Assignment 1

Yiran Zhang and Nicolai Jørgensen

November 27, 2017

1 Question 1

1. In order to organise memory with physical storage on several machines, we are going to split the top-level memory into pages. Additionally, we will maintain a map between the top-level pages and a tuple of machine identifier and local address. The mapping can be implemented to run in $O(logn)$ time, with n being the number of allocated pages, which can be kept small by choosing suitably large page sizes.

   In addition, we will for each machine maintain a list of its pages and the number of pages it has allocated. The length of the list can be used to load balance the system, by evenly splitting the memory between available machines. When a machine leaves the system, all of its pages can be copied to prevent data loss.

2. In the pseudocode below, we assume calls to address directly into the big contiguous memory space. That is, there is no virtual memory addressing going on.

```
READ(addr):
  (PageNo, Offset) = (addr / page_size, addr % page_size)
  if (exists(page_map, PageNo)):
    (Machine, MachineAddr) = lookup(page_map, PageNo)
    v = RemoteREAD(Machine, MachineAddr + Offset)
    if      v == segfault: return segfault
    else if v == timeout:  try again a set number of times, if still no success, retur
    else:                  return v
  else:
    return segfault

WRITE(addr, value):
  (PageNo, Offset) <- (addr / page_size, addr % page_size)
  if (exists(page_map, PageNo)):
    (Machine, MachineAddr) = lookup(page_map, PageNo)
    return RemoteWRITE(Machine, MachineAddr + Offset, value)
  else:
    if system has space for a new page:
      (Machine, MachineAddr) = allocate_new_page(PageNo),
      add(page_map, PageNo, (Machine, MachineAddr))
      RemoteWRITE(Machine, MachineAddr + Offset, value)
    else:
      return segfault

RemoteREAD(Machine, Addr):
  SEND(Machine, \{ READ, Addr \})
  RECEIVE(Machine, Value)
  On timeout:  return timeout
  else:        return Value

RemoteWRITE(Machine, Addr, Value):
```


Yiran Zhang
Nicolai Jørgensen

```
    SEND(Machine, \{ WRITE, Addr, Value \})

allocate_new_page(PageNo):
  find machine with least pages (O(logn))
  try to allocate page:
  on fail:
    remove machine from list of available machines for allocation
    allocate_new_page(PageNo)
  on success:
    return (machine, allocated page addr)
```

The page numbers and offsets are calculated using simple integer division and modulo. The functions `lookup`, `get` and `add` refer to a map structure with $O(logn)$ running times implemented with e.g a binary search tree.

3. We believe that memory access against the unified memory space need not necessarily be atomic. However, memory access to addresses in individual machines still need to preserve this basic integrity. That is, we should be able to distribute the memory access computation between machines.

4. Our name mapping strategy makes an assumption about the system setup. We assume that we know the addresses, and thus also the quantity, of machines in the unified memory space. Then, we use that information to dynamically spread allocated pages over the machines.

   Our system also allows for dynamic leaves and joins of machines in the memory space. Joining is simple, we just inform the system that a new machine is available with no pages allocated yet. Leaving is a bit more complicated, but can be done by iterating over the pages it had allocated. Each page should be allocated on and copied unto another machine. The leave operation should inform the system if some data could not be copied.

2 Question 2

1. Concurrency may influence latency positively or negatively, depending on where and how it is applied. If a task can be split into multiple independent parts, computing each of the parts concurrently can reduce the overall processing time and thus the latency of the system. Similarly, if a system receives independent requests from a number of clients, then having extra processing units will decrease the average latency of a request. Concurrency may not always result in a latency improvement. Parallelizing a program is not free. There is an amount of overhead that is incurred when the program has to coordinate its subordinate threads. Similarly, some threads may stall for periods of time waiting for intermediate results not yet computed.

   Concurrency may provide a latency boost, but there are individual considerations to make none the less. These are complexity of programming, applicability of concurrency to the problem and overhead incurred.

2. Batching is the process of bundling several transactions or messages into a single one in order to reduce the overhead. Batching naturally arises in program bottlenecks, where the requests will tend to pile up. An example is memory access to the hard disc. Memory access is really slow, and requests might pile up while another access is being processed. Batching similar requests together will reduce the overhead of sending individual requests back and forth.

