Fault tolerance

In order to force fault tolerance, specifically node failure as investigated in this report, redundancy must be enforced. Specifically, a replica number is required from user and a write is returned as successful only if all replicas are updated correctly. However, with traditional hashing, node failure might still lead to lose data. For multiple servers file system, usually two functions are used:

1. Data hash function

   This function maps different data keys, usually in string format, into a hash space

2. Distribute hash function

   This is used to distribute the data into multiple servers, usually a module hash is used and hash function is decided by number of servers.

As shown in first row Fig. 5, different color denotes different storage nodes are responsible to store different data based on the distributed hash function. However, when one node is down, the black node as shown in this example, the distribute hash function changed and each storage will be responsible for other data. In other words, all current data will be inaccessible.

The root cause for above problem is, data hash and distribute hash are two different functions and when one node is down the distributed hash changes. In this work, a consistent hashing method is used.

Consistent Hashing:

The consistent hashing is illustrated in Fig. 6. The same hash function, md5 hash in his report, is used to encode data key and distribute data among servers and a ring is formed. The hash value increases along the clock wise direction and rounds up. For any data, let's say e for example, follow along the ring in clockwise direction, the first storage node met will be responsible for this data.

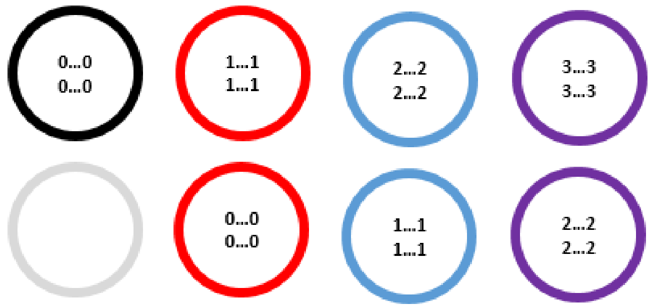


Figure 1 Tradition hash: one dead node cause all hashes invalid

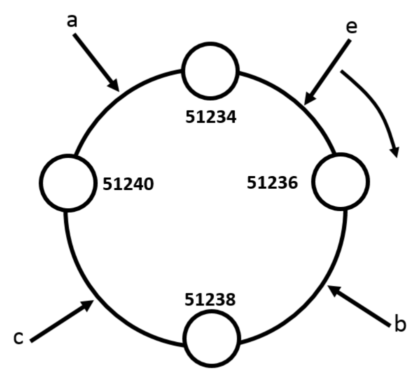


Figure 2 Illustration of consistent hashing

Consistent hashing for fault tolerance:

As mentioned above, for any key, the first responsible storage node is the one first met on the ring starting at the data's ring positions. And all other replicas stored on nodes following the first nodes. For example, if 3 replicas are stored for data e, the replicas will stored at nodes 51236, 51238 and 51240, respectively.

Failure detection:

If the server is not responding with a write or read request, the server will be treated as dead and removed from ring.

