# Technical Report of Distributed SF1

iZENEsoft R&D

May 14, 2013

| Date | Author | Notes |
|---|---|---|
| 2012-04-16 | Zhongxia Li | Initial version. |
| 2013-05-08 | Vincent Lee | Added replication sychronization. |

**Abstract**

This report describes the initial design of the distributed SF1 system, including distributed topology and node management in association with ZooKeeper, distributed search and index, also mentions some further issues. Compared to distributed search we also have distributed recommendation, but we just mention distributed search in this report because they are very similar. The distributed SF1 will be gradually extended and optimized to adapt the requirements of our distributed application.

## Contents

# 1 Distributed Topology

## 1.1 Overview

We firstly go through the initial design for distributed SF1R, and make clear about some terminologies.

- **Cluster** A Cluster is a group of distributed SF1 servers (nodes) coordinated to provide search service.
- **Node** A node, or SF1 node, refers to a running SF1 process in a cluster of distributed SF1 system. It's possible that more than one SF1 nodes run on one machine, though it's unusual. Typically, we deploy one SF1 node on one machine, and in this case each host or machine in network is a node.
- **Replica** For a cluster of nodes, we may have one or more replication(s) to support failover, we call each replicated cluster a Replica.
- **Replica Set** For each node, we may have one or more backup node(s) in different replications, we call each set of reduplicated nodes a Replica Set.
- **Master** A SF1 node can work as Master or Worker or both. A Master is the controller and router of a cluster of distributed search servers (nodes), it receives all the outside query requests, dispatch requests to sub servers and gather results.
- **Worker** Compared to Master, each search server (node) is a Worker, which provide partial search data and response for partial search result.
- **Data Shard** In a distributed SF1, search data will be partitioned to different Data Shards. Currently each data shard will reside on each one Worker, so we also use shard ids to identify workers.

As an example of distributed topology, see figure 1, we have three nodes in the cluster of distributed SF1, and there are two Replicas.

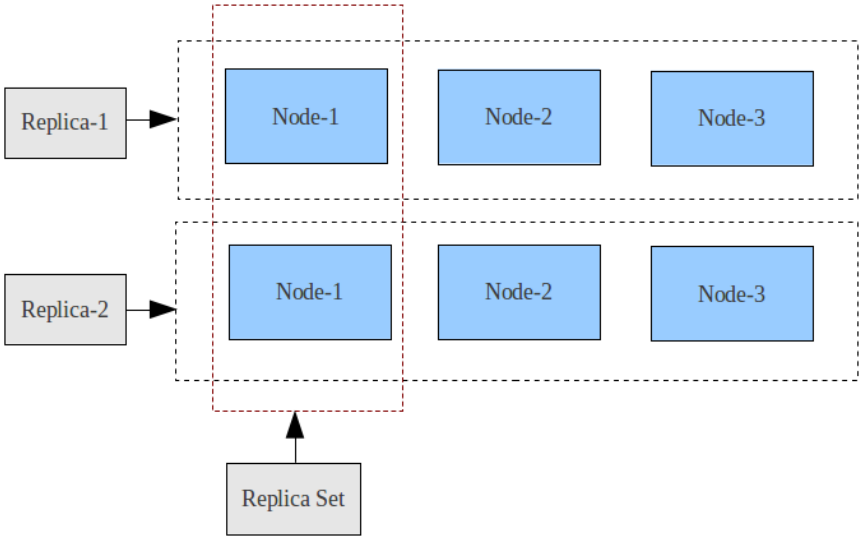


Figure 1: Example of Distributed Topology

## 1.2 Configuration of distributed SF1

The distributed SF1 is aimed at providing a flexible and configurable distributed solution, we need to configure each SF1 node (by editing the sf1 configuration file) to deploy a cluster of distributed SF1.

Commonly, we have to set *clusterid* for a SF1 node first to specify which cluster it will join in.

To configure current SF1 node as **Master**, we can enable *MasterServer*, notice that the distributed SF1 is collection related, which means different collections can have different distribution cases, we can see that a collection supported by a Master can be distributive or not. A undistributed collection will reside on local host only, while distributed collections can reside on different set of Workers.

To configure current SF1 node as **Worker**, we can enable *WorkerServer*, we have to specify the shard id which indicates which part of data the Worker will keep, and specify the collections it will support.

As an example, we'll give the configurations of an application scenario of distributed SF1 showed in figure 2. Considering that we have three SF1 server nodes, node1 and node2 work as Master respectively, and they all work as Workers.

- Configuration for Node1

```
<DistributedCommon clusterid="sf1-example" username="lscm"
    localhost="172.16.0.36" workerport="18151" masterport↩
        ="18131" datarecvport="18121" />

<DistributedTopology enable="y" type="search" nodenum="3">
```

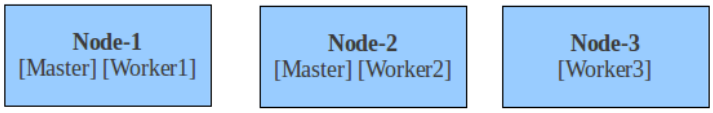


Figure 2: An exmaple cluster of Distributed SF1

```
    <CurrentSf1rNode nodeid="1" replicaid="1">
      <MasterServer enable="y" name="undefined" shardnum="3">
        <Collection name="web" distributive="y" shardids↩
            ="1,2,3" />
        <Collection name="b5mo" distributive="n" />
        <Collection name="b5mp" distributive="y" shardids="1,2"↩
            />
        <Collection name="b5mc" distributive="y" shardids="2,3"↩
            />
      </MasterServer>

      <WorkerServer enable="y" shardid="1">
        <Collection name="web" />
        <Collection name="b5mp" />
        <Collection name="b5mc" />
      </WorkerServer>
    </CurrentSf1rNode>
  </DistributedTopology>
```

