Principles of Computer System Design - Assignment 1

Thorbjørn Christensen
Steffen Karlsson
Kai Ejler Rasmussen

November 22, 2014

Exercises

Question 1: Fundamental Abstractions

1. We split the address used in READ and WRITE into a group of X least significant bits and a group of $Bits_{max} - X$ bits. The most significant group will refer to a machine using a name resolution service, in the client, and the least significant group will refer to a memory address on the specified server. Each server is added sequentially in the bit domain to allow for a sequential memory domain. The scalability of this solution depends on the distribution between the two groups in the naming. A possible distribution in a 64 bit domain could be $2^{56_{bits}}$ memory addresses and $2^{8_{bits}}$ possible machines.

   This design, if configured correctly, will automatically change server when the address rolls over in the most significant group.

2. The READ and WRITE API are both using a name resolution service (ie. lookup table or other service) to resolve the ip of the receiving server. The value is then read or written using RPC.

```
1 READ(name)
2     ip <- lookupServer(name & SERVER_BITMASK)
3     memory <- name & MEMORY_BITMASK
4     return getValueFromServer(ip, memory)
```

```
1 WRITE(name, value)
2     ip <- lookupServer(name & SERVER_BITMASK)
3     memory <- name & MEMORY_BITMASK
4     writeValueToServer(ip, memory, value)
```

3. The READ/WRITE API of the abstraction layer should be atomic to keep consistency, since generic READ/WRITE operations are atomic. To achieve this, a lock is obtained during the operations on the remote machine.

4. The design allows for dynamic joins and leaves as long as the sequence is intact. If the servers 1,2 and 3 is available, a fourth server can be added as 00000100 (big-endian) in a byte to allow for roll over. The design does, however, not allow for serves to be added or removed out of sequence. For example removing server 2 gives the invalid sequence $1, 3, 4$. To accommodate for this, a centralized service layer could be added to reallocate memory addresses to similar to how bad blocks are reallocated on persistent storage.

Question 2: Techniques for Performance

1. Concurrency can reduce latency by splitting a request into several subtasks. In theory this speeds up the processing by a factor of n, where n is the number of subtasks in parallel. However, there is still some overhead due to the splitting and serialization of collecting the results again which slows the concurrency. This overhead should be trivial for large requests compared to the performance gained.

   Concurrency increases performance if the task allows for it, however, it is difficult to implement and test concurrency. Concurrency can also decrease performance in cases where the overhead takes up more computation time than the actual request.

2. Batching removes overhead from several requests by grouping them and processing them within the same request. This can be used to group read/write request to harddrive and minimize the movement of the disk arm by reordering the requests.

   Dallying is the concept of delaying requests, thus increasing latency, for the chance that the request won't be needed. For example, delaying a write request expecting a newer request to overwrite it will allow for deletion of the first request. This is also known as write absorption. Removing requests in dallying should make up for the latency it imposes in order to increase performance.

3. Having few central servers can give increased latency around the world, however, introducing geographically close caching servers solves this. These caching servers will act as a fast path for the most common requests, but cache misses will cause the request to contact the central servers, ie. the slow path, through the caching servers.

Questions for Discussion on Architecture

Question 1

a. Its strongly modular because the server-side and the client packages are independent, their common logic is separated from the two main modules.

b. Taking a look at the two different client interfaces, namely `StockManager` and `BookStore` one might conclude that following - in a real life case - could be using the first interface `StockManager` for internal usages in the book company. Whereas the `BookStore` then most likely will be used for

outside clients who wish to either rate or buy a book etc.

The reason for separating such interface instead of one including all, is that you want to separate "business" logic from clients and for safety/privacy reasons - i.e. an costumer i following setup isn't able to modify state of a book.

c. Running the clients and services locally in the same JVM, forces the program to use the `CertainBookStore`, which both is a `StockManager` and a `BookStore` and therefor contains the functionality of all different interactions with the server. In this situation is the client and the server sharing the address space.

Question 2

a. The naming service is available throughout the `BookStoreHTTPMessageHandler`, which parses the requested api-call, extracts the name (one of following available in: `BookStoreMessageTag`) and executes the requested code.

b. The naming service the client is using to communicate with the server is in this case the operating system, since the host is hard-coded to localhost on a specific port. Localhost is then mapped to the ip-address 127.0.0.1, which HTTP uses for communication.

Question 3

All communication from client to server is implemented using the `SendAndRecv` method from the `BookStoreUtility`, this function secures an *at-most-once* RPC semantic. The function doesn't try to resend if any failure and therefore it is not a *at-least-once* semantic. In addition we doesn't have any tags on our requests and therefore can't secure that a request is send *exactly-once*.

Question 4

a. Since non of our `GET` requests have any side-effects, it is secure to use web proxies for this kind of architecture.

b. The only two places in this system where a web proxy would make sense, is behind the two clients, namely: `StockManager` an `BookStore`. Since the `StockManager` most likely is for internal usage, due to the API-calls exposed by the client, do we assume that the need for a web-proxy here isn't quite as high as the need behind the `BookStore` client, which is exposed to all the potential customers.

Question 5

a.

b.

Question 6

a.

b.