

# Rapport de Simulation

Glineur Pierre      Tiako Ngouadje Cardin Patson  
Matricule1            Matricule2

11 janvier 2026

## Table des matières

<b>1 Exécution</b>	<b>2</b>
1.1 Structure du projet . . . . .	2
1.1.1 Composants réseau (racine du projet) . . . . .	2
1.1.2 Moteur de simulation (dossier <code>simulator/</code> ) . . . . .	2
1.1.3 Scripts de test et analyse (racine du projet) . . . . .	2
1.2 Instructions de reproduction . . . . .	2
1.2.1 Scénarios de test . . . . .	3
1.2.2 Script d'analyse globale . . . . .	3
<b>2 Implémentation</b>	<b>3</b>
2.1 Approche choisie . . . . .	3
2.1.1 Détails d'implémentation par mode . . . . .	3
2.2 Difficultés rencontrées . . . . .	5
2.3 État final . . . . .	5
<b>3 Simulations</b>	<b>5</b>
3.1 Analyse des résultats et réponses aux questions de l'énoncé . . . . .	5
3.1.1 Stop and Wait (Modes 2 & 3) . . . . .	5
3.1.2 Pipelining avec fenêtre fixe (Mode 4) . . . . .	5
3.1.3 Pipelining avec fenêtre dynamique (Mode 5) . . . . .	6
<b>4 Conclusion</b>	<b>6</b>

# 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).
- **mode3avec transmission.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
```

**Modes 2 & 3 (Stop-and-Wait)** : Les deux modes utilisent une fenêtre fixe de taille 1. L'émetteur envoie un paquet et doit attendre l'ACK avant de poursuivre. La différence fondamentale réside dans le timer :

**Mode 2 (ACK simple)** : Pas de timer, le code retourne sans action :

```
def _start_timer(self):
    if self._mode == ReliabilityMode.ACNOWLEDGES:
        return # En mode 2, pas de timer
    # ... suite du code
```

Si un paquet est perdu, la simulation attend indéfiniment l'ACK.

**Mode 3 (Retransmission)** : Le timer est armé lors de chaque envoi. À l'expiration, la méthode `_on_timeout()` retransmet :

```
def _on_timeout(self):
    if self._unacked_packets:
        oldest_pkt = self._unacked_packets[0]
        self.info(f'Timer expired!')
        self._nic.send(oldest_pkt)  # Retransmettre
        self._timer_pending = False
        self._start_timer() # Réarmer le timer
```

**Modes 4 & 5 (Pipelining)** : L'émetteur envoie plusieurs paquets dans une fenêtre de transmission avant d'attendre les ACKs.

**Mode 4 (Fenêtre fixe)** : La fenêtre a une taille pré-définie et constante (ex. 4 paquets). Si la fenêtre est pleine, les nouveaux paquets sont stockés dans le buffer.

**Mode 5 (Fenêtre dynamique)** : La fenêtre débute à 1 et augmente d'un paquet à chaque ACK reçu, mimant le comportement TCP. En cas de timeout, elle réinitialise à 1 :

```
if self._mode == ReliabilityMode.PIPELINING_DYNAMIC_WINDOW:
    self._window_size += 1 # Augmente à chaque ACK

# En cas de timeout :
if self._mode == ReliabilityMode.PIPELINING_DYNAMIC_WINDOW:
    self._window_size = 1 # Retour à 1 (congestion)
```

**Gestion des ACKs cumulatifs :** Lors de la réception d'un ACK, la méthode `receive()` supprime du frigo tous les paquets ayant un numéro de séquence inférieur ou égal (comportement cumulatif) :

```
# Supprimer le paquet et tous les précédents (cumulatif)
self._unacked_packets = [p for p in self._unacked_packets
                          if p.serial_number > pkt.serial_number]
```

Cela permet d'acquitter implicitement plusieurs paquets en un seul ACK.

**Synchronisation du buffer et de la fenêtre :** À la réception d'un ACK, une boucle extrait du buffer les paquets en attente, tant que l'espace dans la fenêtre est disponible :

```
while self._buffer and len(self._unacked_packets) < self._window_size:
    next_pkt = self._buffer.pop(0)
    self._transmit_packet(next_pkt) # Reprendre transmission
```

Cela maintient un flot continu de transmissions sans surcharger le frigo.

## 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 Stop and Wait (Modes 2 & 3)

- Mode 2 (ACK) : Sans timer, nous observons que la simulation se bloque indéfiniment en cas de perte (Lien L1 avec `lost_prob = 0`). Cela illustre les limites du protocole rdt 2.2 décrit aux pages 34-40 du cours.
- Mode 3 (Retransmission) : Le log montre @0.500000 Timer expired! Retransmitting. L'émetteur détecte la perte et survit à l'incident.
- Temps moyen de transmission : [Insérez votre calcul ici : Moyenne des temps entre `sends` et `received ACK`].

### 3.1.2 Pipelining avec fenêtre fixe (Mode 4)

- Observations : Contrairement au mode 3, l'émetteur envoie plusieurs paquets (fenêtre de 4) dès  $t=0.000000$  sans attendre les premiers ACKs.
- Comparaison : Le temps moyen de transmission diminue drastiquement car le lien ne reste pas inactif durant le temps de propagation (RTT).

### 3.1.3 Pipelining avec fenêtre dynamique (Mode 5)

- Comportement attendu : La fenêtre commence à 1, augmente à chaque ACK, et chute à 1 en cas de timeout.
- Analyse du Goulot : Avec un lien L2 à 500 kbps et une file au routeur de 10, nous observons une congestion.
- Débit moyen effectif : il faudra Insérer le calcul :  $(\text{Paquets} * \text{Taille} * 8) / \text{Temps total}$ .
- Justification : L'ajustement dynamique permet de trouver le débit maximal du lien critique (500 kbps) sans saturer indéfiniment le routeur.

## 4 Conclusion

La comparaison entre les modes fixe et dynamique montre que le mode dynamique (inspiré de TCP) est plus résilient et efficace pour s'adapter aux capacités variables des nœuds intermédiaires (routeurs).