

## Programming Assignment 3: Multiple Clients/Servers II

### Due: March 31, noon

In this lab, you will improve on the multiple client-server room reservation solution from Assignment 2. Again, you can develop Assignment 3 from scratch, base it on your solution to Assignment 2, or use the published sample solution (however, as before, I won't be answering questions about the implementation, as with many other "free" software, the rule is **"use at your own risk"**). The key changes in this assignment are to improve the performance in two ways:

- (1) Introduce client-side caching
- (2) Have the set of servers elect a leader, who will be the only server to actively respond to user requests

Similar to Assignment 2, each server needs to be able to handle multiple clients (including delaying/sleeping a while after the reply to model the effect of heavy compute load and long network latency) and a client may interact with a number of fully replicated servers.

### Caching Clients

So far, a client makes a request to the server for every command. In networks/scenarios where the client is potentially far away from a server, this may introduce high latencies. Many distributed applications use caching to overcome that problem. In essence, caching means that results returned from a server are stored closer to (or, in the extreme, at) the client. If a client repeats a request, the reply is available locally and can be returned much faster. Of course, such a scheme will have to worry about changes to the data on the servers (which will invalidate cached data). The textbook discusses the motivation for caching in the WWW in Chapter 2, more specifically Section 2.2.5 in the 7<sup>th</sup> edition of the textbook. Devise and implement a caching scheme for the room reservation application and implement it in the client.

### Server Groups with Leader Election

In the Assignment 2 solution, each server replies to a client request, which is clearly suboptimal. Among other issues, we had to devise a (clever?) way to suppress multiple answers. This was done because the servers do not know about each other's existence. In this assignment, the group of servers should

- Elect a group leader, who will reply on behalf of the collection of servers to client requests. A number of different leader election algorithms exist, for a survey see <http://www2.cs.uregina.ca/~hamilton/courses/330/notes/distributed/distributed.html>. The sample solution implements a simplified version of the Bully Election Algorithm but you are welcome to choose another approach or to design your own solution as well. Your solution should select the process with the highest unique ID (see below) as group leader.
- Monitor the group leader to ensure it has not crashed/disappeared. If this were to happen, a new leader needs to be elected. Such a mechanism is usually termed a "failure detector" or "heartbeat message". You may want to explore some possible strategies doing a search for

**Assignment 3 Handout**

terms such as “distributed systems failure detector” or “distributed systems heartbeat mechanism”.

As the servers need to be able to take over as group leader at any given moment, they all still need to receive all client requests and process them, yet only one (the chosen leader) will send the reply. Similar to Assignment 2, we will assume that IP multicasting in a LAN environment is reliable and ordered. Also, you can assume that the multicast is synchronous (i.e., an upper bound on message latency exists). That implies that if you send a message to a server and do not get a reply within, say, 1 second, you can safely assume that the server has crashed. In general, in the Internet, such an assumption would not be valid: we do not know how long it takes to receive a reply, nor do we know whether a lost message indicates that the server is down or the network congested. However, again similar to Assignment 2, your solution should not assume that a server knows how many other servers are up and running.

To distinguish different instances of a server, they need to have a unique identifier. Often, this identifier would be derived from the IP address and Process ID (PID) of the host a server is running on. In Python, you can retrieve a processes PID via `os.getpid()` (requires to `import os`). We are most likely testing the submission on a single host, so assume that the PID is sufficient to uniquely identify each server. To test various scenarios, it will be handy to allocate an ID to a specific server (either a very low one or a very high one). So your server implementation should accept an optional third command-line. If the user specifies this parameter, it replaces the PID as unique server ID.

## Testing Considerations

### 1) Client:

- When replying to user requests, make it visible whether the reply is based on data returned from the server(s) or serviced from a client-side cache.
- The client should report an error when receiving more than one reply to a client request.

