Ecole Publique d'Ingénieurs en 3 ans

Rapport

# SIMULATION D'EPIDEMIE

**Tony PINSEL-LAMPECINADO, Fabien LE BEC, Hamza EL KILALI,**

Système d'exploitation
**Alain LEBRET**


www.ensicaen.fr

# TABLE DES MATIERES



# TABLE DES FIGURES

# 1. Introduction

Ce projet a pour but de réaliser une application en langage C qui simule la propagation d'une contamination d'origine virale dans une ville, ainsi que la lutte menée contre cette contamination par certains de ses citoyens. Il a été réalisé en trinôme. Le projet met en œuvre les différents mécanismes vus en cours de systèmes d'exploitation tels que la communication interprocessus (signaux, tubes, files de messages, mémoire partagée), les threads ainsi que les sémaphores. Le contexte d'utilisation de ces mécanismes est décrit par la suite dans ce rapport.

# 2. Initialisation de la simulation

## 2.1. Structure utilisée

Nous avons décidé de placer notre stucture de données appelée `City` dans une mémoire partagée. La structure contient un tableau de cases à 2 dimensions représentant le terrain, un tableau de 37 citoyens, le nombre de tours effectués, un tableau de news ainsi que 3 systèmes de flags dont les rôles seront expliqués par la suite.

## 2.2. Initialisation de la ville

La phase d'initialisation de la ville sert à créer et à remplir cette mémoire partagée. Elle est effectuée par l'appel à la fonction `init_city`. Dans un premier temps, on initialise la mémoire partagée de manière classique. Après cela, les cases du terrain sont initialisées. Cette étape doit prendre en compte les positions obligatoires des casernes et de l'hôpital ainsi que le positionnement aléatoire des 12 maisons et des 3 terrains vagues contaminés en début de simulation. Enfin, il y a création des 37 citoyens avec leurs spécificités `init_citizen` qui permet de spécifier en argument le type de citoyen qui doit être créé.

## 2.3. Choix de conception

La phase d'initialisation nous a demandé un certain temps de réflexion sur les structures que nous allions utiliser. Par exemple, nous ne savions pas comment faire pour que les citoyens connaissent leur position sur le terrain et que dans le même temps les cases de la ville connaissent les citoyens présents sur elles-mêmes. Cela aurait pu être implémenté à l'aide de pointeur, cependant l'utilisation d'une mémoire partagée nous empêchait d'utiliser les pointeurs. C'est pourquoi nous avons décidé de placer un tableau de 37 entiers dans chaque case et de numéroter les citoyens. Ainsi si le citoyen numéro i est présent sur une case alors le i-ème entier du tableau prend la valeur 1 et sinon elle vaut 0.

## 2.4. Fin de la phase d'initalisation

La fin de cette phase d'initialisation met en place plusieurs mécanismes.

### 2.4.1. Lancement des threads citoyens

Tout d'abord, il faut lancer un thread par citoyen. Pour cela, on fait appel à la fonction `citizen_manager`. Ce programme lance les différents threads qui resteront ouvert jusqu'à la fin de la simulation, c'est-à-dire le 100ème tour. Dans le cas d'un journaliste, une file de messages, commune avec l'autre journaliste et l'agence presse, est ouverte.

### 2.4.2. Ouverture de l'agence de presse

Lors de son lancement, dans un thread, l'agence de presse ouvre en lecture la file de messages créée par les journalistes. Elle ouvre aussi un accès sur la mémoire partagée pour stocker les dépêches que doit afficher l'interface graphique.

## 3. Interface graphique


*Figure 1 : illustration de l'interface graphique*

L'interface graphique a été réalisée à l'aide de la bibliothèque *CDK* (*Curses Development Kit*) qui utilise des fonctions de la bibliothèque *ncurses*. L'interface graphique est faite de 4 composants : l'affichage du jour, l'affichage des dépêches, les différents histogrammes et enfin les différentes maps.

### 3.1. Affichage du jour et des dépêches

L'affichage du jour et des dépêches s'est fait assez facilement à l'aide du composant *CDKLabel* qui permet de créer facilement un label contenant du texte avec des contours. Il fallait cependant faire attention pour le label contenant les dépêches d'avoir assez de place en largeur et en hauteur pour afficher les dépêches dès l'initilisation car la taille de l'encadrer se base sur le texte présent à l'initialisation du composant et ne peut plus se mettre à jour par la suite.

