Waltz Internals

Introduction

Waltz is a distributed/replicated write-ahead log. It aims to be a generic write-ahead log to help a micro-service architecture to perform reliable/consistent transactions in a distributed environment. Waltz uses a quorum for the guarantee of persistence and consistency. It also provides a machinery for concurrency control which is essential for a transaction system to prevent inconsistent transactions from being entered into the system. Waltz provides a single image of globally consistent transaction log to micro-services.

A basic usage pattern in micro-service architecture is the following. A micro-service creates a transaction data from a request from outside world and its current state (database state). The transaction data is a packet that describes intended data updates. Then the micro-service sends it to Waltz before updating its database. Once the transaction is persisted in Waltz, the micro-service can safely updates its database. Even when a micro-service failed, it recovers its state from the Waltz log. Furthermore, the log can be consumed by other services to produce derived data such as a summary data and a report. They can produce consistent results independently from the main transaction system. It is expected that this will reduce direct dependencies among micro-services and makes the whole system more resilient to faults.

In addition, Waltz has a built-in concurrency control which provides a non-blocking optimistic locking. The granularity of a lock is controlled by applications and the lock must be explicitly specified in a transaction. This can be used for preventing two or more micro-service instances from writing inconsistent transaction data into the log.

Waltz log is partitioned. Partitioning provides higher read/write throughput by reducing contentions. The partitioning scheme is customizable by applications.

All transaction data are replicated to multiple places synchronously. Waltz uses a quorum write system. As long as more than half of replicas are up, Waltz is available and guarantees consistency of data. Replicas may be placed in different data centers for disaster resilience.

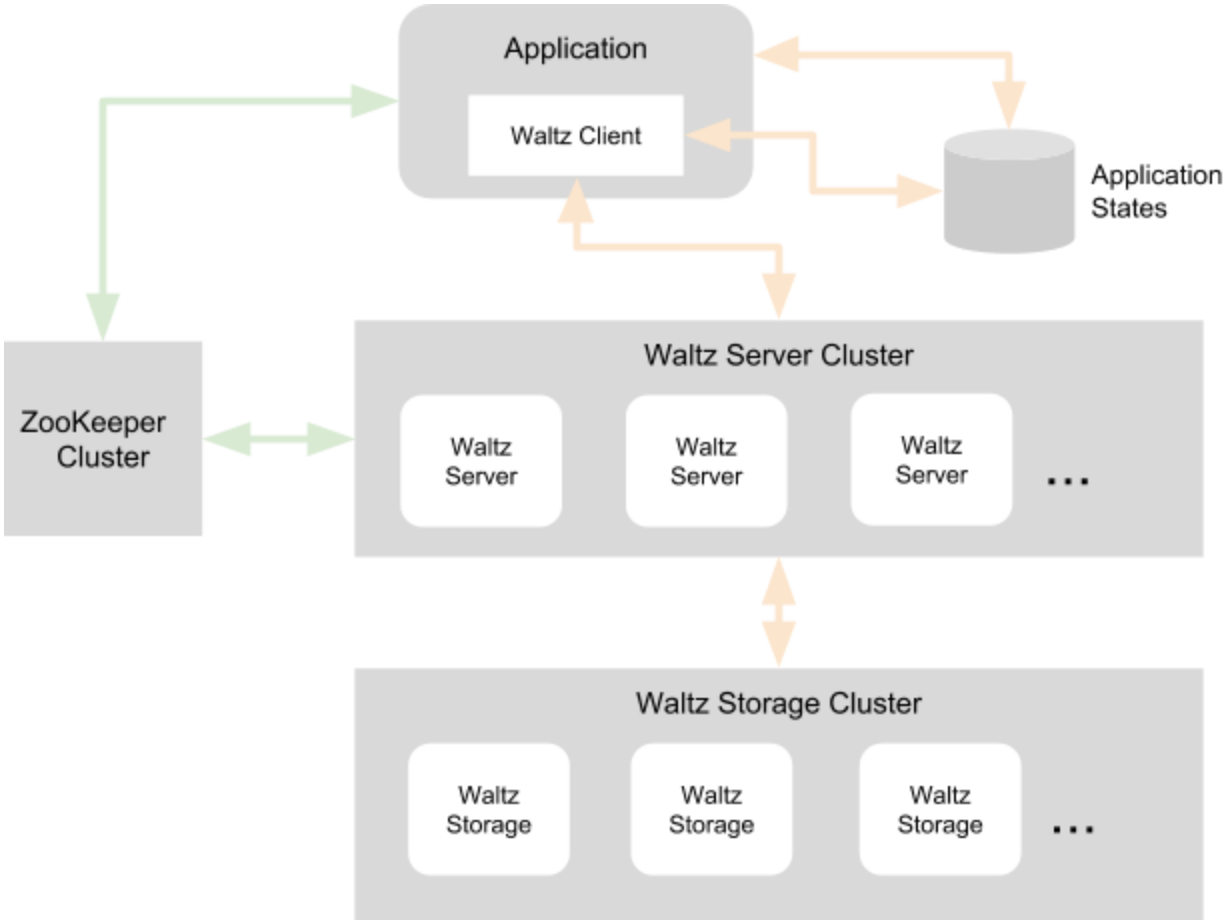
In sum, Waltz provides globally consistent transaction information to applications in highly scalable/reliable ways. Waltz can be used as the source of truth in a large distributed system.

Terminology and Components

- **Waltz Client**: A client code that runs in an application. It discovers Waltz servers to communicate through Zookeeper. Multiple clients can write to the same log concurrently.

- **Waltz Server**: A server that works as a proxy to Waltz storages and also is responsible for concurrency control.
- **Waltz Storage**: A storage server that provides persistency of data. It stores transaction data in its local disk.
- **Transaction ID**: A persistent unique id sequentially assigned to a committed transaction in a partition.
- **Transaction Header**: Transaction header is a 32-bit integer value. Its semantics is application defined. The transaction headers are streamed back to clients along with transaction IDs so that clients can do certain optimizations before fetching the transaction bodies through RPC. For example, it can be used to signify the type of transaction, and an application can fetch and process only transactions it is interested.
- **Transaction Data**: Transaction data are opaque to Waltz. They are just byte arrays for Waltz. An application must provide a serialization method to Waltz Client.
- **High-water Mark**: A high-water mark is the highest transaction ID seen. For a Waltz server it is the ID of the most recently committed transaction. For a client, it is the ID of the most recently consumed transaction.
- **Lock ID**: A partition scoped id for optimistic locking. It consists of a name and long id. Waltz does not know what a lock ID represents. The scope of a lock ID is limited to the partition. Therefore, a partitioning scheme must be decided taking a lock ID scope into consideration.
- **Waltz Client Callbacks**: An application code that Waltz client invoke to communicate with the application.
- **Transaction Context**: A client application code that packages the logic to generate a transaction. The code may be executed multiple times until the transaction succeeds, or the transaction context decides to give up.
- **Zookeeper**: We use Zookeeper to monitor participating servers. We also store some cluster wide configuration parameters (the number of partitions, the number of replicas) and metadata of storage state. The storage metadata is updated/access only when faults occur, thus the load on Zookeeper is small.

The system consists of four kinds of processes, application processes, Waltz Server processes, Waltz Storage processes, and Zookeeper server processes.



An application process serves application specific services. It generates transaction data and sends them to Waltz Server using Waltz Client running inside the application. The application receives all committed transactions in the commit order from Waltz Server and updates application's database.

Waltz Server works as a proxy to Waltz Storage. Servers receive read/write requests from clients and forward them to storages. It also works as a lock manager and a cache of transaction data.

Waltz Storage manages the log storage files. A log is segmented into multiple files. For each segment there is an index file which gives mapping from transaction IDs to transaction records. Lastly, Waltz uses Zookeeper to control the whole Waltz system. For example, Zookeeper is used to store metadata regarding storage states. Zookeeper is also used to keep track of Waltz Server instances. Waltz Client instances are informed where Waltz Server instances are.

