Assignment 4

Nicolai Jørgensen and Yiran Zhang

December 18, 2017

1 Reliability

We assume that p is the probability of failure over some amount of time and we are computing the probability of the system being connected after one such amount of time.

1. The system will become disconnected if one of the two wires fail. That means the failure probability is $Pr(X \leq 0)$, where $X \sim binom(2, p)$. That is, X is a binomial-distributed random variable with $n = 2$ and $p = p$.

2. In this case, the system will become disconnected if two of the three links fail, so the failure probability is $Pr(Y \leq 1)$ where $Y \sim binom(3, p)$.

3. We compute the two probabilities:

$$Pr(X \leq 0) = \binom{2}{0} 0.000001^0 (1 - 0.000001)^2 \approx 0.99999800$$

$$Pr(X \leq 1) = \sum_{i=0}^{1} \binom{3}{i} 0.0001^i (1 - 0.0001)^{(3-i)} \approx 0.99999997$$

From this we conclude that the town council should buy the low-reliability links.

2 ARIES

1. Here is the dirty page table computed in the analysis phase:

PageID	RecLSN
P2	3
P1	4
P5	5
P3	6

And here is the transaction table:

TransID	Status	LastLSN
T1	Active	4
T2	Active	9

2. The set of winner transactions is $\{T3\}$ since it is the only one that finished.

   The set of loser transactions is $\{T1, T2\}$ since these did not finish before the crash.

3. The redo phase starts at the minimum `recLSN` in the dirty page table. This means `LSN 3`.

   The undo phase ends at the oldest `LSN` of the transactions in the loser set. That would mean `LSN 3`, since that is the first `LSN` associated with `T1`.

4. The set of log records that may ause pages to be rewritten during the redo phase will consist of all `update` or `CLR` records after `LSN 3`, where the redo phase starts.

   This means that the set is $\{3, 4, 5, 6, 8, 9\}$.

5. The set of log records to undo is the set of updates of the loser transactions. That means LSNs $\{9, 8, 5, 4, 3\}$.

6. This is what is appended to the log after the recovery procedure is completed following a crash after `LSN 10`.

```
LSN   LAST_LSN   TRAN_ID   TYPE              undoNextLSN  PAGE_ID
---   --------   -------   ----              -----------  -------
11    9          T2        ABORT                          -
12    4          T1        ABORT                          -
13    11         T2        CLR: Undo LSN 9   8
14    13         T2        CLR: Undo LSN 8   5
15    14         T2        CLR: Undo LSN 5   -
16    15         T2        end
17    12         T1        CLR: Undo LSN 4   3
18    17         T1        CLR: Undo LSN 3   -
19    18         T1        end
```

3 Implementation

1. In our implementation, replicate requests from the master are handled synchronously, using the threadpool of the **CertainBookStoreReplicator** object. Each of the replication requests are handled by a **Callable CertainBookStoreReplicationTask** object, that simply calls the HTTP proxys replicate function. The proxy uses **performHTTPRequest** to ask the clients to replicate using a **REPLICATE** signal. Here we assume that outside actors can't send requests directly to our backend, since that would allow anyone to fabricate replicate requests.

   In **SlaveCertainBookStore**, the replication requests are multiplexed and the corresponding methods are called. Here, we make use of the fail-stop assumption: If anything goes wrong in the bookstore, it is going to fail immediately with no period of invalid state. Then, the system is going to recognize that the component has failed and can be restarted.

   We also implemented load balancing for incoming requests in **ReplicationAwareBookStoreHTTPProxy** and **ReplicationAwareStockManagerHTTPProxy**. We did so using a randomized method. Here we assume some amount of profiling work has been done, because we define a constant **EXP_PERCENT_WRITES**, which define the expected number of write requests compared to the total number of write requests to the service. Then, we compute

$$p_{master} = \frac{1}{\#slaves + 1} - EXP\_PERCENT\_WRITES$$

   This defines the share of read requests that the master unit should receive such that it has the same expected number of requests as the slaves. Then, if p_{master} is positive we pick the master with p_{master}. If we don't pick it, or if p_{master} is negative, we pick a slave unit uniformly at random.

This way, every unit has the same number of expected requests.

For our testing we added two tests: **testWriteFailureCanStillRead** where we cause the master to throw an exception and then ensure that the service still processes read requests, and **testSlaveErrorMasking**, where we test that a failure in a slave causes no change in availability.

2. The obvious advantage is that the read requests can be distriuted over an amount of servers, which should provide both a reliability and performance boost. Another advantage is that the scheme is simple. We know that write requests always are performed on the single master and then sent to each of the slaves. This means that synchronizing servers is a simple process. Compare this to the case where multiple servers perform writes synchronously. Then we would need a much more advanced replication scheme to ensure consistency.

   With regard to performance, we expect that performance should increase in a read-heavy workload, since this makes good use of parallellism. Write requests will still slow the system down and constitutes a bottleneck. Every write request still has to be handled on every server and then there is the added overhead of sending HTTP requests back and forth on the backend.

3. The client should simply state its latest timestamp seen. Then the proxy can make sure that a response is only sent to the client with a later timestamp. If this is not yet possible, it can delay the request.

   Here we make use of the assumption that timestamps consistently refer to the same states across all servers.

4. If a network partition seperated some slaves from the master, they would no longer be able to receive replicate requests. Then, the master would mark these servers as failed slaves and continue replicating to the slaves it can still see. The seperated slaves would then never update their state again.

   A client seeing this old state might conceivably at some point send illegal requests to the master, causing it to fail as well.