### 3.2. Histogrammes

Concernant les différents histogrammes affichant les statistiques sur l'état de santé des citoyens, le composant *CDKHistogram* permet la création d'un histogramme. La difficulté ici était la gestion des couleurs qui se base sur un système de pair avec une couleur de premier plan et une couleur d'arrière-plan. De par l'utilisation du terminal comme interface graphique, ce système reste très primaire et ne dispose que de 8 couleurs (64 paires). De ce fait, lorsque la barre de l'histogramme dépassait la valeur écrite en blanche, cette dernière devenait presque illisible. La solution était d'inverser les couleurs de la pair lorsque la barre de l'histogramme atteignait la moitié. Des énumérations ont été utilisées pour définir le type d'un histogramme et ainsi n'avoir qu'une seule fonction pour la création et une seule pour la mise à jour, offrant ainsi plus de modularité.

### 3.3. Maps

La création des trois maps a été la partie la plus complexe dans l'élaboration de l'interface graphique car la bibliothèque *CDK* ne dispose pas de composant permettant de représenter une grille. Il fallait donc utiliser le composant le plus basique, un *CDKLabel*, où chaque ligne du label avait une fonction particulière : les trois premières définissant l'en-tête de la map (titre, légendes) et les suivantes les différentes cases de la map. Cela a nécessité de nombreuses heures de travail pour en arriver à un résultat satisfaisant. Le composant map est modulable, la fonction de création prend en paramètres le type de map, le titre, la légende, la largeur d'une case, etc. La fonction de mise à jour de la map ne met à jour que les lignes contenant des cases, pas l'en-tête. Tout ici est représenté sous forme d'une chaîne de caractères, notamment les codes couleurs, il fallait donc bien prendre en compte tous ces caractères de « mise en forme » lors du calcul de l'espace mémoire nécessaire pour une ligne mais aussi lors de la manipulation des lignes, la manipulation des chaînes de caractères n'étant pas la chose la plus triviale en C.

### 3.4. Alignement des composants

La bibliothèque *CDK* ne fourni qu'une gestion très basique de l'alignement d'un composant : à gauche, au centre et à droite. Pour améliorer ce système, de nombreuses macros ont été définies ainsi qu'un système de padding pour faciliter les calculs lors de l'alignement des composants entre eux.

## 4. Déroulement d'un tour

Le déroulement d'un tour est synthétisé par le schéma de la Figure 2. Un système de flag a été utilisé pour prévenir les différents acteurs d'un nouveau tour. Les signaux ont été utilisés pour le timer et pour fermer proprement le programme lors de l'appuie sur la touche `CTRL-C`. Enfin, des sémaphore ont été mis en place lors de l'écriture d'un acteur dans la mémoire partagée pour garantir un principe d'exclusion mutuelle.

## 4.1. Mise en place du timer

Le timer a pour rôle de périodiser la durée des tours. En effet, la fonction `timer` envoie des signaux au programme `epidemic_sim` pour lui demander d'exécuter un nouveau tour. Une fois que le signal est reçu, les flags de citoyen et de la presse sont paramétrés à 0, ce qui enclenchera une mise en route des threads citoyens et de l'agence de presse.

## 4.2. Threads des citoyens

Lorsque que le flag d'un citoyen vaut 0, le thread effectue un nouveau tour. Comme décrit dans l'énoncé le citoyen a un certain pourcentage de chance de se déplacer. Après quoi, il effectue les actions spécifiques à son tour comme par exemple soigner d'autres citoyens présents sur sa case si il est médecin ou encore brûler les cadavres si il est pompier. De plus, une fois qu'il a effectué ces étapes, il faut mettre à jour les données du citoyen, celle de la ville ainsi que les différents systèmes de contamination.

Afin d'éviter que 2 threads utilisent en même temps une même ressource de la mémoire partagé nous avons décidé d'utiliser des sémaphores.

Un de nos regrets concernant cette partie du projet est la profusion de lignes de code et de fonctions écrites pour gérer les actions des citoyens. Cependant nous n'avions pas d'idée de conception pour gérer plus facilement tout ces évènements qui auraient sans doute été plus compréhensible dans un langage orienté objet tel que Java.

