Contents

./storage_server.go

```go
package main

import "./storage"

func main() {
    dblisten, err := storage.StartServer()
    if err != nil {
        panic("ERROR: No se pudo levantar el servidor de storage")
    }
    storage.ProcessRequests(dblisten)
    dblisten.Close()
}
```

./worker_server.go

```go
package main
import (
    "./worker"
    "sync"
)
func main() {
    //Set up workers
    resourcesmap := make(worker.ServerResourcesMap)
    lCh := make(chan string, 10000)
    wg := new(sync.WaitGroup)
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go worker.Worker(i, lCh, wg, resourcesmap)
    }

    ns, err := worker.NameServer(lCh)
    if err != nil {
        panic("No se pudo levantar el NameServer")
    }
    worker.ProcessRequests(ns, lCh)

    close(lCh)
    wg.Wait()
}
```

./storage/model.go

```go
package storage
import (
    "strings"

)
const (
    Dir = iota
    Link
    File
)
type Node struct {
    Type int
    Size int
    Path string // must have trailing /
    Files map[string]*Node
}
type NodeMap map[string]*Node
type Server struct {
    Hostname string
    Finished bool
    Root_dir *Node
}

type ServerMap map[string]*Server
func StrNodeType(node Node) string {
    if node.Type == Dir {
        return "dir"
    } else if node.Type == Link {
        return "link"
    } else {
        return "file"
    }
}
// Devuelve el nodo del subdirectorio pedido. El segundo
// elemento es true si hubo exito, o false si no se encontro.
func GetSubdir(path string, root_dir *Node) (*Node, bool) {
    steps := strings.Split(path, "/")
    node := root_dir
    for _, subdir := range steps {
        if subdir == "" {
            continue
        }

        _node, ok := node.Files[subdir]
        if ! ok {
            return nil, false
        }
        node = _node
    }
    return node, true
}

func updateParentsSize(path string, root_dir *Node, size int) {
    //TODO: factorizar usando funciones de orden superior
    node := root_dir
    node.Size = node.Size + size
    for _, subdir := range strings.Split(path, "/") {
        //fmt.Printf("Escribiendo nodo %s, bajando a subdir %s\n", node.Path, subdir)
        if subdir == "" {
            continue
        }
        node = node.Files[subdir]
        node.Size = node.Size + size
    }
}
func AddDir(dict ServerMap, server string, node Node) {

    // It always exists
    dbnode, _ := GetSubdir(node.Path, dict[server].Root_dir)
    /*encoded, _ := json.Marshal(node)
    fmt.Printf("add_dir a escribir: %s\n", encoded)
    encoded, _ = json.Marshal(node)
    fmt.Printf("add_dir -- en la DB: %s\n", encoded)*/
    dbnode.Files = node.Files
    updateParentsSize(node.Path, dict[server].Root_dir, node.Size)
}
func ShallowCopy(n Node) Node {
    shallow := Node{n.Type, n.Size, n.Path, make(NodeMap)}
    for k, v := range n.Files {
        shallow.Files[k] = &Node{v.Type, v.Size, v.Path, make(NodeMap)}
    }
    return shallow
}

func GetDir(dict ServerMap, host string, path string) ResultsResponse {
    var response ResultsResponse
    if server, ok := dict[host]; ok {
        response.Finished = server.Finished
        response.Node = Node{File, 0, "/", make(NodeMap)}
        node, exists := GetSubdir(path, server.Root_dir)
        if exists {
            shallow_node := ShallowCopy(*node)
            response.Node = shallow_node
        }

    } else {
        response.Finished = false
        response.Node = Node{File, -1, "/", make(NodeMap)}
    }
    return response
}
```

./storage/transfer.go

```go
package storage

import (
    "encoding/json"
    "fmt"
    "net"
    "bufio"
    "time"
)
const (
    Read = iota
    Write
    Newserver
    Finishserver
)
const DBHOSTPORT = "db:11000"

type Query struct {
    Type int
    Hostname string
    Node Node
}
type ResultsResponse struct {
    Finished bool
    Node Node
}
// Wrap sencillo de write para poder imprimir e ir viendo.
func send(conn net.Conn, s string) {
    //fmt.Printf(">%s\n", s)
    buf := []byte(s)
    count := 0
    for count < len(buf) {
        byteSent, err := conn.Write(buf[count:])
        count += byteSent
        if err == nil {
            return
        }
    }
}
// Manda una query al servidor de storage.
