TOULOUSE INP N7

DYNAMIC SOURCE ROUTING

Projet de Développement Formel de Systèmes

Gabriel Rinaldi - Anton Xu
3A-SIL
2025 - 2026

Sommaire

Résumé

Ce rapport présente le développement formel d'une modélisation du protocole Dynamic Source Routing (DSR) au moyen de la méthode Event-B et de l'outil Rodin. Le projet a consisté à construire un modèle en quatre niveaux de raffinements progressifs, allant d'un système de communication abstrait jusqu'à une version complète du protocole intégrant la découverte de route (RREQ/RREP), l'acheminement des données, et la maintenance via messages d'erreur (RERRP). Nous décrivons les structures de données introduites, les événements majeurs, ainsi que les principales difficultés rencontrées, notamment la gestion des liens dynamiques, la diffusion de messages et le traitement des données bloquées. Le rapport se termine par un bilan technique et plusieurs pistes d'amélioration pour enrichir ou optimiser le modèle.

Introduction

Ce document est le rapport d'un projet de l'ENSEEIHT de développement formel sur Dynamic Source Routing. Nous avons eu 1 mois pour en réaliser une modélisation la plus avancée possible. Nous avons fait 3 étapes de raffinement (DSR 1 à 3) de notre modèle de base (DSR 0). Nous avons réussi à introduire un mécanisme de découverte de route et de table de routage locale avancée avec une maintenance de route plus primaire. Nous commencerons par présenter l'architecture de développement, avant de détailler les différentes étapes de raffinements ainsi que les choix de conception effectués. Nous terminerons par un tableau de preuves, une animation du modèle sous ProB et le bilan du projet.

Architecture du développement

L'architecture de notre développement formel repose sur une approche en raffinements successifs, inspirée d'une modélisation progressive du protocole DSR (Dynamic Source Routing). L'objectif général a été d'introduire les éléments du protocole par couches, en garantissant à chaque étape la correction grâce aux invariants et aux preuves associées.

Notre architecture s'organise ainsi autour de quatre niveaux de modèles Event-B :

Niveau	Concepts introduits	Finalité
DSR_0	Communication, messages, topologie	Base abstraite de notre système
DSR_1	Différenciation entre arrivée et réception de messages	Création de l'ensemble des messages arrivés
DSR_2	Routes + tables de routages locales	Structure du système en routes
DSR_3	Découverte de route + maintenance de route + récupération de données bloquées	Implantation la plus complète possible du protocole DSR

Raffinements

▸DSR_0

Protocole de communication de base pour l'envoi, la réception, la perte de paquets de données et les changements de topologie du réseau.

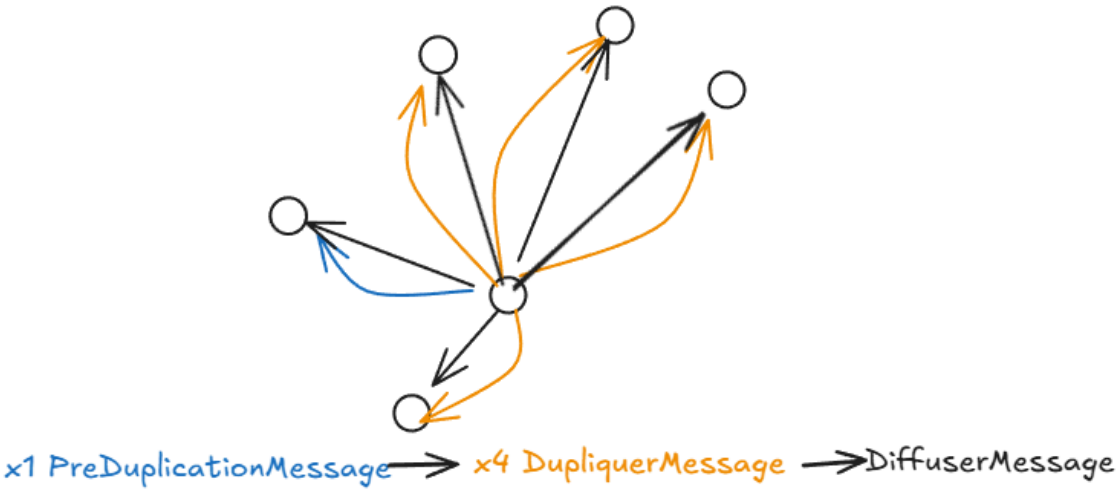
1. Variables créées

- Variables du contexte : ***MESSAGES***, ***NOEUDS*** et ***SAUTSMAX*** (nombre de sauts maximum que peut faire un même message dans le système, égale à 10 par défaut).
- ***Noeuds*** ⊆ NOEUDS :
  Tous les nœuds de notre système.
- ***Messages*** ⊆ MESSAGES :
  Tous les messages de notre système.
- ***Liens*** ∈ Noeuds ↔ Noeuds :
  Représente les liens unidirectionnels sur lesquels peuvent communiquer les nœuds.
- ***Source*** ∈ Messages → Noeuds :
  Chaque message créé a un nœud Source.
- ***Destination*** ∈ Messages → Noeuds :
  Chaque message créée a un nœud Destination.
- ***StockNoeud*** ∈ Messages ⇸ Noeuds :
  Un message peut être stocké sur un nœud.
- ***EnTransitVers*** ∈ Messages ⇸ Noeuds :
  Un message peut être en transit vers un autre nœud.
- ***MessagesPerdus*** ⊆ Messages :
  Un message peut être perdu via event Perdre ou Suppression d'un lien sur lequel transite un message.
- ***MessagesLiens*** ∈ dom(EnTransitVers) → Liens :
  Le lien sur lequel transite un message.
- ***NbSauts*** ∈ *Messages* → ℕ :
  Représente le nombre de sauts qu'a fait un message.
- ***DestDiffusion*** ∈ Messages ⇸ (Messages ⇸ Noeuds) :
  Représente les messages en cours de duplication dom(DestDiffusion) sont les messages d'origine à dupliquer et ran(DestDiffusion) sont **les duplications du message d'origine** en plus de lui-même**, chacune mappée à un destinataire différent**. Variable très utile dans la partie *DiffuserMessage.*

2. Événements créés

➜ Il est possible de **créer des liens, des nœuds et des messages** avec *AjoutNoeud*, *AjoutLien* et *AjoutMessage*. De plus, nous pouvons **supprimer un lien** à tout moment avec *SuppressionLien*, ce qui peut engendrer **la perte d'un Message** (s'il est actuellement sur le lien : variable *MessageLiens*) qui sera dans l'ensemble *MessagesPerdus*.

