P434: Report 1

Joshua Elms - Due January 31, 2024

1 Architecture

My architecture is object-oriented. I encapsulate three entities in my code: Server, Client, and KVStore. The Server class contains all the functionality to bind a socket to an address, listen for requests, and dispatch the necessary methods to handle get/set requests. The initialization of a Server object instantiates a KVStore within the Server, which the Server will direct get and set requests to. A Client object expects to connect to a Server on instantiation, after which the Client can send get/set requests and expect well-formed responses from the Server.

2 File structure

The persistent storage employs binary files and standard read/write IO procedures for Python.

Three constants are relevant to the file format: INT_SIZE, INT_ORDER, and KEY_SIZE. INT_SIZE and KEY_SIZE specify the size, in bytes, that the Server, Client, and KVStore will use to represent keys and the integers that specify value sizes. INT_ORDER is a string, either "big" or "little", which specifies the endian-ness of the integers used to store value sizes. In my tests, I use:

- INT_SIZE = 4
- INT_ORDER = "big"
- KEY_SIZE = 60

Each entry in the file contains first the key (KEY_SIZE b), then the size of the value (INT_SIZE b), and finally the value itself (size specified by previous value). No delimiters are used between the key-size-value triplets or the entries themselves. While not entirely binary, an example data file with two entries, INT_SIZE = 1, and KEY_SIZE = 3 might look like this:

cat00001000whiskersdog00000100bark

We can see that the first entry is KEY-"cat", SIZE-8, VALUE-"whiskers", and the second entry is KEY-"dog", SIZE-4, VALUE-"bark".

3 Design specifics

3.1 Dependencies

My project depends only on the user having a version of python ≥ 3.10. Older versions of python would probably work but for the type-hinting union operator that I use in my function headers. I use many standard libraries, but none that the user would have to install separately. These packages are:

1. socket: forms the backbone of the Server and Client classes by enabling message passing
2. pathlib: Path class used to represent and modify paths in an OS-agnostic way
3. time: only functions used are to sleep and gather current time value

4. subprocess: Popen used to run Server and Client from within a single file during testing
5. json: dumps and loads used to pass and parse arguments for Server and Client in testing
6. sys: used to gather arguments passed to Server and Client during testing
7. shutil: used to copy and delete files for KVStore

3.2 Connection

The Server class binds a socket to an address upon instantiation. The Client class will be passed the address of the Server instance and will try to connect to it upon its instantiation. If the Client cannot connect to the Server immediately, it will continue trying at five-second intervals until its timeout is exhausted. The Server has a separate timeout which specifies how long it should wait for a Client to connect to it before closing. If the Server closes while connected to a Client, the Client will recognize this and inform the user with a message. If the Client closes while connected to the Server, the Server will recognize this and return to listening for a Client until its timeout is exhausted.

While the Server is not truly concurrent (able to handle multiple requests simultaneously with threads), it does have a backlog parameter which enables it to handle a variable number of Clients waiting in a queue to be served after the current Client. If more Clients attempt to connect than the backlog allows, the newcomers will be ignored. All of the timeouts and the backlog previously mentioned are parameters that the user can set upon instantiation of either the Server or Client.

3.3 Input validation

The first pass at input validation is implicit: the Client class does not provide a general Client.send_request method, but rather implements Client.set and Client.get. These methods have required parameters (key and value for set, key for get) without which the methods will not execute. In this way, the messages passed to the Server by the Client are coerced into being well-formed by nature. The keys and values passed to the Client methods, however, are validated on the Client side as well. Keys are checked to ensure type (must be bytes) and size (must be between 1 and KEY_SIZE bytes long). The same validation is performed for values, except values must be bytes of length 1 to $2^{\text{INT\_SIZE}*8-1}$ bytes long. If any of these validations are violated, the Client will raise an appropriate error (TypeError or ValueError) for the user to handle.

3.4 Message Passing and Format

The messages passed from Client to Server and Server to Client are also evaluated to ensure that they are well-formed. This evaluation takes place in the get and set functions of Client and Server alike, and if passed values are found not to meet expectations, the appropriate error will be raised or message passed, depending on the context.

Messages are sent in batches of another global constant, BUF_SIZE. This is often only relevant to large values, as neither keys nor the rest of the formatted get/set messages exceed the typically large BUF_SIZE (4096b in my testing). The functions receiving these buffered messages are prepared to connect any pieces they receive and wait until either the END signal or the specified number of bytes have been received, depending on the type of message sent by the other party. Short sleeps (SLEEPTIME, 0.05s in my testing) are executed before either actor attempts to receive data to allow time for the message to be passed. This could be shorter with empirical testing to determine which sleep duration is optimal.

Messages are formatted as specified in the assignment; set requests are passed by the client in two lines, while get requests are passed in just one. Examples are shown below.

