Technical University of Moldova
Faculty of Computers and Information Technologies
Specialty Computer Science

Report

Laboratory #3

Distributed Data Collections

Author:
Cristian Cartofeanu

Lecturer:
Dumitru Ciorbă

November 17, 2015

Contents

1 Objective

The goal of the laboratory study lies in the transport protocols TCP / IP in the development of distributed applications containing data collections.

The primary objectives:

- Use of UDP protocol in unicast and multicast transmission
- Use of TCP protocol in data transmission
- Processing collections of objects

2 UDP/TCP protocols

UDP is the User Datagram Protocol. It is different from TCP (Transmission Control Protocol) in that it does not establish a connection to the destination. UDP is designed to send datagrams. Datagrams can be though of as discrete blocks of data or messages with limited overhead. UDP does not guarantee that the datagrams will be delivered in any specific order or even at all! So you might be asking, Why do we use them if they are not guaranteed to arrive at their destination?. Good question! They are very useful for certain types of data that does not need 100 reliability, and therefore it does not need the overhead that TCP imposes.

So what kind of data does not need to be reliably delivered? In our situation, we are running a server that receives node updates from several nodes across the network once every seconds. Thats a relatively big amount of data (for the model purpose; in reality is very small amount), but more importantly each new message from a single node makes the previous message obsolete. The idea behind it is that the client sends a request to all of the nodes that are interested in the connection (obviously by interested I mean they joined the multicast address), in which it is specified a type of message which expects an answer. Now, the prototype of the message would be return mavenNode, where mavenNode is the node that is the most influential. The coined term influential implies two assumptions:

a. A node is considered more influential than another node if it has more neighbours than the other one. A neighbour is considered any adjacent node that is directly linked to the respective node (all the edges that are connected to a vertex in a graph, if viewed from a graph perspective). In this graph, the most influential node is the node A since the vertex A has three direct edges that join it with the vertices E, C and B.

b. A node is considered more influential than another node if it satisfies condition number one and also if it contains more information than the other candidate mavenNode (in case both the current node and the mavenNode candidate have the same number of neighbours).

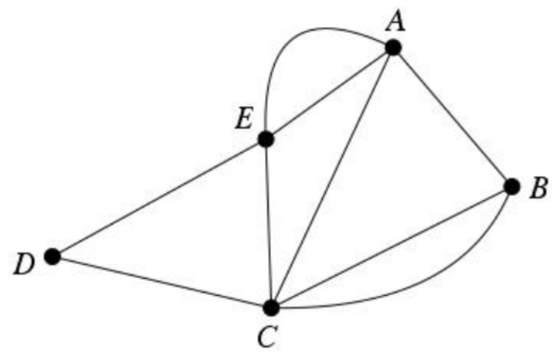

Figure 1: Example of a graph

In this analysed model the fact that a few messages are lost its not problematic, because the client sends a message at a specified interval of time to all the nodes in a multicast connection, querying what node is the knowledge accumulator. The result is overwritten at every step if some changes occur and if not, the lost data is to be recovered as soon as a new query starts in the set interval.

Examples of services that are transmitted over UDP include streaming video, voice over IP phone calls, DHCP, and multi-player online games.

2.1 Use of UDP protocol in unicast transmission

2.1.1 Close look at a UDP Datagram

A user datagram has a fixed 8 byte header. The header is very simple and contains four, 16 bit fields. The fields are the source port, destination port, total length, and a checksum. Lets take a look at the struct behind it.

```
typedef struct {
    uint16_t sourcePort;
    uint16_t destPort;
    uint16_t length;
    uint16_t checksum;
} UDPHeader_t;
```

2.1.2 UDP Receiver prototype

Alright, now that we know how UDP works, lets check out how to set up a UDP client to start receiving UDP packets. To get started we are going to use the dgram class in NodeJS. One particular thing to notice is that it doesnt matter whether one is sending or receiving datagrams, you must still bind the socket. This is because datagram sockets are connectionless. Every sender is also a receiver by default and can receive messages.

```
var clientUDP = dgram.createSocket( udp4");
var SRC_PORT = 6025;

clientUDP.bind(SRC_PORT, function () {
    setInterval(multicastNew, 4000);
    console.log("datagram socket ready to receive");
});

clientUDP.on('listening', function () {
    var address = clientUDP.address();
    console.log('UDP Client listening on ' + address.address + ":" + address.por
});

clientUDP.on('message', function (message, rinfo) {
    console.log('Message from: ' + rinfo.address + ':' + rinfo.port + ' - ' + me
});
```