➜ **L'envoi d'un message** peut être fait soit par diffusion avec *DiffuserMessage*, soit par un envoi classique avec *Envoyer*. Afin de faire une diffusion, il faut au préalable **choisir le message qu'on veut diffuser** avec *PreDuplicationMessage* et **le dupliquer** autant de fois que l'on veut pour l'envoyer à autant de voisin que l'on souhaite avec *DupliquerMessage* (pas obligatoire dans ce raffinement il est possible de ne pas le dupliquer). Une fois les duplications réalisées (ou pas), le **message peut être diffusé** sur le lien à tous les voisins et **est prêt à être réceptionné ou perdu** avec *Recevoir* ou *Perdre*.



3. Invariants introduits

- $\forall m \cdot (m \in Messages \Rightarrow NbSauts(m) \leq SAUTSMAX)$:
  Aucun message ne dépasse le nombre maximal de sauts.
- $dom(StockNoeud) \cap dom(MessagesLiens) = \varnothing \land dom(MessagesLiens) \cap MessagesPerdus = \varnothing \land dom(StockNoeud) \cap MessagesPerdus = \varnothing$:
  Un message est soit perdu, soit sur un lien, soit en stock.

- $dom(StockNoeud) \cup dom(EnTransitVers) \cup MessagesPerdus = Messages$:
  Tous les messages sont soit stockés, soit en transit sur un lien, soit perdu (partition).
- $\forall m \cdot (m \in Messages \Rightarrow Source(m) \neq Destination(m))$:
  La source d'un message est différente de sa destination.
- $\forall m0 \cdot (m0 \in dom(DestDiffusion) \Rightarrow DestDiffusion(m0) \in Messages \nrightarrow Noeuds)$:
  Chaque duplication que l'on doit diffuser doit aller vers un nœud différent.
- $\forall l \cdot (l \in Liens \Rightarrow prj1(l) \neq prj2(l))$:
  Un lien ne peut pas se faire entre deux nœuds identiques.

▸DSR_1

Introduction de l'architecture store and forward pour les paquets
de données passant du nœud source au nœud de destination en passant par d'autres nœuds intermédiaires.

1. Variables créées

Ajout de la variable ***MessagesArrives*** :
Ensemble des messages arrivés à destination.

2. Événements créés

Dans ce raffinement, il est désormais possible de réceptionner un message avec deux événements : ***Recevoir* ou *Arriver***. Ces deux événements **raffinent *Recevoir.*** Arriver signifie que le message est arrivé à sa destination alors que recevoir est une simple réception.

3. Invariants introduits

- $\forall m \cdot (m \in MessagesArrives \Rightarrow Destination(m) = StockNoeud(m))$:
  Un message est arrivé s'il est dans le stock de son nœud destination.
- $\forall m \cdot (m \in dom(MessagesLiens) \Rightarrow m \notin MessagesArrives)$:
  Un message en transit sur un lien ne peut pas être arrivé et donc un message arrivé ne peut plus être envoyé.
- $dom(DestDiffusion) \cap MessagesArrives = \varnothing$:
  On ne duplique pas un nœud qui est déjà arrivé (inutile de le diffuser).

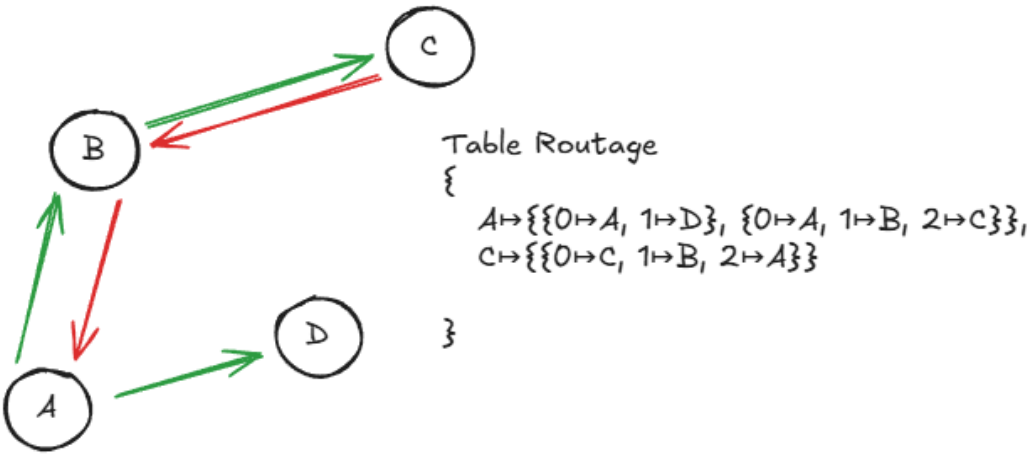
▸DSR_2

Ajout des tables de routage locale et de la notion de routes.

1. Variables créées

- ***Routes*** $\in \mathbb{P}(0 \cdot\cdot SAUTSMAX \nrightarrow Noeuds)$:
  L'ensemble des routes créées dans notre système. Une commence à 0 et peut faire des sauts de 1 à 1 entre tous les nœuds du système qu'importe l'existence des liens. Exemple : r = {0 ↦ n1, 1 ↦ n2, 2 ↦ n3} est la route qui part de n1 et qui va vers n3 en passant par n2, r appartient à Routes.

- ***TableRoutageLocale*** $\in Noeuds \rightarrow \mathbb{P}(Routes)$:
  La table de routage locale pour chaque nœud, c'est-à-dire que chaque nœud a un ensemble de routes qui représente sa table de routage locale (voir image ci-dessous).



