Design Summary

Each service has a local transaction service (TXS). The clocks on the services MUST be synchronized. New read/write transactions MUST start on the quorum leader. New read-only transactions MAY start on any service that is joined with the met quorum. Transaction starts and ends are ONLY recorded in the local TXS.

Any open transaction pins the commit point on which it is reading. Any commit point GT that commit point is also pinned. The *minReleaseAge* specifies the minimum time that a commit point MUST be retained and thus can also pin commit points. For HA, the minReleaseAge MUST be GT ZERO (0). When the minReleaseAge is ZERO (0), the RWStore will use its “session protection” mode. When the minReleaseAge is GT ZERO (0), deletes of records are logged on “delete blocks” for eventual recycling once the commit point in which they were deleted has been recycled. The HA replication mechanism relies on a non-zero retention period - it CAN NOT be used with session protection.

During the 2-phase commit protocol, the services must agree on the new minimum commit point that will remain visible. This amounts to updating the *release time* on the TXS for the service. This agreement is reached using a distributed consensus protocol that: (a) does not block new transaction starts; (b) atomically updates the release time on each service (that is, the release time is updated while holding a lock such that the update is either applied or not applied with respect to new transaction starts or other TXS decisions); (c) preserves the monotonicity guarantee for the release time (TXS.getReleaseTime() is monotonically increasing; and (d) all services agree on the TXS.getReleaseTime() value after each 2-phase commit).

- getReleaseTime() is monotonically increasing. This value is automatically set on startup (by the Journal) and then maintained. All services joined with the met quorum must agree on the new release time. This is the purpose of the interaction during the commit protocol.
    - The releaseTime is set during startup using getLastCommitTime(). This is done in Journal.newTransactionService(), which is called from the Journal constructor, using TXS.updateCommitTimeForBareCommit().
    - The **barrierLock** on the followers is used to make the consensus protocol for the new releaseTime on the followers MUTEX with the serviceJoin. This is necessary to avoid a situation in which the service grants a tx on a commit point that has been aged out of the database by the 2-phase release age protocol. HAJournalServer includes a critical section that is run while holding the barrierLock using AbstractHATransactionServer.runWithBarrierLock(). This provides the necessary handshaking without an RMI and without exposing the barrierLock outside of the Journal

class.

- Non-blocking TX starts are always allowed for:
  - Any commit time GTE now-*minReleaseAge*.
  - Any commit time GTE the *readsOnCommitTime* of the *earliestActiveTx*.
  - The AbstractJournal.getLastCommitTime() (i.e., the current commit point is always visible).
  - If the request commit time is LT TXS.getReleaseTime(), then it is denied (non-blocking).
  - **Otherwise**, the new tx start must block during the CRITICAL SECTION (this is accomplished using the local barrierLock on the service).
- The actual earliest visible commit point (and associated timestamp) can be updated when the release time is updated. (The RWStore currently tracks the releaseTime, but it looks like the Journal and DS might need to.)
- The releaseTime consensus protocol exchanges resolves timestamps to commit points and then using the commit time associated with the resolved commit point in order to guarantee that we are using a timestamp that is consistent with the committed state for the new releaseTime (rather than depending on the current value of the local clock).
- The set of services that have a vote in the release time should be the set of services that will participate in the 2-phase commit protocol. It is Ok if a service drops out of the 2-phase commit protocol during the determination of the minimum visible commit time, e.g, with an IOException, but it MUST NOT continue to participate in the 2-phase commit. This requires that we track which services drop out during the determination of the minimum visible commit time.
- The consensus protocol verifies that the clocks are reasonably close. It does this by having each service self-report a timestamp at the moment when it sends is local minimum visible commit time to the leader. If any of those timestamps are LT the timestamp on the leader before the RMIs or GT the timestamp on the leader when the message is received, then the clocks are off and an error will fail the consensus protocol and result in a failed commit. This helps to ensure that the clocks are consistent for the 2-phase and that they will be consistent if the leader fails and another service is elected leader.

The following provides a lifeline view of the interaction between the leader and the followers required to coordinate the TXS on each service. The leader queries the followers to determine their earliest visible commit point. The leader takes the minimum over the responses of the followers. It then informs each follower of that minimum. New transactions that would reach back before the then current local minimum must block during this CRITICAL SECTION.