- Configuration for Node2

```
  <DistributedCommon clusterid="sf1-example" username="lscm"
      localhost="172.16.0.161" workerport="18151" masterport↩
          ="18131" datarecvport="18121" />

  <DistributedTopology enable="y" type="search" nodenum="3">
    <CurrentSf1rNode nodeid="2" replicaid="1">
      <MasterServer enable="y" name="undefined" shardnum="3">
        <Collection name="web" distributive="y" shardids↩
            ="1,2,3" />
        <Collection name="qa" distributive="n" />
        <Collection name="b5mp" distributive="y" shardids="1,2"↩
            />
        <Collection name="b5mc" distributive="y" shardids="2,3"↩
            />
      </MasterServer>

      <WorkerServer enable="y" shardid="2">
        <Collection name="web" />
        <Collection name="b5mp" />
        <Collection name="b5mc" />
      </WorkerServer>
    </CurrentSf1rNode>
  </DistributedTopology>
```

- Configuration for Node3

```
  <DistributedCommon clusterid="sf1-example" username="lscm"
      localhost="172.16.0.162" workerport="18151" masterport↩
          ="18131" datarecvport="18121" />

  <DistributedTopology enable="y" type="search" nodenum="3">
    <CurrentSf1rNode nodeid="3" replicaid="1">
      <WorkerServer enable="y" shardid="3">
        <Collection name="web" />
        <Collection name="b5mp" />
```

```
        <Collection name="b5mc" />
      </WorkerServer>
    </CurrentSf1rNode>
  </DistributedTopology>
```

# 2 Distributed Node Management

## 2.1 ZooKeeper

Apache ZooKeeper is a high-performance coordination service for distributed applications, and it was employed in the distributed SF1 as distributed coordination service. More detailed information about ZooKeeper can be referred at the Apache web site [ZooKeeper].

The key data structure that used in ZooKeeper for coordination tasks is the *ZooKeeper Namespace*, the ZooKeeper Namespace defined for distributed SF1 is showed below.

The namespace kept in ZooKeeper service provides a consistent view of the topology of the whole distributed system, e.g., the *znode* identified by path */SF1R-[clusterid]/SearchTopology/Replica1/Node1* reflects the status of *Node1* in the cluster, and we can set data to znodes to share information or status among all the distributed nodes.

- *ZooKeeper Namespace for Distributed SF1*

```
/
|--- SF1R-[clusterid]
      |--- SearchTopology
           |--- Replica1
                |--- Node1  (Master/Worker)
                |--- Node2  (Master/Worker)
                |--- Node3  (Worker)
           |--- Replica2
           ...
      |--- SearchServers       # A search server is a master
           |--- Server00000000
           |--- Server00000001

      |--- RecommendTopology
           |--- Replica1
           |--- Replica2
      |--- RecommendServers
           |--- Server00000000
           |--- ...
```

- *Communication with ZooKeeper*

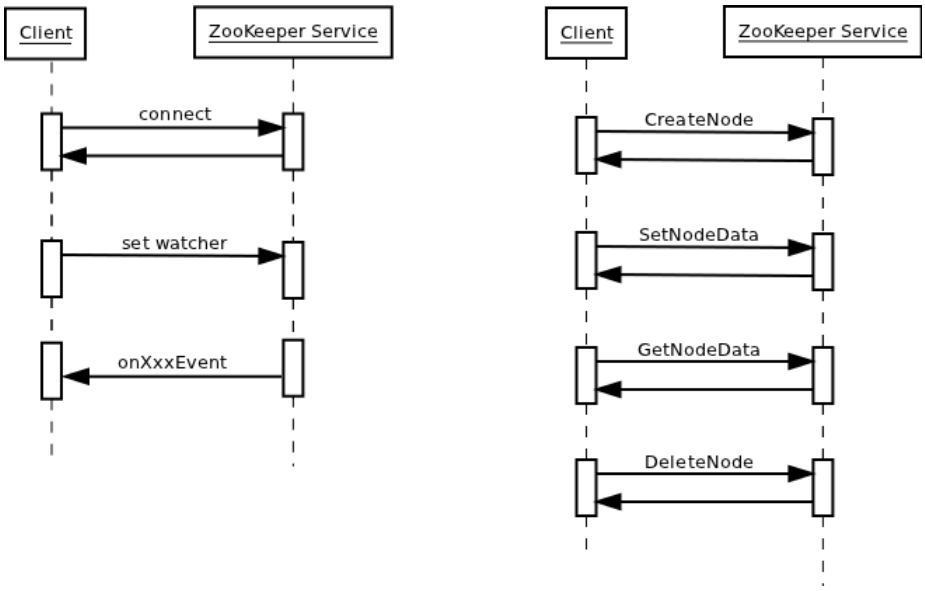


Figure 3: Basic communication operations with Zookeeper

- *Distributed coordination using ZooKeeper*

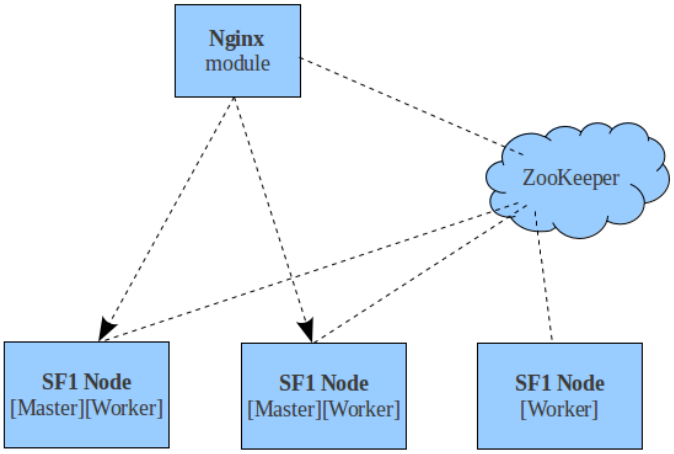


Figure 4: Coordination with Zookeeper

## 2.2 Node Manager

In distributed scenario, the node manager component of SF1 server performs node management, it registers current SF1 node to the namespace of ZooKeeper service to share info and detects other nodes of the cluster. The following shows the structure and sequences of node manager.

