Transaction：

1. 主从节点收到 Tx 后，将 Tx 封装为 Request；

Request：

2. 主从节点将构造的 Request 广播给其他节点；
   a) **存储**：主从节点将不重复的 request 存到 obcBatch.reqStore. outstandingRequests；
   b) **定时器**：主从节点判断此时没有在进行数据同步，没有在执行交易，没有在视图转换，并且 outstandingRequests 中有未处理完的 request，就软开启 newViewTimer；

RequestBatch：这部分都是主节点在处理

3. 主节点收到 Request，调用 obcBatch.leaderProcReq()进行处理：
   a) **存储**：将其存到 obcBatch.batchStore 和 obcBatch.reqStore.pendingRequests；
   b) **定时器**：检查并开启打包阈值定时器 obcBatch.batchTimer；
4. 到达打包时间或数量阈值，主节点调用 obcBatch.sendBatch()将 batchStore 中的所有 request 打包构造并广播 requestBatch；
   a) **定时器**：主节点会停止打包阈值定时器 obcBatch.batchTimer；
   b) **存储**：主节点会清空 obcBatch.batchStore 中的 request；
5. 此时主节点会进入到 pbftCore.recvRequestBatch()方法；
   a) **存储**：主节点将 requestBatch 存储到 pbftCore.reqBatchStore 和 pbftCore.outstandingReqBatches；
   b) **定时器**：主节点检查并软重置 pbftCore.newViewTimer，停止 obcBatch.nullRequestTimer；

Pre-prepare:

5.1 主节点检查 requestBatch 还没有存在 pbftCore.certStore 中，将构造并广播 Pre-prepare 消息；
   a) **存储**：将 Pre-prepare 消息存到 pbftCore.certStore 和 pbftCore.qset；

5.2. 主节点调用 maybeSendCommit()方法判断这个 Pre-prepare 消息是否到了 prepared 状态，到了就构造并发送 commit 消息；

Prepare:

6. 从节点收到 Pre-prepare 消息后，调用 pbftCore.recvPrePrepare()进行处理；
   a） **验证**：验证主节点身份、验证 n 是否在水线范围内、是否需要强制视图转换、验证是否跟 certStore 中的 Pre-prepare 消息存在冲突，后面三种不合法的话就出发视图转换；
   b） **存储**：从节点节点直接将 requestBatch 存储到 pbftCore.reqBatchStore 和 pbftCore.outstandingReqBatches，将 PrePrepare 存储到 pbftCore.certStore 和 qset 中；
   c） **定时器**：从节点软重置 pbftCore.newViewTimer，停止 obcBatch. nullRequestTimer；
7. 从节点验证当前请求到达 Pre-prepared 状态并且自己还没有发送过 Prepare 消息，就构造并广播 Prepare 消息；
8. 主从节点收到 Prepare 消息后，调用 pbftCore.recvPrepare()进行处理：
   a） **验证**：验证主节点身份、验证 v、n;
   b） **存储**：主从节点将 Commit 消息存储到 pbftCore.certStore.commit 中；

8.1 主从节点调用 pbftCore.calcPSet(),将到达 Prepared 状态的 requestBatch 信息存储到 pbftCore.pset；

8.2. 主从节点调用 pbftCore.maybeSendCommit(),判断这个 requestBatch 是否达到 Commited 状态，到了就发送 Commit 消息；

Commit:

9. 主从节点收到 Commit 消息后，调用 pbftCore.Commit()，进行如下处理：
   a）**验证**：验证 v、n;
   b）**存储**：主从节点将 Commit 消息存储到 pbftCore.certStore.commit;

主从节点验证这个 requestBatch 是否到达了 Committed 状态，到达了，进行如下处理：
   a）**定时器**：停止 pbftCore.newViewTimer;
   b）**存储**：主从节点删除 pbftCore.outstandingReqBatches 中 committed 的 requestBatch;

