}
```

La mise à jour de la variable `bytes_` intervient juste avant la libération du paquet de données. Elle marque la progression du nombre d'octets reçus en séquence par l'objet TcpSink (ce n'est pas nécessairement le nombre d'octets du paquet qui peut être dupliqué).

Ces modifications étant faites, il suffit de recompiler le code de NS, puisque les éléments sur lesquels nous venons de travailler en sont des composants.

2.3. Utilisation de la nouvelle variable dans l'interpréteur

L'exemple suivant illustre l'utilisation de cette nouvelle variable dans un environnement de type réseau local. Cette illustration montre également la mise en œuvre du composant " LAN " fourni par NS. Le code Tcl de cet exemple est reproduit ci-dessous.

```
# création d'un simulateur
set ns [new Simulator]
set bw bw1.dat

# création du fichier de trace
set f0 [open $bw w]

# lorsque la simulation sera terminée, cette procédure est appelée
# pour lancer automatiquement le visualisateur
proc finish {} {
        global ns f0
        close $f0
        exit 0
}
```

Après avoir créé le simulateur proprement dit, préparé le fichier de trace et défini la procédure de terminaison du simulateur, nous allons créer les nœuds du réseau et construire la topologie associée.

```
# création de  noeuds
set n0 [$ns node]
```

```
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
set n5 [$ns node]

# création d'un réseau local Ethernet
$ns make-lan "$n0 $n1 $n2 $n3" 10Mb 0.05ms LL Queue/DropTail Mac/Csma/Cd
# création d'un autre Ethernet avec passerelle entre les deux
$ns make-lan "$n3 $n4 $n5" 10Mb 0.05ms LL Queue/DropTail Mac/Csma/Cd
```

L'étape suivante consiste en la définition des trafics générés. Nous avons choisi de définir ici deux échanges : l'un CBR, l'autre de type transfert de fichier qui utilisera TCP et notre version modifiée de TcpSink.

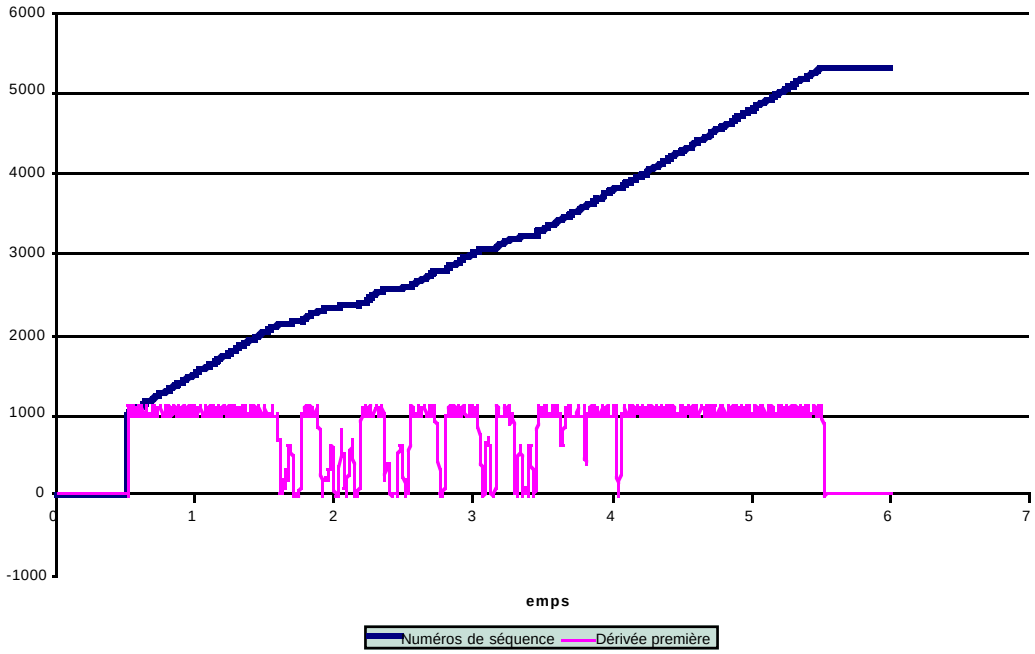
```
# création d'un agent générateur de paquets à vitesse constante
# paquets de 500 octets, générés toutes les 5 ms
# implantation de cet agent dans le noeud n0
set cbr0 [new Agent/CBR]
$ns attach-agent $n0 $cbr0
$cbr0 set packetSize_ 800
$cbr0 set interval_ 0.01ms

# création d'un agent vide, destiné à recevoir les paquets
# il est implanté dans n5
set null0 [new Agent/Null]
$ns attach-agent $n5 $null0

# le trafic issus de l'agent cbr0 est envoyé
# vers null0
$ns connect $cbr0 $null0

# une connexion TCP
set src [new Agent/TCP]
set dst [new Agent/TCPSink]
$ns attach-agent $n1 $src
$ns attach-agent $n5 $dst
$src set fid_ 2
# set tcp0 [$ns create-connection TCP $n1 TCPSink $n5 2]
set tcp0 [$ns connect $src $dst]
# $tcp0 set window_ 50
set ftp0 [new Application/FTP]
$ftp0 attach-agent $tcp0

#set sink [new Agent/LossMonitor]
#$ns attach-agent $n5 $sink
```

La topologie et les trafics sont définis. Il reste maintenant à créer le scénario d'exécution et les éléments de mesure. Pour cela, nous définissons une procédure record qui sera appelée régulièrement au cours de la simulation. C'est cette procédure qui accédera à la variable bytes_ que nous avons créée. Ici, la valeur de la variable est écrite dans un fichier en même temps que la date de la collecte de l'information. Ce fichier pourra ensuite être récupéré pour un traitement graphique ou numérique.

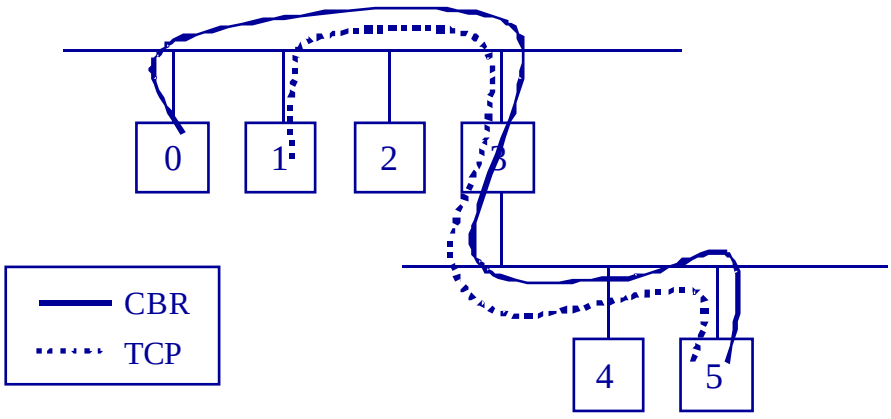
```
proc record {} {
  global f0  dst
  set ns [Simulator instance]
  set time 0.01
  set bw0 [$dst set bytes_]
  set now [$ns now]
  puts $f0 "$now $bw0"
```

```
  # $dst set bytes_ 0
  $ns at [expr $now+$time] "record"
}

# scénario de début et de fin de génération des paquets par cbr0
$ns at 0.0 "record"
$ns at 1.5 "$cbr0 start"
$ns at 0.5 "$ftp0 start"
$ns at 3.5 "$cbr0 stop"
$ns at 5.5 "$ftp0 stop"

# la simulation va durer 5 secondes de temps simulé
$ns at 6.0 "finish"

# début de la simulation
$ns run
```

L'exécution de ce programme génère un fichier de trace des numéros de séquence en fonction du temps. La courbe suivante représente la variation, dans le temps de ce numéro de séquence, ainsi que la dérivée de cette valeur (variation instantanée).



La topologie du réseau étudié est rappelée ci-après.



III. Développement de nouveaux composants

1. Principes de base

NS est un simulateur à événements discrets orienté objet. Il est écrit en C++ avec une interface textuelle (ou shell) qui utilise le langage OTcl (Object Tool Command Langage). L'OTcl est une extension objet au langage de commande Tcl. Le langage C++ sert à décrire le fonctionnement interne des composants de la simulation. Pour reprendre la terminologie objet, il sert à définir les classes. Quant au langage OTcl, il fournit un moyen flexible et puissant de contrôle de la simulation comme le déclenchement d'événements, la configuration du réseau, la collecte de statistiques, etc. L'application NS se compose de deux élèments fonctionnels: un interpréteur et un moteur de simulation. Au moyen de l'interpréteur l'utilisateur est capable de créer le modèle de simulation ce qui revient à assembler les différents composants nécessaires à l'étude. Les composants du modèle de simulation sont appelés objets ou encore instances de classe. Le moteur de simulation effectue les calculs applicables au modèle préalablement construit par l'utilisateur via l'interpréteur.

NS bénéficie de toutes les possibilités qu'offrent les techniques objets comme l'héritage, le polymorphisme, la surcharge, etc. L'héritage permet d'élaborer des arborescences de classes. Le modèle de simulation est construit à partir d'une arborescence de classes qui en fait se dédouble:

- une définie en OTcl dite arborescence interprétée. Elle est utilisée par l'interpréteur et est visible par l'utilisateur.
- une définie en C++ que l'on nommera compilée. Elle forme l'arborescence utilisée par le moteur de simulation (que l'on appellera par la suite simulateur). C'est l'ombre de l'arborescence interprétée.

Les deux arborescences sont très proches l'une de l'autre. Du point de vue de l'utilisateur, il y a une correspondance univoque entre une classe d'une arborescence et une classe de l'autre arborescence. La figure 1 montre la dualité des classes qui peut exister dans NS.

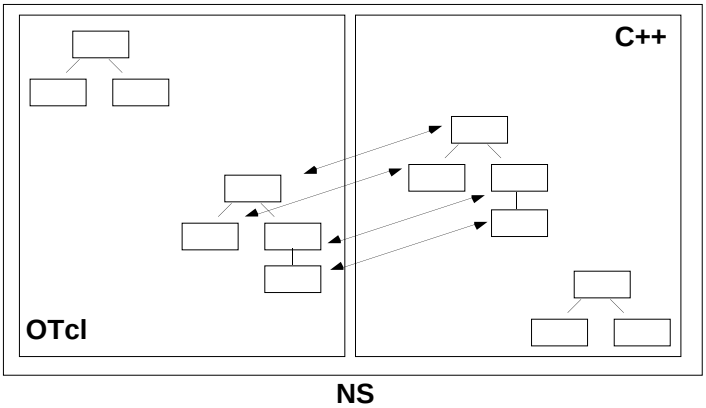


Figure 1: Dualité des classes OTcl et C++

Le principe général de la création des objets du modèle de simulation est le suivant: l'utilisateur crée un nouvel objet via l'interpréteur OTcl. Le nouvel objet interprété est cloné en un objet compilé correspondant dans le simulateur.

En réalité toutes les classes ne sont pas dans les deux arborescences de NS. On peut avoir des classes qui ne sont que dans l'interpréteur: elles servent à faire par exemple des assemblages (ou agrégations de classes) pour faciliter la manipulation. On parlera de classe OTcl dans la suite de ce document pour faire référence à ce type de classe. On peut avoir des classes qui sont purement dans le simulateur: elles ne sont pas visibles de l'utilisateur et servent au fonctionnement interne d'un composant comme par exemple certaines structures de données.

1.1. Arborescence des classes compilées

La figure 2 représente l'arborescence de classes utilisée par le simulateur. Les classes visibles au niveau de l'interpréteur comportent une déclaration dans la classe TclClass. Nous y reviendrons lorsque nous étudierons la création des objets. Nous verrons également l'objectif des différentes classes. Le nom des classes correspond à celui utilisé dans le code source.

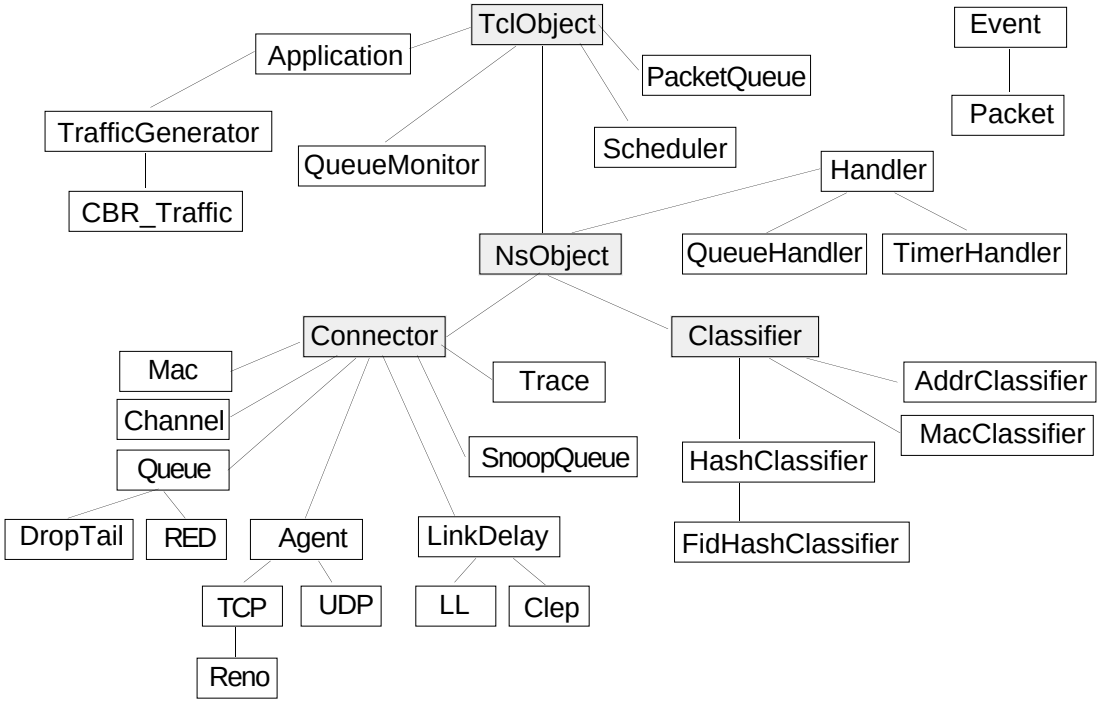


Figure 2: Arborescence de dérivation des classes C++ du simulateur

1.2. Arborescence des fichiers

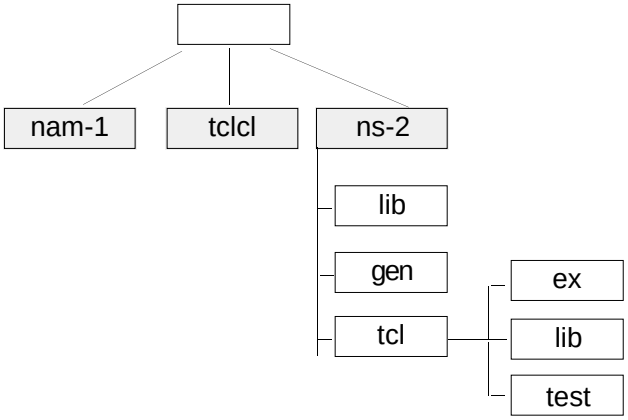


Figure 3: Arborescence des fichiers de la distribution NS.

La distribution de NS comprend principalement 3 répertoires:

- `ns-2`, l'application NS. Ce repertoire contient l'ensemble des fichiers .h et .cc de NS.
- `nam-1`, l'outil de visualisation des résultats de la simulation: l'animateur réseau.
- `tclcl`, sources du code assurant la liaison entre l'interpréteur et le simulateur. Citons l'un des principaux fichiers: `tcl-object.tcl`.

Dans le répertoire `ns-2`, on trouve les répertoires:

- `tcl` pour tous les codes interprétés.
- `bin` pour les utilitaires et les exécutables pour la réalisation du binaire ns-2
- `lib` pour la bibliothèque de libg++
- `gen` pour les sources générées lors de la réalisation du binaire ns-2 par le makefile.
- `test_output` pour les résultats des simulations.
- tous les fichiers .h et .cc des classes C++ du simulateur.

Le répertoire `tcl` contient les répertoires:

- `lib` pour les méthodes OTcl des classes de l'arborescence interprétée. Dans ce répertoire, les fichiers `ns_lib.tcl` et `ns_default.tcl` ont un rôle particulier. Le premier est interprété automatiquement au lancement de NS. Il contient l'ensemble du code interprété de NS via la commande Tcl "`source`" (lecture d'un fichier) des différents fichiers OTcl. Les valeurs par défaut affectées aux objets sont déclarées dans le second. Les autres fichiers de ce répertoire sont l'API OTcl de NS. Ils ont un nom préfixé par `ns_`.
- `ex` pour les scripts de simulation donnés à titre d'exemple.
- `test`, il recense l'ensemble des scripts des suites de tests pour la validation du simulateur. On peut également prendre le contenu de ce répertoire comme des exemples. Les autres répertoires contiennent les codes interprétés des contributions.

Pour un utilisateur qui souhaite écrire seulement des scripts, l'impression des commandes disponibles de l'interpréteur constitue une bon complément à la documentation existante. Quelle que soit l'étude à mener, les principaux fichiers sont:
`ns-lib.tcl, ns-node.tcl, ns-link.tcl, ns-agent.tcl, ns-default.tcl, ns-packet.tcl`

Pour l'utilisateur qui souhaite faire des développements de composants NS, les fichiers importants sont:

```
connector.{h,cc}, classifier.{h,cc}, packet.{h,cc}, timer-handler.{h,cc},
queue.{h,cc}, agent.{h,cc}, udp.{h,cc}, app.{h,cc}, trafgen.{h,cc}, trace.{h,cc}
```

2. Liaisons entre l'interpréteur et le simulateur

Comme nous pouvons le voir dans l'arborescence de dérivation des classes C++, la classe TclObject est la racine de toutes les autres classes à la fois dans l'arborescence compilée et interprétée. La classe TclObject constitue donc la classe de base de la plupart des autres classes. Tous les objets du modèle de simulation sont des instances de la classe TclObject. Cette classe sert aux objets dont la création est initiée par l'interpréteur. Elle possède les interfaces nécessaires aux liaisons entre les variables qui doivent être visibles à la fois en C++ et OTcl. Elle définit la fonction `command()` qui est très utile pour ajouter des commandes à l'interpréteur. La classe TclObject ainsi que les autres sources du répertoire tclcl sont partagés entre NS et le projet MASH de Berkeley. Dans la hiérarchie des classes OTcl, le nom de la classe racine se nomme autrement et porte le nom de SplitObject.
La classe NsObject est une sous-classe de la classe TclObject mais reste cependant une super-classe aux autres classes. La principale différence avec la classe TclObject tient à ce qu'elle est capable de recevoir des paquets et traiter des événements. Elle hérite de la classe Handler. Cette classe constitue la principale racine des objets dans NS. Elle contient les fonctions communes nécessaires au simulateur et définit la fonction virtuelle "`void recv (Packet *, Handler* callback =0)=0;`". Cette fonction est une fonction générique utilisée par tous les objets dérivés de NsObject pour recevoir un paquet. Elle est réellement définie par les sous-classes. Nous reviendrons sur la classe NsObject lorsque nous verrons l'ordonnanceur (scheduler).

L'utilisateur crée les objets de sa simulation via l'interpréteur. C'est à dire que ces objets existeront à la fois dans l'interpréteur et dans le simulateur. La création d'un objet est en fait l'instanciation de deux objets qui sont chacun au même niveau dans leur arborescence de classe. La classe TclClass sert à établir l'arborescence de classes OTcl en fonction de celle de C++. La classe TclClass n'a pas d'instance créée dynamiquement. Tous les objets de cette classe sont créés par déclaration d'une variable statique. Par exemple, pour un estimateur de débit par fenêtre temporelle, nous avons:

```
static class TimeWindow_EstClass : public TclClass {
public:
        TimeWindow_EstClass() : TclClass ("Est/TimeWindow") {}
        TclObject* create(int, const char*const*) {
                return (new TimeWindow_Est());
        }
}class_timewindow_est;
```

Le nom de la classe OTcl est `Est/TimeWindow`. Au lancement du simulateur toutes les variables de TclClass sont créées, le constructeur est appelé, dans notre exemple TimeWindow_EstClass (), le constructeur de la classe TclClass est lui-même appelé. Ce dernier a en charge de procéder à l'enregistrement de la classe en OTcl sous le nom donné en paramètre (ici `Est/TimeWindow`).

Pour que la hiérarchie des classes du simulateur soit reproduite dans l'interpréteur, NS utilise la convention suivante pour nommer les classes OTcl. Le caractère '/' est utilisé comme séparateur. Le nom de l'ascendant est à gauche, celui du descendant se situe à droite. Dans notre exemple: TimeWindow est la classe dérivée de la classe Est (estimateur). La déclaration de la classe OTcl `Est/TimeWindow` est faite en précisant que la classe parent est Est.

Ainsi grâce à la classe TclClass, nous avons l'arborescence de classes C++ reproduite en OTcl. Pourquoi faire cette copie? La réponse tient en un mot: héritage. Quand un objet est créé, il a les attributs de sa classe mais aussi ceux de ses ascendants. Les objets sont créés via l'interpréteur et sont d'abord des instances de classes OTcl. Pour que ces objets aient les mêmes attributs que leur clone dans le simulateur, nous devons avoir la même hiérarchie en OTcl et C++.

Nous allons maintenant étudier la création d'un objet. Le schéma général de la construction suit les enchaînements montrés par la figure 4.

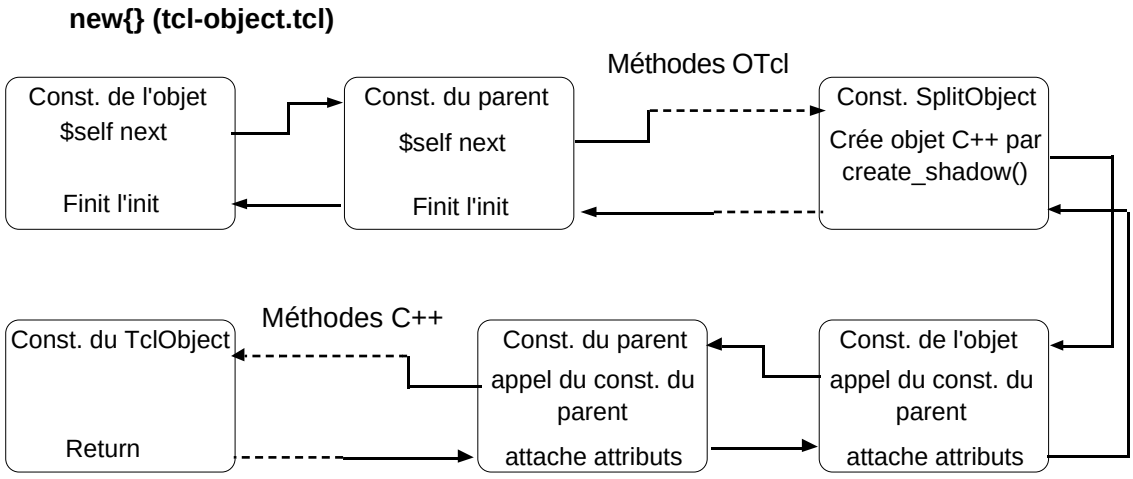


Figure 4: Cinématique de la création d'un objet NS

Pour illustrer cette figure prenons l'exemple de la classe Trace/Enq. Cette classe sert à créer un objet qui écrit une trace à chaque arrivée de paquet. Cet exemple est intéressant car il mélange dans l'arborescence de classes de l'interpréteur une classe purement OTcl (Trace/Enq) avec une classe OTcl/C++ (Trace). La hiérarchie des classes pour cet exemple est montré par la figure 5. Une instance de la classe Trace/Enq est crée par la commande:

```
set trace [new Trace/Enq]
```

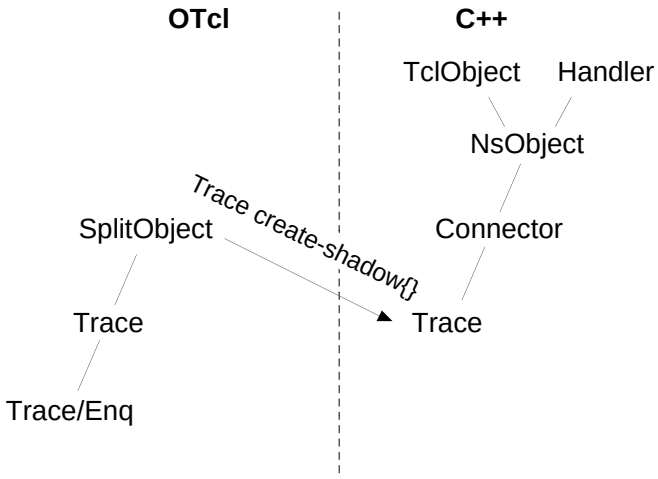


Figure 5: Constructeurs utilisés par Trace/Enq.

La liaison entre l'interpreteur et le simulateur est effectuée par la TclClass TraceClass:

```
class TraceClass : public TclClass {
public:
        TraceClass() : TclClass("Trace") { }
        TclObject* create(int argc, const char*const* argv) {
                if (argc >= 5)
                        return (new Trace(*argv[4]));
                return 0;
        }
} trace_class;
```

La séquence des opérations est la suivante:

- Comme toute création d'objet en OTcl, la méthode `init{}` de la classe Trace/Enq est appelée par la méthode OTcl `create{}`.
- Par la méthode next, l'`init{}` de la classe Trace/Enq appelle le constructeur de la classe parent, ici Trace.
- le constructeur de Trace fait appel également à `init{}` de cette classe qui fait elle-même appel au constructeur du SplitObject.
- La méthode `init{}` du SplitObject (`tcl-object.tcl`) appelle la méthode C++ `create_shadow()` pour la classe Trace via `create-shadow{}` de Trace. Trace/Enq n'a pas de méthode `create-shadow{}` propre, c'est la première méthode trouvée dans la hiérarchie des classes parentes qui sera appelée. La méthode `create-shadow{}` est ajoutée à la classe Trace et attachée à la fonction `create_shadow()` lors de l'initialisation de NS. `Trace::create_shadow()` appelle la méthode `TraceClass ::create()`. Dans la déclaration de la TraceClass, on voit que `create()` appele le constructeur d'une trace et le retourne comme un TclObject.
- Le constructeur de Trace appelle le constructeur de classe parente (ici Connector). Ce schéma d'appel s'applique jusqu'à la classe TclObject. Ici nous assistons ni plus ni moins à la création d'un objet selon la méthode définie par C++.
- Une fois l'objet TclObject créé, nous redescendons la hiérarchie des classes jusqu'à notre classe d'appel initial en exécutant chaque fois le code de chaque constructeur. Ce code consiste principalement à initialiser les attributs de l'objet et à les lier ou attacher avec leur

homologue de l'interpréteur. Nous reviendrons par la suite sur cet aspect des attributs communs au simulateur et à l'interpréteur.

- la méthode Trace `create-shadow{}` est terminée, le code de `l'init{}` de chaque classe de l'arborescence finit d'être évalué jusqu'à celui de la classe d'appel.
- L'objet est créé. Une référence sur cet objet est rendue. Il possède également un identificateur de la forme _o<nnn>.

Quand il n'y a pas de classe purement OTcl comme c'est le cas pour l'estimateur de débit par fenêtre temporelle, le schéma de la construction d'une instance de cette classe est représentée dans la figure 6.

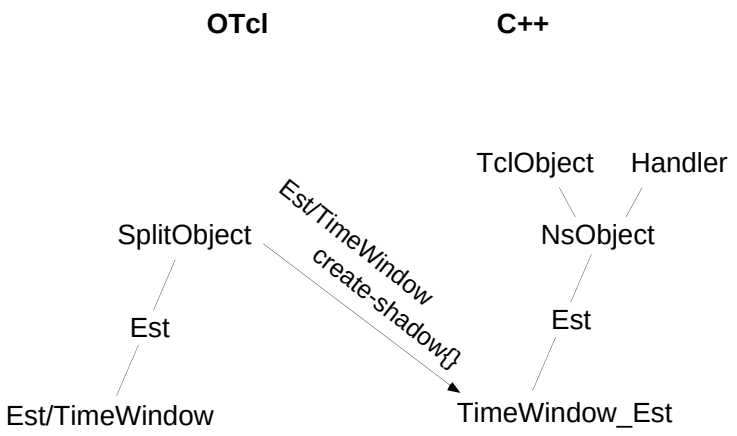


Figure 6: Constructeurs pour l'estimateur de débit par fenêtre temporelle

2.1. Attributs

Un attribut est une variable d'instance. Il indique une variable membre d'une classe. NS permet que les attributs des objets entre l'interpréteur et le simulateur soient liés de manière à ce qu'un changement de valeur d'un attribut compilé soit répercuté dans l'attribut interprété et inversement. NS offre 5 types d'attributs: réel, entier, débit, temps et booléen.

La liaison entre l'attribut des objets de l'interpréteur et du simulateur est établie par le constructeur de l'objet compilé au moyen de la fonction `bind()`:

```
bind("<nom variable OTcl>", &<variable C++>);
```

Pour que la liaison fonctionne, il faut que l'attribut existe préalablement dans l'interpréteur. Pour cela, les attributs sont définis pour la classe avec une valeur par défaut. La plupart de ces initialisations sont effectuées dans le fichier `tcl/lib/ns-default.tcl`. La syntaxe est la suivante:

```
<classe> set <attribut> <valeur>
```

Les attributs sont supposés être initialisés à partir de tclcl. L'initialisation se produit après l'appel au constructeur C++. Le code de l'initialisation est dans le fichier `ns-default.tcl`. La conclusion de cette remarque est qu'il ne faut pas procéder aux initialisations d'attributs dans un constructeur C++.

2.2. Commandes

2.2.1. Classe Tcl

L'accès à l'interpréteur à partir du C++ se fait par les méthodes fournies par la classe Tcl. Les méthodes fournies permettent notamment de procéder à l'évaluation de commandes Tcl à partir du code C++.

Tous les objets créés dans le simulateur sont enregistrés dans une table avec comme clé l'ID attribué à l'objet lors de sa création. Cette classe comporte également les méthodes pour ajouter, supprimer et retrouver un élément dans la table.

2.2.2. Méthodes Command

Pour chaque objet de la classe TclObject, NS fournit la méthode `cmd{}`. Cette procédure sert à exécuter des méthodes pour l'objet interprété qui sont définies pour l'objet compilé. L'appel à la procédure `cmd{}` revient à faire appel à la méthode `command()` de l'objet compilé correspondant.

Par exemple pour Trace (`trace.cc`), le schéma général de `command()` est le suivant:

```
int Trace::command(int argc, const char*const*argv){
  Tcl& tcl = Tcl::instance(); //récupère la référence sur l'interpreteur
  if (argc == 2){
        if (strcmp(argv[1], "detach") == 0){
                [...]
                return (TCL_OK);
        }
  }
}
```

L'appel peut être explicite "`$trace cmd detach`" ou implicite "`$trace detach`". Les deux formes sont équivalentes.

2.3. Initialisation de NS

Après avoir vu les principales classes de base du simulateur, nous allons présenter les principales étapes effectuées lors du démarrage de NS. Ce code figure dans le fichier `tclcl/Tcl.cc` et `tclcl/ns_tclsh.cc`. Pour illustrer notre propos, nous reprenons l'exemple de l'estimateur de débit par fenêtre temporelle du début cette section.

- Au lancement de NS, toutes les variables statiques dérivées de TclClass sont créées et enregistrées dans une liste chainée au moyen de `TclClass()` comme c'est le cas pour la classe `TimeWindow_EstClass`.
- A l'initialisation de NS qui est effectuée par `Tcl_AppInit()` :
    - L'interpréteur est créé.
    - Le code de `tclcl/tcl-object.tcl` est évalué.
    - Ensuite chaque classe dérivée de la classe TclClass déclare dans l'interpréteur une sous-classe de TclObject avec une méthode OTcl pour lancer la construction de l'objet dans le simulateur (cf. `TclClass::bind()`). Cette méthode se nomme `create-`

`shadow{}` et appele `TclObject::create_shadow()`. Avec notre exemple le code OTcl suivant est exécuté:

```
class Est/TimeWindow -parent Est
Est/TimeWindow instproc create-shadow {...}
```

- Le code de `tcl/lib/ns-lib.tcl` est évalué. L'évaluation de ce fichier se termine par l'évaluation du fichier `tcl/lib/ns-default.tcl` qui procéde à l'initialisation des attributs des classes.

L'environnement est prêt. La main est maintenant passée à l'interpréteur, l'utilisateur peut dorénavant taper ses commandes Tcl.

3. Architecture du réseau

Dans cette section, nous présentons les classes de bases utilisables pour définir l'architecture et la topologie du modèle. Les classes Node et Link servent à la composition de la topologie du réseau. Elles modélisent les noeuds et les arcs d'un graphe.

3.1. Noeud

La classe Node est une classe OTcl: elle n'a donc pas d'existence en tant que telle dans le simulateur. Cette classe et ses méthodes sont définies dans le fichier `tcl/lib/ns-node.tcl`. Un noeud est une collection de classifiers et d'agents. Le classifier démultiplexe les paquets. L'agent est habituellement l'entité d'un protocole. L'assemblage des composants est représenté par la figure 6:

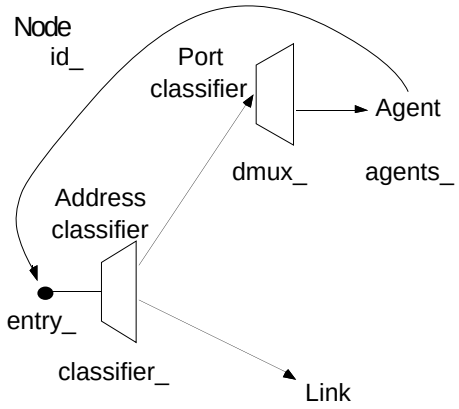


Figure 6: Composants d'un noeud.

Les mots se terminant par le caractère "_" indique des instances variables du noeud. Ce sont pour la plupart des références à un composant du noeud. L'entrée du noeud est indiquée par la référence `entry_` qui pointe sur le premier objet à l'entrée du noeud.

Le trafic émis par l'agent est transmis au point d'entrée du noeud. Que ce soit des paquets reçus de ses voisins ou émis par ses agents, le traitement effectué par le noeud reste identique. Quand un noeud reçoit un paquet, il examine les champs du paquet (habituellement sa destination) et le commute vers la bonne interface de sortie. Cette tâche est exécutée par le classifier. La table des adresses du

classifier est remplie par la procédure de routage (`add-routes{}`). Celle du "port multiplexer" se remplit à chaque ajout d'un agent au noeud.

Une adresse (ou `node id_`) est affectée au noeud lors de sa création. L'adresse est un entier codé sur 8 bits contenu dans la variable d'instance `id_` et aussi dans la variable `address_`. L'identification d'un agent est composé par un mot de 16 bits: les 8 bits de poids forts identifient le noeud et les 8 bits de poids faible servent de numéro de port et identifient l'agent dans le noeud. Une simulation peut comporter donc 256 noeuds au maximum. L'extension de l'espace d'adressage est réalisée par la commande "`Node expandaddr`". L'espace d'adressage des noeuds se code alors sur 22 bits.

Cette commande doit être appelée avant la création des noeuds. "`Node expandaddr`" a une commande équivalente:

```
$ns set-address-format expanded
```

Cette commande augmente la taille de l'adressage de 16 à 32 bits. Elle fait partie de la nouvelle API sur le format des adresses NS. Le code est localisé dans le fichier `tcl/lib/ns-address.tcl`. La commande "`Node expandaddr`" reste toujours disponible par soucis de compatibilité.

3.1.1. Classifier

Un classifier a pour objectif de retrouver une référence à un autre objet de la simulation à partir d'une comparaison sur un critère dont la valeur est contenue dans le paquet. Il a un rôle de démultiplexeur en quelque sorte. Le classifier sert notamment pour un noeud à modéliser la fonction de relayage (forwarding). Le classifier contient une table de *n* slots. A chaque slot correspond la référence à un NsObject. Quand un paquet est reçu par le classifier, il identifie le slot de sortie par `classify()`. Si le slot est invalide, le classifier lance `no-slot{}`. Les méthodes du classifiers sont:

- `install{}`, installe une référence à un élément dans le classifier.
- `element{}`, retourne la liste des éléments installés dans le classifier.
- `slot{}`, retourne la référence installée à un slot spécifique.
- `clear{}`, efface le contenu d'un slot spécifique.

Il existe plusieurs sortes de classifiers dans NS. Il y a ceux qui font :

- des classifications avec plusieurs critères ou des critères spécifiques comme par exemple le flow id, l'adresse de destination.
- des traitements particuliers comme le classifier replicator. Ce classifier duplique un paquet sur tous ses slots.

Concernant le classifieur d'adresse du noeud, la table de slots est indexée par l'`id_` du noeud de destination. A réception d'un paquet et suivant l'adresse de destination, le classifier détermine quel est l'objet de simulation suivant c'est à dire à qui donner ce paquet. C'est soit au port classifier du noeud, soit à un lien de sortie.

3.1.2. Détail sur le noeud

Quand un paquet est reçu par un noeud, ce dernier le conserve, détermine sa route et l'envoie à un autre NsObject. Il n'y a pas de durée ou de temps consommé pour ces opérations. Pour prendre en compte ces temps, il convient d'ajouter un élément de délai dans le noeud comme c'est fait pour le lien.

3.2. Lien

Le lien sert à relier les noeuds. Il modélise le système de transmission. Le lien est principalement caractérisé par un délai de propagation et une bande passante. C'est une classe OTcl qui regroupe un ensemble de composants dérivés de la classe Connector. Cette classe et ses méthodes sont définies dans le fichier `tcl/lib/ns-link.tcl`. Des liens plus sophistiqués peuvent être dérivés de cette classe. Quelque soit le type du lien, il comporte 5 références sur des objets qui le composent. Le lien peut se représenter selon la figure 7.

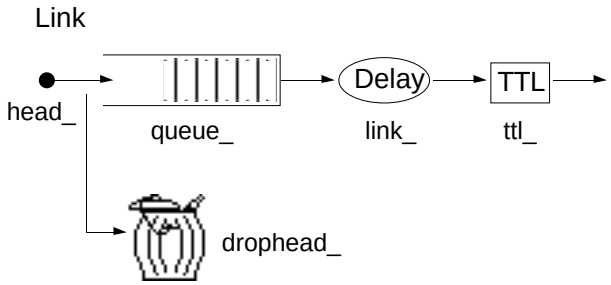


Figure 7: Composants d'un lien.

Les 5 instances variables suivantes ont la sémantique:

- `head_`, point d'entrée du lien, il pointe sur le premier objet du lien.
- `queue_`, référence la file d'attente de paquets. Cette file modélise celle que l'on trouve habituellement aux interfaces de sortie.
- `link_`, référence l'élément qui caractérise le lien souvent en terme de délai et bande passante.
- `ttl_`, référence l'élément qui manipule la durée de vie de chaque paquet.
- `drophead_`, référence l'élément qui traite les pertes de paquets au niveau de la file d'attente.

Parmi les variables du lien, une référence est gardée sur le noeud amont (`fromNode_`) et sur le noeud aval (`toNode_`).

3.2.1. Connector

Tous les objets du lien sont dérivés de la classe C++ Connector. Un Connector à la différence d'un classifier, envoie les paquets à un seul récepteur. Dans le cas normal le Connector a un voisin connu sous la référence `Connector::target_`. Si il doit détruire le paquet il le passe à l'objet référencé par `Connector::drop_`. En somme, le Connector délivre le paquet à son voisin indiqué par `target_` ou `drop_`. Si `drop_` est indéfini, le paquet à jeter est supprimé de la simulation. `target_` doit être défini sous peine d'obtenir un "Segmentation fault" lors de son utilisation à la simulation. L'initialisation de ces variables se fait par les commandes OTcl `target` ou `drop-target` qui sont définies par la

procédure `Connector::command()`. Par exemple pour relier l'objet référencé par `queue_` à celui référencé par `link_`, la syntaxe de la commande est la suivante:

```
$queue_ target $link_
```

La variable C++ `target_` de l'objet queue contient la référence à l'objet link.

La variable `target_` définit l'objet auquel le paquet sera envoyé. Ainsi tous les paquets reçus par l'objet Connector une fois traités, sont passés à la `target_` du Connector. Le passage d'un paquet directement à son voisin peut se faire sans passer par l'ordonnanceur par "`Connector::send(Packet* p, Handler* h)`". Si le second argument est absent, par défaut il prend la valeur nulle . Ce paramètre sert de "callback" (voir le paragraphe: traitement séquentiel en temps simulé).

3.3. Agent

L'agent est un autre composant d'un noeud. Il modélise les constructeurs et les consommateurs de paquets IP. La classe agent fournit des méthodes utiles au développement de la couche transport et à d'autres protocoles du plan de signalisation ou de gestion. Cette classe est à la fois dans l'interpréteur et dans le simulateur. C'est la classe de base pour définir des nouveaux protocoles dans NS. Elle fournit l'adresse locale et de destination, les fonctions pour générer les paquets, l'interface à la classe Application. Actuellement NS comporte de nombreux agents citons: UDP, protocoles de routage, différentes versions de TCP, RTP, etc.

Les principales commandes sont celles qui ont trait à des attachements:

- `$ns_ attach-agent $node_ $agent_` attache l'agent au noeud. Concrètement cela signifie que la référence de l'agent est mis dans le port classifier pour qu'il puisse recevoir des paquets. La variable `target_` est initialisée avec la valeur `entry_` (point d'entrée du noeud). C'est par cette commande qu'un numéro de port (variable OTcl portID_) est attribué et que l'adresse de l'agent est déterminée. La variable `Agent::addr_` contient cette adresse.
- `$ns_ connect $agent1 $agent2` établit une connexion entre les deux agents. Cette commande consiste uniquement à initialiser la variable `Agent::dst_` (adresse de destination) de chaque agent.
- `$agent_ attach-source $source_` attache une source de trafic à un agent.
- `$source_ attach-agent $agent_` idem

La création d'un nouvel agent demande de définir son héritage, de créer sa classe et de définir les fonctions virtuelles de l'API Agent. Ensuite il faut définir les fonctions qui feront office de liaison avec l'interpréteur. Le travail se termine par l'écriture de code OTcl pour l'accès à l'agent.

Les fonctions utiles aux classes dérivées ou pouvant être redéfinies sont:

- `allocpkt()` crée un paquet et initialise ses différents champs avec des valeurs nulles.
- `trace()` constitue la fonction de trace (ou tracer) d'une variable d'un agent. Nous y reviendrons par la suite.

- `send()` et `recv ()` servent à envoyer et recevoir des paquets constitués.
- `sendmsg()` envoie une suite binaire (c'est un ou des paquets non constitués).

Lors du développement d'un nouvel agent, il ne faut pas oublier d'appeler à la dernière ligne `command()` la méthode équivalente de la classe parent. Ainsi la dernière ligne doit avoir cette instruction "`return Agent::command(argc, argv);`". Le destinataire d'un agent n'est pas forcément toujours le même, l'exemple dans `tcl/rtglib/route-proto.tcl` montre comment changer le destinataire.

Avec l'évolution de NS, le rôle de l'agent a évolué. Dans des versions passées, l'agent faisait également office de source de trafic. C'est la raison pour laquelle on a la classe Agent/CBR ou Agent/UDP/CBR. Ces deux classes sont identiques. Il n'est pas conseillé d'utiliser ces classes actuellement. La bonne démarche est d'utiliser la classe Agent/UDP et la classe Application/Traffic/CBR. La séparation de la source de trafic de l'agent apporte une plus grande flexibilité.

3.4. Paquet et en-tête

La gestion des paquets dans NS met en oeuvre trois classes:

- la classe Packet modélise les unités de données échangées entre les composants du réseau.
- la classe PacketHeaderClass est dérivée de la classe TclClass et fournit les méthodes pour accéder et communiquer avec l'interpréteur. Elle sert à localiser dans le paquet l'en-tête d'un protocole particulier.
- la classe PacketHeaderManager sert à gérer les en-têtes de paquets qui seront disponibles dans le simulateur.

La classe Packet est définie dans le fichier `packet.{h,cc}`. La classe Packet est dérivée de la classe Event. Un paquet peut être donc placé dans l'échéancier avec l'heure de son arrivée. Seules les instances de la classe NsObject peuvent recevoir et émettre des paquets. Un paquet NS se schématise selon la figure 8 et contient :

- un élément liant pour pouvoir les chaîner entre eux,
- la longueur de la zone mémoire du paquet dédiée aux en-têtes de protocoles,
- une zone mémoire dédiée spécialement aux en-têtes de protocoles,
- une zone mémoire pour les éventuelles données d'application.

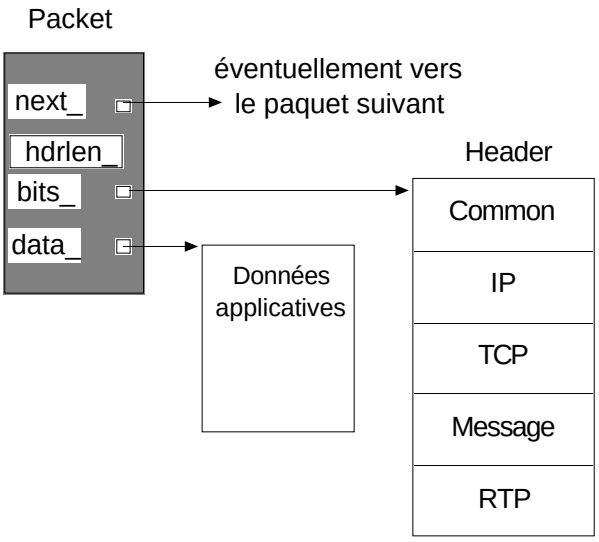


Figure 8: Le paquet NS.

La zone mémoire dédiée spécialement aux en-têtes de protocoles est considérée comme un tableau d'octets et a une taille suffisante pour tous les en-têtes. Cette zone est la concaténation de tous les en-têtes de paquets disponibles dans le simulateur. Le contenu et la taille de cette zone sont définis à l'initialisation de l'objet Simulator. Sa taille est conservée dans la variable `Packet::hdrlen_`. L'en-tête d'un protocole particulier est accédé par une adresse relative dans la zone mémoire à partir de l'adresse indiquée par `bits_`. Pour illustrer la gestion des paquets et des en-têtes dans NS, nous allons prendre l'exemple de l'ajout d'un format de paquet pour le protocole imaginaire smart. Nous commençons par définir son en-tête c'est à dire le format de la PDU de ce protocole:

```
struct hdr_smart {
int     id_;
int     seqno_;

static int offset_;     /* Utilisé pour la localisation de cet en-tête dans la
                           zone des en-têtes du paquet */
/* Fonctions pour accès aux champs de l'en-tête */
inline int& id() {return (id_);}
inline int& seqno() {return (seqno_);}
};
```

L'adresse relative du début de l'en-tête est conservée dans la variable static `offset_` de l'en-tête du protocole. Nous verrons par la suite que cette valeur est également conservée dans l'interpréteur sous le nom `off_<hdrname>_`.

Pour utiliser cet en-tête dans le simulateur, il faut maintenant l'ajouter dans la liste maintenue par le PacketHeaderManager. Ceci se fait dans le fichier `tcl/lib/ns-packet.tcl` en ajoutant la ligne `{ SMART off_smart_ }` qui donne le suffixe du nom de la classe OTcl de l'en-tête de ce paquet et le nom de la variable OTcl destinée à indiquer la localisation de cet en-tête dans la zone des en-têtes du paquet NS. La classe OTcl du paquet porte le nom de PacketHeader/SMART.

```
# set up the packet format for the simulation

PacketHeaderManager set hdrlen_ 0

foreach pair {
```

```
                { Common off_cmn_ }
                { IP off_ip_ }
                { TCP off_tcp_ }
                { SMART off_smart_ }

        } {
        set cl PacketHeader/[lindex $pair 0]
        set var [lindex $pair 1]
        PacketHeaderManager set vartab_($cl) $var
}
```

Selon notre exemple, le paquet du simulateur aura le format représenté par la figure 9.

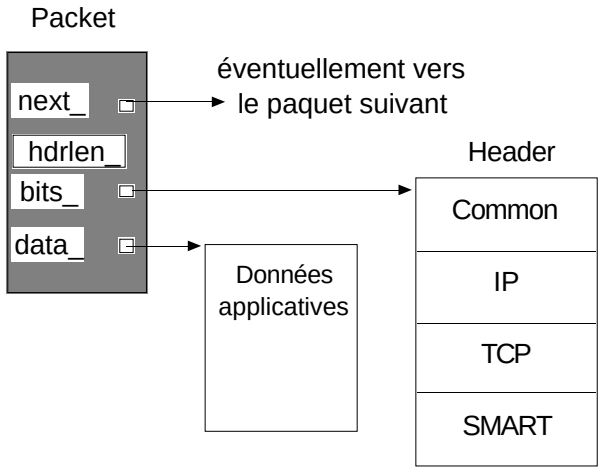


Figure 9: Exemple d'un format du paquet .

La zone d'en-tête comportera l'espace mémoire nécessaire aux quatre en-têtes rangés selon l'ordre indiqué par le schéma 9. La simulation pourra se faire uniquement avec ces quatre protocoles. Il est a noter que l'en-tête "common" doit toujours être présent. Elle sert à la fonction de trace et à d'autres fonctions indispensables à la simulation. Les principaux champs de l'en-tête "common" sont:

- `size_`, taille simulée du paquet.
- `ptype_`, type du paquet.
- `ts_`, timestamp.
- `ref_count_`, durée de vie en nombre de sauts. Le paquet est supprimé quand la durée de vie vaut 0.

Le calcul de `Packet::hdrlen_` est effectué par PacketHeaderManager. Pour cela il a besoin de connaître le taille de l'en-tête de chaque paquet. Il faut donc indiquer quelle structure d'en-tête (c'est à dire dans notre exemple hdr_smart) utiliser pour chaque protocole. Ceci est fait au moyen de la classe PacketHeaderClass. Cette classe sert également à établir les liens sur les attributs du paquet entre l'interpréteur et le simulateur. Pour notre exemple, la déclaration est la suivante:

```
class SmartHeaderClass: public PacketHeaderClass{
public:
        SmartHeaderClass(): PacketHeaderClass ("PacketHeader/SMART",
                                sizeof (hdr_smart)){
                offset(&hdr_smart::offset_);
        }
        void export_offsets() {
                field_offset("id_", OFFSET(hdr_smart, id_));
        };
```

```
} class_smarthdr;
```

Le constructeur `SmartHeaderClass()` appelle le constructeur `PacketHeaderClass()` avec deux paramètres:

- le nom de la classe OTcl de cet en-tête. Le suffixe du nom de cette classe est identique à celui qui figure dans la liste du PacketHeaderManager. Au total, la chaine de caractères "SMART" est écrite 2 fois (une fois dans l'interpréteur et une fois dans le simulateur).
- la taille de la structure hdr_smart.

La PacketHeaderClass contient 2 variables privées:

- `hdrlen_`, taille de la structure de l'en-tête.
- `offset_`, un pointeur sur la variable static `offset_` de l'en-tête du paquet. Le constructeur initialise ce pointeur par l'appel à `offset()`. Dans notre exemple, le constructeur est `SmartHeaderClass()`.

Dernier aspect de notre exemple, la procédure `export_offsets()` sert à acceder à un champ particulier de l'en-tête de l'interpréteur. Ici c'est le champ `id_`. Cette fonctionnalité peut être utile pour configurer la classification d'un classifier sur la valeur de l'`id_` d'un paquet.

Attention: Il ne faut pas oublier de définir la variable static `hdr_smart::offset_` par la ligne suivante dans le fichier `smart.cc`:

```
int hdr_smart::offset_;
```

L'accès à l'en-tête du paquet smart s'effectue par `Packet::access()`. Supposons que le composant qui utilise ce paquet soit l'agent smart.

```
class SmartAgent: public Agent {
public:
        SmartAgent(): Agent (PT_SMART) {
                bind("off_smart_", &off_smart_);
        };
[...]
protected:
        int off_smart_;
};
```

A la création de l'agent smart, la variable `SmartAgent::off_smart_` est liée avec la variable OTcl `off_smart_` qui contient l'adresse relative de l'en-tête. Le résultat est que l'agent smart connaît l'adresse relative de l'en-tête smart dans la zone mémoire dédiée aux en-têtes de protocoles. Pour que la liaison fonctionne, il faut que le composant C++ ait un objet en miroir dans l'interpréteur. C'est le cas de la classe Agent. La raison est que toutes les variables `off_<hdrname>_` sont des instances variables de SplitObject. Pour accéder ensuite à l'en-tête smart, le code est le suivant:

```
void SmartAgent::sendmsg(int nbytes) {
Packet*     p= allocpkt();
hdr_smart* sh= (hdr_smart*)p->access(off_smart_);
sh->id()     = 10;
[...]
}
```

A retenir pour l'écriture C++ de composants, l'offset du "common header" (`off_cmn_`) est initialisé par le constructeur de NsObject et l'offset de IP (`off_ip_`) est initialisé quant à lui avec le constructeur de l'Agent.

3.4.1. Quelques détails "croustillants" sur la gestion des paquets

L'initialisation de la partie en charge de la gestion des paquets se fait en plusieurs étapes:

- au lancement de l'application (défini comme l'allocation mémoire des variables statiques)
  - La PacketHeaderClass de chaque protocole est activée. Le calcul de `hdrlen_` de chaque en-tête et le positionnement du pointeur `PacketHeaderClass::offset_` sur la variable `offset_` de l'en-tête de chaque protocole sont effectués.
- à l'initialisation de l'application proprement dite (appel à `TclAppInit()` (`ns_tclsh.cc`))
  - Lancement de `PacketHeaderClass::bind()` pour les différents protocoles par `TclClass::init()` (`Tcl.cc`).
  - Initialisation de l'attribut `PacketHeaderClass::hdrlen_` pour chaque classe de paquet, selon notre exemple la commande suivante est executée:

    ```
    PacketHeader/SMART set hdrlen_ 4
    ```
  - Ajout de la commande `offset{}` pour chaque classe de paquet. `offset{}` appelle `PacketHeaderClass::method()` (`packet.cc`). Selon notre exemple, cela revient à définir :

    ```
    PacketHeader/SMART instproc offset{ {offset ""} }{...}
    ```
  - Appel à `export_offsets()` pour chaque classe de paquet.
- à l'initialisation de l'intance de la classe OTcl Simulator (appel à `Simulator init{}` (`ns-lib.tcl`))
  - Création d'une instance de PacketHeaderManager. Il effectue le calcul de l'offset pour chaque en-tête.
  - Initialisation des variables d'instances OTcl `off_<hdrname>_` de SplitObject et de la variable `offset_` contenue dans l'en-tête de chaque protocole.

4. Eléments de la simulation

4.1. Simulateur

La simulation est configurée, contrôlée et exploitée via l'interface fournie par la classe OTcl Simulator. Cette classe joue le rôle d'API. Elle n'existe que dans l'interpréteur. Un script de simulation commence toujours par créer une instance de cette classe par la commande:

```
set ns_ [new Simulator]
```

L'objet Simulator (ici représenté par la référence `ns_`) contiendra une référence sur chaque élément de la topologie du réseau simulé et sur les options mises pour la simulation comme par exemple les noms des fichiers de trace. La topologie du réseau est construite avec des instances des classes OTcl Node et Link. Les méthodes de la classe Simulator sont définies dans le fichier `tcl/lib/ns-lib.tcl`.

L'initialisation du Simulator par la méthode `init{}` initialise le format des paquets de la simulation et crée un ordonnanceur.

4.2. Ordonnanceur

L'ordonnanceur est défini dans le fichier `scheduler.{h,cc}`. L'ordonnanceur a en charge de choisir l'événement le plus proche en terme de temps, d'exécuter les traitements, de faire progresser le temps simulé et d'avancer à l'événement suivant etc. Les événements sont rangés dans un échéancier. Un seul événement est traité à la fois. Si plusieurs événements doivent être traités au même instant. Ils sont exécutés en série mais au même instant en terme de temps simulé. On parle de quasi-parallélisme. Le temps simulé est l'échelle de temps du modèle de simulation. Le fonctionnement par défaut de l'ordonnanceur est "Calendar Scheduler". D'autres méthodes de gestion existent [3].

L'API de l'ordonnanceur est rendu par l'objet Simulator. Les principales commandes sont:

- `now{}`, retourne le temps simulé.
- `at <time> "<action>"`, `at{}` ajoute un événement dans l'échancier.
- `run{}`, lancement de la simulation.
- `halt{}`, arrêt de l'ordonnanceur.
- `cancel $eventID`, annule un événement préalablement placé avec "at". `EventID` est la valeur retournée lors du placement de l'événement par `set eventID [$ns_ at <time> "<action>"]`

L'événement est défini par la classe Event et se caractérise par l'heure de déclenchement et par l'événement à réaliser (handler): c'est l'objet qui va consommer l'événement.

```
class Event {
public:
        Event* next_;           /* event list */
        Handler* handler_;      /* handler to call when event ready */
        double time_;           /* time at which event is ready */
        int uid_;               /* unique ID */
        Event() : time_(0), uid_(0) {}
};

/*
 * The base class for all event handlers. When an event's scheduled
 * time arrives, it is passed to handle which must consume it.
 * i.e., if it needs to be freed it, it must be freed by the handler.
 */
class Handler {
 public:
        virtual void handle(Event* event) = 0;
```

Tous les NsObject sont des "Handler". La fonction virtuelle handle devient:

```
void NsObject::handle(Event* e)
{
        recv((Packet*)e);
}
```

Comme on peut le lire, c'est la fonction de réception paquet qui est appelée car les classes dérivées de la classe NsObject manipulent des événements de type paquet. La classe NsObject se dérive en premier en deux sous-classes:

- connector,
- classifier.

La différence se situe sur la capacité de connexion. A la réception d'un paquet, un connector fait suivre le paquet au NsObject référencé par `target_`, le classifier doit déterminer à quel NsObject est destiné ce paquet. Il utilise une table `slots_[]` dont chaque slot contient une référence à un NsObject. Le connector sert à assembler les objets en relation 1-1 tandis que le classifier établit des relations 1-N entre les objets.

4.3. Consommation de temps

Aucun objet dans la simulation ne peut faire avancer le temps. Pour consommer du temps, il faut obligatoirement passer par l'ordonnanceur. Ceci se réalise par `scheduler::schedule()` :

```
void Scheduler::schedule(Handler*, Event*, double delay);
```

Mais avant de pouvoir faire appel à cette fonction, il faut obtenir la référence sur le simulateur par l'appel à la fonction `Scheduler::instance()`.

`scheduler::schedule()` consiste à insérer un événement dans l'échéancier de la simulation avec une heure de déclenchement. L'heure de déclenchement correspond au temps simulé actuel plus le temps à consommer (exprimé par le paramètre `delay`). Lorsque le temps simulé arrive à l'heure de déclenchement de l'événement, l'objet référencé par la variable de type Handler est activé.

Cette méthode est également valable pour passer un paquet à un voisin en un temps non nul. Quelque soit l'utilisation de `schedule()`, le paramètre `Handler` doit avoir une valeur non nulle.

4.4. Traitement séquentiel en temps simulé

Le temps simulé est découplé du temps réel. Si aucun objet ne fait de consommation de temps, vis à vis du temps simulé tous les traitements se font en même temps (mais par rapport au temps réél ils sont exécutés en série). Un simulateur est naturellement une machine pseudo-parallèle. Pour modéliser un objet à traitement séquentiel, il est nécessaire d'adjoindre un mécanisme de blocage qui empêche l'objet de commencer un nouveau traitement avant d'avoir fini le précédent (en terme de temps simulé).

4.4.1. Exemple de la file d'attente

Pour illustrer ce mécanisme, prenons, l'exemple du modèle classique de file d'attente: une file d'attente et un serveur. Ce modèle est souvent pris comme représentation du lien pour les systémes à multiplexage temporelle asynchrone. La file d'attente modélise l'interface de transmission et le serveur, le système de transmission. Tant que le serveur est occupé en transmission de paquet,

aucune autre transmission ne peut avoir lieu. La durée de la transmission dépend à la fois de la longueur du paquet et du débit du lien. Le serveur effectue la transmission du paquet selon un mode séquentiel. Le composant modélisant un serveur a en charge de:

- Empêcher de sortir des paquets de la file tant qu'il y en a un en transmission. La transmission dure T_t Temps de transmission.
- Autoriser une nouvelle transmission à la terminaison de la transmission du précédent paquet ($t_0 + T_t$)
- Activer l'objet récepteur d'un paquet à l'instant de réception T_t+T_p. On pose T_p Temps de propagation.

La classe Queue et LinkDelay représente respectivement la file d'attente et le serveur. Dans le lien, LinkDelay est l'élément de délai qui se situe en aval de la queue. Une queue est bloquée jusqu'à ce qu'elle soit re-activée par son voisin aval (ici délai). C'est avec ce mécanisme que le délai de transmission est simulé. Soient les déclarations simplifiées suivantes :

```
class Queue: public Connector {
public:
        void resume();
protected:
        Queue();
        int blocked_;
        QueueHandler qh_;
};

Queue::Queue(): qh_(*this){}    // appel du constructeur QueueHandler

class QueueHandler: public Handler {
public:
        inline QueueHandler (Queue& q): queue_(q) {}
        void handle(Event *);
private:
        Queue& queue_;
};

void QueueHandler::handle(Event*)
{
        queue_.resume();
}

void Queue::resume()
{
        // Enlever un paquet de la file d'attente et l'affecte à p
        if (p != 0) {
                target_->recv(p, &qh_); // passe le paquet à LinkDelay::recv
                                        avec le callback qh_
        } else blocked_ = 0;
}

void Queue::recv(Packet* p, Handler*)
{
        // ajouter le paquet dans la file d'attente
        if (!blocked_) {
                // non bloque, on peut emettre le paquet
                blocked_ = 1;
                target_->recv(p, &qh_); // passe le paquet à LinkDelay::recv
                                       avec le callback qh_
                }
        }
```

```
}

Class LinkDelay : Public Connector {
public:
protected:
        delay_;         // temps de propagation
        bandwith_;      // Bande passante, pour le calcul du temps de
                           transmission
        Event intr_;    // In transit
}

void LinkDelay::recv (Packet* p, Handler* h)
{
double txt =txtime(p);          // temps de transmission
Scheduler& s = Scheduler::instance();

s.schedule(target_, p, txt + delay_);    //Evenement E1
s.schedule(h, &intr_, txt);              //Evenement E2
}
```

La dernière méthode opère à réception d'un paquet et place deux événements dans l'échéancier. L'événement E1 indique l'instant auquel arrivera le paquet à l'objet aval référencé par `target_`. L'événement E2 repère l'instant auquel l'objet amont a terminé la transmission du paquet. Le paramètre Handler h est le QueueHandler `qh_`. Le QueueHandler lorsqu'il aura la main via `handle()` appelera `Queue::resume()`. Le QueueHandler est ce que l'on appele un *callback*.

4.5. Temporisateur

Il existe deux mécanismes de temporisateurs dans NS:

- un pour l'ordonnancement des événements générés par l'interpréteur. Il est défini dans le fichier `tcl/ex/timer.tcl`.
- un second défini en C++ pour les composants du simulateur.

Les temporisateurs C++ sont dérivés de la classe TimerHandler. Le temporisateur est développé sur un système de callback avec utilisation de la classe Scheduler. Il reprend les idées exposées pour effectuer des traitements séquentiels. Pour obtenir un temporisateur particulier, il faut définir une nouvelle classe. Prenons l'exemple d'un temporisateur de détection d'inactivité pour un agent TCP.

```
class InactiveTimer : public TimerHandler {
  public:
        InactiveTimer (TCPAgent *a) : AgentTimerHandler() { a_ = a; }
        virtual void expire(Event *e);
  protected:
        TCPAgent *a_;
};

void InactiveTimer:expire(Event *e) {
        a_->close()
}
```

`InactiveTimer::expire()` appelle `TCPAgent ::close()` pour fermer la connexion TCP. Les autres fonctions de manipulation du temporisateur sont celles définies dans la classe TimerHandler.

Le temporisateur Tcl sert à appeler des méthodes OTcl de l'interpréteur durant la simulation . Il est réalisé par `at{}` de Simulator. Pour faire un temporisateur, il faut dériver la classe Timer et définir `timeout{}`. Cette méthode est appelée à expiration du temporisateur.

5. Interprétation

NS fournit deux moyens pour extraire des données de la simulation:

- La *trace* enregistre dans un fichier les changements d'états d'un paquet ou de valeur d'une variable.
- Le *moniteur* est un objet pouvant faire des calculs sur différentes grandeurs tel que le nombre de paquets ou d'octets arrivée, etc.

5.1. Trace

Il existe deux types de traces: les traces effectuées à partir d'une file d'attente d'un lien et les traces de variables. Le code des fonctions de trace sont respectivement dans `trace.cc` et `tclcl/tracedvar.cc`. Le fichier de traces est un fichier de texte stucturé en lignes.

5.1.1. Queue

Chaque type d'événement pouvant faire l'objet d'un enregistrement se définit par une sous-classe de Trace. La classe Trace consiste à réception d'un paquet à effectuer l'écriture des principales caractéristiques du paquet dans le fichier et à passer le paquet à l'objet aval. Une ligne de trace commence par une lettre indiquant le type de trace afin de différencier chaque sous-classe de Trace:

- `+` mise en file d'attente
- `-` sortie de la file d'attente
- `d` suppression de la file d'attente
- `r` réception au noeud
- `l` perte (suite à une erreur binaire)

Pour effectuer la trace d'un lien, il faut établir la configuration de lien représentée par la figure 10. 4 objets de trace sont ajoutés et sont présentés par leur référence dans la figure:

- `enqT_`, pour l'arrivée des paquets
- `deqT_`, pour le départ des paquets de la file d'attente
- `drpT_`, pour la perte des paquets dûe à la congestion de la file d'attente
- `rcvT_`, pour la réception des paquets au noeud suivant.

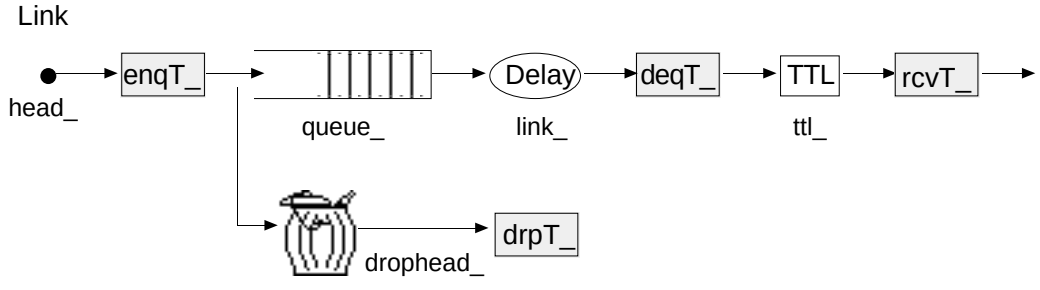


Figure 10: Composition du lien avec les objets de tracage.

Cette configuration se crée par la commande OTcl Simulator `trace-queue{}`. La commande `create-trace{}` et principalement utilisée par `trace-queue{}`. `Trace::recv()` est appelée par la référence d'une trace d'un lien. Elle formate la ligne puis l'écrit dans le fichier par `dump()`.

```
void Trace::recv(Packet* p, Handler* h)
{
        format(type_, src_, dst_, p);
        dump();
        namdump();
        /* hack: if trace object not attached to anything free packet */
        if (target_ == 0)
                Packet::free(p);
        else
                send(p, h);
}
```

Une ligne de trace se présente sous le format suivant:
`opération, temps, noeud source, noeud destination, type de paquet, taille du paquet, drapeaux, flot id, adresse source du paquet (noeud.port), adresse destination du paquet (noeud.port), numéro du paquet, id du paquet.`

5.1.2. Variable

Les variables `double` ou `int` peuvent être tracées. C'est à dire que chaque fois que leur valeur change, une trace est écrite dans le fichier de traçage. Le traçage de variable est expliqué dans le paragraphe 2.2.3 de [3].

Dans NS, les variables pouvant être tracées sont dérivées de la classe de base TracedVar. Deux classes sont fournies: TracedInt et TracedDouble. En reprenant notre exemple développé au paragraphe paquet et en-tête, supposons que nous voulions tracer la variable NbPaquet de l'agent smart. Il faut déclarer dans la classe SmartAgent:

```
TraceInt        NbPaquet_;
```

Ensuite au moyen de l'interpréteur, on active le traçage de la variable en indiquant son nom et eventuellement l'objet fera l'écriture dans le fichier (on l'appelera le traceur).