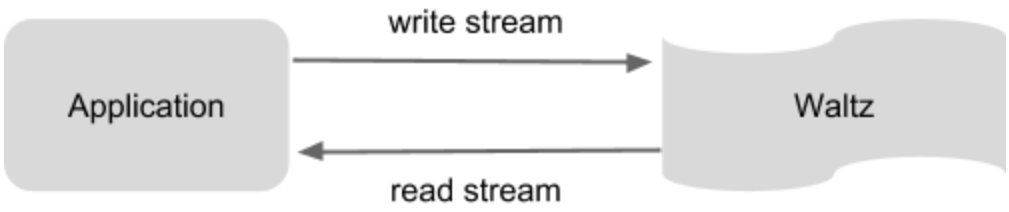
Application Programming Model

Basic Idea

An application should use Waltz as a write-ahead log. It should write to Waltz successfully before it considers a transaction committed. In this sense, Waltz has the transaction authority, i.e., Waltz log is the source of truth.

Read/Write operations are done over the network. An operation may fail at any point in communication. Also the application process or waltz process may fail due to a code bug, a machine failure, etc. It is imperative to know which transactions are persisted in Waltz and which transactions are read by the application already after restart.

To address the above concerns, We designed Waltz based on a stream oriented communication instead of a RPC based communication.



Since transactions are stored as a log, a series of records, and are assigned unique transaction ids. The transaction ID is a monotonically increasing and dense (no gap) ID. Waltz uses the transaction ID as a *high-water mark* in streaming. Waltz asks an application for its current high-water mark, the highest ID of transactions that the application consumed successfully. Based on this client high-water mark, Waltz starts streaming all transactions to the application after the high-water mark. This makes it easy for Waltz client code to discover which transaction has succeeded or failed to write. Waltz client automatically re-executes failed transactions by invoking an application provided code that constructs a transaction data.

An application may consists of multiple server processes which share the same application database. In this case, each application server receives not only transactions originated from that server but all transactions available for consumption. This is necessary for application servers to do seamless failover. So, it is important to ensure that there is no duplicate processing (or processing is idempotent), the instances collectively process all transactions, and each instance applies a non-deterministic subset of transactions to the application's database. In general it should be assume that a transaction may be processed by an application instance other than the instance which created the transaction. We provide a code that coordinate processing for database applications (AbstractClientCallbacksForJDBC).

Sending all transaction data to all application server instances is wasteful. To address this issue we employ lazy loading of transaction data. The stream does not actually contains transaction data. Transaction data is loaded on demand only when an application requires it for processing.

Finally, Waltz addresses consistency issues caused by concurrent updates. The transaction system should take care of update conflicts. They can happen when concurrent transactions overwrites each other, or when a transaction is performed based on a stale data. In traditional database systems, this is taken care by locking. In Waltz a similar machinery is provided. Waltz implements an optimistic locking. When Waltz finds a transaction conflicting with an already committed transaction, it rejects the conflicting transactions.

An Example

[TBS]

Client API

WaltzClient

A client application creates an instance of `WaltzClient` by giving an instance `WaltzClientCallbacks` and an instance `WaltzClientConfig`. As soon as an instance of `WaltzClient` is created, it attempts to connect Zookeeper cluster (the Zookeeper connect string is specified in `WaltzClientConfig`). Waltz uses `WaltzClientCallbacks` to talk to an application.

An application call `execute` method to execute a transaction.

```
public void execute(TransactionContext context);
```

`TransactionContext` encapsulates a code that build a transaction. An application must define a subclass of `TransactionContext`.

TransactionContext

```
/**
 * The abstract class of the transaction context.
 */
public abstract class TransactionContext {

    public final long creationTime;

    public TransactionContext() {
        this.creationTime = System.currentTimeMillis();
    }

    /**
     * Returns the partition id for this transaction.
     *
```

```
     * @param numPartitions the number of partitions
     * @return partitionId
     */
    public abstract int partitionId(int numPartitions);

    /**
     * <p>
     * Executes the transaction. An application must implement this method.
     * </p><p>
     * The application sets the header and the data of the transaction using the builder,
     * and optionally sets locks.
     * When this returns true, the Waltz client build the transaction from the builder and
     * send an append request to a Waltz server.
     * If the client failed to send the request, it will call this method again to execute
     * the transaction again.
     * </p><P>
     * If the application finds that the transaction must be ignored,
     * this call must return false.
     * </P><P>
     * If an exception is thrown by this method, the client will call
     * {@link TransactionContext#onException(Throwable)}.
     * </P>
     * @param builder TransactionBuilder
     * @return true if the transaction should be submitted, false if the transaction
     * should be ignored.
     */
    public abstract boolean execute(TransactionBuilder builder);

    /**
     * A method that is called on completion of this transaction context that did not
     * fail due to expiration or exception.
     * After this call, there will be no retry attempted by the Waltz client.
     * The {@code result} parameter is {@code true} if the transaction is successfully
     * appended to Waltz log,
     * otherwise {@code false}, i.e., the transaction is ignored.
     *
     * @param result {@code true} if the transaction is successfully appended to Waltz log,
     * otherwise {@code false}
     */
    public void onCompletion(boolean result) {
    }

    /**
     * A method that is called on expiration of this transaction context.
     * After this call, there will be no retry attempted by the Waltz client.
     */
    public void onExpiration() {
    }
```

```
    /**
     * A method that is called on exception.
     * After this call, there will be no retry attempted by the Waltz client.
     */
    public void onException(Throwable ex) {
    }

}
```

Waltz Client Callbacks

An application must implement `WaltzClientCallbacks` which has three methods shown below. They are invoked by `WaltzClient` to retrieve the client high-water mark, and to supply new committed transactions to the application to update application's states, and to allow the application to handle exceptions.

```
/**
 * The interface for Waltz client callback methods.
 */
public interface WaltzClientCallbacks {

    /**
     * Returns the current high-water mark of the client application.
     * {@link WaltzClient} calls this method to know which offset to start transaction feeds.
     * @param partitionId
     * @return
     */
    long getClientHighWaterMark(int partitionId);

    /**
     * Applies a committed transaction to the client application.
     * {@link WaltzClient} calls this method to pass a transaction information that is
     * committed to the write ahead log.
     *
     * @param transaction
     * @throws Exception
     */
    void applyTransaction(Transaction transaction) throws Exception;

    /**
     * A method called by the Waltz client when {@link #applyTransaction} throw an exception.
     * @param partitionId
     * @param transactionId
     * @param exception
     */
    void uncaughtException(int partitionId, long transactionId, Throwable exception);

}
```

Other important classes/interfaces

- `TransactionBuilder`

- `Transaction`
- `WaltzClientConfig`
- `PartitionLocalLock`
- `Serializer`

Client-Server Communication

Client-Server communication uses persistent TCP connections. A client creates two connections per server. One is for streaming, and the other for RPC. The networking module is built on top of Netty.

Request ID (ReqId)

Why we need 2 conn per server?

The request ID is a unique ID attached to a request message and corresponding response messages.

Field	Data Type	Description
Client ID	int	The unique ID of the client. The uniqueness is guaranteed by ZK.
Generation	int	The generation number of the partition.
Partition ID	int	The partition ID
Sequence number	int	The sequence number.

Mounting a partition

A client establishes a communication to servers in the following manner for each partition.

1. The client finds a server to which the partition is assigned.
2. The client sends a mount request to the server.
3. If the server has the partition,
    a. The server starts the transaction feed. The client accepts the feed data.
    b. If the feed reached the client high water mark, it sends the mount response with `partitionReady = true`. Q2: what is the high water mark for feed? Sever side high water mark is based on lock, right?
    c. The client receives the mount response and completes the mounting process.
4. Otherwise,
    a. The server sends the mount response with `partitionReady = false`.
    b. The client receives the mount response and find that the partition is not ready.
    c. Repeat from 1.