We start out by calling the dgram.bind() method to bind the socket to port SRC_PORT on any available IPv4 Ethernet devices. Once the socket has been bound we can listen for any incoming datagram packets by registering an onMessage callback after activating the onListen event handler. One thing to note is that the onMessage callback gives us a msg buffer object. To read the actual datagram we must call the message object (with some optional attributes such as length for instance). The Datagram object returned contains the InternetAddress object rinfo with the details of the sender, and the data that was sent.

2.1.3 UDP Sender prototype

To create a program that sends datagrams, we basically do the same thing as before. We need to bind() the socket, then call the socket.send(buf, offset, length, port, address[, callback]) method. The send method takes the buffer object as expected, also it takes the destination address and port directly. The address and port parameters tell the datagram where to go. Remember that this is a connectionless protocol so each time we want to send data we need to provide a destination.

```
var PORT = 3000;

function multicastNew() {
    var messageClient = new Buffer ('someMessage');
    clientUDP.send(messageClient, 0, messageClient.length, PORT , MULTICAST_ADDR
```

```
    });
}

clientUDP.bind(SRC_PORT, function () {
    setInterval(multicastNew, 4000);
    console.log("datagram socket ready to send");
});
```

Notice that the bind() method call takes PORT as the port number (the remote port to which the message will be sent). Also take note that I am setting the destination to MULTICAST_ADDR. This causes the datagram to be sent to the ip address that is reserved for multicast connections and is in range from 224.0.0.0 to 239.255.255.255. To test this out we can fire up the client program.

2.2 Use of UDP protocol in multicast transmission

2.2.1 Multicast

Since UDP is a connectionless protocol, a single UDP socket can be used to send and receive data. The bind() call establishes what port and address we can receive data on, and the send() call allows us to send data to anywhere we want. We can easily make a UDP server resembling a node in a peer-to-peer network by combining the two.

```
var node = dgram.createSocket({ type: 'udp4', reuseAddr: true });
var PORT = 3000;
var SRC_PORT = 6025;
var MULTICAST_ADDR = '239.255.255.250';

node.on('listening', function () {
    var address = node.address();
    console.log('UDP node listening on ' + address.address + ":" + address.port)
});

node.on('message', function (messageClient, rinfo) {
    console.log('Multicast message from: ' + rinfo.address + ':' +
    if (messageClient=='someMessage') {
        //logic processed here
          // var result; for storing the result of the operation
        node.send(result, 0, result.length, rinfo.port ,
            console.log("Sent '" + result + "'");
        });
    };
});
node.bind(PORT, function () {
    node.addMembership(MULTICAST_ADDR);
});
```

In the above example, each time a Datagram is received, it is checked whether the message from the sender (client) is of a specific format and based on some logic operations, a result is sent back to the sender. The Datagram object carries the source InternetAddress, and source port that we can use in the send() method to return the message.

Multicasting opens us up to have a single source and multiple destinations. This is very convenient for certain applications like streaming media, or in the laboratory model, the distribution of a collection of JSON objects. The source program(client) sends datagram packets to a multicast group address (239.255.255.250). Each interested client(node) then joins the multicast group and can receive the datagrams being sent.
The source that is sending the multicast datagrams has an easy time, all that is necessary is to send the packets to a multicast group address instead of a normal destination address. Multicast addresses are in the range of 224.0.0.0/4. That is all IP address from 224.0.0.0 to 239.255.255.255.

To receive multicast content, extra steps must be taken to join the multicast group that you want to receive packets from. To join a multicast group you must issue a join command. To do this in NodeJS, you can use the socket.addMembership(multicastAddress[, multicastInterface]) method. Another step that must be taken is to add a multicast route to your local routing table. Most people forget this step and cant figure out why they are not receiving any packets. On OSX you can add the appropriate route with the command sudo route -nv add -net 228.0.0.4 -interface en0. Of course you should replace en0 with whatever your device is actually called (on OS X my primary nic device is en1).