// Devuelve la respuesta, y si falló la transmisión o no.
// La transmisión puede ser reintentable.
func SendQuery(query Query) (string, bool) {
    conn, err := net.Dial("tcp", DBHOSTPORT)
    if err != nil {
        fmt.Println(err)
        return "", false
    }
    defer conn.Close()

    bytejson, err := json.Marshal(query)
    if err != nil {
        fmt.Println("Error de serializacion:", err)
        //De este error no se puede recuperar (query malformada)
        panic("Query no valida")
    }

    send(conn, fmt.Sprintf("%s\n", bytejson))
    connbuf := bufio.NewReader(conn)
    str, e := connbuf.ReadString('\n')
    if e != nil {
        return "", false
    }
    fmt.Printf("Recibido de la DB -- %s --\n",str)
    return str, true
}
// Manda un comando al servidor de storage.
// Si hay error de TCP, reintenta en un segundo.
// Espera hasta el OK del servidor.
func sendCommand(query Query) {
    str, success := SendQuery(query)
    for !success {
        time.Sleep(time.Second)
        str, success = SendQuery(query)
    }

    if str != "OK\n" {
        fmt.Println("Unknown message ", str)
    }
}
func UpdateNode(host string, node Node) {
    query := Query{Write, host, node}
    sendCommand(query)
}
func FinishJob(host string) {
    fmt.Println("Job finished for host ", host)
    var _node Node
    query := Query{Finishserver, host, _node}
    sendCommand(query)
}
```

./storage/controller.go

```
package storage
import (
    "fmt"
    "net"
    "bufio"
    "encoding/json"
    "sync"
    "io/ioutil"
)
type persistenceResources struct {
    lock *sync.Mutex
    persistedSize int
}
type persistenceResourcesMap map[string]*persistenceResources
func persistServer(servermap ServerMap, presmap persistenceResourcesMap, host string) {
    resources := presmap[host]
    resources.lock.Lock()
    defer resources.lock.Unlock()
    encoded, _ := json.Marshal(servermap[host].Root_dir)
    err := ioutil.WriteFile(host, encoded, 0644)
    if err != nil {
        panic(err)
    }
    resources.persistedSize = servermap[host].Root_dir.Size
}
```

```
// Process a given query and produces a response to be sent to the
// client (without \n)
func processQuery(query Query, servermap ServerMap, presmap persistenceResourcesMap) string {
    //TODO: queryResponse
    //fmt.Printf("El nodo recibido es el de path %s\n", query.Node.Path)
    switch query.Type {
    case Read:
        fmt.Printf("Es una consulta del host %s sobre el directorio %s\n", query.Hostname, query.Node.Path
        response := GetDir(servermap, query.Hostname, query.Node.Path)
        encoded, _ := json.Marshal(response)
        return fmt.Sprintf("%s",encoded)
    case Write:
        fmt.Printf("Es una escritura del host %s sobre el directorio %s\n", query.Hostname, query.Node.Pat
        AddDir(servermap, query.Hostname, query.Node)
        if servermap[query.Hostname].Root_dir.Size > presmap[query.Hostname].persistedSize {
            persistServer(servermap, presmap, query.Hostname)
        }
    case Newserver:
        if _, ok := servermap[query.Hostname]; ok {
            return "ALREADYEXISTS"
        }
        servermap[query.Hostname] = &Server{
            query.Hostname, false, &Node{Dir, 0, "/", make(NodeMap)}}
        presmap[query.Hostname] = &persistenceResources{&sync.Mutex{}, 0}

    case Finishserver:
        fmt.Printf("Terminando server %s\n", query.Hostname)
        servermap[query.Hostname].Finished = true
        persistServer(servermap, presmap, query.Hostname)
    }
    //encoded, _ := json.Marshal(servermap[query.Hostname].root_dir)
    //fmt.Printf("Ahora el servidor queda como: %s\n", encoded)
    fmt.Printf("El tamaño actual del host %s es %d\n", query.Hostname, servermap[query.Hostname].Root_dir.
