UCLouvain

ÉCOLE POLYTECHNIQUE DE LOUVAIN

TRTP receiver capable de 1,5+ Mpps

LINGI1341

Groupe 138	d'Herbais de Thun Sébastien	28751600
	Heuschling Thomas	28871600

1 Abstract

Nous avons conçus une implémentation du receveur TRTP multithreadé capable de saturer une connexion 1 GbE et d'utiliser en grande partie une connexion 10 GbE. Dans nos tests, la vitesse de transfer maximale observée était de 700 MiB/s[(1)]. Et celle-ci n'était limitée non pas par notre implémentation mais notre capacité à générer les paquets assez rapidement. Ces performances sont possibles grâce à l'usage de *syscall avancé*, d'opération *atomiques* et l'usage de multiple *pipelines* traitant l'information en parallèle tout en restant *thread safe* et libre de toute fuite mémoire. De plus notre implémentation est accompagnée d'une suite étendue de tests couvrant la grande majorité du code et d'une documentation complète détaillant l'architecture and le fonctionnement des différents composants utilisés.

[(1)] cfr. section sur les performances

Contents

2 Introduction

The goal of this project was to implement a receiver based on the TRTP protocol capable of handling multiple clients at the same time. To try and have a more interesting and more in depth analysis we decided to realise an optionally multithreaded version. This means it can use multiple threads if desired or run on a single one. In the first mode we can reach with limited software/hardware tinkering speeds exceeding 2Gb/s and in single threaded mode speeds exceeding 1Gb/s easily saturating a typical 1GbE card. In addition, our implementation can treat hundreds of concurrent connections with an excellent success rate of over 99%[(1)]. This means our implementation fulfills the requirements and more by providing a high performance and reliable implementation.

In this document the architectural decisions and the techniques employed will be explained followed by in-depth performance analysis in different conditions including stress tests and link simulation. The end goal is to understand why a multithreaded implementation is interesting and why the specifics of our implementation give great performance with reasonable CPU usage. It will also try to explain the hard limit seen at 2.1Gb/s during testing.

[(1)] Note that all of the failures are due to sender unable to cope with the data rate

3 Architecture

Dés le début du projet, nous avions en tête de réaliser une implémentation multithreadé afin de maximiser les performances. Bien que l'architecture que nous avions originellement conçue était très naïve, elle s'est avérée être un bon point de départ pour ce qu'est notre architecture finale. Celle-ci est flexible fonctionnant autant en mode *séquentiel*, c'est à dire sur un unique thread, que sur beaucoup de thread séparés. De plus, le fonctionnement parallèle est amélioré par la présence de deux fichiers de configurations permettant d'optimiser d'avantage l'exécution sans devoir changer le code source.

L'architecture se base sur la possibilité en UDP d'écrire et de lire depuis un même socket depuis plusieurs thread en même temps. Ceci est possible car l'UDP n'est pas un protocole qui garantit l'ordre d'arrivée des paquets. Notre programme utilise donc une série de threads pour lire depuis le socket et une série de thread pour traiter les données et envoyer les paquets de (N)ACK. Ce premier type sera appellé *receiver* tandis que le deuxième sera appellé *handler*.

Chacun de ces threads a ensuite été amélioré par l'usage de *sycalls* avancés permettant de drastiquement diminuer non seulement le nombre de *syscalls* exécutés mais également le nombre de paquets de types ACK que le receveur doit émettre. Ceci sera expliquer dans plus de détails dans les sections qui suivent.

Ces threads communiquent entre eux à l'aide de deux queues de communication. La première sert à envoyer les paquets depuis le *receiver* au *handler* et la deuxième fait le chemin inverse. Bien que cela soit plus complexe, cela permet une meilleure utilisation de la mémoire allouée en réutilisant les *buffers* précédemment alloués. De plus ce *stream* utilise des opérations atomiques[1] lors de l'ajout d'éléments afin d'augmenter les performances et de diminuer la latence dans le *receiver*.

3.1 Table de hachage type *linear probing*

Le but de cette table de hachage ou *hash table* est de stocké la liste des clients concourants et de pouvoir rapidement identifié les nouvelles connexions. Elle stocke une structure *client_t* qui contient les données relatives à une connexion telle que l'adresse IP, le port et l'état de la fenêtre de réception. Cette table est partagée entre tous les *receivers* et est correctement protégée par un *mutex*.

La table de hachage donne un énorme avantage par rapport aux autres implémentations car elle est, en moyenne, en complexité temporelle $\mathcal{O}(1)$ sans requérir l'usage d'un *socket* par connexion qui pourrait causer problème pour un très grand nombre de connexion concourantes[2].

