COMPX234-A3: Assignment 3 specification

Integrity. You will be using **GitLab, GitHub or Gitee** to show evidence of the step-by-step development and demonstrate you understand the work.

You are **not allowed to** use LLM prompting for code generation. You must explain clearly each step of your design and implementation, with the steps documented via commits in GitLab/GitHub/Gitee. When the evolution is not demonstrated via the commit history, the student may get 0 marks for the assignment and/or be called for a viva voce to explain the code. Also, practical knowledge of the implementation needs to be demonstrated in the Final Exam. Any suspected cheating cases will be passed to the Student Disciplinary Committee for investigation. We acknowledge that code generation is a very powerful tool, but at this moment, students need to be able to develop their work without that kind of assistance. (For the record, we also generated versions of this assignment using different LLMs and we will be vigilant for signs of LLM usage.) Instead, get help from humans: **attend the labs** and ask the demonstrators who are there to assist your work on the assignment.

Dates and submission. The specification is being released on 16 April 2025. The submission deadline is 2nd May, 2025, which corresponds to two weeks work (and 24 hours of practical work in total). The submissions will be taken from GitLab, GitHub or Gitee. We will capture the most recent commit *prior* to when the assignment marking begins, which can be at any moment after the deadline (if you delay, you assume the risk).

Summary. In this assignment, you are going to develop a client/server networked system that implements a “tuple space”. Clients send requests to include, read, or delete tuples. The server needs to deal with multiple clients at the same time. Each client connects to the server, starting “a session”, and sends one or more requests during the session, until it closes.

After sending a request, the client waits for the response to arrive before sending the next request to the server. This is what we call “synchronous behaviour”, which simplifies the client implementation. After a number of requests and responses, the client terminates the session and closes the connection. The server detects that the client closed the connection and also ends the session with that client.

Tuples. The server will implement a “tuple space”. Each tuple is a key-value pair, in which both key and value are strings of up to 999 characters each. The key needs to be unique, i.e. there cannot be two tuples with the same key. You can imagine this as a simple table containing two columns, namely a key and a value.

Here is an example of what a tuple space would be, in this case words and their meanings:

key	value
greeting	(usually plural) an acknowledgment or expression of good will (especially on meeting)
lighthouse	a tower with a light that gives warning of shoals to passing ships
Andre Maginot	French politician who proposed the Maginot Line (1877-1932)

The server implements **three** operations (the use of colours here is just to enhance visualisation):

- v = READ(k): if a tuple with key k exists, the tuple (k, v) is read and the value v is returned; if k does not exist, the operation fails and v returns empty;
- v = GET(k): if a tuple with key k exists, the tuple (k, v) is deleted and the value v is returned; if k does not exist, the operation fails and v returns empty;
- e = PUT(k, v): if the key k does not exist already, the tuple (k, v) is added to the tuple space and e returns 0; if a tuple with key k already exists, the operation fails and e equals 1 is returned.

More precisely, the responses from the server can be:

- OK (k, v) read
- OK (k, v) removed
- OK (k, v) added
- ERR k already exists (in case of a PUT using a key k that already exists)
- ERR k does not exist (in case of a READ or GET using a key k that does not exist)

Request file. A client is activated via the command line and receives as one of the arguments the name of a text file containing requests, one per line. For example, a file would look like:

```
PUT Manchester-United 20
PUT Liverpool 19
PUT Arsenal 31
PUT Everton 9
PUT Manchester-City 9
READ Liverpool
READ Arsenal
PUT Arsenal 13
GET Arsenal
PUT Arsenal 13
GET Manchester-United
READ Manchester-United
...
```

As a rule, the “collated size” of the k and v (as a single string separated by spaces) cannot exceed 970 characters. If the input file to the client violates the condition, output an error message and ignore the entry.

