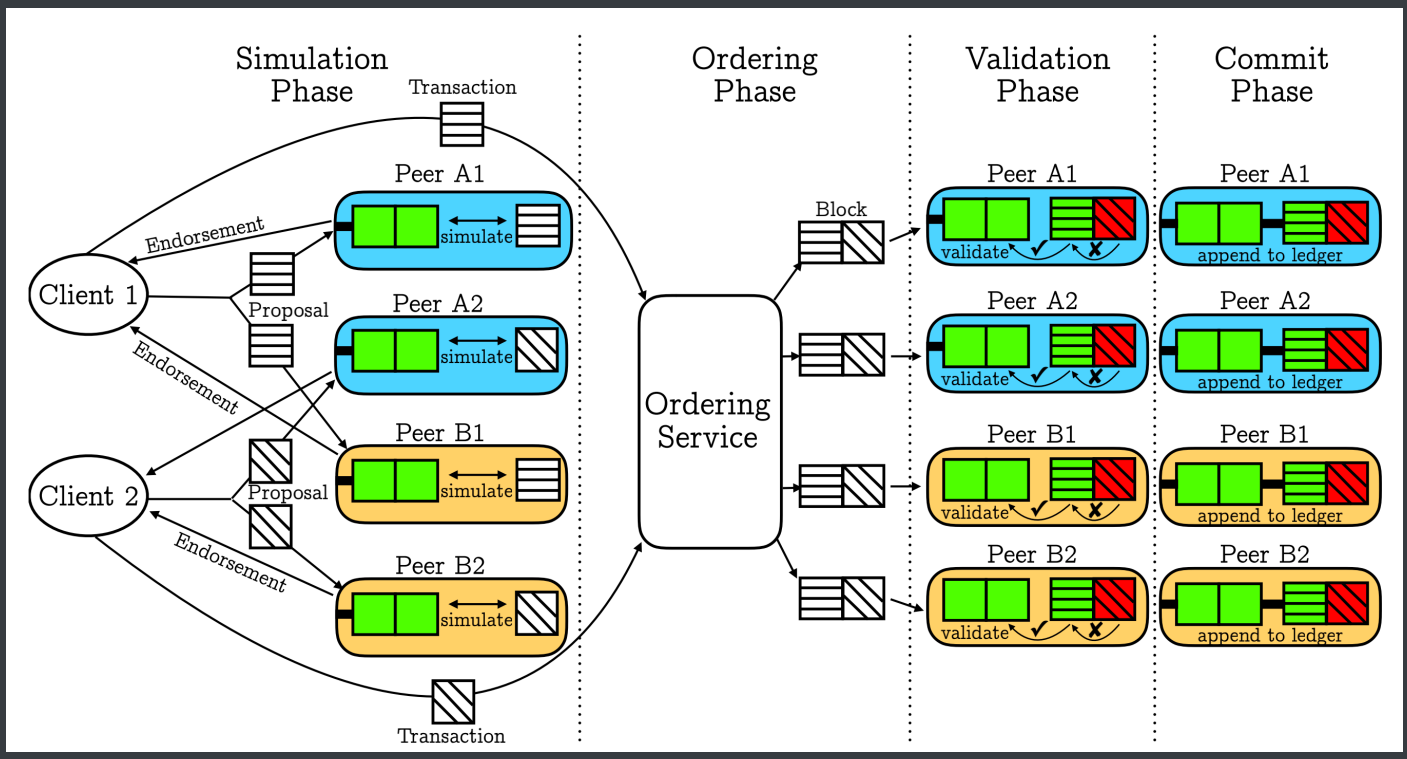
Fabric 提高TPS吞吐量的改造

Fabric生命周期的图



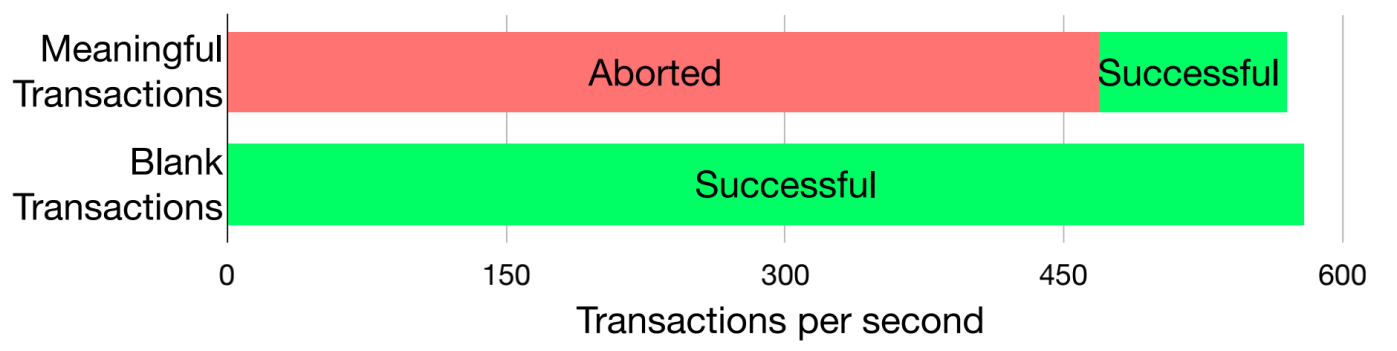
交易的生命周期：

1. SDK 生成 Proposal，其中包含调用链码的相关参数等信息，并将 Proposal 发送到多个不同的 peer 节点
2. peer节点收到来自client的proposal 请求
    1. peer 根据 Proposal 的信息，调用用户上传的链码
    2. 链码处理请求，将请求转换为对账本的读集合和写集合
        - 链码设计优化的目标：如何更快的处理多笔交易？
    3. peer 对读集合和写集合进行签名，并将 ProposalResponse 返回给 SDK。
        - 读集合：该笔交易中会涉及到的各个键，以及各个键的现在所处的版本
        - 写集合：更新该交易所涉及到的键，将其历史版本更新到最新版本
    4. （注：如果是查询就到这里戛然而止了）
3. SDK 收到多个 peer 节点的 ProposalResponse，并将读集合与写集合和不同节点的签名拼接在一起，组成 Envelope
4. SDK 将 Envelope 发送给 orderer 节点，并监听 peer 节点的块事件
5. orderer 节点收到足够的 Envelope 后，生成新的区块，并将区块广播给所有 peer 节点
6. Commit peer 节点对收到区块进行验证，并向 SDK 发送新收到的区块和验证结果

    1. commit peer 根据所收到的envelop中所包括的各个交易，进行对各个交易的读写集进行验证工作，如果各个交易的键值对版本发生了冲突，那么只会有一个交易成功入链
7. SDK 根据事件中的验证结果，判断交易是否成功上链

作者为什么认为交易重排会提升fabric的吞吐量呢?

1. 有效交易和无效交易都可以入链
2. 那么作者的实验中造成fabric 性能瓶颈的地方在哪里呢?
    1. 交易验证 + 生命周期中的网络开销



Fabric性能提升的关键：交易合法性的验证

当Fabric-一个块中包含多笔涉及到同一键值的交易

Transaction	Read Set	Write Set	Is Valid?
1. T_1	–	$(k_1, v_1 \rightarrow v_2)$	✓
2. T_2	$(k_1, v_1), (k_2, v_1)$	$(k_2, v_1 \rightarrow v_2)$	×
3. T_3	$(k_1, v_1), (k_3, v_1)$	$(k_3, v_1 \rightarrow v_2)$	×
4. T_4	$(k_1, v_1), (k_3, v_1)$	$(k_4, v_1 \rightarrow v_2)$	×

为什么会出现版本不一致的情况?

