

Bigdata® High Availability (HA) Architecture

Introduction

This whitepaper describes an HA architecture based on a *shared nothing* design. Each node uses commodity hardware and has its own local resources (CPU, RAM, and DISK). A commodity network is used as the interconnect for the nodes.

Bigdata® High Availability (HA) Architecture.....	1
Introduction.....	1
Design.....	2
Metadata service.....	4
Transaction service.....	4
Load balancer service.....	4
Data service.....	4
Failure modes.....	5
Master election.....	5
Read errors.....	5
Resynchronization.....	5
Point in time rollback.....	6
Hot spares.....	6
Management.....	7
Alerts.....	7
Location aware allocation.....	7
Offsite replication.....	7
Backup and restore.....	7
Future directions.....	7

Design

Bigdata consists of several kinds of services, including:

- Data services (DS), which manage shards and are the vast majority of all service instances in a deployment;
- Metadata service (MDS), which is a locator service for shards;
- Transaction service (TXS), which coordinates transactions and the release time for historical commit points;
- Load balancer service (LBS), which aggregates performance counters and centralizes load balancing decisions.
- Client services (VS), which provide for the execution of distributed jobs.
- Jini Lookup Services, (LUS), which provide for service discovery.
- Zookeeper quorum servers (ZK), which provide for master election, configuration management, and global synchronous locks.

This document specifies how high availability is achieved for each of the bigdata® services¹. Jini and zookeeper have their own mechanisms for high availability and will not be addressed further in this document.

Bigdata services are grouped into logical instances (logical DS “A” vs “B”) and logical instances are realized by a failover set of physical services (physical DS #1, #2, #3, etc.). Logical service identifiers are the names of zookeeper child nodes for a given service class (MDS, DS, LBS, TXS, etc). Physical service instances are associated with a child node in zookeeper for the logical service group. Each physical service instance has a UUID which corresponds directly to its jini ServiceId. Physical services instances discover jini lookup service(s) when they start and register themselves for discovery. Physical services instances are created within a specific logical service group and that group membership is invariant. For example, the zookeeper hierarchy, in part, might look as follows with the service class (MDS), logical service group (A), and physical service instances for the metadata service.



For each logical service group, there is a set of k physical service instances². The physical service instances are arranged into a failover chain using a master election protocol. Writes are directed to the master physical service instance while reads may be directed to any of the failover instances for that service. During normal operations, the members of the failover chain are up to date at each commit point on the master.

¹ More detail on the overall bigdata architecture is available online at <http://www.bigdata.com/blog>.

² k is normally given as three (3).

The basic design pattern supporting failover is *write replication* across a chain of failover services. The failover chain is established in zookeeper using a master election protocol. Physical service instances subscribe to a queue in zookeeper. The head of the queue is the master. The order of the service instances is the order of the failover chain. Bigdata does not use a quorum model. It is sufficient to have a single master for any given logical service group, but such services can not be considered to be highly available. In order to be highly available, there must be a sufficient replication count for each logical service group. Standard practice is to choose a replication count of $k = 3$. It is important to note that a replication count of 3 merely increases the likelihood that any given datum remains available.

Write replication takes place at a low level within the persistence store architecture. A variety of mechanisms, primarily record level checksums, are used to reduce the chance of missed errors. In addition to record level checksums, the write replication pipeline, which operates at the level of write cache buffers, uses checksums of the entire buffer to detect problems during write replication.

There are three distinct persistence store mechanisms within bigdata:

- Write Once, Read Many (WORM) store, which is based an append-only, log-structured store and is by the data services used to absorb writes. This is often referred to as the “journal”;
- Read/Write (RW) store, which is capable of reusing allocation slots on the backing file and is used for services where we want to avoid decomposing the data into managed shards; and
- Index segment (SEG) store, which is a read-optimized B+Tree file generated by a bulk index build operation on a data service.

Both the WORM and RW persistence store designs support write replication, record level checksums, low-level correction of bad reads, and resynchronization after service joins a failover chain. The index segments are only present on the data service nodes and are part of the mechanisms for dynamic sharding. Once an index segment file has been generated, it is replicated to the other physical data service nodes for the same logical data service instance.

Bigdata provides a strong guarantee of consistency for the nodes in a failover chain such that any physical service instance in the chain may service any request for any historical commit point which has not be released by the transaction manager. This allows us to use *shard affinity* among the failover set for a given logical data service to load balance read operations. Since the members of the failover group have consistent state at each commit point, it possible for any node in the failover set to perform index segment builds. In general, it makes sense to perform the build on the node with the strongest affinity for that shard as the shard’s data is more likely to be cached in RAM.

