Université Mohamed Khider – Biskra
Faculté des Sciences Exactes et des Sciences de la Nature et de la Vie
Département d’Informatique
2ème année Master
Option: Génie Logiciel et Systèmes Distribués
Module : Logique de Réécriture et ses applications
Responsable du module: Okba Tibermacine

Modélisation des communications en Maude II

Ce cours est basé sur les diapositives du prof. Peter Olveczky et Ingrid Chieh Yu

Aujourd'hui

- Modélisation de Réseaux
- Multicast via des Liens
- Protocoles de communication
  - Connexion ordonnée fiable via un média de communication non ordonnée et non fiable.
  - Alternating Bit Protocol
  - Sliding Window Protocol

Formes de communication

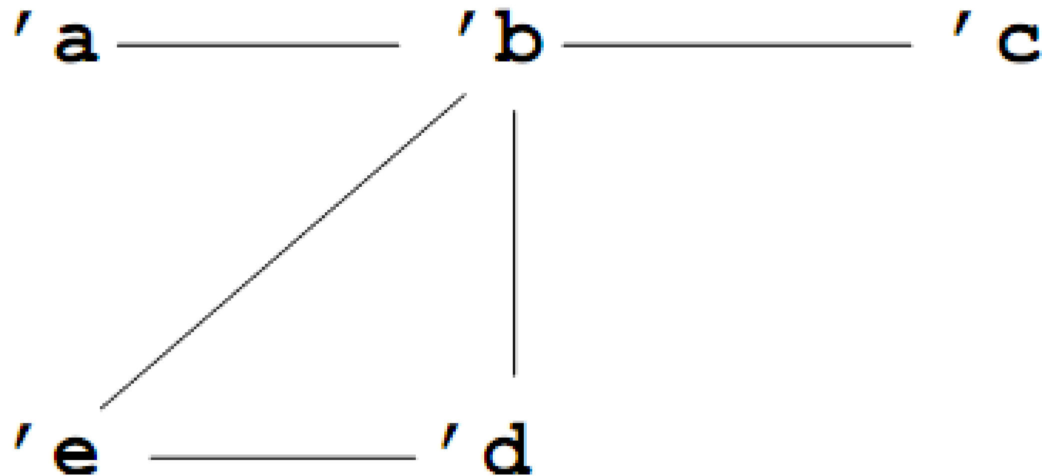
- Cours précédent
  - Communication synchrone.
  - Communication Asynchrone via un moyen de communication « non ordonné » :
    - Envoi de message à / lire de messages à partir d'une Configuration
    - Unicast: message d'un émetteur vers un récepteur.
    - Multicast: message d'un émetteur vers plusieurs récepteurs
      - Un multi-message est transformé à un ensemble de messages simples
    - Broadcast : message d'un émetteur à tous les autres nœuds
      - Un message-broadcasté est transformé à un messages simple dans tous les nœuds
    - Perte et duplication de messages

Formes de communication (II)

- Cours précédent
  - Communication Asynchrone via un moyen de communication « non ordonné » :
    - Messages envoyé via un objet lien (LINK Object)
    - Unicast
    - Perte et duplication de messages
    - Liens avec capacité limitée
    - Concurrence: lors de l’insertion/ surpression de message dans le même lien
    - Multicast non modélisé !

En outre, On doit étudier le multicast via des liens,

Modélisation d’un réseau



- Un nœud dans un réseau est modélisé par un objet. Une liaison peut être représentée par des liens, ou autrement (liste d’djacence ,,etc.)

