University of Copenhagen

PCSD: Assignment 1

Jens Fredskov (chw752)
Henrik Bendt (gwk553)
Ronni Lindsgaard (mxb392)

December 1, 2013

1 Fundamental Abstractions

We assume a global fixed block size (e.g. $4KB$), and that the Single Address Space is dynamically reserved (i.e. a non-fixed size).

The Single Address Space is a mapping from index to a block on a machine. This means that each machine can have a variable number of blocks. If a machine joins it is simply appended to the list (meaning that we extend the Single Address Space with the number of blocks provided by the new machine). If a machine leaves abruptly, the pointers in the Single Address Space is simply pointing to a null, meaning that a timeout will occur when we try to read or write to the blocks mapped to that machine. If a machine announces its departure the memory blocks of the machine are transferred to other free blocks on other machines. This implies that each entry in the mapping also contains a flag (of 1 bit) determining whether the block is in use and also that there are enough free blocks to transfer the used blocks of the machine leaving. Furthermore the old entry must now point to the entry of the new block.

We do not make any assumptions on the number of machines other than that the combined number of blocks must be less than or equal to the maximum size N we can hold in the memory.

The pseudocode for the API looks as follows:

```
function READ(addr)
    $(F, M, B) \leftarrow table[addr]$
    if $F = 1$ then
        $(V) \leftarrow$ FETCH$(M, B)$
        return $V$
    else
        return $empty$
    end if
end function
function WRITE(addr, val)
    $(F, B, M) \leftarrow table[addr]$
    SET$(M, B, val)$
    $table[addr] \leftarrow (1, M, B)$
end function
function FETCH(M, B)
    SEND$(M, (``fetch'', B))$
    SETTIMEOUT(1000)
    RECEIVE$(M, (res, val))$
    if timeout then
        return $nil$
    else if $res = ``error''$ then
        segfault
    else if $res = ``ok''$ then
        return $val$
    end if
end function
function SET(M, B, val)
    SEND$(M, (``set'', B, val))$
    SETTIMEOUT(1000)
    RECEIVE(M, res)
    if timeout then
        segfault
    else if $res = ``error''$ then
        segfault
    else if $res = ``ok''$ then
        return
    end if
end function
```

READ does a lookup on the address table and calls fetch on the block and machine, if the block is used, otherwise returns empty. Returns either the value, segfault or nil (in case of timeout).

WRITE does a lookup on the address table and calls set on the block and machine and given value. If set is successful, update the lookup table with write flag set to 1. Else return error.

Atomicity is ensured by having a two phased locking mechanism just above the READ/WRITE API, where the queue of requests are handled by some schedule. The WRITE API ensures atomicity by using TCP and timeout.

READ and WRITE operations against main memory are atomic. It would probably be safest to also have an atomic implementation against our memory abstraction, as the layer above might not know whether it is using regular main memory or the abstraction. To ensure atomicity we could e.g. use locks. The simplest implementation would be to have shared locks for reading and exclusive locks for writing. Of course this comes with the complications of locks such as deadlocks which needs to be handled in by following some protocol. Alternatively we could use versioning, which however might be difficult due to the possibly large memory that the abstraction could provide.

We have made no assumptions on the number of machines or the size of the memory each provides. The naming chosen should allow for dynamic joining and leaving, as machines are named consecutively as they arrive. If enough space is available when a machine leaves these should be transferred to other machines thus not making memory locations unavailable. Note however that the READ/WRITE API does not show this dynamic joining and leaving which is assumed to be located in other functions.

2 Techniques for Performance

An example using concurrency could be as follows: When a single core processor runs multiple threads concurrently, each thread is swapped when waiting for memory (context switch). Here it reduces the latency for all threads, as other threads can do their computations while the previous are waiting for I/O. If however the overhead of swapping threads exceeds the time waiting for memory, the concurrency becomes a latency factor for the thread waiting for memory which is already retrieved. Thus it is not always a positive nor a negative factor but can be a trade off.