1. fabric 使用gossip这种peer to peer 的分布式一致性协议，他是一种最终一致性协议，有可能在orderer发布区块的一段时间后，各个peer可能并不会同时接收到区块，每个节点可能接受到的区块时间不一致，最终导致每个节点的状态不一样。
2. 由于每个节点的状态不一致，所以会出现用户提出的proposal后，各个peer对该proposal的模拟结果不一样
3. 所以在验证阶段会出现版本不一致的情况

验证交易失败缺点：

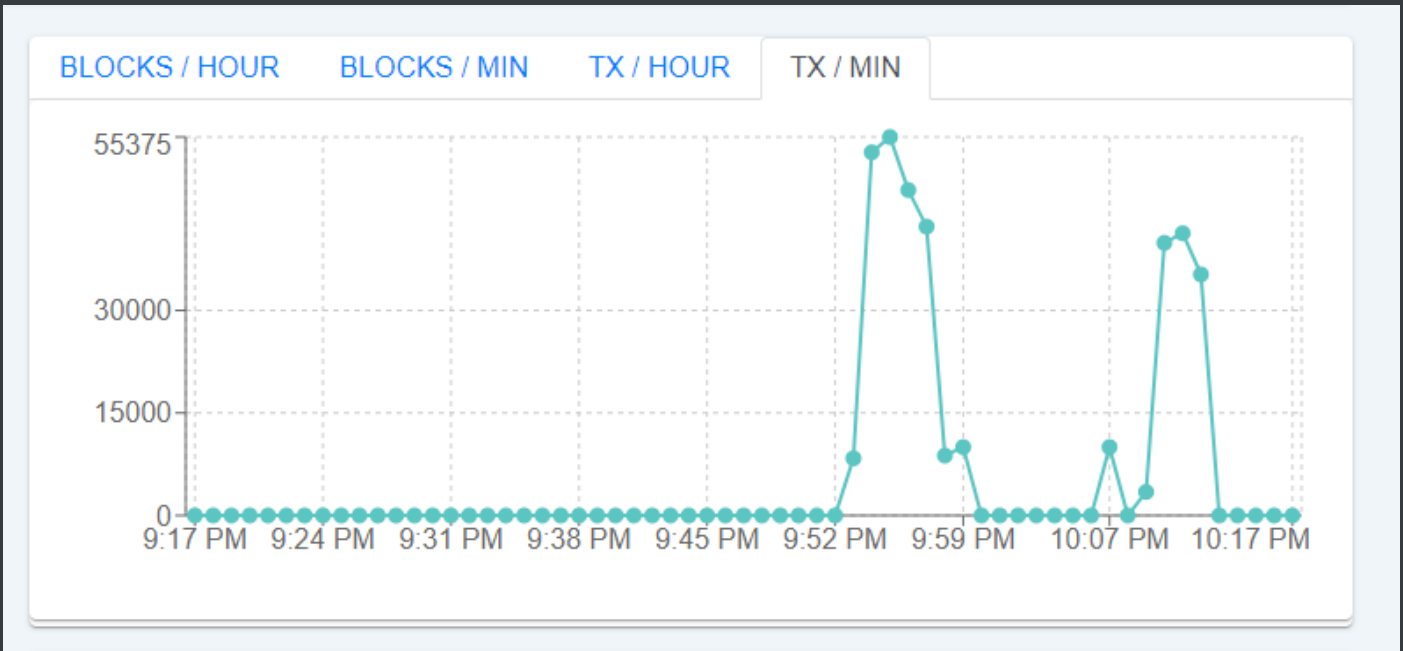
1. Fabric 中的区块会包含非法的交易，如果业务产生了大量因为 Key 冲突而失败的交易，这些交易也会被记入各个节点的账本，占用节点的存储空间。
2. 同时由于冲突的原因，并行的交易很多会失败，不但会导致 SDK 的成功 TPS 大幅下降，失败的交易还会占用网络的吞吐量。

根据fabric对交易的验证方式，现在有两种对fabric进行改造的思路

1. 在commit peer进行validate时，根据大宗商品的特殊的场景，尽可能的把对读写集进行验证的部分进行去掉。
2. 把对交易的读写集的验证工作尽可能的提前到orderer，在orderer对交易进行排序时，根据各个交易的不同的读写集的版本号，对交易进行重新排序。
    1. fabric默认的排序方式是先到的交易排在前面

目前可以达到的性能指标：1300-1500tps之间（因为fabric explorer在高并发的情境下，同步区块会占用很大的带宽，有时会崩掉，吞吐量数据只能在后台查看）

- 最近一次的测试数据，目前还没有达到fabric的吞吐量上限
  2020-10-09 06:48:46.996
  2020-10-09 06:39:34.418
  600000 - 9min左右
  i7-9700 4核 96线程
  i7-9750H 6核 80线程
  R7-4800H 8核 96线程
  系统的占用情况：系统初始化时，cpu占用率较高，最高可达97%左右，运行后期，cpu的占用率可以降下来。



一、Commit节点对Validate阶段进行改造

验证阶段需要做的事情：

1. 验证签名
2. 验证是否符合背书策略
3. 验证读写集
4. 更新couchdb
5. 更新历史账本

根据大宗商品的具体生产环境，每个交易入链时都会重新生成一个键（历史存证），在后续的生产工作中，对该键值的修改次数并不多，即对该交易的健的修改次数并不多。

在对交易进行验证时，当一个块中的多笔交易都对同一个键值进行修改时，会涉及到对该键值对版本号的验证工作，如果多笔交易中对同一键值所修改的版本号不一致时，这时只会有一笔交易会通过认证，入链。

因此可以根据大宗商品这种特殊的交易场景，对验证阶段进行改造，尽量的避开验证多笔交易的读写集的阶段，尽可能多的让每一笔交易成功入链，从而提高区块链的吞吐量，提高系统的tps

二、Orderer交易重排 + 提前阻断

如果能够提前的检测出一笔交易是非法交易，就可以提前中断该交易的生命周期，节省出一大部分的时间成本。

对于数据库来讲，根据数据库的ACID原则（原子性、一致性、隔离性与持久性），在交易生成后就可以判定一个交易是否合法，即要么执行该交易，要么不执行该交易，而不会在持久化时在去判断一个交易是否合法。但是在fabric中，验证一笔交易是否和法是在validation阶段完成的，而不是在simulation阶段完成的，那么，是否可以在simulation阶段提前中断一笔交易的合法性呢?

可以在commit phase 之前，还有三个提前中断非法交易的机会，分别是simulation、ordering和validation

---

2.1 交易重排

当对块内的交易进行排序之后，fabric的吞吐量会有很大程度上的提高

Transaction	Read Set	Write Set	Is Valid?
1. T_4	$(k_1, v_1), (k_3, v_1)$	$(k_4, v_1 \rightarrow v_2)$	✓
2. T_2	$(k_1, v_1), (k_2, v_1)$	$(k_2, v_1 \rightarrow v_2)$	✓
3. T_3	$(k_1, v_1), (k_3, v_1)$	$(k_3, v_1 \rightarrow v_2)$	✓
4. T_1	–	$(k_1, v_1 \rightarrow v_2)$	✓