    return "OK"
}
func StartServer() (net.Listener, error) {
    //Setup listen socket
    dblisten, err := net.Listen("tcp", ":11000")
    if err != nil {
        fmt.Println(err)
        return nil, err
    }
    fmt.Printf("Listening on port 11000")
    return dblisten, nil
}
func ProcessRequests(dblisten net.Listener) {
    servermap := make(ServerMap)
    presmap := make(persistenceResourcesMap)
    for {
        //fmt.Println("Esperando conexion..")
        conn, err := dblisten.Accept()
        //fmt.Println("[DB] Recibida conexion")
        if err != nil {
            fmt.Println("No se pudo aceptar la conexion: ", err)
            continue
        }
        msg, err := bufio.NewReader(conn).ReadString('\n')

        if err != nil {
            fmt.Println("Problema leyendo mensaje:", err)
            conn.Close()
            continue
        }
        msg = msg[:len(msg)-1] //trailing \n
        // Unserialize message
        var query Query

        err = json.Unmarshal([]byte(msg), &query)
        if err != nil {
            fmt.Println("No se pudo de-serializar el mensaje: ", err)
        } else {
            response := processQuery(query, servermap, presmap)
            send(conn, fmt.Sprintf("%s\n", response))
        }
        fmt.Println("Cerrando conexion.")
        conn.Close()
    }
}
```

./daemon/daemon.go

```
package main

import (
    "fmt"
    "net"
    "time"
    "bufio"
    "encoding/json"
    "strings"
    "../storage"
)
const DBHOSTPORT = "db:11000"
const WORKERHOSTPORT = "worker:50000"
func send(conn net.Conn, s string) {
    fmt.Printf(">%s\n", s)
    buf := []byte(s)
    count := 0
    for count < len(buf) {
        byteSent, err := conn.Write(buf[count:])
        count += byteSent
        if err == nil {
            return
        }
    }
}

// Initializes the DB structure for server host
// if it did not exist. Returns true if server
// was created, false if it already existed.
func createDBAnalysis(host string) bool {
    node := storage.Node{storage.Dir, 0, "/", make(storage.NodeMap)}
    query := storage.Query{storage.Newserver, host, node}
    str, sent := storage.SendQuery(query)
    if !sent {
        time.Sleep(time.Second)
        return createDBAnalysis(host)
    }
    if str == "OK\n" {
        return true
    } else if str == "ALREADYEXISTS\n" {
        return false
    }
    fmt.Println("Unknown message ", str)
    return false
}
func runAnalysis(host string) string {
    conn, err := net.DialTimeout("tcp", WORKERHOSTPORT, time.Second)
    if err != nil {
        if neterr, ok := err.(net.Error); ok && neterr.Timeout()  {
            fmt.Println("Timeout con worker nameserver. Reintentando en 1s.")