## 4.3. Agence de presse

À chaque tour, l'agence de presse récupère toutes les dépêches de la file de messages qui est l'unique canal de communication avec les journalistes. La dépêche de priorité 10 la plus récente est écrite dans la mémoire partagée pour être ensuite affichée par l'interface graphique. La même chose est faite pour les dépêches de priorité inférieure. Les minorations sur les différents chiffres des dépêches sont appliquées par les journalistes.

## 4.4. Sauvegarde des données du tour

A la fin de chaque tour, la fonction `manage_end_of_the_round` est appelée. C'est elle qui inscrit les données dans le fichier « evolution.txt » qui sera utilisé pour effectuer la modélisation sur Gnuplot à l'aide du fichier de commande. Le fichier est ouvert en écriture lors de l'initialisation de la simulation et il sera fermé juste avant l'ouverture de Gnuplot.

## 4.5. Gestion de l'arrêt du programme

La gestion de l'arrêt du programme de manière propre, aussi bien pendant qu'après la simulation, nous a semblé essentielle. De nombreux mécanismes tels que la mémoire partagée ou encore la file de messages peuvent avoir un comportement inattendu s'ils ne sont pas bien fermés. Néanmoins, la partie la plus sensible à la fermeture est l'interface graphique car elle utilise le terminal, une fermeture brutale alterne le comportement du terminal. Pour répondre

à ce problème, le mécanisme des signaux a été utilisé avec notamment l'utilisation de *SIGINT* et *SIGQUIT*.