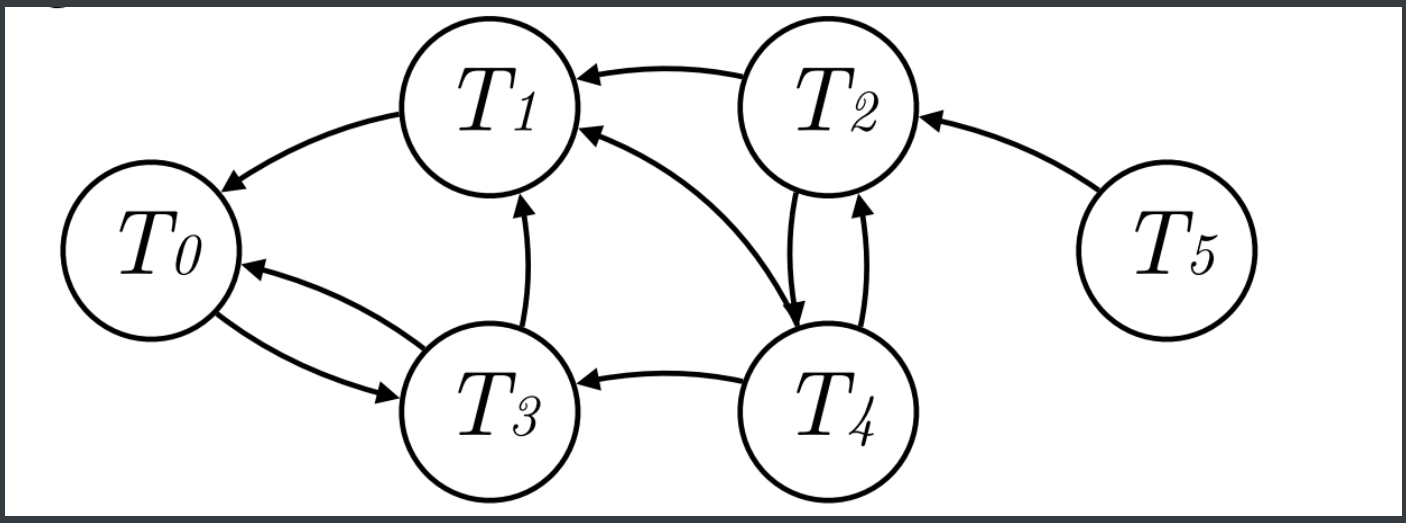
那么，使用什么算法，对块中对交易进行重新排序？？

1. 生成一个块，并拿到了这个块中的所有交易

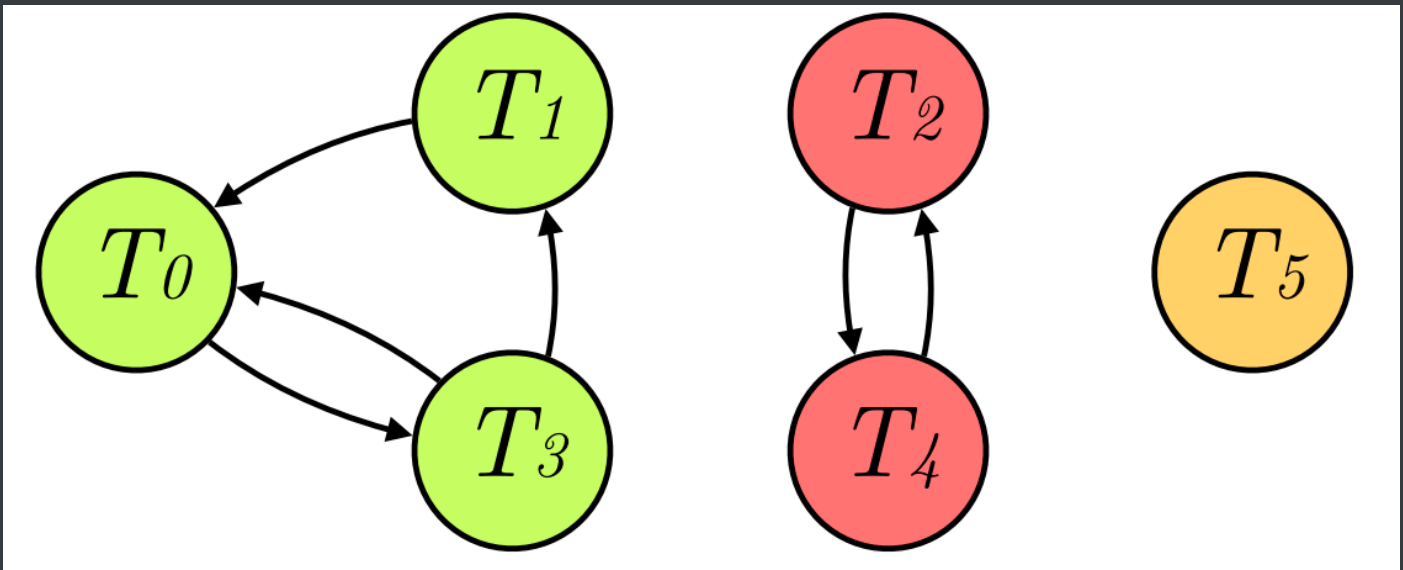
Transactions	Read Set									
	K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8	K_9
T_0	**1**	**1**	0	0	0	0	0	0	0	0
T_1	0	0	0	**1**	**1**	**1**	0	0	0	0
T_2	0	0	0	0	0	0	**1**	**1**	0	0
T_3	0	0	**1**	0	0	0	0	0	**1**	0
T_4	0	0	0	0	0	0	0	0	0	**1**
T_5	0	0	0	0	0	0	0	0	0	0

Transactions	Write Set									
	K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8	K_9
T_0	0	0	**1**	0	0	0	0	0	0	0
T_1	**1**	0	0	0	0	0	0	0	0	0
T_2	0	0	0	**1**	0	0	0	0	0	**1**
T_3	0	**1**	0	0	**1**	0	0	0	0	0
T_4	0	0	0	0	0	**1**	**1**	0	**1**	0
T_5	0	0	0	0	0	0	0	**1**	0	0

2. 根据该区块中所有的交易的读写集之间的矛盾关系（对K1进行修改的交易的优先级，要小于对K1进行读取的交易），生成一个冲突图



3. 利用Tarjan's algorithm算法，将交易冲突图分解为多个强连通图



4. 利用Johnsons算法，从强连通图中分解为多个环

c1 = T0 - T3 - T0

c2 = T0 - T3 - T1 - T0

c3 = T2 - T4 - T2

5. 标记出块中的每一笔交易，在各个环中出现的次数

Cycle	T_0	T_1	T_2	T_3	T_4	T_5
c_1	1	0	0	1	0	0
c_2	1	1	0	1	0	0
c_3	0	0	1	0	1	0
$\sum$	2	1	1	2	1	0

6. 从出现在各个环中次数最多的交易开始，增量的中断这些交易，并刷新各个交易在环中出现的次数，直到强连通图中所有的环都被解除掉
7. 上面T0和T3的次数都是2，这里选择交易下标编号较小的一个，首先选择T0。
8. 将第一个和第二个环从表格中移除，移除交易T0，相应的，T3的次数也相应减少为0。
9. 现在上面表格只剩下环c3，将环3移除，移除交易T2。
10. 此时表格中没有环了，算法结束。

Table 5: Removing T_0 clears the cycles c_1 and c_2.

Cycle	~~T_0~~	T_1	T_2	T_3	T_4	T_5
~~c_1~~	~~1~~	~~0~~	~~0~~	~~1~~	~~0~~	~~0~~
~~c_2~~	~~1~~	~~1~~	~~0~~	~~1~~	~~0~~	~~0~~
c_3	0	0	1	0	1	0
$\sum$	0	0	1	0	1	0

Table 6: Removing T_2 clears the last cycle c_3.

Cycle	~~T_0~~	T_1	~~T_2~~	T_3	T_4	T_5
~~c_1~~	~~1~~	~~0~~	~~0~~	~~1~~	~~0~~	~~0~~
~~c_2~~	~~1~~	~~1~~	~~0~~	~~1~~	~~0~~	~~0~~
~~c_3~~	~~0~~	~~0~~	~~1~~	~~0~~	~~1~~	~~0~~
$\sum$	0	0	0	0	0	0

7. 对冲突图中的各个交易进行合理的排序，并重新打包为一个新的块

尽可能将读读取键值的交易放在前面，写入键值的交易放在后面，这样就能最小化写交易带来的版本更新的负面影响。

注：在Hyperledger Fabric的重排序阶段，接收多少个交易之后才能打包成一个区块呢?

