Principles of Computer System Design (PCSD), Block 2, 2012 / 2013

Assignment 3

This assignment is due via Absalon on January 8, 23:55pm. This assignment is going to be evaluated pass-fail, as announced in the course description. While individual hand-ins will be accepted, we strongly recommend that this assignment be solved in groups of two students. Groups may at a maximum include three students.

A well-formed solution to this assignment should include a PDF file with answers to all exercises as well as questions posed in the programming part of the assignment. In addition, you must submit your code along with your written solution. Evaluation of the assignment will take both into consideration.

Do not get hung up in a single question. It is best to make an effort on every question and write down all your solutions than to get a perfect solution for only one or two of the questions.

Note that all homework assignments have to be submitted via Absalon in electronic format. It is your responsibility to make sure in time that the upload of your files succeeds. While it is allowed to submit scanned PDF files of your homework assignments, we strongly suggest composing the assignment using a text editor or LaTeX and creating a PDF file for submission. Email or paper submissions will not be accepted.

Exercises

Question 1 (Communication abstractions). [Tanenbaum, partial] In the following, you will show how to implement different communication abstractions based on other abstractions:

(a) Suppose that you could make use of only transient asynchronous communication primitives, including only an asynchronous receive primitive. How would you implement primitives for transient *synchronous* communication? Show the pseudocode of your solution.

(b) Suppose that you could make use of only transient synchronous communication primitives. How would you implement primitives for transient *asynchronous* communication? Show the pseudocode of your solution.

(c) Explain how you would implement *persistent asynchronous* communication by means of RPCs. Show the APIs for the main system components and how these components call each other (you do not need to show detailed pseudocode).

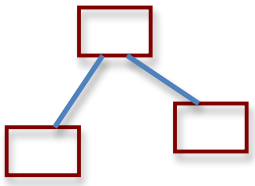
Question 2 (End-to-End Principle). Please argue how the following system functionality exemplifies the end-to-end principle:

1. Exokernel.
2. Encrypted data transmission (e.g., SSL).

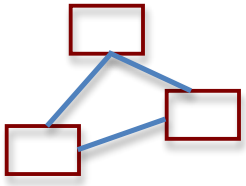
Study the design of these systems and write one paragraph for each, in your own words, explaining how the end-to-end principle motivated their design. In particular, for each system, discuss at least one alternative design that violates the end-to-end principle and explain the trade-offs. If you use the Web or other mediums to get information, remember to cite your sources.

Question 3 (Reliability). [Saltzer & Kaashoek] The town council wants to implement a municipal network to connect the local area networks in the library, the town hall, and the school. They want to minimize the chance that any building is completely disconnected from the others. They are considering two network topologies:

1. "Daisy Chain":



2. "Fully connected":



Each link in the network has a failure probability of p.

(a) What is the probability that the daisy chain network is connecting all the buildings?
(b) What is the probability that the fully connected network is connecting all the buildings?
(c) The town council has a limited budget, with which it can buy either a daisy chain network with two high reliability links (p = .000001), or a fully connected network with three low-reliability links (p = .0001). Which should they purchase?

Question 4 (Distributed Transactions). [Coulouris] A three-phase commit protocol has the following parts:

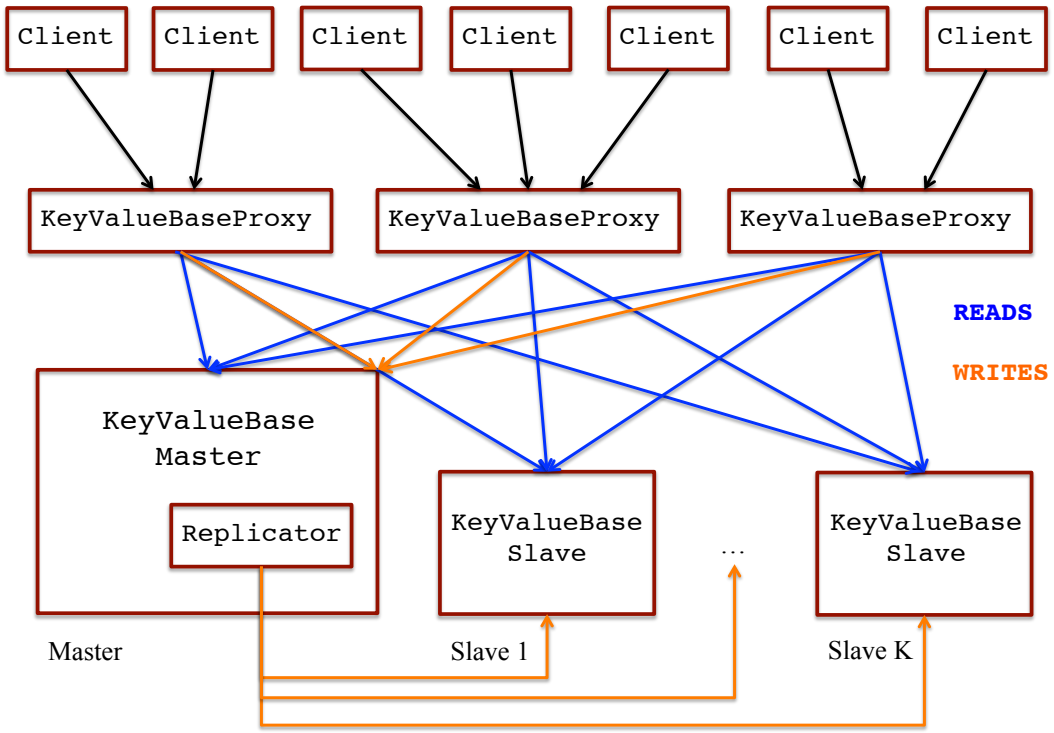
> *Phase 1:* is the same as for two-phase commit.
>
> *Phase* 2: the coordinator collects the votes and makes a decision; if it is *No,* it *aborts* and informs participants that voted *Yes;* if the decision is *Yes,* it sends a *preCommit* request to all the participants. Participants that voted *Yes* wait for a *preCommit* or *doAbort* request. They acknowledge *preCommit* requests and carry out *doAbort* requests.
>
> *Phase* 3: the coordinator collects the acknowledgments. When all are received, it *Commits* and sends *doCommit* to the participants. Participants wait for a *doCommit* request. When it arrives they *Commit.*

Explain how this protocol avoids delay to participants during their 'uncertain' period due to the failure of the coordinator or other participants. Assume that communication does not fail.

Programming Task

In this programming task, you will add a replication mechanism to `KeyValueBase`, targeting scalability and availability for read operations. We will adopt a model of a single master instance, with interface `KeyValueBaseMaster`, which orders all write operations and replicates them to a set of slave instances, with interface `KeyValueBaseSlave`. The master-slave configuration is hidden by a proxy component, with interface `KeyValueBaseProxy`, which exposes the traditional `KeyValueBase` interface to clients and offers additional configuration management functions to administrators of the service.

The following figure summarizes the replication mechanism you will implement:



Master

The master instance (`KeyValueBaseMaster`) is essentially the `KeyValueBase` implementation you have developed over the previous two assignments; however, one important component to be added to that implementation is a `Replicator` component. The interface of the `Replicator` is very similar to the `Logger` interface introduced in Assignment 2:

- `makeStable(LogRecord record)` → `FutureLogAck`: This function accepts a log record corresponding to a write operation and returns a *future* to its caller. The future is only populated after the corresponding log record has been synchronously applied to all slaves of

`KeyValueBaseMaster`. Every slave implements the `KeyValueBaseSlave` interface, which offers a `logApply(LogRecord record)` function. The calling code of `KeyValueBaseMaster` is responsible for guaranteeing atomicity by blocking on the future after executing the operation's logic. In other words, the code only returns any results to the client after the write has been applied to all replicas.

The `Replicator` supersedes the `Logger` from your previous assignment. If durability is to be provided for the master, then the `Replicator` is also responsible for writing the log record to stable storage *before* forwarding it to the slaves. In the solution to this assignment, you may choose to either *ignore durability altogether* or just extend the code you already had for Assignment 2.

One important issue is how to deal with failures during replication. In this assignment, you may assume a *fail-stop* model: Network partitions *do not occur*, and failures can be *reliably detected*. We model the situation of replicas hosted in a single datacenter, and a service configured so that network timeouts can be taken to imply that the replica being contacted indeed failed. In other words, the situations that the replica was just overloaded and could not respond in time, or that the replica was not reachable due to a network outage are just assumed *not to occur* in our scenario.

Note the implications of this model: A write only completes when it is successfully applied to every live replica in the system. The fail-stop model precludes that replicas would be failed but still live. As reads cannot go to failed replicas, we need to ensure that reads serialize either before or after a given write. This property will be offered to the client by the proxy is attached to, with the restrictions described further below.