```
# $smart utilise le traceur générique fourni dans trace.cc
set traceur [new Trace/Var]
$smart trace "NbPaquet_" $traceur
```

La commande `trace{}` est traitée par `TclObject::command()` (`Tcl.cc`) qui appelle `TclObject::TraceVar()`. Cette fonction associe le traceur à la variable. Quand la valeur change, `TraceInt::assign()` affecte la nouvelle valeur à la variable et appelle `trace()` du traceur. L'instruction en question dans `assign()` est: `tracer_->trace(this);`.
Si l'agent smart veut avoir son propre traceur, il faut définir `SmartAgent::trace(TracedVar *)`. L'activation par l'interpréteur devient alors:

```
# $smart utilise son propre tracer
$smart trace "NbPaquet_"
```

5.2. Moniteur

Un moniteur utilise des objets observateurs dits sniffeurs (snoop) qui sont insérés dans la topologie réseau. Le rôle de ces petits sniffeurs est de faire remonter les informations d'états du réseau au moniteur. Celui-ci est un point de ralliement de ces informations, où des calculs sont effectués. La figure 11 montre l'assemblage des composants du lien lorsqu'il y a un QueueMonitor. La classe QueueMonitor est définie dans le fichier `queue-monitor.cc`. Le moniteur est un composant qui effectue des calculs relatifs au lien. Dans un souci de performance, il est préférable d'utiliser des traces et de procéder à des traitements post-mortem à la simulation grâce à des outils comme perl ou awk. Des exemples de scripts perl sont dans `/bin/{getrc, set_flow_id}`.

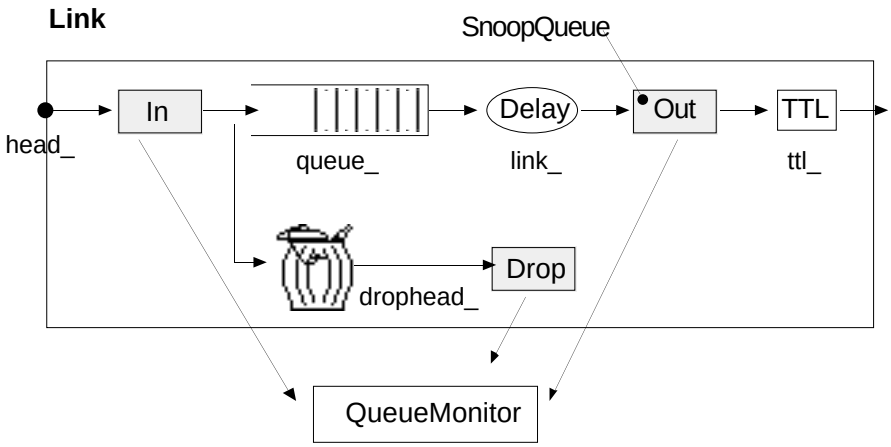


Figure 11: Installation d'un moniteur sur un lien.

Un moniteur peut servir à collecter des statistiques sur l'ensemble des flots ou par flot. Dans ce dernier cas, le moniteur est de la classe FlowMon et est défini dans le fichier `flowmon.cc`. Les statistiques collectées peuvent être un comptage ou une intégrale. Le calcul d'intégrale est approximé par une somme de valeurs discrètes. Les statistiques sont élaborées par paquet et par octet. Les statistiques collectées sont par exemple:

- `parrivals_` mesure combien de paquet ont été reçus par la file.
- `barrivals_` mesure combien d'octets ont été reçus par la file.
- `pdepartures_` mesure les paquets qui ont quitté le lien. A noter que parrivals_ = pdrops_ + pdepartures_.
- `pktsInt_` intégration de la taille de la file en paquet.

La façon la plus simple mais également la plus grossière d'utiliser un moniteur consiste à utiliser la méthode:

```
Simulator instproc monitor-queue { n1 n2 qtrace { sampleInterval 0.1 } }
```

qtrace est le fichier de trace du monitor. Par exemple, pour procéder à la surveillance des flots d'un lien entre deux routeurs, le code tcl est le suivant:

```
#  set the links and create a flow monitor
#
#     r1 ---> r2
#
```

```
set slink [$ns link $nodes(r1) $nodes(r2) ]  # retourne la ref sur le lien
set fm [open all.fm w]          # ouvre un fichier de trace
set fmon [$ns makeflowmon Fid ] # crée un objet flow monitor
$ns attach-fmon $slink $fmon    # attache le fmon au lien
set bytesInt_ [new Integrator]  # cree l'objet d'intégration en octet
set pktsInt_ [new Integrator]
$fmon set-bytes-integrator $bytesInt_  # initialise la ref du fmon
$fmon set-pkts-integrator $pktsInt_
$fmon attach $fm                # attache le fichier de trace au fmon
$ns at $finishTime "$fmon dump"
```

La dernière commande vide les différentes variables dans le fichier all.fm à la fin de la simulation. D'autres exemples sont disponibles dans le fichier `tcl/test/test-suite-simple.tcl`. Il n'est pas conseillé de faire des dump d'un flow monitor pendant la simulation au risque de fausser les statistiques. Il est préférable d'effectuer des traces intermédaires comme le montre le code ci-dessous:

```
# open a trace file
set tFile [open queue.tr w]
# set up the queue-monitor
$ns monitor-queue $n0 $n1 $tFile {sampleInterval}
# sampleInterval is optional and defaults to 0.1 seconds
# start tracing using either
[$ns link $n0 $n1] start-tracing
# or
# [$ns link $n0 $n1] queue-sample-timeout
# which averages over the last sampleInterval
```

Attention, il faut toujours fermer le fichier de trace à la fin de la simulation. Sinon les résultats peuvent être faussés.

6. Routage

Une fois les différents éléments topologiques créés, il reste encore l'assemblage des liens avec les noeuds et la constitution de la table de routage pour chaque noeud. Cette tâche est du ressort du routage. Nous n'allons pas dans cette version décrire les différentes fonctionnalités du routage de NS qui sont très nombreuses. Dans le cas par défaut, nous présentons comment les routes sont déterminées. Le routage par défaut dans NS consiste en un routage *unicast statique* c'est à dire déterminé avant le démarrage de la simulation. Le routage utilise l'algorithme SPF de Dijkstra. Le graphe du réseau est construit à partir de la variable d'instance `link_` du simulator. La table des liens `link_(src:dst)` indique le coût associé à un lien identifié par ses deux extrémités. La source et la destination sont les identifiants des noeuds. Un lien coûte par défaut 1. Les fonctions et procédures du routage unicast statique peuvent être trouvées dans les fichiers `tcl/lib/ns-route.tcl`, `tcl/rtglib/route-proto.tcl` et `route.cc`.

Le calcul de route est effectué dans la classe RouteLogic. Une instance de cette classe est créée au lancement de la simulation par la fonction `run{}` de simulator. La configuration des tables de routage de chaque noeud commence par la ligne suivante dans le `run{}`:

```
# ns-lib.tcl
[$self get-routelogic] configure
```

La premiere commande crée une instance de RouteLogic (`ns-route.tcl`) et retourne la référence sur cet objet. La méthode `configure{}` (`ns-route.tcl`) appelle `Agent/rtProto/Static init-all{}` (`route-proto.tcl`) qui elle même appelle `compute-routes{}` (`ns-route.tcl`) qui à sont tour appelle `compute-flat-routes{}` (`ns-route.tcl`). Cette fonction construit la représentation du graphe qui sera utilisée par l'algorithme de Dijkstra. Ensuite pour chaque noeud, elle détermine le next-hop pour une route vers chacun des autres noeuds. Le calcul du next-hop est effectué par `lookup{}` (`ns-route.tcl`).

```
Simulator instproc compute-flat-routes{} {
[...]
        set nh [$r lookup $i $j]
        if { $nh >= 0 } {
                $node add-route $j [$link_($i:$nh) head]
        }
[...]
}
```

`Lookup {}` demande le next-hop pour le noeud identifié par `i` pour atteindre le noeud identifié par `j`. `Lookup{}` appelle la fonction C++ correspondante (via le mécanisme `command()`). Si un next-hop existe, une entrée est ajoutée dans le classifier au slot identifié par j par `add-route{}`. `add-route{}` ajoute une route dans le noeud. Le classifier joue le rôle de table de routage du noeud.

Pour configurer la table de routage manuellement, un exemple est proposé dans `tcl/ex/many_tcp.tcl`.

7. Autres composants et fonctionnalités

7.1. Politiques de gestion de files d'attente et d'ordonnancement

La file d'attente représente l'endroit où les paquets sont conservés ou jetés. L'ordonnancement identifie la décision du choix du paquet à servir ou à jeter. La gestion de la file d'attente identifie les disciplines utilisées pour réguler l'occupation d'une file d'attente.

La classe Queue constitue la classe de base pour les différentes politiques de gestion de files d'attente et d'ordonnancement mise en oeuvre dans NS. Elle comporte un mécanisme de callback présenté dans le paragraphe "traitement séquentiel en temps simulé". Cette classe ne fait pas d'opérations de manipulation sur une structure de donnée de stockage de paquets. La classe PacketQueue est dédiée à cela. Elle est mise en oeuvre sous forme d'une liste chaînée de paquets.

7.2. Agent/Loss

Cet agent est un puits de trafic. Il utilise UDP et la numérotation en séquence de RTP. Il sert à comptabiliser la réception des paquets. En surveillant les numéros de séquence des paquets, il détecte les pertes. Il comporte les compteurs suivants:

- `nlost_`, nombre de paquets perdus
- `npkts_`, nombre de paquets reçus

- `nbytes_`, nombre d'octets reçus
- `lastPktTime_`, heure du dernier paquet arrivé
- `expected_`, numéro du prochain paquet attendu

7.3. Agent/Message

L'agent Message est très simple. Il permet de véhiculer des messages de taille maximum de 64 octets dans des paquets. Il est utilisé dans les modéles du multipoint.

7.4. Erreurs binaires

Des erreurs binaires peuvent être introduites dans le modèle. Elles sont introduites au niveau des liens au moyen de deux classes:

- ErrorModule,
- ErrorModel.

Les fonctions OTcl sont définies dans `tcl/lib/ns-errmodel.tcl`. Les deux classes sont définies dans le fichier `errormodel.cc`.

La classe ErrorModule sert à indiquer au niveau d'un lien les flux affectés par les erreurs. Elle consiste principalement en un classifier qui sera inséré dans le lien. Les avantages qu'il apporte sont que les erreurs peuvent affecter un flot donné. Il peut y avoir aussi plusieurs types de modèles d'erreurs différents au sein du même lien. Enfin il peut collecter tous les paquets en erreur des différents modèle d'erreurs. Les flots indéterminés sont affectés à son entrée `default`. L'insertion de ErrorModule dans un lien est effectuée par `SimpleLink errormodule{}`.

La classe ErrorModel détermine quel paquet doit être altéré (drop). Beaucoup de modèles d'erreurs sont dérivés de cette classe de base. Elle est elle-même dérivée de la classe Connector. Ce qui veut dire qu'elle peut être directement mise dans le lien sans avoir recours à ErrorModule. L'altération des paquets est effectuée de manière probabiliste, déterministe ou selon un fichier de traces. L'unité d'erreur est le paquet, le bit ou le temps. On peut également spécifier la sorte de paquet altéré (voir par exemple les classes SRMErrorModel, MrouteErrormodel).

A titre d'exemple, nous presentons comment mettre un ErrorModel sur un lien entre le noeud 1 et le noeud 2:

```
$ns duplex-link $node1 $node2 10Mb 10ms DropTail

#Set up random variable
set rng [new RNG]       # Random Number Generator
$rng seed 0
set rv [new RandomVariable/Uniform]
$rv use-rng $rng
$rv set min_ 0.0
$rv set max_ 1.0
#Set up bit errors
set em [new ErrorModel]
$em ranvar $rv
$em set rate_ 0.0012
```

```
set err [open err.tr w]
set tr [$ns create-trace Loss $err $node1 $node2]

$ns lossmodel $em $node1 $node2
```

Pour insérer, un modèle d'erreurs dans les deux directions du lien, il faut utiliser `install-error{}` à la place de `lossmodel{}`:

```
$self install-error $src $dst $e1
```

7.5. Applications

La classe Application modélise l'application en terme de source de trafic. L'application est associée avec un agent qui correspond à l'entité de transport. L'interface entre l'agent et l'application reprend celle des sockets (cf `agent.h`). Les applications sont de deux types:

- sources, elles sont employées à générer un flux pour un transport TCP. On y trouve Telnet et Ftp.
- générateur de trafic, ils sont employés pour les transports en mode non connecté comme UDP. On y trouve CBR, Exponentiel, Pareto. Le trafic peut également être généré à partir d'un fichier de traces contenant l'heure et la longueur des données à générer.

8. Ajout d'éléments dans NS

Pour étendre les fonctionnalités de l'interpréteur, il faut ajouter une ligne dans le fichier `ns-lib.tcl` comme par exemple:

```
source tcl/lan/ns_vlan.tcl
```

Le tutorial de Marc Greis contient un exemple de l'ajout d'un nouveau protocole. Lorsque ce protocole est au-dessus de IP ou utilise IP, il est codé sous forme d'un agent (classe dérivée de la classe agent). Si ce n'est pas le cas, il doit être codé comme une classe dérivée de la racine (classe NsObject).

Les étapes de l'ajout d'un protocole dans NS sont:

- déclaration de l'en-tête du paquet du protocole (ou unité de données du protocole)
- déclaration de la classe du protocole (méthodes et attributs du protocole). Le nom des attributs est suffixé par le caractère "_".
- définition de la liaison entre le code C++ et le code OTcl par la déclaration des variables statiques dérivées de la TclClass et de PacketHeaderClass respectivement pour l'agent et pour le paquet (ou unité de données) du protocole.
- ajout du protocole ID dans `packet.h` par un `#define`. L'indicateur de fin de liste des protocoles ID est `PT_NTYPE`. Le protocole ID sert à identifier le type de paquet.
- ajout du nom du protocole dans `PT_NAMES` (`packet.h`). L'ajout doit se faire à la position indiquée par la valeur du protocole ID définie précédemment. Pour les traces, la liste sert à remplacer l'ID du protocole par une chaîne de caractères.

- ajout du paquet dans la liste gérée par le packet manager. Ceci est rendu nécessaire pour le calcul de l'offset de l'en-tête du paquet.
- enfin modification du `makefile.in` par l'ajout du fichier objet du protocole à la liste des fichiers objet de ns.
- recompilation de ns

Lorsque l'on a un modèle à écrire on a le problème de choisir ce qui doit être en Tcl et en C++. Tcl est facile à écrire et à maintenir tandis que C++ est considérablement plus rapide à la simulation et plus économique en mémoire. Le conseil que l'on peut donner est d'écrire les fonctions centrales et intensément utilisées en C+ et le code en expérimentation en Tcl.

9. Script de simulation

Le script de simulation consiste à indiquer la topologie du réseau, à activer des traces aux endroits pertinents, à engendrer des événements particuliers à des instants donnés. De nombreux exemples sont disponibles dans la distribution, citons les plus simples:

```
example.tcl
simple.tcl
test-tbf.tcl
vlantest-flat.tcl
```

L'API OTcl de NS est principalement définie dans le fichier `tcl/lib/ns-lib.tcl`.

On peut avoir également recours à des générateurs de scripts. Le générateur crée et exécute une simulation pour des spécifications données d'agent, de topologie et de routage. Des informations complémentaires sont accessibles à l'URL: `http://www-mash.CS.Berkeley.EDU/ns/ns-scengeneration.html`.

10. Deboggage

Pour débugger un script de simulation, il faut d'abord utiliser l'interpréteur comme d'un débuggeur en arrêtant le script juste avant l'erreur et utiliser les différentes commandes Tcl pour visualiser les états du modèle. Pour cela OTcl fournit la méthode `info{}` bien utile pour retrouver des informations sur les états des objets et des classes:

```
<classe> info superclass        liste des super-classes
<classe> info subclass          liste des sous-classes
<classe> info heritage          hierarchie d'heritage
<classe> info instances         liste des instances
<classe> info instprocs         liste des insprocs
<classe> info instcommands      liste des méthodes qui sont définies en C++
<classe> info instargs <instproc>       les arguments de l'Insproc
<classe> info instbody <instproc>       le corps de la méthode
<classe> info instdefault <instproc>    la valeur par défaut de l'argument

<objet> info class              classe de l'objet
<objet> info vars               listes des instances variables de l'objet
```

Si l'erreur est dans le code C++, il faut vérifier si la compilation a été effectuée avec l'option `--enable-debug`. Au moyen de gdb, il faut localiser l'erreur. Cependant il peut être également nécessaire d'executer le code Tcl pas à pas dans gdb. Dans ce cas il faut utiliser le débuggeur Tcl tcl-debug1.17. La compilation doit être faite avec l'option `--tcl-debug`. La page `http://www-mash.CS.Berkeley.EDU/ns/ns-debugging.html` décrit comment effectuer le déboggage tcl-debug via gdb.

La méthode `gen-map{}` de Simulator peut également aider à l'identification des objets. Elle liste la totalité des objets. Cette liste peut s'avérer très utile pour l'utilisation des débuggeurs.

IV. Exemple: nouveau protocole

L'exemple du protocole "ping" développé dans [3, 4] est repris dans cette section. Nous allons illustrer les propos de la section précédente et montrer comment un nouveau protocole s'insère dans NS. Le protocole "ping" consiste à ce qu'un noeud émette un paquet vers un autre noeud. Le noeud de destination réfléchit le paquet à la source. Le temps de transfert aller- retour est ensuite calculé.

Pour ajouter un protocole, il faut localiser sa place dans l'architecture de TCP/IP. Si c'est un protocole au dessus de IP, la classe agent est recommandée. La classe Agent met en place l'interface pour générer et manipuler un paquet IP. Si c'est un protocole de transmission, il peut être intéressant de dériver la classe LinkDelay pour utiliser le mécanisme de blocage lorsqu'il y a un paquet en transit (voir Traitement séquentiel en temps simulé).

Le protocole "ping" utilise IP, ce sera donc une classe dérivée de la classe Agent. Il convient maintenant d'écrire le code C++ de ce composant. Dans le fichier d'en-tête `'ping.h'`, il faut déclarer la structure de l'unité de données de protocole. Dans la terminologie NS, on parlerait (à tort) de paquet ping. La déclaration du paquet ping consiste à définir un en-tête de paquet qui sera ajouté par le PacketHeaderManager aux en-têtes que comportera un paquet de NS (voir § Paquet et en-tête).

```
struct hdr_ping {
  char ret;
  double send_time;
};
```

Le champ `ret` prend la valeur 0 quand le paquet est émis par la source et 1 quand il est réfléchi par la destination. Ce champ sert à distinguer un paquet émis d'un paquet réfléchi. Le champ `send_time` est l'estampille temporelle. Il contient l'instant d'émission du paquet (en heure simulée) et servira au calcul du temps de transfert aller et retour.

La classe PingAgent est définie comme une sous-classe de Agent.

```
class PingAgent : public Agent {
 public:
  PingAgent();
  int command(int argc, const char*const* argv);
  void recv(Packet*, Handler*);
 protected:
  int off_ping_;
};
```

La classe PingAgent comporte un constructeur et deux fonctions qui sont redéfinies (command() et recv()). La variable `off_ping_` sert à accéder à l'en-tête du protocole ping (`struct hdr_ping`) contenu dans un paquet NS. Elle indique la localisation de l'en-tête `hdr_ping` dans la zone des en-têtes du paquet NS. La variable `off_ping_` d'un PingAgent est initialisée en étant un attribut lié avec l'attribut `off_ping_` d'un SplitObject. Le SplitObject est la classe racine d'un objet de l'interpréteur. Un objet PingAgent dans l'interpréteur hérite des attributs de la classe SplitObject. La valeur de

l'attribut `off_ping_` du SplitObject est calculée dans l'interpréteur à la création du simulateur (`new Simulator`) par la fonction `ns-packet.tcl:create_packetformat{}`.

Le constructeur PingAgent établit les liaisons entre les attributs de l'objet de l'interpréteur et celui du simulateur.

```
PingAgent::PingAgent() : Agent(PT_PING)
{
  bind("packetSize_", &size_);
  bind("off_ping_", &off_ping_);
}
```

Le constructeur de PingAgent appelle le constructeur de Agent avec comme argument le type du paquet (PT_PING). Cette information sera mise dans le paquet et sert principalement à identifier un protocole dans les traces. La valeur du littéral PT_PING est définie dans le fichier `packet.h` sous la forme d'une constante numérique au moyen d'un #define:

```
#define PT_SRM          16
#define PT_PING         17
#define PT_NTYPE        18 // Fin de liste
```

Dans le même fichier, il faut editer "PT-NAMES" pour inclure le nom sous lequel apparaîtra les traces pour ce protocole. A la 17ième position, ajouter la chaîne "Ping".

L'attribut `size_` est hérité de la classe Agent. Il indique la taille d'un paquet et servira à calculer le temps de transmission du paquet. La taille indique la longueur des données et des en-têtes (au moins IP). L'initialisation de la taille est effectuée via l'interpréteur. Dans le fichier `tcl/lib/ns-default.tcl`, il faut ajouter une ligne pour l'initialisation de la taille du paquet. Tous les objets de la classe Agent/Ping (nom de notre agent dans l'interpréteur) auront l'attribut `packetSize_` initialisé à la valeur 64 qui pour l'objet cloné dans le simulateur aura le nom d'attribut `size_`.

```
Agent/Ping set  packetSize_ 64
```

La classe PingAgent redéfinit `command()`. Cette fonction est appelée quand une commande Tcl pour un objet de la classe Agent/Ping n'a pas été localisée dans l'interpréteur. Cette fonction est un moyen bien pratique pour ajouter des commandes Tcl à l'interpréteur dont le code est écrit en C++.

```
int PingAgent::command(int argc, const char*const* argv)
{
  if (argc == 2) {
    if (strcmp(argv[1], "send") == 0) {
              // Create a new packet
      Packet* pkt = allocpkt();
              // Access the Ping header for the new packet:
      hdr_ping* hdr = (hdr_ping*)pkt->access(off_ping_);
              // Set the 'ret' field to 0, so the receiving node knows
              // that it has to generate an echo packet
      hdr->ret = 0;
              // Store the current time in the 'send_time' field
      hdr->send_time = Scheduler::instance().clock();
              // Send the packet
      send(pkt, 0);
              // return TCL_OK, so the calling function knows that the
              // command has been processed
```

```
      return (TCL_OK);
    }
  }
  // If the command hasn't been processed by PingAgent()::command,
  // call the command() function for the base class
  return (Agent::command(argc, argv));
}
```

`PingAgent::command()` définit la nouvelle commande Tcl "send". L'envoi consiste à préparer un paquet et à l'envoyer par l'instruction send(pkt, 0). Cette ligne appelle la fonction `Connector::send{target_->recv(p,h);}`. Rappelons qu'un agent est dérivé d'un connector. La target de l'agent est le point d'entrée du noeud où est attaché l'agent. C'est souvent le classifieur d'adresses. La variable `target_` est initialisée lors de l'attachement de l'agent à un noeud (`Simulator attach-agent{}`).

La dernière fonction à définir pour un PingAgent est celle de la réception d'un paquet. Elle est appelée par le port classifier du noeud. Lorsque cet objet reçoit un paquet, il détermine à quel agent remettre le paquet et appelle recv() de cet agent.

```
void PingAgent::recv(Packet* pkt, Handler*)
{
          // Access the IP header for the received packet:
  hdr_ip* hdrip = (hdr_ip*)pkt->access(off_ip_);
          // Access the Ping header for the received packet:
  hdr_ping* hdr = (hdr_ping*)pkt->access(off_ping_);

          // Is the 'ret' field = 0 (i.e. the receiving node is being pinged)?
  if (hdr->ret == 0) {
          // Send an 'echo'. First save the old packet's send_time
    double stime = hdr->send_time;
          // Discard the packet
    Packet::free(pkt);
          // Create a new packet (Agent::allocpkt)
    Packet* pktret = allocpkt();
          // Access the Ping header for the new packet:
    hdr_ping* hdrret = (hdr_ping*)pktret->access(off_ping_);
          // Set the 'ret' field to 1, so the receiver won't send another echo
    hdrret->ret = 1;
          // Set the send_time field to the correct value
    hdrret->send_time = stime;
          // Send the packet
    send(pktret, 0);
  } else {
          /* A packet was received. Use tcl.eval to call the Tcl
             interpreter with the ping results.
             Note: In the Tcl code, a procedure 'Agent/Ping recv {from rtt}'
             has to be defined which allows the user to react to the ping
             result.  */
    char out[100];
          // Prepare the output to the Tcl interpreter. Calculate the round
          // trip time
    sprintf(out, "%s recv %d %3.1f", name(),
            hdrip->src_ >> Address::instance().NodeShift_[1],
            (Scheduler::instance().clock()-hdr->send_time) * 1000);
    Tcl& tcl = Tcl::instance();
    tcl.eval(out);
          // Discard the packet
    Packet::free(pkt);
  }
}
```

A noter que l'attribut `off_ip_` est défini et géré par la classe Agent. Lorsque le paquet réfléchi arrive à la source, le résultat est affiché par l'interpréteur au moyen de l'instance procédure `recv{}` de Agent/Ping appelée à partir du code C++. L'appel d'une commande de l'interpréteur est effectué par la fonction `Tcl::eval()`, l'argument est la commande Tcl à faire évaluer.

Pour terminer l'écriture du code C++ de l'agent ping, il faut définir la liaison entre les objets du simulateur et ceux de l'interpréteur. A la création d'un objet dans l'interpréteur, un objet équivalent est crée dans le simulateur. Tout d'abord, nous allons effectuer la liaison pour l'en-tête du paquet ping. Bien que dans notre exemple, nous n'accédons pas depuis l'interpréteur aux champs de l'en-tête du paquet ping, nous devons effectuer la liaison avec l'interpréteur pour le calcul de localisation (offset) de l'en-tête ping dans la zone mémoire des en-têtes du paquet. Ce calcul est effectué par l'interpréteur à l'aide du PacketHeaderManager.

```
static class PingHeaderClass : public PacketHeaderClass {
public:
  PingHeaderClass() : PacketHeaderClass("PacketHeader/Ping",
                                        sizeof(hdr_ping)) {}
} class_pinghdr;
```

La liaison pour le paquet ping est définie par la variable statique class_pinghdr de la classe PingHeaderClass. La classe de cette variable est dérivée de la classe PacketHeaderClass qui constitue la racine pour l'enregistrement de tous les en-têtes de paquets. Le constructeur PacketHeaderClass() comporte deux arguments:

- le nom de la classe OTcl pour l'en-tête. Le nom est toujours préfixé par "PacketHeader/" qui est la classe racine OTcl.
- la taille de l'en-tête.

