Ministerul Educației al Republicii Moldova
Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică
Departamentul Inginerie Software și Automatică

RAPORT

Lucrarea de laborator nr.1
La disciplina: Programarea aplicațiilor distribuite
Tema: Agent de mesaje

A efectuat: Petrenco Mihai
st.gr.TI-142

A verificat: Pecari Mihail
lect.univ.

Chișinău, 2017

Obiectivele lucrării:

- Studiul agenților de mesagerie;
- Elaborarea unui protocol de comunicare al agentului de mesaje;
- Tratarea concurentă a mesajelor;
- Alegerea protocolului de transport;
- Alerea și elaborarea strategiei de păstrare a mesajelor;

Modul de lucru:

În cadrul lucrării de laborator a fost implementat un agent de mesaje scris în limbajul Swift pentru platforma OSX. Agentul de mesaje utilizează protocolul TCP pentru transmiterea datelor, serializate sub format JSON.

În figura 1 este prezentat un exemplu al conectării serverului cu un client și ascultarea datelor de către server.

```
let server = TCPServer(address: "127.0.0.1", port: 9999)
switch server.listen() {
case .success:
    let client = server.accept()
    while true {
        if let _ = client?.read(1024) {
            //Process received bytes...
        }
    }
case .failure(let error):
    print(error)
}
```

Protocolul de comunicare utilizat pentru transfer de date este descris pe repozitoriul personal. Clientul generează un dicționar JSON, îl convertește într-un obiect de tip Data și îl transmite către server prin metoda client.write(data:Data).

Obiectul dat este recepționat de către server în mod asincron și prelucrat conform comenzii trimise. Concurența în aplicație este implementată utilizînd Grand Central Dispatch.

Conținutul obiectului JSON constă din:
- comanda respectivă;
- un set de cozi ce sunt afectate de comanda dată;
- un set de mesaje ce urmează a fi trimise către sau de la server;

Toată logica legată de managementul cozilor se află în clasa QueueStore. Clasa respectivă conține un array de tupluri. Fiecare tuplu din array conține numele cozii, subscriberii cozii respective și mesajele sale.

```
private(set) static var queues = [(name: "Main", subscribers: [TCPClient](), contents: [String]())]
```

Fiecare coadă din tabloul de tupluri este adăugată și eliminată dinamic. Astfel, odată ce un client apelează comanda PUT pentru un tablou inexistent, el automat este creat, iar mesajul este inserat.

Analogic, odată cu eliminarea tuturor elementelor din coadă, ea este distrusă. Metoda de inserare în tablou este prezentată astfel:

```
//This adds a new element to the queue or creates a new queue
public static func add(contents: String, in queue: String) {

    for index in 0..<queues.count {
        if queues[index].name == queue {
            queues[index].contents.append(contents)
            return
        }
    }
    queues.append((name: queue,subscribers: [TCPClient](), contents: [contents]))
}
```

În caz că serverul își oprește funcționalitatea, datele sale sunt salvate într-un fișier JSON în memoria calculatorului. Odată cu restartarea serverului, ele sunt încărcate din JSON și cozile sunt restabilite.

```
//Converts the Message Queue into a valid JSON Data object
static func convert(queue: [(name: String, subscribers: [TCPClient], contents: [String])]) -> Data? {
    var array = [[[String:Any]]]()
    for element in queue {
        let name = ["Name":element.name]
        let contents = ["Contents":element.contents]
        let collection: [[String:Any]] = [name,contents]
        array.append(collection)
    }

    do {
        let data = try JSONSerialization.data(withJSONObject: array, options: .prettyPrinted)
        return data
    } catch {
        print("Could not generate Data.")
        return nil
    }
}
```

```
//Saves the Message Queues as a JSON object
public static func save() {
    let desktopDir = FileManager.default.urls(for: .desktopDirectory, in: .userDomainMask)
    let fileDir = desktopDir.first!.appendingPathComponent("MessageQueue.json")

    guard let data = Serializer.convert(queue: QueueStore.queues) else {
        print("Could not obtain convert queue to Data")
        return
    }

    do {
        try data.write(to: fileDir)
    } catch {
        print("Could not save JSON to file")
    }
}
```

Agentul de mesaje implementează patternul Publisher-Subscriber. Astfel, un client se poate abona la o coadă anumită, utilizînd metoda Subscribe, și toate mesajele trimise către coadă prin intermediul comenzii PUT vor fi automat trimise către clientul respectiv, fără a fi nevoie de utilizat explicit comanda GET.

```
//Function executed when SUBSCRIBE command is selected
func subscribeFunctionality(from client: TCPClient, at queues: [String]) {
    for queue in queues {
        QueueStore.addSubscriber(for: queue, client: client)
    }
}

//Subscriber functionalities
func checkSubscriber(at queue: String) {
    for element in QueueStore.queues {
        if element.name == queue {
            for subscriber in element.subscribers {
                let json = Serializer.generate(with: "RESPONSE", and: element.contents.last!)
                let data = Serializer.toData(from: json)
                let _ = subscriber.send(data: data!)
            }
        }
    }
}
```

Astfel, de fiecare dată cînd serverul procesează o comandă de tip PUT, el apelează metoda checkSubscriber(), care transmite conținutul ultimului mesaj către toți abonații cozii.

Rutarea avansată în sistemul respectiv constă în faptul că clientul poate trimite, sau obține date, precum și abona, la mai multe cozi simultan. Acest mecanism a fost implementat prin faptul că la momentul generării JSON-ului, în loc ca să stabilim numele cozii, acum implementăm un tablou de tip String pentru fiecare coadă.

Astfel, comanda trimisă va fi procesată asupra fiecărei cozi din tabloul respectiv de cozi.

```
//Decomposing the JSON dictionary into variables
func process(json: [String:Any], from client: TCPClient, having color: NSColor) {

    guard let command = json["Command"] as? String else {
        return
    }
    guard let queues = json["Queues"] as? [String] else {
        return
    }
    guard let content = json["Content"] as? String else {
        return
    }

    switch getCommand(from: command) {
    case .put:
        self.putFunctionality(from: client, with: content, in: queues)
        self.notifyActivity(from: client, having: color, with: "Added message to queues.")
    case .count:
        self.countFunctionality(from: client, at: queues)
        self.notifyActivity(from: client, having: color, with: "Asked for queue count.")
    case .remove:
        self.removeFunctionality(from: client, at: queues)
        self.notifyActivity(from: client, having: color, with: "Removed message from queue.")
    case .get:
        self.getFunctionality(from: client, at: queues)
        self.notifyActivity(from: client, having: color, with: "Asked for queue contents.")
    case .subscribe:
        self.subscribeFunctionality(from: client, at: queues)
        self.notifyActivity(from: client, having: color, with: "Subscribed to queues.")
    default:
        print("Unknown shit")
    }
}
```

Concluzie

În cadrul lucrării de laborator au fost studiate principiile de funcționare a unui agent de mesaje și modul de funcționare a unui sistem distribuit. Utilizînd protocolul de transport ales, împreună cu un protocol de transport creat în cadrul lucrării, am implementat o conexiune client-server și am asigurat o comunicare asincronă dintre membri.

Asupra sistemului respectiv au fost implementate anumite funcționalități, precum serializarea și salvarea datelor în memoria secundară, implementarea unui mecanism de rutare a cozii, dezvoltarea unui pattern de tip publisher-subscriber și încercarea de a implementa mecanismul de rutare avansată a mesajelor.

Laboratorul respectiv a oferit cunoștințe practice necesare pentru dezvoltarea și implementarea aplicațiilor distribuite.