Mount Request

Field	Data Type	Description

Request ID	ReqId	Client generated unique request ID
Client High-water Mark	long	The highest ID of transactions applied to the client application’s database
sequence number	int	A sequential ID that identifies the network client sending this request. This is used to detect stale network clients.

Mount Response

Field	Data Type	Description
Request ID	ReqId	Client generated unique request ID stored in the corresponding mount request
Partition Ready	boolean	True if the partition is ready, otherwise false

Writing and Reading Transactions

Append Request

The append request submits a transaction to Waltz.

Field		Data Type	Description
Request ID		ReqId	Client generated unique request ID
Client High-water Mark		long	The highest ID of transactions applied to the client application’s database
Lock Request	Length	int	The length of the hash value array
	Hash Values	int[]	Lock hash values
Transaction Header		int	Application defined 32-bit integer metadata
Transaction Data	Length	int	The length of transaction data byte array.
	Bytes	byte[]	A byte array
Checksum		int	CRC-32 of the transaction data

Feed Request

The feed request initiates the transaction feed from Waltz server.

Field	Data Type	Description
Request ID	ReqId	Client generated unique request ID
Client High-water Mark	long	The highest ID of transactions applied to the client application’s

		database

Feed Data

The feed data is stream to a client in response to the feed request.

Field	Data Type	Description
Request ID	ReqId	Client generated unique request ID stored in the corresponding Feed Request.
Transaction ID	long	The ID of the transaction
Transaction Header	int	Application defined 32-bit integer metadata

Transaction Data Request

A transaction data request is sent through as a RPC request.

Field	Data Type	Description
Request ID	ReqId	Client generated unique request ID
Transaction ID	long	The ID of the transaction to fetch

Transaction Data Response

A transaction data response is sent through as a RPC response.

Field		Data Type	Description
Request ID		ReqId	Client generated unique request ID stored in the corresponding Feed Request.
Transaction ID		long	The ID of the transaction fetched
Success Flag		boolean	True if the fetch is successful, otherwise false.
Transaction Data	Length	int	The length of transaction data. (exists only when the success flag is true)
	Bytes	byte[]	A byte array (exists only when the success flag is true)
Checksum		int	CRC-32 of the transaction data (exists only when the success flag is true)
Error Message		String	Error message (exists only when the success flag is false)

Other Messages

Flush Request

A client sends a flush request to wait for all pending transaction to complete regardless of successfully or not. A flush response will be send back to the client when all append requests sent before this request were completed.

Field	Data Type	Description
Request ID	ReqId	Client generated unique request ID

Flush Response

Field	Data Type	Description
Request ID	ReqId	Client generated unique request ID stored in the corresponding Flush Request.
Transaction ID	long	The high-water mark after pending transactions are processed.

Lock Failure

A lock failure message is sent back to a client when a lock request was failed.

Field	Data Type	Description
Request ID	ReqId	Client generated unique request ID stored in the corresponding Append Request
Transaction ID	long	The ID of the transaction that made the lock request fail.

Generation Number

The generation number is used to ensure that Waltz server instances and clients work consistently in the dynamically changing environment.

Waltz Server cluster consists one or more Waltz server instances. A partition is assigned to a single server instance at any moment. The cluster manager is responsible for the assignments. When a new instance comes up, or an old instance goes down, the cluster manager detects it and reassign partitions to make certain that there is one and only one instance for each partition. Everytime this happens, the cluster manager bumps up the partition's generation number. Waltz servers ignores any append request when the generation number does not match.

Detection of Failed Append Requests

Each client has a registry of append requests called the transaction monitor. The transaction monitor determines a state (success/failure) of each append request using the transaction feed.

For each transaction in the feed,

1. The client checks if its transaction monitor contains the ReqId in the feed data.
2. If the transaction monitor has the ReqId, the transaction was issued by this client and successful, so,
    a. The transaction monitor marks the transaction as success.
    b. The transaction monitor marks any pending transaction older than this transaction as failure.
    c. The transaction monitor clears the entries of completed (either success or failure) transaction in the registry
3. Otherwise, ignore

A client automatically reconnects to Waltz servers after a network failure or a Waltz server failure. When a reconnect happens, it is possible that the server may have lost some requests. It may take a while for the client to recognize it especially for a service generating append requests in a slow pace. To ensure a quicker detection, the following reconnect procedure has designed.

1. The client blocks append requests
2. The client establishes a connection to the server
3. The client send a mount request
4. The client starts receiving transaction feed and apply the regular failure detection
5. The client makes all pending requests fail when the mount response is received
6. The client unblocks append requests

The completion of a mount request ensures that the server does not have any pending request from this client. The client can safely get rid of pending requests as failures.

Server-Storage Communications

Quorum Writes

Waltz is a replicated transaction log. It does not use a master-slave replication method, but it uses a quorum write method. A quorum in Walt is equivalent to a majority vote. We will use "quorum" to mean "majority" in this document.

A quorum system has a number of benefits over master-slave replications. In master-slave replications, the master is the authoritative source of data, and slaves are always catching up with some latency. When a master dies due to a fault, we may want to promote one of slaves to

a new master to continue a service. However, there is no guarantee that the slave has finished replication of all data before the death of the old master or knows the final commit decision that the master made. On the other hand, for a quorum system like Waltz a commit is consensus among participating storage servers, i.e., there is no central authority that may fail to propagate the commit information to other servers. A writer does not *decide* whether or not the write is committed, but it merely *observe* the commit is established by quorum. This distinction is important for recovery. A recovery process simply observes whether or not a particular write is committed by investigating the state of storages.

Sessions

Waltz server is responsible for replicating transaction data to Waltz storages. Each Waltz server has a set of partitions assigned by the cluster manager. A partition is always assigned to a single server. No two servers write to the same partition at the same time. This is guaranteed by monotonically increasing session IDs. A server always establishes a session to access storage servers. When a server starts a new session, it acquires a new session ID (we use Zookeeper for this), and attaches the session ID to all messages to storage nodes. Storage nodes compare the session ID of the message with their current session ID. If the session ID of the message is greater than the ones they have, they take the session ID of the message as the most recent session ID and reject any message with lower session ids from then on.

At the beginning of a session, Waltz server gathers storage states and figures out the last commit. Then it sends a truncate message to storages to remove any uncommitted transaction in storage nodes. If there is an unreachable storage server, the cleanup will be done later when the storage server becomes available again.

1. Get storage state information from Zookeeper
2. Gather storage state information from storage servers (last session ID, max transaction ID, the last known clean transaction ID)
3. See which storage node was active in the last session
4. If a storage server was not in the last session, simply truncate the log to the last known clean transaction ID.
5. Compute the highest commit transaction ID
6. Update storage information in zookeeper
7. Send the commit transaction ID to all available nodes (this becomes the new known clean transaction ID)
8. Clean up storage servers with dirty transactions.

Waltz prevents transaction logs from forking under any circumstance. Write requests are serialized by the server and streamed to storage servers. Ordering is guaranteed to TCP connection semantics and a single threaded processing per partition in a storage node. Furthermore, Waltz server creates a new session and runs the recovery whenever a storage node becomes unavailable even when there are enough number of healthy storage nodes. By

design there is no way for a storage server to rejoin the session once it has left the session due to a fault.

Note that we don't assume a thing like a clean session close. A recovery is always run before a new session starts writing.

On Disk Data Structures

Waltz Storage provides persistency of data. It stores transaction data in its local disk.

Directory Structure

Waltz Storage stores transaction data in the local file system. The root directory is called the storage directory which is configured using a configuration file. The storage directory contains the control file (*waltz-storage.ctl*) which contains a version information, creation timestamp and partition information. Under the storage directory, there are partition directories. Each partition directory contains data files and index files. For each partition, transaction data are split into segments chronologically. A new segment is created when the current segment grow beyond the configured size. Each segment consists of a data file and an index file.

```
<storage directory>/          # the root directory of the storage (configurable)
    waltz-storage.ctl                # the control file
    0/                               # the directory for partition 0
        00000000000000000000.seg     # the segment data file. The file name is
                                   # <first transaction id in the segment>.seq
        00000000000000000000.idx     # the segment's index file
        ....
    1/
        ....
```

Control File

Control File Header

The control file begins with the header which contains the following information.