Modélisation des réseaux avec des listes d’adjacence (de voisinage)

- Chaque nœud / objet à une liste d’adjacence (de voisinage)

```
class Node | neighbors : OidSet, ...

sort OidSet . subsort Oid < OidSet .
op none : -> OidSet [ctor] .
op _;_ : OidSet OidSet -> OidSet [ctor assoc comm id: none] .
```

- Le réseau précédent peut être représenté par l’état :

```
< 'a : Node | neighbors : 'b >
< 'b : Node | neighbors : 'a ; 'c ; 'd ; 'e >
< 'c : Node | neighbors : 'b >
< 'd : Node | neighbors : 'b ; 'e >
< 'e : Node | neighbors : 'b ; 'd >
```

Réseaux via des liens

- Nœuds dans le graphe : Objets de type Nœud
- Liaison dans le graphe : Objets de type Lien
- Topologie non-orientée : deux liens unidirectionnel ou bien une nouvelle classe pour des liens « bidirectionnel » ?
- État initial :

```
< ’a : Node | neighbors : ’b >
< ’a to ’b : Link | content : nil >
< ’b to ’a : Link | content : nil >
< ’b : Node | neighbors : ’a ; ’c ; ’d ; ’e >
< ’b to ’c : Link | content : nil >
< ’c to ’b : Link | content : nil >
...
```

Multicast via des liens

- Envoi d'un message à plusieurs voisins via un lien dans un seul pas de réécriture:
  - 2 tentatives :
    - une qui ne fonctionne pas correctement !
    - une approche plus compliqué mais qui fonctionne perfaitement.
  - Le modèle (non seulement une « exécution Maude ») doit être correct .

Multicast via des liens : Tentative 1

- Tentative 1 : l'objet envoi un message 'un par un'

```
rl [initSending] :
      < O : Node | neighbors : OS, state : available >
      =>
      < O : Node | state : send M to OS > .

crl [intoLink] :
      < O : Node | state : send M to O' ; OS >
      < O to O' : Link | content : L >
      =>
      < O : Node | state : send M to OS >
      < O to O' : Link | content : L :: M >
if OS =/= none .
```

Multicast via des liens : Tentative 1 (II)

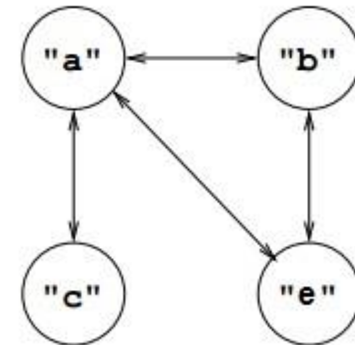
```
rl [lastIntoLink] :
     < O : Node | state : send M to O’ >
     < O to O’ : Link | content : L >
     =>
     < O : Node | state : available >
     < O to O’ : Link | content : L :: M > .
```

- Utilisation de plusieurs pas de réécriture
- Ce n’est pas un envoi simultané à tous les récepteurs !

Multicast via des liens : épandeurs (Spreaders)

- Un objet épandeur « Spreader » est connecté aux liens

```
class Spreader | dest : OidSet,
                 msgToSend : DefMsg,
                 sendTo : OidSet .
```



Exemple de topologie

```
op spreader : Oid -> Oid [ctor] .
```

- Un message envoyé au épandeur utilise des équations afin d'insérer le message dans tous les liens entre l'émetteur et les récepteurs.

Objet : épandeurs (Spreaders)

- Un objet épandeur « Spreader » dans un état donné

```
< spreader("a") : Spreader | dest : "b" ; “c" ; "e",
                              msgToSend : m,
                              sendTo : “c" ; "e" >
```

- Épandeur du nœud "a" .
- (dest:) les nœuds destinataires dans les liens issus de "a" sont les objets "b", "d" et "e“.
- il doit envoyer/répandre le message m dans les liens.
- sendTo : est les liens dans lesquels le message doit être inséré.
  - msgToSend vaut noMsg quand aucun message doit être inséré dans les liens.

Envoi via l'épandeur [spreader]

- Envoi du message m a tous les voisins via des liens et des épandeurs :

```
rl [send-m] :
< O : Node | >
< spreader(O) : Spreader | msgToSend : noMsg >
=>
< O : Node | >
< spreader(O) : Spreader | msgToSend : m > .
```

- Avantages :
  - Ne suppose pas que l'émetteur connaître ses voisins
  - …. Toutefois, la solution suppose que son épandeurs les connaitre.