Question 1: *Briefly explain the changes you needed to implement to your original version of* `KeyValueBase` *in order to obtain an implementation for* `KeyValueBaseMaster`. *(1 paragraph)*

Slaves

The slave instances (`KeyValueBaseSlave`) implement the read operations of `KeyValueBase`, as well as the `logApply` operation above called by the `Replicator`. In contrast to the pure interface of `KeyValueBase`, every read operation in `KeyValueBaseSlave` returns not only the result of the read, but also a timestamp (*LSN*) corresponding to the last log record applied in the replica. These timestamps are used by the proxies to avoid stale reads.

Question 2: *Briefly explain the changes you needed to implement to your original version of* `KeyValueBase` *in order to obtain an implementation for* `KeyValueBaseSlave`. *(1 paragraph)*

Proxies

The proxies expose the well-known `KeyValueBase` interface to clients. Proxies maintain no state other than the system configuration (i.e., the locations of the master and slave instances, as well as their last observed status). The failure model only admits fail-stop failures, so that means that each proxy will independently detect the failure of either master or any of the slaves the next time that proxy attempts to send a request to that component of the system. This strategy simplifies configuration management: The same configuration is loaded in all proxies when they start, and then each proxy detects failures as needed. The statuses stored in different proxies for system components may differ, but that will never be observed by clients under the failures that are admissible.

Each proxy will load balance read operations across all instances, and ensure that writes are always applied to the master instance. Due to load balancing, two read operations for the same key sent to different replicas may return out-of-order results if a write is currently being replicated. The proxy hides this situation from the client by ensuring reads respect timestamp order, e.g., by re-issuing the offending read to other replicas, or to the master itself.

You may assume that clients always employ the *same proxy* to access the system. If a proxy fails in the middle of processing a write from the client, then the client would need to be *restarted* with another proxy. You do not need to implement client recovery for this assignment; in addition, simply assume that client and proxy *share fate* in your setup[1].

Question 3: *Briefly explain your implementation of* `KeyValueBaseProxy`. *(1 paragraph)*

Initialization and Configuration

Other than exposing the `KeyValueBase` interface, the `KeyValueBaseProxy` interface offers a method to initialize the system configuration:

- `config(Configuration conf)`: This function accepts a configuration (i.e., the locations of the master and slave instances) and initializes all instances in the configuration to active status. The function is only called when the proxy is first started, and should return an appropriate exception if called a second time.

The same function is also offered by `KeyValueBaseMaster`, and is also to be called once during system startup.

After configuration, a client may call the usual `init` function of `KeyValueBase`. The proxy receiving the call will forward the call to the master, which in turn invokes the call also at the slaves. For simplicity, assume that the initialization file loaded by `init` is available in a shared filesystem readable by all instances (or alternatively, that it is also replicated in the same path in every instance).

For this assignment, you *do not* need to implement *recovery* from failures. If a slave fails, then the master and all the proxies which attempt to communicate with that slave will reliably detect the failure and flag the slave appropriately as failed in the configuration. No more operations will be then forwarded to that slave. However, note that your system must *continue to operate* and be available for reads and writes even in the case of slave failure. If the master fails, the system becomes unavailable for writes, but remains available for reads as long as there are slaves available. Finally, if all instances fail, the system becomes unavailable.

Question 4: *Suppose an administrator of the system decides that their cloud provider is too expensive, and wants to migrate the service to another cloud provider. Even though reconfiguration is not available for this system, how would a clever administrator generate a sequence of controlled failures that would allow for reconfiguration and migration of the service, while generating minimal disruption to the clients of the service? Explain. (2 paragraphs)*

Question 5: *Explain what would happen to your implementation of the system if a network partition would separate a subset of the replicas from the master, while still allowing those replicas to be contacted by the proxies.*

---

[1] For more on fate-sharing, see Section 4 of D. Clark, "The Design Philosophy of the DARPA Internet Protocols", SIGCOMM 1988, available at: http://ccr.sigcomm.org/archive/1995/jan95/ccr-9501-clark.pdf

Experiments

Now that we have made `KeyValueBase` scalable and available for readers, we should experimentally quantify how well our design fares in these dimensions.

Question 6: *Design, describe the setup for, and execute an experiment that shows how the read throughput of the service scales with the addition of more slaves. Remember to thoroughly document your setup.*

Question 7: *Design, describe the setup for, and execute an experiment to show that the service is able to sustain the loss of slaves without disruption (but perhaps with degradation in throughput). Remember to thoroughly document your setup.*