Field	Data Type
format version number	int
creation time	long
key	UUID
the number of partitions	int

reserved for future use	96 bytes

The header size is 128 byte.
The key is UUID which is generated when the cluster is configured by CreateCluster utility. The key identifies the cluster to which the cluster it belongs. If an open request comes from a Waltz Server whose key does not match the the key in the control file, Waltz Storage rejects the request.

Control File Body

After the header follows the actual body of control data. It is a list of *Partition Info*. The number of *Partition Info*s is the number of partitions recorded in the header.

Field		Data Type	
partition id		int	
partition info struct 1	session id	long	
	low-water mark	long	Global low-water mark, only set on recovery
	local low-water mark	long	Useful for local recovery, like index file recovery
	checksum	int	
partition info struct 2	session id	long	
	low-water mark	long	
	local low-water mark	long	
	checksum	int	

A partition info struct records the session ID, the low-water mark, the local low-water mark, and the checksum of the struct itself. The low-water mark is the high-water mark of the partition in the cluster when the session is successfully started. The local low-water mark is the highest valid transaction ID of the partition in the storage when the session is successfully started. The local low-water mark can be smaller than the low-water mark when the storage is falling behind.

Two partition info structs are updated alternately when a new storage session started, and the update is immediately flushed to the disk. The checksum is checked when a partition of opened. Since the atomicity of I/O is not guaranteed, it is possible that an update is not completely written to the file when a fault occurs during I/O. If one of the structs has a checksum error, we

ignore it and use the other struct, which means we rollback the partition. We assume at least one of them is always valid. If neither of structs is valid, we fail to open the partition.

Segment Data File

Data File Header

Field	Data Type
format version number	int
creation time	long
cluster key	UUID
partition id	int
first transaction ID	long
reserved for future use	88 bytes

The header size of 128 bytes. The cluster key is a UUID assigned to a cluster.

The first transaction ID is the ID of the first transaction in the segment.
The data file body is a list of transaction records. Each transaction record contains the following information.

Transaction Record

Field		Data Type
transaction ID		long
request id		ReqId
transaction header		int
transaction data	length	int
	checksum	int
	data	byte[]
	checksum	int
checksum		int

When new records are written, Waltz Storage flushes the file channel to guarantee the record persistence before responding to Waltz Server. The index file is also updated, but flush is delayed to reduce physical I/Os until checkpoint. The checkpoint interval is 1000 transactions (hardcoded). When a checkpoint is reached, Waltz Storage flushes the index file before adding a new record. This means, if a fault occurs between checkpoints, we are not sure if the index is valid. So, the index file recovery is necessary every time Waltz Storage starts up. Waltz Storage scans the records from the last checkpoint and rebuild index for record after the last checkpoint.

Segment Index File

Index File Header

Exactly same as the data file header.

Index File Body

Index File Body is an array of transaction record offsets.

Field	Data Type
transaction record offset	long

Each element corresponds to a transaction in the segment. The array index is <*transaction id*> - <*first transaction id*>. Each element is byte offsets of the transaction record in the data file.

Checkpoint Interval

In the recovery process described above, the last known clean transaction ID is updated more often than a stable environment since it is updated during the recovery process. A drawback is that the number of transactions after the last known clean transaction ID can become large when no fault occurs for a long period of time. This is bad when a recovery requires a truncation to the last known clean transaction ID. So, Waltz provides a configuration parameter "storage.checkpointInterval" which is an interval in transactions for forced initiation of a new session.

Handling Snapshot or Backup

Waltz does not provide a snapshot or backup making functionality. It is not a high priority at this moment since Waltz storage is fault tolerant. If necessary, use of a journaling file system like ZFS is a possible solution to this for now.

Let’s assume a snapshot is available somehow. We may restore stale storage files from a snapshot when storage files on a storage node is damaged by a disk failure or mistake. The issue is that the state information in Zookeeper and the state information storage becomes

inconsistent. Waltz already handle this case. The storage is simply truncated to the last known clean transaction ID (recorded in the storage) to remove any possibly dirty transaction, then the catch-up process will be started and bring the storage up-to-date.

Concurrency Control - Optimistic Locking

Waltz implements a concurrency control using an optimistic locking. Instead of acquiring an exclusive lock on a resource explicitly, an optimistic locking verifies that there is no other transaction has modified the same resource during the transaction execution. Waltz uses the same concept, but it is made much lighter weight by taking advantage of log's high-water mark.

In Waltz a granularity of lock is not a record but an application defined lock ID. A lock ID consists of a name and a long integer ID. Waltz does not know what a lock ID represents.

Usually an optimistic locking is done by versioning records. When a record is updated, the version number of the record read by the updater is compared to the record in the database. If they match, it means there was no other transaction modified the same record, thus the update succeeds. If not, the update fails since the record has already been modified by other transaction.

It can be very expensive to maintain the version numbers of all possible lock IDs. Instead, Waltz uses a probabilistic approach, similar to *Bloom Filter*, to keep the memory consumption predictable while achieving a low false negative rate. It uses hash values of lock IDs and the partition's high-water mark.

The data structure is a fixed size array. Waltz server maintains the array *A* of high-water marks with size *L*. We have *N* independent hash functions $hash_i$ ($i = 0 .. N-1$). The estimated high-water mark of lock ID is $min \{ h \mid h = A[hash_i(id)], i = 0 .. N-1 \}$. If the estimate is smaller than the transaction's high-water mark, the transaction is safe. There will be no false-negative. The false positive rate depends on the load. If we allocate a large array, we can keep the false negative rate reasonably low.

The above array is called a *lock table*. Lock tables are maintained by Waltz Server. There is one lock table for every partition. It means that the scope of a lock ID is limited to the partition. Therefore, a partitioning scheme must be decided taking a lock ID scope into consideration.

A client does not need to remember high-water mark for each lock ID. It only has to maintain the current client high-water mark, which is required for streaming anyways.

The optimistic lock is checked as follows.

1. The client remembers the client high-water mark at the beginning of transaction (just before invocation of the `execute` method of `TransactionContext`)

2. The client computes a transaction (the `execute` method of `TransactionContext`)
3. The client sends the transaction data and the high-water mark to Waltz server
4. Waltz server estimates the high-water mark for the lock IDs sent along with the transaction data using the lock table.
5. If the client's high-water mark is higher than the estimated high-water mark, the client was up to date when the transaction is computed, thus success.
6. Otherwise, failure.

