Covert Channel - Transmission de données

Ce projet consiste en deux programmes, `send.c` et `recv.c`, qui réalisent la transmission de données entre deux processus à l'aide d'un canal dissimulé. L'en-tête `util.h` contient des constantes, à faire varier pour ajuster la précision du message reçu, et les prototypes des fonctions utilisées par les deux programmes.

Introduction

Dans le cadre de ce projet, nous avons développé deux programmes, `send.c` et `recv.c`, qui communiquent entre eux et transmettent des données à travers le cache du processeur.

L'objectif principal de ce travail est d'atteindre un ***taux d'erreur*** très faible dans la transmission des données, tout en maintenant une bonne ***vitesse*** d'execution des deux programmes.

Ce compte rendu présente les détails techniques de nos programmes, les ajustements effectués pour optimiser leur performance, ainsi que les paramètres clés qui ont été utilisés pour atteindre ces résultats impressionnants.

Contexte

Les canaux dissimulés sont des moyens de communication qui permettent de transmettre des informations en exploitant des ressources partagées, telles que le cache du processeur, de manière non intentionnelle et non détectable.

Ils sont souvent utilisés pour contourner la sécurité ou les mécanismes de protection des données, et peuvent servir à des fins malveillantes ou pour protéger la confidentialité des communications.

---

send.c

Le programme `send.c` est responsable de la transmission des données. Il lit le contenu d'un fichier texte, le convertit en binaire et envoie les bits un par un à l'aide d'un canal dissimulé.

Fonctions principales

- `message_to_binary` : Convertit un message texte en une représentation binaire.
- `read_file_content` : Lit le contenu d'un fichier texte et renvoie une chaîne de caractères.
- `main` : Lit le contenu du fichier texte, le convertit en binaire, puis envoie les bits via le canal dissimulé.
- `synchronisation` : Fonction de synchronisation pour aligner les opérations entre les programmes `send.c` et `recv.c`.

recv.c

Le programme `recv.c` est responsable de la réception des données. Il écoute le canal dissimulé, reçoit les bits un par un, les convertit en texte et affiche le message reçu, le taux d'erreur, la capacité réelle ainsi que le temps écoulé entre le début de la réception et sa fin.

Fonctions principales

- `read_file_content` : Lit le contenu d'un fichier texte et renvoie une chaîne de caractères (comme dans send.c).
- `message_to_binary` : Convertit un message texte en une représentation binaire (comme dans send.c).
- `binary_to_message` : Convertit une représentation binaire en un message texte.
- `calc_error` : Calcule le taux d'erreur en comparant les bits reçus avec les bits attendus.
- `write_to_file` : Ecrit les cycles enregistrés par la fonction `memaccesstime` dans un fichier *result.txt* afin de les visualiser via le programme python `visualization.py`.
- `calc_true_capacity` : Calcule la capacité réelle du programme via la méthode de communication par canaux cachés.
- `main` : Écoute le canal dissimulé, reçoit les bits, les convertit en texte, puis affiche le message reçu et le taux d'erreur.
- `synchronisation` : Fonction de synchronisation pour aligner les opérations entre les programmes `send.c` et `recv.c` (comme dans `send.c`).

Nous avons aussi besoin du message ici afin de pouvoir avoir la longueur du message qui est nécessaire au long du programme, mais aussi pour pouvoir comparer le message reçu avec l'original afin de relever le taux d'erreur dans la réception.

Voici ci dessous la fonction de calcul du taux d'erreur dans le programme `recv.c`, la méthode utilisé est identique à celle vu en cours, hormis que les valeurs ***br***, le nombre de bit reçu, et ***bs***, le nombre de bit envoyé, n'y apparaissent pas car ceux-là sont égaux içi.

```c
// Extrait de recv.c
double calc_error(const int *received_bits, const int *expected_bits, int
bit_count)
{
    int error_count = 0;
    for (int i = 0; i < bit_count; i++)
    {
        if (received_bits[i] != expected_bits[i])
        {
            error_count++;
        }
    }
    return ((double)error_count / bit_count) * 100;
}
```

---

util.h

L'en-tête `util.h` contient des constantes et des fonctions partagées par les deux programmes :

- `DELAYLOOP_VALUE` : Constante pour contrôler la durée d'attente entre chaque mesure.
- `DELAY_FOR_EACH_BIT` : Constante pour contrôler la durée entre les transmissions de bits.
- `CACHE_SYNC_INIT` : Constante pour contrôler la durée initiale de synchronisation.
- `CACHE_SYNC_LOOP` : Constante pour contrôler la durée de synchronisation pour chaque itération de la boucle.
- `synchronisation` : Prototype de la fonction de synchronisation pour aligner les opérations entre les programmes `send.c` et `recv.c`.