- *Node Manager*

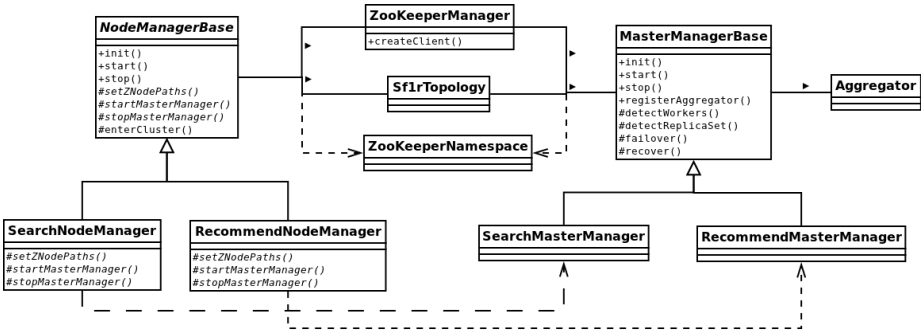


Figure 5: Class diagram of node manager

- *Startup of Node Manager*

  If SF1 server was configured as distributed SF1 node, it will start node manager to register itself into cluster. If it was also configured as Master server, it will start master manager to detect Workers in cluster and perform some related tasks, there are also some *further* tasks such as *failover* and *recover*.

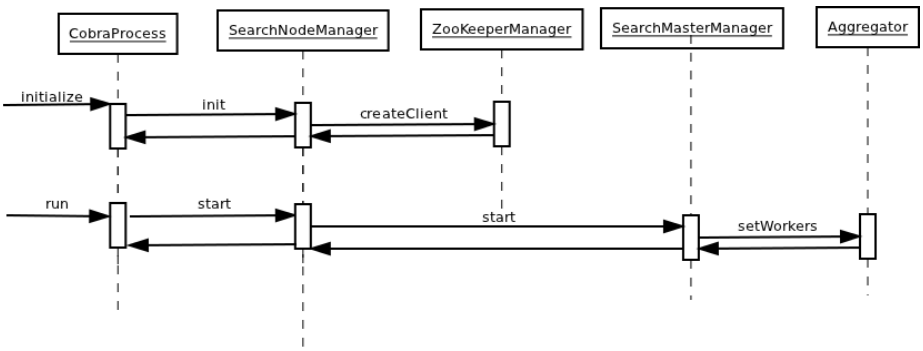


Figure 6: Sequence of startup of node manager

# 3 Distributed Search

In distributed SF1, all search requests will be sent to Masters in the cluster. When Master received a search request, basically it will dispatch the request to all the Workers and then aggregate (merge) results returned from different Workers, these works are performed through a *Aggregator* framework.

As an exmaple, in figure 7, we showed how *keyword search* is processed as a distributed search. For other kinds of requests, the basic processes are more or less the same.

Note that if a Worker works on the same SF1 node as Master, it will perform local function call to the local Worker instead of sending rpc request for efficiency.

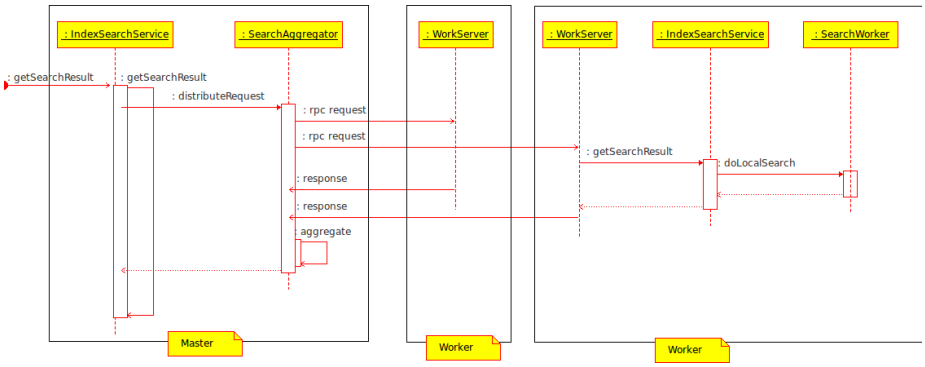


Figure 7: Distributed keyword search

# 4 Distributed Index

In distributed SF1, search data will be partitioned and dispatched to multiple Workers (each data shard resides on each Worker) and the key problem is *Data Sharding* or *Data Partitioning*, concretely we'll partition SCDs which will be indexed by SF1. Each document in SCD is recorded with properties in key-value pairs, also we can treat the whole document as a key-value pair by using a specified key. SCDs for distributed SF1 will be partitioned by Master, and dispatched to related Workers, as show in figure 8.

Generally, there are some strategies for this hashing like partitioning, such as *consistent hashing* which has the essential property that removal or addition of one node changes only the set of keys owned by the nodes with adjacent IDs, and leaves all other nodes unaffected. *Locality-preserving hashing* ensures that similar keys are assigned to similar nodes, this can enable a more efficient execution of range queries. We may want to consider sharding strategy with issues of node change, range query, and relation between different collections. But it's hard to satisfy all the needs in a strategy, so we may have to weigh the advantages and disadvantages. Currently we just provided simple hashing based strategy for data sharding.

Further, we may need to add *Replication* strategy for data redundancy.

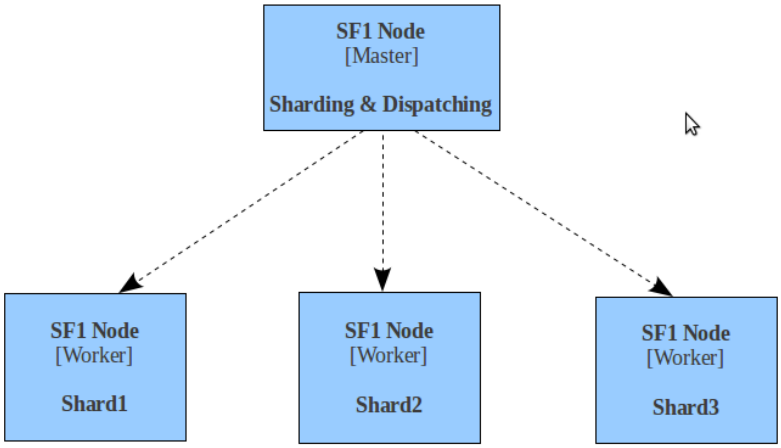


Figure 8: Distributed index

# 5 Distributed Replication Synchronization

## 5.1 Overview

*Distributed SF1R* nodes overview is shown as below. 9 Each Node can have several replicas, and all the replicas in the same replica set have the same nodeid. We call them together as a replica set. Basically, there will be one primary node and some secondary nodes in the specific replica set. Each node will supply the master and worker parts.