   Dallying is a strategy for handling requests, which consists of speculatively delaying the processing of a request. If many requests accumulate through dallying they can be batched together, or the result of the request might not be needed after all. In the case of memory access from before, non-critical requests may be delayed until a batch process can take care of many at once. Another example of dallying is lazy evaluation in Haskell. Computations are delayed until such a point that their values are actually needed. This allows programmers to make outrageous requests such as infinitely recursive data structures without looping forever.

3. Here I will assume that by caching is meant the memory system optimization that utilizes fast memory hardware to mask the latency of hard disc storage.

   Caching is indeed an example of a fast path optimization. Fast path optimization refers to designing system to make it fast in the common case. The concept of "locality of reference" occurs almost naturally when designing programs. We are specifically referring to temporal and spatial locality, which means that if one memory address is accessed, it and its neighbours are more likely to be accessed in the near future. Caching the entire page of an address makes lookups to these addresses faster, thus optimizing the common case.

   If by caching was meant web browsers storing recently visited web pages, then it is indeed a fast path optimization in almost the same way. Users will often want to go back in their history to look at a recent page. Having it already loaded then avoids having to do an expensive and redundant network exchange with the web server.

Caching is indeed an example of a fast path optimization. Fast path optimization refers to designing system to make it fast in the common case. The concept of "locality of reference" occurs almost naturally when designing programs. We are specifically referring to temporal and spatial locality, which means that if one memory address is accessed, it and its neighbours are more likely to be accessed in the near future. Caching the entire page of an address makes lookups to these addresses faster, thus optimizing the common case.

3 Questions for Discussion on Architecture

1. a) All-or-nothing semantics means that if a transaction fails at any point, no part of the transaction data is saved. For example, in `TestRateInvalidISBN()`, we test that an illegal rating and a valid rating together makes no change to state.

   b) For the `rateBooks` we write the following tests: that a single valid rating is processed correctly, that multiple ratings accumulate on a book, that books

cannot be rated if ISBN is invalid, that books cannot be rated if a rating is invalid and that trying to rate a book not in the store causes an error.

For the `getTopRatedBooks` we write the following tests: that books can not be got if K is not valid and that a valid K can be processed correctly.

For the `getBooksIndemand` we test that books in demand can be retrieved.

2. a) The architecture is modular in the sense that there is a clear seperation between client code, server code and the communication layer between them. Each of these modules can be upgraded or completely replaced without the rest of the program needing changes.

   b) The architecture isolates the clients from the backend code by way of an intervening communication layer. Thus, clients and service can only speak to each other through RPCs.

   For all intents and purposes have seperate state and environment. This is provided by the strong modularity. For example, the service could fail and the clients would still run and preserve their own state.

   c) The same kind of isolation is not enforced. Once the JVM breaks down, both the server and the clients will crash, as can be done in the tests.

3. a) Yes, there is naming system, it binds the names with the address or resource. By specifying the name we can get the resource, the service address and the provider information. Therefore, clients can interact with a service through names.

   b) We use IP address to allows the clients to discover and communicate with servers.

4. At-most-once semantics is implemented in the architecture. Using at-most-once the RPCs will either return a result or some error. In our implementation, when the RPC succeeds it will return the result, otherwise it will throw an exception. The implementation of the backend logic ensures that no state changes happens when a call fails.

5. Yes, it is safe to use web proxy servers. We can encrypt the communication between the proxy server and the clients and service to protect against man-in-the-middle attacks.

   The proxies should be deployed between the software proxies that handle message sending and the server that receives and handles the HTTP requests. Notice on the diagram that these are exactly the messages sent using HTTP. These messages are the boundary between the clients and the service.

   As a conservative measure, the proxy servers should probably only be used for non-critical requests. For example, buying books from a web proxy when the main service is down would not be ideal.

6. Yes there is scalability bottlenecks in this architecture with respect to the number of clients. A large number of clients could easily overload our single backend server.

7. Yes, the use of proxies between clients and the server would change the ways clients might experience failures.

   First of all, there might be difference in the time where clients are notified about the crash, because of e.g different processing times/communication times.

   If the web proxies not only did forwarding and communication handling but also caching, then errors might be effectively masked. In case of RPCs that merely retrieve data, the proxies could simply serve their cached state to the clients.

   In case of RPCs with consequences, such as rating books, the proxies could store requests in a queue to be processed when the service comes back online. This is of course only a good idea for some systems, specifically where the service can be quickly restarted. The clients should obviously somehow be informed about this.

   Other than that, using caching in web proxies would mean that the clients lose the guarantee that the data they are retrieving is the actual service state. They might be served old data and the requests they send might be delayed.