For our final project, we made the decision to implement an AP distributed system. We felt that the AP system would be the most robust in terms of dealing with partition failures and would have a higher throughput than a CP system. We wanted the client to be able to write to any node they chose and not experience any kind of hangups. In addition we wanted our system to be eventually consistent. We also wanted to attempt to implement a system that would allow our nodes to keep track of deleted values and prevent them working their way back into node stores.

Our team decided to build all of our projects in Node.js which with is many modules allowed us to abstract a large portion of tedious code and focus more on building the overall structure of the project. Each node communicates with each other via http requests and a restful API we implemented with the Node.js package Express. Our approach was to have each node be able to accept reads and writes and then make a good faith attempt to alert the other nodes on the system if a value in the store was changed. Every node keeps a dynamic list of nodes that are connected and those that are not. When a node crashes every other node will eventually be alerted to its state. If and when a crashed node comes back on line it will attempt to contact all members of the system, if it can it will begin to update it’s stores with the connected nodes in an attempt to remain consistent with the system as a whole.

When a node receives a write, it goes through its list of connected nodes and passes on the PUT request so all nodes have a chance to update their stores with the most recent write. Every node in the system sends out and expects a heartbeat from every other node. When this heartbeat message cannot connect we assume the node is crashed and add it to the list of dead nodes. If a node comes back to life or a partition is lifted, that node will continue sending its heartbeat. If any node receives a heartbeat from a node on its dead list, the two nodes will compare their stores and update values based on the most recent timestamp. In the case that timestamps are the same the nodes compare each other’s unique ID, the bigger ID number’s value will be selected. Finally when a client decides to delete a value, instead of completely deleting the value right away, we replace the value with a Boolean false. This way when nodes attempt to update their stores they can see that a value has been deleted. Otherwise a node with an old value could keep reintroducing its existence back into the system.