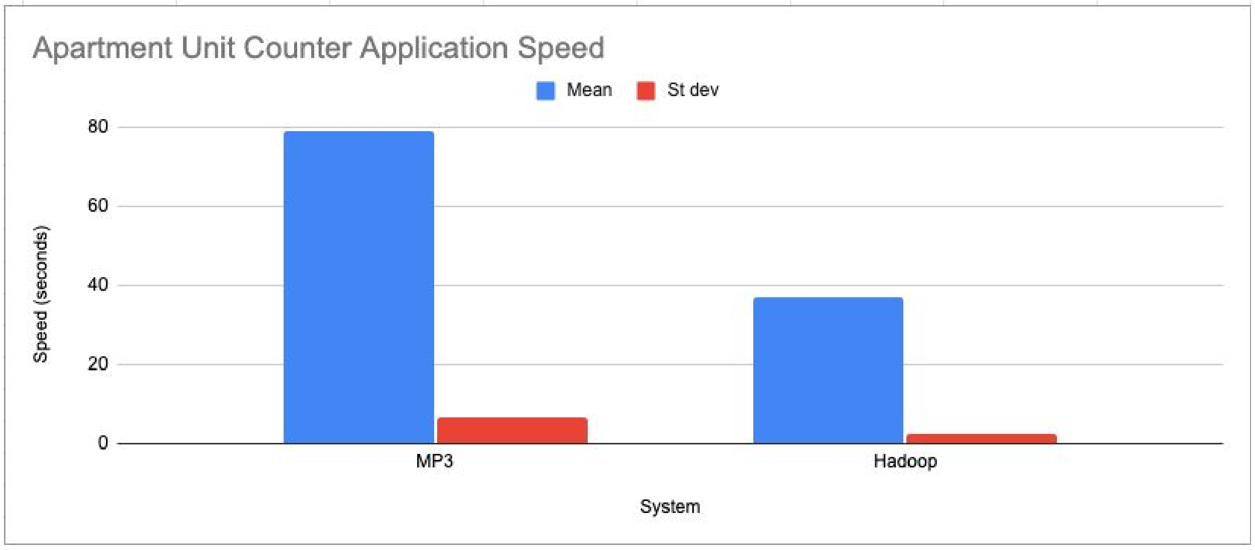
Michael Mallon (mmallon3) & Robert Krokos (rkroko2)

For this MP, we built upon the existing system we built in python3 for MP2. This system is based on having a handful of applications running inside a controller, which we refer to as the "node manager". The node manager holds an instance of the failure detector, the SDFS, and the MapleJuice system. Then, we use callback functions to transfer data between the applications (file lists, membership lists, etc). We wrote two different applications for MapleJuice, a master node class and a worker node class. The elected master is the same as the SDFS master, and the rest of the nodes are workers.