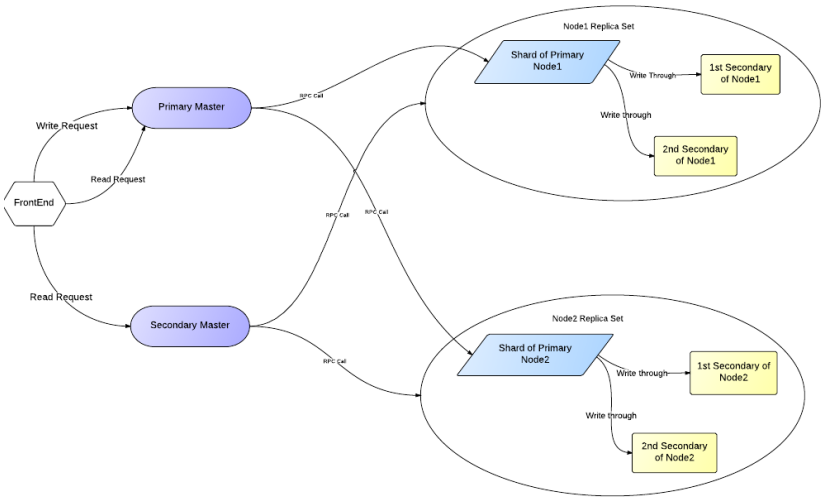


Figure 9: Distributed SF1R Architecture

1. *Primary Master*

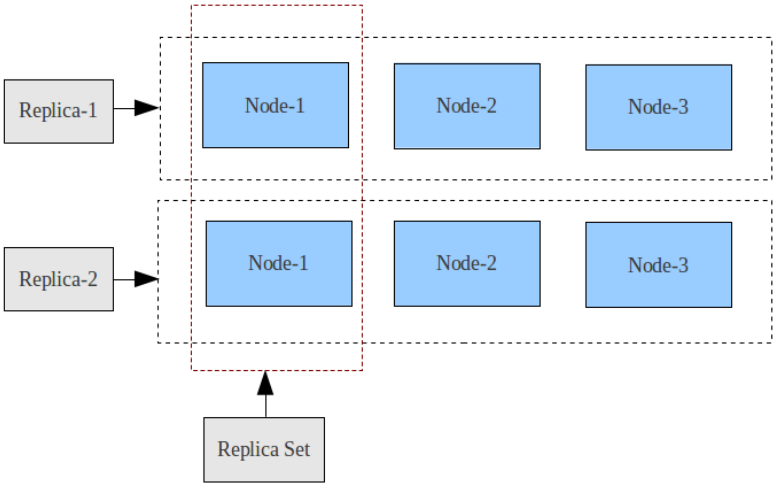


Figure 10: Distributed SF1R Replica Set

The master part on the primary node, it will take charge of distributing the search request to all shard workers and merging the results from all shard workers. And also take the charge of pulling the write request from zookeeper and distribute the write request to all shard workers on primary.

2. *Secondary Master*
   The master part on the secondary node, it will take charge of distributing the search request to all shard workers and merging the results from all shard workers. And also take the charge of pushing the write request to zookeeper.

3. *Shard Worker*
   The worker part on the node will do the actually work as one of shard worker and return the result to the master. For write request, the shard worker on primary node will take charge of broadcasting the request to all secondary nodes in the same replica set.

## 5.2 Architecture and Internal

### 5.2.1 The role of ZooKeeper

The ZooKeeper in the distributed sf1r will serve as the node activity detecting and primary electing. The distributed write queue is also used in the zookeeper to make sure all write request can be saved temporally. In sync mode, the zookeeper is used to keep the processing state of write request and notify the primary and the secondary nodes about the state changes. By using zookeeper we can make sure all nodes will see the same topology view in the distributed sf1r since the zookeeper has implemented the PAXOS protocols to make sure the data consistency. The topology in zookeeper is as below:

```
/                                   # Root of zookeeper namespace
|--- SF1R-[CLUSTERID]               # Root of distributed SF1 namesapce↩
    ,
                                    [CLUSTERID] is specified by user ↩
                                        configuration.

     |--- Topology                  # Topology of distributed service ↩
          cluster
          |--- Replica1             # A replica of service cluster
               |--- Node1           # A SF1 node in the replica of ↩
                    cluster, it can be a
                                       Master or Worker or both.
               |--- Node2
          |--- Replica2
               |--- Node1
               |--- Node2

     |--- Servers                   # Servers in topology is a master ↩
          node.
          |--- Server00000000
                    |--- Search, Recommend       # A master node ↩
                         supply Search and
                                                 #  Recommend service↩
                                                      as master
          |--- Server00000001
     |--- WriteRquestQueue                       # Root of waiting ↩
          write request queue
          |--- Node1                             # Waiting Write ↩
               request queue for node1
               |--- WriteRequestSeq0000000000    # the waiting write ↩
                    request
          |--- Node2
     |--- PrimaryNodes
          |--- Node1                             # backup primary ↩
               nodes for node1
               |--- Primary0000000000            # first backup ↩
                    primary server as current
                                                 #  primary for ↩
                                                      replica set ↩
                                                      node1.
               |--- Primary0000000001
          |--- Node2
               |--- Primary0000000000
               |--- Primary0000000001
     |--- WriteRequestPrepare                    # prepare root node ↩
          for sync write
          |--- Node1                             # prepare for node1 ↩
               in sync write.
          |--- Node2
     |--- Synchro                # For synchronization task
```

### 5.2.2 The new write routine on distributed sf1r

Figure 11 has shown the working flow of the write request:

1. A client caller send write request to one of node in the specific replica set on distributed sf1r.

2. The master part on the node will push the request to the write queue on zookeeper. For each replica set it has a standalone write queue on the zookeeper for its own.

