Cloud Computing HW2

Handling Failure Modes

- Security Breaches - The system's security suite was written lightly for the purpose of making it easy to run a demo. In particular, the security groups of RedisServer and EndpointServer enable all traffic on the apps default ports. In fact the demo RedisServer is occasionally being hacked[1]. In a production setup the security groups would have to be configured to enable only the EndpointServer instances and the WorkerNodes to access RedisServer. Since WorkerNodes are created dynamically, their IP addresses need to be updated in the RedisServer security group. An easy way to accomplish that is to run WorkerNodes on fixed IP addresses. Also, RedisServer would have to be configured properly with a user and a password and run in protected mode.
- High System Load (high traffic / high CPU/RAM usage) - Except for the WorkerNodes that are in fact being dynamically created based on the system load, the other system components could benefit from improving the load balancing. A load balancer node that receives requests to the system and distributes them to the EndpointServers would ensure requests are distributed uniformly between the instances s.t. latency SLA's are respected and potentially turn up and turn off instances of EndpointServer based on the system load. Splitting the load between multiple RedisServer instances is also possible. 'Enqueue' requests are easy to handle because they may be stored on any instance that is being worked on by some WorkerNode. 'pullCompleted' requests are more tricky because an EndpointServer may need to access multiple 'CompletedWork' queues to collect enough tasks (and there's the edge case of having less completed work than requested). In order to support that, a more complex logic s.a. having an allocator process that manages the completed tasks queues and allocates the tasks that are returned with each request.
  Note: the WorkerNodes config in my implementation is very naive (1 worker node at most that is turned up when the system receives tasks and turned off if some time passed since the last task was sent). In a production system a more robust configuration is probably required that can match the estimated load on the system (more instances / quicker turnup / have at least some instances that are up at all times to avoid task starvation).
- App Failures - App failures may occur and some monitoring means is required to make sure they're caught and handled. Also the app should be able to recover from failures and drop the "query of death".
- Unreachable Servers / Network Issues - the system / every component should have a primary instance that performs health checks with other instances and in case of connectivity issues it can instantiate additional instances of its type to replace the absent instances.
- Poor Performance - can increase system load which can severely affect the system health. One way to improve performance is to improve data locality by running

[1] https://stackoverflow.com/questions/50264694/my-redis-auto-generated-keys

RedisServers locally with EndpointServers which would reduce the amount of writes over RPCs and hence improve performance.

- Machine / Hardware Failure - having redundancy in every component in the system can cover for this case. The system can also be replicated in multiple availability zones to protect from severe events (these should be configured rather than hardcoded like in my demo).
- Race Condition - In my implementation, a race condition may occur between two processes fulfilling a 'pullCompleted' request because they pop completed work a single item at a time due to the default redis version limitations[2]. Therefore every task won't return exactly the latest work items completed in order.

[2] https://github.com/go-redis/redis/issues/1754