




ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΣΥΣΤΗΜΑΤΟΣ

ΑΣΚΗΣΗ 3^η

Η άσκηση αυτή υλοποιήθηκε σε *bash* και *C++* (χωρίς χρήση των *STL* και *std::string*). Το παραδοτέο αποτελείται από τους εξής φακέλους:

-  **webcreator** : περιέχει το ζητούμενο *bash* script *webcreator.sh* . Μπορείτε να τρέξετε το script από αυτόν τον φάκελο ως **`./webcreator.sh root_directory text_file w p`**.
-  **webserver** : περιέχει τον πηγαίο κώδικα του *webserver* υλοποιημένο σε *C++* μαζί με ένα *makefile* για την μεταγλώττιση του κώδικα. Μπορείτε να μεταγλωττίσετε και να τρέξετε τον *server* από αυτόν τον φάκελο ως **`./myhttpd -p serving_port -c command_port -t num_of_threads -d root_dir`**.
-  **webcrawler** : περιέχει τον πηγαίο κώδικα του *webcrawler* υλοποιημένο σε *C++* μαζί με ένα *makefile* για την μεταγλώττιση του κώδικα. Επίσης, περιέχει τον κώδικα του *jobExecutor* μαζί με το δικό του *makefile* στον υποφάκελο *webcrawler/jobExecutor*. Το *makefile* του *crawler* καλεί κατάλληλα και το *makefile* του *jobExecutor*. Πρέπει να τρέξετε τον *webcrawler* από τον φάκελο *webcrawler* αναγκαστικά, έτσι ώστε το *path* που δίνω ως ορίσμα (γίνεται *#defined* στο *«executables_paths.h»*) στην *execcl()* μέσα στον κώδικα να είναι σωστό και η *execcl()* να μην αποτύχει, αλλιώς το πρόγραμμα θα τερματίσει απευθείας με *error message* ότι δεν βρήκε το εκτελέσιμο του *jobExecutor*. Μπορείτε να τρέξετε τον *web crawler* ως **`./mycrawler -h host_or_IP -p port -c command_port -t num_of_threads -d save_dir starting_URL`**.

Οι σειρές των επιλογών για τους *crawler* και *server* δεν έχει σημασία (αρκεί το *starting_url* για τον 1^ο να είναι τελευταίο). Η επιλογή *-t* μπορεί να παραληφθεί, στην οποία περίπτωση μία *default* τιμή (4) θα δοθεί. Ο αριθμός των *worker* του *jobExecutor* για τον *crawler* είναι *#defined* (σε 5) στο πηγαίο *webcrawler.cpp* (αν υπάρξουν λιγότεροι από τόσοι φάκελοι ο αριθμός αυτός θα μειωθεί στο πλήθος τους στον *jobExecutor*).

Ακολουθεί μια γενική περιγραφή των τριών εφαρμογών. Πιο συγκεκριμένη και αρκετά λεπτομερή επεξήγηση θα βρείτε σε *comments* στον ίδιο τον κώδικα.