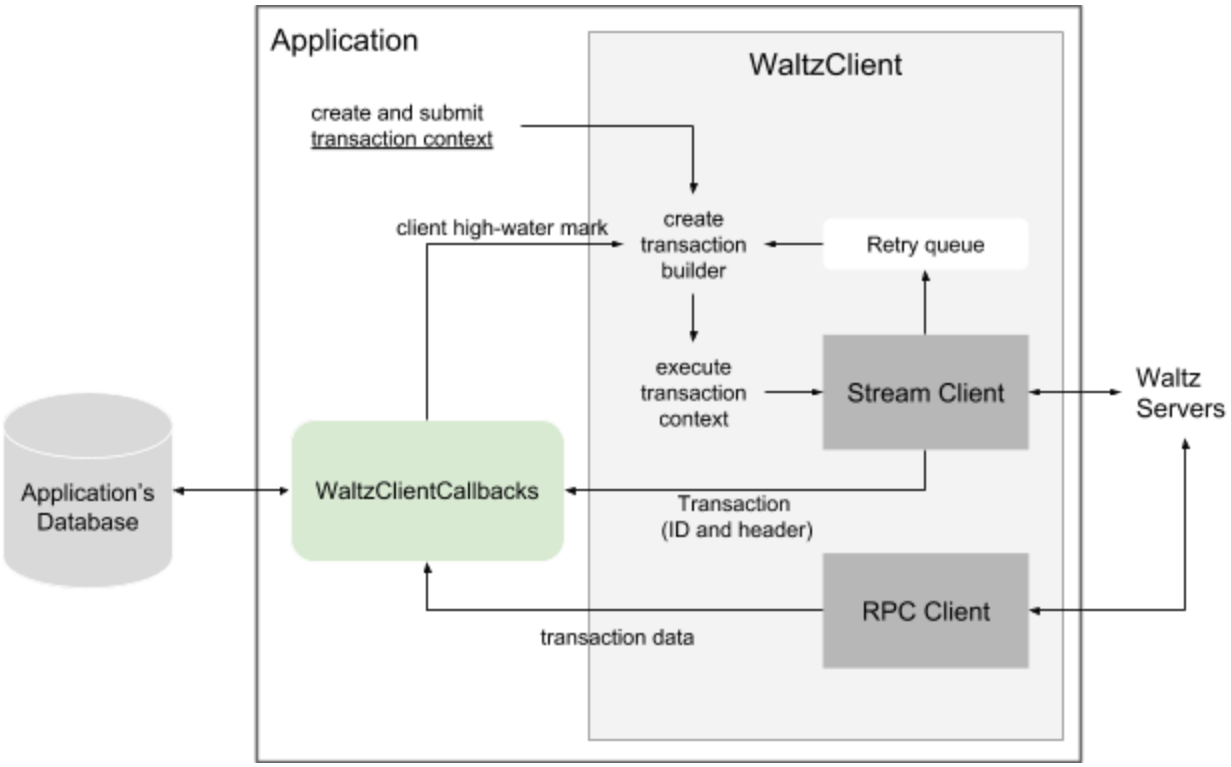
Back Pressure

The system will apply back pressure to control the rate of request processing at different parts of the system.

- Waltz Client limits the number of outstanding append requests. When the number of requests reaches the limit subsequent requests are blocked.
- This is important to avoid exhausting the heap memory when traffic is very high. Inbound messages to Waltz server are throttled by the number of messages in the queue. We turn on/off auto-read from the network buffer according to the number of messages. When it is turned off, Netty's event loop stop reading message from the network buffer, thus stop decoding message. The clients will eventually stop writing to the network channel since the server's network buffer will get filled.

Waltz Client

Client-Application Interactions



An application must define a subclass of `WaltzClientCallback` and supply an instance of it to the constructor of `WaltzClient`. Through the callbacks, Waltz client gets the latest client high-water mark for a partition and also applies a committed transaction to the application.

An application must define a subclass of TransactionContext to send a transaction data to Waltz. A transaction context encapsulates a application logic and data that are necessary to generate a transaction data.

1. The application submits an instance of TransactionContext
2. The Waltz client calls the `getClientHighWaterMark` method of `WaltzClientCallbacks` to get the latest client high-water mark
3. The Waltz client constructs `TransactionBuilder` with the client high-water mark.
4. The Waltz client invokes the `execute` method of the context with the instance of `TransactionBuilder`.
5. The application code (the `execute` method of `TransactionContext`) builds a transaction data.

6. The Waltz client sends a append request to a Waltz server to write the transaction data.
7. The control returns to the application.
8. In the background, the Waltz client is monitoring the status of the append request.
    a. If the append request was not successful, the Waltz client puts the transaction context into the retry queue.
    b. The Waltz client executes the failed transaction context asynchronously.

Transaction data successfully appended to the log will be streamed back to the application eventually.

1. The Waltz client receives feeds of successful transactions. Feeds includes transaction IDs and transaction headers
2. The Walz client invokes the `applyTransaction` callback
3. The application code (the `applyTransaction` callback) process the transaction and updates the client high-water mark.

Client-Server Interactions

There are two kinds of interaction with servers, streaming and RPC. Streaming guarantees ordering of requests and responses (including transaction feeds). RPC does not guarantee or ordering. RPC is used for retrieving transaction data for a particular transaction ID. Everything else uses streaming.

`WaltzClient` has two internal clients, `StreamClient` (the implementation class is `InternalStreamClient`) and `RpcClient` (the implementation class is `InternalRpcClient`). They manages streaming connections and RPC connections, respectively. They have similar structures. Actually their implementation classes extend the common superclass `InternalBaseClient`.

InternalBaseClient

`InternalBaseClient` manages connection to all servers the client is communicating. A single connection to a server is represented by an instance of `WaltzNetworkClient` in `InternalBaseClient`. An instance of `InternalBaseClient` has a network client map keyed by the endpoint (the server address).

`InternalBaseClient` has another map, a partition object map keyed by the partition ID. A partition object works like a proxy to the partition managed by a remove server. A partition object has an associated network client. This association is not permanent. It is updated accordingly when the partition assignment to servers are changed. `InternalBaseClient` calls the `mountPartition` method of the network client to tell it that it is now responsible responsible for that partition. The network client, in turn, calls the `mounting` method of the partition object to

tell the partition the network client is now responsible. If the network channel is ready, the network client calls back `InternalBaseClient`'s `onMountingPartition` method to finish mounting. If not, `onMountingPartition` is called later when the channel becomes ready.

InternalStreamClient

`InternalStreamClient` extends `IntenalBaseClient`. The `onMountingPartition` method of `InternalStreamClient` sends a mount request to the server to set up a streaming context. Another important method is `onTransactionReceived`, which is invoked when the network client received a transaction from a server. This method invokes the `applyTransaciton` method of `WaltzClientCallbacks`.

InternalRpcClient

`InternalRpcClient` extends `IntenalBaseClient`. The `onMountingPartition` method of `InternalRpcClient` does not send a mount request to the server because `RpcClient` does not require streaming context at all. Instead, it sends all pending TransactionDataRequests to the new server. `onTransactionReceived` is not supposed to be used. So, it just throws an exception.

Partition

Both `InternalStreamClient` and `InternalRpcClient` have a map of partition objects. Each partition object has the following important data structures

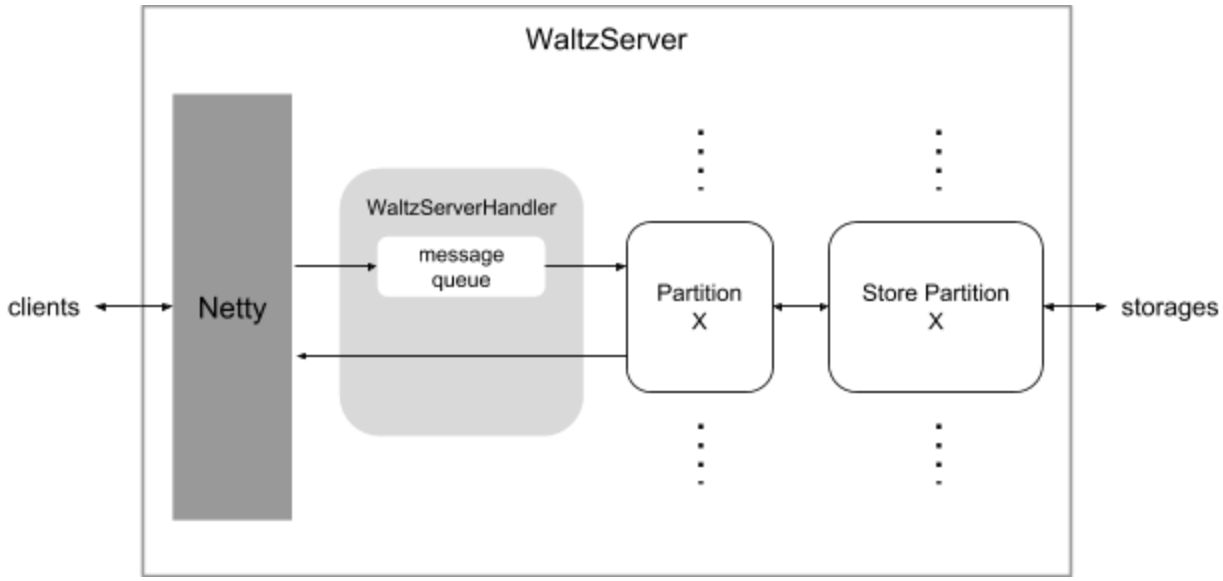
- `TransactionMonitor`
- A map of `Future`s keyed by transaction ID
- `WaltzNetworkClient`

`TransactionMonitor` is used by `InternalStreamClient` to keep track of states of append requests. The map of future is used by RpcClient to keep track of pending transaction data requests. `WaltzNetworkClient` is a network client responsible for communication with the server for this partition.