- Prochaine étape : épandeur doit insérer m dans chaque lien de O

de l'épandeur vers les liens

- Comment l'objet épandeur insère un message dans chaque lien ?

--- Insertion récursive de message M dans tous les liens:

```
eq < spreader(O) : Spreader | msgToSend : M, sendTo : O' ; OS >
< O to O' : Link | content : L >
=
< spreader(O) : Spreader | sendTo : OS >
< O to O' : Link | content : L :: M > .
```

--- Insertion de message M dans les liens terminée

```
 eq < S : Spreader | dest : OS, msgToSend : M,
                          sendTo : none >
=
< S : Spreader | msgToSend : noMsg, sendTo : OS > .
```

Utilisation des épandeurs

- L’épandeur peut être utilisé pour envoyer des messages à quelques voisins via des liens.

```
rl [send-m-to-good-neighbors] :
< spreader(O) : Spreader | msgToSend : noMsg >
< O : Node | goodNeighbors : OS >
=>
< spreader(O) : Spreader | msgToSend : m, sendTo : OS >
< O’ : Node | > .
```

Etat initial avec épandeurs

```
(réseau de la page 5 )

  < "a" : Node | neighbors : "b" >
  < spreader("a") : Spreader | dest : "b",
                                        msgToSend : noMsg,
                                        sendTo : "b" >
  < "a" to "b" : Link | content : nil >
  < "b" to "a" : Link | content : nil >
  < "b" : Node | neighbors : "a" ; "c" ; "d" ; "e" >
  < spreader("b") : Spreader | dest : "a" ; "c" ; "d" ; "e",
                                   msgToSend : noMsg,
                                   sendTo : "a" ; "c" ; "d" ; "e" >
   ...
   L'attribut neighbors ne sont plus nécessaire
```

Zone Partageable

- Une variable partageable X avec une valeur n peut être vue comme un objet :

```
< X : SharedVar | value : n >
```

- Un objet O peut écrire dans la variable partageable X :

```
rl [updateX] :
< O : C | att1 : M >
< X : SharedVar | >
=>
< O : C | >
< X : SharedVar | value : M > .
```

- Un Objet O' peut lire la valeur de la variable partageable X :

```
rl [readX] :
< O' : C' | > < X : SharedVar | value : M >
=>
< O' : C' | a1 : M > < X : SharedVar | > .
```

Conclusion

- Quelques Techniques pour modéliser différent types de réseaux/communication
- Difficile ? Essayant avec Actor ou Java !
- Modélisant / exécutant quelques protocoles de communication réels

Modélisation d'un simple protocole de communication

Les protocoles assurent une communication fiable et ordonnée via un médium de communication non fiable et non ordonnée.

- Transporter une séquence de message dans ordre correcte dans un réseau qui peut perdre et/ou duplique des messages.
  - La duplication de messages et une abstraction de renvoyer un message quand un délai est expiré.
- 3 protocoles:
  - À base d'un numéro de séquence (TCP) – médium de communication non ordonné.
  - Alternating bit protocol – médium de communication ordonnée
  - Sliding Window protocol - médium de communication ordonnée et non ordonnée.

Premier protocole (I)

- échange de messages « Ordinaire » (non ordonné)
- Un médium de communication non fiable – les messages peuvent être perdus et/ou dupliqués.
- Supposant que la communication s'effectuée uniquement entre une seule paire d'objets.
- Elle est basée sur les numéros de séquences et aux accusés de receptions(acknowledgments).

Premier protocole (II)

- Protocole d'émetteur :

1. Envoi un message avec le numéro de séquence 1 sur plusieurs reprises jusqu'à la réception d'accusé de réception pour le numéro 1 …
2. Ensuite l'envoi du message avec un numéro de séquence 2 sur plusieurs reprises jusqu'à la réception d'accusé de réception pour le numéro 2 …
3. Et ainsi de suite ….
4. Jusqu'à la lecture du dernier accusé de réception pour le dernier numéro de séquence.
5. Ignorer toujours les accusés des numéros de séquences inférieure du numéro de séquence actuel.

