# Distributed Systems Project

## Distributed Key Value storage

# Key decisions for Values

- 3:1 reserved internal connections to external
- Multicast immediate on request to all replicas
  - Voting based response: mode of n responses
- On storage to replica failure, replica pointer is increased by the replication factor to avoid replica duplication
  - i.e., the 3rd replica will always map to a server location divisible by 3.
  - If the mode of responses was determined as acceptable, this process can be done after the client has already been serviced, leading to an increased response time before replication was ensured
- Epidemic protocol based service to attempt to synchronize server status
  - Server offline? If we know they had a replica of a key, offer it back into the system.
  - Server hotjoin? They will get outvoted until they brought up to date

# Storage and internals

- Due to the innate structure of Key Value communication, JSON formatting was used to store and send data.
- JSON is an increasingly popular industry standard format, similar to XML, and optimized for Key Value pairs. (An associative array)
  - Excellent integration with Java, and the same protocol could be used with any other programming language in human readable format
  - Allows string requests to be sent to the server via any raw network connection
  - Adding a new parameter, such as the intended location of the replica, is as simple as accumulating under the key.

# Extra Functionality & API

- Each node has a webserver that is human readable, excellent for debugging
- “One button” PHP script to restart the entire service
- Simple API:

  Send: { “put”:true, “key”:”cats”, “value”:”meow” }

  System puts: { “cats”:”meow” }

  Response: { “ErrorInfo”:”Success!”, “ErrorCode”:0 }

- Data internals get implicit structure
- *“Reading a JSON document into Perl or Ruby or Python or Javascript is trivial because it maps directly to native variables. You just slurp it in.”*
  - Schwern, Stack Overflow

# Example Full Clients

**Java:**

```
String makeRequest(  String request,  String url,  int port  ){
        Socket TCP_socket = new Socket( url, port );
        new PrintWriter ( TCP_socket.getOutputStream(), true ).println( request );
        String response = new BufferedReader(new InputStreamReader(TCP_socket.getInputStream()) ).readLine();
        TCP_socket.close();
        return response;
}
```

**Unix:**

```
$ echo request | nc url port > response
```

# Future Considerations

- Epidemic protocol: need to consider progress when many servers offline
  - O(logn) approaches O(n) for average case
- Ex: If there are two servers remaining out of 100, what are the odds that one server discovers the other has just died?
  - Possible solution: prioritize efforts to poll alive server
  - Problem: What if the server isn't actually dead, just a network hiccup (it won't be able to announce it's alive, it never thought it was dead)
- Java is great for high level and OO, but the majority of the code was Procedural Programming
  - May have been faster if we used Python / C
- Pipelining internal requests, multiple requests to the same server
  - Reduces parallelism, but also reduces socket connections