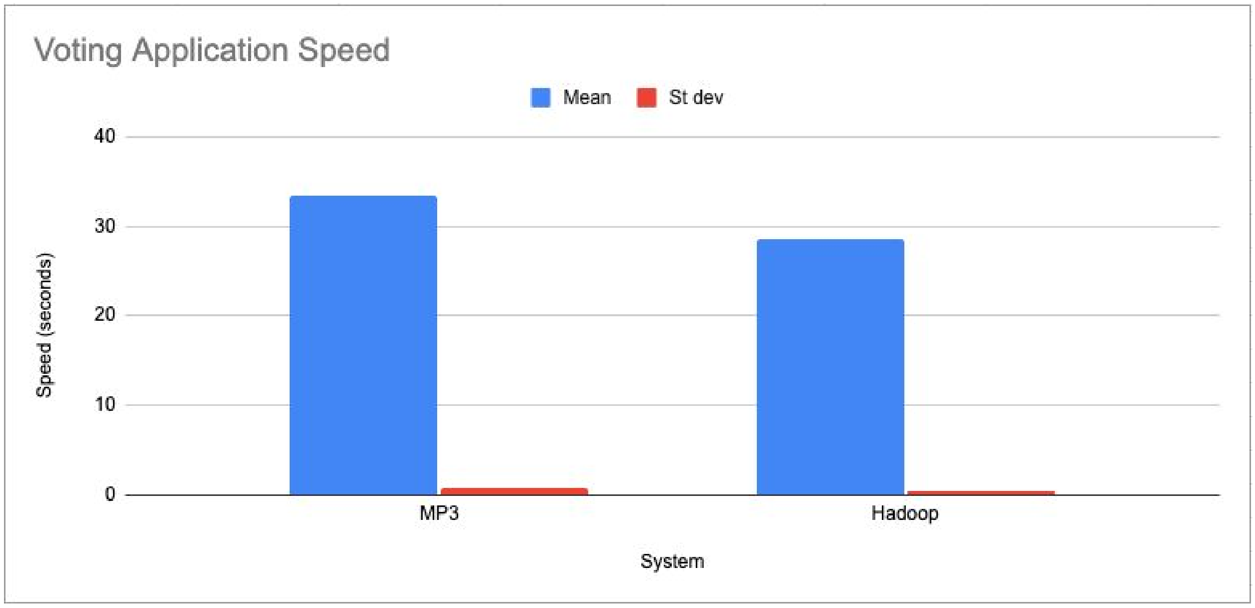
After the system is started, the master node will be waiting for maple/juice requests. Once a request is received, it is added to an "operation queue". We have an additional thread on the master called the "queue handler", which will read the metadata for each request and handle it. This ensures that we only handle a single maple/juice request at a time.

To process a maple/juice request we have 3 staged: split, process, combine. Each operation corresponds to a "target node", which is the node that sent the initial request. If this node happens to fail during/before the operation, then the target node will be switched to any available node. Once the master processes a request, it will have the target node split up the input files and select a set of nodes to process the files with the specified exe. The master node will send the details for this request to all the nodes, and wait for all of them to complete. If a node fails, the master uses a "work table" to keep track of all work assigned to that node, and can reassign the failed node's work to a different node. Each node has their own set of files they write to, and the target node will combine these into a single set of files during the combine phase.

To run the maple/juice functions, we have the user send over a python file that follows a specific API. The API is used for setting the actual maple/juice functions, specifying input format, and generating test files. Once a node is in the processing stage, it will pull the exe from the SDFS and run the file by importing it as a python module. Our 2 applications were the consensus application from HW1, and an app that sees how many apartments with >30 units are in the Champaign database.



In the graph shown above, we had our system run the apartment unit counter application against the Hadoop cluster we configured. In this test, we found that Hadoop has a much higher performance than our system. We believe this has to do with how we transfer data throughout our cluster. To allow for even work distribution throughout the cluster, our MP will open each input file and divide it into small 25 line files. This is quick when there is a small amount of data throughput, but for a large file like this database we ended up with ~120 files. Since our system can only process one read/write at a time, our nodes likely spend a lot of time waiting to obtain data from the SDFS. This also means there will be ~1000 reads and writes to the SDFS, which leaves a lot of room for latency to build up. Hadoop moves around a significantly smaller amount of files, which means most of the time is spent processing the data.



In the graph shown above, we had our system run the voting ranking counter application against the Hadoop cluster. Though our system didn't beat Hadoop, it came a lot closer than the last test. This is likely due to the fact that this test is done on a smaller dataset, so there is a lot less time spent transferring files around. This test used 2 chained mapreduce operations, so it is much more focused on processing speed vs high data throughput.

If we were to change anything to speed up our system, we would opt for larger file chunks. This would mean there is a lot less time spent uploading and downloading files. Another way to make the system significantly faster would be to allow for concurrent reads and writes to the SDFS. Since we can only perform a single read/write operation at a time, our performance is significantly bottlenecked.