Premier protocole (III)

- Protocole du récepteur:
  1. Envoi d'un accusé de réception sur plusieurs reprises pour le plus grand numéro de séquence reçu …
  2. Ignorer les messages avec un numéros de séquence <= au plus grand numéro déjà reçu.

Premier protocole (IV)

Modélisation de types de messages

1. On déclare les messages ordinaires comme des chaines de caractères :

```
sort MsgContent .
subsort String < MsgContent .
```

2. Message d'accusé de réception :

```
op ack : -> MsgContent [ctor] .
```

3. Enveloppes (wrappers) avec un numéro de séquence :

```
msg _withSeqNo_ : MsgContent Nat -> Msg .
msg msg_from_to_ : Msg Oid Oid -> Msg .
```

Premier protocole (IV)

Modélisation de la perte et la duplication des messages

```
var M : Msg . vars O O' : Oid .

rl [messageLoss] :
msg M from O to O' => none .

rl [duplicateMsg] :
msg M from O to O' => (msg M from O to O') (msg M from O to O') .
```

Transmettre une séquence de strings:

```
sort StringList .         subsort String < StringList .
op nil : -> StringList [ctor] .
op _++_ : StringList StringList -> StringList [ctor assoc id: nil] .
```

Premier protocole : protocole d’envoi

Protocole d’envoi :

```
class Sender | msgsToSend : StringList,
               currentMsg : StringList,
               currentSeqNo : Nat,
               receiver : Oid .
```

msgsToSend: messages non encore envoyés

currentMsg: message courant à envoyé

currentSeqNo: numéro de séquence du message courant

receiver: Object récepteur

Premier protocole : protocole d’envoi (II)

Début : soit le premier message le message courant à envoyé

```
vars N N’ : Nat . vars O O’ : Oid .
var S : String . var SL : StringList .

rl [start] :
< O : Sender | msgsToSend : S ++ SL,
                 currentSeqNo : 0 >
=>
< O : Sender | msgsToSend : SL, currentMsg : S,
                 currentSeqNo : 1 > .
```

Premier protocole : protocole d’envoi (III)

Envoi du premier message sur plusieurs reprises :

```
rl [sendCurrentMsg] :
< O : Sender |    currentMsg : S,
                  currentSeqNo : N,
                  receiver : O’ >
=>
< O : Sender | >
msg (S withSeqNo N) from O to O’
```

Premier protocole : protocole d’envoi (IV)

Réception d’un accusé de réception pour le message courant

```
Préparation à l’envoi du prochain message
    rl [receiveCurrentAckNotLast] :
    (msg (ack withSeqNo N) from O’ to O)
    < O : Sender | currentSeqNo : N,  msgsToSend : S ++ SL >
    =>
    < O : Sender | currentSeqNo : N + 1, currentMsg : S,
                   msgsToSend : SL > .
```

Remarque : on a besoin d’utiliser les parenthèses dans full Maude

Premier protocole : protocole d’envoi (V)

Réception d’un accusé de réception pour le dernier message .

```
Prépare l’envoi du prochain message
    rl [receiveCurrentLastAck] :
    (msg (ack withSeqNo N) from O’ to O)
    < O : Sender | currentSeqNo : N,
    msgsToSend : nil >
    =>
    < O : Sender | currentSeqNo : N + 1,
    currentMsg : nil > .
```

l’émetteur n’envoi plus de messages !

Premier protocole : protocole d’envoi (VI)

Réception d’un ancien accusé de réception.

```
Prépare l’envoi du prochain message
    crl [rcvTooOldAck] :
    (msg (ack withSeqNo N) from O’ to O)
    < O : Sender | currentSeqNo : N’ >
    =>
    < O : Sender | >
    if N < N’ .
```

Ceci termine le protocole d’envoi !