Execute：

9.1（开始） 调用 pbftCore.executeOutstanding()进行执行；

这个方法通过 for 循环，调用 pbftCore.executeOne(),执行 pbftCore.certStore 中一个到达 committed 状态的

requestBatch;

9.1.1　在 pbftCore.executeOne()中，验证是否落后，落后就进行数据同步，不落后分两种情况：

第一，如果是 null request,直接调用 pbftCore.execDoneSync()；

第二，如果是正常的 requestBatch，调用 pbftCore.consumer.execute()，然后在 pbftCore.consumer 是 obcBatch,所以，本质上是调用 obcBatch.executeOne()；

9.1.2　在 obcBatch.executeOne()中，先调用 obcBatch.reqStore.remove(req)，

a) **存储**：删除 reqStore 中的要执行的 request；

然后，将 requestBatch 变成 txs，调用由【执行模块】实现的接口 obcBatch.stack.Execute()；其他模块执行完后，会调用 obcBatch 实现的 Executed()接口，最终会返回一个 executedEvent 事件；

9.1.3 在 obcBatch.ProcessEvent()中 executedEvent 分支会调用由【提交模块】实现的 obcBatch.obcGeneric.stack.Commit()接口，其他模块提交完这个块后，肯定最终会返回一个 committedEvent 事件，然后再返回 execDoneEvent 事件；（其实在 executed 之后是没有任何处理的，直接调用了 Commit 进行提交，所以这里可以直接执行并提交就可以，估计写这个代码的人做过传统的分布式，熟悉两阶段提交等）

9.1.4 在 pbftCore.ProcessEvent(event）的 execDoneEvent 分支，调用 pbftCore.execDoneSync()，这个时候是共识模块更新相关变量 lastExec、currentExec，检查是否需要 gc；注意，在 pbftCore.execDoneSync()的结尾又调用了 9.1 中的 pbftCore.executeOutstanding（），继续执行 committed 的 requestBatch；

9.1（结束）pbftCore.executeOutstanding()结束的时候，调用 pbftCore.startTimerIfOutstandingRequests()；如果 outstandingReqBatches 中还有需要待处理的 requestBatch,软重置 newViewTimer；如果没有待处理的 requestBatch 就重启 nullRequestTimer。

Checkpoint:

10. 经过 K 轮之后，主从节点调用 pbftCore.Checkpoint()，构造、存储并广播 Checkpoint；

a) **存储**：将 Checkpoint 到自己的 pbftCore.chkpts 中，chkpts 存的是<n,id>序号，链状态；

11. 主从节点收到 checkpoint 之后，调用 pbftCore.recvCheckpoint()进行处理；

11.1 首先，调用 pbftCore.weakCheckpointSetOutOfRange(Checkpoint)，验证这个 Checkpoint 的编号是不是不在水线范围内；

如果低于我的最低水线说明其他节点落后了，不做处理；（因为落后的节点也会收到 checkpoint，会发现自己落后的）；

如果高于我的最高水线 H，则将这个 Checkpoint 存到 pbftCore.hChkpts 中，如果有 f+1 个 checkpoint 的序号都比我的 H 高，说明我落后了，需要将 skipInProgress 置为 true 进行状态同步，并将 batch 的缓存清空，移动水线；

11.2 到这里说明这个 Checkpoint 消息的序号在我的水线范围之内，如果收到了 f+1 个 Checkpoint，说明对这次 gc 到达了弱一致性，这个时候节点调用 pbftCore.witnessCheckpointWeakCert(Checkpoint),将 highStateTarget 字段更新为弱一致性的 chkpt 序号，这样如果自己落后了，就可以提前进行状态更新；

11.3 如果此时,pbftCore.checkpointStore 中有 2f+1 个同样序号的 checkpoint；

如果在 pbftCore.chkpts 中没有存此序号对应的链状态（id）说明我落后了，返回 nil，其他部分会发现我落后了，再进行同步处理；

如果在 pbftCore.chkpts 中有此序号对应的链状态，说明到达稳定检查点，调用 moveWatermarks()，再调用 pbftCore.processNewView()；

