Principles of Computer System Design
[0.3cm]
[0.4cm] **Assignment 1**

[0.4cm] Johannes de Fine Licht
Philip Graae

1 *

Question 1: Fundamental Abstractions

1.1 *

1.

An array of start/end addresses as consecutive integers mark the beginning of a new underlying machine memory. When given an address in the single address space, binary search is used to find the correct interval, which is mapped to a machine. The offset to the beginning of the target machine is subtracted from the single address space address to obtain the local, physical address. The map of addresses is stored centrally, but the mapping itself can be done from the caller, as they will have the address table stored locally. These will listen to the central node, and if a new machine is added, a message is sent from the central address map to all listening clients.
When an address has been translated, a message is sent to the machine owning the address in question. The caller waits for a response limited by a timeout period. If a response is received, the operation finishes, indicating whether the operation was successful or not to the caller. If a timeout occurs, the local entry of the target machine will be marked as offline, broadcasting this by a message to the central node, who will again broadcast the offline status to all other listeners. The central node is then responsible for periodically pinging any offline machines, returning their status to normal if a response is received and broadcasting this to listeners. All such messages will have a timestamp so the central node uses the most up-to-date information.
When we add new machines, the time it takes to do the binary search and determine which machine holds the given memory interval will increase, this will negatively affect scalability. This extra time should however be insignificant compared to the time it takes to send messages between the machines. Another consideration with regards to scalability, is that having more machines should on average decrease the amount of requests each machine has to handle and the gain from this should far outweigh the added time it takes to do the binary search. Scalability could also be improved by adding communication nodes between the central table and the callers, responsible for communicating change in state of the address table between the central node and the callers.

1.2 *

2.

```
// Caller-side
data read(address)
    entry = binarySearch(addressTable, address)
    localAddress = address - entry.offset
    requestRead(entry.machine, localAddress, TIMEOUT˙LIMIT)
    (response, timeout) = waitForResponse()
    if (timeout)
        reportOffline(entry.machine)
        throw exception
    if (response.status == ERROR)
        throw exception
    return response.data

// Machine-side
void handle(request)
    if (request.type == READ)
        dispatchReaderThread(request)
    else if (request.type == WRITE)
        dispatchWriterThread(request)
```

Translates the address by using the locally stored table, then sends a read-request to the machine mapped to the found entry. The machine listens for requests and dispatches reader/writer threads to service the request.

```
bool write(address, data)
    entry = binarySearch(addressTable, address)
    localAddress = address - entry.offset
    requestWrite(entry.machine, localAddress, data, TIMEOUT˙LIMIT)
    (response, timeout) = waitForResponse()
    if (timeout)
        reportOffline(entry.machine)
        throw exception
    if (response.status == ERROR)
        throw exception
```

Same as above.

1.3 *

3.

Operations against regular main memory on typical architectures are atomic when performed on aligned addresses of the native word size. For unaligned addresses or for operations of other sizes, read/write tearing can occur.

For our abstraction, atomicity should not be guaranteed by the memory model. Rather, an ownership concept should be introduced, keeping a table of memory intervals on the local machines owned by specific processes, denying access to any process that is not an owner. To achieve atomicity, we would need to implement locking. The lock would be implemented by a client sending a lock request for a given memory address to the node to which the address is mapped. The node will respond with a message acknowledging that the client now has the lock. Once the client receives this acknowledge message, he knows that any succeeding operations will be atomic with respect to any other client. To avoid deadlocks, we will need to have timeouts. The client should timeout if the lock acknowledgement never arrives. The node should timeout on the lock and send a message to the client that the lock has timed out if the client never sends any instructions after taking the lock, or if the client forgets to release the lock. We should reset the timeout timer every time the client sends a new instruction. By using timestamps in the messages, the client would be able to know which of his requests have not been carried out, because they were sent after the lock was released. If someone else already has the lock, the client should be put in queue and be notified that he is in a queue. The client can then choose to timeout on this wait if he wishes to. The client should then notify the node that he no longer wishes to take the lock. To avoid inter-client deadlocks, we would need to ensure that any client can only hold one lock at a time. To enforce this, we would need to send lock requests through the central table and the central table could then refuse to pass along lock requests if the client has not yet released a previous lock.

1.4 *

4.

Adding machines would be straightforward, as this would simply involve adding entries to the address table. The design allows replacing machines, but changing memory intervals would be a very expensive operation, requiring the memory of all machines to be shifted accordingly. If the intervals are not changed, swapping out machines is simply a question of updating the table entry and communicating this change to the network. If a machine goes down unexpectedly, then all data in that machines memory will of course be lost and immediately unavailable. The only real way to mediate this type of failure, would be to have redundant machines, so that we always have two copies of the data in the cluster, once the cluster becomes aware that a machine is down, the central table is updated so that all future requests are sent to the redundant machine. This would of course at least double the monetary cost of the implementation, or cut the capacity in half, depending on how you look at it.

