ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΣΥΣΤΗΜΑΤΟΣ

3Η ΕΡΓΑΣΙΑ 2017

Δημήτριος Κωνσταντάκης – 1115201300079

Η εργασία έχει υλοποιηθεί σε Ubuntu Linux 12.04 και σε γλώσσα C++.

Η άσκηση αποτελείται από τα εξής τμήματα:

MirrorInitiator

Το αρχείο MirrorInitiator δέχεται τα ορίσματα από το command line και στη συνέχεια καλεί μία από τις δύο ρουτίνες,είτε την initiate_Mirroring είτε την terminate_Servers και στην συνέχεια την receive_Response.

Χρήσιμες οδηγίες για εκτέλεση:

Όταν θέλουμε να εκτελέσουμε ένα αίτημα κατοπτρισμού τότε η εντολή είναι όπως στην εκφώνηση. Όταν όμως θέλουμε να τερματίσουμε όλους τους servers αυτό το επιτυγχάνουμε με μια παρόμοια εντολή με την εξής διαφορά: Για κάθε ζεύγος ContentServer Address και Port ,στο πεδίο του dirfileX βάζουμε τη λέξη Termination. Επίσης , βάζουμε στο τέλος της εντολής μετά το αίτημα κατοπτρισμού τη λέξη –exit (πληροφορία για τον MirrorServer). Παράδειγμα :

Εντολή για Τερματισμό:

./initiator –n <MirrorServerAddress> -p <MirrorServerPort> -s <ContentServerAddress1: ContentServerPort1:Termination:0,…> -exit

MirrorServer

Ο MirrorServer κατά την εκκίνηση του , αφού δημιουργήσει μία κλάση MirrorServer καλεί την ρουτίνα masterThread, η οποία δέχεται αιτήματα κατοπτρισμού. Πριν μπεί στην δομή επανάληψης έχει δημιουργήσει τα νήματα workers και τους έχει δώσει τις παραμέτρους που χρείαζονται. Όταν λάβει ένα αίτημα κατοπτρισμού , τότε δημιουργεί μία λίστα με τα requests και στη συνέχεια δημιουργεί τα νήματα mirror Managers. Κάθε ένα νήμα από αυτά παίρνει ένα request.

managers threads: Λαμβάνουν ως όρισμα ένα request . Ελέγχουν αν υπάρχει ο content Server και αν το μήνυμα αυτό δεν περιέχει την πληροφορία τερματισμού και στη συνέχεια συνδέονται με το Content Server και του στέλνουν αίτημα LIST. Αφού διαβάσουν τον αριθμό των files/directories που θα λάβουν ,καθώς και το local path του Content Server τότε διαβάζουν ένα ένα τα ονόματα από τον ContentServer. Κάθε ένα όνομα που διαβάζουν το κάνουν push στην Ουρά (recordQueue), αφού γίνει match με το dirfileX του αιτήματος.

queue: Η ουρά αυτή ακολουθεί την λειτουργία Producer-Consumer. Όταν ένα αντικείμενο γίνεται push ή pop τότε με τη χρήση mutex εξασφαλίζεται ότι κάθε φορά μόνο ένα thread θα βρίσκεται στο Critical Section. Όταν εισάγουμε ένα αντικέιμενο (Record) τότε κάνουμε broadcast στους workers (cond_nonempty). Στην σύναρτηση pop η pthread_cond_wait μπλοκάρει μέχρι να λάβει το signal (cond_nonempty) από την push. (η pop καλείται από τους workers) .

workers threads: Οι workers μπλοκάρουν στην pop μέχρι να λάβουν signal ότι υπάρχει κάποιο Record μέσα στην ουρά. Όποιος καταφέρει και ξυπνήσει και πάρει το Record τότε αρχικά αυξάνει την μεταβλητή runningWorkers (μέσω της pop) και στη συνέχεια τσεκάρει αν είναι Record Τερματισμού.(Τα Record τερματισμού έχουν address Termination και χρησιμοποιούνται για να τερματιστούν οι workers (όταν

