

Bigdata® High Availability Quorum Design

Bigdata® High Availability Quorum Design	1
Introduction.....	2
Overview.....	2
Shared nothing	3
Shared disk.....	3
Quorum Dynamics	4
Join, Meet, Leave, Break	5
Write pipeline.....	6
Service Initialization	7
Synchronization	7
Failure modes.....	8
Quorum can not meet.....	8
Odd man out.....	9
Hot spares.....	9
Robust messaging	10
Point in time recovery	10
Appendix A – Detailed zookeeper integration design	11
Integration	11
Persistent quorum state	12
Zookeeper Watchers	14
Appendix B – Detailed synchronization design.....	15
WORM.....	16
RW	16
Data Service	16
Metadata Service.....	17
Transaction Service.....	17
Appendix C – High Availability APIs.....	17
Appendix D – Notifications and alerting	17
Appendix E – High Availability Management Tools	17
Appendix F – Additional Resources	17

Introduction

Bigdata® uses a quorum model for its persistent services. The high availability quorum model builds on existing distributed systems infrastructure components, including jini and zookeeper. Each service has a replication factor k , which is an odd number, and a current replication count n . A quorum “meets” when $(k+1)/2$ services (e.g., 2 out of 3, or 3 out of 5) agree on their shared state. Each quorum has a *leader* and zero or more *followers*. Writes are only permitted for “met” quorums and must be directed to the quorum leader. The services in a quorum are arranged in a write pipeline which efficiently replicates low-level writes (cache blocks) across sockets. All writes are check summed to detect read errors and failover reads are supported so bad reads do not cause runtime errors. Hot spares may be automatically recruited to replace failed nodes. New services can inherit from the high availability architecture. For example, a highly available cache fabric could be derived using the non-blocking cache to buffer an RW journal used as a local persistence store.

Overview

Bigdata® uses a quorum model for its persistent services. A highly available service has a replication factor k , which must be an odd integer (1, 3, 5, 7, etc.). Only services with a $k \geq 3$ or more are highly available. The quorum meets when $(k+1)/2$ nodes have an agreement on persistent state of the logical service. This agreement is formed around the *lastCommitTime* in the *current root block* for that logical service, which is a concise summary of the persistent state for the service.

A met quorum is comprised of at least $(k+1)/2$ nodes. Each met quorum has a *leader* and may have zero or more *followers* (a singleton quorum has no followers and is only permitted for $k=1$, which is not a highly available configuration). The quorum state is summarized by a *quorum token*. A distinct quorum token is assigned by the leader each time a new leader is elected. Each node in the quorum is assigned a *quorum index* in $[0:k-1]$. The index of the quorum leader is always ZERO (0). The remaining nodes in the quorum are *followers*. All writes must be directed to the quorum leader and are replicated along a write pipeline to the followers based on their assigned quorum *index* order. All writes are check summed to detect read errors and failover reads are supported so bad reads do not cause runtime errors. In order to ensure data consistency, writes will not be accepted unless there is a quorum.

The quorum model ensures that updates can not be lost following a restart. Consider a scenario with $k=3$, in which case the minimum quorum size is 2. If one node fails, then a quorum still exists. If there is a sudden failure of the cluster, then on restart two nodes will have an agreement as to the *lastCommitTime*. A quorum will form around those nodes and the third node will resynchronize its persistent state with the quorum and then join the quorum. If we did not insist on a quorum, then it would be possible to have two nodes fail and continue to accept writes on the third node. However, if there were a sudden failure of the cluster, each node could now have a distinct *lastCommitTime*. On restart, the restored state would depend arbitrarily on which node(s) restarted with the cluster and the order in which they restarted. Since each node has a different

lastCommitTime, there is no agreement about the actual service state. This situation can be reconciled in an arbitrary manner by either rolling back all of the online nodes to their earliest common *lastCommitTime* or by forcing the trailing nodes to synchronize with the leading node. The former option can result in lost updates while the latter option implies that the clients may not have seen the commit and might retry unsafe operations as a result. The quorum model avoids the uncertainty of situations such as this and avoids the essentially arbitrary measures to impose consistency on the service after a restart. It does this by refusing to accept writes when a service is under quorum.

Shared nothing

This design document considers a bigdata federation which has been configured to retain a limited amount of history, for example, 24 hours, or 4 weeks, and is being used in a *shared nothing* configuration. For a shared nothing deployment, the persistent state of the services is stored locally by each node. The shared nothing architecture is simple and scales well. The synchronization time for a shared nothing node is directly related to the amount of data to be transferred.