I) WEB CREATOR

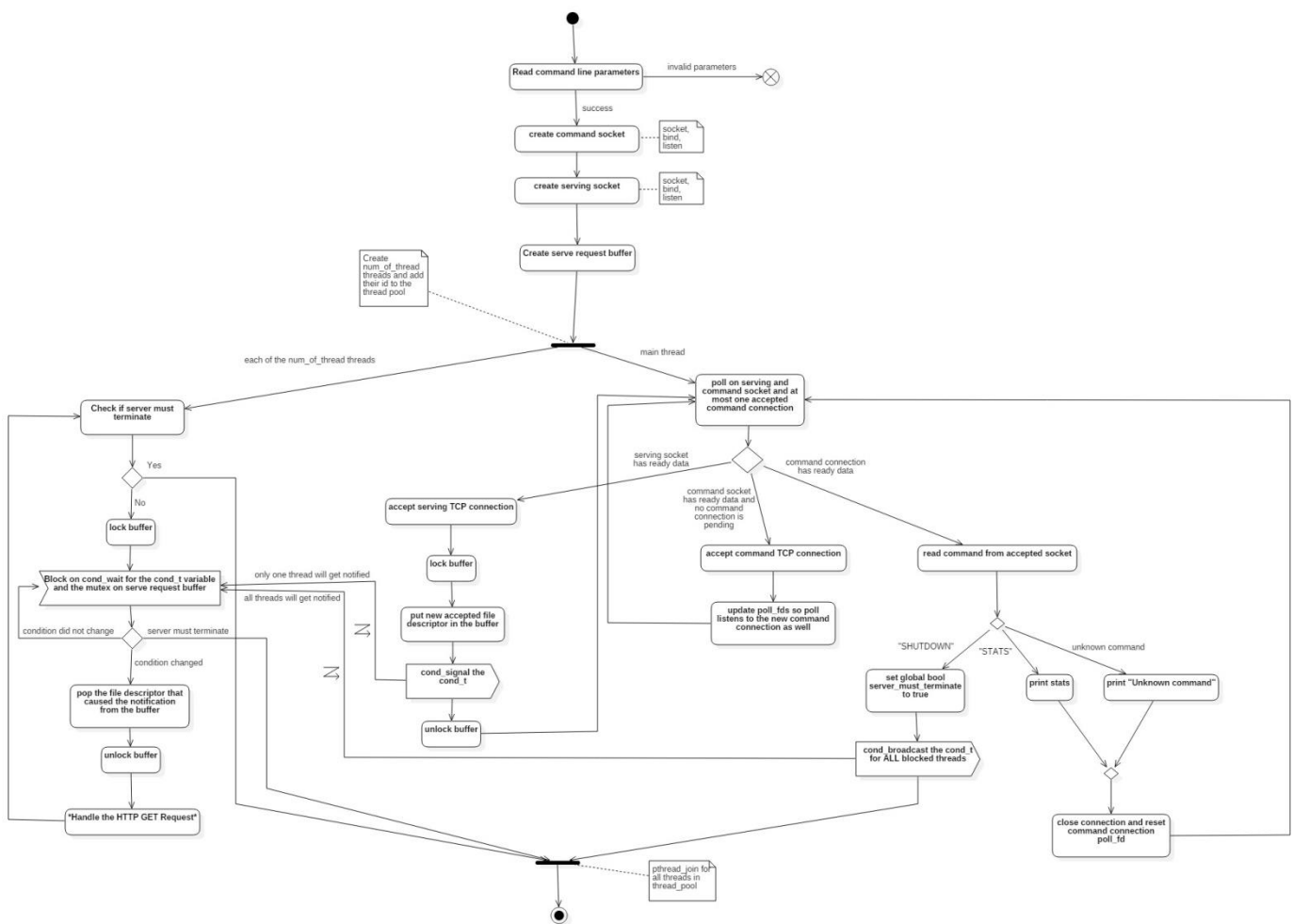
Το *webcreator.sh* ακολουθεί κατά βήμα τις οδηγίες τις εκφώνησης και υπάρχουν *step-by-step comments* που εξηγούν τι γίνεται που. Εδώ θα ήθελα να σημειώσω τα εξής:

- Δεν υπάρχει πρόβλημα αν το *text_file* τελειώνει τις γραμμές σε *'\r\n'* ή σχέτο *'\n'*. Το αποτέλεσμα θα είναι ιστοσελίδες που τελειώνουν μόνο σε *'\n'*. Προσωπικά χρησιμοποίησα το βιβλίο στο *link* της εκφώνησης (το οποίο τελειώνει τις γραμμές σε *'\r\n'*).
- Εισάγω σε κάθε σελίδα ένα *link* στο τέλος κάθε «πακέτου» *m / (f+q)* γραμμών. Αυτό το *link* θα εισαχθεί στην τελευταία γραμμή αυτού του πακέτου σε τυχαίο σημείο ανάμεσα (ή και στην αρχή/τέλος) των λέξεων της γραμμής και μπορεί τυχαία να είναι *internal* ή *external link* (ανάλογα και με το τι *links* έχουν ήδη χρησιμοποιηθεί).
- Οι ακραίες περιπτώσεις για τα *w* και *p* αντιμετωπίζονται σύμφωνα με ότι ειπώθηκε στο *piazza*, δηλαδή:
 - Αν *f == p*, τότε επιτρέπεται η χρήση της ίδιας της σελίδας ως *internal link*, για να υπάρξουν αρκετά *internal links*.

