Principles of Computer System Design

Martin Grunbaum (martin@itsolveonline.net)

December 1, 2013

1 Introduction

In this paper we briefly consider a memory-layer abstraction to provide a large, single address space backed by a number of machines. We also explore thoughts on performance improvement techniques and finally reflect over a programming task in Java related to the concepts of Remote Procedure Calls.

2 A memory abstraction

The goal of the memory abstraction presented here is to provide an interface as close to or exactly the same as performing read/write operations on a single machine, despite actually being backed by a cluster of machines behind-the-scenes. This provides the obvious benefit that software and hardware that relies on this interface can continue to do so uninhibited, without requiring expensive or considerable alterations.

Given some varying number of machines, K, a global fixed block-size of $4KB$ for memory cells or blocks, and consequently a dynamic number of cells, from cell 0 to cell $N-1$, the idea is to intercept READ and WRITE calls and map them to the correct machines in the cluster, as well as handling the joining and leaving of machines. A machine being unavailable must not cause critical issues with the system.

Consider the following pseudocode for the READ and WRITE functions:

```
function READ(cell)
    if $cell \geq 0$ and $cell \leq lookup.maxIndex$ then
        $(Valid, Machine, Addr) \leftarrow lookup[cell]$
        if $Valid = 1$ then
            $Value \leftarrow$ FETCHVALUE$(Machine, Addr)$
            if $Value = None$ then
                return $None$
            else
                return $Value$
            end if
        else
            return $None$
        end if
    else
        return $None$
    end if
end function

function WRITE(cell, Value)
    if $cell \geq 0$ and $cell \leq lookup.maxIndex$ then
        $(Valid, Machine, Addr) \leftarrow lookup[cell]$
        if Valid then
            $Response \leftarrow$ STOREVALUE$(Machine, Addr, Value)$
            if $Response = Ok$ then
                $lookup[cell] \leftarrow (1, Machine, Addr)$
                return $Ok$
            else
                $lookup[cell] \leftarrow (0, Machine, Addr)$
                return $None$
            end if
        else
            return $None$
        end if
    else
        return $None$
    end if
end function
```

The two functions make use of helper methods StoreValue and FetchValue to actually transmit a message to the machine in question via TCP/IP, with an implied time-out mechanism resulting in the Send method setting `Ok` = `False`.

```
function FETCHVALUE(machine, addr)
    $(Ok, Value) \leftarrow$ SEND$(machine, (``getValue'', addr))$
    if $Ok = 1$ then
        return $Value$
    else
        return $None$
    end if
end function

function STOREVALUE(machine, addr, value)
    $(Ok, Value) \leftarrow$ SEND$(machine, (``storeValue'', addr, value))$
    if $Ok$ then
        return $Ok$
    else
        return $Timeout$
    end if
end function
```

It is assumed that the system also keep track of machines joining and leaving the cluster, and updates the available addresses in the lookup table appropriately. When a machine joins, the address space is expanded appropriately and the lookup table is updated. When a machine leaves or crashes, the addresses with data in them can be transferred to other machines in the lookup table (along with the data itself) if possible, and the entries related to the machine that has left will be removed from the table, shrinking it appropriately. There is some overhead in this process, but the underlying assumption is that machines joining and leaving will not be a frequent enough occurrence (say, several times a second) to be an actual issue.

The system has a centralized component in that there is a handler that intercepts reads and writes. In theory it would be possible to run multiple components, if one takes care to account for the new issues that arise with keeping state in sync or accounting for a mismatch of state between handlers. The system scales well with number of machines, as the lookup table grows or shrinks automatically as needed. The system can handle timeouts (a request is sent to a machine in the cluster, but never answered) as well as machines leaving both in a timely or untimely fashion, which will look as if they were timeouts.

Read and write coherence is assumed to follow the coherence of the underlying machines. Atomicity can be implemented through two-phase locking and *before-or-after* atomicity, as a higher-level construct on top of the memory abstraction. A waits-for graph should also be put into place, to handle deadlocks that may occur.

Normal memory *can* have atomic reads and writes, although this is typically implemented as special instructions at a higher level. The issue with atomicity is that being safer makes things slower, so it is always a balance between wanting to provide rapid access to memory and giving some kind of guarantee of correctness. One atomicity approach is *before-or-after atomicity*, which guarantees that the result of any READ or WRITE occurs before or completely after any other READ or WRITE.

3 Techniques for Performance improvement

3.1 Concurrency

Concurrency opens up the possibility of having multiple threads, CPUs and so on processing instructions at the exact same time. Intuitively, if we can run two instructions concurrently, and they do not interfere with each other, then we can probably *decrease* the latency of some set of instructions. However, concurrency does not simply scale infinitely in a positive way, and it can be deceptive at times. Having to account for the possibility of concurrency introduces increased general overhead, to make sure things happen in a way that is still sensible. Concurrent code must often make use of locks, which entail waiting until someone else is done with something.