2. Événements créés

➜ Dans ce raffinement, il est désormais possible de **créer des routes** à partir de la fonction *CreerRoute*, on peut créer la **route de notre choix** parmi tous les nœuds de notre système. Cette route est **ajoutable à la table de routage locale** du noeud auquel démarre la route (0 ↦ n) avec *AjouterRouteTable*. Il est aussi possible de **supprimer les routes** des tables de routages avec *SupprimerRouteTable.* Il est aussi possible de **tronquer une route** à partir d'un nœud d'une table de routage avec *TronquerRouteTable* (utile pour la maintenance de route plus tard).

Ex : si on veut tronquer r = {0 ↦ n1, 1 ↦ n2, 2 ↦ n3} à partir de nt = n3 alors r => {0 ↦ n1, 1 ↦ n2}.

➜ Comme *TableRoutageLocale* utilise **une fonction totale**, il faut ajouter le couple {n ↦ {}} à chaque *AjoutNoeud.*

3. Invariants introduits

- $\forall r \cdot (r \in Routes \Rightarrow r = \varnothing \lor (finite(dom(r)) \land (0..(card(dom(r)) - 1) = dom(r))))$:
  Une route est soit vide soit elle n'a pas de trou dans son domaine, c'est à dire r = {0 ↦ n1, 2 ↦ n3} est impossible.

- $\forall n, routes \cdot (n \mapsto routes \in TableRoutageLocale \Rightarrow (\forall r \cdot (r \in routes \Rightarrow r(0) = n)))$:
  Une table de routage locale d'un nœud n ne possède de routes qui ne commencent que par n (sinon la route est dans la mauvaise table de routage locale).

▸DSR_3

Ajout des mécanismes de découverte de route, maintenance de route et envoi de données grâce aux routes. Ajout de notion de Header pour les messages.

1. Variables créées

- Variable du contexte : ***MESSAGETYPE*** tel que *partition (MESSAGETYPE, {RREP}, {RREQ}, {RERRP}, {DONNEES})* :
  Représente tous les types de messages présents dans notre système.

- ***Header*** ∈ *Messages* → *(0‥SAUTSMAX ⇸ Noeuds)* :
  Ajout de notions de route dans les header selon les types en tant qu'en-tête. Pour les RREQ : le header permet de **sauvegarder la route empruntée** par le rreq à chaque saut du message.
  Pour les RREP, RERRP et DONNEE : le header représente **la route à suivre par les liens du système** (pour RREP : le header représente la route inverse).

- ***TypeMessage*** ∈ *Messages* → *MESSAGETYPE* :
  Représente le **type associé à chaque message**, on a mis une fonction totale car **chaque message a un type**.

- ***MessagesBloques*** ⊆ *Messages* :
  Représente les messages qui sont **bloqués dans le système**, dans ce raffinage cela ne représente que les données.

- ***LienCasse*** ∈ *Messages* ⇸ *(Noeuds ⇸ Noeuds)* :
  Lorsqu'un lien est cassé après qu'une route a été créée, et qu'une **donnée circule sur cette route et se retrouve bloquée** alors il y a un **envoi d'un message RERRP** vers le noeud source pour **tronquer la table de routage** en enlevant des routes toutes la partie qui concerne ce lien et les noeuds suivants de la route. (*rerrp* ↦ *{lienDisparu})* est ajouté à cet ensemble.

- ***RoutesTrouvee*** $\in \mathbb{P}(0\cdot\cdot SAUTSMAX \rightarrow Noeuds)$:
  Lorsqu'un RREQ est envoyé et qu'un RREP est reçu d'un nœud, ce nœud peut **créer une route avec *CreerRouteDecouverte* qu'il ajoutera à RoutesTrouvee** pour ensuite l'ajouter à sa table de routage locale.

- ***LienRREQetRREP*** $\in Messages \rightarrow Messages$:
  Permet de **lier un rreq arrivé à son rrep de retour**, ce qui permet de savoir si un RREP a déjà été créé pour un RREQ donné.

2. Événements créés

➜ Dans ce raffinement, il est désormais possible de découvrir des routes. Il faut pour cela envoyer un RREQ et recevoir un RREP en retour du nœud auquel on veut créer la route. Tout d'abord, l'implémentation de l'événement ***CreerRREQ*** qui raffine l'événement *AjoutMessage* qui peut être envoyé si un lien existe à des voisins qui ont un lien avec le nœud source.

Nous avons pour cela deux méthodes :

- ***EnvoyerRREQ* :** on prend un rreq et l'envoie si un lien existe et à chaque envoit on ajoute dans le header du message le noeud avec son nombre de saut pour obtenir la route qu'il a emprunté, ce qui **permet de créer la route que l'on souhaite** en envoyant le RREQ où l'on souhaite.

- **Diffuser des RREQ :** on sélectionne un rreq avec ***PreDuplicationRREQ*** (qui raffine `PreDuplicationMessage`), on le duplique si nécessaire pour pouvoir le diffuser à tous les voisins sauf de là d'où il vient avec ***DupliquerRREQ*** qui raffine *DupliquerMessage* (grâce au header, on l'empêche de faire marche arrière) . Dans ce raffinement le DupliquerMessage est obligatoire avant une diffusion, présence de gardes. Enfin une fois ces étapes réalisées on peut ***DiffuserRREQ*** qui raffine *DiffuserMessage*. Cette méthode **permet de créer toutes les routes possibles du système entre deux nœuds**.

➜ Pour ce qui est des RREP, ils ont juste la route des RREQ en header et font la **route inverse**, en utilisant les événements *CreerRREP*, *EnvoyerRREPouRERRP*, *RecevoirRREPouRERRP* et *ArriverRREP* qui raffinent respectivement *AjoutMessage*, *Envoyer*, *Recevoir* et *Arriver*.

➜ Une fois le mécanisme de RREQ et RREP fini, il est possible **si le RREP est arrivé** à destination du nœud source de **créer les routes découvertes.** Pour chaque RREP arrivé un événement *CreerRouteDecouverte* qui raffine CreerRoute peut être fait. Enfin, on **ajoute cette route à la table de routage** locale avec AjouterRouteTable.