### 2) Server:

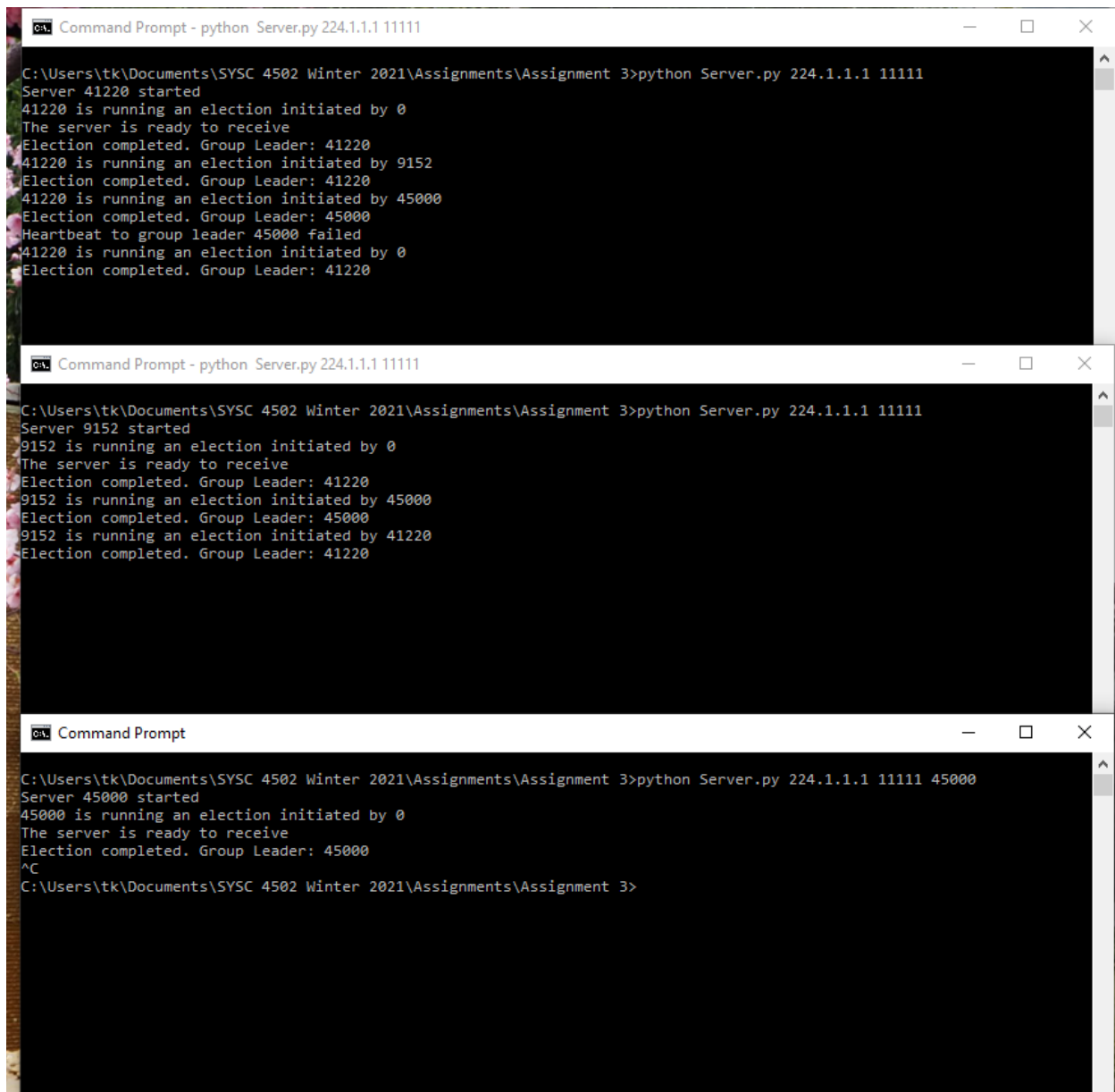
- Each server should indicate when it runs a leader election algorithm and what the outcome of the election is.
- Each server should indicate whether a heartbeat message to its group leader failed.
- Test the server implementation with at least the following scenarios:
  - a. A new server starts up with an ID less than the highest ID currently in use. In this case, the (new) server may start a leader election, but will learn that it is not the group leader.
  - b. A new server starts up with the highest ID currently in use. This is the common case when starting a server later than other servers. In this case, the current group leader will have to be replaced by the new server (as the rule was that the process with the highest ID is group leader).
  - c. Kill the current group leader. In this case, at least one other server should notice that the group leader has failed, triggering a new election.

One error scenario you may also wish to consider, but it is not a requirement, is that case where no server is currently up and running. What would happen to the client process in that case?

## Sample Scenario

I created two sets of screenshots with my sample solution. The first shows the leader election and heartbeat messages. The second shows the caching on the client side.