Client.set sends:
b‘set name \x00\x00\x00\x07 \r\n’
b‘Josh \r\n’

Server responds with:
b‘STORED \r\n’

Client.get sends:
b‘get name \r\n’

Server responds with:
b‘VALUE name \x00\x00\x00\x07 \r\n’
b‘Josh \r\n’
b‘END \r\n’

3.5 File IO

Because the Server's memory has to persist, a file is used to store all of the key-value pairs generated by Clients. Each Server contains a KVStore class within it, and KVStore is the component that handles writing to and reading from this persistent storage. The method used for storage prioritizes memory efficiency (both in the file and during read/write actions) over readability or low latency. This is implemented by a simple procedure, described below.

When performing a set operation, first attempt to get(key) to see if the key already exists. If not, continue with the set by appending the key-size-value triplet to the end of the file. If the key does exist, though, check whether the new value assignment differs from the existing value. If not, leave the entry alone. If the value is different, then the existing file will be copied into a temporary file. Both files will be opened and the file pointers will be set to the entry that needs to be updated. The temporary file pointer will then be moved to the next entry and all entries after that which is being updated will be written into the original file; essentially, the entry of interest is overwritten by everything after it. Finally, the updated entry will be placed at the end of the original file. This approach means that all of the entries before that which is being updated will be ignored; they never need to be read or written, saving IO time. Additionally, the key that has been most recently updated will be written to the end of the file, which means that it is readily available for an update if it is modified again soon. Finally, the temporary file is deleted once the original file is ready. This approach allows the keys most often updated to have the shortest update times.

4 Usage Instructions

Usage of this project is simple. Unzip the archive provided, ensure you have a valid version of python as specified in the Dependencies section, and run “python testing.py” to perform various tests. The main function in “testing.py” has a parameter you can set to change which test is run. If you are interested in building on the code or directly interfacing with the Server or Client classes, the project is heavily documented and the test cases provide useful examples to work from.

5 Test Cases

1. Test 1: 3 clients making well-formed requests (serially, only one is handled at a time).

```python
case 1:
    # open server
    server_proc = subprocess.Popen(["python", "popen_server.py", json.dumps(server_info)], close_fds=True)
    procs.append(server_proc)

    time.sleep(1)

    # prepare client instructions
    base_client_info = dict(HOST=HOST, PORT=PORT)
    # open client
    for i in range(1, 4):
        time.sleep(0.1)
        client_info = {
            **base_client_info,
            "id": i,
            "instructions": (
                set("name", "John Doe"),
                get("name"),
                set("age", i*5),
                get("age"),
                set("name", "Jane Doe"),
                get("name")
            )
        }
        client_proc = subprocess.Popen(["python", "popen_client.py", json.dumps(client_info)])
        procs.append(client_proc)
```

```
jmelms@login2:~/distributed_computing/exercise_1> python testing.py
Beginning test 1
Server: Listening on 127.0.0.1:65000...
SERVER: Connected by ('127.0.0.1', 58882)
CLIENT1: 'set name John Doe' received response: b'STORED \r\n'
CLIENT1: 'get name' received response: b'John Doe'
CLIENT1: 'set age 5' received response: b'STORED \r\n'
CLIENT1: 'get age' received response: b'5'
CLIENT1: 'set name Jane Doe' received response: b'STORED \r\n'
CLIENT1: 'get name' received response: b'Jane Doe'
CLIENT1: Connection closed
SERVER: Client disconnected
SERVER: Connected by ('127.0.0.1', 58888)
CLIENT2: 'set name John Doe' received response: b'STORED \r\n'
CLIENT2: 'get name' received response: b'John Doe'
CLIENT2: 'set age 10' received response: b'STORED \r\n'
CLIENT2: 'get age' received response: b'10'
CLIENT2: 'set name Jane Doe' received response: b'STORED \r\n'
CLIENT2: 'get name' received response: b'Jane Doe'
CLIENT2: Connection closed
SERVER: Client disconnected
SERVER: Connected by ('127.0.0.1', 58892)
CLIENT3: 'set name John Doe' received response: b'STORED \r\n'
CLIENT3: 'get name' received response: b'John Doe'
CLIENT3: 'set age 15' received response: b'STORED \r\n'
CLIENT3: 'get age' received response: b'15'
CLIENT3: 'set name Jane Doe' received response: b'STORED \r\n'
CLIENT3: 'get name' received response: b'Jane Doe'
CLIENT3: Connection closed
SERVER: Client disconnected
SERVER: Timeout reached at 3 seconds
Server: Detached from 127.0.0.1:65000
```