Premier protocole : protocole de réception (I)

- Le comportement du récepteur est simple:
- envoi en permanence (sur plusieurs reprises) des accusés de réception pour le plus grand numéro de séquence reçu.
- Pour analyse, on stocke les numéros de séquences reçus
  - Ca ne fait pas partie du protocole !

```
class Receiver | greatestSeqNoRcvd : Nat,
                 sender : Oid,
                 msgsRcvd : StringList .
```

Premier protocole : protocole de réception (II)

Envoi d'accusé de réception sur plusieurs reprises , si un message est reçu (greatestSeqNoRcvd > 0) :

```
var NZ : NzNat .


rl [sendAck] :
< O : Receiver | greatestSeqNoRcvd : NZ, sender : O' >
=>
< O : Receiver | >
msg (ack withSeqNo NZ) from O to O' .
```

Premier protocole : protocole de réception (III)

Réception d’un nouveau message :

```
rl [rcvNewPacket] :
(msg (S withSeqNo s N) from O’ to O)
< O : Receiver | greatestSeqNoRcvd : N,
                 msgsRcvd : SL >
=>
< O : Receiver | greatestSeqNoRcvd : s N,
                 msgsRcvd : SL ++ S > .
```

Premier protocole : protocole de réception (III)

Ignorer un message si ce n'est pas le prochain message attendu :

```
crl [rcvOldPacket] :
(msg (S withSeqNo N) from O' to O)
< O : Receiver | greatestSeqNoRcvd : N' >
=>
< O : Receiver | >
if N =/= s N' .
```

Ceci termine le protocole.

Premier protocole : protocole de réception (III)

- Pourquoi le protocole ne se termine pas ?
- Est-ce qu'on peut le rendre terminant ?

Premier protocole : Analyse (I)

Définir un état initial :

```
subsort String < Oid .
op init : -> Configuration .
eq init = < "Sender" : Sender | msgsToSend :
                       "Maude" ++ "est" ++ "très" ++ "Beau",
                        currentMsg : nil,
                        currentSeqNo : 0,
                        receiver : "Receiver" >
< "Receiver" : Receiver | greatestSeqNoRcvd : 0,
                           msgsRcvd : nil,
                           sender : "Sender" > .
```

Premier protocole : Analyse (II)

Premier teste : réécriture (rewriting):

```
Maude> (frew [1000] init .)
result Configuration :
< "Receiver" : Receiver | greatestSeqNoRcvd : 4,
               msgsRcvd : ("Maude" ++"est" ++ "très" ++ "beau"),
                       sender : "Sender" >
< "Sender" : Sender |
                     currentMsg : nil, currentSeqNo : 5,
                     msgsToSend : nil, receiver : "Receiver" >
 (msg ack withSeqNo 4 from "Receiver" to "Sender")
 ...
msg ack withSeqNo 4 from "Receiver" to "Sender"
```

Premier protocole : Analyse (III)

- La réécriture (rewriting): à donnée le résultat attendu .
- Pour rassurer de ce résultat on utilise la recherche : search
  - Le calcul est non terminant : On peut pas vérifié les états finaux
  - Un espace de recherche infini (pourquoi ?) : la recherche peut continue à l'infinie .
- Recherche sur les états indésirables :
  - Un bon signe que le calcule prend un temps long sans aboutir à un état indésirable
  - … mais ca n'assure pas l'absence d'atteindre un état indésirable.
- On peut rendre l'espace de recherche fini
  - E.g : on envoi pas plus de n+1 messages équivalant en supposant que le nombre maximum de messages perdus/dupliqués est n !

Premier protocole : Analyse (IV)

- Chercher sur un état dans les éléments enregistrés sont en désordre : "est" après "très" .

```
(search [1] init
=>+
C:Configuration
< "Receiver" : Receiver | msgsRcvd :
                    SL:StringList ++ “tres"
                    ++ SL’:StringList ++ “est"
                    ++ SL’’:StringList > .)
```

- Aucun résultat dans quelques minutes
- C: configuration représente les autres messages et objets !