- 达到一定数量的交易。
- 达到了一定的数据大小。
- 从接收到这些交易的第一个交易开始，已经过去了一定的时间。

结合上面的算法，为了避免上面表格中的不同key的数量太多，影响算法性能，论文提出增加一个打包条件：

- 所接收到的所有交易中的不同key集合达到一定的数量。

2.2 Tarjan算法 – 将图分解为强连通图

维基百科

```java
public void tarjan(int cur, int pre) {
    dfn[cur] = low[cur] = time++;
    // 遍历该节点的所有的邻居
    for(int next : adj[cur]) {
        if(next == pre) continue; // 如果访问的是父节点，则跳过
        if(dfn[next] == 0) { //如果没有访问过该邻居，则访问
            tarjan(next, cur);
            low[cur] = Math.min(low[cur], low[next]); // 取该节点和起邻居节点，能够回访到的最早时间
            // TODO: 想一想为什么这是关键路径(low[next] > dfn[cur])?  如果求的是割点（low[next] >= dfn[cur]）应该怎么办?
            //  割边：next可以回访到的最早节点，比cur节点还要晚，所以cur -> next是割边。
            //  割点：next可以访问到的最早节点最多就是cur，所以cur是割点。
            if(low[next] > dfn[cur]) { // cur -> next 这条边是桥
                ans.add(Arrays.asList(cur, next));
            }
        } else { // 如果访问过该邻居
            // TODO: 为什么访问过该节点不去找next能访问到到最早节点，而是找next的访问时间
            low[cur] = Math.min(low[cur], dfn[next]);
        }
    }
}
```

1. DFN [ ] 作为这个点搜索的次序编号（时间戳），简单来说就是 第几个被搜索到的。每个点的时间戳都不一样。
2. LOW[ ] 作为每个点在这颗树中的，最小的子树的根，每次保证最小，他能访问的到的最小的父亲结点的时间戳。如果它自己的LOW [] 最小，那这个点就应该从新分配，变成这个强连通分量子树的根节点。

最外层循环用于查找未访问的节点，以保证所有节点最终都会被访问。 strongconnect进行一次深度优先搜索，并找到节点v的后继节点构成的子图中所有的强连通分量。

当一个节点完成递归时，若它的lowlink仍等于index，那么它就是强连通分量的根。算法将在此节点之后入堆栈（包含此节点）且仍在堆栈中的节点出堆栈，并作为一个强连通分量输出。

1. 复杂度:对每个节点，过程strongconnect只被调用一次；整个程序中每条边最多被考虑一次。因此算法的运行时间关于图的边数是线性的，即 O(|V|+|E|)}。
2. 判断节点v'是否在堆栈中应在常量时间内完成，例如可以对每个节点保存一个是否在堆栈中的标记。
3. 同一个强连通分量内的节点是无序的，但此算法具有如下性质：每个强连通分量都是在它的所有后继强连通分量被求出之后求得的。因此，如果将同一强连通分量收缩为一个节点而构成一个有向无环图，这些强连通分量被求出的顺序是这一新图的拓扑序的逆序 。

2.3 Johnson算法 – 将强连接图中的环分解出来

Johnson算法简介

```
// 伪代码
```

时间复杂度上界为O((n+e)(c+1))，空间复杂度为O(n+e)，其中n为顶点数，e为边数，c为存在环数。在算法执行时，每两个连续的环的输出之间的时间不会超过O(n+e)。

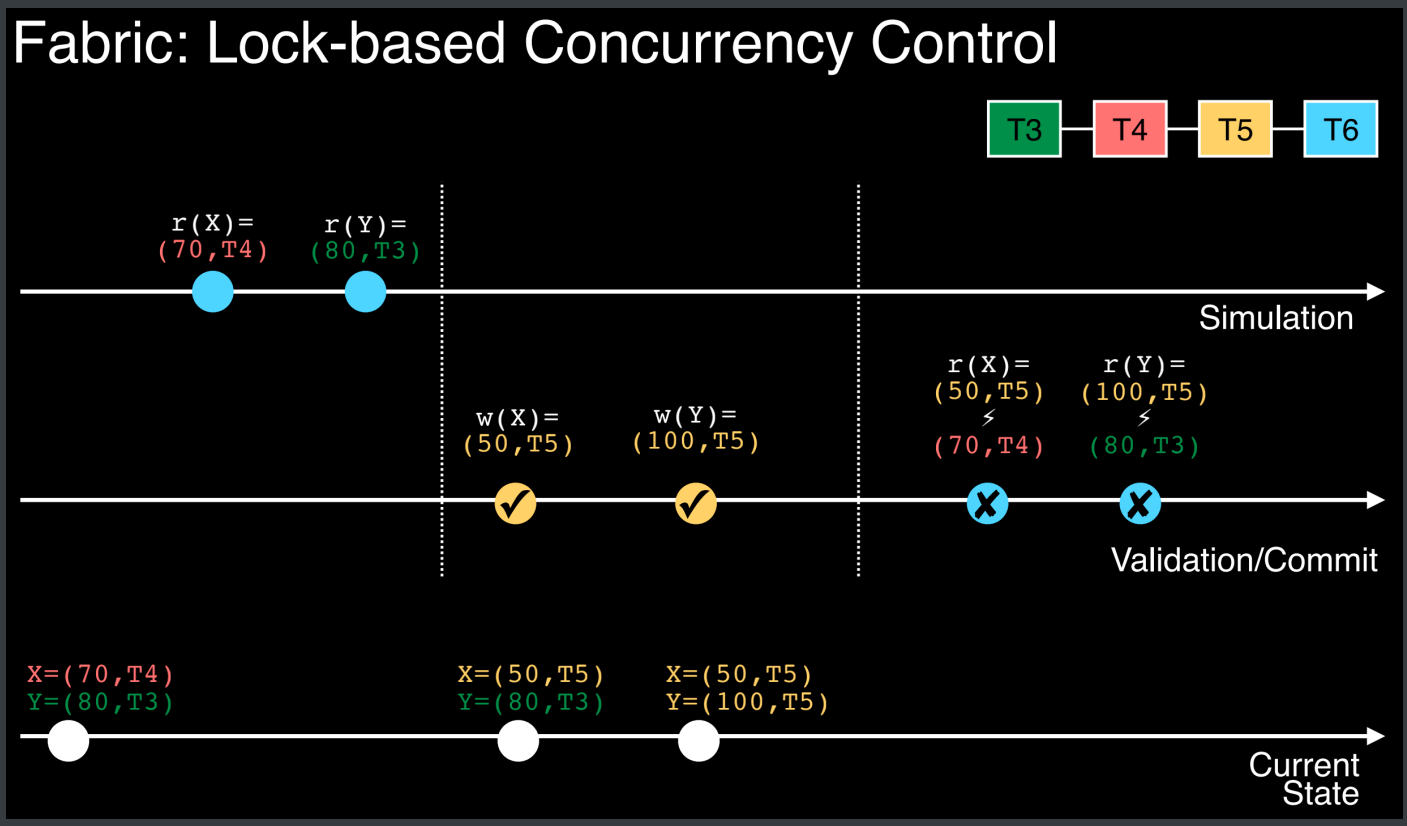
2.4 提前阻断

2.4.1 在模拟阶段中断交易

在fabric中simulate和validate是通过加锁的方式（即在同一peer中这两个步骤并不会同时执行），来确保并行执行是安全的，不会带来负面影响的。举例，假设T5正在chaincode中进行模拟交易，这时peer接收到含有T1 T2 T3 T4的一个块，这时对这四笔交易的验证工作会进入等待状态，直到T5模拟执行结束后，才会进入到对这四笔交易的验证阶段。