➜ Les messages que s'échangent les noeuds sont désormais **des *DONNEES*** ce qui a conduit à la création des événements *CreerDONNEES*, *EnvoyerDONNEES*, *RecevoirDONNEES* et *ArriverDONNEES* qui raffinent respectivement *AjoutMessage*, *Envoyer*, *Recevoir* et *Arriver*. La particularité d'une donnée est que l'**on ne peut pas la créer sans une route** qui sera par la suite **son header**. L'envoi et la réception de cette donnée se fait par une route découverte au préalable.

➜ Un lien pouvant se supprimer à tout moment, une donnée **peut avoir une route dans son header pas à jour**. Dans ce cas, **la donnée reste bloquée** sur un nœud et l'événement *BloquerDONNEE* deviendra utilisable. *BloquerDonnee* est un nouvel événement qui ajoute cette donnée dans MessagesBloques.

➜ **Les Messages bloqués peuvent créer des RREP** qui peuvent être envoyés au nœud source dont sa table de routage n'est plus à jour, avec les événements *CreerRREP*, *EnvoyerRREPouRERRP*, *RecevoirRREPouRERRP* et *ArriverRERRP* qui raffinent respectivement *AjoutMessage*, *Envoyer*, *Recevoir* et *Arriver*. À la création d'un RERRP (qui a le même header qu'un RREQ), le lien cassé est ajouté au message d'erreur (dans *LienCasse*) et à l'arrivée du RERRP, il faut **tronquer la table de routage du nœud source** de la donnée. Après l'arrivée du message d'erreur, l'événement *TronquerRouteTable* est lancé pour tronquer toutes les routes locales du nœud ayant le lien rompu à l'intérieur.

➜ Une fois que la table de routage est mise à jour, la donnée est toujours bloquée sur un nœud avec une mauvaise route dans son header. **Si ce nœud a une route dans sa table de routage locale vers le nœud destination de la donnée**, il est **possible de débloquer cette donnée** avec *DebloquerDONNEE.* Cet événement consiste à mettre à jour le header de la donnée bloquée en lui **modifiant la route de son header**. **Sinon il faudra lancer un protocole de découverte de route** partant du nœud où la donnée est bloquée vers son nœud destination.

3. Invariants introduits

- $\forall m1,m2 \cdot (m1 \mapsto m2 \in LienRREQetRREP \Rightarrow TypeMessage(m1) = RREQ \land TypeMessage(m2) = RREP)$:
  Les messages dans LienRREQetRREP sont des RREQ mappés à des RREP.

- $\forall m \cdot (m \in dom(LienCasse) \Rightarrow TypeMessage(m) = RERRP)$:
  Les informations de liens cassés sont stockées dans des paquets d'erreur de route (RERRP).

- $\forall m,lc \cdot (m \mapsto lc \in LienCasse \Rightarrow finite(lc) \land card(lc) = 1)$:
  Il n'y a qu'une information de lien cassé par RERRP.

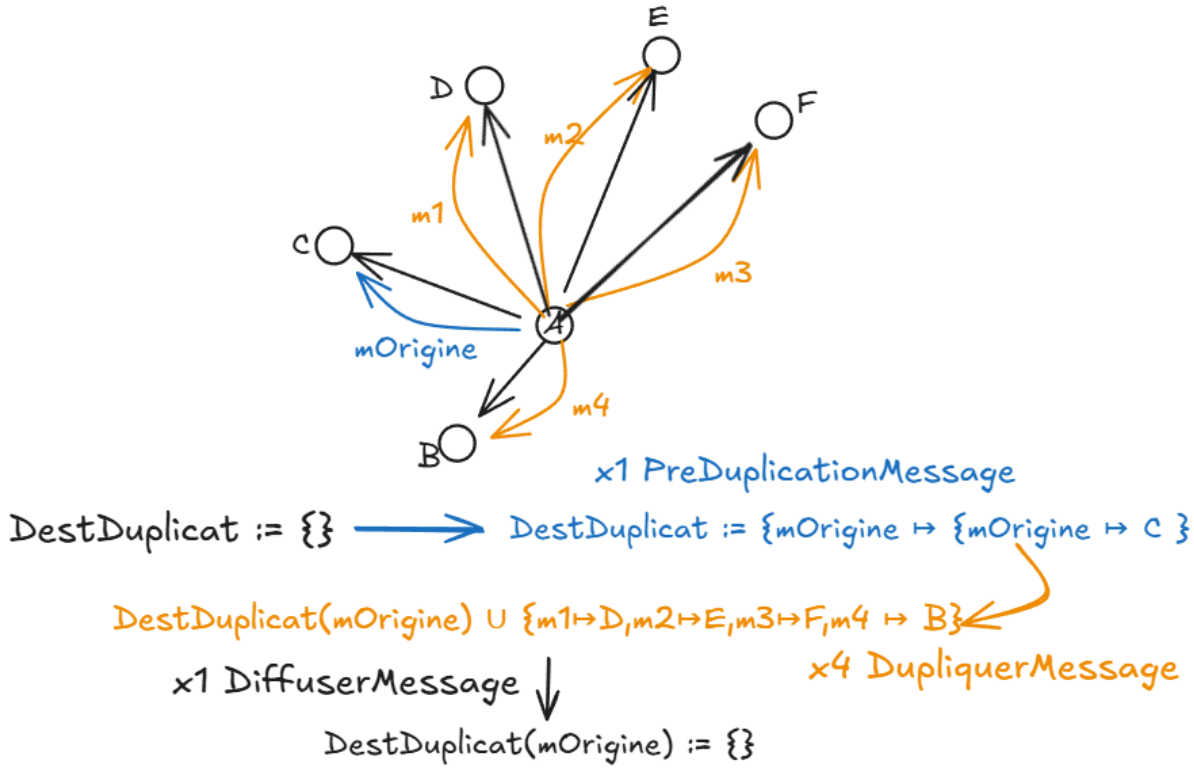
- $\forall h \cdot (h \in ran(Header) \Rightarrow h = \varnothing \lor (finite(dom(h)) \land (0‥(card(dom(h)) - 1) = dom(h))))$:
  Comme les routes, les en-têtes de messages n'ont pas de trou dans son domaine.