```
static class PingClass : public TclClass {
public:
  PingClass() : TclClass("Agent/Ping") {}
  TclObject* create(int, const char*const*) {
    return (new PingAgent());
  }
} class_ping;
```

L'enregistrement de la classe Agent/Ping dans l'interpréteur est effectué par le constructeur PingClass de l'objet static class_ping qui est construit au démarrage de l'application NS. Une instance Agent/Ping est créée à partir de l'interpréteur par la commande: `new Agent/Ping`. Cette commande appelle via `create_shadow()` la méthode `PingClass::create()` pour la création d'un `PingAgent` dans le simulateur. Nous avons ainsi la création de deux objets Agent/Ping: un pour l'interpréteur et son clone `PingAgent` pour le simulateur. Le code complet de ping est fourni en annexe de ce document.

Enfin pour utiliser l'en-tête hdr_ping dans un paquet NS, il faut indiquer au PacketHeaderManager d'initialiser l'attribut `off_ping_`. Dans le fichier `tcl/lib/ns-packet.tcl`, il faut ajouter à la liste des protocoles la ligne : `{ Ping off_ping_ }`. Le protocole ("Ping") doit avoir le même nom et la même syntaxe que le suffixe du nom du paquet OTcl ("PacketHeader/Ping").

Pour la compilation de ping, il faut ajouter "ping.o" à la variable OBJ_CC du makefile. Ensuite vous pouvez compiler NS simplement en tapant "make". Une fois, l'exécutable NS prêt, il ne reste plus qu'à tester le protocole ping par ce petit script :

```
#Define a 'recv' function for the class 'Agent/Ping'. This function will be
# called by PingAgent::recv() function.
Agent/Ping instproc recv {from rtt} {
        $self instvar node_
        puts "node [$node_ id] received ping answer from \
              $from with round-trip-time $rtt ms."
}
#Create a simulator object
set ns [new Simulator]

#Open a trace file
set nf [open out.nam w]
$ns namtrace-all $nf

#Define a 'finish' procedure
proc finish {} {
        global ns nf
        $ns flush-trace
        close $nf
        exec nam out.nam &
        exit 0
}

#Create three nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]

#Connect the nodes with two links
$ns duplex-link $n0 $n1 1Mb 10ms DropTail
$ns duplex-link $n1 $n2 1Mb 10ms DropTail

#Create two ping agents and attach them to the nodes n0 and n2
set p0 [new Agent/Ping]
$ns attach-agent $n0 $p0

set p1 [new Agent/Ping]
$ns attach-agent $n2 $p1

#Connect the two agents
$ns connect $p0 $p1

#Schedule events
$ns at 0.2 "$p0 send"
$ns at 0.4 "$p1 send"
$ns at 0.6 "$p0 send"
$ns at 0.6 "$p1 send"
$ns at 1.0 "finish"

#Run the simulation
$ns run
```

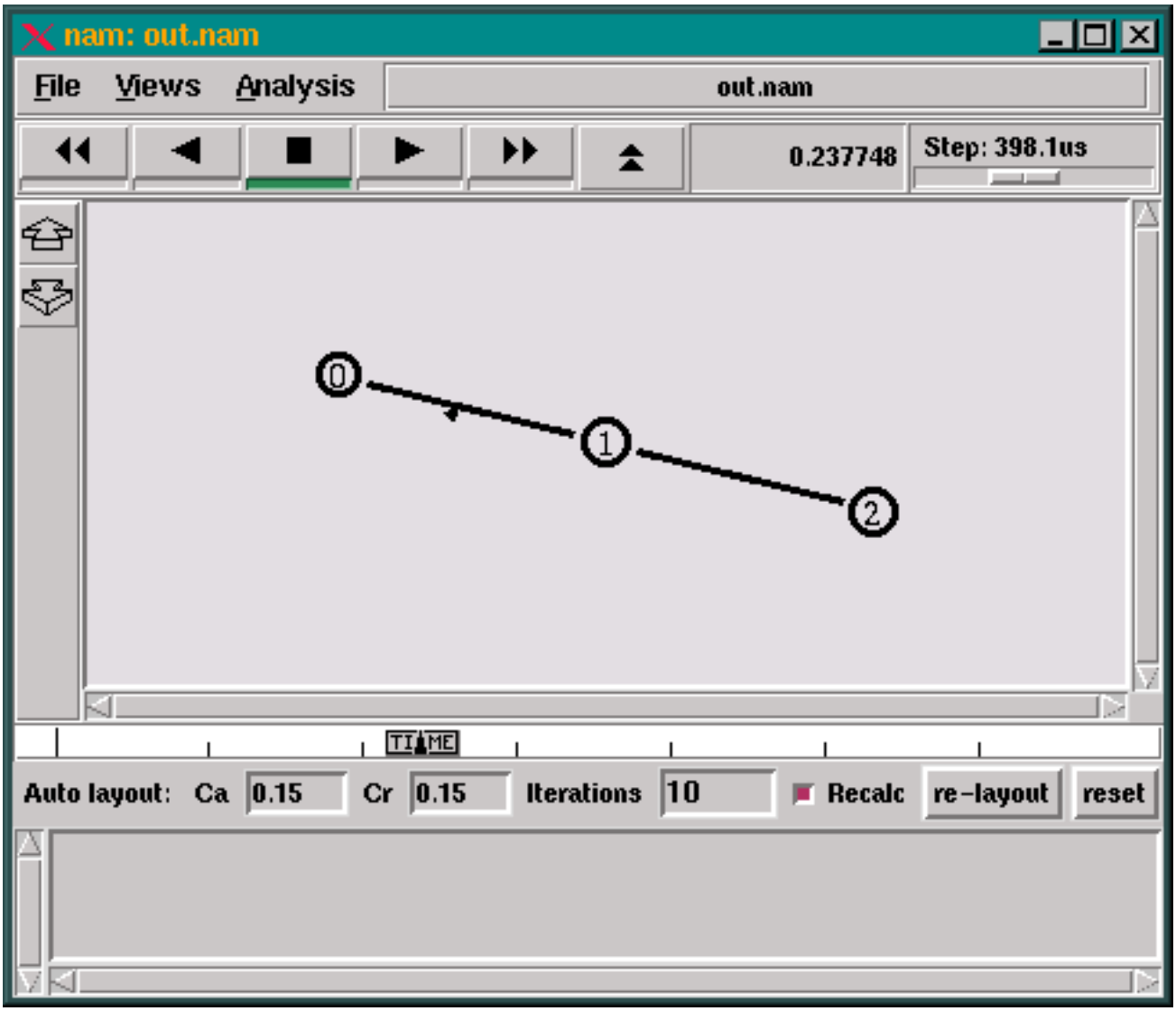


Figure 1: Execution d'un ping.

Bibliographie

[1] **Paxson, V. and Floyd, S.** (1997). Proceedings of the Winter Simulation Conference, Dec. 1997.
Why we don't know how to simulate the Internet.

[2] **Ousterhout, J.K.** (1994). Ed.: Addison-Wesley Publishing Company, 458 p.
Tcl and the TK Toolkit.

[3] **Fall, K. and Vradhan, K.** (1998). Oct. 1998.
http://www-mash.cs.berkeley.edu/ns/nsDoc.ps.gz.
NS Notes and Documentation.

[4] **Greis, M.** (1998). Oct. 1998.
`http://titan.cs.uni-bonn.de/~greis/ns/ns.html`
NS Tutorial.

Annexe: Variables d'instances des classes OTcl

Simulator

```
alltrace_
color_
interfaces_
link_
namConfigFile_
namtraceAllFile_
namtraceSomeFile_
Node_
nullAgent_
queueMap_
scheduler_
started_
traceAllFile_
```

Node

```
address_
agents_
classifier_
dmux_
id_
ifaces_
multiclassifier_
multiPath_
neighbor_
np_
ns_
routes_
rtObject_
switch_
```

Link

```
color_
cost_
deqT_
dest_
drophead_
drpT_
dynamics_
dynT_
enqT_
errmodule_
fromNode_
head_
ifacein_
ifaceout_
link_
ns_
qMonitor_
qtrace_
queue_
rcvT_
sampleInterval_
snoopDrop_
snoopIn_
snoopOut_
source_
toNode_
trace_
```

`ttl_`

Annexe: URL

http://www.scriptics.com/man/tcl8.0/contents.htm	Manuel de Tcl
http://www.svrloc.org/~charliep/mobins2/download/	Mobility support, Mobile IP et Wireless Channel Support pour NS-2
http://www-mash.cs.berkeley.edu/nam/nam.html http://mash.cs.berkeley.edu/dist/vint/nam-src-current.tar.gz	Visualisateur de NS
http://titan.cs.uni-bonn.de/~greis/rsvpns/	RSVP pour NS
http://titan.cs.uni-bonn.de/~greis/ns/ http://titan.cs.uni-bonn.de/~greis/ns/nstutorial.tar.gz	Tutorial de NS
http://www-mash.cs.berkeley.edu/ns/ns-topogen.html	Générateur de topologies réseau
http://freebsd1.lums.edu.pk/~umair/NS/doc/index.html	TCP/IP over ATM
http://www.isi.edu/~kannan/code/tcl-debug-1.7.tar.gz.	Tcl-debug package
http://www-mash.cs.berkeley.edu/ns/workshop3.html	3rd Workshop on NS
http://www.arl.wustl.edu/~sherlia/rm/pgm.tar.gz	Cisco's PGM protocol
http://networks.ecse.rpi.edu/~sunmin/rtProtoLS/	Agent/rtProto/LS (Link State)
ftp://uiarchive.cso.uiuc.edu/pub/packages/ddd/www/ddd.html	graphical debugger compatible avec NS
http://www.monarch.cs.cmu.edu/cmu-ns.html	CMU's mobility module
ftp://cc-lab.u-aizu.ac.jp/pub/sar_dist.tar.gz	segmentation and reassembly at the link layer (sar)
http://moat.nlanr.net/Routing/rawdata/ http://salamander.merit.edu/ipma/java/ASExplorer.html http://www.rsng.net	Mesures sur l'Internet

Annexe: Exemple Ping

ping.h

```
/*
 * File: Header File for a new 'Ping' Agent Class for the ns
 *       network simulator
 * Author: Marc Greis (greis@cs.uni-bonn.de), May 1998
 *
 */


#ifndef ns_ping_h
#define ns_ping_h

#include "agent.h"
#include "tclcl.h"
#include "packet.h"
#include "address.h"
#include "ip.h"


struct hdr_ping {
  char ret;
  double send_time;
};


class PingAgent : public Agent {
 public:
  PingAgent();
  int command(int argc, const char*const* argv);
  void recv(Packet*, Handler*);
 protected:
  int off_ping_;
};


#endif
```

ping.cc

```
/*
 * File: Code for a new 'Ping' Agent Class for the ns
 *       network simulator
 * Author: Marc Greis (greis@cs.uni-bonn.de), May 1998
 *
 */


#include "ping.h"


static class PingHeaderClass : public PacketHeaderClass {
public:
  PingHeaderClass() : PacketHeaderClass("PacketHeader/Ping",
                                        sizeof(hdr_ping)) {}
} class_pinghdr;


static class PingClass : public TclClass {
public:
  PingClass() : TclClass("Agent/Ping") {}
  TclObject* create(int, const char*const*) {
    return (new PingAgent());
```

```
  }
} class_ping;


PingAgent::PingAgent() : Agent(PT_PING)
{
  bind("packetSize_", &size_);
  bind("off_ping_", &off_ping_);
}


int PingAgent::command(int argc, const char*const* argv)
{
  if (argc == 2) {
    if (strcmp(argv[1], "send") == 0) {
      // Create a new packet
      Packet* pkt = allocpkt();
      // Access the Ping header for the new packet:
      hdr_ping* hdr = (hdr_ping*)pkt->access(off_ping_);
      // Set the 'ret' field to 0, so the receiving node knows
      // that it has to generate an echo packet
      hdr->ret = 0;
      // Store the current time in the 'send_time' field
      hdr->send_time = Scheduler::instance().clock();
      // Send the packet
      send(pkt, 0);
      // return TCL_OK, so the calling function knows that the
      // command has been processed
      return (TCL_OK);
    }
  }
  // If the command hasn't been processed by PingAgent()::command,
  // call the command() function for the base class
  return (Agent::command(argc, argv));
}


void PingAgent::recv(Packet* pkt, Handler*)
{
  // Access the IP header for the received packet:
  hdr_ip* hdrip = (hdr_ip*)pkt->access(off_ip_);
  // Access the Ping header for the received packet:
  hdr_ping* hdr = (hdr_ping*)pkt->access(off_ping_);
  // Is the 'ret' field = 0 (i.e. the receiving node is being pinged)?
  if (hdr->ret == 0) {
    // Send an 'echo'. First save the old packet's send_time
    double stime = hdr->send_time;
    // Discard the packet
    Packet::free(pkt);
    // Create a new packet
    Packet* pktret = allocpkt();
    // Access the Ping header for the new packet:
    hdr_ping* hdrret = (hdr_ping*)pktret->access(off_ping_);
    // Set the 'ret' field to 1, so the receiver won't send another echo
    hdrret->ret = 1;
    // Set the send_time field to the correct value
    hdrret->send_time = stime;
    // Send the packet
    send(pktret, 0);
  } else {
    // A packet was received. Use tcl.eval to call the Tcl
    // interpreter with the ping results.
    // Note: In the Tcl code, a procedure 'Agent/Ping recv {from rtt}'
    // has to be defined which allows the user to react to the ping
    // result.
    char out[100];
```

```
    // Prepare the output to the Tcl interpreter. Calculate the round
    // trip time
    sprintf(out, "%s recv %d %3.1f", name(),
            hdrip->src_ >> Address::instance().NodeShift_[1],
            (Scheduler::instance().clock()-hdr->send_time) * 1000);
    Tcl& tcl = Tcl::instance();
    tcl.eval(out);
    // Discard the packet
    Packet::free(pkt);
  }
}
```