Batching is used to group requests to avoid the individual overhead and instead get a maybe smaller group overhead. Batching could be used in an I/O bottleneck, where there is an overhead for each message send and where requests can be combined into one message.

Dallying is used to hold back requests in the hope that a later request cancels out the first one, thus removing having to do the first request at all. It can also be used to hold back specific requests in the hope of creating a batch. Dallying could be used in an I/O bottleneck to eliminate writes to the same space, so only the latest is written.

Caching is a fast path optimization because it tries to make a fast path to the most used data, reducing the average latency of accessing data.

3 Additional tests

topRatedBooks Two tests were added testing the behaviour of the method. They both run on a random input using the Random library. The first test checks that if the numBooks parameter is larger than the current number of books in the store an exception will be thrown. This is tested by taking the current number of books in the store, adding at least 1 to that number and pass it in as an argument. The second test works similar but the test ensures that exceptions are thrown when the input is less than 0. This is done by simply adding with -1 and subtracting 1 afterwards.

rateBook The tests added to testRateBook is similar as they are also based on random input. Ratings are generated in the same random manner as for getTopRatedBooks where we ensure that the parameter is either less than 0 or greater than 5. Furthermore we run a test (generalisation of the already implemented method) checking that all books match. There is an issue here, as the test method uses the .equals() method which is not to be trusted. It was however part of the handed out code. If it proves to be an issue, this will be fixed in future versions.

getBooksInDemand

- Tests if no books are in demand when no books are present on the server (and especially no books are in demand).
- Tests if no books are in demand when one book is present on the server (but no books are in demand).
- Tests the book just seen as in demand is still in demand with no other intermediate calls and is the only one in demand.
- Tests a book is still in demand after added another book which is not in demand.
- Tests if all of multiple books in demand are returned as in demand.

Do not test for when a book, which was just in demand, is not in demand anymore (after added copies).

4 Questions for Discussion on Architecture

1. The architecture achieves strong modularity by using RPC with not only functionality divided by client/server, but also by using a proxy and server class intervening the client and server. Thus if the clients fail, the server still stands and vice versa, because they only communicate via messages with RPC, that is non-locally (at least not in the same thread). Thus the same isolation is enforced on the same JVM, though the JVM can crash, which will halt all running services, instead of only the locally running service as when the services run on different machines.

2. The proxies themselves do not use a naming service in from the architecture. When a proxy is instantiated a must be given a string pointing to the address of the server, e.g “http://localhost:8081”. Thus the naming mechanisms used by the proxies are those of the underlying HTTP protocol. When a local server has been started it is also possible to use `getInstance()` which will return the interface for the given local server. Notice however that we then do it this way we do not utilize the proxies and HTTP handler part of the architecture, but work directly on the *CertainBookStore*.

3. The architecture implements at-most-once semantics by calling via the proxy just once. In case of error an exception is thrown, otherwise returns possible value or void. The proxy won’t try to call multiple times and will block for each call. It does not ensure a call neither, as it does not handle errors or try to correct or retry.

4. It should be safe to use proxy servers for the *BookStoreHTTPServer*-instances, as it issues thread-safe method calls to the CertainBookStore-instance and also the *CertainBookStore* uses atomic transactions. Also each proxy will be on its own, so if it crashes, no other proxy nor the server is affected.

5. Currently the architecture only employs a single *CertainBookStore* which needs to handle all requests at some point. This means that even with multiple *BookStoreHTTPServer*s we will at some point get blocked by waiting for the *CertainBookStore* as all its methods are thread-safe and therefore synchronized. If we have multiple instances of *CertainBookStore* running (handling the same logically server), then a bottleneck would appear when synchronizing each instance.

6. Clients would not experience failures differently by just using web proxies. Caching could however save some client calls, if the call is referring to some cached data. Thus these client calls would not experience the server crash.