Metadata service

The metadata service uses a single RW store. The RW store can scale to terabytes of data. A single 1TB RW store instance is sufficient for the MDS to address petabytes of managed shard data³. The metadata service maps the keys for each scale-out index onto the shard which covers a given key range. Failover is handled by write replication for the RW store backing the metadata service. Data services coordinate with the metadata service using full distributed transactions when a shard is split, joined or moved. However, the vast majority of all operations against the metadata service are reads, and those operations are heavily cached by the caller.

Transaction service

The transaction service uses a single RW store to maintain a map of the historical transaction commit points. The data scale is quite small. Failover is handled by write replication. The transaction service is responsible for assigning monotonically increasing transaction identifiers to read-write transactions. As such, the transaction service nodes *must* coordinate their clocks^{4 5}.

Load balancer service

The load balancer maintains a rolling history of performance counters aggregated from the nodes in the network⁶. Those data are backed by an RW store with a suitable history retention policy. Failover for the load balancer is not critical, but is trivially handled by write replication for the backing store holding the performance counter data.

Data service

The data services maintain the index shards. Each data service has a *live* journal, which is a WORM store. Periodically that journal reaches its nominal capacity of 200MB and a new journal is opened while asynchronous processing builds index segments from the data on the old journal. When the view of a shard becomes sufficiently complex, a compacting merge is performed for that shard. When the size on disk of the shard after a compacting merge exceeds the nominal target of 200MB, the shard is split. Shards may be moved from time to time in order to balance the distribution of the write workload on the cluster.

Failover for data services relies on write replication of the live journal and on replication of the generated index segments. Data service instances negotiate for shard affinity. A node with affinity for a given shard will handle reads and index segment builds for that shard.

³ In fact, this is probably a capacity underestimate. The real requirement may be closer to 200MB of MDS data to address 1 petabyte of shard data. We need to perform testing with the MDS using the new RW store in order to obtain a better estimate of this capacity scaling factor.

⁴ In fact, bigdata can also function if the transaction service hands out sequentially increasing transaction identifiers rather than identifiers based on the system clock.

⁵ While synchronization of clocks across the cluster is not required, it is strongly recommended as it makes it much easier to correlate events, log messages, etc. when the nodes have synchronized clocks.

⁶ The aggregation of performance counter data may be separated from the load balancer function in the future, in which case the load balancer itself would be stateless.

Failure modes

Master election

The master election protocol is delegated to zookeeper. The zookeeper quorum is responsible for detecting and handling node death, network partition, and related events. Bigdata services rely on the zookeeper master election decisions and do not use quorums internally. Writes are always directed to the master for a bigdata logical service instance. Reads may be directed to any member of the failover set for the logical service which is current (synchronized with the master).

When a service is deemed “dead” by zookeeper, its connection with zookeeper is invalidated and its ephemeral nodes are synchronously removed from the zookeeper state and events are distributed to watchers. Any node in a failover group may have its connection invalidated for a wide variety of reasons, including transient false positives. Failover occurs as soon as a master has its connection invalidated. Resynchronization is extremely fast when the service is in fact online (a false positive) but may take hours if a new node is being recruited due to the volume of data to be copied. False positives are not uncommon and can arise from a combination of GC policy⁷ and application load.

Read errors

Writes on the disk are assumed to be available for read back⁸. Writes across the replication chain are assumed to provide k instances with the same data. In order to detect bad writes, media rot, and read errors, all records read or written on the disk include checksums. If there is a checksum error on read, then the remaining physical instances for a given logical service will be queried to discover whether any instance has the correct data for that record. If a copy of that record can be discovered whose checksum is valid, then that record will be replicated to the instance(s) reporting bad data for that record.

If a read error can not be corrected, if a write is refused by the IO subsystem, or if the number of read errors is growing then a node should be failed. Before failing a node, it is desirable to recruit a hot spare and add it to the failover set. Once the hot spare is fully synchronized, the bad node may be dropped.

Resynchronization

During resynchronization, nodes are not fully available. Resynchronization proceeds by first releasing historical resources which are no longer used (releasing their space on the disk in case the node was nearing disk exhaustion), then synchronizing the current

⁷ A full GC pass over a 10G heap can take several seconds during which the JVM suspends the application threads. The use of an incremental GC policy can alleviate this at the expense of ~20% throughput. We are also working to better manage RAM and thread demands in an effort to reduce the rate of false positives.