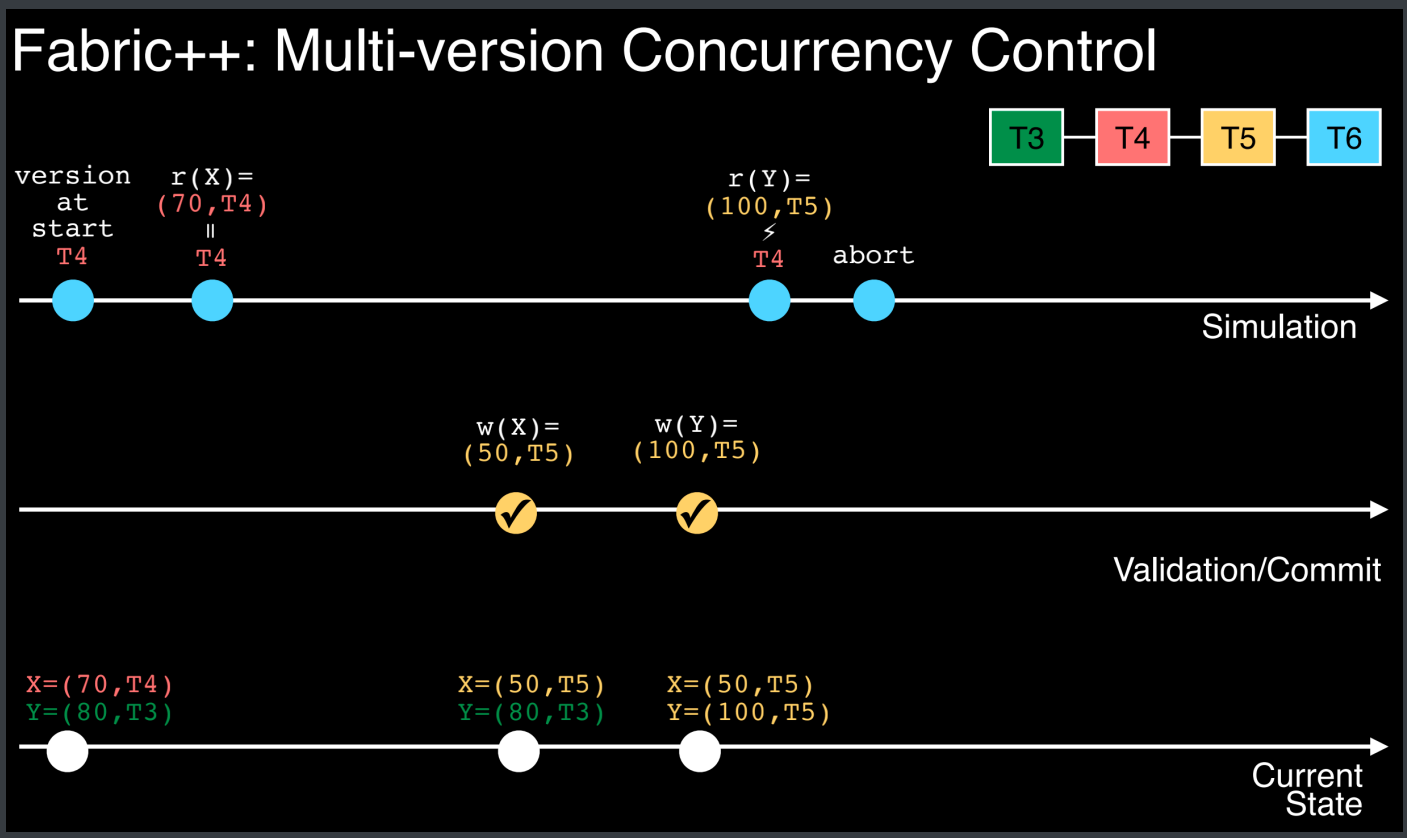
在fabric++中，作者利用fabric的mvcc的机制把锁去掉了，保证在同一个peer中可以同时进行simulate和validate，而不会因为对方正在执行而陷入等待状态。在同时执行的过程中，simulation阶段通过不断地检查key的版本，来解决在simulate时读到的版本和在模拟前罪行的版本不一致的问题（脏读）。因为把锁去掉了，所以在simulate进行读取交易的版本时，会重新读取目前最新的key的版本，如果和上次读取的情况不一样，则说明发生了交易更新，把该交易置为非法交易。

Fabric目前对并发的处理方式-加锁：

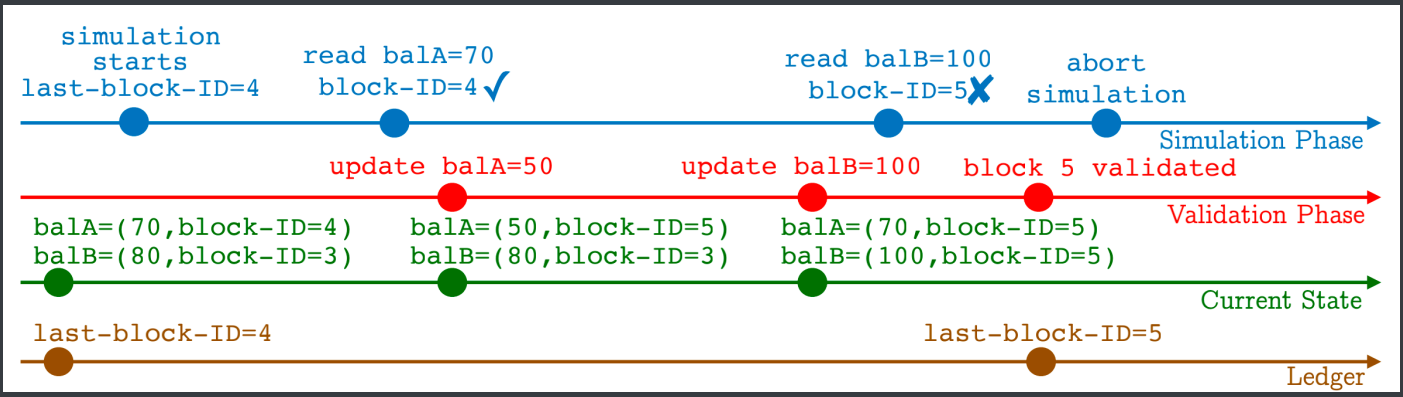


Fabric++对并发的改进方式-去掉锁然后每次重新读取key的版本：

在模拟执行一笔交易前，更新了 该交易中涉及到的键（X, Y）的最新的交易是T4



Fabric++ 去掉锁之后的效果：



源代码：在core/ledger/kvledger/txmgmt/txmgr/lockbased_txmgr.go中，实现交易管理器的并发控制功能。

- 当多笔交易涉及到对 账本/世界状态 进行更改时，需要完成对他们的并发控制。

```
// LockBasedTxMgr a simple implementation of interface `txmgmt.TxMgr`.
// This implementation uses a read-write lock to prevent conflicts between transaction simulation and committing
type LockBasedTxMgr struct {
    ledgerid            string
    db                  *privacyenabledstate.DB
    pvtdataPurgeMgr     *pvtdataPurgeMgr
    commitBatchPreparer *validation.CommitBatchPreparer
    stateListeners      []ledger.StateListener
    ccInfoProvider      ledger.DeployedChaincodeInfoProvider
    commitRWLock        sync.RWMutex
    oldBlockCommit      sync.Mutex
    current             *current
    hashFunc            rwsetutil.HashFunc
}
```

模拟交易 源码的起点在：

1. core/endorser/endorser.go --- ProcessProposal 开始处理client 发出的proposal
2. core/endorser/endorser.go --- ProcessProposalSuccessfullyOrError 通过e.Support.GetTxSimulator 获取到txmgr.commitRWLock锁
3. core/endorser/endorser.go --- SimulateProposal 在e.callChaincode调用链码模拟交易之后 通过txParams.TXSimulator.Done() 释放掉commitRWLock锁

验证交易 源码的起点在：

1. internal/peer/node/start.go --- initGossipService peer节点启动gossip服务
2. gossip/gossip/gossip_impl.go --- New 新建gossip组件之后开始start()线程，开始接收gossip信息
3. internal/peer/node/start.go --- peerInstance.Initialize 完成初始化channel的操作
4. core/peer/peer.go --- createChannel 创建channel后，最后通过p.GossipService.InitializeChannel完成初始化gossip channel的操作
5. gossip/service/gossip_service.go --- InitializeChannel 初始化gossip channel 时，创建state.NewGossipStateProvider后，使用receiveAndQueueGossipMessages 完成接收gossip信息，并添加到缓冲区

