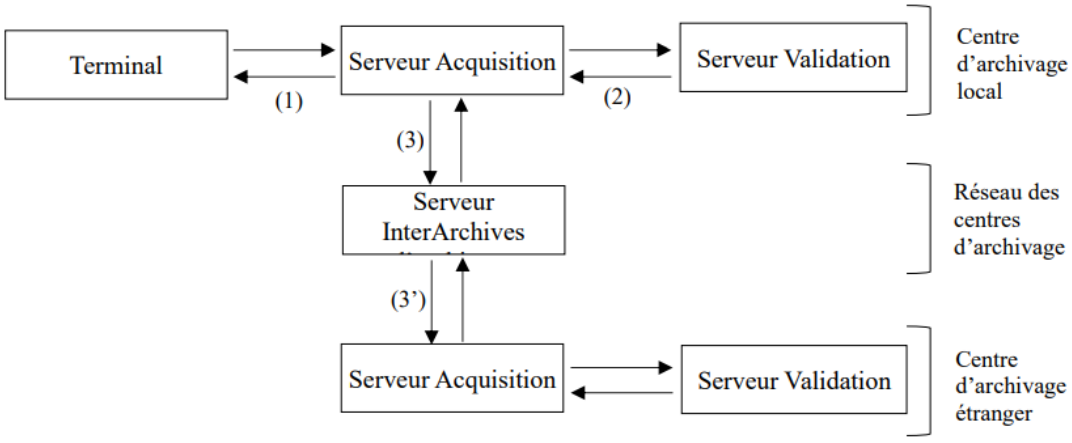
Rapport Projet EL-3032 – 2021

Projet de programmation système en C



Sommaire

Introduction

Ce projet consiste à créer une application qui simule un réseau de centres de gestion de test PCR dans des aéroports. Pour ce faire nous nous baserons sur le schéma fourni (figure 1) dans le sujet du projet, en effet celui décrit très bien les cahiers des charges et l'architecture que nous avons choisis pour l'application.

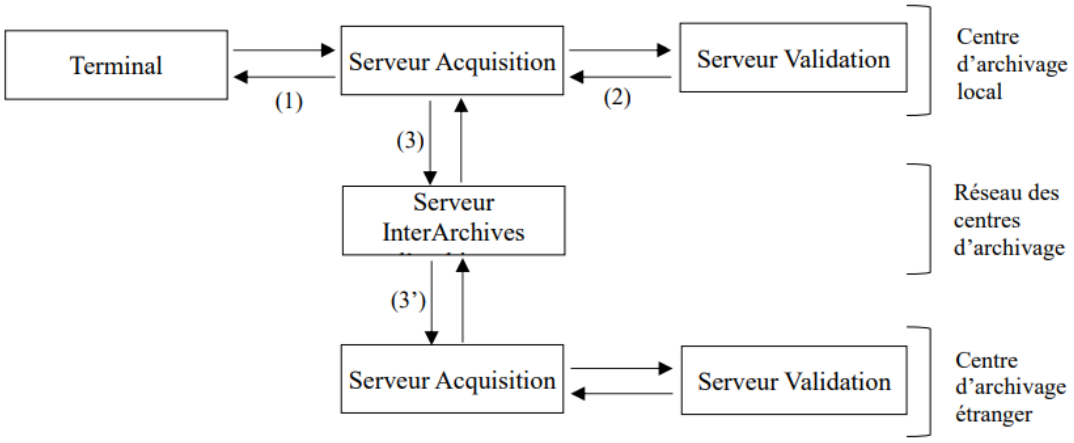


Figure 1

Vous trouverez le mode d'emploi du logiciel dans le readme.md de notre projet GitHub.
lebonq/projet_testPCR: Projet test_pcr, simulation d'un réseau d'archivage de test PCR en C. (github.com)
Ce mode d'emploi permet d'utiliser et de tester les programmes facilement.
Nous avons décidé de découper notre programme en 4 composants distincts. Le terminal, le serveur de validation, le serveur d'acquisition et pour finir le serveur inter-archive.
Nous les décrirons tous chacun leur tour dans ce rapport.