3. The primary node will be notified and try to get the new write from zookeeper. If success, the master part on primary node will set the prepared state and pop the write request from the queue.

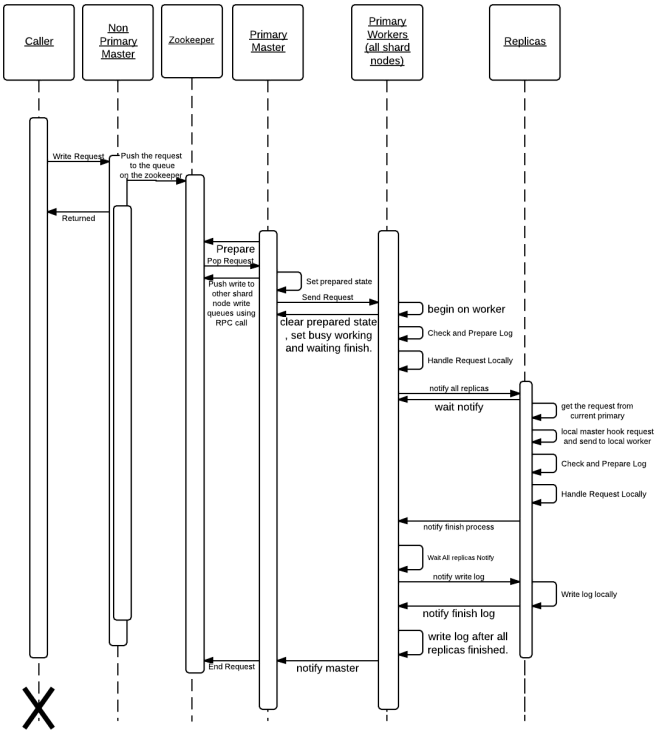


Figure 11: Write request handling Flow

4. If the write request should be distributed to other shard workers, it will be pushed to the write queue belong to that shard node.

5. The primary shard worker begin process on local. After finished on primary worker, it will notify other nodes in the same replica set by updating the zookeeper nodedata. Then the primary worker will wait all secondary nodes until finished or down by accidentally.

6. The secondary worker will be notified by zookeeper while there is a new write from primary. After the secondary worker finished, it will notify the primary by updating the zookeeper nodedata and wait the primary to get ready to write log.

7. After all secondary finished the write request(or part of them down), the

primary will notify all secondary to write the log.

8. The secondary worker will get notified from primary after all secondary finished write request. Then it will write the log to disk and notify primary that the log has been written.

9. After all secondary finished writting log, the primary worker will write log to disk and the write request is finished finally.

10. The primary worker notify the primary master on the same node that it is ready for next new request. And the primary master will check if any new request on the write queue and continue to handle the next write request.

#### 5.2.3 Starting Node In Distributed SF1R

How a node start in distributed sf1r is shown as below: 12. Starting a node will add a secondary to the replica set if there is already a primary, otherwise the starting node will enter the replica set as a primary.

1. *Staring node as primary*
   Each distributed node will be registered as a backup primary node in the same replica set on the zookeeper. The first backup primary node in the replica set will be treated as primary.

   - Check data: while starting, the node will check whether local data is ok by checking if some flag file exists on local. If last down by accidentally, the node can not be started as primary. Because we can not identify the correctness of the local data without checking by comparing with the other node.
   - Register as backup primary: After checking data, the first node will start and register a backup primary in the replica set. Because this is the first backup primary, it will be treated as the current primary. The start is finished at last.

2. *Starting as secondary*
   After the first node in the replica set started, all other nodes will be started as secondary.

   - Recover: If last exit is normal, no recovery needed. Otherwise, the secondary will restore the data from the newest backup.
   - Sync to Newest log: After recovery finished, the secondary will pull new log data from primary, and redo the log since last down. After synced to the newest log, the secondary will notify the primary to stop accecpt new request and begin to enter cluster.
   - Check consistent: If primary agreed the enter of secondary, the secondary will check the data consistent with primary by computing the CRC of the collection data files. If the consistent finished success the node starting as secondary is done at last. Otherwise, the secondary node will fail to start.

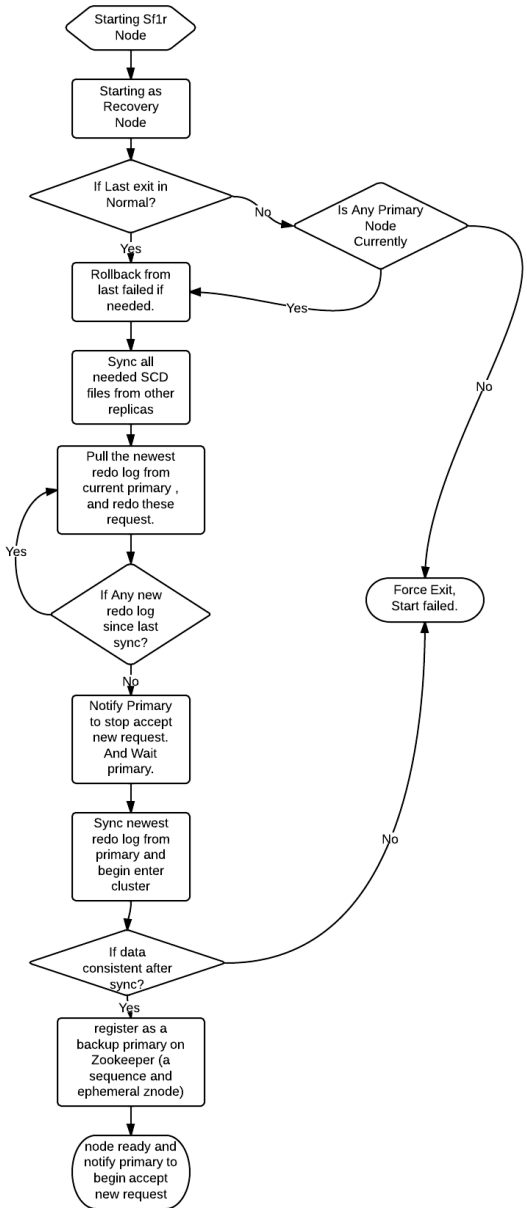


Figure 12: Starting of a node

### 5.2.4 Failing in Distributed SF1R

In distributed sf1r, there are many cases to cause a failing node. Basically, there are two major fails: primary fail and non-primary fail. Beside, because the zookeeper is used, we need handle the zookeeper connection lost as one kind of fail. The handle of fail is shown in Fig 13.