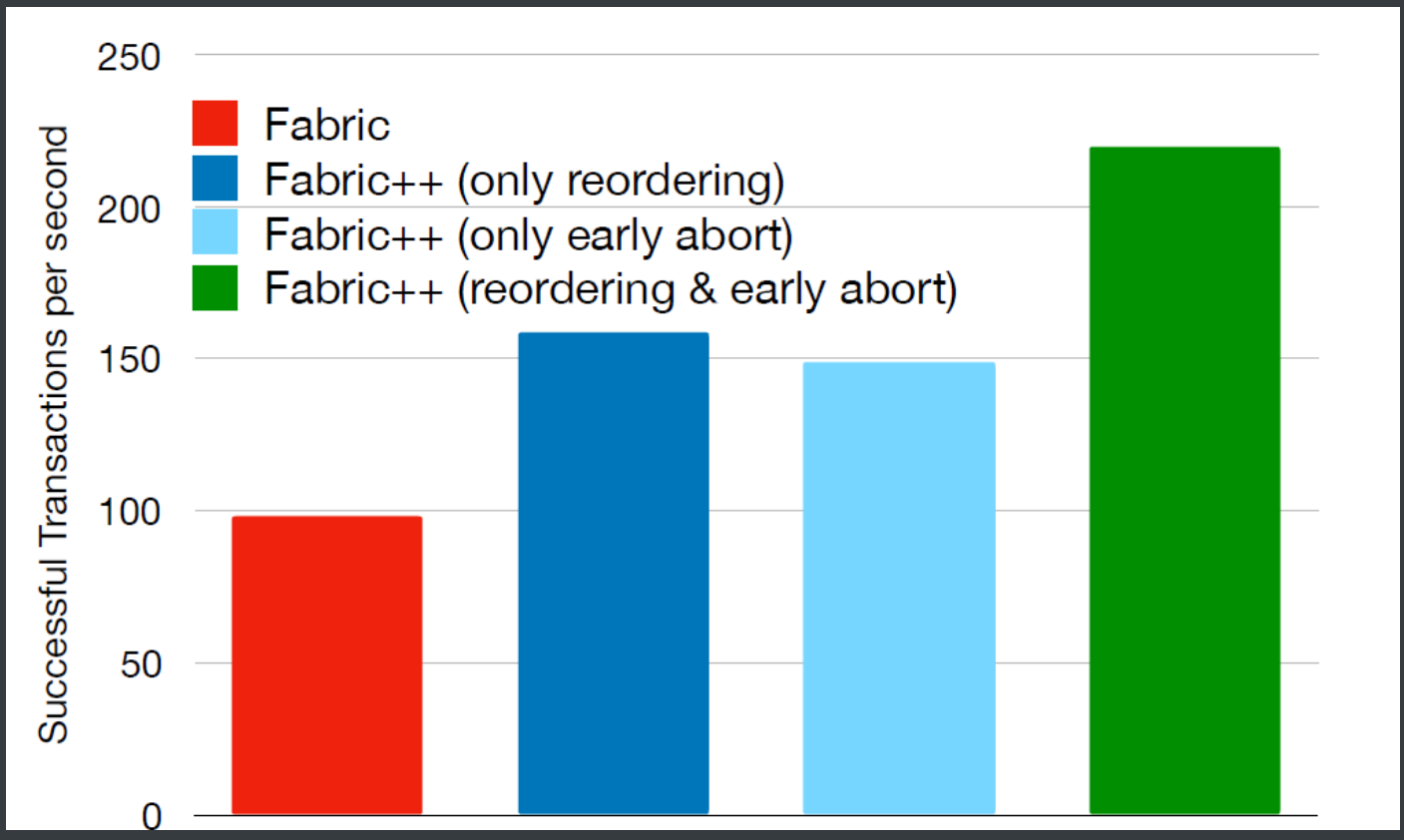
6. gossip/state/state.go --- deliverPayloads 将缓冲区中的消息取出来，并commitBlock 开始验证区块的工作
7. core/ledger/kvledger/kv_ledger.go --- commit 将收集到的区块信息提交到世界状态数据库中时，通过l.txmgr.Commit()获取到 txmgr.commitRWLock锁
8. core/ledger/kvledger/txmgmt/txmgr/lockbased_txmgr.go --- Commit 在验证交易并提交后，通过txmgr.commitRWLock.Unlock()释放掉锁

2.4.2 在排序阶段提前中断交易

除去在reorder阶段删掉强连通图中的交易之外，如果在一个区块中含有两笔对同一键值进行读取的交易，则检查这两笔交易中的键值的版本号，如果不同则抛弃排在后面(可选)的那个交易。

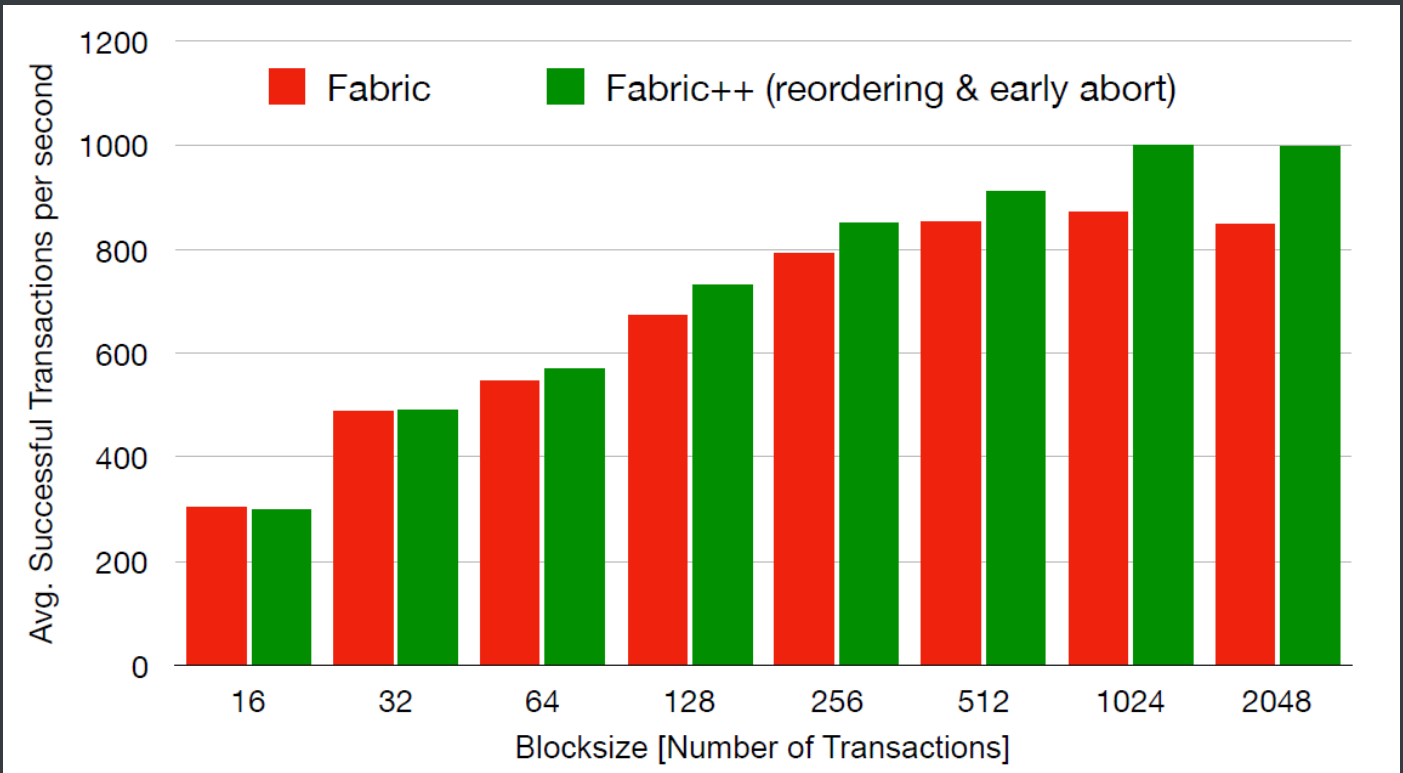
Transactions	Key/version
T1	K1/V3
T2	K3/V1
~~T3~~	~~K1/V2~~