Terminal

Rôle du composant

Le Terminal permet de communiquer avec un centre d'archivage local auquel il envoie une demande de validation sous la forme d'un message qui contient : le numéro du test, la nature du message (demande, réponse) et sa valeur (soit la durée de validité du test ou alors son résultat)
C'est aussi le Terminal qui permettra l'affichage des résultats de la requête

Architecture

L'architecture du terminal traite les demandes séquentiellement. Nous n'avons aucun intérêt à paralléliser les demandes du terminal. Dans le cas d'un aéroport, l'opérateur qui utilisera le terminal s'occupera des voyageurs 1 par 1. Il devra donc attendre la réponse avant de passer au voyageur suivant.
Pour échanger nos messages et communiquer avec le reste du centre nous utilisons la bibliothèque fournit pour écrire et lire les messages échangés entre les différents composants de notre système.
Cela permet de garder une cohérence entre les différents composants de notre système.

Pour cela notre composant suit 3 étapes principales :

- L'entrée utilisateur
- L'envoie de la demande de validation du test
- La réception de la réponse

Entrée utilisateur

Pour traiter les demandes nous avons besoin de recueillir les informations, le numéro du test, entré par l'opérateur.
Pour cela nous lisons dans le buffer de l'entrée standard. C'est une solution simple mais efficace.

Envoyer demande

Une fois le numéro du test récupéré nous l'envoyons directement au serveur d'acquisition. Pour cela nous utilisons des tubes, qui sont des buffers de données qui permettent de communiquer entre différents processus donc entre nos composants. Une fois écrit dans ce tube nous attendons la réponse du serveur d'acquisition.

Réception réponse

Une fois la réponse reçue, nous affichons un message "*Le test (n') est (pas) valide*" sur l'écran, donc la sortie standard. De plus, nous avons ajouté une fonctionnalité qui permet de stocker dans un fichier d'historique toutes les réponses reçues par le terminal. Cela permet de garder une trace de l'activité sur le terminal.

Comment le tester ?

Pour tester ce composant il faut l'exécuter avec la commande :
./Terminal 0 1 nomCentreArchivage

Lorsqu'on tape le code **0001187519303827**, il faut ensuite écrire soit même le message de réponse. Ici cette réponse serait **|0001187519303827|Reponse|1|** .

Ce terminal est très flexible car il est utilisable sur un fichier, un tube ou l'entrée standard. Cependant nous utilisons le buffer de l'entrée standard pour récupérer les codes ce qui n'est pas très flexible et demande beaucoup de ressources pour être nettoyé à chaque fois.
Cependant il répond parfaitement au cahier des charges car il est 100% fonctionnel avec une fonctionnalité supplémentaire qui est l'historique.

Validation

Rôle du composant

Le composant validation permet de dire si un test est valide ou non, c'est à dire que le test n'est pas périmé dans le temps et qu'il est négatif alors il est valide. Sinon il n'est pas valide.
Le tout en cherchant dans un fichier qui liste les résultats.

Création base de données

Pour créer un fichier de résultats nous avons mis en place un programme qui permet de le générer automatiquement.
Nous avons dû modifier la bibliothèque fournis car l'aléatoire dépendait du temps en secondes, or notre fichier est généré en moins d'une seconde, nos valeurs aléatoires étaient toutes les mêmes. Nous avons modifié le programme **alea.c** de la bibliothèque pour que l'aléatoire dépende des millisecondes.
Une fois ce problème résolu nous générons un numéro de code aléatoire, une date de prélèvement entre 4 et 48 heures avant l'arrivée et si le test est négatif ou positif (8% de test positif). Le tout en pouvant spécifier l'id du centre pour lequel on veut générer ce fichier. (L'utilisation de ce programme est décrite dans le mode d'emploi).

Architecture

Tout comme le terminal nous avons dans un premier temps opté pour un traitement séquentiel des demandes.
On lit dans le buffer la demande reçue puis on parcourt le fichier de résultat pour trouver les données relatives au test, puis le valider. Une fois validé, nous écrivons dans le buffer de sortie qui est connecté au serveur d'acquisition.
L'intérêt d'utiliser des buffers est qu'ils sont compatibles avec n'importe quel processus et surtout nous permettent de des tester le processus validation tout seul sans les autres composants.

Comment le tester ?

Pour tester ce composant il faut l'exécuter avec la commande :
./Validation 0 1 nomfichier.txt
Le programme s'ouvre il faut donc rentrer une demande à la main par exemple si le test numéro **0001000000000000** est dans notre fichier de résultat on peut écrire : **|0001000000000000|Demande|250000|**
La réponse sera affichée sur l'écran, elle doit être en accord avec les données de notre fichier résultats.