1. Abort write request If the write request failed to finished, the node need rollback to the state before write. This is aborting a write request.

- When abort hanppen: Abort may happen if the primary failed to finish or a secondary send abort request to primary. The expired session on zookeeper connection while running request will also trigger the aborting.
- How to abort: While aborting triggered, the primary node will notify all secondary nodes to abort the request. If the secondary node did not begin to run the write, the abort can ignore since no write happen on the node. Otherwise, the secondary node will do the aborting and notify primary after aborting finished. While aborting on the local node, it will find the latest backup and restore data from it. After that the node will redo the logs between the latest backup and current failed state. By redoing the log, the node can restore state to the old state before the failed write actually running.

2. Auto failover and recover

   The master will be notified if the fail node is in the watching list of master and the master will try to failover this node by using the backup node with the same nodeid in other replica set. If the fail node restarts later, the master will recover this node.

3. Primary Fail Current primary fail will trigger a backup primary to begin electing and the backup primary will take the charge of primary if its log is newest.

   (a) Primary fail while idle: While idle, the primary fail will notify all replicas in the same replica set and the first backup primary will become new primary and notify all other replicas by updating self state to electing. The other backup nodes will just update the current primary info and notify the new primary that I am ready to follow the new primary. After all other nodes followed the new primary, the new primary will end the electing and ready for the next new write request.

   (b) Primary fail while write request running local: If the primary failed while running the write request but not finished yet. Because the other nodes in the replica set haven't began to run the write, we can handle it just the same as case 1.

   (c) Primary fail While write request finished local: In this case, some of the secondary nodes have already begun to run write request on the node. So we need abort the current write request if the node has begun and after aborting the node will reenter the cluster to sync to the new primary. If the node has not yet start to run the write, it can begin electing the same as case 1.

   (d) Primary fail while ready to write log: In this case, all secondary nodes have already finished write request, but not all secondary nodes finished the log. This is almost the same as the case 3 except the node finished log do not need to abort the current write request. All others not finished log need abort request and re-enter cluster.

   (e) Primary fail while electing: In this case, it means the current primary failed and the first backup also failed while trying to become new

primary. It is almost the same as the case 1 except the second backup take charge of the new primary.

(f) Primary fail while others recovering: In this case, the recovering node need follow the new primary and redo recovering from the mew primary. The nodes not recovering can be handled just like the other case in 1-5.

In all case above, the node re-enter cluster if the log is fall behind others. The newest log id will be updated to zookeeper once the node finished log.

4. Non-Primary Fail

   (a) Non-Primary fail while idle: Nothing will happen except auto failover on master since no write running.

   (b) Non-Primary fail while write request running local: The primary will stop waiting this fail node and check if others finished the write request.

   (c) Non-Primary fail while recovering: The primary will stop waiting this node to enter cluster and check if any new write request can be handled.

   (d) Non-Primary fail while electing: The new primary will stop waiting this node to follow itself and check if others followed it.

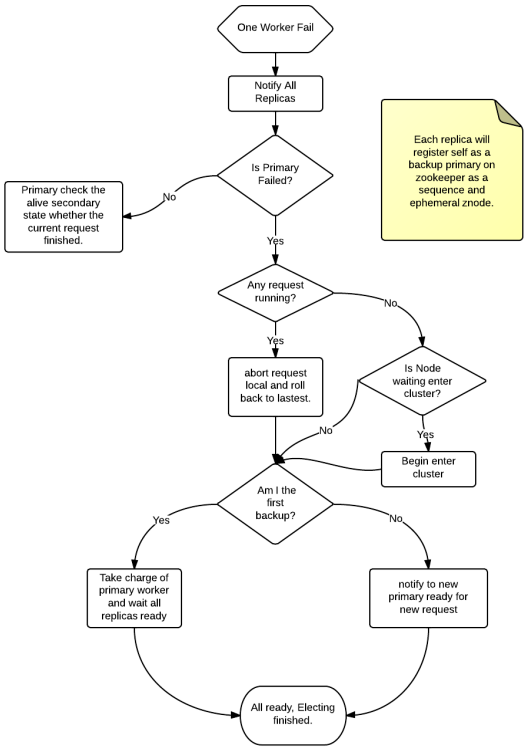


Figure 13: Fail of a node

5. ZooKeeper Connection Lost ZooKeeper will lost connection by auto-reconnect or expired session. This may happen while network is unstable or zookeeper server is restarting.

    - Auto-reconnect: In this case, the zookeeper will make sure all nodes created by self keep the same after auto-reconnected. But we need refresh other nodes state such as primary or other secondary nodes after auto-reconnect finished. If primary node has changed during auto-reconnect, the node need do the same thing just like the primary fail case. If nothing happened during auto-reconnect, the node can continue to run without any change.
    - Expired Session: expired session will cause all zookeeper self-created node and event invalid, so we need reconnect to the zookeeper by hand. After expired, the node will set the state and keep waiting until the node is ready to re-enter cluster. Any unfinished write will be aborted.

#### 5.2.5 Async write support

As described above, the write need keep communicating with zookeeper while doing the write. This will cause much downgrade of write performance if there are many replicas in the replica set because of the network latency. In order to improve the performance of write, the async write has been implemented to reduce the communication with zookeeper while handling the write request.