2.5 交易重排 & 提前中断 性能提升结果



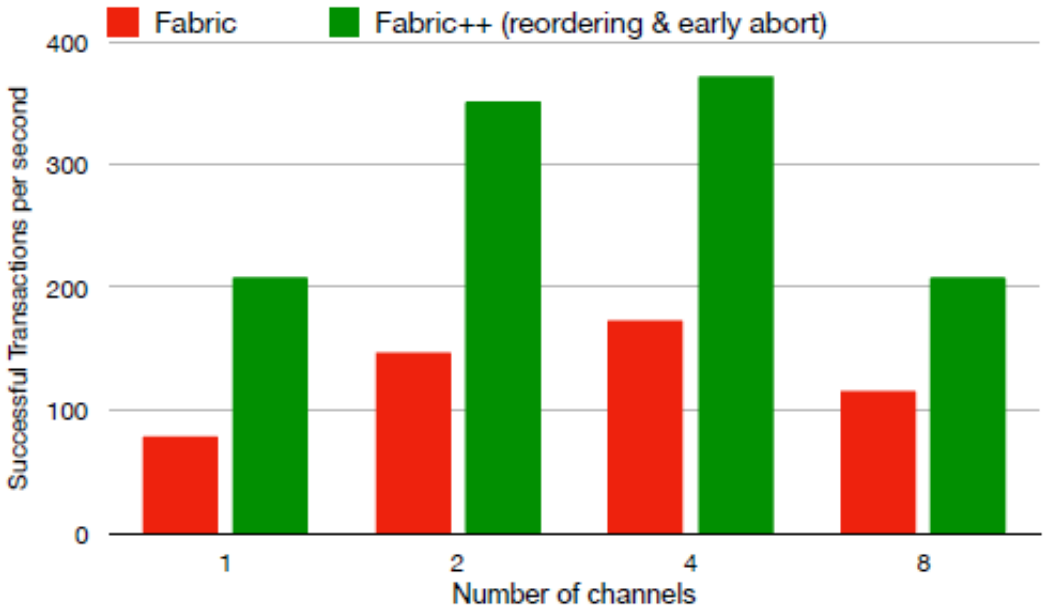
注：Blocksize也会对吞吐量的影响，fabric默认的Blocksize为10个交易，通过实验发现，增加块大小也会增加Fabric成功交易的吞吐量，由于目标是获得更高的整体吞吐量，所以使用的是1024个交易的Blocksize

- 除此之外：出块的时间 和一些其他的配置信息 也会影响系统的吞吐量

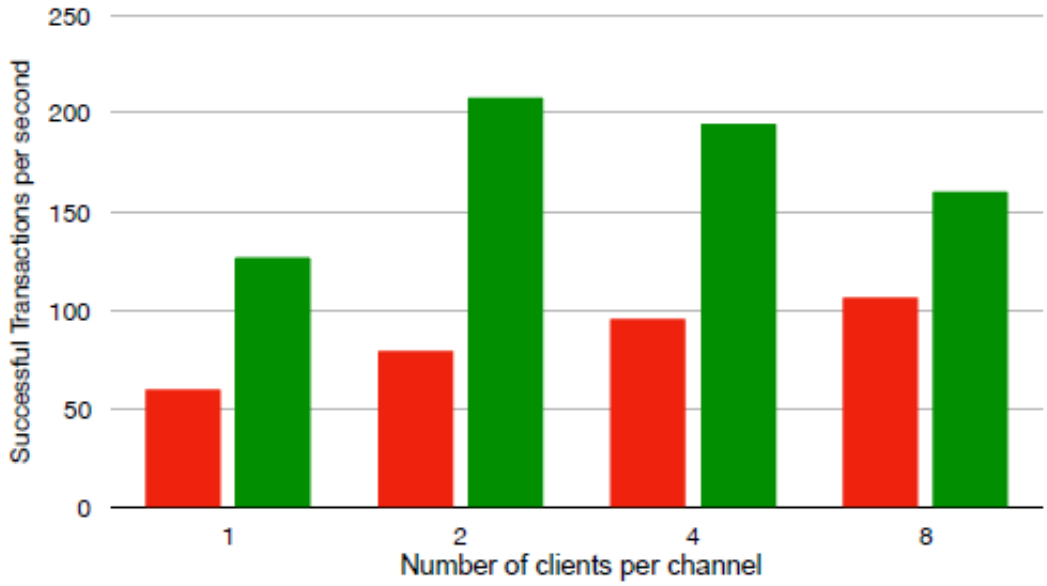


ps：通道数 / 客户端数 增加时，会引发系统的资源竞争现象，吞吐量也会随之下降。

- 一开始随着通道数量的增长，每秒成功的交易数量也逐渐增多，且Fabric++明显优越于Fabric，但是当通道数量超过4个的时候，交易成功数量又开始减少，说明通道之间会竞争资源。同理，客户端之间也会竞争资源。



(a) **Varying the number of channels** from 1 to 8. Per channel, we use 2 clients to fire the transaction proposals.



三、SmartContract 优化

链码设计的优化目标：可以让链码在单位时间内并发的处理多笔交易

1 避免Key冲突

在 Fabric 区块链账本中，数据是以 KV 的形式存储的，链码可以通过 `GetState` 、 `PutState` 等方法对账本数据进行操作。

1. 设计高效的chaincode数据模型，完全避免交易发生冲突，但局限性比较大。
    1. 局限性是什么?

添加交易：

```
func (s *SmartContract) update(APIstub shim.ChaincodeStubInterface, args []string)
pb.Response {

  // Extract the args
  name := args[0]
  op := args[2]
  _, err := strconv.ParseFloat(args[1], 64)

  // Retrieve info needed for the update procedure
  txid := APIstub.GetTxID()
  compositeIndexName := "varName~op~value~txID"

  // Create the composite key that will allow us to query for all deltas on a
particular variable
  compositeKey, compositeErr := APIstub.CreateCompositeKey(compositeIndexName,
[]string{name, op, args[1], txid})

  // Save the composite key index
  compositePutErr := APIstub.PutState(compositeKey, []byte{0x00})

  return shim.Success([]byte(fmt.Sprintf("Successfully added %s%s to %s", op,
args[1], name)))
}
```

查询交易：

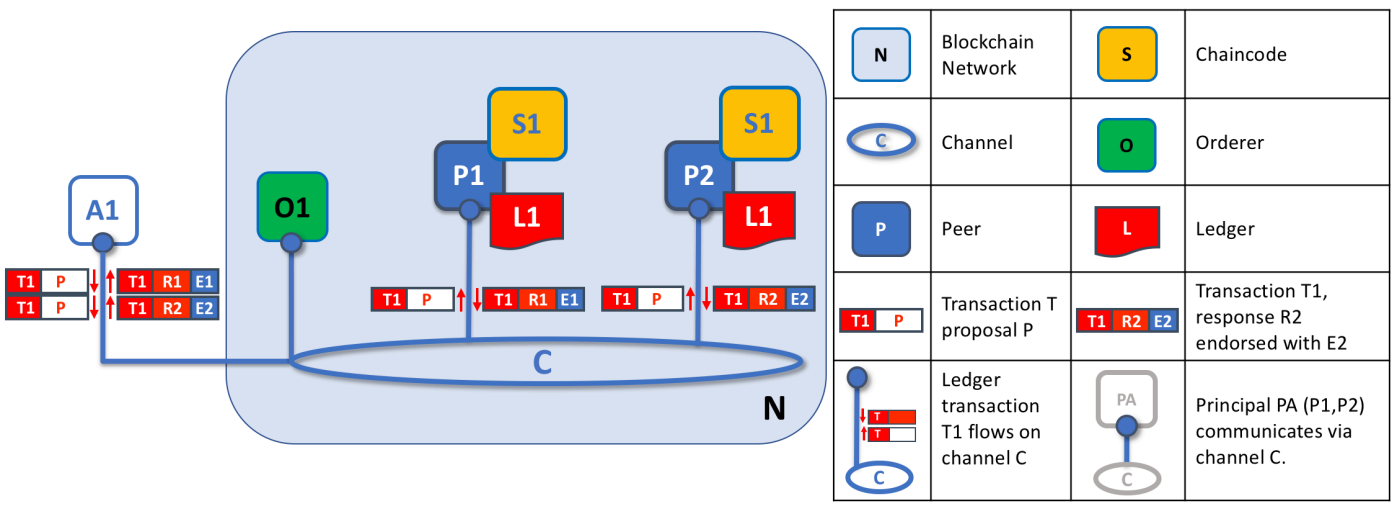
```
func (s *SmartContract) get(APIstub shim.ChaincodeStubInterface, args []string)
pb.Response {

  name := args[0]
  // Get all deltas for the variable
```

```
  deltaResultsIterator, deltaErr :=
APIstub.GetStateByPartialCompositeKey("varName~op~value~txID", []string{name})
  defer deltaResultsIterator.Close()

  // Check the variable existed
  if !deltaResultsIterator.HasNext() {
    return shim.Error(fmt.Sprintf("No variable by the name %s exists", name))
  }

  // Iterate through result set and compute final value
  var finalVal float64
  var i int
  for i = 0; deltaResultsIterator.HasNext(); i++ {
    // Get the next row
    responseRange, nextErr := deltaResultsIterator.Next()

    // Split the composite key into its component parts
    _, keyParts, splitKeyErr := APIstub.SplitCompositeKey(responseRange.Key)

    // Retrieve the delta value and operation
    operation := keyParts[1]
    valueStr := keyParts[2]

    // Convert the value string and perform the operation
    value, convErr := strconv.ParseFloat(valueStr, 64)

    switch operation {
    case "+":
      finalVal += value
    case "-":
      finalVal -= value
    default:
      return shim.Error(fmt.Sprintf("Unrecognized operation %s", operation))
    }
  }

  return shim.Success([]byte(strconv.FormatFloat(finalVal, 'f', -1, 64)))
}
```

