CS-UY 3254 — Project Phase 4

Jeff Epstein

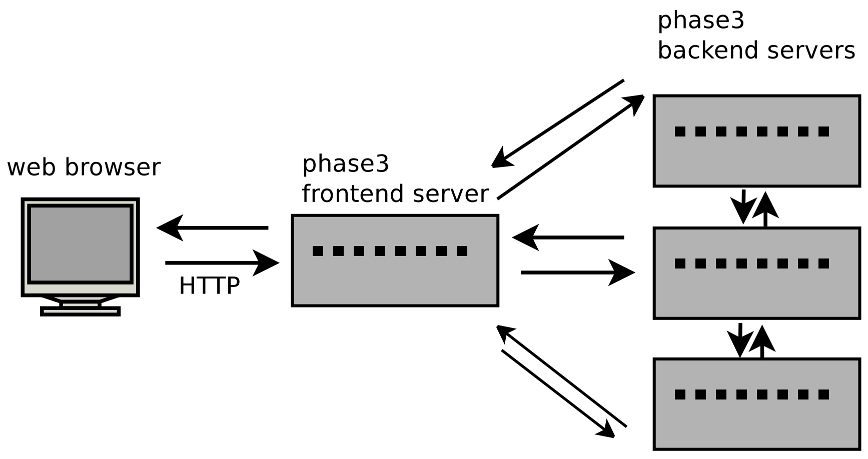
1 Introduction

This semester, we will implement an ongoing programming project, consisting of four phases. You must successfully complete each phase before advancing to the next. At the end of the semester, you will have created a reliable, redundant, distributed application.

We continue with the fourth phase of the project. I assume that you have successfully completed the previous phase; if that's not the case, please contact me to discuss your options.

2 Assignment

In this phase, we're going to add a replicated data store. The data stored in the back end must be replicated across an arbitrary number of replicas.



As in the previous phase, the front end will communicate with the back end. However, you have flexibility in how you implement the replication and how you implement the communication between the front end and the back end. Although not shown in the diagram, of course your application should also work with multiple front end instances.

2.1 Requirements

The requirements for this phase build on the requirements for the previous phase. In addition:

1. You can use any replication strategy that we discussed in class. If you prefer a well-known, publicized replication strategy that we did not discuss in class, you can probably use it, but please get your professor's permission first. Please do not attempt to improvise or "roll your own" replication strategy.

2. The command line arguments accepted by the front end and back end are augmented to allow multiple back ends. The `--listen` argument required by both the front end and the back end is still required. In addition:

   - The value of the front end's `--backend` argument, used to indicate the endpoint of the back end, will now be interpreted to be a comma-separated list of endpoints, rather than just a single endpoint. If the hostname is omitted in an endpoint, assume the local host. All the back ends' endpoints will be passed to the front end this way.
   - The back end must accept a non-optional `--backend` argument, whose value is also a comma-separated list of endpoints. If the hostname is omitted in an endpoint, assume the local host. Each back end will be passed a list of all *other* back ends, i.e. excluding itself. Depending on what replication strategy you are using, your back ends may not need to communicate with each other directly, but your back end is still required to accept this argument, even if it just ignores it.

   In all cases, you may assume that the configuration passed to your application on the command line is correct. If your application receives a consistent configuration, it should work. On the other hand, if the user (for example, your professor) gives your application an inconsistent configuration (for example, if back end instance A is given the endpoint of back end instance B, but back end instance B is not given the endpoint of back end instance A), then you are not responsible for the results.

   Here is an example of how your program might be started in a configuration of three back ends:

```
$ ./backend --listen 8090 --backend :8091,:8092 &  # start backend 1
$ ./backend --listen 8091 --backend :8090,:8092 &  # start backend 2
$ ./backend --listen 8092 --backend :8090,:8091 &  # start backend 3
$ ./frontend --listen 8080 --backend :8090,:8091,:8092 # start frontend
```

   This is a consistent configuration: all back ends know about each other, and the front end knows about all back ends. Notice that the front end is passed a list of *all* back ends, while each back end is passed a list of *other* back ends. In this case, all back ends and the front end are running on a single host. The back ends are listening on ports in the range 8090 – 8092, and the front end is accepting web requests on port 8080. In the example above, hostnames are omitted, but your program should work even if they are specified.

   You may assume that the front end is started after all back ends have been started (that is, your back ends may wait for an "all clear" message from the front end before beginning to synchronize with each other). You may assume that your application will be configured with at an odd number of back end replicas, and at least three.

   If the number of working replicas falls below quorum, your system may reject requests until the quorum is recovered.

   You do not need to design your application to support reconfiguration. That is, once started, the number of nodes in the cluster will not change. A failed node does not mean that it has been removed from the cluster. Therefore, the number of nodes in quorum will not change.

   In addition to the command line arguments required here, you may add additional mandatory command line arguments to your back end, as needed. For example, if you want each back end to have a unique identifier, you might want to add a `--id` flag, so that the unique identifier can be passed in as part of the configuration.