- Αν $w=1$, τότε αντί για την χρήση του τύπου κάνω το $q=0$, αφού δεν μπορούν να υπάρξουν external links αν έχουμε μόνο ένα site directory.
- Αν $w=2$ και $p=1$, τότε αντί για την χρήση του τύπου κάνω το $q=1$, αφού μόνο μία εξωτερική σελίδα υπάρχει για να χρησιμοποιηθεί ως external link.
- Τα links, internal και external, είναι όλα της μορφής: `"../sitei/pagei_j.html"` . Με αυτόν τον τρόπο μπορεί κανείς να περιηγηθεί σε αυτά κλικάροντάς τα μέσω browser και με και χωρίς τον server (δηλαδή και απευθείας μέσω file system). Ο server αν πάρει HTTP GET με link που αρχίζει με `".."` απλά θα αγνοήσει αυτές τις `".."` και θα κρατήσει το root-relative κομμάτι: `"/sitei/pagei_j.html"`.
- Αν μετά την όλη διαδικασία κάποια σελίδα δεν έχει incoming link τότε αυτό θα εκτυπωθεί ως warning μήνυμα που θα αναφέρει ποια σελίδα είναι αυτή. Αν δεν υπάρχει τέτοια σελίδα τότε θα εκτυπωθεί κανονικά το `"All pages have at least one incoming link"`.

II) WEB SERVER

Για τον σχεδιασμό του web server χρησιμοποίησα (πριν γράψω κώδικα) την UML για να φτιάξω ένα UML activity diagram, που ουσιαστικά εξηγεί τον βασικό σχεδιασμό του webserver (υπό την έννοια της ροής εκτέλεσης του του κάθε thread). Το σχεδιάγραμμα αυτό βρίσκεται και στον ίδιο φάκελο με το README υπό μορφή εικόνας (είναι το `"webserver.jpg"`) και είναι το παρακάτω:



Να σημειωθεί ότι το παραπάνω σχεδιάγραμμα απλά απεικονίζει την βασική λογική του server (αφού αναπτύχθηκε πριν την υλοποίηση) και όχι ακριβώς τι γίνεται στον τελικό κώδικα, πχ:

- Στην πραγματικότητα στο `accepted command fd` διαβάζω με `buffer` δεδομένα και μπλοκάρω στην `poll` επαναληπτικά μέχρι να πάρω '\n' και τότε να εκτελέσω το `command` και όχι πχ την πρώτη φορά που η `poll` αναφέρει δεδομένα όπως μπορεί να παρερμηνευτεί από το παραπάνω σχήμα.
- Επίσης, λ.χ., το «`print stats`» και το `update` τους που γίνεται μέσα στο «`*Handle HTTP GET Request*`» εμπεριέχει κι αυτό ένα `mutex` καθώς πρόκειται για κοινά δεδομένα για γράψιμο&διάβασμα από πάνω από ένα `threads`).

Ορισμένες σχεδιαστικές επιλογές που θα ήθελα να αναφέρω εδώ είναι οι εξής:

