Distributed and Operating Systems *CS 677, Spring 2020*

Lab 03 - Pygmy.com - Performance Evaluation

Response times

/search and /lookup, with and without caching

To get latencies for /search and /lookup operations with and without caching, I wrote a test-suite that reports the latencies. To make things easier, our implementation of UIServer exposes **/invalidate** endpoint which we can use directly to invalidate cache entry rather actually performing a /buy operation! The experiment performs the following steps:

- Forcibly invalidate cache entry of a book.
- Perform lookup of a book and note the latency.
- Perform lookup of the book again so that a cache-hit occurs, and note the latency.

The above steps are performed **50** times for each of the **7** books in pygmy.com and the latencies are averaged out and then reported (i.e. latencies reported below are averaged from 350 requests).

Here are our observation:

- When a cache-miss occurs, the /lookup operation takes **184.35 milliseconds** to retrieve the results. When a cache-hit occurs, the /lookup operation takes, a ridiculously low, **0.954 milliseconds**.

The same experiment is repeated for /search operations with the exception that /search results are not invalidated frequently but only every 10 /search operations (this is because in a real-world scenario, /buy operation will never invalidate a /search response).

- When a cache-miss occurs, the /search operation takes **2.7 milliseconds** to retrieve the results. When a cache-hit occurs, the /search operation takes **0.456 milliseconds**.

Figure 1 shows the output of the experiment, and the latencies as noted above.

```
elnux3 the-new-pygmy.com) > ./report-cache-latency.sh


================================================================================
Starting test #
================================================================================

Starting /lookup latency tests..
Average /lookup latency in the event of cache-miss: 184.35142857142858 milliseconds..
Avergae /lookup latency in the event of cache-hit: 0.9542857142857143 milliseconds..
Starting /search latency tests..
Average /search latency in the event of cache-miss: 2.7 milliseconds..
Avergae /search latency in the event of cache-hit: 0.456 milliseconds..
```

Figure 1: Cache miss/hit latencies for /search and /lookup

To reproduce these results, open an edlab terminal (of course after already having deploying all the servers) and from the root directory of the source, execute:
chmod +x *.sh ./report-cache-latency.sh

/buy

To compute response times in /buy operation, we followed same approach as in lab-2: we logged the entry and exit time of each of the 3 servers involved in the /buy operation (UI/Order/Catalog). These entry/exit timestamps are logged by the servers into a **.delay** file which we parse using a Java program. The tier-wise breakdown of /buy operation collected over **70** /buy requests are as follows:

- CatalogServer takes, on average, **1002.885 milliseconds** to execute a /buy order. Note that this time includes:
  - wait-time to acquire the token-ring
  - time to replicate the order on the other replica CatalogServer
  - logging the transaction into the CatalogServer Write Ahead Log (which being a disk-I/O operation is time-consuming).

  Also note that in lab-2, with no replication, CatalogServer's response time used to be **1.7842 millisecond**.
- OrderServer takes, on average, **2207 milliseconds** to finish executing a /buy operation. Note that this includes the time taken by the CatalogServer to exceute the order. Also note that in lab-2, with no replication, OrderServer's response time used to be **7.301 millisecond**.
- UIServer takes, on average, **3009.943 milliseconds** to finish executing a /buy operation. Note that this includes the time taken by the CatalogServer and OrderServer to execute the order and writing to their respective WALs. Also note that in lab-2, with no replication, UIServer's response time used to be **8.791 millisecond**.

As is obvious from the data above, there has been a drastic increase in the response time for /buy operation in all the three tiers. This is expected because replication provides data redundancy and fault-tolerance at the cost of increased processing time (State machine replication, using the token-ring algorithm, is a costly affair and hence the higher response times). But the silver lining is that even if one OrderServer and one CatalogServer replica go down at the same time, pygmy.com can continue to serve requests.

Caching, on the other hand, has drastically improved response times for /search and /lookup operations!

The above observation was when only a single request queued to pygmy.com at any given time i.e. no concurrent requests. We further computed response times of /buy operation when **3** different threads were making concurrent requests to pygmy.com. Here is the tier-wise breakdown for when 3 different threads were making concurrent requests:

- CatalogServer takes, on average, **1100.714 milliseconds** to execute a /buy order.
- OrderServer takes, on average, **2701.857 milliseconds** to finish executing a /buy operation.
- UIServer takes, on average, **3135.244 milliseconds** to finish executing a /buy operation. Note that this includes the time taken by the CatalogServer and OrderServer to execute the order and writing to their respective WALs. Also note that in lab-2, with no replication, UIServer's response time used to be **8.791 millisecond**.

Figure 2 shows this observation: Overall, there is not much difference in processing times between consecutive and concurrent (3 at a time) requests, except for the approximately 500 millisecond increase in OrderServer's /buy path.

The **.delay** files using which I reported the above observations is present in docs/perf directory of repository.