Premier protocole : Analyse (V)

- Enquêter sur la perte des messages : "est" n’est pas reçu

```
(search [1] init
=>+
C:Configuration
< "Receiver" : Receiver | msgsRcvd :
                    “MAUDE“++ “tres"
                    ++ SL’:StringList > .)
```

- Aucun résultat dans quelques minutes

Premier protocole : Analyse (VI)

- Quand l'attribut greatestSeqNoRcvd est à 4 , les messages reçus doivent être dans un ordre correct :

```
(search [1] init
=>+
C:Configuration
< "Receiver" : Receiver | msgsRcvd : SL:StringList,
                  greatestSeqNoRcvd : 4 >
such that SL:StringList =/=
          "MAUDE" ++ "est" ++ "très" ++ "beau" .)
```

Autres protocoles

- Le premier protocole assure une communication fiable est ordonnée via un médium de communication non fiable et non ordonné.

- Les trois autre protocoles :

  1. Le « deuxième protocole »
  2. alternating bit protocol
  3. sliding window

Le deuxième protocole (TP- Group A)

- Le deuxième protocole est juste comme le premier, toutefois les messages sont envoyés via des liens,
  - Remarque: utiliser des liens de classe Link dans les règles, et laisser l'état initial avoir des liens de la sous-classe UnrelLink.
    - On peut tester les deux sur des liens fiables -relaibele links- (l'état initial contient des liens de classe Link) et pour les liens non fiable -unreliable links- (l'état initial contient des liens de la sous-classe UnreLink)

Alternating Bit Protocol (TP Groupe B)

Observation sur le deuxième protocole :

- Uniquement deux numéros de séquence adjacents sont utilisés dans un état
  - si l'émetteur a le numéro de séquence courant égale à 22, ainsi il recevra un ack avec le numéro de séquence 22 ou 21.
- On a besoin qu'a deux (2) numéros de séquence,
  - Adéquat pour les petits paquets, ainsi le nombre de séquence n'agrandit sur aucune limite.
- Ceci est le «Alternating Bit Protocol »
  - Prend le premier protocole
  - Utiliser des liens pour la communication
  - Utiliser uniquement les numéros de séquence 0 et 1.

Sliding Window Protocol – I (TP Groupe C)

Sliding Window Protocol généralise les protocoles précédents

- Inconvénient avec les autres protocoles : envoi d'un seul paquet et attendre la réception d'un acquittement (Ack) pour ceci avant d'envoyer le paquet suivant,
- On veut envoyer plus de paquets avant la réception d'un acquittement (ack),
- Sliding window: émetteur et récepteur ont une fenêtre (window) de paquet à envoyer et de recevoir.
- e.g. si la fenêtre est de taille 3, et le récepteur a envoyé un ack pour le paquet N° 15, l'émetteur ainsi peut envoyer les messages 16,17 et 18.
  - L'émetteur doit avoir ces message dans sa fenêtre (window) pour les envoyer autant de fois (sur plusieurs reprise),

Sliding Window Protocol – II

- Le récepteur contient une fenêtre de réception dont les messages reçus et non pas encore acquittés (not ack) sont entreposés temporairement,
- Exemple :

1. Récepteur a reçu et acquitté tout les paquets jusqu'au 15 ème paquet,
2. Récepteur reçois paquet 17
3. Récepteur entrepose paquet 17 dans sa fenêtre et acquitte (ack) le 15
4. Récepteur reçois le paquet 16
5. Là le recepteur peut acquitter (ack) le paquet 17 et enlever 17 et 16 de sa fenêtre temporaire,
6. Et ainsi de suite ,,,

Sliding Window Protocol – III

Optimisation en cas de communications via des liens : pour éviter des numéros de séquence larges, vous pouvez reproduire la solution de Alternating Bit :

- Avoir un némuro de séquence module N, ou N doit être au moins deux fois plus grand que la taille de la fenêtre.