Fault Recovery

A network client is closed whenever an unexpected error occurs. We basically don't try to recover using the same connection. A new network client is created immediately to recover a connection and perform the partition mounting process again. Each network client has a sequence number incremented on every creation. The stream client send the mount request includes the sequence number of the network client so that server can discard all messages from old connections after it receives the mount request. This process is exactly same as handling of partition assignment change.

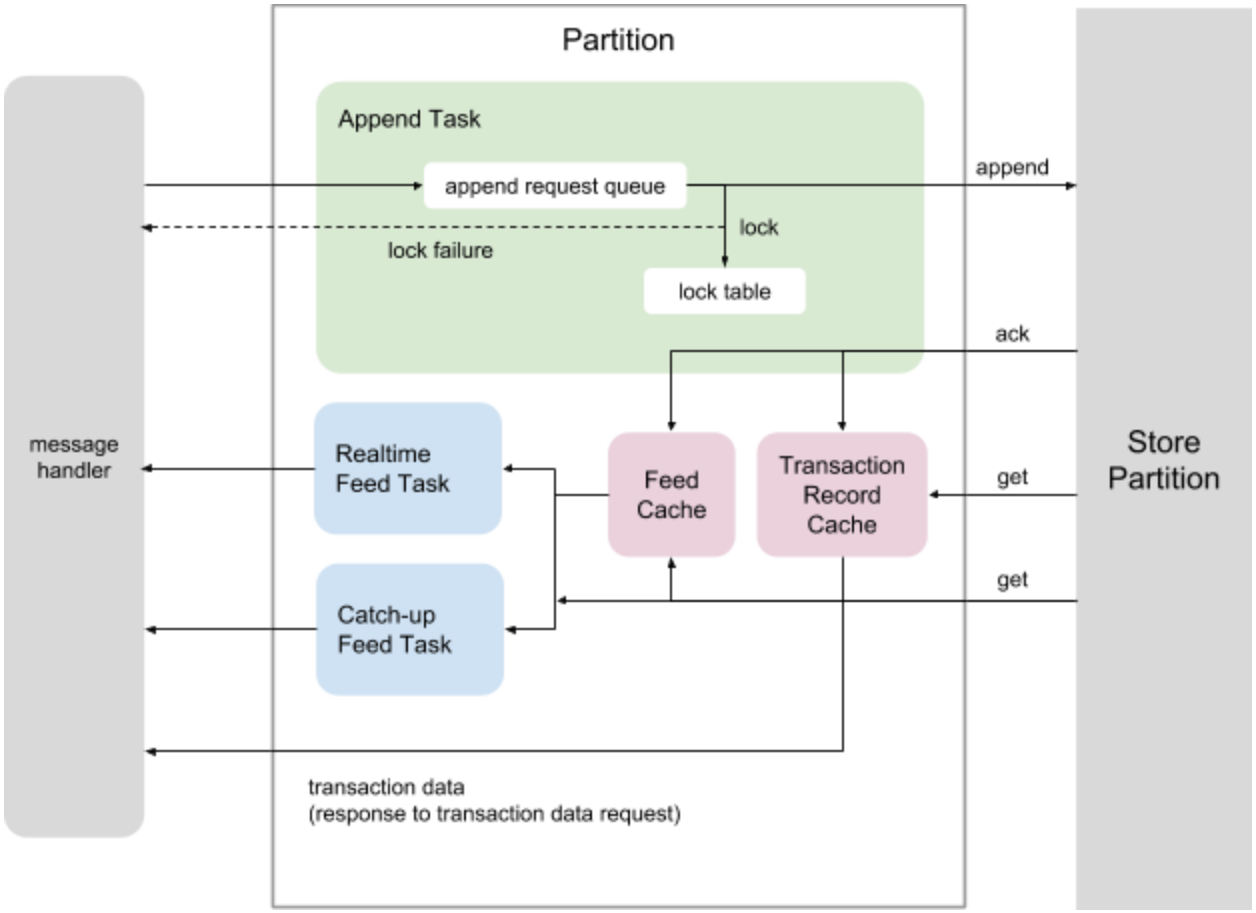
Waltz Server



Waltz Server receives messages from the clients. We implemented the networking layer on Netty. Netty calls `WaltzServerHandler` for each message. Messages are enqueued into the message queue and dispatched to partition by the message handler thread running in `WaltzServerHandler`.

A partition object represents a Waltz log partition. The persistent log data are replicated and stored in multiple storage servers. Actual interaction with storages are done by a store partition object. A partition object and a store partition object are created when the partition is assigned to the server.

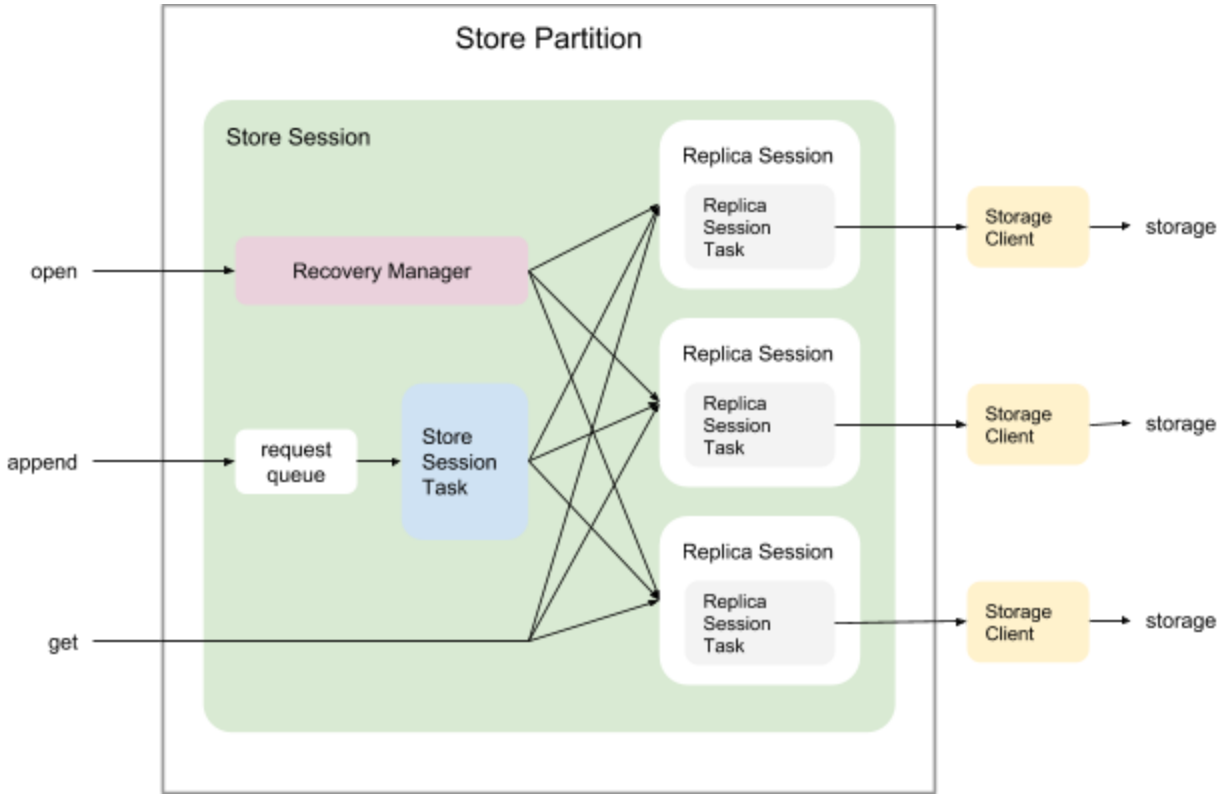
Partition Object



Each partition object has three tasks (threads), Append task, Realtime Feed task, and Catch-up Feed Task.