Data services may have an *affinity* for views of various shards. In the shared nothing architecture, shard affinity improves utilization of local disk, memory, and CPU resources by distributing the read workload according to shard affinity.

Shared disk

If a bigdata federation is configured to retain large amounts of (or infinite) history, then the persistent state of the services must be stored on a parallel file system, SAN, or NAS. This is a *shared disk* architecture. While quorums are still used in the shared disk architecture configuration, the persistent state is centralized on inherently robust managed storage. Unlike the shared nothing architecture, synchronization of a node is very fast as only the live journal must be replicated to a new node (see below).

In order to rival the write performance of the shared nothing architecture, the shared disk architecture uses write replication for the live journal on the data services and stores the live journal on local disk on those nodes. Each time the live journal overflows, the data service opens a new live journal, and closes out the old journal against further writes. Asynchronous overflow processing then replicates the old journal to shared disk and batch builds read-only B+Tree files (index segments) from the last commit point on the old journal. Each time a new index segment file is generated, it is replicated to the shared disk. Once the index segment builds are complete, the old journal may be released from the local file system.

The data services hold the vast majority of all persistent state in a bigdata® federation. This state is divided between journals, which buffer writes and allow read back from each commit historical commit point, and index segment files. The data service uses a WORM journal, so writes are block append but reads are random IOs. For this reason, it makes sense to cache journals locally.

Index segment files are the majority of the persistent state for a data service. However, unlike the journal, index segment files are optimized for efficient remote reads and often have significantly larger branching factors than the corresponding B+Tree on the journal, so IOs are generally larger and more efficient. In the index segment, the B+Tree nodes are in one region of the file in key order, and are typically read into memory using a streaming IO when the index segment is opened. Likewise, the leaves are in key order in another region of the file and are double linked for efficient traversal. Special iterators are also able to *pin* a leaf-range of the index segment for a shard view, using one IO per index segment. Finally, index segment files may have a bloom filter which can be used to avoid point lookups proven not to exist by the bloom filter.

Therefore, in the shared disk architecture, when a node develops an *affinity* for a shard, it locally caches journals required to read on that shard. Since the active read set of a data service node is significantly smaller than the total persistent state of the data service, the journals may be cached on solid state disk (SSD), providing ultra fast access time. Using a cache management strategy, resources which are no longer in active use are simply deleted from the local file system since their master copy is on shared disk.

The shared disk architecture provides a good balance between throughput (writes are buffered on local disk), managed storage, and fast reads using locally cached resources. However, the shared disk architecture is more complex and requires the deployment of either a parallel file system, SAN, or NAS in addition to the bigdata® federation.

TBD: Expand on the shared disk architecture in a separate design document. The shared disk architecture is ultimately more flexible, but it is more complex and is not yet implemented.

Quorum Dynamics

A quorum has a number of states and state transitions. A *met quorum* has at least $(k+1)/2$ services with an agreement on their shared state as summarized by the *lastCommitTime* on the current root block for the service. A met quorum has a valid current quorum token. The services joined with a met quorum will share the same quorum token value. The leader will be writable. The other services will be read only. The leader and followers will be arranged in a write pipeline. That pipeline is organized by the host and socket address of the next quorum member to which the service will relay cache blocks written by the leader to its local persistence file.