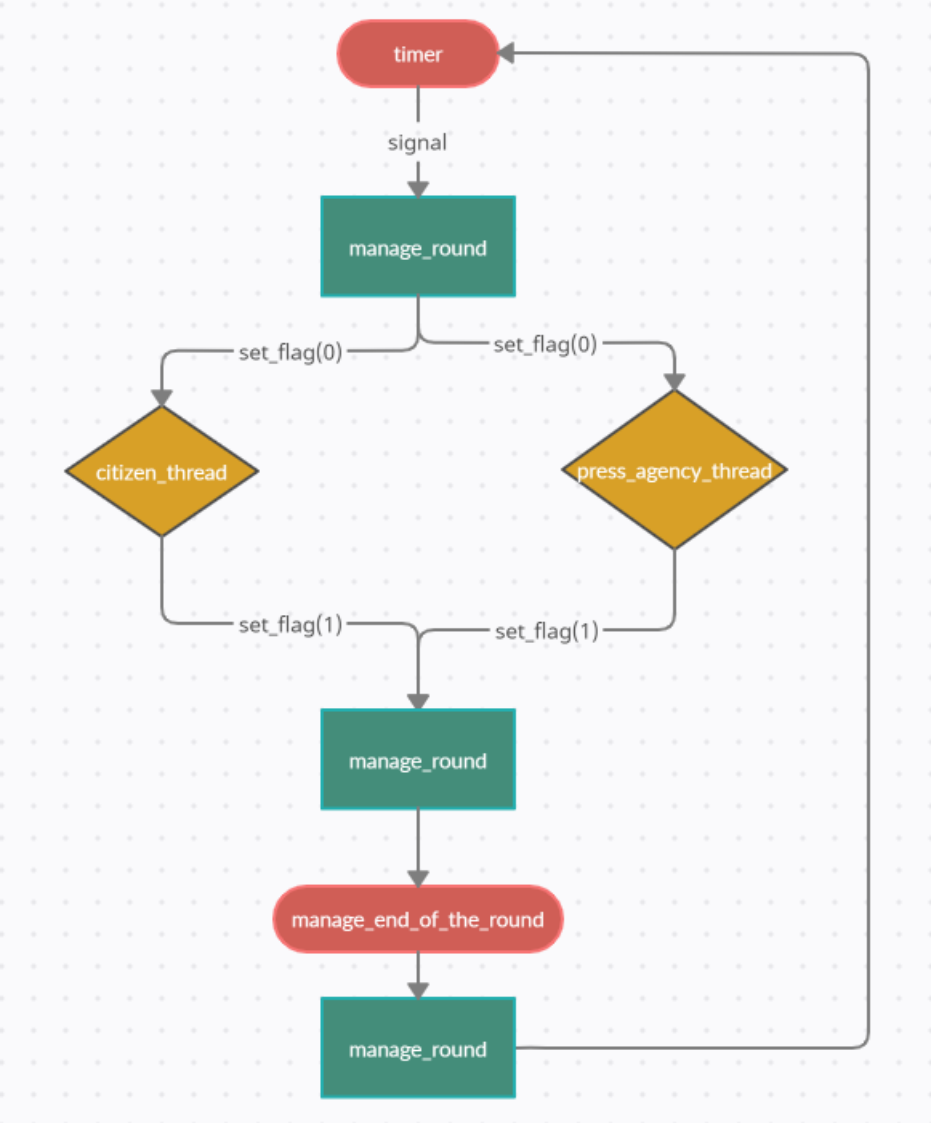


*Figure 2 : schéma du déroulement d'un tour*

# 5. Résultats de la simulation

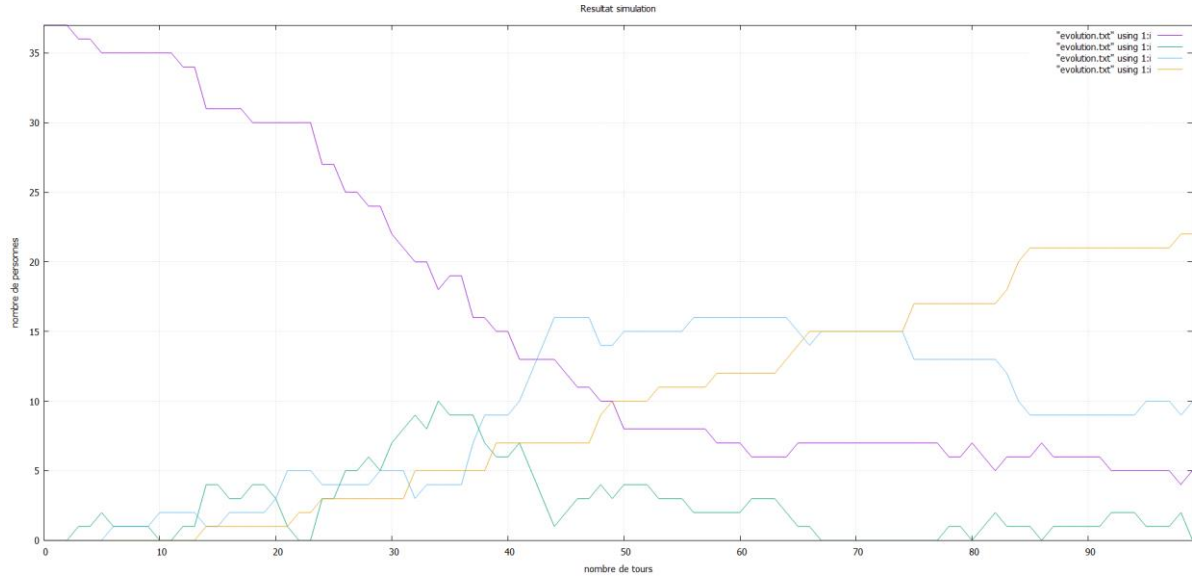


*Figure 3 : graphique obtenu après simulation*

Bien que fonctionnel, notre programme ne permet pas d'obtenir les résultats attendus. Nous n'avons pas pu implémenter toutes les conditions du sujet à cause d'un problème de temps. En effet, ce projet a été réalisé en parallèle de trois autres projets. Nous avons donc été contraints, après près d'une trentaine d'heures passé sur ce projet, de ne pas implémenter les dernières conditions (elles concernent notamment les contaminations entre les cases).

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Ecole Publique d'Ingénieurs en 3 ans

6 boulevard Maréchal Juin, CS 45053
14050 CAEN cedex 04

ministère éducation nationale enseignement supérieur recherche

Normandie Université

Cti Commission des titres d'ingénieur

cdefi Conférence des Directeurs des Écoles Françaises d'Ingénieurs

European Accreditation of Engineering Programmes EUR-ACE®

CONFÉRENCE DES GRANDES ÉCOLES