Principaux choix réalisés, difficultés rencontrées et solutions

Suppression de liens

Nous avions commencé notre projet en utilisant un ensemble *EnTransitVers* comme en TP. Ce qui était problématique lors de *SuppressionLien* qui peut être fait à tout moment et peut engendrer la perte d'un message dans le cas où un message est sur ce lien. Nous n'avions **pas le nœud émetteur du message**. Pour palier ce problème, nous avons **remplacé la variable *EnTransitVers* par *MessagesLiens***, au lieu d'avoir un message mappé à son nœud destination, nous avons **un message mappé au lien** sur lequel il se trouve.

Diffusion de messages



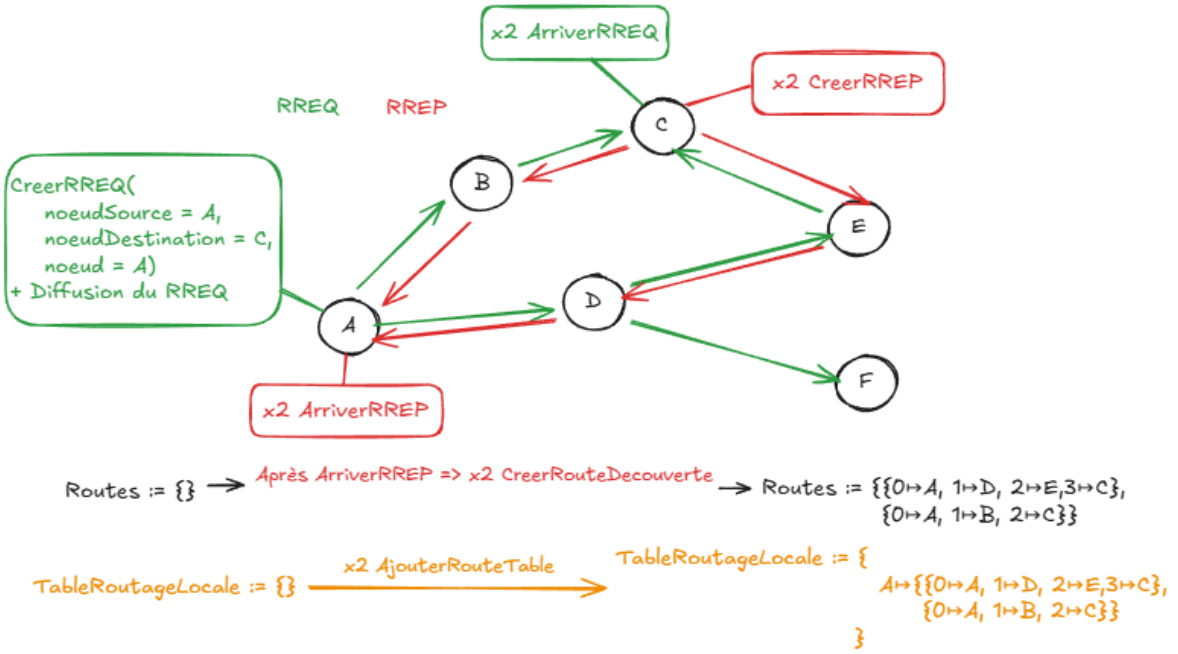
Comme vous avez pu le voir dans la partie raffinement, ce mécanisme de diffusion est réalisé en **3 événements** ce qui nous a valu une **grande réflexion et une restructuration de la totalité de nos raffinements**. Pour ce qui est des ajouts de variables et d'événements, nous étions forcés d'appliquer ces modifications au DSR_0 ce qui a chamboulé un peu tout le projet et cassé un grand nombre de preuves.

Le plus dur était de trouver un moyen de **créer plusieurs messages à la fois** qui ont chacun une destination différente avant de les diffuser, c'est ce qui a mené à la création de l'ensemble *DestDiffusion* et des événements *PreDuplicationMessage* et *DupliquerMessage*.

Cela a permis de **stocker le message d'origine avec *PreDuplicationMessage*** et **d'ajouter des duplications avec *DupliquerMessage*,** puis **la diffusion nettoie l'ensemble et envoie les messages**.

Notre système fonctionne bien, il n'y a pas de conflit si on fait plusieurs diffusion en même temps, il est compatible avec les ajouts et suppressions de liens à tout moment. Cependant, la diffusion est faite en 3 événements ce qui est assez fastidieux.

Découverte de route



Comme expliqué dans la partie raffinement, nous avons deux façons de découvrir des routes :

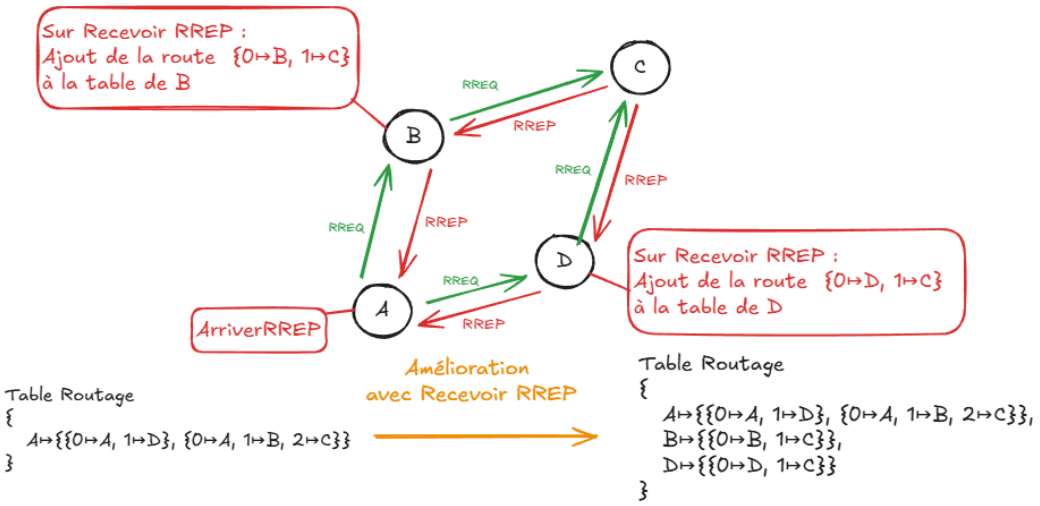
- En **envoyant des RREQ** : On crée manuellement la route que l'on souhaite en envoyant le RREQ où l'on souhaite et en envoyant le RREP. **Le RREP n'aura qu'une route possible la route inverse**.