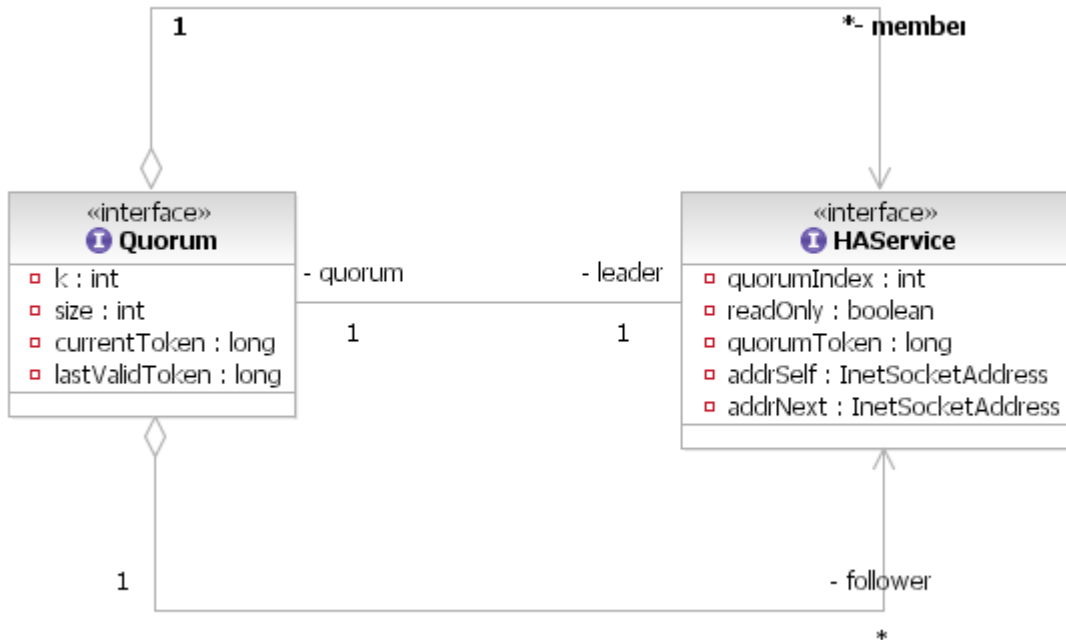


Figure 1 A quorum and a highly available service.

A quorum which is not met can not accept application level writes. Its current quorum token will be ZERO (0), which is an invalid value. The member services of the quorum will have a quorum token of ZERO (0) and will be read-only (low-level writes used for synchronization bypass the read-only nature of the services public interfaces).

Join, Meet, Leave, Break

A **quorum join** assigns a *quorum index*. A member service joins the quorum when there are at least $(k+1)/2$ member services with a shared agreement on the *lastCommitTime*. The first service to join the quorum is the *leader* and has a quorum index of ZERO (0). The services which join the quorum after the leader are *followers* and each is assigned a quorum index which reflects its join order (1, 2, ..., k-1). The current quorum token will be ZERO (0) until the quorum has met, at which point the leader will assign a token to the quorum.

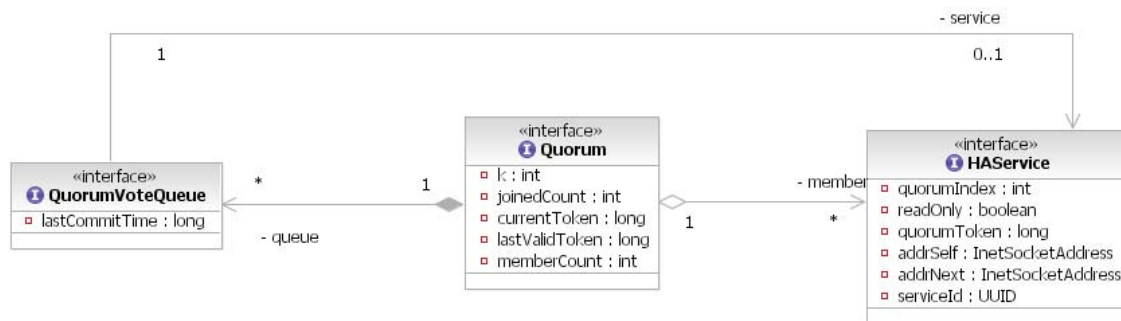


Figure 2 Services queue up for the lastCommitTime under which they are willing to join a quorum.

A **quorum meet** occurs once $(k+1)/2$ services have joined the quorum. When the quorum meets, the leader publishes the new quorum token, which is the value of the last published quorum token *plus one*. This value is published both on the quorum and on the leader. Once the followers acknowledge that they have the current token (by marking themselves with the token in zookeeper), the leader marks itself as writable and does a low-level *abort* in order to reload the commit record, commit record index, and name2addr index from the current root block. At this point the quorum is online and writes will be accepted by the leader. Clients can watch the quorum in zookeeper and begin writing on the leader once it marks itself as writable.

A **quorum leave** clears the quorum *token* to indicate that the journal is not part of any valid quorum, forces the service into a read-only mode, and does a low-level *abort* in order to discard any buffered writes. As part of the *abort* protocol, the service will reload the current root block and discard any buffered writes.

When a follower (or a member service trying to synchronize with the quorum) leaves, an error is thrown. This condition is trapped by the leader. The leader reconfigures the write pipeline to cut out the service which left the quorum and then retransmits the last write cache block along the reconfigured write pipeline. Applications do not notice quorum leave events for followers.