2 *

Question 2: Techniques for Performance

2.1 *

1.

Concurrency typically reduces the average latency experienced from a system by facilitating multiple processes simultaneously, but adds overhead to individual processes and requests, increasing the minimal latency experienced. Overhead comes from synchronization/concurrency control between hardware threads or from bottlenecks in a concurrent pipeline.
Even in an embarassingly parallel problem, concurrency can have a negative impact through effects such as false sharing between processor cores. This is a consequence of the cache coherency protocol, in which different processors operating on the same cache line will invalidate the line and force inter-core synchronization. In extreme cases this can lead to negative scaling with the amount of processor cores of an otherwise entirely parallel problem.

2.2 *

2.

2.2.1 *

Batching The process of grouping a number of requests waiting to be processed and exploiting this for performance. This can come from sending multiple requests to the next stage of a pipeline, from reordering requests or even from vectorization of computations.

An example is when sending messages is an expensive operation in a bottleneck of a pipeline, and grouping several requests in a single message can increase the throughput.

2.2.2 *

Dallying Done by purposely delaying execution of requests waiting to be processed in order to exploit batching, or to completely eliminate requests that are invalided by others.
This can happen when a request with a later timestamp wants to write to the same location as a request with an earlier timestamp, allowing the earlier request to be discarded.

2.3 *

3.

Caching is an example of a fast path optimization, because it optimizes the most common scenario. In the context of CPU caches, this means loading more memory than requested and putting it close to the die, making subsequent accesses to the same or adjacent memory faster by orders of magnitude. If a subsequent memory access is done to a far-away location in memory, however, a resulting cache-miss not only triggers a slow load from memory, but also has to repopulate the cache.

3 *

Programming Task

3.1 *

rateBooks and getTopRatedBooks

The functions are implemented along with the message handling, and are tested in BookStoreTest. The tests include positive tests for rating books and retrieving top rated books based on these ratings, and negative tests that catch the exceptions thrown when invalid input is passed or the rated book doesn't exist.

3.2 *

getBooksInDemand

The function is implemented along with the message handling, and is tested in StoreManagerTest. This involves buying both available and unavailable books, then retrieving the books in demand and verifying that the correct books are in the list. Negative tests are done to verify that errors are reported for invalid input.
In addition a manual fix was done to testRemoveBooks which was reporting false positives.

4 *

Questions for Discussion on Architecture

4.1 *

1.

4.1.1 *

(a)

The architecture owes its modularity to the interface abstraction of the bookstore, allowing free substitution of the underlying classes as long as they conform to the interfaces.

4.1.2 *

(b)

The stock manager and the book store are interfaced by two distinct HTTP services, so separation of access can be achieved by only providing the relevant one. However, since a single handler processes all requests and no verification is done to the authenticity of the source, a third party knowing the message interface could easily gain access to all server-side functionality.

4.1.3 *

(c)

When running on the same JVM, the isolation is still intact, as the functionality is still interfaced through the two distinctions of access.

4.2 *

2.

4.2.1 *

(a)

There is a naming service in the passing and handling of messages, which are string representations of application-space operations.

4.2.2 *

(b)

Since communication between client and server happens over HTTP, the first naming mechanisms that requests will encounter is the DNS system, translating human-friendly domain names into actual IP addresses.

4.3 *

3.

The architecture employs at-most-once semantics, as no retries (at-least-once) or further communication (approaching exactly-once) is attempted on a failed request.

4.4 *

4.

4.4.1 *

(a)

Adding proxies is safe, as sending messages from multiple proxies is no different from sending multiple messages from a single proxy. The manipulation of data happens centrally, and does not give rise to data races.

4.4.2 *

(b)

Proxy servers can be placed between clients and the server handler service. This is possible for both the book store and the store manager, although the former is a more likely scenario.

4.5 *

5.

4.5.1 *

(a)

The scaling bottleneck will be the book store itself, as all methods are synchronized, and as such can only be managed by one thread on one machine.

4.5.2 *

(b)

Because all clients need to ultimately access the same book container, this will scale poorly with a large number of clients.

4.6 *

6.

4.6.1 *

(a)

When a proxy detects a timeout to the server, they could hold on to the request and try again later, or return a more useful error message to the client.

4.6.2 *

(b)

Caching could be used to service requests where it is not critical to receive the most up-to-date information, such as retrieving top-rated books, editor picks or books in general.

4.6.3 *

(c)

Web caching should not affect the semantics of the bookstore service, as all requests explicitly send messages to the server/proxy.