Bosse/Carns Identity Server Part II White Paper

Client-server model:
One shot clients with separate servers allows us to keep our information protected from other clients because data is not stored on the same devices as the clients. The separate server will give us more control over the server pool, while allowing us to keep our client lightweight and easy to distribute. It will also prevent the need for a way to dynamically add servents to our pool, and build into our pre-existing server/client model from the last project, with less issues down the road.

Coordinators:
We'll be using the bully algorithm to elect coordinators, taking advantage of Node's socket.io module's multicasting features and using periodic heartbeat messages to the coordinator (also using sockets) to tell if the coordinator is still running.

Consistency Model:
We are choosing to use causal consistency as our consistency model because we believe it to be both achievable and applicable to our problem domain. With causal consistency, all operations performed on the data store by a single client are related, which fits nicely to the problem of an identity server - if a client creates a user and then updates their login name, those operations should be seen in the same order by all other clients, and the operations should be propagated to data store replicas in that order. Likewise, if two separate clients concurrently attempt to create a user with the same login name (which is not allowed), those operations should be ordered according to the distributed system's accepted logical clock, and the clock will decide which user's creation attempt is rejected. To implement our consistency model, we'll use vector clocks in each of our server processes to capture causality of operations on our data store. However, for part II of the identity server project, MongoDB will be able to do all the heavy-lifting for us regarding our consistency model through its client sessions.

Server Synchronization & Data Redundancy:
In the previous project, we chose to use MongoDB as our data store. The benefits of this decision start to show as we begin to make our data storage more redundant. MongoDB supports a distributed data set out of the box. We will begin by spinning up a group of mongod instances that are known as a replica set. Every replica set has one replica set primary, which is the mongod instance that receives all of the write operations. This server applies these writes to its oplog, which the secondary replica sets then apply to their data as well. The secondary replica sets are eventually consistent. However, using write concern we can ensure that a response is not sent from mongo to our server until it has been replicate to *n* replica sets. This allows us to determine how up to date our secondary replica sets are in reference to the write operations, in the scenario our primary mongo node goes down. This will allow us to keep our data redundant and synchronized across multiple nodes, effectively making our servers a middleware, of sorts.