When the leader leaves, an error will be thrown back to the application. The application must trap this exception, query zookeeper for the new quorum leader, and then resubmit the operation to the new leader. This behavior can be encapsulated by a smart proxy pattern.

A **quorum break** occurs when the number of services with a shared agreement on the *lastCommitTime* falls below $(k+1)/2$. Quorum breaks can occur for a number of reasons, including normal shutdown, network partition, zookeeper session timeout, confluence of independent errors, etc. The application handles a quorum break in a manner similar to a leader leave. It must watch the quorum until it meets, then determine the leader, and finally resubmit the operation to the leader. Again, this behavior can be encapsulated by a smart proxy pattern.

The purpose of the quorum break is to preserve the consistency of the system by not permitting application level writes when the logical service is under the minimum quorum size. Quorum breaks can be trivial events and are normally healed when the cluster comes back online, when the network partition is cured, when a new zookeeper session is assigned, etc. Quorum breaks are much less common than quorum leaves since they require at least two failures, even at $k=3$.

TBD: A quorum join/leave/meet/break/sync graphic.

Write pipeline

The services are arranged into a **write pipeline** by their quorum index order. Only the leader accepts writes from the application, and then only once the quorum has “met”.

The followers are “read-only.” Writes are relayed from the leader along the write pipeline to the followers using an efficient low-level transfer of write cache blocks. The write cache blocks are relatively large and are relayed to the downstream node in chunks as they are read from the socket buffer. For the WORM journal, these are 1M blocks containing bytes to be written at a specific file offset. For the RW journal, these are 1M blocks containing a set of records with inline metadata about the length and the file offset at which each record will be written.

TBD: A graphic for the write pipeline.

Service Initialization

When a new journal is created, or when an existing journal is re-opened, it is fully initialized from its local configuration information using a *singleton* quorum manager. This is done so that the journal may initialize itself before considering the dynamics of a broader quorum. Once the journal is open, the quorum manager is *replaced* with a quorum manager instance for the logical service. The replacement of the quorum manager triggers a *quorum leave* since the journal is no longer part of its singleton quorum.

A new journal will have two identical root blocks. The *lastCommitTime* will be ZERO (0) in those root blocks. When all services in the quorum are new, each of the services will have a *lastCommitTime* of ZERO (0) and the criteria for a quorum meet are trivially satisfied. When the resulting quorum “meets,” the leader will begin accepting writes.

Synchronization

Since a quorum meets when there are $(k+1)/2$ services which agree on a *lastCommitTime*, there may be up to $((k+1)/2) - 1$ services which did not join the quorum. For $k=3$, there can be 1 service which is not in the quorum; for $k=5$, there can be two services which are not in the quorum, etc.

A service can fail to *meet* with a quorum for a variety of reasons, including: machine down, JVM down, network partition, zookeeper session timeout, a disagreement about the *lastCommitTime*, etc. Such services must *synchronize* with the quorum before they can join. The goal of synchronization is to guarantee eventual consistency without allowing any intermediate illegal states (the persistent state of the service must remain self-consistent). This is simplest for the WORM journal and most complex for the RW journal.

Synchronization requires that the service compute a delta between its current state and the current state of the quorum. Correcting that delta can require acquiring state from the quorum or, for some rare conditions, discarding state locally. Synchronization deltas are point to point and may flow from any members of a met quorum. Precisely how this delta is computed and how synchronization takes place depends on the nature of the service and is discussed in detail in Appendix B – Detailed synchronization design.

Synchronization is simplest for the WORM journal due to its “write once” semantics. Likewise, the data service essentially has WORM semantics (data is never overwritten, even though historical commit points can be released). Synchronization is most complex for the RW journal since it manages allocation slots which can be overwritten at different points in the history of the store. Also, while synchronization for WORM journals and data services requires only ordered reads and writes, synchronization for RW journal requires scattered reads and gathered writes and is thus more expensive.

During synchronization, the service joins the write pipeline even though it is not a joined with the quorum. This allows it to receive updates, but it can not participate in commits until it has captured the delta. Data may be replicated from any service in the met quorum during synchronization and it is possible to scatter reads across the met quorum in order to distribute the IO workload.

The service locally tracks the remaining delta so it can continue to make progress even if its connection with the quorum is intermittent. Once the service is caught up on the delta, it will join the quorum at the next commit point.