2 减少Stub读取和写入账本的次数

Fabric 中的链码与 peer 节点之间的通信与 SDK 和区块链节点的通信类似，也是通过 GRPC 来进行的。

1. 当在链码中调用查询、写入账本的接口时（例如 `GetState` 、 `PutState` 等），链码容器发送 GRPC 请求给 peer 节点。
2. Peer查询本地的账本，然后把结果返回给链码容器。
3. 链码容器收到结果后，再返回到链码的逻辑中。



当链码在一次 `Query/Invoke` 中调用了多次账本的查询或写入接口时，会产生一定的网络通信成本和延迟，这对网络的整体吞吐率会有一定的影响。在设计应用时，应尽量减少一次 `Query/Invoke` 中的查询和写入账本的次数。在一些对吞吐有很高要求的特殊场景下，可以在业务层对多个 Key 及对应的 Value 进行合并，将多次读写操作变成一次操作。

3 减少链码的运算量

当链码被调用时，会在 peer 的账本上挂一把读锁，保证链码在处理该笔交易时，账本的状态不发生改变，当新的区块产生时，peer 将账本完全锁住，直到完成账本状态的更新操作。

如果链码在处理交易时花费了大量时间，会让 peer 验证区块等待更长的时间，从而降低整体的吞吐量。

在编写链码时，链码中最好只包含简单的逻辑、校验等必要的运算，将不太重要的逻辑放到链码外进行。

四、Java SDK 的优化

1 复用channel && client 对象

SDK 在初始化 channel 对象阶段会有一定的资源及时间消耗，同时每一个 channel 对象都会建立自己的事件监听连接，向 peer 获取最新的区块及验证结果，从而消耗较多的网络带宽。

```
private X509Certificate certificate;
private PrivateKey privateKey;
private Wallet wallet;
private Gateway gateway;
private Gateway.Builder builder;
private Network network;
private Channel channel;
private Contract contract;
private Collection<Peer> peerSet;

public FirstSampleHandler() {
    try {
        this.certificate = readX509Certificate(credentialPath.resolve(Paths.get( first: "signcerts",  ...more: "Admin@org1.example.com-cert
        this.privateKey = getPrivateKey(credentialPath.resolve(Paths.get( first: "keystore",  ...more: "priv_sk")));

        // 加载一个钱包，里面有接入网络所需要的identities
        this.wallet = Wallets.newInMemoryWallet();
        // Path walletDir = Paths.get("wallet");
        // Wallet wallet = Wallets.newFileSystemWallet(walletDir);
        this.wallet.put( label: "user", Identities.newX509Identity( mspId: "Org1MSP", certificate, privateKey));
    } catch (Exception e) {
        System.out.println("读证书错误");
        e.printStackTrace();
    }

    try {
        // 设置连接网络所需要的gateway connection配置信息
        this.builder = Gateway.createBuilder()
                .identity(this.wallet,  id: "user")
                .networkConfig(NETWORK_CONFIG_PATH);

        // 创建Gateway连接
        this.gateway = builder.connect();
        // 接入channel
        this.network = gateway.getNetwork( networkName: "mychannel");

        this.channel = network.getChannel();
        this.contract = network.getContract( chaincodeId: "mycc");
        this.peerSet = channel.getPeers();
    } catch (Exception e) {
        System.out.println("连接错误");
        e.printStackTrace();
```

1. 应用程序在针对一个业务通道进行操作的时候，如果创建过多 channel 对象，可能会影响业务的响应时间，甚至会由于 TCP 连接数过多而引发业务阻塞。
2. 在应用程序中，如果针对一个业务通道频繁发送交易，则创建该通道的第一个 channel 对象后应尽量复用。
3. 如果 channel 对象长时间闲置，可以使用 `channel.shutdown(true)` 释放资源。
4. 通过 HFCAClient 产生本地用户时，其中包含了用户私钥的生成和 `Enroll` 操作，也有一定的时间消耗。

```java
@Override
public void setUserContext(final HFClient client, final Identity identity, final String name) {
    X509Identity x509Identity = (X509Identity) identity;

    String certificatePem = Identities.toPemString(x509Identity.getCertificate());
    Enrollment enrollment = new X509Enrollment(x509Identity.getPrivateKey(), certificatePem);
    User user = new GatewayUser(name, x509Identity.getMspId(), enrollment);

    try {
        CryptoSuite cryptoSuite = CryptoSuiteFactory.getDefault().getCryptoSuite();
        client.setCryptoSuite(cryptoSuite);
        client.setUserContext(user);
    } catch (ClassNotFoundException | CryptoException | IllegalAccessException | NoSuchMethodException
            | InstantiationException | InvalidArgumentException | InvocationTargetException e) {
        throw new GatewayRuntimeException("Failed to configure user context", e);
    }
}
```

2 只将交易发送给必要的背书节点（负载均衡）

假设每个组织都会有2个 peer 背书节点，如果一个业务通道内有 N 个组织，在使用 SDK 提交 Proposal 的时候，会默认发送给所有的 peer 背书节点（2*N个）。

- 这时每个 peer 节点都要处理一遍Proposal，影响整体的吞吐量。
- 当个别peer处理缓慢时，会拖慢交易的响应时间。

```java
@Override
public ProposalResponse evaluate(final Query query) throws ContractException {
    int startPeerIndex = currentPeerIndex.getAndUpdate(i -> (i + 1) % peers.size());
    Collection<ProposalResponse> failResponses = new ArrayList<>();

    for (int i = 0; i < peers.size(); i++) {
        int peerIndex = (startPeerIndex + i) % peers.size();
        Peer peer = peers.get(peerIndex);
        ProposalResponse response = query.evaluate(peer);
        if (response.getStatus().equals(ChaincodeResponse.Status.SUCCESS)) {
            return response;
        }
        if (response.getProposalResponse() != null) {
            throw new ContractException(response.getMessage(), Collections.singletonList(response));
        }
        failResponses.add(response);
    }

    String message = "No responses received. Errors: " + failResponses.stream()
            .map(ProposalResponse::getMessage)
            .collect(Collectors.joining( delimiter: "; "));
    throw new ContractException(message, failResponses);
}
```

解决：

只把请求发送给自己信赖的peer节点，或者是承担负载均很任务的节点

```
public static void queryFromPeer(Network network) {
    try {
        GatewayImpl gateway = (GatewayImpl) network.getGateway();
        Channel channel = network.getChannel();

        QueryByChaincodeRequest queryByChaincodeRequest = gateway.getClient().newQueryProposalRequest();
        queryByChaincodeRequest.setChaincodeName("mycc");
        queryByChaincodeRequest.setFcn("query");
        queryByChaincodeRequest.setArgs("a");

        Collection<Peer> peerSet = channel.getPeers();
        Collection<Peer> endorserSet = new LinkedList<>();
        for (Peer peer : peerSet) {
            if(peer.getName().equals("peer0.org1.example.com")) {
                endorserSet.add(peer);
            }
        }

        // 方便查看容器log，多查几次
        for (int i = 0; i < 10; i++) {
            Collection<ProposalResponse> proposalResponses = channel.queryByChaincode(queryByChaincodeRequest, endorserSet);
            for (ProposalResponse prores: proposalResponses) {
                String result = prores.getProposalResponse().getResponse().getPayload().toStringUtf8();
                System.out.printf("Result from %s: %s\n", prores.getPeer().getName(), result);
            }
        }
    } catch (Exception e) {
        System.out.println("QueryLedger Error");
        e.printStackTrace();
    }
}
```