Test case: All nodes are live and write "all live" to files

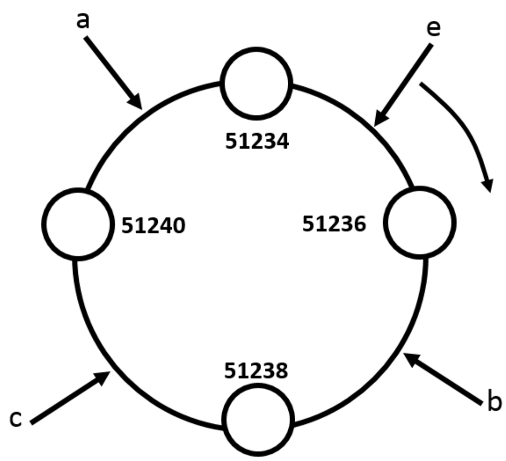


Figure 3 all ring nodes are live

```
fujunl@BME-YANG-7CFNP22: ~/compute-system-design/class-project-fault-to
*************write operation**************
data --/a-- is successfully stored on server http://localhost:51234
data --/a-- is successfully stored on server http://localhost:51236
*************write operation**************
data --/b-- is successfully stored on server http://localhost:51238
data --/b-- is successfully stored on server http://localhost:51240
*************write operation**************
data --/c-- is successfully stored on server http://localhost:51240
data --/c-- is successfully stored on server http://localhost:51234
*************write operation**************
data --/e-- is successfully stored on server http://localhost:51236
data --/e-- is successfully stored on server http://localhost:51238
```