- Ανα πάσα στιγμή το `main thread` κάνει `monitor` δύο ή τρία `file descriptors`: το `serving socket`, το `command socket` και (αν υπάρχει εκείνη την στιγμή) **ένα** `accepted command socket` από όπου θα διαβάσει μία εντολή μέχρι το '\n'. Αυτό το `monitoring` γίνεται έτσι ώστε να μπορεί ο `server` να ικανοποιεί `serving requests παράλληλα` με (το πολύ **ένα**) `command request`.
- Όταν γίνει δεχτό ένα `command connection`, δεν θα γίνει δεκτό άλλο `command` μέχρι το προηγούμενο να κλείσει.
- Στα `HTTP GET Request`, το `link` που ζητείται από τον `server` πρέπει να είναι ένα `root-relative link`, που για τα πλαίσια της άσκησης θα είναι της μορφής: «`/sitei/pagei_j.html`», όπου `i, j` θετικοί ακέραιοι. Επίσης, επιτρέπεται να είναι της μορφής «`../sitei/pagei_j.html`», οπότε ο `server` θα αγνοήσει τις πρώτες δύο «`..`» (έτσι είναι τα `link` του `webcreator`). Ο `server` θα συνενώσει το `root_dir` με αυτό το `root-relative link` για να πάρει το ολοκληρωμένο `filepath` για το ζητούμενο αρχείο.
- Ένα `HTTP GET Request` θεωρείται `valid` αν: 1. Η πρώτη γραμμή είναι της μορφής «`GET <link> HTTP/1.1`», 2. Υπάρχει πεδίο «`Host:`» και 3. Κάθε άλλο πεδίο τελειώνει σε `:'`. Υποθέτω πως κάθε `HTTP GET request header` θα τελειώνει με `"\n\n"` ή `"\r\n\r\n"` αναγκαστικά.
- Τα `HTTP GET responses` που στέλνει ο `server` συμφωνούν με το πρωτόκολλο `HTTP` και μπορεί να είναι:
 - 200 OK: Η σελίδα βρέθηκε και έχει σταλεί στο `content` πεδίο
 - 400 Bad Request: Το `HTTP GET Request` που ελαβε ο `server` δεν ήταν `valid`
 - 403 Forbidden: Ο `server` δεν είχε δικαίωμα για άνοιγμα του ζητούμενου αρχείου (για `read`)
 - 404 Not Found: Ο `server` δεν βρήκε το ζητούμενο αρχείο

III) WEB CRAWLER

Για τον σχεδιασμό του `web crawler` είχα επίσης κάνει αντίστοιχο διάγραμμα σε `UML`, αλλά η υλοποίησή μου άλλαξε τόσο πολύ αφότου την ξεκίνησα, που πλέον δεν αντικατοπτρίζει το τι γίνεται οπότε δεν θα το συμπεριλάβω εδώ. Αντιθέτως, θα εξηγήσω με λόγια το πως λειτουργεί ο `crawler`.

Αρχικά ο `crawler` ελέγχει ότι το εκτελέσιμο του `jobExecutor` βρίσκεται στην `#defined` θέση του, κάνει `parse` τις παραμέτρους του και καλεί τις `socket`, `bind`, `listen` για το `command socket` του. Έπειτα θα δημιουργήσει τις `global` δομές του οι οποίες είναι οι εξής:

1. Μία `FIFO Queue` (`FIFO_Queue.h` - αντίστοιχη του «`serve_request_buffer`» του `server`) - μη περιορισμένου μεγέθους - ονομάτι «`urlQueue`», η οποία φυλλάει τα `URLs` που έχουν βρεθεί και ο `crawler` θα ζητήσει από τον `server`. Τα `URLs` σε αυτήν την δομή μπορεί να είναι είτε `root-relative` είτε `full HTTP URLs` (πχ: `http://linux01.di.uoa.gr:8080/site0/page0_1234.html`), αναλόγως με το τι `link` βρήκε ο `crawler` (στο `starting_url` ή σε ήδη κατεβασμένη σελίδα).
2. Μία γρήγορη δομή αναζήτησης για `strings` (`str_history.h` : ένα `search tree` με – on average αν το δέντρο είναι ισοροπομένο - $O(\log n)$ εισαγωγή και αναζήτηση) ονόματι «`urlHistory`», η οποία φυλλάει το `root-relative` κομμάτι των (ήδη `root-relative` ή `full HTTP`) `URLs` που έχουν ήδη τοποθετηθεί στην ουρά στο παρελθόν, έτσι ώστε να μην τα ξαναβάλουμε δεύτερη φορά σε αυτήν.