            conn, err = net.DialTimeout("tcp", WORKERHOSTPORT, time.Second)
            if err != nil {
                return "TIMEOUT"
            }
        }
        return "Error en conexión al worker nameserver"
    }
    defer conn.Close()
    if !createDBAnalysis(host) {
        return fmt.Sprintf("Ya hay un pedido de analisis para el host %s", host)
    }

    send(conn, fmt.Sprintf("%s /\n", host))
    return "OK"
}
func prettyPrintResults(node storage.Node) string {
    str := ""
    str = str + fmt.Sprintf("%s (%d bytes)\n", node.Path, node.Size)
    for k, v := range node.Files {
        str = str + fmt.Sprintf("\t%s (%s, %d bytes)\n", k, storage.StrNodeType(*v), v.Size)
    }
    return str

}
func getResult(path string, host string) string {
    fmt.Println("Armando resultados")
    node := storage.Node{storage.Dir, 0, path +"/", make(storage.NodeMap)}
    query := storage.Query{storage.Read, host, node}
    str, sent := storage.SendQuery(query)
    if !sent {
        return "No se pudo enviar el comando al servidor"
    }
    //Procesar respuesta
    var response storage.ResultsResponse
    err := json.Unmarshal([]byte(str), &response)
    if err != nil {
        fmt.Printf("Error decoding:%s", err)
    }
    str = prettyPrintResults(response.Node)

    //fmt.Println("Lo que armo pretty print:", str)

    if ! response.Finished {
        return fmt.Sprintf("El host %s no tiene un analisis terminado. El resultado parcial es: \n %s", ho
    }
    return str
}

func processRequest(command string) string {
    fmt.Println("Recibido: %s\n", command)
    if strings.Contains(command, "analyze") {
        host := ""
        fmt.Sscanf(command, "analyze %s", &host)
        return runAnalysis(host) + "\n"
    } else if strings.Contains(command, "summary") {
        host := ""
        path := ""
        fmt.Sscanf(command, "summary %s %s", &host, &path)
        return getResult(path, host) + "\n"
    }
    return fmt.Sprintf("Unknown command received: %s", command)
}
func main() {
    server, err := net.Listen("tcp", ":11001")
    if err != nil {
        fmt.Println(err)
        panic("No se pudo levantar el demonio")
    }
    defer server.Close()
    fmt.Printf("Listening on port 11001")
    for {
        conn, err := server.Accept()
        if err != nil {
            fmt.Println("No fue posible recibir la conexion: ", err)
            continue
        }
        buf := bufio.NewScanner(conn)
        for buf.Scan() {
            send(conn, processRequest(buf.Text()) + "\n")
        }
        conn.Close()
    }



    //crear la estructura en la DB
}
```

./worker/ftp.go

```
package worker
```

```go
import (
    "bufio"
    "fmt"
    "net"
    "strings"
    "time"
)

const (
    noError = iota
    timeoutError
    noListener
)

// Inicia la conexión de control al host destino en puerto 21
// e inicia la escucha en un puerto indicado para la conexion de datos.
// Devuelve la conexión de control, el buffer de control y el escuchante
// en el puerto aleatorio.

func setupFTP(host string, port int) (net.Conn, *bufio.Reader, net.Listener, int) {
    conn, err := net.DialTimeout("tcp", fmt.Sprintf("%s:21", host), time.Second)
    if err != nil {
        fmt.Println(err)
        return nil, nil, nil, timeoutError
    }
    lhost, lport, _ := net.SplitHostPort(conn.LocalAddr().String())
    fmt.Printf("Local address is is %s:%s\n", lhost, lport)
    connbuf := bufio.NewReader(conn)
    for {
        str, _ := connbuf.ReadString('\n')
        if strings.Contains(str, "220") {
            break
        }
    }
    dataserver, err := net.Listen("tcp", fmt.Sprintf(":%d", port))
    if err != nil {
        fmt.Println("No se pudo levantar la conexion de datos:", err)
        conn.Close()
        return nil, nil, nil, noListener
    }
    //time.Sleep(2*time.Second)
    send(conn, "USER anonymous\r\n")
    send(conn, "SYST\r\n")
    for {
        str, err := connbuf.ReadString('\n')
        /*if len(str) > 0 {
            fmt.Printf("<%s", str)
        }*/
        if err != nil {
            fmt.Printf("Error: %s", err)
            continue
        }
        if strings.Contains(str, "230") {
            break
        }
    }
    return conn, connbuf, dataserver, noError
}

// Envía el comando cmd (no es necesario el \n) por la conexión de
// control conn, indicando que mande los datos al puerto port.