Leader (commitNow() Thread) **CRITICAL SECTION**	**Follower (RMI Thread)** **CRITICAL SECTION**	**Leader (RMI threads)**

• barrier = new CyclicBarrier(nfollowers, Runnable r);
• responses = new ConcurrentHashMap();
• AtomicLong *minimumVisibleCommitTime* = getEarliestVisibleCommitTime();
• Map<UUID,Future> futures = gatherMinimumVisibleCommitTime(); (Multi-cast RMI in separate threads. Does not block).
• barrier.await(timeout); // blocks
If barrier times out:
• **abort();**
• **Cancel RMI on followers.**		

	gatherMinimumVisibleCommitTime(leadersValue):void • barrierLock.lock(); • localValue = getEarliestVisibleCommitTime(); • reportedValue = min(localValue,leadersValue); • leader.notifyEarliestCommitTime(serviceId,reportedValue); // RMI	
		notifyEarliestCommitTime(serviceId,reportedValue) • responses.put(serviceId,reportedValue) • barrier.countDown(); • barrier.await(timeout); // blocks RMI thread blocks on barrier.
		When last thread meets the barrier:

		• Find minimum value of earliest visible commit time over all responses. • Set on *minimumVisibleCommitTime*. Note: Runs in the last thread to meet at the barrier.
Barrier breaks: • Leader updates local TXS with *minimumVisibleCommitTime* against which it can read. • Leader awaits RMI Futures (blocks)		Barrier breaks: • Return *minimumVisibleCommitTime* to followers. -- end of thread --
	• Follower updates local TXS with *minimumVisibleCommitTime* against which it can read. • barrierLock.unlock(); • Follower returns from RMI. -- end of thread --	
RMI Futures done. If error, abort. Leader continues 2-phase commit protocol.		

getEarliestVisibleCommitTime() {

- releaseTime = journal.getTransactionService().getLocalTransactionManager().getReleaseTime();
- commitRecord = journal._commitRecordIndex.findNext(releaseTime);
- commitTime = commitRecord==null?0L:commitRecord.getCommitTime();
- return commitTime;

}

CLUSTER: The cluster and the Journal both always know how much state is locally available by consulting the commit record index, and hence the earliest commit point that they could read on (outside of the CRITICAL SECTION). However, for the cluster this there is an additional indirection through the index to find the journal that would cover the commit point on which the caller wants to read.

Abstract of relevant code.

```
commit() {
  commitTime = localTransactionService.newTimestamp(); // TXS assigns commitTime.
  if(commitNow(commitTime)!=0L) {
     transactionManager.notifyCommit(commitTime); // TXS integration: notify did commit.
  }
}
commitNow(commitTime:long) { owns: writeLock()
   notifyCommitters();
   if(!bufferStategy.requiresCommit()) return 0L;
   rootBlockCommitter.handleCommit();
   txs.updateReleaseTimeConsensus(); // <== TXS CRITICAL SECTION GOES HERE.
   IHistoryManager.checkDeferredFrees(journal);
   newCommitRecord = ...;// Moved to after checkDeferredFrees()
   writeCommitRecord(newCommitRecord); // <== Moved to after checkDeferredFrees()
   commitRecordIndex.add(newCommitRecord);
   commitRecordIndex.writecheckpoint();
   bufferStrategy.commit(); // flush WCS to backing store. All stable except new root block.
   newRB = prepareNewRootBlock();
   if(!HA) {
     if(doubleSync) bufferStrategy.force();
     bufferStrategy.writeRootBlock(newRB);
   } else {
     nyes = quorumService.prepare2Phase(joinedServices,nonJoinedServices,newRootBlock);
     if(isQuorum(nyes)) {
       quorumService.commit2Phase(joinedServices,nonJoinedServices,token);
     } else {
       quorumService.abort2Phase(token);
     }
```

```
   }
   return commitTime;
}

RWStore.checkDeferredFrees(journal) { owns: allocationLock
  // Note: has access to journal's historicalIndexCache.
  if(sessionProtection) return;
  final long lastCommitTime = journal.getLastCommitTime();
  final long latestReleasableTime = txs.getReleaseTime();
  if(lastCommitTime <= latestReleasableTime) return;
  freeDeferrals(journal, m_lastDeferredReleaseTime + 1, latestReleasableTime);
     journal.removeCommitRecordEntries(from...,to...);
}
```

Work items.

- IHistoryManager.checkDeferredFrees() must be exposed on IHABufferStrategy (versus IHistoryManager). Probably lift a different method name onto IHABufferStrategy and have it call through to IHistoryManager.checkDeferredFrees(). May also need to override the JournalTransactionService imposed by the Journal.
- The TXS delegation pattern needs to be removed. The TXS is 100% local with this design.
- QuorumCommitImpl.commit2Phase() - should succeed if the majority of the services succeed (currently fails if any fails).
- Compressed HALog files (ideally compress the transfers as well).