Server-side output: The screenshot below shows a scenario with three servers and the messages your sample solution should produce (the format is not important, as long as the key content is there, i.e., we can deduce from the output what happened in terms of elections and monitoring the group leader):



```
Command Prompt - python Server.py 224.1.1.1 11111
C:\Users\tk\Documents\SYSC 4502 Winter 2021\Assignments\Assignment 3>python Server.py 224.1.1.1 11111
Server 41220 started
41220 is running an election initiated by 0
The server is ready to receive
Election completed. Group Leader: 41220
41220 is running an election initiated by 9152
Election completed. Group Leader: 41220
41220 is running an election initiated by 45000
Election completed. Group Leader: 45000
Heartbeat to group leader 45000 failed
41220 is running an election initiated by 0
Election completed. Group Leader: 41220

Command Prompt - python Server.py 224.1.1.1 11111
C:\Users\tk\Documents\SYSC 4502 Winter 2021\Assignments\Assignment 3>python Server.py 224.1.1.1 11111
Server 9152 started
9152 is running an election initiated by 0
The server is ready to receive
Election completed. Group Leader: 41220
9152 is running an election initiated by 45000
Election completed. Group Leader: 45000
9152 is running an election initiated by 41220
Election completed. Group Leader: 41220

Command Prompt
C:\Users\tk\Documents\SYSC 4502 Winter 2021\Assignments\Assignment 3>python Server.py 224.1.1.1 11111 45000
Server 45000 started
45000 is running an election initiated by 0
The server is ready to receive
Election completed. Group Leader: 45000
^C
C:\Users\tk\Documents\SYSC 4502 Winter 2021\Assignments\Assignment 3>
```

---

### **Assignment 3 Handout**

In this scenario, I first started the server (PID 41120) in the top window. It ran an election (initiated by 0 is a way to indicate that this the initial election, for example, when starting up) and elected itself as group leader. I next started the server in the middle window (PID 9152). At startup, it initiated an election, which involved itself and server 41120 (who indicates that it ran an election initiated by 9152). Both terminate with electing 41120 as group leader. Next I started the server in the bottom window, assigning it a PID that was larger than the two previous ones (PID 45000). It ran an election at startup, causing the other two servers to run an election initiated by 45000. As a result, all three server instances elect 45000 as group leader.

After some time, I terminated the group leader (bottom window). Server 41120 noticed this first, with its heartbeat message failing, and initiated a new election. Server 9152 receives that election message and likewise starts an election. At the end, both servers agree on Server 41120 being the new group leader.

Client-side output: the screenshot below shows the interaction of a single client with the set of servers. There are no errors, some of the responses are cached, some are derived from contacting the group of servers. Note that, initially, no information is cached, so initially all requests have to be sent to the server(s).

# Carleton University

Department of Systems and Computer Engineering

**SYSC 4502**

**Communications Software**

**Winter 2022**

## Assignment 3 Handout

```
Command Prompt
C:\Users\tk\Documents\SYSC 4502 Winter 2021\Assignments\Assignment 3>python Client.py 224.1.1.1 11111
Next command: rooms
(From servers): SA318
ME4494
SA314
MC5050
ME4326
SA412
Next command: timeslots
(From servers): 8:30-9:30
9:30-10:30
10:30-11:30
11:30-12:30
12:30-13:30
13:30-14:30
14:30-15:30
15:30-16:30
16:30-17:30
17:30-19:00
19:30-21:00
Next command: check ME4326
(From servers): No room reservation found
Next command: rooms
(From cache): SA318
ME4494
SA314
MC5050
ME4326
SA412
Next command: days
(From servers): Monday
Tuesday
Wednesday
Thursday
Friday
Next command: reserve ME4326 8:30-9:30 Friday
(From servers): Reservation confirmed
Next command: check ME4326
(From servers): ME4326 8:30-9:30 Friday
Next command: days
(From cache): Monday
Tuesday
Wednesday
Thursday
Friday
Next command: quit
(From servers): bye
C:\Users\tk\Documents\SYSC 4502 Winter 2021\Assignments\Assignment 3>
```

## Submission Requirements

Submit your solution using Brightspace. Do not submit any archive (i.e., ZIP file or RAR), instead submit individual files. Your solution should, at the very least, provide an implementation of the client and the server. Your programs should run as is in the default lab environment (i.e., with Python 3.10.1), and the code should be well documented. Marks will be based on:

- Completeness of your submission
- Correct solution to the problem
- Following good coding style
- Sufficient and high-quality in-line comments
- Adhering to the submission requirements

The due date is based on the time of the Brightspace server and will be strictly enforced, as with prior assignments. If you are concerned about missing the deadline, you can always submit a (partial) solution early, and resubmit an improved solution later. This way, you will reduce the risk of missing the submission deadline completely.

Submit, with your files, a document (`Readme.doc`, `Readme.docx`, or `Readme.pdf`) that briefly describes the following aspects of your solution:

- (1) What caching strategy did you choose and implement on the client side
- (2) What leader election strategy/algorithm did you implement
- (3) How do non-leaders monitor the continued health of the group leader
- (4) How to start up both clients and server (including command-line parameters and their meaning).