One example of latency actually increasing with more concurrency, is if you run a large number of threads on tasks that do not take very long to complete. The overhead of performing context switching very often could very well exceed the gain from concurrent execution.

3.2 Dallying & Batching

Dallying and batching are both techniques used to increase performance. **Batching** means to queue up several requests locally, until a *batch* of requests are available to all send off at the same time. There are many things that are potentially useful about this: any one-time setup cost of a request is reduced from nxcost to just cost, it may be possible to re-order the requests to gain efficiency, it may be possible to simply remove some of the requests if they are redundant and more. One useful situation for this is a high-latency, high-bandwidth situation such as a satellite uplink. To make the most of the connection, it can be useful to send a large amount of data at a time.

Dallying means to *wait* with a request, in the hopes of improving **batching** or in the hopes of a future request removing the need for the current one. Write dallying is often called *write absorption*, for example, where a write to a specific variable or cell is dallied in the hopes that a new write will be requested to the same area/variable/cell.

3.3 Caching

Fast path optimization is about constructing the fastest route to a response, by observing common routes and perhaps reducing the cost of those routes, at the cost of improving the cost of other routes.

Caching fits this paradigm well, as you store a fast path to recently fetched results in the hopes that they will be requested again. There are a myriad of caching strategies that can be employed depending on the situation, and caches can be employed both server-side and client-side. Modern browsers will often cache the result of HTTP requests if possible, and employ strategies that allow the browser to simply ask for resources that have changed since the last time.

4 Architecture discussion of the programming task

Modularity is an important property in many aspects of programming and computer hardware. In its essence, modularity in IT is about designing components in such a way as to split them up into separate *modules*. An immediately useful way to take advantage of this is to separate the concerns of each module, also called *separation of concerns*. In a service-oriented architecture with clients and services, modular design is often about separating clients and services and being able to support many different kinds of clients without many different underlying implementations.

An implementation that may have begun as a local service, can be stubbed out and represented in the same manner from the point of view of the client, even though it actually makes calls to a remote service. Remote procedure calls (**RPC** calls) are one way to do this, letting procedure calls happen remotely instead (as the name suggests). This has the added benefit of isolating the memory, CPU and computing resources of the client and server-side, as a protective measure.

RPC calls are often coupled with a semantic related to consistency, such as at-least-once or at-most-once. The implementation provides no guarantee of delivery outside of what TCP/IP provides, which in this case means an at-most-once RPC setup. If the request times out, expires or is in any other way failed, an exception is simply thrown.

The current solution hardcodes a server URL into the client-side HTTP code, which is a very inflexible way of approaching things. Normally, to scale to a larger number of client requests, clients would hit one of several load-balancers, possibly after a DNS round-robin has already forwarded the request. The load-balancer would then pick an HTTP proxy to serve the request, based on a load-balancing strategy such as round-robin or least-load. In the current solution, the HTTP proxies would be backed by the same Singleton model, which would then be a choke-point. Normally, a database with better support for concurrency will back a model, allowing a much more scalable solution than having to rely on method-locking via the `synchronized` keyword in Java.

Suppose that the server hosting the `CertainBookStore` Singleton crashes. Clients would then, in the current architecture, fail to reach the server and experience exceptions. Using web proxies, its normal to provide appropriate failure messages, but from a programming point-of-view those are scarcely different than the current situation. However, if web proxies were to employ a caching strategy that cached response-objects based on request data, a failure could be masked for the objects already cached.

5 Concluding remarks on tests

The tests that came with the codebase commit several cardinal sins: they test more than one thing, they instantiate a large part of the state as a part of the method, they are very long and unwieldy, they do not test exceptions the right way and more.

Several new sets of tests have been developed under the `com.acertainbookstore.xxx.tests` packages, although I was not able to finish all of them in time. The `CertainBookStore` singleton was fixed to actually work as a proper singleton in Java should, a clear method was added to its API in order to facilitate testing and integration tests were separated from internal tests. Other changes were made as well, mostly minor cosmetic ones to the API. One particular change helped facilitate testing a great deal: the separation of URL path handling and model API stubbing **out** of the HTTP message handler. It now properly makes use of a class to take input and craft the appropriate responses, which allow the responses to be tested separately from the URL path handling, in theory.

More such decoupling should be applied to the codebase, for example to `CertainBookStore` which should have its model component and functionality separated and interfaced so that it can be mocked properly for testing. The singleton aspect of the server should be provided by a different class, so that the core functionality itself would not have to rely on a gimmick like the `clear` method I had to add. See for example how server-side state is managed for the tests in `BookStoreHTTPProxyTest`.

As a final note, tests should be done on a separate test instance that do not interfere with anything live. Ideally the HTTP tests should live on the server, by spawning HTTP clients to run the tests against a locally mocked up model that is discarded between each test.