2.3 Use of TCP protocol in data transmission

The use of the tcp protocol is dictated by the need of sending some relevant data over the network. For a better understanding, the following picture illustrates the workflow of the program.
The maven node is, as described in section UDP/TCP protocols, defined based on two assumptions. For checking those assumptions each node performs some calculations and based on the result, which is the number of neighbours (adjacent nodes to the examined node) and also on the data it holds.

In the Fig. 2 we see two highlighted nodes that are potential candidates for being the knowledge accumulator (MAVEN).

In our application we use the following algorithm for choosing Maven node:

```
// Choosing the maven node by some criteria(>neighbours && >employees)
   if (neighbours >= mavenNeighbours ) {
```

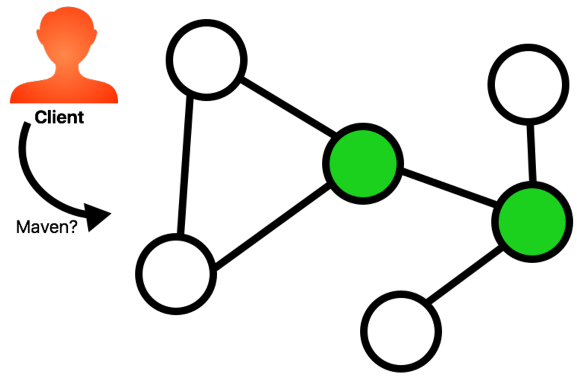

Figure 2: Client request Maven node

```
        if (employees >= mavenEmployees) {
            mavenHost = host;
            mavenNeighbours = neighbours;
            mavenEmployees = employees;
        };
    };
```

Now, keep in mind that each node stores a json object inside and based on how many elements the json object contains (prototype version), two or more candidates may knock out each other and claim the maven property.

As soon as the mavenNode is selected based on the specified criteria, it sends the information back to the client, based on the message asked from the client. In our case, the knowledge accumulator sends the number of neighbours and the data it possesses to the client via Unicast. Consequently, the client receives the answer and after that it initiates a TCP request to the mavenNode

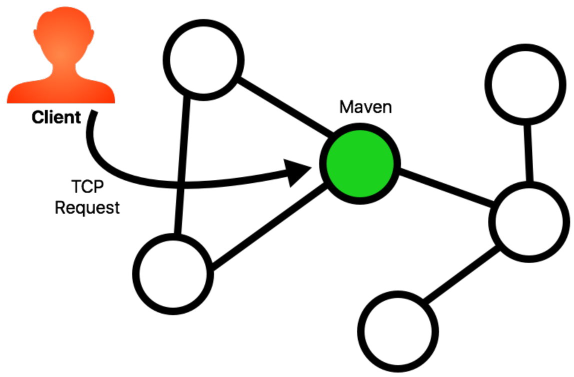

Figure 3: Client initiate TCP to maven node

The TCP request is based on the TCP socket connection, which is initialised in the following way:

```
var socket = new JsonSocket(new net.Socket()); //Decorate a standard net.Socket
socket.connect(PORT, mavenHost);
socket.on('connect', function() {
      socket.sendMessage({command: 'RequestData'});
      socket.on('message', function(RequestData) {
                 //RequestData logic done on node side
           }
         });
});
```

The process of sending data from the mavenNode to the client is presented below:

```
var server = net.createServer();
server.listen(PORT);
server.on('connection', function(socket) {
      socket = new JsonSocket(socket);
      socket.on('message', function(message) {
                if (message.command == 'RequestData') {
                         //RequestData logic handled here
                         }
                 } else if (message.command == 'otherCommand') {
                         //otherCommand logic goes here
                 }
         });
});
```

3 Processing collections of objects

For proccesing the collection of object we have used a JavaScript library (Underscore). Underscore.js is a utility-belt library for JavaScript that provides support for the usual functional suspects (each, map, reduce, filter...) without extending any core JavaScript objects.