// Espera a que el servidor indique que se transfirieron los datos.
func sendFTPCommand(cmd string, conn net.Conn, connbuf *bufio.Reader, port int) {
    first_octet := port / 256
    second_octet := port - first_octet*256

    send(conn, fmt.Sprintf("PORT 127,0,0,1,%d,%d\r\n", first_octet, second_octet))
    for {
        str, err := connbuf.ReadString('\n')
        /*if len(str) > 0 {
            fmt.Printf("<%s", str)
        }*/
        if err != nil {
            fmt.Printf("Error: %s", err)
            return
        }
        if strings.Contains(str, "200") {
            break
        }
    }
    send(conn, fmt.Sprintf("%s\r\n", cmd))
    //Now we wait until the command is complete (226)
    for {
        str, _ := connbuf.ReadString('\n')
        /*if len(str) > 0 {
            fmt.Printf("<%s", str)
        }*/
        if strings.Contains(str, "226") {
            break
        }
        if strings.Contains(str, "425") {
            fmt.Printf("Could not build data connection, retrying in one second.\n")
            time.Sleep(time.Second)
            sendFTPCommand(cmd, conn, connbuf, port)
            return
        }
    }
}
```

./worker/nameserver.go

```
package worker
import (
    "net"
    "fmt"
    "bufio"
)
func ProcessRequests(ns net.Listener, lCh chan string) {
    j := 0
    for {
        j = j + 1
        conn, err := ns.Accept()
        fmt.Println("Recibida conexion nueva #", j)
        if err != nil {
            fmt.Println("No se pudo aceptar la conexion: ", err)
            fmt.Println("Ignorando..")
            continue
        }

        msg, _ := bufio.NewReader(conn).ReadString('\n')
        lCh <- msg[:len(msg)-1] //remove trailing \n
        conn.Close()
    }
}
func NameServer(lCh chan string) (net.Listener, error) {
    //Set up nameserver
    ns, err := net.Listen("tcp", ":50000")
    if err != nil {
        fmt.Println(err)
        return nil, err
    }
    fmt.Printf("Listening on port 50000")

    return ns, nil
}
```

./worker/worker.go

```
package worker

import (
    "fmt"
    "net"
    "strings"
    "math/rand"
    "sync"
    "time"
    "../storage"
)
// Para manejar los recursos que utiliza el analisis de
// un servidor necesitamos un lock. Esto también, en
// la prueba de concepto, permite fijarnos cuándo terminó
// el análisis.
type ServerResources struct {
    nqueued int
    nthreads int
    lock *sync.Mutex
}
type ServerResourcesMap map[string]*ServerResources

const MAXTHREADSANDQUEUED = 3

// Inicia los recursos para un servidor FTP si no estaba ya disponible
// Poscondicion: existe en el mapa la key host, hay al menos un task
// queued.
func initServerResources(host string, dict ServerResourcesMap) {
    if _, ok := dict[host]; !ok {
        dict[host] = &ServerResources{1, 0, &sync.Mutex{}}
    }
}

// Wrap sencillo de write para poder imprimir e ir viendo.
func send(conn net.Conn, s string) {
    //fmt.Printf(">%s\n", s)
    buf := []byte(s)
    count := 0
    for count < len(buf) {
        byteSent, err := conn.Write(buf[count:])
        count += byteSent
        if err == nil {
            return
        }
    }
}

func registerThread(host string, dict ServerResourcesMap) {
    initServerResources(host, dict)
    hostres := dict[host]
    hostres.lock.Lock()
    fmt.Printf("[THREAD] Registering thread!!")