11.4 接下来说一下 pbftCore.moveWatermarks()函数做了什么。首先，更新低水线为当前的稳定检查点 h= n / instance.K * instance.K；随后就是清除过期的缓存；

a) **存储**：主从节点删除 pbftCore.certStore、pbftCore.reqBatchStore、pbftCore.checkpointStore、pbftCore.pset、pbftCOre.qset 中低于新的最低水线的 requesBatch；删除 pbftCore.chkpts 中序号低于 h 的全局状态；

11.4.1 在移动水线 moveWatermarks 方法里面，最后会调用 pbftCore.resubmitRequestBatches()方法，如果 pbftCore.outstandingReqBatches 中存在还没进入到共识阶段的 requestBatch，主节点调用 pbftCore.recvRequestBatch()继续对这些块进行共识；

View-Change:

当发现主节点作恶，newTimer 定时器超时，或者需要进行强制视图转换等，构造并发送 vc 消息。

15. 主从节点调用 pbftCore.sendViewChange()构造并发送 vc 消息；这个函数的主要作用是计算 vc 中的 Pset 和 Qset，清空相关缓存，构造并广播 vc 消息；
15.1 首先，停掉 newViewTimer 定时器，然后 view++;
15.2-15.3：其次，调用 calcPSet()、calcQSet()更新 Pset 和 Qset,这里的话是在原来 Pset 和 Qset 基础上，将根据 pbftCore.certStore，将到达 prepared 和 preprepared 的 requestBach 消息分别追加到 Pset、QSet 中；
15.4-15.6：在构造 vc 之前，是不是可以清除一下小于这个视图的一些缓存？
a)**存储**：清除 pbftCore.certStore、pbftCore.newViewStore、pbftCore.viewChangeStore 中小于新视图编号的共识消息，viewChange、newView 消息；
15.7 构造 viewChange，其中 CSet 是 pbftCore.chkpts 中自己构造的 checkpoint；然后，对 viewChange 消息签名，并广播；
15.8 在 sendViewChange()的最后，会开启 pbftCore.vcResendTimer 定时器；

NewView:
16. 主从节点收到 vc 消息后，将会调用 pbftCore.recvViewChange(vc)进行处理，他的主要作用是验证 vc 的合法性，若收到 2f+1 个合法的 vc，返回 viewChangeQuorumEvent 事件；
16.1 首先，调用 pbftCore.verify(vc)进行验签；
16.2 其次，调用 pbftCore.correctViewChange(vc)验证 vc 的合法性，验证 Pset、Qset 中消息的视图编号是否比 vc 中的视图编号小，n 是否在高低水线之间；验证 Cset 中的 Checkpoint 是否在高低水线之间；
16.3 如果这个 vc 是合法的，那么将调用 16.3 pbftCore.viewChangeStore(vc)，将 vc 存到自己的 viewChangeStore 中；
16.4 此时，判断一下，若收到 f+1 个视图编号比节点自己的 view 大的合法的 vc，那么以 f+1 个 vc 中最小的 view 编号构造并发送 vc；这个其实是论文的活性证明里面提到的，是为了尽快的进入到下一个视图；
16.5 最后，若收到 2f+1 个视图编号和节点自己的 view 一样大的合法的 vc，并且没有在视图转换调用 pbftCore.vcResendTimer.Stop()，关闭关闭 pbftCore.vcResendTimer 定时器；调用 instance.startTimer()，开启 pbftCore.newViewTimer；返回 viewChangeQuorumEvent 事件；