θέλουμε να τερματίσουμε τον MirrorServer δημιουργούμε τόσα Records τερματισμού όσοι είναι και οι workers)). Αν είναι κανονικό Record τότε ελέγχουμε αν είναι Directory . Αν είναι directory δεν χρειάζεται να κάνουμε fetch οπότε απλά καλούμε την συνάρτηση create_Directory (αφού κλειδώσουμε τον mutex filemtx). Μόλις ολοκληρώσει την δημιουργία φακέλων και ξεκλειδωθεί ο filemtx τότε μειώνει τους runningWorkers κατά 1. Αν δεν είναι directory, τότε είναι file. Σε αυτήν την περίπτωση δημιουργεί μια σύνδεση με τον ContentServer του Record και καλεί την fetch αφού κλειδώσει τον filemtx. Η fetch φτιάχνει πρώτα όλα τα directories και στην συνέχεια το όνομα του file . Μετά από αυτά στέλνει αίτημα Fetch στον Content Server και λαμβάνει το αρχείο(σε πακέτα , χρήση της fwrite). Αφού ολοκληρωθεί η μεταφορά αποθηκεύει τον αριθμό των bytes , ξεκλειδώνει τον filemtx και μειώνει τους runningWorkers. Αν οι runningWorkers γινουν 0 κατά την μείωση και η λίστα είναι άδεια τότε στέλνεται signal στην cond_alldone.

Η main thread αφού αρχικοποιήσει τους managers τότε καλεί την terminate_Mirroring η οποία μπλοκάρει μέχρι να να λάβει signal (cond_alldone) και οι μεταβλητές queueSize και runnignWorkers να είναι 0. Όταν αδειάσει η ουρά και δεν τρέχει κάποιος worker (συνθήκη while0 τότε στέλνει τα αποτελέσματα στην Mirror Initiator.

ContentServer

Ο ContentServer κατά την εκκίνηση του , αφού δημιουργήσει μία κλάση ContentServer καλεί την ρουτίνα masterThread, η οποία δέχεται αιτήματα LIST, FETCH και Termination. Κάθε φορά που δέχεται ένα αίτημα LIST ή FETCH δημιουργεί ένα thread (list_thread, fetch_thread) .

list_thread: Δημιουργεί μια δομή delayNode και την βάζει στην λίστα. Κάθε delayNode έχει το δικό του id , το οποίο στέλνεται από τον MirrorServer (= requestID). Σε μία άλλη λίστα FileList βάζει κόμβους fileNode όπου ο καθένας έχει το όνομα και τον τύπο του αρχείου (false for Directory , true for File).Αρχικά στέλνουμε το μέγεθος της λίστας, το local path του ContentServer και για κάθε κόμβο στην λίστα δημιουργείται ένα όνομα (<type>:<dirorfilename>) και στέλνεται στον manager thread. Η λίστα αυτή καταστρέφεται μόλις ολοκληρωθεί το αίτημα.

fetch_thread: Με βάση το id βρίσκει το delay και κάνει sleep για τόσα seconds. Στην συνέχεια ανοίγει το αρχείο που έλαβε στο αίτημα και στέλνει το μέγεθος του σε bytes. Στη συνέχεια στέλνει ανά πακέτα των 256 bytes το αρχείο.

Χρήσιμες Πληροφορίες:

1. Στα path που δίνονται για τους servers (mirror ή Content) το τελικό directory πρέπει να μην έχει στο τέλος τον χαρακτήρα ‘/’.
2. Χρησιμοποιήθηκε ο κώδικας για τα sockets από τα παραδείγματα στις διαφάνειες.
3. Για την αναδρομική συνάρτηση add_dir_content βρήκα πληροφορίες στο link https://stackoverflow.com/questions/4204666/how-to-list-files-in-a-directory-in-a-c-program
4. Έχουν δημιουργηθεί οι φάκελοι ContentServer1, ContentServer2, ContentServer3 και Storage ως παραδείγματα.
5. Έχει υλοποιηθεί το Makefile (make, make clean)