CSE 511: Lab 1 Documentation

Team: Game of Threads
Members: **Soumen Basu**, **Makesh Tarun Chandran**, **Vivek Bhasi**

1. **Problem Statement:**

- Implement a key-value store using three different styles of concurrency and IO management.
- Perform a comparative experimental performance evaluation of the three implementations.
- The three styles are:
  - (i) MT: multi-threaded with one thread handling per client request and using blocking IO for both sockets and files.
  - (ii) EP :a single event-driven thread with polling-based event notification, and (iii)
  - (iii) ES: a single event-driven thread with signaling-based event notification
- For styles (ii) and (iii) use Non-blocking or asynchronous sockets(as appropriate). However, for file IO blocking IO calls will be used that will be made by “helper” threads that will communicate with the main event-driven thread via shared memory mechanisms and exercise appropriate care for synchronization.

2. **What we managed to do:**

   We were successful in implementing the designs and also were able to experimentally evaluate Design 1 and 2. We used Python3 to implement design 1 and 2. Design 3 was implemented in C. The setup is described in detail in the “Experimentation” section of this document. The code is well documented using comments. We couldn’t complete the experimental evaluation of design 3. We could not finish the evaluation part for 10KB data size. But we managed to finish experiments for 100B and 1KB data sizes, for the 90:10 and 10:90 GET:PUT ratios. We couldn’t do 50:50 ratio.

3. **Issues faced:**
   Implementing design 3 is *probably* not possible in Python3. **siginfo_t** in python is a class abstraction of the C structure. But the implementation contains only a few struct variables from the C siginfo_t and it doesn’t contain the fd. Python provides some high level libraries (asyncore/ asyncio) and conditional variables but our code inspection revealed that the underlying mechanism is *select* based polling.

4. **Populating the DB:**
   Can be populated using the script *populate_db.py*

```
python populate_db.py -n <num_entries> -s <key-value-size> -d <db_name>
```

5. **Requirements:**
   - Python 3
   - PickleDB for persistent storage (`pip install pickledb`)
6. **Running Servers:**

   Running the MT server:

   ```
   python multi_threaded_server.py <0.0.0.0> <port> <cache_size> <db_name>
   ```

   Running the EP server:

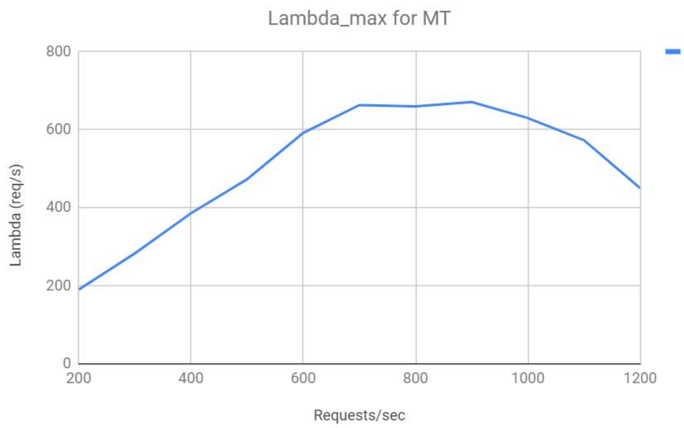
   ```
   python polling_server.py <0.0.0.0> <port> <cache_size> <db_name> <num_helpers>
   ```

   Running the ES server:

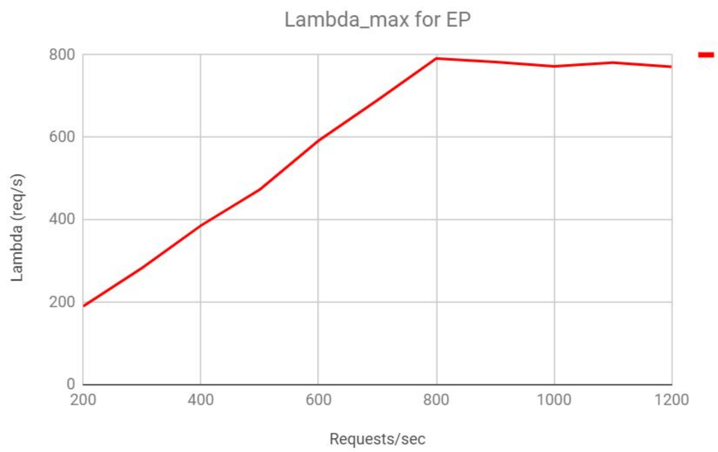
   ```
   gcc -pthread server.c -o server.o

   ./server

   [configs for the server host, port, cache_size, num_helpers are in server.h]
   ```
7. **Key-Value Pair:**
   - Write-back, Allocate on Write Cache policy
   - LRU eviction policy
   - The following APIs are released to client:
     - i. Get(Key) --> Value/ -1
     - ii. Put(Key, Value) --> ACK/ -1
     - iii. Insert(Key, Value) --> ACK/ -1
     - iv. Delete(Key, Value) --> ACK/ -1
8. **Experimental Setup and results:**
   - Server: We ran on a t2.small AWS EC2 instance (Ubuntu 18.04 w/ 20GB gp2)
   - Client Node: t2.xlarge (Ubuntu 18.04 w/ 20GB gp2)
   - Request Distribution is deterministic in nature (after 100s increase the number of requests by 100).
   - Skewed key popularity distribution: 10% keys make 90% requests
   - **Lambda_max Measurement:**
     - i. We started generating requests with the above stated request distribution.
     - ii. We spawned processes from the client node - each process generates 10 req/sec
     - iii. We measure the new requests in server using *Wireshark* and *tcptrace* tools (based on appropriate filtering).
     - iv. The trend that we observe is the lambda in server increases with the number of requests and then flattens, finally decreases from the max. (We believe this is the DoS phase).
   - Lambda_max plots:(Next page)
     - i. MT: lambda_max = 663 (we approximated this to 650)



ii. EP: lambda_max = 791 (approximated to 800)



GET/ PUT Latencies for different settings:

(get:put ratio is the first column) 50 -> 50%-ile latency, 95 -> 95%-ile latency

Key-value sizes: (100, 1000, 10000) B All latency values are in milliseconds (ms).

lambda_max		100				1K				10K			
		GET Latency		PUT Latency		GET Latency		PUT Latency		GET Latency		PUT Latency	
		50	95	50	95	50	95	50	95	50	95	50	95
MT	90:10	63	556	90	690	103	596	129	792	350	1080	341	2109
	10:90	70	592	104	702	127	623	134	872	343	1102	393	1903
	50:50												
EP	90:10	18.15	67.2	18.4	68.6	48.2	127.2	41.4	168.7	181	495	178	705
	10:90	20.13	65.2	19.2	70.2	61.3	153.2	59.2	210.6	243	452	191	790
	50:50												
ES	90:10												
	10:90												
	50:50												

lambda_max/ 2		100				1K				10K			
		GET		PUT		GET		PUT		GET		PUT	
		50	95	50	95	50	95	50	95	50	95	50	95
MT	90:10	12	49	24	52	36	189	52	252				
	10:90	17	48.8	21	57	39	208	61	247				
	50:50												
EP	90:10	6	42.9	8	44.1	18	142	38	155				
	10:90	10	47.2	11	46.2	29	138	32	163				
	50:50												
ES	90:10												
	10:90												
	50:50												

lambda_max/ 10		100				1K				10K			
		GET		PUT		GET		PUT		GET		PUT	
		50	95	50	95	50	95	50	95	50	95	50	95
MT	90:10	8	44.9	12	48	32	209	49	253				
	10:90	8	49	17	54	41	239	57	261				
	50:50												
EP	90:10	7	45	8	46.3	30	169	52	204				
	10:90	8	44.1	12	44.25	34.75	172	51.3	294				
	50:50												
ES	90:10												
	10:90												
	50:50												