

# Distributed Systems: Midterm exam

Summer 2010

May 31, 2010

- This is an open-book, open-notes exam. The use of networked electronic devices is not allowed.
- Please answer all four questions. The weight of each question is indicated on the exam.
- Please be concise in your answers. You will receive partial credit for partially correct answers, but extraneous remarks may be held against you.

**Name:**

**Student id:**

1. MapReduce [20 points]

Ben Bitdiddle has access to a 10000-node cluster for running various MapReduce jobs. This cluster comprises 5000 type A machines and 5000 type B machines, where type A machines are substantially slower than type B machines.

After using the cluster for some time, Ben realizes that some but not all of his MapReduce jobs have high completion times and a low per-machine average CPU utilization.

**i. [10 points]** Explain why this performance anomaly could be occurring. In particular, describe the characteristics of the jobs for which this problem might occur.

**ii. [10 points]** Describe a change to the MapReduce scheduler described in the paper (i.e., how the master task schedules worker tasks) to address the problem above. Your change should solve the problem from the previous question, and preserve the completion time for the MapReduce jobs for which the previous problem does not arise.

2. DSM and consistency [35 points]

Inspired by the distributed shared memory (DSM) system described by Li and Hudak in “Memory Conherence in Shared Memory Systems”, Ben decides to take a drastically different approach to implementing the extent service. In particular, he decides to directly use DSM as an extent service where each extent is treated like a DSM page. For his DSM-based extent service, Ben uses the centralized manager algorithm described in Section 3.2. (We have included this section in the last page of this exam.) In his implementation, each yfs client acts as a DSM node and there is a separate manager node that keeps track of the ownership of pages (i.e. extents) among different yfs clients.

**i. [10 points]** Ben is happy that his new implementation passes the Lab 3 tests, which involve reading, writing and sharing files among multiple yfs clients sequentially. Since DSM implements sequential consistency, Ben argues that there is no need to use the lock service at all for his Lab 4 implementation. Will he be able to pass Lab 4 tests? Why or why not?

**ii. [10 points]** After running his system for several days, Ben starts to notice that some of the nodes become very slow on occasion, and this affects the performance of writers because of the need to invalidate pages that are cached at slow readers. Ben wants to improve the performance of such writers by avoiding that they have to invalidate copies at readers, but he still wants readers to gain some advantage of caching (even if the performance gains from caching are less than in the original design). He also notices that readers are bandwidth-limited: the overhead of remote reads is dominated by the time to transfer large memory pages. Suggest a modification to the DSM protocol in Section 3.2 that addresses Ben’s concerns while maintaining sequential consistency. Describe the main changes to the pseudocode in Section 3.2.

**iii. [10 points]** After running the refined system for a while, Ben realizes that clients sometimes cannot reach the node that runs the ownership manager, leaving them unable to perform any writes, even on data extents that they have locally cached. (Depending on your answer to the previous question reads could also be blocked, but we can ignore that fact for now.) He would like to fix this by allowing such disconnected clients to perform writes speculatively. (Such speculative writes may subsequently fail; an application whose speculative writes eventually fail will have to be rolled back to a point prior to the first failed write.) The system should still be sequentially consistent to all clients whose writes do not fail. Describe how Ben can change his DSM protocol from the previous question (2.ii) to support speculative writes. (If you didn't answer the previous question you can consider the original DSM-based solution from 2.i.)

iv. Ben runs the following program on a computer with two processors and a single shared memory.

CPU0:

```
x = 1;
if (y == 0)
    foo();
```

CPU1:

```
y = 1;
if (x == 0)
    foo();
```

How many times can foo run when the shared memory system implements the following consistency models (justify your answer):

(a) Sequential consistency [**2.5 points**].

(b) Release consistency [**2.5 points**] (note that the program does not use locks).

3. Remote Procedure Call [25 points]

Ben Bitdiddle believes that at-most-once RPC is not necessary for implementing the extent service. Ben uses the naive RPC library in Lab 1 with retransmission enabled, but without all the machinery that implements at-most-once execution. He executes this following sequence of code to store and retrieve the extent with key `eid`: (You should assume there is only one yfs client active in the system.)

```
//cl is the rpcc object for communicating with the extent server
ret = cl->call(extent_protocol::put,eid,"aaa",r);
assert(ret == extent_protocol::OK);
ret = cl->call(extent_protocol::put,eid,"bbb",r);
assert(ret == extent_protocol::OK);
ret = cl->call(extent_protocol::get,eid,buf);
assert(ret == extent_protocol::OK);
cout << buf << "\n";
```

i. [4 points] What is the *expected* content of `buf` at the end of the above sequence of code? (You can assume the sequence of code is the only RPC client active in the system.)

ii. [6 points] What are the potential *unexpected* contents of `buf`? (Recall that the network may lose, duplicate or reorder packets. Furthermore, the naive RPC library simply retransmits any request for which it has not received any reply.) Draw a timing diagram to explain each unexpected content of `buf`. (Your timing diagram should contain a time line for both the client and the server as well as all the messages exchanged between them.) Can these unexpected outcomes occur with a RPC library that implements at-most-once delivery? Why?