2. Test 2: 1 client making dozens of requests quickly. Some requested keys do not exist.

```python
case 2:
    # open server
    server_proc = subprocess.Popen(["python", "popen_server.py", json.dumps(server_info)], close_fds=True)
    procs.append(server_proc)

    time.sleep(1)

    # prepare client instructions
    base_client_info = dict(HOST=HOST, PORT=PORT, vocal=True)
    client_info = {**base_client_info, "id": 1, "instructions": []}
    collatz = lambda x, n=0: n if x == 1 else collatz(x * 3 + 1, n+1) if x % 2 else collatz (x // 2, n + 1)
    # open client
    for i in range(80, 100):
        instr = set(f"len(collatz({i}))", collatz(i))
        client_info["instructions"].append(instr)

    for i in range(90, 102):
        instr = get(f"len(collatz({i}))")
        client_info["instructions"].append(instr)

    client_proc = subprocess.Popen(["python", "popen_client.py", json.dumps(client_info)])
    procs.append(client_proc)
```

```
jmelms@login2:~/distributed_computing/exercise_1> python testing.py
Beginning test 2
Server: Listening on 127.0.0.1:65000...
SERVER: Connected by ('127.0.0.1', 39054)
CLIENT1: 0th instruction 'set len(collatz(80)) 9' received response: b'STORED \r\n'
CLIENT1: 1th instruction 'set len(collatz(81)) 22' received response: b'STORED \r\n'
CLIENT1: 2th instruction 'set len(collatz(82)) 110' received response: b'STORED \r\n'
CLIENT1: 3th instruction 'set len(collatz(83)) 110' received response: b'STORED \r\n'
CLIENT1: 4th instruction 'set len(collatz(84)) 9' received response: b'STORED \r\n'
CLIENT1: 5th instruction 'set len(collatz(85)) 9' received response: b'STORED \r\n'
CLIENT1: 6th instruction 'set len(collatz(86)) 30' received response: b'STORED \r\n'
CLIENT1: 7th instruction 'set len(collatz(87)) 30' received response: b'STORED \r\n'
CLIENT1: 8th instruction 'set len(collatz(88)) 17' received response: b'STORED \r\n'
CLIENT1: 9th instruction 'set len(collatz(89)) 30' received response: b'STORED \r\n'
CLIENT1: 10th instruction 'set len(collatz(90)) 17' received response: b'STORED \r\n'
CLIENT1: 11th instruction 'set len(collatz(91)) 92' received response: b'STORED \r\n'
CLIENT1: 12th instruction 'set len(collatz(92)) 17' received response: b'STORED \r\n'
CLIENT1: 13th instruction 'set len(collatz(93)) 17' received response: b'STORED \r\n'
CLIENT1: 14th instruction 'set len(collatz(94)) 105' received response: b'STORED \r\n'
CLIENT1: 15th instruction 'set len(collatz(95)) 105' received response: b'STORED \r\n'
CLIENT1: 16th instruction 'set len(collatz(96)) 12' received response: b'STORED \r\n'
CLIENT1: 17th instruction 'set len(collatz(97)) 118' received response: b'STORED \r\n'
CLIENT1: 18th instruction 'set len(collatz(98)) 25' received response: b'STORED \r\n'
CLIENT1: 19th instruction 'set len(collatz(99)) 25' received response: b'STORED \r\n'
CLIENT1: 20th instruction 'get len(collatz(90))' received response: b'17'
CLIENT1: 21th instruction 'get len(collatz(91))' received response: b'92'
CLIENT1: 22th instruction 'get len(collatz(92))' received response: b'17'
CLIENT1: 23th instruction 'get len(collatz(93))' received response: b'17'
CLIENT1: 24th instruction 'get len(collatz(94))' received response: b'105'
CLIENT1: 25th instruction 'get len(collatz(95))' received response: b'105'
CLIENT1: 26th instruction 'get len(collatz(96))' received response: b'12'
CLIENT1: 27th instruction 'get len(collatz(97))' received response: b'118'
CLIENT1: 28th instruction 'get len(collatz(98))' received response: b'25'
CLIENT1: 29th instruction 'get len(collatz(99))' received response: b'25'
CLIENT1: 30th instruction 'get len(collatz(100))' received response: 'KEY NOT FOUND \r\n'
CLIENT1: 31th instruction 'get len(collatz(101))' received response: 'KEY NOT FOUND \r\n'
CLIENT1: Connection closed
SERVER: Client disconnected
SERVER: Timeout reached at 3 seconds
Server: Detached from 127.0.0.1:65000
```