Failure modes

A distributed system can fail in a nearly infinite number of ways. bigdata® uses a few simple mechanisms to protect against broad categories of failure. Those mechanisms include:

- Record level checksums, which are used to detect bad reads.
- Block level checksums, which are used to detect errors when replicating writes.
- Failover reads, which are used to handle bad reads gracefully by reading on another node in the quorum.
- Quorums, which disallow writes when the data consistency could not be guaranteed.
- Service failover, which handles service death, network partition, planned or unplanned downtime, etc.
- Hot spares, which allow a service to automatically re-replicate when a node goes down or has been partitioned.

These mechanisms for handling failures build on several pieces of existing infrastructure, including:

- Jini, which provides distributed service registry and discovery.
- Zookeeper, which provides a global synchronous event notification.

The sections below outline some specific kinds of failures and how they are handled.

Quorum can not meet

It is possible to have more than $(k+1)/2$ services online and not have a quorum because there is no agreement of $(k+1)/2$ services on a shared *lastCommitTime*. This is highly unlikely and can arise only from a combination of rare events. If this situation does arise, it will require operator intervention.

For example, such a situation can arise if only $(k+1)/2$ services were in the quorum if there is *also* a failure of the 2-phase commit protocol. A quorum may vote “no” on the *prepare* message, which is a rejected commit but not a protocol error. A protocol failure is more serious involves an error in the handling of the *commit* message itself. A protocol failure can occur as follows: Given $k:=3$ and minimum quorum of 2 services, both services vote “yes” on the *prepare* message of a 2 phase. The *commit* message is then issued and one service commits (it updates its root blocks) while the other service fails to commit (it does not update its root blocks). If the service which did commit also refuses to rollback the commit (that is, it does not restore its root blocks), then this leads to a disagreement between the services over the *lastCommitTime*. This is a failure in the 2 phase commit *protocol* itself which does not protect against this rare combination of failures. When the 2 phase commit protocol fails (as opposed to having the quorum vote “no” on the *prepare* message), each service will leave the quorum so a new quorum can try to meet.

When the quorum can not meet, the operator must choose the *lastCommitTime* which will become the new shared state and then force the services to synchronize on that state. In the case outlined above, the correct *lastCommitTime* is the commit point associated with the service which *did not* commit since the application will block on the commit and hence will not have been notified of a successful commit. Assuming $k = 3$ and service C was offline when the commit protocol failed, let A be the service which committed but rejected the rollback and B be the service which failed the commit, then the “correct” last commit time is the one associated with B. If there had been no intervening commits since C left the quorum, then a quorum could form around (B,C) if C comes back online. Otherwise, when C comes online, there will be a distinct last commit time associated with each service with $C < B < A$ in this example. For this example, the operator should force C and A to synchronize to B. The quorum will meet as soon as either C or A is synchronized.

Odd man out

Given $k=3$ and a quorum of 3 nodes, it is possible for all nodes to vote “yes” to the “prepare” message but have 1 or 2 of the nodes fail during the “commit” message while the third succeeds. If power were to be lost across the data center, the third node would have a more recent *lastCommitTime*, but the application would not have seen a successful “commit.” On restart, the quorum would form around the two nodes which failed the commit, not on the 3rd node. This is important in case the application had any after actions linked to that commit since those actions would not have been executed.

Hot spares

The purpose of a hot spare is to automatically restore a full quorum. Synchronizing a hot spare is a heavy operation since all persistent state must be replicated from the nodes in the met quorum. There are different kinds of events which cause a quorum leave, but not all of these should be cured with a hot spare. For example, the following events should not cause a hot spare to be recruited since they can be cured in other (lighter) ways:

- Scheduled maintenance on a node;
- Zookeeper session timeouts, which can be trivially cured by a reconnect;

- A network partition event if it can be cured by reorganizing the write pipeline (all nodes are online, but there is a network communications failure when propagating messages in quorum index order);
- A network partition event such that the leader can not communicate with the transaction service, but another node in the quorum can;

Synchronization of a hot spare is identical to resynchronization.

TBD: A graphic for hot space recruitment.

Robust messaging

Both the application and the bigdata services must be robust to quorum state changes, including when individual services join or leave the quorum and, as a special case, when the quorum leader fails over. Robust messaging is necessary for applications to make progress on long running operations in the face of both transient and long lived failover events.