    hostres.nqueued  = hostres.nqueued  - 1
    hostres.nthreads = hostres.nthreads + 1
    hostres.lock.Unlock()
}

func unregisterThread(host string, dict ServerResourcesMap) {
    hostres := dict[host]
    hostres.lock.Lock()
    hostres.nthreads = hostres.nthreads - 1
    fmt.Printf("[THREAD] Unregistering thread - tasks %d threads %d\n", hostres.nqueued, hostres.nthreads)
    if hostres.nqueued == 0 && hostres.nthreads == 0 {
        storage.FinishJob(host)
    }
    hostres.lock.Unlock()
}
func distributeJobs(queue []string, host string, hostres *ServerResources) []string {
    hostres.lock.Lock()
    for i:=0; i < MAXTHREADSANDQUEUED - hostres.nthreads - hostres.nqueued; i++ {
        if len(queue) > 0 {
            subpath := queue[0]
            queue = queue[1:]
            hostres.nqueued = hostres.nqueued + 1
            queued := alsoProcess(fmt.Sprintf("%s %s\n", host, subpath))
            // Si no se puede agregar a la cola general, lo seguimos procesando
            // en este thread.
            if !queued {
                queue = append(queue, subpath)
            }
        }
    }
    hostres.lock.Unlock()
    return queue
}
```

```go
func runAnalysis(host string, path string, idworker int, resourcesmap ServerResourcesMap) {
    registerThread(host, resourcesmap)
    defer unregisterThread(host, resourcesmap)
    hostres := resourcesmap[host]
    port := 9100 + idworker
    conn, connbuf, dataserver, e := setupFTP(host, port)
    if e == timeoutError {
        // Volvemos a agregar a la cola.
        hostres.lock.Lock()
        hostres.nqueued = hostres.nqueued + 1
        requeue := alsoProcess(fmt.Sprintf("%s %s\n", host, path))
        if ! requeue {
            panic("Timeout on FTP server and Timeout on Worker queue")
        }
        hostres.lock.Unlock()
        return
    } else if e == noListener {
        // El problema es el address. Intentamos con otro
        runAnalysis(host, path, idworker + 1000, resourcesmap)
        return
    }

    it := 0
    next_thread_check := 0
    queue := []string{path}
    for len(queue) > 0 {
        cpath := queue[0]
        queue = queue[1:]
        sendFTPCommand(fmt.Sprintf("LIST -AQ %s", cpath), conn, connbuf, port)
        node, new_queue := parseLS(cpath, dataserver)
        storage.UpdateNode(host, node)

        // Skip /proc/ and /sys/ because they don't make sense and can cause special errors
        queue = filterStrlist(append(queue, new_queue...), "/proc/")
        queue = filterStrlist(append(queue, new_queue...), "/sys/")
        //fmt.Printf("Adjusted queue is %s\n", queue)

        // See if we can parallelize
        if it == next_thread_check {
            next_thread_check = it + len(new_queue) // check after cleaning out this level
            queue = distributeJobs(queue, host, hostres)
        }
        it = it + 1
    }

    send(conn, "BYE\r\n")
    conn.Close()
    dataserver.Close()
}
// Adds a job to this worker server's general queue.
// In case of timeout it tries once more.
// Returns false on error.
