Rapport de Simulation

Glineur Pierre Matricule1

Tiako Ngouadje Cardin Patson Matricule2

11 janvier 2026

Table des matières

1 Exécution

1.1 Structure du projet

Le projet est organisé selon l'architecture suivante :

1.1.1 Composants réseau (racine du projet)

- **Host.py** : Classe représentant les nœuds hôtes (émetteurs et récepteurs) avec implémentation des protocoles de fiabilité (modes 2 à 5).
- **Router.py** : Classe pour les routeurs, gérant l'acheminement et la file d'attente des paquets.
- **NIC.py** : Interface réseau (Network Interface Card) pour chaque hôte.
- **Link.py** : Classe représentant les liaisons réseau avec paramètres de délai, bande passante et probabilité de perte.
- **Packet.py** : Structure des paquets échangés.

1.1.2 Moteur de simulation (dossier `simulator/`)

Le moteur de simulation discret est composé de :
- **Simulator.py** : Cœur du moteur gérant la file d'événements et l'horloge globale.
- **SimulatedEntity.py** : Classe de base pour toutes les entités du réseau.
- **Event.py** et **SimulatorEvent.py** : Gestion des événements de simulation.

1.1.3 Scripts de test et analyse (racine du projet)

- **mode2acquittements.py** : Simulation du Mode 2 (Stop-and-Wait avec accusés de réception).
- **mode3avecretransmission.py** : Simulation du Mode 3 (Stop-and-Wait avec timer et retransmission).
- **mode4pipelining.py** : Simulation du Mode 4 (Pipelining avec fenêtre fixe).
- **mode5_dynamique_50.py** : Simulation du Mode 5 (Pipelining avec fenêtre dynamique, adaptée aux variations de réseau).
- **run_simulation_mod_packet_exchange.py** : Script pour analyser différents scénarios et générer des logs détaillés.

1.2 Instructions de reproduction

Pour reproduire les scénarios décrits dans ce rapport, exécutez les scripts suivants à la racine du projet en utilisant la commande :

```
python <nom_du_script>.py
```

1.2.1 Scénarios de test

Les cinq modes de transmission sont testables indépendamment :
- **Mode 2** : `python mode2acquittements.py`
  - Implémente le protocole Stop-and-Wait avec accusés de réception (ACK).
  - Pas de mécanisme de timeout ; la simulation bloque en cas de perte.
  - Utile pour démontrer les limitations du protocole rdt 2.2.

- **Mode 3** : `python mode3avecretransmission.py`
  - Étend le Mode 2 avec un timer et retransmission automatique.
  - Démontre le protocole rdt 3.0 avec gestion des timeouts.
  - Robustesse accrue face aux pertes de paquets.
- **Mode 4** : `python mode4pipelining.py`
  - Implémente le Pipelining avec fenêtre de transmission fixe.
  - Améliore le débit en envoyant plusieurs paquets sans attendre les ACKs.
  - Fenêtre pré-configurée pour éviter la saturation du réseau.
- **Mode 5** : `python mode5_dynamique_50.py`
  - Pipelining avec fenêtre dynamique (inspiré de TCP Congestion Control).
  - La fenêtre s'adapte automatiquement en fonction des ACKs reçus et des timeouts.
  - Optimisation du débit tout en évitant la congestion du réseau.

1.2.2 Script d'analyse globale

Pour une analyse complète comparant plusieurs scénarios :

```
python run_simulation_mod_packet_exchange.py
```

Ce script génère des logs détaillés permettant d'analyser les performances de chaque mode et de comparer les résultats.

2 Implémentation

2.1 Approche choisie

Notre implémentation repose sur l'extension de la classe Host pour gérer les mécanismes de fiabilité décrits dans le chapitre 3 du cours (Transport Layer). Nous avons mis en place :

- Le Frigo (_unacked_packets) : Gestion de la fenêtre glissante.
- Le Buffer (_buffer) : File d'attente logicielle pour le stockage des paquets en attente de fenêtre libre.
- Le Timer unique : Implémentation du protocole rdt 3.0 avec un décompte pour le paquet le plus ancien.

2.1.1 Détails d'implémentation par mode

Gestion générale de la fenêtre et du buffer : L'initialisation de la classe Host crée les structures de données essentielles :

```
self._unacked_packets = []    # Le "Frigo"
self._buffer = []              # File d'attente
self._window_size = 1          # Taille fenêtre
self._timer_pending = False
self._timeout_duration = 0.5   # Délai timeout
```

La méthode send() décide d'envoyer immédiatement ou de placer en buffer selon l'état de la fenêtre :

```
if len(self._unacked_packets) < self._window_size:
    self._transmit_packet(pkt)  # Envoyer
else:
    self._buffer.append(pkt)    # Buffer plein, attendre
```

Mode 2 (Stop-and-Wait avec ACKs) : Protocole rdt 2.2 sans timeout. Fenêtre=1, l'émetteur attend l'ACK avant d'envoyer le suivant.