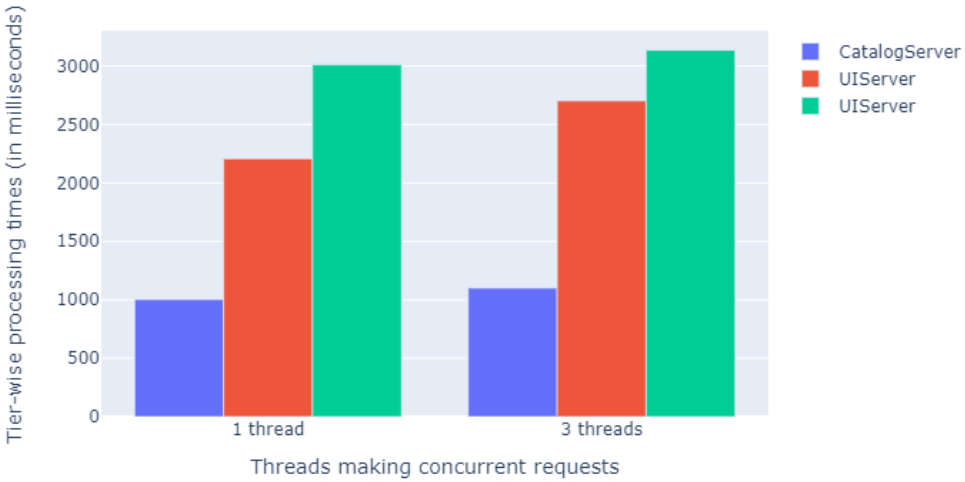

Figure 2: /buy response time w.r.t concurrent requests

Cache consistency

To verify that the contents in the cache reflect the backing database (i.e. cache is consistent), I wrote a test-suite that performs the following:

- Perform initial **/lookup** of a book and note the stock of the book in the inventory. This response could be a cached response or may be not.
- Perform a **/buy** operation on the same book.
- Perform a second **/lookup** and note the stock of the book. This response **should not** be the same as the last one but instead should correctly reflect the new stock of the book, thus proving that the cached-response was correctly invalidated and the cache is consistent!

The above steps are repeated 5 times for each of the **7** books in the inventory. This simple experiment should prove that the server-push mechanism that we implemented correctly invalidates the cache when a /buy operation is processed!

The overhead associated with cache consistency is that after every /buy operation, the cached response is invalidated and a subsequent /lookup has to actually be directed to the CatalogServer. This may be slighly expensive in a buy-heavy system (i.e. where buy requests are as frequent as lookup requests). The latency, in /search and /lookup, in the case of a cache-miss has already been reported in the previous section.

Fault tolerance

To verify that fault tolerance in pygmy.com works as expected, I wrote a test-suite that doe the following:

- Shuts down one of the **CatalogServer** replica.
- Shuts down one of the **OrderServer** replica.
- Forcibly invalidate all cached-responses so that all requests are actually routed to the CatalogServer and/or OrderServer.
- Issues a series of /search, /lookup and /buy requests to verify that all the requests are processed correctly. When the test is ran, some requests will be processed with a delay: note that this happens when the UIServer detects that one of the OrderServer replica has gone down and migrates the request to another replica. The same migration happens at OrderServer's end when it detects that one of the CatalogServer replica has gone down.
- The test also verifies that all the requests are served by the CatalogServer/OrderServer that was up and running. We are able to do this because every response comes stamped with the identity of the CatalogServer/OrderServer replica that served the request. The test checks this stamp and confirms that the response was indeed served by the replica that was running.

The above steps verify the fault-tolerance capabilities of pygmy.com i.e. requests are still processed even if an OrderServer and a CatalogServer replica go down!

But we still need to verify fault-recovery capabilities of pygmy.com. The test proceeds to do the following:

- Restart the **CatalogServer** replica that was stopped in the previous step (i.e. while verifying fault-tolerance). We do this by adding **"recovery"** paramter as command-line argument to CatalogServer which tells the CatalogServer to initiate recovery process before it can start processing requests.
- Also restart the **OrderServer** replica that was stopped earlier. OrderServer has no **recovery** mode since OrderServer doesn't need to synchronize any data (each OrderServer maintains independent WAL logs which need not be synchronized).
- Wait for 20 seconds, during which we expect CatalogServer's recovery mechanism to complete (this mechanism was explained in detail in the design-doc). Within this time, we also expect that both restarted replica servers will have introduced themselves to the UIServer so that future requests can be routed to these servers as well.
- Issue a series of /search, /lookup and /buy requests. Not only should all of them be processed by pygmy.com, the test also verifies at least some of them is processed by the restarted replica servers (the test issues enough requests to ensure that a round-robin load-balancer will direct atleast one request to the newly restarted servers).

At this point, we have verified both the fault-tolerance and fault-recovery capabilities of pygmy.com.