PaxosStore 源码分析「三、共识协议」

2020.02.17 SF-Zhou

本系列的前两篇分析了 PaxosStore 中网络通信和消息传递的实现，本篇将正式介绍 Paxos 算法，并分析 PaxosStore 中共识协议的实现。

1. Paxos 算法

Paxos 算法是 Leslie Lamport 于 1990 年提出的一种基于消息传递且具有高度容错特性的共识（consensus）算法。其论文于 1998 年 TOCS 会议上首次公开发表，中间的八年显然是有故事的。Paxos 算法已经问世 30 年了，至今依然折磨着学习分布式系统的同学们。入门学习的话，建议阅读作者 2001 年重新描述的论文 "Paxos Made Simple"。下面笔者说说自己对 Paxos 的理解。

达成共识 from "黑金"

首先 Paxos 是一个共识算法，即最终目标是达成共识，至于共识是什么、对不对，在这里并不重要，重要的是达成共识后，共识不可修改；其次，一个经典 Paxos 实例只能达

成一个共识，或者说确定（chosen）一个值，这一点很重要；最后，算法执行的环境是异步通信环境，使用非拜占庭模型，允许消息延迟、丢失、乱序，但不允许数据损坏（corruption）。

Paxos 算法中一共有三种角色：proposers，acceptors 和 learners，这里先忽略 learners。算法的步骤描述如下（摘录自论文原文 2.2 节）：

Phase 1.

1. A proposer selects a proposal number n and sends a prepare request with number n to a majority of acceptors.
2. If an acceptor receives a prepare request with number n greater than that of any prepare request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal (if any) that it has accepted.

Phase 2.

1. If the proposer receives a response to its prepare requests (numbered n) from a majority of acceptors, then it sends an accept request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.

2. If an acceptor receives an accept request for a proposal numbered n, it accepts the proposal unless it has already responded to a prepare request having a number greater than n.

因为消息本身的传递是不可靠的，所以可以从各个角色的响应来思考 Paxos 的流程，这会比面向过程的思考容易些。可以认为每个角色都是一个独立的线程，当收到特定的消息时做出对应的响应。这里先定义下消息类和角色自身的状态类：

```
struct Prepare {
  int n;
};

struct Accept {
  int n;
  void *value;
};

struct Promised {
  int n;
  Accept *proposal = nullptr;
};

struct Accepted {
  int n;
};

class Acceptor {
  int last_n = 0;
```

```
  Accept *proposal = nullptr;
};
```

对于 Acceptor，其会对两种请求做出响应：

1. Prepare： 如果 prepare.n > acceptor.last_n， 则更新 acceptor.last_n， 并返回 Promised(n, acceptor.proposal)；
2. Accept： 如果 accept.n >= acceptor.last_n， 则接受该提案，更新 acceptor.last_n 和 acceptor.proposal 并返回 Accepted(n)。

对于 Proposer，除主动发送 Prepare 请求外，同时会接收两种响应：

1. Promised： 当 Promised 数量足够组成多数派时，进入算法的 Phase 2、广播 Accept 请求；
2. Accepted： 当 Accepted 数量足够组成多数派时，Proposer 确认提议通过，反之 Proposer 不确定，但提案可能还是通过的。

Paxos 算法里，对确定（chosen）的理解至关重要。当多数派通过提案的一瞬间，共识即已达成且不可推翻，理解了这一点就理解了 Paxos。

对于单个 Acceptor，个体的 Accepted 和多数派的 Chosen 是有本质区别的，个体的 Accepted 是可以被覆盖的。而当 Chosen 发生后，之后的提案返回的 Promised 中得到的多数派里一定会返回群体 Chosen 的值，这保证了提案的值不会被推翻。关于这点，可以参考文献 2 中的例子。

Paxos 算法的证明网上可以找到很多，然后 MIT 6.824 2015 年前的课程里有对应的实验，参见文献 3。对应的代码笔者自己找了一份，放在 GitHub 上，可以自己完成以加深理解。2016 之后他们改用 Raft 了，有兴趣的话可以继续完成 Raft 的学习。

2. 协议实现

PaxosStore 的实现中弃用了原版的 Paxos 消息协议，并提出了使用**半对称消息**传递来实现 Paxos 过程，这里参考其论文中的描述。首先定义提案 Proposal：

$$\mathcal{P} = (n, v)$$

其中 n 表示提案号，v 表示提案的值。定义状态 State：

$$\mathcal{S} = (m, \mathcal{P})$$

其中 m 表示承诺过不会拒绝的最小提案号，$\mathcal{P}$ 表示已经接受的提案。定义 $\mathcal{S}_X$ 为机器 X 的状态，定义 $\mathcal{S}_Y^X$ 为机器 X 已知的机器 Y 的状态，称之为视图状态。显然 $\mathcal{S}_Y^X (X \neq Y)$ 与 $\mathcal{S}_Y$ 可能是不同的，不同机器上的状态依赖 Paxos 过程完成同步。定义机器 X 发送给机器 Y 的消息 Message：

$$\mathbb{M}_{X \to Y} = \left\{ \mathcal{S}_X^X, \mathcal{S}_Y^X \right\}$$

简单来说，该消息中包含了 X 自己的实际状态，以及自己已知的对方的视图状态。PaxosStore 全局仅使用这一种消息来实现 Paxos 协议，具体的协议过程如下（摘录自原论

文）：

Algorithm 1: Paxos implementation in PaxosStore.