Figure 4 Screen shots for ring write

Files	Storage nodes
a	51234, 51236
b	51238, 51240
c	51240, 51234
e	51236, 51238

Figure 5 File storage location when all nodes are live

Test case: One node is down and write "one down" to files

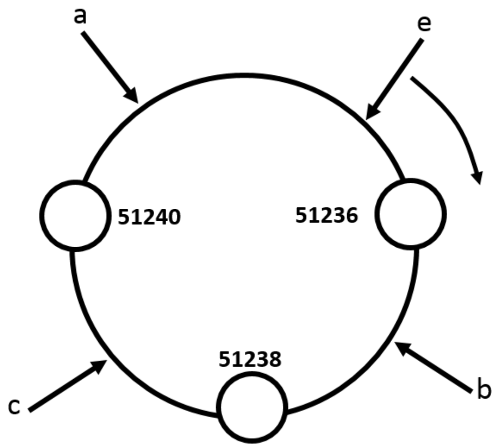


Figure 6 One ring node is down

Figure 7 Screen shots for ring write

Files	Storage nodes
a	51236, 51238
b	51238, 51240
c	51240, 51236
e	51236, 51238

Figure 8 File storage locations when one node is down

Test Case: Two nodes are down and write "two down" to files

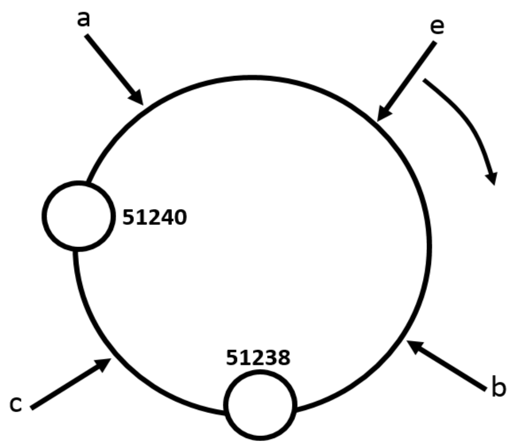


Figure 13 Two ring nodes are down

Figure 14 Screen shots for ring write

Files	Storage nodes
a	51238, 51240
b	51238, 51240
c	51240, 51238
e	51238, 51240

Figure 15 File storage locations when two nodes are down

Test case: Three nodes are down and write "three down" to files

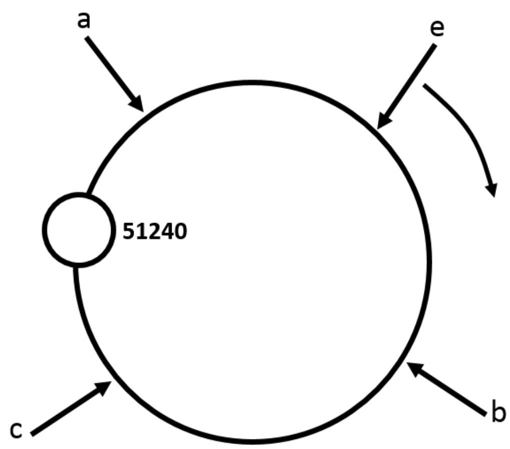
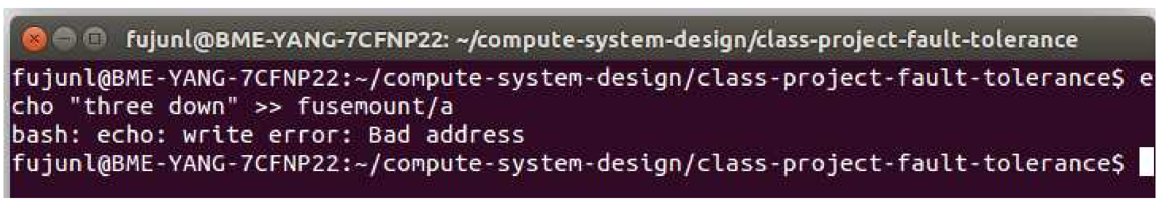
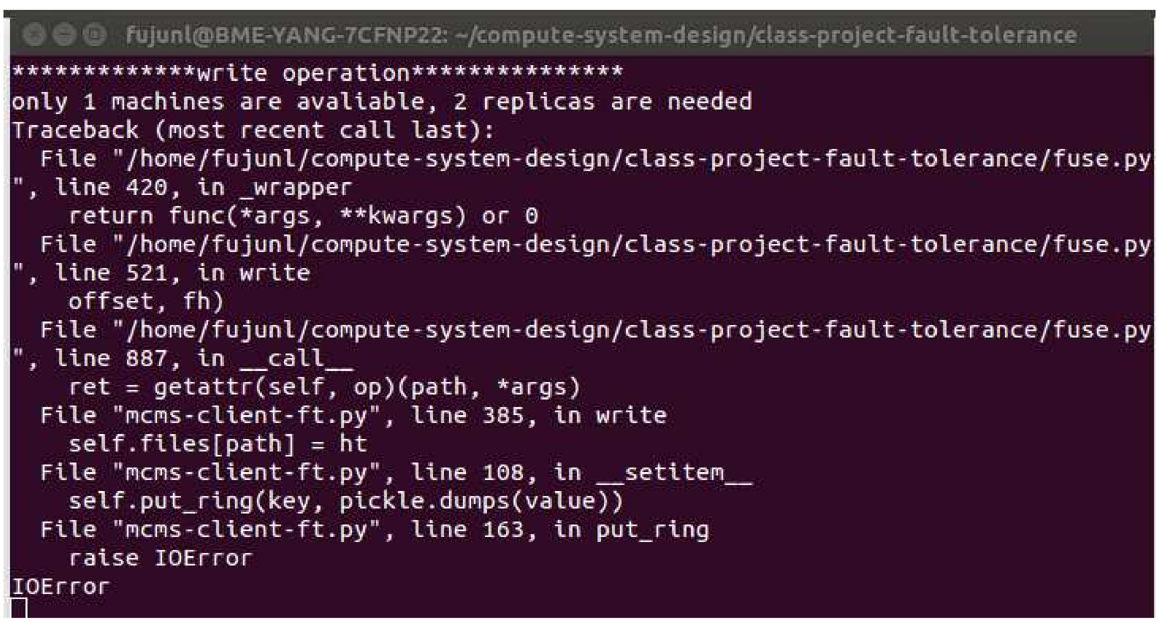


Figure 16 Three ring nodes are down



```
fujunl@BME-YANG-7CFNP22: ~/compute-system-design/class-project-fault-tolerance
fujunl@BME-YANG-7CFNP22:~/compute-system-design/class-project-fault-tolerance$ e
cho "three down" >> fusemount/a
bash: echo: write error: Bad address
fujunl@BME-YANG-7CFNP22:~/compute-system-design/class-project-fault-tolerance$
```