- Async write on primary node: On primary the write flow is almost the same as sync except the notify to zookeeper has been just ignored. In async write, the primary will pop the write request from zookeeper and handle it locally without notify others and keep going on until no more new write on zookeeper. As we can see, the communication with zookeeper has been reduced to only 1 time. This will greatly improve the write performance and will keep the same even the replica set become larger.
- Async write on secondary node: Since the primary no long notify secondary about the write request, the secondary nodes will pull the redo logs from current primary periodically and redo these logs in the log sync thread.
- Drawback of async write: As we can see, the secondary nodes may fall behind from primary node, and if primary failed part of logs may have not synced to other secondary nodes. This will cause write request loss.
- Recovering in async mode: Since the failed primary may have the logs that other secondary nodes don't have, while the failed primary restarting, the node need check whether the log is newer that current new primary. If restarting with newer log, the node need rollback to old and sync to new primary.

#### 5.2.6 SF1R lib for distributed SF1R

The SF1R lib in Fig 14 is used by the distributed nginx. By using this sf1r-lib the client can send request with no need for knowing the topology in the distributed

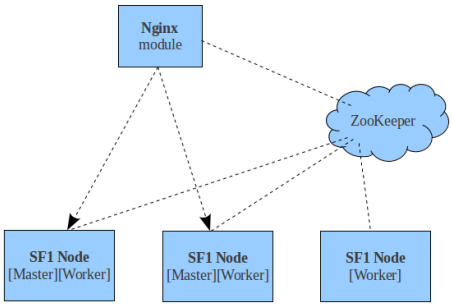


Figure 14: The SF1R lib for nginx

sf1r. This client lib will find the correct sf1r node in topology automatically and send the request to it. The client lib update the topology by using the zookeeper connection and any changes from zookeeper will be notified.

- Support For Separating read and write: Currently, the distributed sf1r node will write the current busy state to the zookeeper while the busy state changed. By reading the busy state from zookeeper node we can temporally disable the read from the node who is busy writing. In this way, we can route the read request and write request to different nodes to implement separating the read and write.

## 5.3 Developer Guide

Because all the write request will be handled on primary first and send to all secondary nodes from primary, the new write request need some adjustment to be adopted by distributed sf1r. Most work has been done and we can add new write simply follow as below step:

### 5.3.1 Add new write api

1. add the controller+action string in the function initWriteRequestSet in the RequestLog.cpp file.
2. if the write api is in the controller that derived from Sf1Controller, then most work will be done OK in base class. If not, you need handle it yourself.
3. If the api will execute on all shard workers, you need push the write request to the queues of other shard workers. (See the index write request in the class IndexTaskService for example.)
4. In the write request handler, coding as below:

```
// check valid first.
DISTRIBUTE_WRITE_BEGIN;
DISTRIBUTE_WRITE_CHECK_VALID_RETURN;
// do pre-check without modify any collection data.
if (precheckfailed())
{
```

```
    return false;
}

// prepare request log before actually modify the collection data.
CreateOrUpdateDocReqLog reqlog;
reqlog.timestamp = Utilities::createTimeStamp();
if(!distributereqhooker->prepare(ReqCreateOrUpdate_Doc, reqlog))
{
    LOG(ERROR) << "prepare failed in " << __FUNCTION__;
    return false;
}

// do modify collection data.
ret = modify_data();
if (!ret)
{
    return false;
}
// flush data to make sure data write to disk.
flush();

DISTRIBUTE_WRITE_FINISH(ret, reqlog);
return ret;
```

5. Make sure the controller and handler will execute in sync.
6. If an api will call another write api, you should set the chain status properly.
7. Define a new req log type in RequestLog.h if necessary and set for rollback and backup action in DistributeRequestHooker.cpp file.

### 5.3.2 Add new write cron job

In order to run cron job in distributed mode, the cron job need a job name cronJobName for each cron job and each cron job just do one job. cronJob task is a function with calltype parameter. You should write cron job function as below:

```
void RecommendTaskService::cronJob_(int calltype)
{
    if (cronExpression.matchesnow() || calltype > 0)
    {
        // check if need to put the job to distributed sf1r write queue.
        if(calltype == 0 && NodeManagerBase::get()->isDistributed())
        {
            if (NodeManagerBase::get()->isPrimary())
            {
                MasterManagerBase::get()->pushWriteReq(cronJobName_, "cron");
                LOG(INFO) << "push cron job to queue on primary : " <<cronJobName_;
            }
            else
                LOG(INFO) << "cron job ignored on replica: " << cronJobName_;
            return;
        }
        // check for valid
        DISTRIBUTE_WRITE_BEGIN;
        DISTRIBUTE_WRITE_CHECK_VALID_RETURN;
        // prepare the log for cron job.
        CronJobReqLog reqlog;
        reqlog.cron_time = Utilities::createTimeStamp();
```

```
        if (!DistributeRequestHooker::get()->prepare(Req_CronJob, ↩
            reqlog))
        {
            LOG(ERROR) << "!!!! failed running cron job. : " << ↩
                cronJobName_ << std::endl;
            return;
        }
        //do the job task here. and flush data after job finished.
        bool ret = DoJob();
        flush();
        //finish job
        DISTRIBUTE_WRITE_FINISH(ret);
    }
}
```

### 5.3.3 Add new write callback

The write callback is used for the write without any write api or cronJob. You can add/remove a write callback with an identify string and call it as below :

```
// add new callback.
DistributeDriver::get()->addCallbackWriteHandler(collection_ + "↩
    _callBackFuncName",
    boost::bind(&CallbackObj::callbackFunc, this, _1));

// removing the callback
DistributeDriver::get()->removeCallbackWriteHandler(collection_ + "↩
    _callBackFuncName");

// call callback by push the specific data.
DistributeDriver::get()->pushCallbackWrite(reqloghead.req_json_data, ↩
    reqdata);
```

The callbackFunc should have the protocol as below and should be with the log type ReqCallback:

```
bool callbackFunc(int calltype);
```

The calltype will be used to tell who is calling this callback. In the callback you should code just like the write request api handler.

### 5.3.4 Auto test

Currently, the distributed sf1r support auto test for new write request. The auto test will try all possible fail case to check if data is consistent after the write request.

For each write request api, you can add the test json request body to the autotest directory. All the json file under it will be tested in all kinds of node fails to make sure after the write request we get the consistent data.