Μοναδική εξαίρεση αποτελούν τα full HTTP URLs που τοποθετούνται στην `urlQueue` αλλά αφορούν διαφορετικό TCP socket (`sockaddr_in`) από αυτό που πήρε ο crawler στα command line arguments του. Το root-relative κομμάτι αυτών των URLs δεν εισάγεται στην `urlHistory`, έτσι ώστε να μην υπάρξει conflict με ίδιο root-relative κομμάτι από URLs για το TCP socket του δικού μας server. Αυτό δεν δημιουργεί κανένα πρόβλημα για αυτά τα URLs, αφού στα πλαίσια της άσκησης δεν τα κατεβάζω όταν γίνουν popped από το `urlQueue`, όπως θα εξηγηθεί παρακάτω. Αν ήθελα να τα κατεβάσω θα έπρεπε ως ιστορικό να σώζω και το root-relative url και το μοναδικό `sockaddr_in` κάθε server (το οποίο θα ήταν κάποιο struct και όχι απλά ένα string).

Ο λόγος που εδώ φυλλάω μόνο root-relative URLs και όχι full HTTP URLs είναι ότι η ίδια σελίδα μπορεί να έχει πάνω από ένα full HTTP URL (γιατί πχ το `host_or_IP` μπορεί να είναι host που θέλει DNS look-up ή απευθείας IP διεύθυνση σε dotted notation *a.b.c.d*), ενώ το root-relative link της είναι μοναδικό. Φυσικά, για να είναι μοναδικό πρέπει να ισχύει η υπόθεση ότι κάνουμε crawling μόνο από έναν server (για την ακρίβεια μόνο ένα TCP socket), που για τα πλαίσια της άσκησης μπορούμε να την κάνουμε: ο server αυτός θα είναι ο server στα command line arguments του crawler.

Το πρόγραμμά μου, όμως, για να είναι πιο πλήρες, κάθε φορά που κάνει pop ένα URL από την `urlQueue`:

- αν αυτό είναι root-relative URL, τότε υποθέτει πως ο server είναι αυτός που πήραμε από command line arguments και κάνει connect σε αυτόν για να ζητήσει την σελίδα.
- αν αυτό είναι full HTTP URL, τότε θα το κάνει parse (με thread safe τρόπο) ώστε να βρεί το `host_or_IP` και το `port_number` (αν δεν υπάρχει θεωρείται το 8080), θα βρεί το αντίστοιχο `sockaddr_in` (το οποίο είναι μοναδικό για κάθε socket) και έπειτα:
 - αν αυτό ταυτίζεται με το `sockaddr_in` του server που πήραμε από command line arguments, τότε θα κάνει connect σε αυτόν για να ζητήσει την σελίδα.
 - αλλιώς, αν πρόκειται για άλλον server (ή τον ίδιο server αλλά σε άλλο port), τότε δεν θα κάνει connect σε κανέναν server αφού δεν θα κατεβάσει αυτήν την σελίδα. Αντιθέτως, απλά θα εκτυπώσει ένα μήνυμα ότι βρήκε ένα URL για διαφορετικό TCP socket από αυτό των command line arguments και θα αγνοήσει αυτό το link.

3. Την ίδια δομή γρήγορης αναζήτησης για strings (`str_history.h`) για την μεταβλητή "**alldirs**", η οποία φυλλάει «ένα ιστορικό» με το file path όλων των folders στους οποίους ο web crawler κατέβαζε σελίδες κατά την διάρκεια του web crawling. Αυτοί θα είναι οι φάκελοι (που θα πρέπει να περιέχουν μόνο text/html αρχεία) οι οποίοι θα περαστούν στον `jobExecutor`, ο οποίος με την σειρά του θα τους μοιράσει στους `workers` του.