Ce composant est tout comme Terminal, très flexible grâce à son utilisation des buffers. L'architecture séquentielle aussi évite tout problème de chevauchement d'écriture dans les buffers et donc de perte de données. Cependant, cette architecture pose quelques soucis, elle est lente par rapport à la solution parallèle. Créer un thread, c'est-à-dire un fil d'exécution, par demande reçue permettra de traiter les demandes presque simultanément. Il faudrait aussi implémenter un sémaphore qui permettrait de sécuriser l'écriture dans le buffer pour éviter que 2 threads n'écrivent en même temps.

Acquisition

Rôle du composant

Le composant acquisition est l'élément central de chaque centre d'archivage. Il est unique et il a pour rôle de gérer le flux de données entre tous les composants du réseau. Il permet de connecter tous les composants terminaux et validation entre eux grâce à des paires de buffers (tuyaux).
Il traite les demandes envoyées par les terminaux, il traite les réponses envoyées par le serveur de validation et route les demandes ou les réponses envoyées par le serveur inter-archive.

Architecture

Cette fois ci nous avons d'abord opté pour une architecture séquentielle peu fonctionnelle et pas très adaptée à l'implémentation du serveur InterArchives.
C'est pourquoi nous avons très vite implémenté une architecture parallèle.
Avant de rentrer dans le détail, il faut comprendre comment nous stockons les tests en attente de réponse.
Pour cela nous avons 3 listes : la liste des numéros, la liste des descripteurs associés au terminal ayant émis la demande et une liste de booléen. La liste de booléen permet de savoir quelles cases des listes doivent être conservées ou pas, c'est à dire si le terminal associé à la demande a reçu la réponse. Si la réponse est reçue alors on peut considérer cette case comme vide.
Notre architecture parallèle contient 3 fils d'exécution (threads) différents : le thread **terminal**, le thread **validation** et le thread **interArchives**.

Thread terminal

Ce thread qui est créé autant de fois que l'on veut créer de terminaux, va lire et enregistrer toutes requêtes envoyées par le terminal qui lui est associé.
Lors de la réception d'une demande envoyée par le terminal associé au thread le numéro de test est enregistré, le "nom" de son buffer de réponse aussi (c'est le buffer qui est lu par le terminal), le tout dans les listes nommées précédemment.
Une fois les données enregistrées la demande est transférée soit au serveur de validation soit au serveur inter archive en fonction de s'il s'agit d'une demande locale ou non.

Thread validation

Ce thread est unique pour chaque processus d'acquisition. Il lit directement dans le buffer où le serveur de validation envoie ses réponses. Lorsqu'il reçoit une réponse du serveur de validation, il cherche la case mémoire correspondant au test qu'il a reçu. Une fois cette case mémoire reçue, il nous suffit de récupérer le nom du buffer du terminal correspondant au test pour lui envoyer la réponse produite par validation.

Thread interArchives

Ce dernier thread est aussi unique dans le serveur d'acquisition.
Il a à peu près le même fonctionnement que le thread terminal. Mais il communique avec le processus InterArchives. Il peut recevoir soit des demandes soit des réponses.

Demande

Lorsqu'il reçoit une demande c'est qu'un centre d'archivage étranger souhaite faire valider un test associé au centre local. Pour cela nous utilisons la même méthode que pour un terminal excepté que le nom du buffer enregistré n'est pas celui du terminal mais celui du serveur interArchives. Comme cela lorsque le thread validation recevra la réponse, celle-ci sera transférée directement à interarchives.

Réponse

La réponse est issue d'une demande locale d'un test étranger. Ces réponses sont traitées exactement comme les réponses reçues dans le thread validation, elles sont transférées directement au terminal correspondant.

Sécurité

Étant donné l'utilisation de thread des données communes sont manipulés. Nous avons implémenté 2 sémaphores : un sémaphore **state** et **nbCaseLibre**.

State

Ce premier sémaphore permet de sécuriser l'écriture dans les buffers. En effet si 2 threads écrivent dans un buffer en "même temps" les données seront corrompus ce qui fera planter le programme. Cela nous permet de rentrer en section critique en étant sûr qu'aucun autre thread écrit en même temps qu'un autre.

nbCaseLibre

Ce sémaphore nous permet de savoir s'il y a encore des cases libres dans notre liste de stockage de tests. Cela permet d'éviter l'interblocage en section critique.

