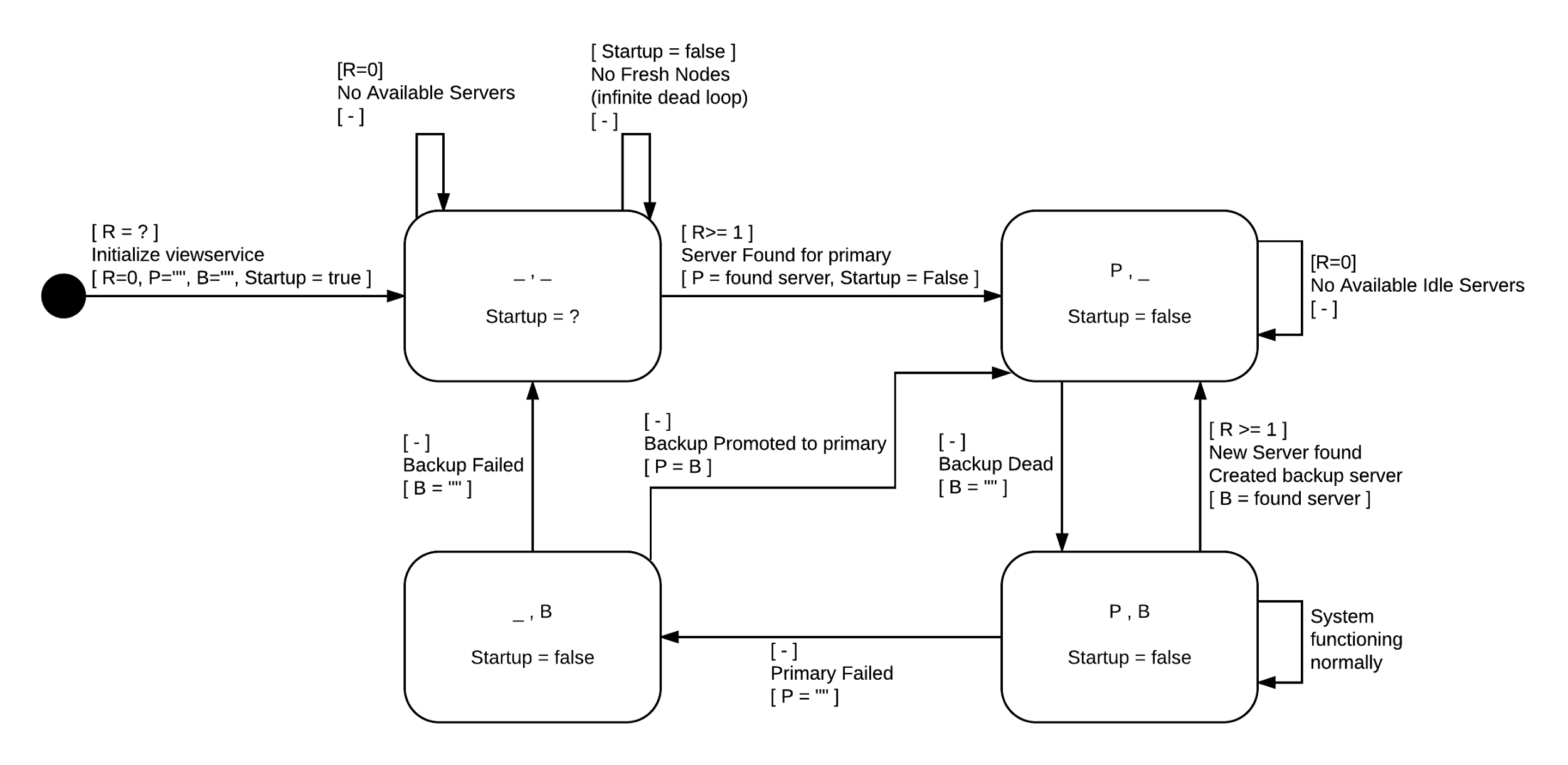
CSC 462 - Part 2 Writeup

Jakob Roberts - v00484900

The following diagram is the state transition diagram for the Viewservice module. The service has 4 different states and can potentially reach an infinite loop of a single state if certain parameters are met and there are too many failures. This diagram does not completely describe the implementation of the PBservice. Although the PBservice uses Viewservice nodes to function, except it employs a datastore for a key and a value that remain consistent with failures across nodes.



I would have liked to have had an exact idea of how we would have been tested as a whole prior to even beginning the Viewservice implementation. If the Viewservice implementation could be optimized to work with the PBservice implementation, the testing would have likely gone much smoother. I have been unfortunately failing on the 3rd to last test, but I believe some further debugging and modifications to my Viewservice will rectify those issues. The tests failed for me on the 3rd to last test “*Test: Repeated failures/restarts with concurrent updates to same key; unreliable …* “ and it is due to a backup synchronization failure due to not being able to connect to a peer, and therefore is likely a problem with the viewservice not checking if a node is dead properly. I will have to spend more time to work on this and correct the issue.

I didn’t change the test suite at all, but I would include a tree/state diagram representation of the tests in the future/make that as part of the assignment specifications. The test cases are what I was pulling my information from when it came to understanding what challenges had to be completed within the scope of the assignment.

As always, as far as design tradeoffs go, I would like to see a system that is implemented with no master and all the nodes use a sort of (I don’t know how to call it) Seagull approach?<see <https://www.youtube.com/watch?v=p-3e0EkvIEM>> If all the nodes played a kind of first come first serve and listed to a majority approach when it came to conflicting information, I believe the system could run quite quickly with some delays to the front end user updates as the information propagates. If the “POST” of data failed due to a conflict with another “POST” and that post overruled the user’s input, an error would have to occur to the user. Once the “POST” is made, it should be save as “Passed” until it may receive a “Failed” status with information, in which the user can re-input a variation of the data or retry the request.

In this type of implementation, any client can access any node at any time and access the data that it has, when accessed, the node says “Hey, I have a customer” and gets itself a backup just in case it dies so that the user can still make their query. I guess that this now connected node is a pseudo-master for just that user. When the request is made, it is propagated to all nodes and has a return route to the user’s actively connected node in case a failure occurs. Regardless of the user’s connected node, the user will still see a failed attempt in the logs and will be able to take further action to retry their request.

I don’t know of a practical application for this, but I believe that it could be easily implemented. It may also be very amusing to have lots of demanding servers that are constantly yelling at each other when a new piece of data is presented to them!

I would like to see a project that we can choose to make an implementation with a master process or an implementation without a master process. I feel like we are restricting the possibilities by assuming that a master must be included in the design implementation.