Les constantes `CACHE_SYNC_INIT` et `CACHE_SYNC_LOOP` représentent un bit sur 8 octets à considérer pour synchroniser l'initialisation et la boucle de transmission des données, respectivement. Ces valeurs sont utilisées pour déterminer le nombre de cycles d'horloge pendant lesquels les programmes attendent avant d'envoyer ou de recevoir un bit, assurant ainsi une synchronisation précise entre les deux parties.

Voici a quoi ressemble notre fichier d'en-tête `util.h` :

```
//DELAYLOOP_VALUE : durée d'attente entre les mesures
#define DELAYLOOP_VALUE 3500

//DELAY_FOR_EACH_BIT : durée entre les transmissions de bits
#define DELAY_FOR_EACH_BIT 600000

// CACHE_SYNC_INIT : durée initiale de synchronisation
#define CACHE_SYNC_INIT 36

// CACHE_SYNC_LOOP : durée de synchronisation pour chaque itération
#define CACHE_SYNC_LOOP 18
```

Résultats et améliorations

Voici un résultat avec comme message à transmettre, le poème de Victor Hugo de son recueil "*Les Contemplations*":

```
Message reçu: Demain, des l'aub%, a l'heure où blanchit la campagne,
Je partirai. Vois-tu, je sais que tu m'attends.
J'irai par la foret, j'irai par la montagne.
Je ne puis demeurer loin de toi plus longtemps.

Je marcherai les yeux fixes sur mes pensees,
Sans rien voir au dehors, sans entendre aucun bruit,
Seul, inconnu, le dos courbe, les mains croisees,
Triste, et le jour pour moi sera comme la nuit.



Taux d'erreur : 0.031726
Débit : 4057.061436 bits/s
```

```
Capacité réelle (T) : 4040.245589 bits/s
0.7769 secondes entre start et end.
```

Avec les paramètres actuels, nous obtenons des résultats satisfaisant pour la transmission de données à travers le canal dissimulé. Le taux d'erreur est très faible, atteignant moins de 0,1 % lorsque les programmes sont correctement synchronisés.

Cependant, il est toujours possible d'explorer différentes approches pour améliorer encore les performances des programmes. Voici quelques suggestions pour continuer à optimiser les performances et réduire le taux d'erreur :

1. Ajuster les paramètres dans `util.h` et expérimenter avec différentes valeurs pour trouver un équilibre optimal entre la vitesse de transmission et la fiabilité.
2. Examiner et optimiser les fonctions de synchronisation, en ajoutant des délais supplémentaires si nécessaire pour assurer une synchronisation précise entre les deux parties.
3. Exécuter les programmes avec la priorité la plus élevée possible pour minimiser les interférences avec d'autres processus et garantir une transmission de données fiable (avec la commande nice du shell).

En résumé, notre projet de transmission de données à travers un canal dissimulé a obtenu des résultats remarquables en termes de taux d'erreur et de vitesse d'exécution.

Les ajustements apportés aux paramètres et aux fonctions de synchronisation ont joué un rôle crucial dans l'amélioration des performances des programmes.

Il était essentiel de continuer à explorer différentes approches et d'ajuster les paramètres pour maintenir et améliorer encore les performances de ces programmes afin qu'ils puissent supporter des messages beaucoup plus long avec une précision et une vitesse d'exécution tout aussi impressionantes.

Défis et limitations

Durant le développement et les tests de nos programmes, nous avons rencontré plusieurs défis et limitations :

1. **Synchronisation** : Assurer une synchronisation précise entre les deux programmes a été un défi majeur. Nous avons dû ajuster les constantes de synchronisation pour trouver un équilibre entre la vitesse de transmission et la fiabilité. Nous avons également essayé différentes méthodes, comme la synchronisation par fichier externe.
2. **Interférences** : Les autres processus en cours d'exécution sur la machine peuvent interférer avec la transmission des données, ce qui peut entraîner des erreurs ou une mauvaise synchronisation. Pour minimiser ces interférences, nous avons exécuté les programmes avec la priorité la plus élevée possible. Cependant, ayant d'abord travaillé sur mon ordinateur personnel et des machines virtuelles Linux, il y avait beaucoup d'interférences que l'on retrouve beaucoup moins lorsque l'on utilise un véritable environnement Linux.

3. **Paramètres optimaux** : Trouver les paramètres optimaux pour un équilibre entre la vitesse de transmission et la fiabilité est un défi. Les tests et les ajustements sont nécessaires pour trouver les meilleures valeurs pour les constantes définies dans `util.h`.

Malgré ces défis, nous avons réussi à développer des programmes performants et à atteindre un taux d'erreur très faible dans la transmission des données.