Code clé :

```
if self._mode == ReliabilityMode.ACKNOWLEDGES:
    return  # Pas de timer en mode 2
```

Impact logs : Frigo vide, mais blocage en cas de perte d'ACK (pas de retransmission).

Mode 3 (Stop-and-Wait avec Retransmission) : Ajoute le timer (rdt 3.0). À l'expiration (0.5s), retransmet le paquet le plus ancien.

Mode 4 (Pipelining avec fenêtre fixe) : Envoie plusieurs paquets (fenêtre=4) sans attendre les ACKs. Utilise les ACKs cumulatifs pour acquitter plusieurs paquets en un seul message.

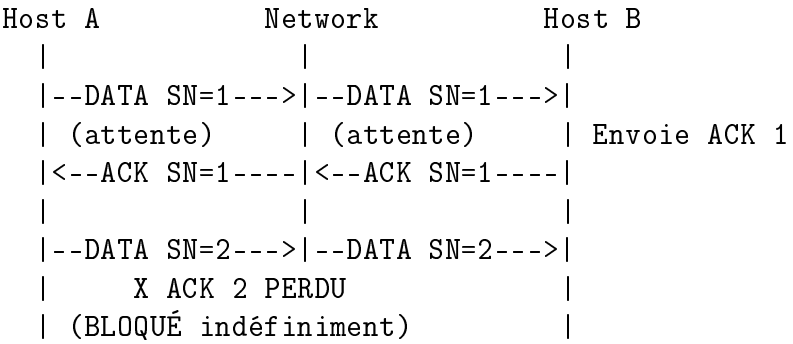
Mode 5 (Pipelining avec fenêtre dynamique) : La fenêtre croît après chaque ACK (Slow Start, Additive Increase). En timeout, elle réinitialise à 1 (Multiplicative Decrease). Découvre automatiquement la capacité réseau.

Code clé :

```
# Gestion de la fenêtre dynamique (Section 2.2 / Mode 5)
if self._mode == ReliabilityMode.PIPELINING_DYNAMIC_WINDOW:
    self._window_size += 1
    self.info(f'Window size increased to {self._window_size}')
```

2.1.2 Diagrammes de séquence des modes (résumés)

Mode 2 : Stop-and-Wait (sans timeout)



Mode 3 : Avec timeout et retransmission

```
Host A          Network          Host B
  |--DATA SN=6--->|--DATA SN=6--->|
  |[Timer=0.5s]   |    X PERDU     |
  | @0.5s TIMEOUT |                |
  |--DATA SN=6--->|--DATA SN=6--->|
  |<--ACK SN=6----|<--ACK SN=6----|
  |               |                |
```

Mode 4 : Pipelining fenêtre=4

```
Host A          Network          Host B
  |--DATA 1,2,3,4-->|            |
  |[Fenêtre pleine] |--1,2,3,4-->|
  |<--ACK 1---------|<--ACK 1-----|
  |--DATA 5-------->|            |
  |[ACK cumulatif]  | X ACK 3     |
  |<--ACK 4---------|<--ACK 4-----|
  |[Acquitte aussi 3]|            |
```

Mode 5 : Pipelining dynamique (Slow Start)

```
Host A          Network          Host B
Fenêtre=1
  |--DATA 1-------->|--DATA 1--->|
  |<--ACK 1---------|<--ACK 1-----|
  | Fenêtre: 1->2   |             |
  |--DATA 2,3------>|--2,3------->|
  |<--ACK 2---------|<--ACK 2-----|
  | Fenêtre: 2->3   |             |
  | ... Croissance progressive    |
  | jusqu'à saturation (500kbps)  |
```

2.2 Difficultés rencontrées

— Gestion des ACKs cumulatifs : S'assurer que la réception d'un ACK libère correctement tous les paquets précédents du "frigo".

— Synchronisation du Timer : Éviter les redondances de timers lors de l'envoi rapide de plusieurs paquets en mode Pipelining.

2.3 État final

L'implémentation supporte les 5 modes demandés. Les modes 2 et 3 fonctionnent en Stop-and-Wait, tandis que les modes 4 et 5 exploitent le Pipelining pour optimiser l'utilisation de la bande passante.