We used three method to process our collection:

- Group
- Filter
- Sort

1) In our example we have grouped the collection by the departament value criteria.

groupBy_.groupBy Splits a collection into sets, grouped by the result of running each value through iteratee. If iteratee is a string instead of a function, groups by the property named by iteratee on each of the values.

```
var filterDepartament = _.groupBy(collection,
        function(value){
        return value.departament
    });
```

2) While filtering we choosed the criteria to be the salary value.

filter_.filter(list, predicate, [context]) Alias: select Looks through each value in the list, returning an array of all the values that pass a truth test (predicate).

```
var filterSalary = _.filter(filterDepartament[filt_depart],
        function(num){ return num.salary > filt_sal; });
```

3) And the last method of processing our collection is sorting. We sort the collection by lastName value.

sortBy_.sortBy(list, iteratee, [context]) Returns a (stable) sorted copy of list, ranked in ascending order by the results of running each value through iteratee. iteratee may also be the string name of the property to sort by (eg. length).

```
var filterName = _.sortBy(filterSalary, 'lastName');
```

4 Implementation description

In order to run our application we will need to install several tools for that:

- NodeJS as a JavaScript environment.
- Node.js modules: net; json-socket; underscore; fs; dgram.
- To run more nodes(servers) on once machine we need to set up more local host addresses on lo0 interface. For this we need to write the following shell command in our terminal (sudo ifconfig lo0 alias 127.0.0.N 255.255.255.0), where instead of N any number between 0-254.

After the installation of the JavaScript environment, modules and setting up more localhost addresses we are ready to go. First we need to start our node.js several times to simulate our graphs nodes. Starting our node.js requires to give some parameters (HOST, PORT, Neighbours HOST, data_file.json).

Example of the command to start one node together with parameters (node node.js 127.0.0.2 3000 '["127.0.0.5", "127.0.0.3"]' data_node_2.json).
After starting several nodes that have logical settings between them we can run our client.js. The purpose of the client in this application is to find the MAVEN node and further to communicate with it using TCP for requesting it's won data plus his neighbours once.

MAVEN node is the node that have the biggest number of neighbours and also the biggest number of data.

After starting the client.js the client sends a UDP multicast message(command) asking for some information(what is the nodes address that listening on multicast address, how many nodes neighbours they have and how many data they have as well) from all nodes. Each node sends back using UDP unicast protocol their address, number of neighbours and number of data they have.

After the client receives the answers from all nodes it process them for finding the MAVEN node.

After the MAVEN node was found our client sends a TCP message.command to him(MAVEN) asking for the information they have plus its neighbours one. The MAVEN node starts collecting the information from it's neighbours and append all to it's own. After it finished to collect data from it's neighbours, it send's back using TCP protocol to the client.

After the client receives the data from the MAVEN node and it's neighbours once, the client start to filter, sort and group it with the specific criteria.

After filtering, sorting and grouping client saves the result to a file.

Here is an example
We have a graph with 4 nodes:

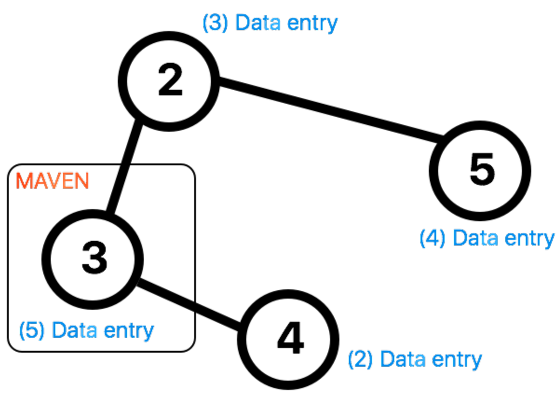

Figure 4: Example implementation on the following graph

Each node have some data that is stored in data files (data_node_2.json, data_node_3.json,...,data_node_5.json).

Using our logic we understand that the MAVEN node is node number 3 because it have the biggest number of neighbours and the biggest data (data entry).

Once we have imagined our graph we can start to create it.