Client output. As the client runs, for each line processed, the client will display what the operation was and its result. For example, starting from an empty tuple space, the output would be

```
PUT Manchester-United 20: OK (Manchester-United, 20) added
PUT Liverpool 19: OK (Liverpool, 19) added
PUT Arsenal 31: OK (Arsenal, 31) added
PUT Everton 9: OK (Everton, 9) added
PUT Manchester-City 9: OK (Manchester-City, 9) added
READ Liverpool: OK (Liverpool, 19) read
READ Arsenal: OK (Arsenal, 31) read
PUT Arsenal 13: ERR Arsenal exists
GET Arsenal: OK (Arsenal, 31) removed
PUT Arsenal 13: OK (Arsenal, 13) added
GET Manchester-United: OK (Manchester-United, 20) removed
READ Manchester-United: ERR Manchester-United does not exist
...
```

In the above example, the value v is the number of premiership titles for each club, but it could be a sentence (any string up to a 992 printable characters are supported, including spaces).

Server output. The server, on its side, displays every 10s a summary of the current tuple space, containing the number of tuples in the tuple space, the average tuple size, the average key size, and the average value size (string), the total number of clients which have connected (finished or not) so far, the total number of operations, total READs, total GETs, total PUTs, and how many errors.

Multi-threaded server. As stated above, the server needs to handle sessions with multiple clients at the same time. For this reason, the server will use multiple threads, each thread being created to deal with a single client. This allows the server implementation to be much simpler than if having to interleave requests from multiple clients. The server will spawn a new thread whenever a new client connects with the server. When the session/connection with the client is closed, the thread terminates.

Command line arguments to client and server. You first start the server providing the port number to be used by the server to wait for incoming connections. This has to be a high port, such as 51234 (50000 <= port <= 59999).

On a different host, you start each of your clients. The client gets three arguments:

- the **hostname** where the server resides (can be “localhost” if client and server are in the same host);
- the **port number** to connect with the server (the same value used by the server);
- the **pathname** to the text file that contains the requests to be processed (sent to the server), in the format described above.

Multi-threaded server that uses TCP sockets. The focus of the assignment is to use TCP sockets to implement a client/server protocol with the three supported operations (READ, GET, and PUT) and a server which correctly implements these operations despite concurrent access of threads to the (shared) tuple space. Your implementation must adhere to the protocol because your client and server should be able to interoperate with the server and client developed by other students.

Protocol. The protocol to be implemented encodes the request messages as follows:

- NNN **R** k
- NNN **G** k
- NNN **P** k v

NNN is three characters indicating the total message size, the first letter indicates the command (**R** for READ, **G** for GET, and **P** for PUT), k is the key and v is the value. The minimum size is 7 (which is when the key k is a single character in a READ/GET) and the maximum is 999.

Examples of request messages that can be transmitted to a server:

- 007 **R** a
- 010 **R** abcd
- 012 **G** 123456
- 053 **P** good-morning-message how are you feeling today?

The response messages (in line with what was defined earlier) are also implemented using strings, whose format is one of the following:

- NNN **OK** (k, v) read
- NNN **OK** (k, v) removed
- NNN **OK** (k, v) added
- NNN **ERR** k already exists
- NNN **ERR** k does not exist

Examples (if k and v are a single alphanumeric character):

- 018 **OK** (k, v) read
- 021 **OK** (k, v) removed
- 014 **OK** k added
- 024 **ERR** k already exists
- 024 **ERR** k does not exist

Language. The code can be developed in either Java or Python (which may be "object-oriented" if desired). The development must be based on features explored/explained in the lectures, such as the "synchronized" keyword if developing in Java.

How to test your networked system. We provide example files to be given to the clients, and the expected outputs for each client and the server. In the test, the tuple space contains 100 different words (from an English dictionary). Each of the files has 100,000 requests.

Follow these steps:

1) start the server at one host
2) run all the clients one after the other (e.g. for Java it could be
   for i in {1..10}; do java myclient server 51234 client-$i.txt; done
3) note the outputs produced by the clients and specially, the server
4) close the server (^c) and start it again
5) run all the clients one after the other (e.g. for Java it could be
   for i in {1..10}; do java myclient server 51234 client-$i.txt &; done
6) note the outputs produced by the server

Marking rubric. Your project in GitLab, GitHub or Gitee must be named "COMPX234-A3".

Assessment	**Marks**
development history (small steps at each commit, with detailed information)	20
code quality (documentation, indentation)	10
pass all tests with the test files provided	50
pass all additional tests (cases not provided, but will explore race conditions and corner cases)	20