An append request is immediately place in the append request queue in the append task object. The thread of the append task polls a request from the queue and tries to acquire locks if the append request contains any lock request. If locking tails, the task sends a lock failure message to the client. If there is no lock failure, the transaction information is passed to the corresponding store partition. A store partition works as a proxy to storage servers. When the success of append operation is acknowledged, the transaction information is stashed into the feed cache and the transaction record cache.

Store Partition Object



A Store Partition object manages read/write to storage servers. A store partition represents a partition which is replicated to multiple storage servers. Read/write operations are performed in a context of a session called a store session. Actual read/write operations to a partition on each storage server is also performed in a session called a replica session.

When a store session is created, replica sessions are created for all known storage servers. Then, a recovery manager is created and starts a recovery to resolve any unresolved write operations and truncate any dirty data on storage servers.

An append request are first placed in the request queue. The store session task polls requests from the queue, batch them up, and sends to all available storage servers through replica sessions. When the number of successful writes reach the quorum, the notification is propagated to the requester through a callback. If a storage is falling behind, the append request to that storage is discarded, and the replica session task starts catch-up process which transfers transaction data from other storages to this storage.

Partition Metadata

Waltz Server stores the metadata of partitions in Zookeeper for recovery. The ZNode path is `<cluster root>/store/partition/<partition id>`. They includes the generation numbers, the store session ID, and the states of replicas (the replicated partitions on storage servers). They are written when a new store session is created.

Partition Metadata

Field		Data Type	Description
Generation		int	The generation number
Session ID		long	The current store session ID
Replica States	Length	int	The size of Replica state array
	List	ReplicaState[]	The list of Replica State

ReplicaId

Field	Data Type	Description
Partition ID	int	The partition ID
Replication ID	byte	The replication ID

ReplicaState

Field	Data Type	Description
Replica ID	ReplicaID	The replica ID
Session ID	long	The ID of the store session that this replica is currently engaged.
Closing High-water mark	long	The high-water mark when this store session closed. It is `ReplicaState.UNSOLVED` when the session starts. This field is set by Recovery Manager when the closing high-water mark of the session is resolved.

Recovery Procedure

A recovery procedure is applied whenever Store Session Manager creates a new store session. The followings are steps of the recovery.

1. Get new store session ID by updating Partition Metadata in Zookeeper. CAS is used to avoid race conditions.
2. Create new Replica Sessions
3. Create a new Store Session with the new session ID and the new Replica Sessions.
4. Open the Store Session
    a. Open each Replica Sessions and start Replica Session Task
5. In each Replica Session Task,
    a. Wait for a connection to establish
    b. After connected,
        i. Get the Replica State from Partition Metadata check the sessionId, to see if a new session
        ii. If the Replica State and the actual storage is inconsistent,
            1. Truncate the transaction log to the low-water mark saved in the storage
        iii. Otherwise,
            1. If the closing high-water mark is already set,
                a. Truncate the log to it.
            2. Otherwise,
                a. If the max transaction ID is greater than the low-water mark, propose its max transaction ID as the closing high-water mark.
        iv. Start catch-up to any replica ahead of itself
            1. Check if the closing high-water mark is resolved (a quorum is established)
            2. If yes,
                a. Set the low-water mark
                b. If the replica has fully caught up,
                    i. Truncate any dirty transactions
            3. Otherwise,
                a. Continue catching-up
6. Update Partition Metadata
    a. For each Replica State,
        i. If the replica is clean,
            1. Set the session id to the new store session id
            2. Set the closing high-water mark to `ReplicaState.UNRESOLVED`
        ii. Else if the closing high-water mark is `ReplicaState.UNRESOLVED`,
            1. Set the closing high-water mark to the resolved closing high-water mark leaving the session id unchanged

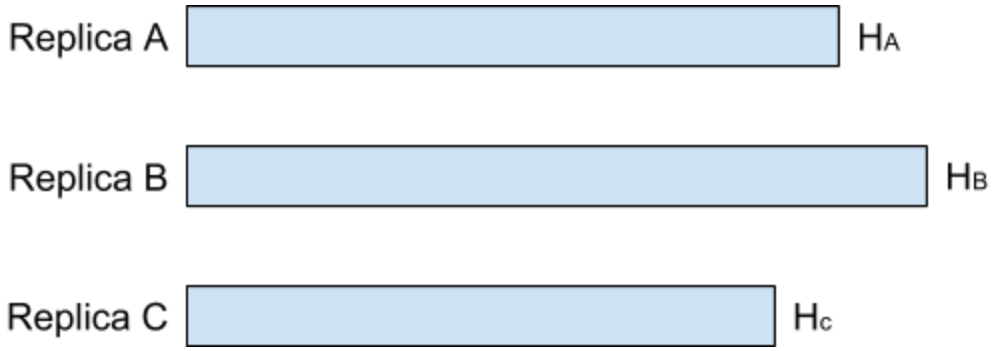
iii. Otherwise,

1. Leave the replica state unchanged.

Quorum check only happen when new When could this happen?

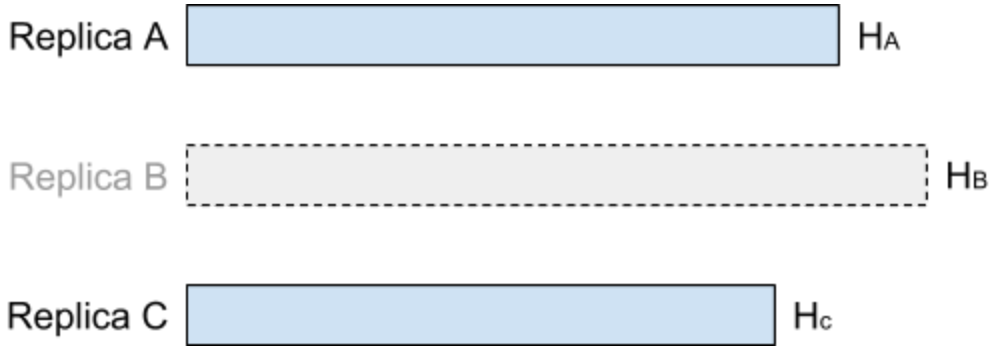
Quorum Checking in the Recovery Process

Quorum checking in the recovery process has a subtle difference from quorum checking in he write operation. We need to find out the highest high-water mark that majority of replicas agree on. In the following figure, the high-water marks of Replica A, B, and C are H_A, H_B, and H_C, respectively. The highest high-water mark that a majority of replicas agree is H_A.