If you want some node fail at the some test point, you can do

```
echo failtype > distribute_test.conf
```

under the working directory. the *failtype* here stand for the kind of test point at which line the sf1r will fail. Currently all test points are shown as below:

```
enum TestFailType
{
    NoAnyTest = 0,
    NoFail,
    PrimaryFail_At_Electing,
    PrimaryFail_At_BeginReqProcess,
    PrimaryFail_At_PrepareFinished,
    PrimaryFail_At_ReqProcessing,
    PrimaryFail_At_NotifyMasterReadyForNew,
    PrimaryFail_At_AbortReq,
    PrimaryFail_At_FinishReqLocal,
    PrimaryFail_At_Wait_Replica_Abort,
    PrimaryFail_At_Wait_Replica_FinishReq,
    PrimaryFail_At_Wait_Replica_FinishReqLog,
    PrimaryFail_At_Wait_Replica_Recovery,
    PrimaryFail_At_Master_PrepareWrite,
    PrimaryFail_At_Master_checkForNewWrite,

    ReplicaFail_Begin = 30,
    ReplicaFail_At_Electing,
    ReplicaFail_At_Recovering,
    ReplicaFail_At_BeginReqProcess,
    ReplicaFail_At_PrepareFinished,
    ReplicaFail_At_ReqProcessing,
    ReplicaFail_At_Waiting_Primary,
    ReplicaFail_At_Waiting_Primary_Abort,
    ReplicaFail_At_AbortReq,
    ReplicaFail_At_FinishReqLocal,
    ReplicaFail_At_UnpackPrimaryReq,
    ReplicaFail_At_Waiting_Recovery,

    OtherFail_Begin = 60,
    Fail_At_AfterEnterCluster,
    Fail_At_CopyRemove_File,

    FalseReturn_Test_Begin = 70,
    FalseReturn_At_UnPack,
    FalseReturn_At_LocalFinished,
};
```

# 6 Operation Note

## 6.1 Configuration

A simple sample configuration is as below:

SF1R Node1, work as Master and Shard Worker 1. For this Master, it provides multiple collections, "b5mp" is distributed to two shard workers.

```
    <DistributedTopology enable="y" nodenum="2">
      <CurrentSf1rNode nodeid="1" replicaid="1">
        <!--master names for B5M are www|stage|beta-->
        <MasterServer enable="y" name="undefined">
            <DistributeService type="search">
              <!--for distributive collection, shardids is set to all ↩
                  as default-->
              <Collection name="b5mp" distributive="y" shardids="1,2" ↩
                  />
            </DistributedService>
            <DistributeService type="recommend">
              <Collection name="b5mp" distributive="y" shardids="1,2" ↩
                  />
            </DistributedService>
        </MasterServer>
        <WorkerServer enable="y">
          <DistributeService type="search">
            <Collection name="b5mp" />
          </DistributeService>
```

```
        <DistributeService type="recommend">
          <Collection name="b5mp" />
        </DistributeService>
      </WorkerServer>
    </CurrentSf1rNode>
  </DistributedTopology>
```

SF1R Node2, work as Master and Shard Worker2 (with shard 2).

```
  <DistributedTopology enable="y" nodenum="2">
    <CurrentSf1rNode nodeid="2" replicaid="1">
      <MasterServer enable="y" name="undefined">
          <DistributeService type="search">
            <Collection name="b5mp" distributive="y" shardids="1,2" ↩
                />
          </DistributedService>
          <DistributeService type="recommend">
            <Collection name="b5mp" distributive="y" shardids="1,2" ↩
                />
          </DistributedService>
      </MasterServer>
      <WorkerServer enable="y">
        <DistributeService type="search">
          <Collection name="b5mp" />
        </DistributeService>
        <DistributeService type="recommend">
          <Collection name="b5mp" />
        </DistributeService>
      </WorkerServer>
    </CurrentSf1rNode>
  </DistributedTopology>
```

*nodenum* stand for the total different shard node workers (with different nodeid) in the topology. The *name* in Master Server can be used to tell the difference for the beta/stage/www.

## 6.2 Monitor status

Some node status will be updated to memory statistical data. If you want to get the running status of distributed sf1r node, you can using the api

```
status/get_distribute_status
```

From this we can known the current primary host and how many write request processed and etc..

## 6.3 Update SF1R

The update can have two major cases: simple update and update to new cluster. The simple update will happen while the update is compatible with old node, otherwise update to new cluster is needed.

- Update config: replace the config file on current primary node and send the api command

  ```
  collection/update_collection_conf
  ```

  After this the updated config will auto deliver to other replicas in the replica set.

- Simple update: If the data or code is compatible with the old one, we can do simple update. Just using the script below:

```
distribute_tools.sh simple_update new-sf1r-tar-file
```

  and wait until the updating node can serve as the new master or worker.

- Update to new cluster: update to new cluster is needed while the new is no longer compatible with the old. We can do this one by one as follow :

  1. Chose one node in the replica set and run the script :

```
distribute_tools.sh update_to_newcluster new-sf1r-tar-file ↩
    new-clusterid
```

  2. Rebuild data if needed by using the api.
  3. Backup node data. Using api

```
collection/backup_all
```

     to generate the backup data on the current node.
  4. Copy the backup data to other replicas and update the left nodes one by one. On the other nodes, using the script

```
distribute_tools.sh rollback new-clusterid
```

     to restore the data from the copied backup data. After all nodes have finished, the update to new cluster is done.

## 6.4 Handle Unexpected down

The node in distributed sf1r may be down at any time. After started, the node will set a force exit flag file and this flag file will be removed if the node is stopped normally. While starting, if the force exit flag exists, we treat it as an accident down. In this case we will restore the data from the latest backup and sync to newest from current primary.

While running the write request, the node will set a rollback flag file to indicate the current running request with the request id. If the write failed to finish, we can know which state we will rollback to. If the rollback file is empty it will rollback to the latest backup.

If all nodes in the replica set are down, we need check the log data carefully. We need find out which node has the newest log data and start this node as the first node in the replica set.

## References

[ZooKeeper] http://zookeeper.apache.org/