- En **diffusant des RREQ** : On découvre **toutes les routes possibles entre deux nœuds**. A la réception de tous les RREP correspondants à la route inverse de tous les RREQ.

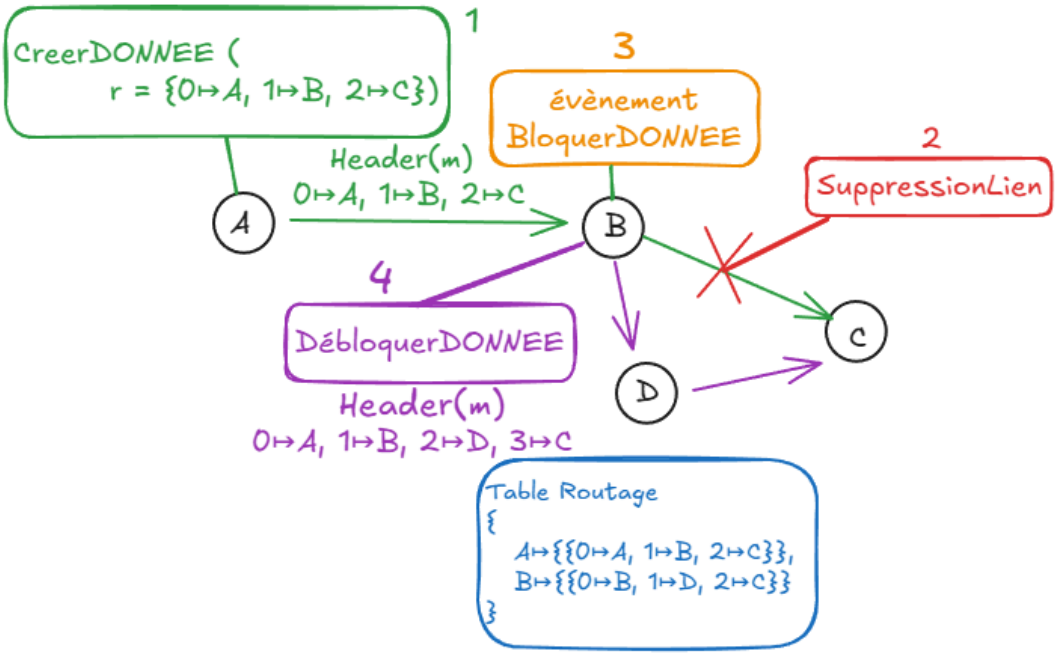
Une fois un RREP arrivé il est possible de créer avec *CreerRouteDecouverte* une route qu'on ajoute dans *Routes* et de l'ajouter à la table de routage locale avec *AjouterRouteTable*.

1. Améliorations

Une des améliorations possible de notre système de découverte de route aurait été de modifier l'événement *RecevoirRREPouRERRP* que l'on a fait identique pour les deux systèmes de réponses. C'est à dire d'avoir un événement *RecevoirRREP* qui **ajoute à la table de routage locale de chaque nœud** par lequel la réponse est passée, **une route qui est une partie de la route finale** (voir schéma ci-dessous).



Blocage et déblocage de données



Dans le cas où une donnée est bloquée à cause de la suppression d'un lien après la découverte des routes (tables de routage locales non à jour). Lors de l'envoi d'une donnée qui a une route obsolète dans son header, **la donnée va se retrouver bloquée** sur un nœud (ici en B). Nous avons pensé à créer un événement *BloquerDONNEE* pour que l'utilisateur comprenne bien que l'une de ses données est bloquée.

1. Packet Salvaging

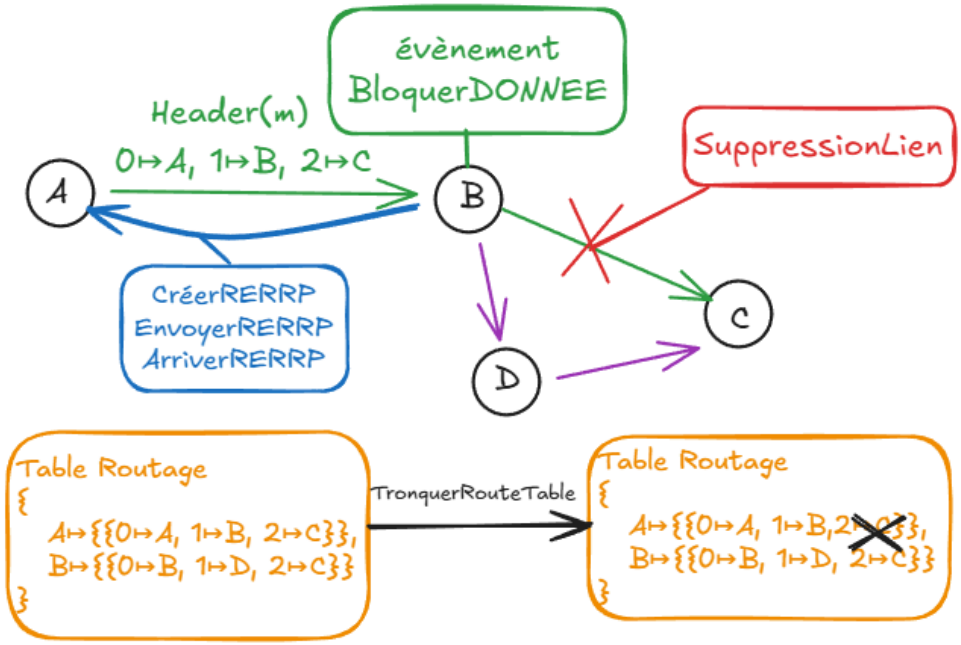
Dans notre système, il est possible de récupérer la donnée bloquée avec l'événement *DebloquerDONNEE*, inspiré de la section 3.4.1 du RFC. Cet événement consiste à **modifier l'en-tête d'un message** grâce à une route présente dans la table de routage locale du nœud dans lequel la donnée est bloquée pour **trouver une autre route par laquelle passer pour arriver à destination** (voir événement 4 du schéma). Si il n'y a pas de route, la donnée restera bloquée jusqu'au moment où une nouvelle route est découverte en partant du nœud où elle est bloquée vers sa destination.