Afin de simplifier l'implémentation de celle-ci, notre table de hachage n'utilise par l'adresse IP du client mais uniquement le port de celui-ci comme clé. Cela rend le calcul de l'index très simple : $Port\%M$ où M est la capacité de la table de hachage. Ensuite, lors du *linear probing*, l'adresse IP est comparée en plus du port afin de garantir l'unicité de chaque client. De plus cette comparaison est expressément faite avec la fonction *memcmp* afin d'en augmenter les performances[3]. Ce qui veut donc dire qu'un maximum théorique de 65536 connexions concourantes sans collisions peuvent être traité. En pratique ce nombre est plus petit. De plus, avec une implémentation utilisant un *socket* par client, la limite typique sur une machine linux serait d'environ 511[4] connexions. Alors que notre implémentation pourrait en traiter $1024 - N$[5] sans modification de la limite sur le nombre de *file descriptor*.

[1] Opérations *thread safe* ne requérant pas de verrou

[2] Le nombre de *file descriptor* maximum autorisé par le système d'exploitation pourrait être atteint.

[3] *memcmp* utilise SSE 4.2 comme observé dans le *call graph*.

[4] $\frac{1}{2} \cdot ulimit_{typique} - 1$ où $ulimit_{typique}$ est la valeur typique du nombre maximal de *file descriptor* par processur, le $\frac{1}{2}$ est dû au socket a à l'ouverture de fichier et le 1 fait référence au socket global.

[5] $ulimit_{typique} - N$ où N est le nombre de *stream*.

3.2 *Streams* - queue de communication

Comme mentionné ci-dessus, la communication entre les différents *threads* est assurée par des queues de communications que nous appelons *streams*. Il s'agit de liste chaînées d'éléments de types *s_node_t*. Elle utilise des opérations atomiques pour l'insertion ce qui permet à l'opération *enqueue* de ne bloquer que très rarement en faisant du *busy waiting* au lieu d'employer des *mutexes*. Cela permet au *receiver* de ne jamais attendre pour insérer dans le stream. En revanche l'opération *dequeue* est protégée par un *mutex* et emploie une variable de condition[(1)] pour attendre la dispobilité de nouveaux éléments.

3.3 *Syscalls* avancés

Les operations typiques effectuées sur des *sockets* sont *recv*, *recvfrom*, *send* et *sendto*. Bien qu'il n'y ait rien de mal avec ces implémentations, l'usage des *syscalls sendmmsg* et *recvmmsg* sont suggerés comme amélioration disponible[1] de ces fonctions de base. Dans notre implémentation, l'usage de *recvmmsg* est particulièrement intéressant car il permet de recevoir plus d'un paquet à la fois rendant possible la réception d'une fenètre complète en un *syscalls* au lieu de 31 avec *recvfrom*/*recv*. De plus, *recvmmsg* permet de lire l'adresse IP et le port du client afin d'effectuer une recherche dans la table de hachage. Et *sendmmsg* permet d'envoyer un seul *ACK* pour une série de paquets reçus mais également pour chaque paquet hors de séquence ou corrompu. Ce qui diminue à nouveau le nombre de *syscalls* effectués.

Il est intéressant de noter que cet avantage est très important[(2)] pour un nombre limité de *receiver* mais diminue lorsque ce nombre augmente. De même, lorsque le nombre de client augmente, l'efficacité de ce système diminue pour faire l'aquisition d'une fenètre complète. Cependant, l'avantage premier qui est de diminuer le nombre de *syscalls* reste constant. C'est donc purement l'avantage du point de vue de l'usage de la bande passante qui diminue et non celui du nombre réduit de *syscalls*.

3.4 *Buffers* - implémentation de la window

3.5 *Receiver* thread

Le *receiver* thread est responsable pour la réception des paquets, identifier l'émetteur du paquet au travers de la table de hachage et de les transmettre au travers du *stream* aux *handlers*. Elle s'occupe également d'identifier les nouveaux clients et des les ajouter à la table.

3.6 *Handler* thread

3.7 Multiple *Pipelines*

References

[1] T. Richter M.J. Christensen. Achieveing reliable udp transmission at 10 gb/s using bsd socket for data acquisition systems.

(1) Permet de déverouiller un *mutex* en attendant d'être notifié. Un autre *thread* peut ensuite notifier la condition et le *mutex* sera automatiquement verouillé. Il est intéressant de noter que la documentation mentionne qu'il est autorisé de notifier une variable conditionelle sans posséder le *mutex* ce qui permet de maintenir le *dequeue* sans verrou.

(2) cfr. partie performance