Η `urlQueue` αρχικοποιείται και ύστερα εισάγεται σε αυτήν το `starting_url`, ενώ η `urlHistory` αρχικοποιείται και εισάγεται σε αυτήν το root-relative μέρος του `starting_url`. Να σημειωθεί ότι εδώ δεν χρειάζεται να ελένξουμε την εξαίρεση του το `starting_url` να είναι πλήρες link για άλλο TCP socket, καθώς σε αυτήν την περίπτωση το web crawling δεν θα κατέβαζε καμία σελίδα και άρα δεν θα χρησιμοποιηθεί το ιστορικό. Η `alldirs` αρχικοποιείται άδεια.

Έπειτα, το main thread του `jobExecutor` θα δημιουργήσει ένα **monitor thread** (`crawling_monitoring.cpp`), το οποίο χρησιμοποιείται για να καταλάβουμε ακριβώς πότε τελείωσε το web crawling. Συγκεκριμένα, το thread αυτό θα μπλοκάρει στην κλήση της `cond_wait()` για την μεταβλητή συνθήκης «*crawlingFinished*» και θα ξεμπλοκάρει από την κλήση της `cond_signal()`, όταν το web crawling έχει τελειώσει (ή επειδή ο crawler πρέπει να τερματίσει πρόωρα). Το web crawling έχει τελειώσει όταν:

1. Η `urlQueue` είναι άδεια.
2. Όλα τα `num_of_threads` threads (που θα δημιουργηθούν αμέσως μετά από τον main thread) είναι μπλοκαρισμένα στην `cond_t` «*QueueIsEmpty*».

Η συνθήκη αυτή ελέγχεται από κάθε ένα από αυτά τα `num_of_threads threads` πριν μπλοκάρουν στην `cond_wait()`. Το τελευταίο τέτοιο thread, όταν η `urlQueue` είναι άδεια, πριν μπλοκάρει θα υπολογίσει ότι η συνθήκη αυτή είναι αληθής και θα κάνει `cond_signal()` την `cond_t «crawlingFinished»`, ξυπνώντας έτσι το monitor thread, το οποίο θα δει ότι το web crawling τελείωσε.

Κατόπιν, το thread αυτό θα κάνει `set` μία global bool «*threads_must_terminate*» και `broadcast()` στα `num_of_threads threads` έτσι ώστε αυτά να τερματίσουν, ενώ το ίδιο θα καλέσει την `pthread_join()` για αυτά.

Ύστερα, εφόσον το webcrawling τελείωσε και βρέθηκε πάνω από ένας φάκελος στην `alldirs`, το thread αυτό θα αρχικοποιήσει τον **jobExecutor** ως ξεχωριστή διεργασία καλώντας `fork()` + `exec()` για το εκτελέσιμό του (το οποίο πρέπει να υπάρχει λόγω του αρχικού ελέγχου της `main`). Πριν κληθεί η `exec()` δημιουργούνται δύο pipes και γίνονται οι κατάλληλες ανακατευθύνσεις έτσι ώστε web crawler και jobExecutor να επικοινωνούν μεταξύ τους μέσω pipes (ο jobExecutor θα διαβάζει/γράφει στο `stdin/stdout` αλλά στην πραγματικότητα θα διαβάζει/γράφει στα pipes αυτά). Μέσω αυτών των pipes το monitor thread θα περάσει στον jobExecutor:

1. Το `terminal_width` εκείνη την ώρα το οποίο χρησιμοποιείται από τον jobExecutor για την υπογράμμιση των απαντήσεων της search, επειδή ο ίδιος δεν έχει πλέον access στο «πραγματικό» `stdout`.
2. Όλους τους φάκελους που έχουν αποθηκευτεί στο `alldirs`, ως τους φάκελους (που περιέχουν text/html αρχεία) που ο jobExecutor θα μοιράσει στους workers του, οι οποίοι θα κάνουν το indexing όλων των text αρχείων μέσα σε αυτούς αγνοώντας όλα τα html tags (= οτιδήποτε ανάμεσα σε «<» «>»).