3. Test 3: Client tests keys that are within size limits and too long (60b and 251b, respectively).

```python
case 3:
    # open server
    server_proc = subprocess.Popen(["python", "popen_server.py", json.dumps(server_info)], close_fds=True)
    procs.append(server_proc)

    time.sleep(1)

    # prepare bad client instructions (key too long)
    base_client_info = dict(HOST=HOST, PORT=PORT, vocal=True)
    client_info = {
        **base_client_info,
        "id": 1,
        "instructions": [
            set("a"*60, "key not too long"),
            get("a"*60),
            set("a"*251, "key too long"),
            get("a"*251)
        ]
        }

    client_proc = subprocess.Popen(["python", "popen_client.py", json.dumps(client_info)])
    procs.append(client_proc)
```

```
jmelms@login2:~/distributed_computing/exercise_1> python testing.py
Beginning test 3
Server: Listening on 127.0.0.1:65000...
SERVER: Connected by ('127.0.0.1', 49716)
CLIENT1: 0th instruction 'set aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa key not too long' received response: b'STORED \r\n'
CLIENT1: 1th instruction 'get aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa' received response: b'key not too long'
Traceback (most recent call last):
  File "/geode2/home/u040/jmelms/BigRed200/distributed_computing/exercise_1/popen_client.py", line 37, in <module>
    response = client.set(key=bparts[1], value=bparts[2])
  File "/geode2/home/u040/jmelms/BigRed200/distributed_computing/exercise_1/core.py", line 303, in set
    assert 1 <= len(key) <= KEY_SIZE, ValueError(
AssertionError: Key length must be between 1-60 bytes, key of size 251b was passed
SERVER: Client disconnected
SERVER: Timeout reached at 3 seconds
Server: Detached from 127.0.0.1:65000
```

4. Test 4: Client tests missing keys and ostensibly (but not actually) malformed requests.

```python
case 4:
    # open server
    server_proc = subprocess.Popen(["python", "popen_server.py", json.dumps(server_info)], close_fds=True)
    procs.append(server_proc)

    time.sleep(1)

    # prepare bad client instructions (malformed)
    base_client_info = dict(HOST=HOST, PORT=PORT, vocal=True)
    client_info = {
        **base_client_info,
        "id": 1,
        "instructions": [
            "get key", # key doesn't exist yet
            "set keyvalue key value", # if command is space delimited, this would be problem. (Client should be able to handle this)
        ]
        }

    client_proc = subprocess.Popen(["python", "popen_client.py", json.dumps(client_info)])
    procs.append(client_proc)
```

```
jmelms@login2:~/distributed_computing/exercise_1> python testing.py
Beginning test 4
Server: Listening on 127.0.0.1:65000...
SERVER: Connected by ('127.0.0.1', 40234)
CLIENT1: 0th instruction 'get key' received response: 'KEY NOT FOUND \r\n'
CLIENT1: 1th instruction 'set keyvalue key value' received response: b'STORED \r\n'
CLIENT1: Connection closed
SERVER: Client disconnected
SERVER: Timeout reached at 3 seconds
Server: Detached from 127.0.0.1:65000
```

6 Known limitations

Most of the limitations of this project have been listed already. The Server is only able to handle one client at a time (but can keep others in a backlog) and keys are limited to the size determined by the instantiation of the KVStore (once made, the file cannot have the key or integer sizes changed). Other limitations will be addressed in the following section on potential improvements to this project.

7 Potential Improvements

I can think of many ways in which this project could be improved; some of these enhance its capabilities, while others make it smoother to interact with. I list them below.

1. Simple concurrency could be implemented via the select module for python, which would allow the Server to bind many sockets to many addresses and pass these to threads, each of which could handle requests independently. I would need to find a way to place a lock on the storage file while set operations are occurring.
2. Currently, keys and values can only be of type bytes. Adding compatibility with bytesarray would enhance the project, as bytesarray is the mutable equivalent of bytes in python.
3. The KVStore is fragile; it does not know what values are being used for INT_SIZE, INT_ORDER, or KEY_SIZE. As of now, these are global constants in the primary file, "core.py". A better system would be to keep metadata on each KVStore file, which could be in the filename, file header, or a separate file. This would allow the KVStore to ensure that it can read and write to the storage file under all conditions.
4. The sleep calls used before receiving data could probably be removed; I've updated my code to be more tolerant of timing. This would require further testing to ensure the project still functions properly.
5. Making the project functional across a network of distributed systems would be a great improvement, though that is explicitly a goal for later this semester.
6. The "core.py" file might be made shorter, more modular, and more readable by encapsulating the socket.receive loops as a function. Currently, I have two general structures that these loops follow, and providing a function for each would ensure that the codebase is easier to follow and maintain in the future.

8 References

As this is not a formal publication, I will simply provide the titles and links of the websites I used during this project.

1. Socket Programming in Python (Guide) - Real Python
2. Socket Programming HOWTO - Python 3.12 Documentation
3. Reading binary file and looping over each byte - Stack Overflow
4. Assignment 1 - Memcached Lite
5. memcached protocol - github
6. Subprocess management - Python 3.12 Documentation
7. JSON encoder and decoder - Python 3.12 Documentation