Ben proposes to implement at-most-once RPC by having the RPC server keep an in-memory *replay* buffer (`replaybuf`). In particular, each RPC client generates a random 64-bit number as the RPC request identifier. Since the RPC identifier space is fairly large ( $2^{64}$ ), we can safely assume that the identifiers generated by all RPC clients are unique. The RPC server remembers each RPC request that it has seen in the `replaybuf`. If a received RPC request is already in the `replaybuf`, then the RPC server treats it as a duplicate request. In order to prevent the `replaybuf` from growing without bound, Ben also removes all entries that are present in the `replaybuf` for longer than  $t$  seconds based on the `first_seen` field. Ben's RPC client implementation is the same as his Lab1's naive implementation with retransmission enabled. The pseudocode for his RPC server implementation is as follows (locking is omitted, but you should assume Ben performs locking correctly):

```

struct rpc_entry {
    struct timeval first_seen;
    bool rep_present;
    marshall rep;
    rpc_entry() {
        gettimeofday(&first_seen, NULL);
        rep_present = false;
    }
};

void rpcs::dispatch(...)
{
    ...
    //replaybuf is a hash map of rpc_entry
    //reqid is the 64-bit identifier associated with the RPC request
    if (replaybuf.find(reqid) != replaybuf.end()) { //replaybuf contains reqid
        if (replaybuf[reqid].rep_present) {
            return replaybuf[reqid].rep to the RPC client
        } else {
            do nothing
        }
    } else {
        replaybuf[reqid] = rpc_entry();
        execute the corresponding RPC handler
        add reply to replaybuf[reqid].rep, set replaybuf[reqid].rep_present to true
    }
}

//executed periodically in a separate thread
void rpcs::expiretimer(...)
{
    foreach repid in replaybuf {
        if ((now - replaybuf[repid].first_seen) > t)
            remove repid entry from replaybuf
    }
}

```

iii. [5 points] Does Bens RPC implementation always guarantee at-most-once execution? If your answer is no, please give a specific counterexample.

iv. [10 points] Suppose the amount of time it takes the network to deliver a packet in one way is bounded by  $\delta$  seconds (in case the packet is not lost). Furthermore, assume the clock drift between any pair of arbitrary machines is at most  $\epsilon$  seconds. In other words, if machine M1 observes that  $x1$  seconds have passed based on its local clock, then any other machine (e.g. M2) will observe that  $x2$  seconds have passed according to M2's local clock such that  $x1 - \epsilon \leq x2 \leq x1 + \epsilon$ . Under these two assumptions, describe how Ben's replaybuf-based RPC library can be made to guarantee at-most-once delivery in the face of lost, reordered, duplicate packets **and** client or server failures. (Your solution should still have retransmissions to deal with occasional losses and it should avoid writing to disk).

4. Logging [20 points] Ben Bitdiddle decides to store the content of his extent server on the local file system of the extent server. He intends to take advantage of the logging capabilities of the local file system to be able to recover from crashes of the extent server. For simplicity, assume that each individual file system operation is atomic with respect to crashes (unlike FSD, which we studied in class, where data operations like `read` and `write` were not logged).

Ben's RPC handlers for the extent server are as follows:

```
int extent_server::put(extent_protocol::extentid_t id, std::string buf, int &)
{
    //id2filename(id) converts the extent id into a unique filename deterministically.
    string f = id2filename(id);
    int fd = open(f.c_str(), O_CREAT|O_TRUNC|O_WRONLY); // creates new file or
                                                    // truncates length to zero

    if (fd < 0) return extent_protocol::IOERR;
    int r = write(fd, buf.c_str(),buf.size());
    close(fd);
    if (r >= 0)
        return extent_protocol::OK;
    else
        return extent_protocol::IOERR;
}

int extent_server::get(extent_protocol::extentid_t id, std::string &buf)
{
    string f = id2filename(id);
    int fd = open(f.c_str(), O_RDONLY);
    if (fd < 0) return extent_protocol::NOENT;
    char *p = new char[MAX_EXTENT_SIZE];
    int n = read(fd, p, MAX_EXTENT_SIZE);
    if (n >= 0) {
        buf = string(p, n);
        return extent_protocol::OK;
    } else {
        return extent_protocol::IOERR;
    }
}
```

- i. [5 points] Ben uses his yfs client to create two empty files named `aaa` and `bbb` one after another in the root directory. Unexpectedly, Ben's extent server process crashes in the middle of creating `bbb` (but the OS keeps running). When Ben restarts his extent server, Ben sees that the recovered root directory is empty. Explain what happened, and describe the possible contents of the last few entries in the file system log upon recovery. (You can assume that there is only one yfs client active in the system for all questions in this section.)

**ii. [5 points]** Describe a simple change that Ben could do to the implementation of the file system to fix this anomaly. Note that the modification may change the semantics of file system operations but not their interface, and only requires a simple change to the information that is logged.

**iii. [5 points]** Ben failed to modify the complex file system codebase and is still frustrated by these anomalies. He asks his friend Alyssa P. Hacker for help. Alyssa remembers that the local file system operation `rename(const char *oldpath, const char *newpath)` is an atomic operation, i.e. if a failure occurs in the middle of this operation, the rename operation appears to either have happened or not at all upon recovery. Perform Alyssa's fix on Ben's code to ensure that Ben's yfs can recover correctly from an extent server failure. You can directly modify Ben's code on the previous page.

iv. [5 points] Does Alyssa's fix ensure that Ben's yfs can always recover correctly during the failure of the extent server process during *all* yfs operations? Explain.