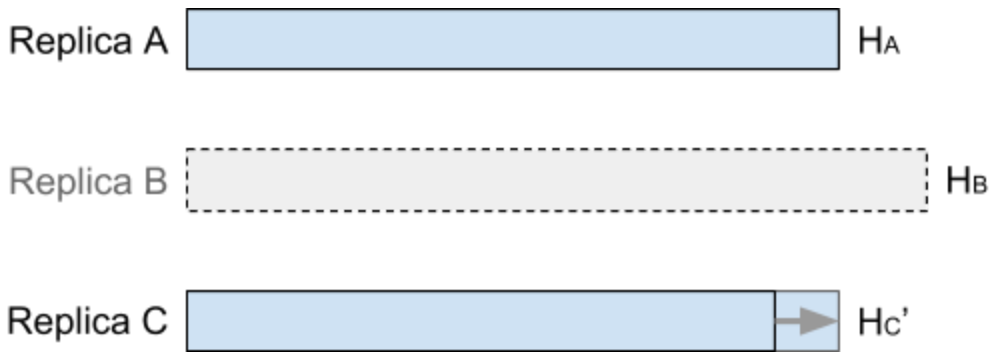
We do the selection by having replicas vote on high-water marks. A replica with the high-water mark X votes for any high-water mark H equals to or less than X. So, Replica A votes for { H_A, H_B }, Replica B votes for { H_A, H_B, H_C }, and Replica C votes for { H_C }. Clearly H_A is the highest high-water mark that a majority replicas vote. This is correct since all transactions above H_A have no chance of getting the write quorum, thus they are uncommitted. All transactions at H_A or below have the write quorum, thus it is safe to declare they are committed.

This works fine when all replicas are participating. However, we cannot expect all replicas are available all the time. There may be a inaccessible storage servers during the recovery process which may have caused the recovery process in the first place. So, the quorum checking must take that into consideration.



If Replica B is inaccessible, only Replica A and C vote. The highest high-water mark with majority vote is H_C, but this is a wrong. The high-water mark is undecidable in this situation.

The solution is to detect the undecidable situation by counting inaccessible replicas and wait until it becomes decidable. It becomes decidable when enough replicas come online. In the above case, it becomes decidable when Replica B comes back. Another situation that it becomes decidable is Replica C catches up with Replica A.



Every time the situation changes, the recovery process re-evaluate the quorum.

Waltz Storage

Waltz Storage is a storage server which provides persistency to Waltz. Its functionality is designed to be limited to relatively simple set of commands. Waltz centralizes the responsibility of transaction consistency and fault a recovery to Waltz Server. Waltz Storage has following ten request/response patterns.

1. Open Request→ { Success Response | Failure Response }
2. Last Session Info Request → { Last Session Info Response | Failure Response }
3. Max Transaction ID Request → { Max Transaction ID Response | Failure Response }
4. Truncate Request → { Success Response | Failure Response }
5. Set Low-water Mark Request → { Success Response | Failure Response }
6. Append Request → { Success Response | Failure Response }
7. Record Header Request → { Record Header Response | Failure Response }
8. Record Request → { Record Response | Failure Response }
9. Record Header List Request → { Record Header List Response | Failure Response }
10. Record List Request → { Record List Response | Failure Response }

Open Request

Field	Data Type	Description
Session ID	long	-1
Sequence number	long	-1
Partition ID	int	-1
Cluster key	UUID	A unique ID of the cluster
Number of partitions	int	The number of partitions

Open request is the first request a Waltz server makes after a connection to a storage server is opened. A Waltz server sends a cluster key which is UUID assigned to a cluster when a cluster is created by an admin tool. The storage server compares it with the one in the control file. If they don't match, it indicate there is a configuration error.

Last Session Info Request

Field	Data Type	Description
Session ID	long	The store session ID
Sequence number	long	The message sequence number
Partition ID	int	The partition ID

Last Session Info request is sent by the recovery manager when a replica connection is opened. It requests the information of the last store session of the storage partition on this storage server. It comes from Partition Info on the control file. The low-water mark is the high-water mark of the partition when the store session started.

Last Session Info Response

Field	Data Type	Description
Session ID	long	The store session ID
Sequence number	long	The message sequence number
Partition ID	int	The partition ID

Session Info	Session ID	long	The last store session ID
	Low-water Mark	long	The low-water mark of the last session

Max Transaction ID Request

Field	Data Type	Description
Session ID	long	The store session ID
Sequence number	long	The message sequence number
Partition ID	int	The partition ID

This requests the max transaction ID of the specified storage partition. The transaction may not be committed yet.

Max Transaction ID Response

Field	Data Type	Description
Session ID	long	The store session ID
Sequence number	long	The message sequence number
Partition ID	int	The partition ID
Transaction ID	long	The max transaction ID of the partition on the storage server

Truncate Request

Field	Data Type	Description
Session ID	long	The store session ID
Sequence number	long	The message sequence number
Partition ID	int	The partition ID
Transaction ID	long	The max transaction ID to retain. Any transaction after this transaction ID will be removed.

Set Low-water Mark Request

Field	Data Type	Description
Session ID	long	The store session ID
Sequence number	long	The message sequence number
Partition ID	int	The partition ID
Low-water mark	long	The low-water mark which is the max transaction ID when this store session is started.

Append Request

Field		Data Type	Description
Session ID		long	The store session ID
Sequence number		long	The message sequence number
Partition ID		int	The partition ID
Record List	Length	int	The number of transaction records
	Record List	Record[]	The list of records

RecordHeader

Field	Data Type	Description
Transaction ID	long	The transaction ID
Request ID	ReqId	Client generated unique request ID
Transaction Header	int	The transaction header

Record

Field	Data Type	Description
Transaction ID	long	The transaction ID
Request ID	ReqId	Client generated unique request ID

Transaction Header		int	The transaction header
Transaction Data	Length	int	The length of transaction data
	Data	byte[]	A byte array
Checksum		int	CRC32 of transaction data

Record Header Request

Field	Data Type	Description
Session ID	long	The store session ID
Sequence number	long	The message sequence number
Partition ID	int	The partition ID
Transaction ID	long	The transaction ID

Request Header Response

Field	Data Type	Description
Session ID	long	The store session ID
Sequence number	long	The message sequence number
Partition ID	int	The partition ID
Record header	RecordHeader	The transaction record header

Record Request

Field	Data Type	Description
Session ID	long	The store session ID
Sequence number	long	The message sequence number
Partition ID	int	The partition ID
Transaction ID	long	The transaction ID

Record Response

Field	Data Type	Description
Session ID	long	The store session ID
Sequence number	long	The message sequence number
Partition ID	int	The partition ID
Record	Record	The transaction record

Record Header List Request

Field	Data Type	Description
Session ID	long	The store session ID
Sequence number	long	The message sequence number
Partition ID	int	The partition ID
Transaction ID	long	The transaction ID
Max number of records	int	The maximum number of records to fetch

Request Header List Response

Field		Data Type	Description
Session ID		long	The store session ID
Sequence number		long	The message sequence number
Partition ID		int	The partition ID
Record header List	Length	int	The number of record headers
	List	RecordHeader[]	The list of transaction record headers

Record List Request

Field	Data Type	Description

Session ID	long	The store session ID
Sequence number	long	The message sequence number
Partition ID	int	The partition ID
Transaction ID	long	The transaction ID
Max number of records	int	The maximum number of records to fetch

Record List Response

Field		Data Type	Description
Session ID		long	The store session ID
Sequence number		long	The message sequence number
Partition ID		int	The partition ID
Record List	Length	int	The number of records
	List	Record[]	The list of transaction record headers

Success Response

Field	Data Type	Description
Session ID	long	The store session ID
Sequence number	long	The message sequence number
Partition ID	int	The partition ID

Failure Response

Field	Data Type	Description
Session ID	long	The store session ID
Sequence number	long	The message sequence number
Partition ID	int	The partition ID

Exception	StorageRpcException	The exception information

StorageRpcException

Field			Data Type	Description
Message			String	The exception message
Stack Trace	Length		int	The number of stack trace elements
	*repeat	Class name	String	The class name
		Method name	String	The method name
		File name	String	The file name
		Line number	int	The line number

Replica Assignments

The assignment of replicas to the storage serves are stored in Zookeeper. The ZNode path is `<cluster root>/store/assignment`. This data is initialized when a cluster is configured. It is a static data for now.

Replica Assignments

Field				Data Type	Description
Number of replicas				int	The number of replicas per partition
Map of Storage to Partition ID List	Length			int	The number of entries
	*repeat	Storage	Connect String	String	The connect string (host:port)
			Replication ID	byte	The replication ID
		List	Length	int	The size of the partition ID list
			List	int[]	The list of partition IDs

[END]