We currently handle a similar issue pertaining to dynamic sharding by throwing a “stale locator” exception. That exception is trapped, and the client automatically discovers the new locators for the key(s) and reissues the request to the appropriate shards. The logic to handle the stale locators is encapsulated in the client’s view of a scale-out index and in other code which is inherently aware of shard locations. This approach has the virtue of simplicity, but could be refined to minimize data movement by separating the RMI requests from the data payloads. With this refinement, the client’s request will be queued on the service, but the payload will not be transferred until the service is about to process the request (this could be anticipated to reduce latency).

Robust operations against the leader (writes) or the quorum (reads) could be achieved in a similar manner when the quorum state changes. Quorum state changes can arise for a variety of reasons ranging from the trivial (zookeeper session timeout), to the routine (planning maintenance), to outright errors (network partitions, JVM crashes, etc). In all cases, the client will eventually notice an exception and must decide if the cause was a service error, an RMI error, or a quorum state change. This can be most easily accomplished by querying zookeeper to determine whether the quorum is still met (same quorum token) and whether the node which was the target of the message failed over (a different value in zookeeper for the node’s quorum join/leave counter). If the exception is correlated with a quorum break or a quorum leave, then the operation can be reissued. Otherwise, the operation will fail. If an operation is to be reissued, the client can inspect zookeeper in order to determine the new leader (writes) or the node which has affinity for the shard (reads). This logic can be encapsulated within a smart proxy pattern for RMI requests, making them automatically robust to errors linked to quorum state changes.

Point in time recovery

Point in time recovery is handled as a roll forward operation. Each B+Tree or B+Tree shard has a checkpoint record and an index metadata record. Together, these define the

current state of the B+Tree or B+Tree shard. In particular, for a B+Tree shard, there is an ordered array of journals and/or index segments having the persistent state of the shard.

In order to “rollback” the database state to some historical commit point, we actually write a new commit point using the ordered list of journal and/or index segment resources from the historical commit point, but add an empty B+Tree on the live journal as the first entry in that list. Any new writes will be absorbed by the B+Tree on the live journal while reads will read through to the ordered list of historical resources. Point in time rollback is thus a fast *roll forward* operation. Since point in time recovery does not overwrite old data, applications are able to inspect and recover data from within the rollback period.

Bigdata also supports lightweight version forking. A fork is nearly identical to point in time recovery, except that the new empty B+Tree on the live journal will be registered under a name which captures the version information. For example, “foo#1” is the first shard of the scale-out index “foo”. After a version fork, “foo#1” will be untouched but “foo#1.1” will be defined. The version fork (“foo#1.1”) will share the same historical resources as “foo#1”, but new writes on the fork will be independent of writes on the original version.

Appendix A – Detailed zookeeper integration design

Zookeeper is inherently robust and uses a quorum model internally. Bigdata® uses zookeeper as discussed below to realize a robust quorum protocol with zookeeper’s public APIs. This design delegates many of the complexities of the management synchronized shared state to zookeeper and facilitates the use of quorum models with other zookeeper and jini aware components.

Integration

Bigdata® and zookeeper coordinate to maintain a quorum for each logical service using patterns similar to those for master elections, but with extensions for quorum semantics and synchronization of nodes so they may join a quorum.

Zookeeper is an inherently robust distributed system based on its own quorum model. Zookeeper provides a hierarchical arrangement of zookeeper nodes (znodes) and provides a single consistent view of the state of that system. Each node may have a byte[] datum and an ordered list of children. Nodes may be persistent or *ephemeral*. Ephemeral znodes represent a *zookeeper connection* by a specific service and may not have children. Clients may establish watchers for state changes in the datum or children of a zookeeper node. Watched state changes result in notices. Zookeeper provides a strong guarantee that those notices are delivered to clients before the next state change. However, to maintain high throughput, clients must then request the current state of watched node or children. This means that clients respond asynchronously to synchronous system state changes.

TBD: Show the state evolution in zookeeper for a quorum.

Persistent quorum state

The persistent state of the quorum is laid out in zookeeper as follows:

```

.../<logicalService> {k, logicalServiceUUID}
    /quorum {currentToken, readOnly, lastValidToken}
        /member
            /ephemeral-service-znodes {ServiceUUID}
        /votes
            /<lastCommitTime>
                /ephemeral-service-znodes
        /joined
            /ephemeral-service-znodes
                {ServiceUUID, readOnly, quorumToken}
        /pipeline
            /ephemeral-service-znodes
                {ServiceUUID, addrSelf}
    
```

The zookeeper paths are defined a follows:

.../<logicalService>	The path dominating all state for that service in zookeeper. The znode is named by the ServiceUUID (see below). This znode is created when the logical service is first declared and has a number of other children not related directly to quorum management.
.../quorum	The path dominating all state for the quorum for that logical service in zookeeper.
.../quorum/member	The path whose children are the <i>ephemeral</i> znodes representing active zookeeper connections for services that self-identify as physical instances of the logical service.
.../votes/	The path dominating the voting procedure used to decide when at least $(k+1)/2$ services have an agreement on the last commit time. A service votes when it starts, but does not vote at each commit point. If the service leaves the quorum, then it casts a new vote based on its last commit time for its current root block. Each service may vote precisely once at any given time. The service must retract its previous vote before it votes again.
.../votes/<lastCommitTime>	The votes are organized under znodes whose name is the lastCommitTime for which the service will cast its vote. The “vote” is just the <i>ephemeral</i> token for the zookeeper connection for the service casting that vote. Voting is a safe procedure. If the designed zpath does not exist, the service must create it before it votes. If there has been a concurrent create of the znode, then that error

	is ignored. If the service is the last one to retract its vote from a commit time, then it must delete the zpath. If there has been a concurrent delete of the znode or if the znode is not empty because of a concurrent vote for that commit time, then the error is ignored.
.../joined	<p>The path dominating all services currently joined with the quorum. The leader is the first child under this node. The other children are followers. When the number of children exceeds $(k+1)/2$, the quorum is “met”. When the quorum meets, the leader will initiate various state changes resulting in a writable quorum. These include: adding itself to the .../pipeline, assigning a new quorum token to itself, watching the followers until they have copied the current quorum token, and finally marking itself as writable. At that point the quorum is ready to receive writes.</p> <p><i>TBD: In order to simplify the quorum state for clients, the leader also takes responsibility for copying the current quorum token and updating the state of the readOnly flag on the .../quorum znode. [Keep this or drop it? The quorum behavior will be fully encapsulated by client libraries so this could in fact make it more difficult to tell the actual state of the quorum since there is more than one place to look for this information.]</i></p>
.../joined/ephemeral	The ephemeral token for the zookeeper connection for a service joined with the quorum. An RMI proxy for the service may be discovered using its ServiceUUID.
.../pipeline	<p>The path dominating the write pipeline state. The write pipeline order is given by the order of the children queued under this znode. The leader is <i>always</i> the first child in the write pipeline.</p> <p>The write pipeline can also contain services which are <i>synchronizing</i> with the quorum in addition to those which are <i>joined</i> with the quorum. For this reason, the number of children for this node does NOT tell you whether or not the quorum is ready to receive writes.</p>
.../pipeline/ephemeral	The ephemeral token for the zookeeper connection for a service joined with the quorum. The address at which that service will accept replicated writes is given as part of its data. An RMI proxy for the service may be discovered using its ServiceUUID.

The data items for the different znodes are indicated in curly braces above and are defined as follows:

k	The service replication factor.
currentToken	The current quorum token or ZERO (0L) if the quorum is not met. This is updated by the leader when the quorum meets.
lastValidToken	The last valid quorum token. This is initially ZERO (0). This is updated by the leader when the quorum meets.
ServiceUUID	The unique identifier for the service, stored as a UUID (this is the practice used by bigdata). The UUID can be converted to a ServiceID. An RMI proxy for the service can be discovered using jini from that ServiceID.
logicalServiceUUID	The UUID for the logical service.
readOnly	A Boolean flag indicating whether or not the service will accept application level writes.
quorumToken	The current quorum token for the service and ZERO (0) if the service is not joined with a quorum. This value is set by the service when it joins with the quorum.
addrSelf	The Internet address and socket port where this service will listen for payloads relayed along the write pipeline.

Zookeeper Watchers

Zookeeper provides a notification mechanism know as a *watcher* by which a service can be signaled when there is a state change in the existence, data, or children for a given zpath. When a service starts, it establishes a pattern of watchers so it can notice critical state change events in zookeeper pertaining to quorum management. The necessary watcher patterns basically correspond to the path hierarchy for the quorum as described above. Some specific watcher patterns are called out below. The watcher dynamics are encapsulated by utility classes in bigdata®.