   As in previous iterations of the project, your program may not store data on disk, nor may data be stored in the front end. All data should be stored in memory-based data structures in the back end.

   In your `README` file, you must provide detailed and specific information about how to start your application. Please provide all commands.

Test cases Please test the following cases:

1. Given a cluster consisting of n back end replicas, your program should withstand failure of up to $\left\lfloor \frac{n-1}{2} \right\rfloor$ replicas. Your program will be tested as follows: some number of back ends will be started, then data will be entered through the web interface, then one or more of the back ends will be forcibly terminated, simulating a crash. The web interface should remain responsive and the data previously entered should remain available, as long as a quorum of back ends remain functioning.
2. After terminating up to $\left\lfloor \frac{n-1}{2} \right\rfloor$ replicas, some of them will be restarted, using the same command that was used to start them initially. The newly restarted replicas should synchronize data with the cluster, so that all running back ends have the same data.
3. After restarting all previously terminated replicas, disjoint sets of back end replicas will be forcibly terminated and restarted, until all replicas have been terminated and restarted. Your application should continue working normally during this process.
4. Next, more than $\left\lfloor \frac{n-1}{2} \right\rfloor$ replicas will be forcibly terminated simultaneously. At that point, writes initiated from the front end should fail.
5. Terminated replicas will be restarted, at which point they should synchronize and normal operation of the cluster should resume. A brief delay is acceptable while replicas synchronize.

3 Rules

The same rules pertaining to operating system, language, libraries, academic honesty, and code quality from the previous phase apply equally to this phase. Please refer to the handout for that phase for specifics.

4 Hints

- I recommend reading papers describing the algorithms you plan to use. Here are some references:
  - Paxos
    - The Part-Time Parliament
    - Paxos Made Simple
    - Paxos Made Moderately Complex
    - Paxos Made Practical
  - Raft
    - In Search of an Understandable Consensus Algorithm
    - The Secret Lives of Data
  - Viewstamped replication
    - Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems
    - Viewstamped Replication Revisited
- In the diagram on the first page, the arrows represent *possible* communication channels. Depending on what strategy you use, you may or may not decide to allow direct communication between replicas; and you may or may not decide to allow direct communication between the front end and all the back ends. You will, necessarily, have to allow communication between the front end and at least one of the back ends.
- As in the previous phase, I recommend testing your program using NYU's Vital cloud infrastructure.

- You can re-use the failure detector that you wrote in phase 3. Depending on what replication strategy you use, you may want to use similar code to allow a replica to detect the failure of another replica.
- I recommend careful consideration of a replication strategy before you start to code.
- Many distributed consensus algorithms are described in terms of message-passing. For example, "node W sends message X to node Y, which responds with message Z." Implicit in this algorithm is the assumption that while node W is waiting for a response from node Y, it must continue to service and respond to incoming messages from other sources. Therefore, a blocking wait (i.e. waiting in such a way that the application stops responding) is not possible.

  One possible implementation strategy is to model each node as a finite state machine. That is, in each node, some set of global variables will indicate which messages are expected and what action is to be taken in each case. As each received message is processed in the receive loop, the state of these variables is updated. The loop must not block: each message is processed as soon as it is received.

  Another implementation strategy is to insert all received messages into a per-node message queue, immediately upon receipt. The queue is shared between threads on that node, so that multiple threads may wait for incoming messages simultaneously. In this implementation, you need to be careful to ensure that each thread removes from the message queue only those messages that it processes, allowing other threads to continue to process messages pertinent to them.

  In your `README`, describe how your application avoids blocking while waiting for messages. I recommend addressing the general problem of message handling *before* attempting to implement a concrete replication algorithm. Using a conventional blocking RPC mechanism is clearly inappropriate.
- Start by designing the structures representing messages. Keep in mind that each messages will probably need to contain the ID of the node sending it (so it can get a response).

5 Submission

You are obligated to write a `README` file and submit it with your assignment. The `README` should be a plain text file (not a PDF file and certainly not a Word file) containing the following information:

- Your name.
- Instructions for building, running, and testing your program.
- The state of your work. Did you complete the assignment? If not, what is missing? If your assignment is incomplete or has known bugs, I prefer that students let me know, rather than let me discover these deficiencies on my own.
- Any other resources you may have used in developing your program.
- Document your program's resilience. For each of the test cases described above,
  1. indicate if your program passes it, to the best of your knowledge.
  2. describe specifically how your program handles the situation. What messages are passed? What can go wrong?
- Explain any important design decisions you made in completing this assignment. In particular:
  1. What replication strategy do you choose? Why did you choose it? How does it suit your application?
  2. Weigh the pros and cons of your replication strategy.
  3. How does your application avoid blocking while waiting to receive messages?

Submit your work on Gradescope. Remember to submit all source files necessary to build and run your project. Do not submit external library code (including Iris). Do *not* submit binary executable files: please use the `go clean` command to remove unnecessary object files before submission. If your project consists of more than one source file, or includes additional external data files (such as HTML files), submit them all together in a zip file that preserves the relevant directory structure.