17. 在 pbftCore.ProcessEvent()中 viewChangeQuorumEvent 分支，主节点调用 pbftCore.sendNewView()，他的主要作用就是构造并广播 newView 消息；
17.1 首先，根据 viewChangeStore 计算 vset;根据 vset 计算最近的稳定检查点 cp；根据 cp 和 vset 计算 xset,xset 是个 map，map 里面存的是<n,batch 的摘要>对；
17. 2 最后，计算好了相关变量，那就构造、存储并广播 NewView 消息
a) 存储：将 newView 存到 pbftCore.newViewStore 中；
主节点直接调用 pbftCore.processNewView()进行处理；

18. 从节点收到 newView 后，调用 pbftCore.recvNewView()验证 newView 消息的合法性，主要验证 view、primary 验证 Vset 中签名的合法性；如果 newView 合法，
a) **存储**：将 newView 存到自己的 pbftCore.newViewStore 中
18.1 随后调用 pbftCore.processNewView(nv)进行处理，首先，验证 CSet,XSet 的合法性（同主节点计算过程）分三种情况：
1）如果 pbftCore. lastExec < cp.SequenceNumber&&之间的 batch 都已经到达了 committed，返回 nil；
2）如果 pbftCore. lastExec < cp.SequenceNumber&&不能自己执行到 cp，则调用状态更新方法；
3）到这里是 pbftCore. lastExec >=cp.SequenceNumber,此时，验证自己是否存有 Xset 中对应的 batch；
a) 存在缺失的 batch，调用 pbftCore.fetchRequestBatches()；
b) 不缺失 batch,调用 pbftCore.processNewView2(nv)；
18.2 然后，主从节点调用 pbftCore.processNewView2(nv)进一步对 newView 进行处理，
a)**定时器**：首先关闭 newViewTimer 定时器，关闭 pbftCore.nullRequestTimer；
根据 XSet 构造 Pre-prepare 消息并存储到 pbftCore.certStore、pbftCore.qset 中；

18.2.1 如果是从节点，则构造并广播 Prepare 消息；
18.2.2 如果是主节点，调用 pbftCore.resubmitRequestBatches()，将在 pbftCore.outstandingReqBatches 中还没进入到共识阶段的 requestBatch,调用 pbftCore.recvRequestBatch()发起共识;
18.2.3 最后的时候，主从节点都会调用 pbftCore.startTimerIfOutstandingRequests()，
a)**定时器**：如果 outstandingReqBatches 中还有 requestBatch，软启动 newViewTimer，否则就重启 nullRequestTimer 定时器；
18.2.4 返回 viewChangedEvent 事件；
19. 在 obcBatch.ProcessEvent()的 viewChangedEvent 分支，
a) **存储**：清空 pendingRequests 并将 obcBatch.pbft.cerStore 中的 req 加入到 pendingRequests；然后，调用 obcBatch.resubmitOutstandingReqs(),将在 outstandingRequests 但不在 pendingRequests 中的 req Inject 到 Queue 中；

StateUpdate：
两种情况发现自己需要进行状态同步：
1、在 pbftCore.weakCheckpointSetOutOfRange()中，收到了 f+1 个序号超过自己高水位的 checkpoint，说明数据落后了，将 skipInProgress 置为 true；
2、 pbftCore.processNewView()中，发现自己落后 newview 中的 checkpoint 的时候会调用 pbftCore.stateTransfer()将 skipInProgress 置为 true，进行同步；
20. 调用 pbftCore.retryStateTransfer()方法进行数据同步；通过 currentExec 字段判断是否有区块正在执行，通过 stateTransferring 字段判断是否正在进行数据同步，都没有的话就进行数据同步 ；
20.1 最终会调用其他模块的 Executor.UpdateState()方法进行状态同步；执行完后会返回 stateUpdatedEvent 事件；
21. obcBatch.ProcessEvent()中的 stateUpdatedEvent 分支，
a) **存储**：清空 obcBatch.reqStore 中的 request，没用了；
22. pbftCore.ProcessEvent()中的 stateUpdatedEvent 分支，
a) **存储**：删除 pbftCore.chkpt 中小于 n 的消息，移水线；