⁸ It is possible to institute a relatively efficient write-flush-read-verify policy in combination with allocating sufficient write cache buffers. However, this can only guard against bad writes in combination with appropriate configuration of the disk hardware cache and even then can not guard against *poor* writes, such as when the disk head flies high during the write or when the disk head is not sufficiently well aligned.

journal, then catching up on index segments and journal files requirement to materialize shard views in reverse time order⁹ ¹⁰. This allows us to bring a node online for failover writes very quickly and to catch up quickly after a transient disconnect. Reads on historical commit points are only allowed as the earliest read time grows backwards towards the current release time for the cluster.¹¹

If a node is nearing exhaustion of its local disk resources, then it will issue an alert and signal the transaction service that it should advance the release time so that the node may reclaim the disk space allocated to the oldest historical commit points. While this will reduce the history retention period for the database, and hence the window for point in time rollback, that is generally preferable to a hard error associated with disk exhaustion.

Point in time rollback

The transaction service may be configured to retain historical commit points for a specified period of time. Applications can read from those historical commit points by requesting a read-only transaction associated with the desired commit time. Point in time rollback is also supported within this history window. When a rollback is requested, the intervening commit points remain available to readers, but new operations will see a version of the database rooted at the rollback time. That is, rollback effectively creates a version fork. Applications may then selectively roll forward (in an application specific manner) those writes which they wish to preserve.

Rollback is a synchronous operation. It may be applied to a *namespace* (a collection of indices corresponding, for example, to a quad store) or to the entire bigdata federation. The rollback operation *replaces* the current definitions of the shards on each data service with the historical definitions of those shards, but specifies the current journal on the data service as the buffer to absorb future writes for the shard.

Hot spares

The cluster should be provisioned with hot spares. Machines are made available as hot spares using an administrative command. Machines are recruited from the “hot spare” pool as necessary in order to replace failed or failing nodes. An administrative action may be used to fail a node if it is experiencing problems or to temporarily increase the replication count for a logical service prior to failing a problematic node. Hot spares should be pre-allocated in a manner conducive to the location aware allocation policy. For example, by having one hot spare per rack.

⁹ The remaining members of the failover set can share the responsibility for replicating resources (journal files and index segment files) onto the node. This can be done based on shard affinity, but an additional protocol is necessary to ensure that all resources are migrated, even those which are not hot and hence do not have an assigned affinity.

¹⁰ The RW store may require a “write log” blocks in addition to its “delete log” blocks in order to facilitate resynchronization of missed writes during transient failover. (Martyn writes “If the delete log works because we are not immediately re-allocating free'd storage, then we should be able to rely on the allocation blocks to indicate newly written data. The “old” store could pass it's allocator blocks for comparison and return of data updates.”)

¹¹ Thus a new node may be “online” for the current view of a shard while it backfills older data from the rest of the failover group.

An administrative action is required in order to increase or decrease the number of logical data services in a federation. When a new logical data service is created, it is populated with k machines and the data is dynamically rebalanced to load those machines. When a logical data service is removed, the shards on that logical data service are first redistributed to the remaining logical data services and then the machines are returned to the pool of hot spares.

Management

This section is TBD.

Alerts

TBD. Pluggable alerts for various critical measures (swapping, disk exhaustion, node failure, etc). Pluggable reporting mechanism (Nagios, etc).

Location aware allocation

TBD. The basic pattern is to locate two out of three on the same rack and the third on a different switch.

Offsite replication

TBD. This design does not explicitly address offsite replication. In principle, write replication can be used to achieve offsite replication if sufficient bandwidth is available.

Backup and restore

TBD. This design does not address backup and restore, as backup operations quickly become untenable for large data sets. Full backup requires the snapshot of all new resources (journals and index segment files) and capture of a write log for RW store files. Full restore requires a cluster with an identical number of machines provisioned in a similar manner. The files must be restored in place and the services then restarted.

Future directions

This design offers a simple execution and management when compared to a *shared disk* design, at the expense of somewhat less flexibility. The concept of *shard affinity* provides a bridge toward a *shared disk* design. However, a shared disk design goes further in allowing a pool of resources to be dedicated to specific operations, such as parallel closure, or low-latency vs. long running queries,