Input: intention proposal number m_i

Procedure `Issue`(m_i): /* invoked at N_A */

  $\mathcal{S}_A^A \leftarrow$ actual state of r_A

  if $\mathcal{S}_A^A.m < m_i$ **then**

    $\mathcal{S}_A^A.m \leftarrow m_i$

    write $\mathcal{S}_A^A$ to the PaxosLog entry of r_A

    foreach remote replica node N_X **do**

      send $\mathbb{M}_{A \rightarrow X}$

Input: proposal $\mathcal{P}_i$ with $\mathcal{P}_i.n = m_i$

Procedure `Issue`($\mathcal{P}_i$): /* invoked at N_A */

  $\mathcal{S}_A^A \leftarrow$ actual state of r_A

  $\mathbb{S}^A \leftarrow$ all the states of r maintained at N_A

  if $\left|\left\{\forall \mathcal{S}_X^A \in \mathbb{S}^A \mid \mathcal{S}_X^A.m = \mathcal{P}_i.n\right\}\right| \times 2 > \left|\mathbb{S}^A\right|$ **then**

    if $\left|\left\{\forall \mathcal{S}_X^A \in \mathbb{S}^A \mid \mathcal{S}_X^A.\mathcal{P}.v \neq null\right\}\right| > 0$ **then**

      $\mathcal{P}' \leftarrow$ the proposal with maximum $\mathcal{P}.n$ in $\mathbb{S}^A$

      $\mathcal{S}_A^A.\mathcal{P} \leftarrow (\mathcal{P}_i.n,\ \mathcal{P}'.v)$

    else

      $\mathcal{S}_A^A.\mathcal{P} \leftarrow \mathcal{P}_i$

    write $\mathcal{S}_A^A$ to the PaxosLog entry of r_A

    foreach remote replica node N_X **do**

      send $\mathbb{M}_{A \rightarrow X}$

Input: message $\mathbb{M}_{X \to Y}$ sent from N_X to N_Y

Procedure `OnMessage(`$\mathbb{M}_{X \to Y}$`):` /* invoked at N_Y */

    $\mathcal{S}_X^X,\ \mathcal{S}_Y^X \leftarrow \mathbb{M}_{X \to Y}$

    `UpdateStates(`$Y,\ \mathcal{S}_X^X$`)`

    if $\mathcal{S}_Y^Y$ is changed **then**

        write $\mathcal{S}_Y^Y$ to the PaxosLog entry of r_Y

        if `IsValueChosen(`Y`)` is *true* **then** commit

    if $\mathcal{S}_Y^X.m < \mathcal{S}_Y^Y.m$ or $\mathcal{S}_Y^X.\mathcal{P}.n < \mathcal{S}_Y^Y.\mathcal{P}.n$ **then**

        send $\mathbb{M}_{Y \to X}$

Input: node ID Y, actual state $\mathcal{S}_X^X$ of r_X

Function `UpdateStates(`$Y,\ \mathcal{S}_X^X$`):` /* invoked at N_Y */

    $\mathcal{S}_X^Y \leftarrow$ view state of r_X stored in N_Y

    $\mathcal{S}_Y^Y \leftarrow$ actual state of r_Y

    if $\mathcal{S}_X^Y.m < \mathcal{S}_X^X.m$ **then** $\mathcal{S}_X^Y.m \leftarrow \mathcal{S}_X^X.m$

    if $\mathcal{S}_X^Y.\mathcal{P}.n < \mathcal{S}_X^X.\mathcal{P}.n$ **then** $\mathcal{S}_X^Y.\mathcal{P} \leftarrow \mathcal{S}_X^X.\mathcal{P}$

    if $\mathcal{S}_Y^Y.m < \mathcal{S}_X^X.m$ **then** $\mathcal{S}_Y^Y.m \leftarrow \mathcal{S}_X^X.m$

    if $\mathcal{S}_Y^Y.m \leq \mathcal{S}_X^X.\mathcal{P}.n$ **then** $\mathcal{S}_Y^Y.\mathcal{P} \leftarrow \mathcal{S}_X^X.\mathcal{P}$

Input: node ID Y

Output: whether the proposals in N_Y form a majority

Function `IsValueChosen(`Y`):` /* invoked at N_Y */

    $\mathbb{S}^Y \leftarrow$ all the states of r maintained at N_Y

    $n' \leftarrow$ occurrence count of the most frequent $\mathcal{P}.n$ in $\mathbb{S}^Y$

    return $n' \times 2 > |\mathbb{S}^Y|$

PaxosStore 中的 Paxos 实现 from 参考文献 4

当节点 A 收到一个写请求时，其会触发 $\textbf{Issue}(m_i)$，发送 Prepare 消息给所有节点；节点 X 收到消息后触发 $\textbf{OnMessage}(\mathbb{M}_{A\to X})$，更新自身的状态，若自身状态更新则会发送消息回节点 A；节点 A 收到回复的消息同样会执行 $\textbf{OnMessage}(\mathbb{M}_{X\to A})$，判断 Preprare 是否被多数派接受，若是则触发 $\textbf{Issue}(\mathcal{P}_i)$ 发送 Accept 消息给所有节点；而后是一轮类似的消息处理，节点 X 收到消息并更新，回复消息，节点 A 收到消息判断 $\textbf{IsValueChosen}$。