Figure 17 Write error happed when three nodes are down since replicas = 2 is enforced



```
fujunl@BME-YANG-7CFNP22: ~/compute-system-design/class-project-fault-tolerance
*************write operation***************
only 1 machines are avaliable, 2 replicas are needed
Traceback (most recent call last):
  File "/home/fujunl/compute-system-design/class-project-fault-tolerance/fuse.py
", line 420, in _wrapper
    return func(*args, **kwargs) or 0
  File "/home/fujunl/compute-system-design/class-project-fault-tolerance/fuse.py
", line 521, in write
    offset, fh)
  File "/home/fujunl/compute-system-design/class-project-fault-tolerance/fuse.py
", line 887, in __call__
    ret = getattr(self, op)(path, *args)
  File "mcms-client-ft.py", line 385, in write
    self.files[path] = ht
  File "mcms-client-ft.py", line 108, in __setitem__
    self.put_ring(key, pickle.dumps(value))
  File "mcms-client-ft.py", line 163, in put_ring
    raise IOError
IOError
```

Figure 18 Screen shots for ring write

In this implementation, only one successful read is required, so as shown in Fig. 19, the read is still successful though three out of four nodes are down.

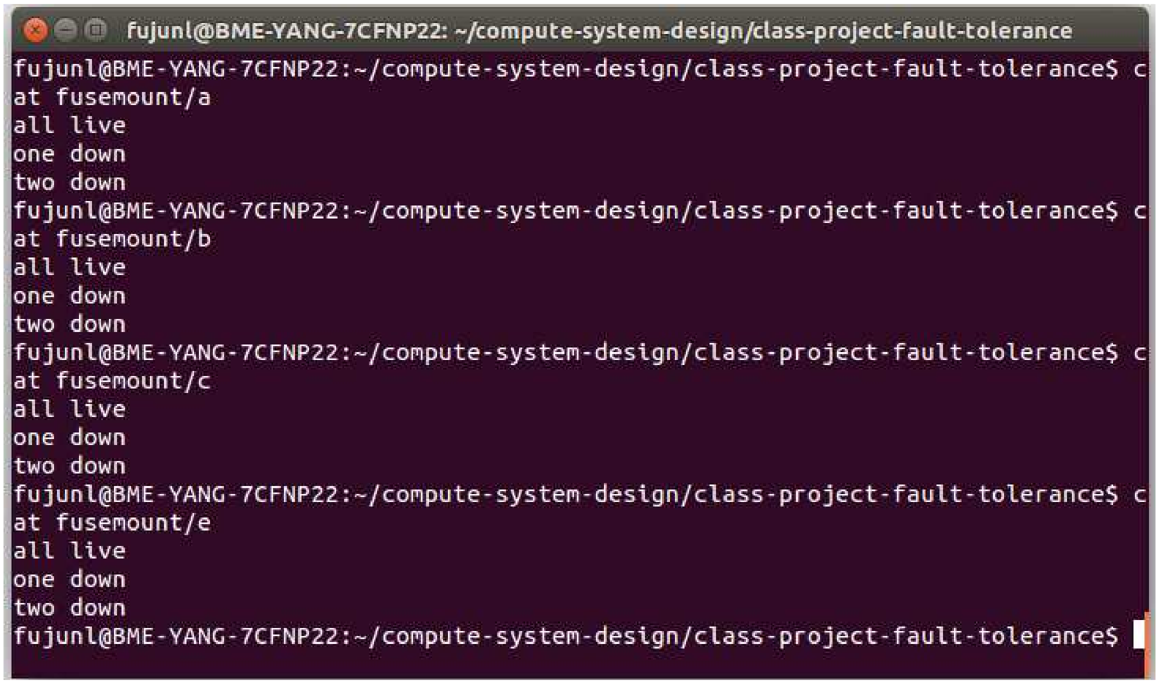


Figure 19 Read is successful when three nodes are down

Potentials Issues:

1. If several nodes are down at down at the same time, and some data whose replicas are stored in those dead node will be lost.
2. Failure detection is not reliable
   In current design, the server is treated as dead if the not responding a read or write request. This response might be lost in network or just late
3. Node recovery is not supported.

The implementation does not support node recovery