Rapport de Simulation

Glineur Pierre Matricule1

Tiako Ngouadje Cardin Patson Matricule2

11 janvier 2026

Table des matières

1 Exécution

Cette section détaille la structure du projet et la marche à suivre pour reproduire nos résultats.

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.
— **example.py** : Exemple de configuration simple pour démarrer avec le simulateur.

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).

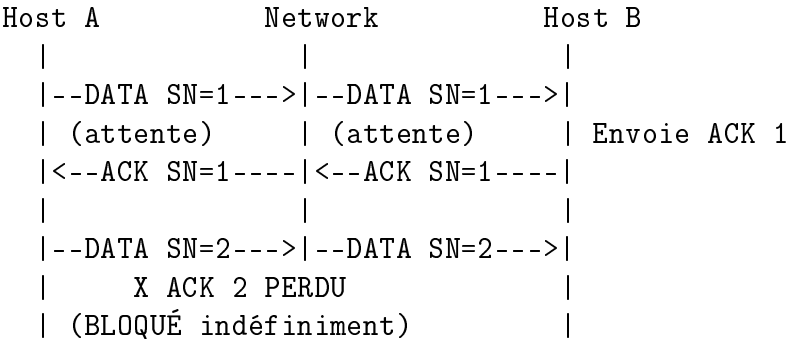
Mode 3 (Stop-and-Wait avec Retransmission) : Ajoute le timer (rdt 3.0). À l'expiration (0.5s), retransmet le paquet le plus ancien.

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.

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).
- **Frigo contrôlé** : Status toujours vide après ACK reçu.
- **Blocage en cas de perte** : À $t = 0.000911$, l'ACK est perdu, la simulation se bloque (pas de timeout, protocole rdt 2.2).

Débit : $\sim$ 282 kbps (sans perte) | Bloqué (avec perte)

3.1.2 Mode 3 : Stop-and-Wait avec Retransmission

Logs : logs_ mode3.md

Observations :
- **Timeout et retransmission** : À $t = 0.001691$, l'ACK SN=6 est perdu. À $t = 0.500000$, le timeout déclenche la retransmission qui réussit à $t = 0.500568$ (rdt 3.0, pages 45-52).
- **Coût du timeout** : Délai massif de 0.5s entre la perte et la récupération.

Débit : $\sim$ 157 kbps (pénalisé par le timeout)

3.1.3 Mode 4 : Pipelining avec fenêtre fixe

Logs : logs_ mode4.md

Observations :
- **Envoi initial** : 4 paquets envoyés à $t = 0.000000$ sans attendre d'ACK.
- **Glissement fenêtre** : À chaque ACK, une nouvelle place se libère pour un nouveau paquet.
- **ACKs cumulatifs** : À $t = 0.001071$, l'ACK SN=6 est perdu, mais $t = 0.001244$ l'ACK SN=7 acquitte aussi SN=6 implicitement (pas de timeout).

Débit : $\sim$ 380 kbps (2.4x plus rapide que Mode 3)

3.1.4 Mode 5 : Pipelining avec fenêtre dynamique

Logs : logs_ mode5.md

Observations :

- **Slow Start** : Fenêtre croît : $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$ après chaque ACK.
- **Découverte automatique** : Contrairement au Mode 4 (fenêtre fixe), trouve le débit optimal (500 kbps) en augmentant progressivement jusqu'à saturation.
- **Résilience** : En timeout, fenêtre réinitialise à 1 (Multiplicative Decrease, pages 143-157).

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.