Τέλος, το monitor thread θα μπλοκάρει σε μία read από το pipe from the jobExecutor, ο οποίος θα στείλει ένα μήνυμα «READY» όταν είναι έτοιμος να δεχθεί commands, μέχρι να λάβει αυτό το μήνυμα. Όταν ξεμπλοκαριστεί θα ενημερώσει μια global bool ότι ο jobExecutor είναι έτοιμος για commands και θα τερματίσει.

Αφότου τερματίσει το monitor thread θα μείνει ως “zombie thread”, μέχρι το main thread να το κάνει join λίγο πριν τερματίσει ο crawler.

Επιστρέφοντας τώρα στο main thread, μετά την δημιουργία του monitor thread, θα δημιουργήσει τα **num_of_threads threads** και θα γεμίσει το thread_pool με τα thread ids τους. Αυτά τα threads θα κάνουν pop URLs από την `urlQueue` μέσα σε ένα forever loop, όπως αναφέρθηκε παραπάνω, και , εφόσον το URL που έκαναν pop είναι για το server και το port στα command line arguments:

1. Θα κάνουν connect και θα στείλουν ένα HTTP GET request στον server αυτόν για την σελίδα
2. Θα διαβάσουν το HTTP GET Response από τον server (με buffer) και:
 - a. Αν αυτό δεν ήταν 200 OK (πχ: 404 Not Found ή 403 Forbidden), τότε θα αναφέρουν το περιστατικό στο cout, θα κλείσουν την TCP σύνδεση και θα προχωρήσουν στο επόμενο loop.
 - b. Αν αυτό ήταν 200 OK, τότε θα διαβάσουν όλο το content της απάντησης και θα το σώσουν ως την σελίδα που κατεβάστηκε (αν υπήρχε ήδη τέτοιο αρχείο θα γίνει overwritten) στον κατάλληλο site folder (ο οποίος αν δεν υπάρχει θα δημιουργηθεί).
3. Θα κάνουν crawl το αρχείο που μόλις κατέβασαν για να βρουν άλλα `` links και να τα προσθέσουν στην `urlQueue`, αλλά μόνο αν δεν υπάρχουν ήδη στην `urlHistory`.

Απο εκεί και πέρα το main thread θα μπει σε ένα forever loop μέσα στο οποίο θα κάνει monitor με την χρήση της `poll()` το command socket, μέχρις ότου να δεχθεί εντολή «SHUTDOWN» . Οι εντολές που υποστηρίζει ο crawler είναι οι εξής:

- **SHUTDOWN:** τερματίζει τον crawler (ακόμα κι αν το crawling δεν έχει τελειώσει) και τον jobExecutor και τους workers του.
- **STATS:** εμφανίζει τα ζητούμενα στατιστικά για τον crawler
- **SEARCH word1 word2 ... word10:** Εμφανίζει το αποτέλεσμα της εντολής `“/search word1 word2 ... word10”` στον jobExecutor.
- **MAXCOUNT keyword:** Εμφανίζει το αποτέλεσμα της εντολής `“/maxcount keyword”` στον jobExecutor.

- **MINCOUNT keyword:** Εμφανίζει το αποτέλεσμα της εντολής “/mincount keyword” στον jobExecutor.
- **WORDCOUNT:** Εμφανίζει το αποτέλεσμα της εντολής “/wc” στον jobExecutor (το οποίο είναι λογικό να είναι μικρότερος αριθμός από Bytes από ότι η STATS αφού έχουν αφαιρεθεί τα html tags).

Προκειμένου ο web crawler να ξέρει ότι η απάντηση του jobExecutor έχει τελειώσει, ο τελευταίος πρέπει να στείλει στο cout ένα προσυμφωνημένο μήνυμα «<\n» μετά από κάθε εντολή και πριν μπλοκάρει στο cin για να δεχθεί επόμενη. Το «<» δεν μπορεί να υπάρξει φυσιολογικά στις απαντήσεις του jobExecutor, αφού οι workers του αγνοούν τα html tags. Το «\n» είναι για να γίνει flushed το cout του jobExecutor. Να σημειωθεί ότι η ανάγνωση των απαντήσεων του jobExecutor από τον crawler γίνεται με low-level read system calls με ένα σχετικά μεγάλο buffer (#defined σε 2048 Bytes).