func alsoProcess(msg string) bool {
    //fmt.Printf("Also process.. -- '%s'", msg)
    conn, err := net.DialTimeout("tcp", ":50000", time.Second)
    if err != nil {
        if neterr, ok := err.(net.Error); ok && neterr.Timeout() {
            fmt.Printf("Timed out: waiting one second and trying again")
            time.Sleep(time.Second)
            conn, err = net.DialTimeout("tcp", ":50000", time.Second)
            return err != nil
        }
        return false
    }
    defer conn.Close()
    send(conn, msg)
    return true
    //fmt.Println("Cerrando alsoprocess")
}

func Worker(i int, linkChan chan string, wg *sync.WaitGroup, resourcesmap ServerResourcesMap) {
    rand.Seed(time.Now().UnixNano()+int64(i)*27)
    defer wg.Done()
    fmt.Println("Started worker ", i)
    for dest := range linkChan {

        parts := strings.Split(dest, " ")
        host := parts[0]
        path := "/"
        if len(parts) == 2 {
            path = parts[1]
        }
        fmt.Printf("[THREAD %d] Recibido trabajo [%s:%s]\n", i, host, path)
        runAnalysis(host, path, i, resourcesmap)
        fmt.Printf("[THREAD %d] Finalizado trabajo\n", i, host, path)
    }
}
```

./worker/parser.go

```
package worker
import (
    "fmt"
    "../storage"
    "net"
    "bufio"
    "strconv"
    "strings"
)

func filterStrlist(list []string, toskip string) []string {
    var filtered []string
    for _, name := range list {
        if name != toskip {
            filtered = append(filtered, name)
        }
    }
    return filtered
}
func filterSpaces(list []string) []string {
    return filterStrlist(list, "")
}
```

```go
func parseLSLine(line string) (string, storage.Node) {
    //fmt.Printf("parse_ls_line: %s\n", line)
    var node storage.Node
    node.Size, _ = strconv.Atoi(filterSpaces(strings.Split(line, " "))[4])
    if line[0] == 'd' {
        node.Type = storage.Dir
    } else if line[0] == 'l' {
        node.Type = storage.Link
    } else {
        node.Type = storage.File
    }
    node.Files = make(storage.NodeMap)
    parts := filterSpaces(strings.Split(line, " ") )
    name := parts[len(parts)-1]
    return name, node
}
// Procesa los resultados de un ls que recibio el escuchante.
// Partiendo desde path, agrega los nodos nuevos
// al árbol que arranca en root_dir y encola los directorios a procesar en
// queue.
func parseLS(path string, dataserver net.Listener) (storage.Node, []string) {
    node := storage.Node{storage.Dir, 0, path, make(storage.NodeMap)}
    queue := make([]string, 0)
    dconn, err := dataserver.Accept()
    if err != nil {
        // handle error
    }
    dconnbuf := bufio.NewScanner(dconn)
    i := 0
    for dconnbuf.Scan() {
        i = i +1
        if i == 1 {
            continue
        }
        _str, _node := parseLSLine(dconnbuf.Text())
        _node.Path = fmt.Sprintf("%s%s/",path,_str)  //TODO if not dir do not add /

        node.Files[_str] = &_node
        node.Size += _node.Size

        if _node.Type == storage.Dir {
            //fmt.Printf("Encolamos %s\n", _node.Path)
            queue = append(queue, _node.Path)
        }
        //fmt.Printf("Path=%s, _str = %s, node = %q\n", path, _str, _node)
        //fmt.Println(_node.nodetype, _str, _node.size)


    }
    return node, queue
}
```

./client/client.go

```go
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
)

func send(conn net.Conn, s string) {
    fmt.Printf(">%s\n", s)
    buf := []byte(s)
    count := 0
    for count < len(buf) {
        byteSent, err := conn.Write(buf[count:])
        count += byteSent
        if err == nil {
            return
        }
    }
}

func main() {
    args := os.Args[1:]
    conn, err := net.Dial("tcp", "daemon:11001")
    if err != nil {
        fmt.Println(err)
        panic("No se pudo conectar con el demonio")
    }
    if args[0] == "analyze" {
        send(conn, fmt.Sprintf("analyze %s\n", args[1]))
    } else if args[0] == "summary" {
        send(conn, fmt.Sprintf("summary %s %s\n", args[1], args[2]))
    } else {
        fmt.Printf("No entendido: %s\n", args)
    }

    dconnbuf := bufio.NewScanner(conn)

    for dconnbuf.Scan() {
        str := dconnbuf.Text()
        fmt.Println(str)
        if len(str) == 0 {
            break
        }
    }
    conn.Close()
}
```