First we need to create 4 localhost address on lo0 interface. For this we can write the following command in our terminal: sudo ifconfig lo0 alias 127.0.0.2 255.255.255.0 sudo ifconfig lo0 alias 127.0.0.3 255.255.255.0 sudo ifconfig lo0 alias 127.0.0.4 255.255.255.0 sudo ifconfig lo0 alias 127.0.0.5 255.255.255.0

After we created our virtual localhost address we run 4 times our node.js file with the logical parameters: (node node.js HOST, PORT, Neighbours HOST, data_file.json).

2) node node.js 127.0.0.2 3000 '["127.0.0.5", "127.0.0.3"]' data_node_2.json

```
Host: 127.0.0.2
Port: 3000
Neighbours: 127.0.0.5,127.0.0.3
Data file to read from: data_node_2.json
TCP Node is listening on 127.0.0.2:3000
UDP Node is listening on 0.0.0.0:3000
```

3) node node.js 127.0.0.3 3000 '["127.0.0.2", "127.0.0.4"]' data_node_3.json

```
Host: 127.0.0.3
Port: 3000
Neighbours: 127.0.0.2,127.0.0.4
Data file to read from: data_node_3.json
TCP Node is listening on 127.0.0.3:3000
UDP Node is listening on 0.0.0.0:3000
```

4) node node.js 127.0.0.4 3000 '["127.0.0.3"]' data_node_4.json

```
Host: 127.0.0.4
Port: 3000
Neighbours: 127.0.0.3
Data file to read from: data_node_4.json
TCP Node is listening on 127.0.0.4:3000
UDP Node is listening on 0.0.0.0:3000
```

5) node node.js 127.0.0.5 3000 '["127.0.0.2"]' data_node_5.json

```
Host: 127.0.0.5
Port: 3000
Neighbours: 127.0.0.2
Data file to read from: data_node_5.json
TCP Node is listening on 127.0.0.5:3000
UDP Node is listening on 0.0.0.0:3000
```

Once we have created our graph and settled up, we can run our client.js by the command node client.js

```
Sending multicast the command "host & neighbours & employers"
   to all nodes that are listening on multicast address "239.255.255.250:3000"
UDP Client listening on 0.0.0.0:6025 (multicast address).

----------------------------Answeres from the nodes----------------------------
--------------Host: 127.0.0.4:3000 has: _1_ neighbours and _2_ employees
Curent Maven Host: 127.0.0.4:3000 has: _1_ neighbours and _2_ employees
------------------------------------------------------------------------------
--------------Host: 127.0.0.5:3000 has: _1_ neighbours and _4_ employees
Curent Maven Host: 127.0.0.5:3000 has: _1_ neighbours and _4_ employees
------------------------------------------------------------------------------
--------------Host: 127.0.0.3:3000 has: _2_ neighbours and _5_ employees
Curent Maven Host: 127.0.0.3:3000 has: _2_ neighbours and _5_ employees
------------------------------------------------------------------------------
--------------Host: 127.0.0.2:3000 has: _2_ neighbours and _3_ employees
Curent Maven Host: 127.0.0.3:3000 has: _2_ neighbours and _5_ employees
------------------------------------------------------------------------------
Addres of our maven is: 127.0.0.3:3000 (2 neighbours and 5 employees)
```

```
Data recived: {JSON}

Filter result: {JSON}
```

```
Filtered result was saved in data.txt in the current directory!
```

As we see the client.js found the correct MAVEN (node 3) that have the address 127.0.0.3 and further collected the data that was filtered and saved.

5 Conclusion

During this laboratory work we have learned bout UDP/TCP protocol in detail. We have used a UDP unicast and multicast transmmition protocol together with the TCP. Also we processed the collection of data that was manipulated under some specific condition.

6 References

[1] UDP Socket Programming with Dart (Unicast and Multicast)

[2] Casciaro Mario, Node.js Design Patterns, 2014, Packt Publishing, ISBN-13: 978-1783287314

[3] Rohit Rai., Socket.IO Real-time Web Application Development, 2013, Packt Publishing, ISBN: 978-1-78216-078-6, p. 50

[4] https://github.com/CristianChris/processing-of-distributed-collections-of-data