ΠΑΡΑΤΗΡΗΣΕΙΣ ΓΙΑ WEBCRAWLER ΚΑΙ ΣΥΝΔΕΣΗ ΜΕ ΑΣΚΗΣΗ 2 (JOBEXECUTOR)

- Στην άσκηση 2, είχα κάνει την παραδοχή ότι ο jobExecutor και οι workers του δεν μπορούν να τερματίσουν με «/exit» (ούτε θα έπρεπε με την χρήση terminating signal) πριν τελειώσει το text file parsing των workers. Αυτό σημαίνει ότι άπαξ και ο jobExecutor γίνει initialized από το monitoring thread, προκειμένου να τερματιστεί ομαλά ο crawler θα πρέπει αναγκαστικά να περιμένει να τελειώσει το textfile parsing των workers. Αφού τελειώσει αυτό, τότε ο jobExecutor θα εκτελέσει την «/exit» εντολή που του στέλνει ο crawler και θα τερματίσει.

Ενναλλακτικά, αν θέλαμε να τερματίζουμε κατεθείαν πάση θυσία θα μπορούσαμε να δοκιμάσουμε να στέλναμε terminating signal αντί για exit στον jobExecutor, αλλά επειδή αυτά τα signals δεν γίνονταν handle πριν τελειώσει το text file parsing στην Άσκηση 2, αυτό θα σήμαινε ότι ο jobExecutor θα τερμάτιζε με leaks ενώ οι workers δεν θα τερμάτιζαν ποτέ, πράγμα που δεν θέλουμε.

- Το παραπάνω δεν ισχύει στην περίπτωση που το SHUTDOWN το στείλουμε πριν γίνει initialized ο jobExecutor από το monitoring thread, δηλαδή κατά την διάρκεια του webcrawling, καθώς σε αυτήν την περίπτωση ο jobExecutor δεν θα εκτελεστεί ποτέ από το πρόγραμμα.
- Το γεγονός ότι ο jobExecutor (για την ακρίβεια οι workers του) πρέπει να τελειώσει το text file parsing πριν μπορεί να δεχθεί εντολές επηρεάζει τις εντολές προς τον jobExecutor του crawler με τον εξής τρόπο: αν δοθεί τέτοια εντολή (SEARCH, MAXCOUNT, MINCOUNT, WORDCOUNT) και ο jobExecutor δεν είναι έτοιμος να την δεχθεί, τότε δεν θα του την στείλουμε (καθώς η εκτέλεση θα μπλόκαρε εκεί μέχρι ο jobExecutor να ετοιμαστεί και να απαντήσει), αντιθέτως θα εκτυπώσουμε στο socket ότι ο jobExecutor δεν είναι ακόμα έτοιμος για εντολές και θα κλείσουμε την σύνδεση.

ΑΛΛΕΣ ΠΑΡΑΤΗΡΗΣΕΙΣ

- Ο *save_dir* φάκελος στον οποίο ο web crawler κατεβάζει ιστοσελίδες, δεν γίνεται purge. Αυτό σημαίνει ότι μπορεί να χρησιμοποιηθεί και μη άδειος φάκελος για την δουλειά αυτή (οι φάκελοι που θα περαστούν στον jobExecutor θα είναι μόνο αυτοί στους οποίους κατέβηκαν σελίδες κατά το crawling χάρης στην δομή *alldirs*), αρκεί να μην περιέχει υποφακέλους της μορφής *sitei/* που να κάνουν conflict με αυτούς που φτιάχνει ο crawler. Επίσης, όμως σημαίνει ότι είναι ευθύνη του χρήστη της εφαρμογής να φροντήσει για την κατάσταση του *save_dir* πριν εκτελέσει τον crawler (πχ ειδικά σε διαδοχικές εκτελέσεις όπου γίνονται crawl διαφορετικές σελίδες ίδιων σε ονομασία site).