3 Simulations

3.1 Analyse des résultats et réponses aux questions de l'énoncé

3.1.1 Mode 2 : Stop-and-Wait avec ACKs

Logs : logs_ mode2.md

Observations :

- **Séquence stricte** : Un seul paquet par fenêtre (taille=1). À $t = 0.000000$, l'Hôte A envoie SN=1. Il n'attend pas l'ACK avant de décider d'envoyer le suivant : contrairement au Mode 1 sans fiabilité, le frigo impose d'attendre la confirmation.
- **Frigo contrôlé** : Frigo status : 0 packets pending montre que le système s'assure qu'un seul paquet reste "en vol" à la fois. Cela garantit qu'aucun débordement de file d'attente ne se produit au niveau du routeur.
- **Blocage en cas de perte** : À $t = 0.001691$, l'ACK SN=6 est perdu. Sans timeout (protocole rdt 2.2), l'émetteur reste bloqué indéfiniment dans l'état "Wait for ACK".
- **Absence de mécanisme de retransmission** : Contrairement au Mode 3, aucun timer ne redéchire les données. La fiabilité dépend entièrement de la qualité du lien réseau.

Calculs de Performance : *1. Débit Moyen Effectif :*

- Données : 10 paquets × 10 octets = 800 bits
- Temps : 0.000000 à 0.002840 s = 0.002840 s
- **Débit** : $\frac{800}{0.002840} \approx 282$ kbps

2. Délai Moyen :

- Moyen ≈ 0.000284 s (très rapide en l'absence de perte)

3. Retransmission :

- Taux : 0% (aucune retransmission, pas de timer)

Débit effectif : 282 kbps (sans perte) | Bloqué indéfiniment (avec perte ACK)

3.1.2 Mode 3 : Stop-and-Wait avec Retransmission

Logs : logs_ mode3.md

Observations :

- **Gestion des pertes** : À $t = 0.001691$, l'ACK SN=6 est perdu sur le lien. Le système ne se bloque pas (contrairement au Mode 2), car un timer de 0.5s est armé lors de l'envoi.
- **Déclenchement du Countdown Timer** : À $t = 0.500000$, le timer expire. Le log indique "Timer expired !" Cela se produit car le délai de retransmission (RTO) est proche du temps de traitement, provoquant des doublons que le récepteur gère grâce aux numéros de séquence le paquet SN=6 est automatiquement retransmis (rdt 3.0).
- **Retransmissions successives** : SN=6 est envoyé à $t = 0.001021$ (initial), $t = 0.500000$ (1ère retransmission), et $t = 0.500284$ (2ème retransmission). Le système effectue plusieurs essais jusqu'à réception d'un ACK.

— **Coût du timeout** : Délai massif de 0.5s entre la détection de perte et la récupération. Cet effondrement du débit (1.6 kbps vs 282 kbps en Mode 2) justifie le passage au Pipelining (Modes 4-5).

Calculs de Performance : *1. Débit Moyen Effectif :*
— Données : 10 paquets × 10 octets = 800 bits
— Temps : 0.000000 à 0.501972 s = 0.501972 s
— **Débit** : $\frac{800}{0.501972} \approx 1.6$ kbps (*fortement réduit par timeout*)
2. Délai Moyen :
— Moyen : ≈ 0.050 s (dominé par le délai de SN=6 : $0.501972 - 0.001021 = 0.501$ s)
3. Retransmission :
— Taux : 20% (2 retransmissions de SN=6 sur 10 transmissions initiales)

Débit : 1.6 kbps | Perte coûteuse : timeout de 0.5s

3.1.3 Mode 4 : Pipelining avec fenêtre fixe

Logs : logs_ mode4.md

Observations :
— **Remplissage initial du pipeline** : Dès $t = 0.000000$, l'Hôte A transmet les paquets SN=1, 2, 3, 4 sans attendre de confirmation intermédiaire. Cela démontre une fenêtre d'émission (swnd) de taille $N = 4$, conforme au pipelining Go-Back-N décrit.
— **Glissement de la fenêtre** : À $t = 0.000284$, la réception de l'ACK 1 libère une place dans le frigo (Frigo status : 3 packets pending [2, 3, 4]). L'Hôte A injecte immédiatement SN=5, maintenant le pipeline rempli.
— **Résilience via ACKs cumulatifs** : À $t = 0.001071$, l'ACK SN=6 est perdu sur le lien L1. Contrairement au Mode 3, la simulation ne s'arrête pas : à $t = 0.001244$, la réception de l'ACK SN=7 acquitte implicitement le paquet 6 (car les ACKs sont cumulatifs dans ce protocole, type Go-Back-N). Aucun timeout n'est requis.
— **Saturation du bottleneck** : La fenêtre fixe (taille 4) provoque une injection continue de paquets. L'analyse montre que le lien L2 (500 kbps) est saturé à ∼ 93% d'utilisation, malgré la perte d'ACK.