2. Améliorations

Nous avons hésité à ne pas créer l'événement *BloquerDONNEE* pour éviter d'ajouter un événement supplémentaire mais nous avons trouvé plus intuitif pour l'utilisateur de le faire de cette manière. On aurait pu **utiliser un seul événement *CreerRERRP*** et ajouter la donnée bloquée dans *MessagesBloques* lors de la création d'un RERRP.

Maintenance de routes

1. Maintenance par troncage



En reprenant l'exemple précédent, lorsqu'une donnée est bloquée, il est possible de créer un message **RERRP** qui a pour destination le nœud source et lui associe le lien rompu dans l'ensemble *LienCasse*. Une fois que le message d'erreur arrive sur ce nœud source, il est possible de **tronquer toutes les routes de sa table de routage locale ayant le lien cassé** avec *TronquerRouteTable*.

2. Maintenance par suppression

Un autre fonctionnement, distinct du troncage de routes, est la suppression entière de route. L'événement *SupprimerRouteTable* est possible lorsqu'un nœud détecte que le lien vers le premier saut d'une route dans sa table n'existe plus, rendant effectivement la route obsolète dans son entièreté. Cela supprime donc localement ladite route dans le nœud détecteur. Cependant, aucun RERRP n'est créé en conséquence.

3. Améliorations

Notre système de maintenance pourrait être amélioré car la détection d'un lien cassé survient seulement lorsqu'une donnée est bloquée ou lorsque le lien cassé est le premier saut d'une route (voisin direct). Peut-être que créer **une diffusion du RERRP** serait une solution pour améliorer notre système et **mettre à jour toutes les tables de routage du système**.

Une autre amélioration possible mais complexe de la maintenance de route serait d'implémenter les acquittés de réception (ACK) décrits dans la section 3.2 du RFC où tout nœud est **capable d'entendre et confirmer la diffusion de ses voisins**.

Tableaux de preuves

Les preuves ont été la partie la plus complexe et chronophage du projet. Avec certaines modifications de conceptions qui ont cassé les preuves de nos trois raffinements réunis. La plupart des preuves se font automatiquement mais quelques-unes ont dû être résolues à la main.

Modèle	Obligations générées	Prouvées automatiquement	Prouvée manuellement
DSR_0	114	98	16
DSR_1	40	38	2
DSR_2	26	19	7
DSR_3	133	93	40

▸DSR_0

Element Name	Total	Auto	Man.
Proof Obligations	114	98	16
INITIALISATION	14	14	0
AjoutNoeud	6	6	0
AjoutLien	3	2	1

SuppressionLien	6	6	0
AjoutMessage	13	11	2
SuppressionMessage	13	13	0
Envoyer	9	8	1
PreDuplicationMessage	4	3	1
DupliquerMessage	19	15	4
DiffuserMessage	15	9	6
Recevoir	5	4	1
Perdre	4	4	0
Invariants	93	77	16

▸DSR_1

Element Name	Total	Auto	Man.
Proof Obligations	40	38	2
INITIALISATION	5	5	0
SuppressionLien	1	1	0
AjoutMessage	3	3	0
PreDuplicationMessage	1	1	0
DupliquerMessage	4	4	0
DiffuserMesage	6	5	1
SuppressionMessage	5	5	0
Envoyer	3	3	0
Recevoir	4	4	0
Perdre	1	1	0
Arriver	6	5	1
Invariants	36	34	2

▸DSR_2

Element Name	Total	Auto	Man.
Proof Obligations	26	19	7
INITIALISATION	4	4	0
AjoutNoeud	3	3	0
CreerRoute	4	3	1
AjouterRouteTable	4	3	1
SupprimerRouteTable	2	2	0
TronquerRouteTable	8	4	4
Invariants	19	13	6

▸DSR_3

Element Name	Total	Auto	Man.
Proof Obligations	133	93	40
ArriverRREQ	7	6	1
RecevoirRREQ	8	7	1
EnvoyerRREQ	2	2	0
CreerRREP	12	3	9
EnvoyerRREPouRERRP	2	2	0
ArriverRREP	4	3	1
RecevoirRREPouRERRP	1	1	0
CreerRouteDecouverte	1	1	0
AjouterRouteTable	0	0	0
SupprimerRouteTable	1	0	1
TronquerRouteTable	2	1	1
CreerRERRP	16	10	6
ArriverRERRP	1	1	0
Invariants	75	46	29

Animation des modèles avec ProB

Vérification du modèle et des interblocages avec ProB

Nous avons modélisé notre DSR_3 avec des SET de taille 3, le temps d'exécution est déjà de presque 2 minutes :

Model Checking finished
No errors found, but not all possible states have been visited (Due to animation parameter restrictions).
==========

Coverage statistics:
Total Number of States:50668
Total Number of Transitions:211606
Node Statistics:
deadlocked:0
invariant_violated:0
invariant_not_checked:0
open:0
live:50668
explored_but_not_all_transitions_computed:50668
total:50668
Operations Statistics:
SETUP_CONSTANTS:1
INITIALISATION:1
AjoutNoeud:1189
CreerDONNEES:252
EnvoyerDONNEES:420
ArriverDONNEES:240
DupliquerRREQ:1482
DebloquerDONNEE:252
BloquerDONNEES:312
RecevoirDONNEES:1110
RecevoirRREPouRERRP:930
ArriverRREP:876
EnvoyerRREPouRERRP:1806
CreerRREQ:5735
CreerRREP:1728
CreerRouteDecouverte:2802
RecevoirRREQ:7014
Perdre:16180
ArriverRREQ:8252
PreDuplicationRREQ:8055
DiffuserRREQ:11745
EnvoyerRREQ:9099
SupprimerRouteTable:4362
AjoutLien:46804
SuppressionLien:48613
SuppressionMessage:27984
AjouterRouteTable:4362
Uncovered Operations:
TronquerRouteTable
CreerRERRP
ArriverRERRP

On peut observer dans ce cas avec des SET de taille 3, il n'y a pas assez de Messages pour créer un RERRP car les 3 premiers messages sont utilisés par le RREQ, RREP, et la DONNEE.