Comment le tester ?

Le programme peut être testé seul avec sa version séquentielle, tout se passe dans des fichiers textes.
./Acquisition tailleBufferListeTest fichierresulatcentre.txt
Les réponses seront écrites directement dans les fichiers du dossier.

Le principale point faible de ce composant est l'utilisation de buffer (tube) pour communiquer avec les autres composants du centre. En effet, s'il reçoit beaucoup de demande d'un coup les buffers de communication font se remplir jusqu'à déborder et nous perdrons des données. De plus nous retrouvons les buffers de communications des terminaux grâce aux numéros de test, or si nous recevons 2 demandes avec le même numéro de test au même moment (ce qui est normalement impossible). Nous pourrions ne pas retrouver le bon buffer de communication associé au test.

InterArchives

Rôle du composant

Le serveur interArchives va permettre l'interconnexion entre les différents serveurs d'acquisition, ce qui va permettre la réception et la transmission de réponses après s'être occupé du routage de celles-ci. Tout comme acquisition il permet de connecter les composants entre eux grâce aux buffers de communication (tubes).

Architecture

Pour ce composant nous avons utilisé une architecture parallèle. InterArchives communique avec les différents entre grâce à une paire de tuyaux (donc nos buffers vus précédemment). Ce composant sert à créer les serveurs acquisitions en fonction de notre fichier de configuration. Il reçoit des demandes de validation ou encore les réponses envoyées pour faire suite à une demande de validation.
Pour la configuration de tous les réseaux nous passons par un fichier de configuration qui contient toutes les informations citées précédemment pour lancer les composants.
Pour ce faire durant l'initialisation nous créons un système de liste comme dans acquisition qui fait correspondre l'ID du centre avec son buffer de réception. Mais aussi un système de liste exactement comme dans acquisition.
Le traitement de demande réponse est effectué dans un thread, il y aucun de threads que de serveur acquisition.
Le processus va tout d'abord commencer par lire le message reçu et va par la suite définir, si ledit message est une demande ou une réponse.

Demande

Lorsqu'une demande arrive nous la traitons comme dans le thread terminal d'acquisition a la seule exception près que nous ne l'envoyons pas dans le serveur de validation mais dans le buffer de réception du centre ou la demande doit être traité.

Réponse

Les réponses sont traitées comme dans le thread validation de composant acquisition, c'est-à- dire qu'elles sont renvoyées au même serveur d'où vient la demande.

Comme pour Acquisition, étant donné notre utilisation de thread de données communes. Ainsi pour raison de sécurité, l'utilisation des sémaphores précédemment présenté **State** et **nbCaseLibre** pourra de nouveau être observé. Ainsi il sera par exemple impossible d'écrire dans la mémoire pendant que celle-ci est lue.

Comment le tester ?

Il n’est pas possible de tester le composant InterArchives seul. Cependant on peut s’assurer de son bon fonctionnement en le testant avec tous les autres composants eux même testés au préalable.
./InterArchive fichierdeconfiguration.txt nbDeServeurAcquisition TailleBufferListe

La structure ce composant étant très similaire au composant acquisition nous retrouvons exactement les mêmes problèmes.

Conclusion et piste d’amélioration

Nous avons beaucoup aimé ce projet car il nous a permis de mettre en œuvre de manière concrète nos connaissances en informatique. L’utilisation de Git, la rédaction d’un readme (mode d’emploi), la mise en place d’un raisonnement pour choisir la meilleure architecture pour tel ou tel composant, l'écriture d’un makefile efficace, l’utilisation de sémaphore. Mais aussi le choix de structure de données et d'écriture de bibliothèque tier pour faciliter la création de fichier.
Tous ces points nous ont permis de grandement améliorer nos compétences techniques en C et développement d’application.

Les 3 point clés que nous voudrions améliorer avec plus de temps sont :

- Remplacer les tubes par des sockets
- Améliorer la lecture dans le buffer de l'entrée standard du terminal
- Mettre en place une architecture parallèle dans le composant validation

Ce projet a été entièrement réalisé grâce au travail personnel des membres de l’équipe.