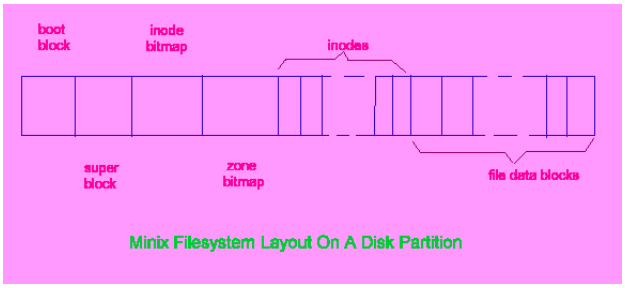
h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Rapport

Minix File system

ITI 2ème soir
2014 / 2015



Sebastien Chassot, Andre-Luc Robyr

Juin 2015

Table des matières

1 Introduction

2 machine d'état

Le mieux est de faire une machine d'état afin que le client et le server soient toujours dans le même état.

Dans le cas de ce TP, il y a trois cas possibles

1. le client fait un READ (0x0)
2. le client fait un WRITE (0x1)
3. le client ferme la connexion (0x2)

READ

— la requête client est un read + un offset + length
— le server return ack + payload

WRITE

— la requête client est un write + un offset + payload
— le server return un ack

CLOSE

— la requête client est un close
— le server return un ack

3 Le serveur de blocs

la problématique de la déconnexion du client

Détecter la déconnexion d'un client n'est pas trivial. En effet, lors d'un *shutdown()*,

Solutions envisagées

Une des solutions les plus simple serait de traiter un échange à la fois ; le client se connecte, envoit une requête, reçoit la réponse et se déconnecte. Le server fait la même chose ; attend sur *accept()* qu'un client se connecte, attend une requête y répond et se déconnecte.

Cette solution est simple, permet de mieux maîtriser l'état du client et du server mais est couteuse en connexions.

Une autre solution est d'accepter un client et rentrer dans une boucle, un *fork()* ou un thread et traiter les requêtes du client tant qu'il ne se déconnecte pas. Cette solution est plus élégante mais elle nécessite de détecter la déconnexion du client.

Une alternative possible est d'ajouter au protocole une requête (un message) de déconnexion qui synchronise le client et le server.

Problème de client malicieux

Si un client lance une requête et annonce une certaine longueur, le server va faire un read, tant que le client maintient la connexion et n'envoie pas les données, il bloque le server.