znode / zpath	What is watched	Watcher description
.../<logicalService>	data	The replication factor of the logical service is stored here. Nodes need to be aware of the replication factor in order to handle changes in the target replication factor for a quorum.
.../quorum	data	The current quorum token is stored here. If the quorum breaks, then the token will be cleared to ZERO (0). Watchers will notice quorum meets and breaks.
.../votes/<lastCommitTime>	children	The service must watch the children of the <i>lastCommitTime</i> for which it has voted in order to recognize when $(k+1)/2$ nodes have reached an agreement on the persistent state of the service. When this occurs, the service must add itself as an ephemeral child to the .../quorum/joined

		path.
.../pipeline/	children	A service which is joined with the quorum must watch the children of this zpath so it will know how and when to configure the downstream address to which it will relay write cache blocks.
.../joined	children	<p>A service which joins the quorum must watch the children of this zpath so it can notice when the <i>leader</i> is elected, when service <i>leaves</i> the quorum, and when the quorum <i>breaks</i>.</p> <p>The leader watches the children in order to notice when they have all copied its quorum token, at which point it marks itself as writable and the application can write on the quorum.</p>
../joined/children[0]	data	<p>A service which joins the quorum must watch the data of the leader, which is always at index zero in the list of children. When the leader assigns itself a quorum token, the service must add itself to the .../pipeline/ children and then copy that quorum token onto itself.</p> <p>A client that desires to notice when the quorum is ready to receive writes must watch this node as well. When the leader marks itself as writable, the client can begin writing on the quorum.</p>

Appendix B – Detailed synchronization design

We are faced with some choices concerning how fast we can bring a node online. The delta is computed with reference to the last commit point on the node. However, the quorum will often already have some buffered writes, some of which may have already been replicated along the write pipeline. Because of this there are some options regarding how the delta is computed, when the service joins the write pipeline, and how quickly the node can be synchronized. This is especially true in the case of long running transactions, which are generally used only on bulk loads for a non-clustered deployment. For the clustered database, updates are generally shard wise atomic and occur with great frequency under write high workloads.

In the degenerate case where there are no writes on the pipeline since the last commit, the node can join the write pipeline immediately and will join the quorum as soon as it has captured the delta.

For cases where there are already writes since the last commit, the options are as follows: (1) If we join the write pipeline immediately, then we must either (a) obtain a delta starting in the middle of the current write set, which is easy for the WORM; or (b) wait until the next commit point and obtain a second delta which we also need to capture. Alternatively, (2) we can then wait until the quorum reaches a commit point and then join the write pipeline, which brings us back to the degenerate case but synchronization does not start until the next commit point.

Both (1b) and (2) mean that we must wait at least two commits before the node can be synchronized since, even when the node joins the write pipeline immediately, there may be writes already buffered on the leader. Only with option (1a) can we begin to synchronize immediately and join the quorum at the first commit after we have captured the delta.

Synchronization for the different services is as follows:

WORM

The service looks at the current root block of the leader and requests all bytes between the *nextOffset* for the service and the *nextOffset* for the leader. The synchronization state is simply the offset of the last byte replicated onto the backing file.

RW

TBD. This has already been summarized in the HABranch.txt file. Gather and organize those notes and the notes from the end of this document here.

Data Service

The data service has a “live” journal, which is synchronized exactly the same manner as a WORM journal. Once the data service joins the write pipeline, it will also begin to receive newly built index segment files. In addition the data service must replicate any historical journals or index segment files. These files are replicated moving backwards in commit time on a journal by journal basis so the data service is able to come online as quickly as possible for current reads and slowly builds up its historical views.

As each journal is received, the data service queries the quorum to determine all shard views which could be read on using that journal and demands the index segment files for those shards. Once it has received those files, it requests the prior journal from the quorum and begins to extend its commit history backwards, capturing more and more history locally. In the event of a failure which would reduce the logical data service to beneath a met quorum, it is possible to simply sacrifice some history and bring the data service online as soon as all views for the live journal have been replicated.

The synchronization state for the data service includes the delta the journal it is currently synchronizing and the set of index segment files it requires for the views on that journal.

Metadata Service

The metadata service uses a RW journal. High availability is per the RW journal.

Transaction Service

The metadata service uses a RW journal. High availability is per the RW journal.

Appendix C – High Availability APIs

TBD

Appendix D – Notifications and alerting

TBD

Appendix E – High Availability Management Tools

TBD

Appendix F – Additional Resources

For more information on the bigdata architecture, see:

- <http://www.systap.com/bigdata.htm>
- <http://www.bigdata.com/blog>
- http://www.bigdata.com/whitepapers/bigdata_architecture_whitepaper.pdf
- http://www.bigdata.com/whitepapers/bigdata_ha_whitepaper.pdf