Pour le cas avec des SET de taille 4, en 40 min, seulement 10% du model-checking s'est réalisé, donc nous avons annulé ce checking.

Propriétés LTL

Par cause de problèmes de syntaxe, nous n'avons pas pu réussi à faire vérifier par ProB des propriétés LTL complexes qui lient des variables entre elles avec des quantificateurs existentiels/universels. Mais quelques propriétés simples sont passées.

Voici des propriétés LTL qui seraient pertinentes :

- Une fois qu'un message est arrivé, il doit le rester :
  `G({m : MessagesArrives} => G({m : MessagesArrives}))`

- Après avoir ajouté un lien, on peut supprimer un lien
  `G ([AjoutLien] => X e(SuppressionLien))`
  Nous avons réussi à vérifier une propriété simple comme celle-là.

- L'existence d'une donnée amène à l'existence d'un message arrivé, bloqué, ou perdu (aurait été bien plus utile si on avait réussi à lier le premier m avec le second) :
  `G({#m.(m : Messages & TypeMessage(m) = DONNEES)} => F({#m.(m : MessagesArrives or m : MessagesPerdus or m : MessagesBloques)}))`

- Une découverte de route aura un jour une réponse ou sera perdue :
  `G({m1 : Messages & TypeMessage(m1) = RREQ} => F({m2 : Messages & TypeMessage(m2) = RREP & m1 |-> m2 : LienRREQetRREP or m1 : MessagesPerdus}))`

- Un événement d'arrivée fait croître l'ensemble des messages arrivés :
  `G(([ArriverDONNEES] or [ArriverRREQ] or [ArriverRREP] or [ArriverRERRP]) => increasing({MessagesArrives}))`
  Cette propriétée LTL fonctionne.

Bilan technique et perspectives

Bilan technique

Sur le plan technique, le projet nous a permis de construire progressivement une version simplifiée mais cohérente du protocole DSR. Le découpage en trois raffinements nous a beaucoup aidés : chaque étape ajoutait une nouvelle fonctionnalité sans que tout s'écroule.

Un point important a été la gestion des liens dynamiques. Comme un lien peut disparaître à tout moment, il a fallu revoir certaines variables utilisées au début (comme EnTransitVers) et repenser la façon dont on représente un message en transit. MessagesLiens s'est révélée plus adaptée.

Le mécanisme de diffusion a été l'un des points les plus compliqués. Le fait de devoir tout faire en trois événements (pré-duplication, duplication, diffusion) était lourd, mais c'était la seule manière qu'on a trouvée de gérer correctement plusieurs duplications en parallèle tout en gardant un comportement clair. Cela a cassé énormément de preuves au passage, mais au final le système fonctionne proprement.

La découverte de route (RREQ/RREP) et la maintenance (RERRP) ont été les parties les plus intéressantes. On a réussi à obtenir un fonctionnement assez proche de ce qui est décrit dans le RFC : un RREQ traverse le réseau, un RREP revient en sens inverse, et on peut construire une route au nœud source à partir de ça. La maintenance avec les messages d'erreur fonctionne également, même si elle reste assez basique : seule la table du nœud source est mise à jour.

Enfin, la partie Packet Salvaging nous a permis de gérer le cas où une donnée est bloquée. Si le nœud connaît une autre route, il peut la réutiliser sans relancer une découverte. Sinon, il faut lancer une nouvelle recherche. Là aussi, la logique reste simple mais correcte.

Perspectives

Plusieurs améliorations seraient possibles si on avait eu plus de temps :

1. **Une maintenance des routes plus complète :** Actuellement, seul le nœud source met à jour sa table lorsqu'il reçoit un RERRP. Une amélioration serait de diffuser ce message pour que tous les nœuds concernés mettent leur table à jour automatiquement.
2. **Mieux exploiter les RREP :** On aurait pu faire en sorte que chaque nœud intermédiaire enrichisse aussi sa table de routage quand un RREP passe par lui. Ça éviterait de devoir redécouvrir des routes plus tard.
3. **Ajouter les acquittements (ACK) :** DSR utilise des ACK pour détecter très tôt qu'un lien est cassé. Cela permettrait de détecter les liens cassés avant d'envoyer une donnée, d'éviter certaines données bloquées.
4. **Faire fonctionner les propriétés temporelles :** On aurait pu utiliser des contraintes LTL pour garantir des propriétés sur notre modèle.
5. **Simplifier certaines structures :** Plusieurs variables pourraient être nettoyées :

- LienCasse pourrait stocker uniquement le hop cassé
- un seul RERRP par donnée bloquée suffirait
- fusionner certains événements trop proches

Bilans personnels et individuels

Anton

Implémenter DSR a invité à réfléchir sur beaucoup d'aspects concrets d'un protocole de routage, nous avons dû vraiment comprendre les attendus de son fonctionnement. J'ai aimé le cadre de développement formel proposé par Event-B et Rodin, malgré le manque d'ergonomie du logiciel. Nous avons majoritairement travaillé ensemble sur le développement, mais j'ai moins participé à la rédaction du rapport. Dans l'ensemble, je suis très satisfait de la modélisation que nous avons pu réaliser, notre modèle fonctionne correctement et toutes les obligations de preuves sont prouvées.

Gabriel

J'ai trouvé le sujet intéressant et ludique, j'ai beaucoup aimé réfléchir à la conception du DSR mais la plateforme Rodin est assez lourde avec beaucoup de latences. Les heures perdues à essayer de faire une preuve sont assez frustrantes. Pour ce qui est du temps de travail, je pense avoir travaillé 15 heures sur la conception, 30 heures sur les preuves et 8 heures sur le rapport.