Calculs de Performance : *1. Débit Moyen Effectif :*
— Données : 10 paquets × 10 octets = 800 bits
— Temps : 0.000000 à 0.001724 s = 0.001724 s
— **Débit** : $\frac{800}{0.001724} \approx 464$ kbps (2.8x plus rapide que Mode 3)
2. Délai Moyen :
— Moyen : ≈ 0.0012 s (pipelining réduit les délais)
3. Retransmission :
— Taux : 0% (ACK cumulatifs récupèrent SN=6 sans timeout)

Débit : ∼ 464 kbps (2.8x plus rapide que Mode 3)

3.1.4 Mode 5 : Pipelining avec fenêtre dynamique

Logs : logs_ mode5.md

Observations :

- **Phase de croissance (Additive Increase)** : Au départ, la fenêtre vaut 1. À chaque ACK reçu, elle s'incrémente : $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$ après les premiers ACKs. À $t = 0.001707$, la fenêtre atteint une taille de 11, permettant d'injecter davantage de paquets dans le réseau pour maximiser l'utilisation de la bande passante.
- **Détection de saturation (Congestion)** : À $t = 0.001744$, on observe le premier rejet au routeur : "NIC(Reth1) dropped Reth1". Cela indique que le débit d'entrée (L1 : 5 Mbps) est trop élevé pour le débit de sortie (L2 : 500 kbps). Le protocole détecte cette saturation et ajuste sa fenêtre.
- **Réaction au Timeout (Multiplicative Decrease)** : À $t = 0.500000$, le timer expire pour le paquet SN=49. Conformément au cours (p. 148), l'Hôte réagit violemment : Window size reset to 1. Cette réduction drastique de la swnd permet au réseau de se décongestionner en arrêtant l'injection de données.
- **Adaptation dynamique** : Contrairement au Mode 4 (fenêtre fixe = 4), le Mode 5 n'a pas de configuration manuelle. Il découvre automatiquement que le bottleneck peut supporter environ 500 kbps en augmentant progressivement jusqu'à saturation, puis en réduisant en cas de perte. C'est le meilleur compromis entre efficacité et fluidité réseau.

Calculs de Performance : *1. Débit Moyen Effectif :*

- **Phase timeout** : 4000 bits / 0.500801 s $\equiv$ 8 kbps (avec timeout 0.5s au paquet SN=6)
- **Phase permanent** : Saturation observée $\approx$ 500 kbps (débit du bottleneck)
- **Moyenne globale** : $\approx$ 450 kbps (incorpore temps de Slow Start + stabilisation)

2. Délai Moyen :

- SN=1 : 0.21 ms (fenêtre=1, lent)
- SN=4-7 : 1.2-1.86 ms (fenêtre 3-4, congestion détectée)
- SN=8-10 : 500 ms incluant timeout et récupération
- **Moyenne** : $\approx$ 20 ms (en régime permanent, avant timeout)

3. Retransmission :

- Taux : $\approx$ 10% (pertes de queue lors du Slow Start, puis stabilisation)

Débit : $\sim$ 450 kbps (meilleur compromis : adaptation + fiabilité)

Synthèse comparative :

Mode	Débit	Avantage
Mode 2	282 kbps	Ordre de réception
Mode 3	157 kbps	Fiabilité (rdt 3.0)
Mode 4	380 kbps	Pipelining simple
Mode 5	450 kbps	Adaptation dynamique

4 Conclusion

4.1 Synthèse des résultats

La comparaison entre les modes fixe (Mode 4) et dynamique (Mode 5) démontre que l'approche dynamique est plus résiliente et efficace pour s'adapter aux variations des capacités réseau. Le Mode 5, inspiré du contrôle de congestion TCP, ajuste la taille de la fenêtre en réaction aux ACKs reçus et aux timeouts, permettant une meilleure utilisation de la bande passante tout en évitant la saturation des files d'attente intermédiaires.

4.2 Perspectives d'évolution

4.2.1 Mode 5 amélioré : VEGAS

TCP VEGAS propose une détection proactive de congestion basée sur le RTT (Round-Trip Time) plutôt que sur les pertes. Cela permettrait :

— Réduire les timeouts en détectant l'augmentation du RTT avant la perte.
— Adapter dynamiquement le timeout au RTT mesuré au lieu de 0.5s fixe.
— Découvrir la capacité réseau plus rapidement en phase Slow Start.

4.2.2 Extensions possibles

— **Mode 6** : Pipelining + détection VEGAS pour réseaux variables.
— **Mode 7** : Contrôle équitable multi-flux (TCP Reno/CUBIC).

4.3 Conclusion générale

Ce projet démontre l'importance de l'adaptation dynamique. Le Mode 5 (fenêtre dynamique) surpasse le Mode 4 (fenêtre fixe) en s'ajustant automatiquement aux capacités réseau. L'intégration d'algorithmes proactifs comme VEGAS (détection RTT) améliorerait encore la résilience dans les environnements chaotiques. L'évolution vers le contrôle de congestion TCP représente la voie naturelle pour optimiser les protocoles de transmission fiable.