可以看到，PaxosStore 使用了统一的消息格式，同时使用了统一的消息处理函数 $\textbf{OnMessage}$，以此驱动整个 Paxos 协议过程。这大大简化了代码的实现，按照论文描述的，核心算法实现只使用了约 800 行 C++ 代码。这种实现的正确性已经在微信的大规模应用上得到验证，另外也通过了基于 TLA+ 的形式化规约和验证，见参考文献 5。

3. 代码实现

PaxosStore 中基本是按照论文中描述的方案来实现的。首先状态 $\mathcal{S}$ 对应 src/Common.h 中的 EntryRecord_t：

```
struct PaxosValue_t {
  uint64_t iValueID;
  vector<uint64_t> vecValueUUID;

  bool bHasValue;
  string strValue;
```

```
  PaxosValue_t() : iValueID(0), bHasValue(false) {}
  PaxosValue_t(uint64_t iValueID_, const vector<uint64_t> vecValueUUID_, bool bHasValue_,
               const string &strValue_)
      : iValueID(iValueID_),
        vecValueUUID(vecValueUUID_),
        bHasValue(bHasValue_),
        strValue(strValue_) {}

  bool operator==(const PaxosValue_t &tOther) const {
    if (iValueID == tOther.iValueID && vecValueUUID == tOther.vecValueUUID &&
        bHasValue == tOther.bHasValue && strValue == tOther.strValue) {
      return true;
    }

    return false;
  }
};

struct EntryRecord_t {
  uint32_t iPreparedNum;
  uint32_t iPromisedNum;
  uint32_t iAcceptedNum;

  PaxosValue_t tValue;
  bool bChosen;

  bool bCheckedEmpty;  // For Read Opt.

  uint64_t iStoredValueID;  // For PutValue Opt.
```

```
  bool operator==(const EntryRecord_t &tOther) const {
    if (iPreparedNum == tOther.iPreparedNum && iPromisedNum == tOther.iPromisedNum &&
        iAcceptedNum == tOther.iAcceptedNum && tValue == tOther.tValue &&
        bChosen == tOther.bChosen) {
      return true;
    }

    return false;
  }
};
```

$\mathcal{S}.m$ 对应 iPromisedNum，$\mathcal{S}.\mathcal{P}.n$ 对应 iAcceptedNum，$\mathcal{S}.\mathcal{P}.v$ 对应 tValue。每个节点会存储自己的状态，以及其他节点的视图状态，这些状态被存储于 clsEntryStateMachine 中，定义于 src/EntryState.h：

```
enum enumEntryState {
  kEntryStateNormal = 0,
  kEntryStatePromiseLocal,
  kEntryStatePromiseRemote,
  kEntryStateMajorityPromise,
  kEntryStateAcceptRemote,
  kEntryStateAcceptLocal,
  kEntryStateChosen
};

class clsEntryStateMachine {
 public:
```

```
 static uint32_t s_iAcceptorNum;
 static uint32_t s_iMajorityNum;
 static uint32_t GetAcceptorID(uint64_t iValueID);

private:
 int m_iEntryState;

 uint32_t m_iMaxPreparedNum;

 uint32_t m_iMostAcceptedNum;
 uint32_t m_iMostAcceptedNumCnt;

 std::vector<EntryRecord_t> m_atRecord;

 uint32_t CountAcceptedNum(uint32_t iAcceptedNum);
 uint32_t CountPromisedNum(uint32_t iPromisedNum);

 int CalcEntryState(uint32_t iLocalAcceptorID);

 int MakeRealRecord(EntryRecord_t &tRecord);

 void UpdateMostAcceptedNum(const EntryRecord_t &tRecord);
 bool GetValueByAcceptedNum(uint32_t iAcceptedNum, PaxosValue_t &tValue);

public:
 static int Init(clsConfigure *poConf);

 clsEntryStateMachine() {
   m_iEntryState = kEntryStateNormal;
```

```
  m_iMaxPreparedNum = 0;

  m_iMostAcceptedNum = 0;
  m_iMostAcceptedNumCnt = 0;

  m_atRecord.resize(s_iAcceptorNum);
  for (uint32_t i = 0; i < s_iAcceptorNum; ++i) {
    InitEntryRecord(&m_atRecord[i]);
  }
}

~clsEntryStateMachine() {}

int GetEntryState() { return m_iEntryState; }

uint32_t GetNextPreparedNum(uint32_t iLocalAcceptorID);

const EntryRecord_t &GetRecord(uint32_t iAcceptorID);

int Update(uint64_t iEntityID, uint64_t iEntry, uint32_t iLocalAcceptorID, uint32_t iAc
           const EntryRecord_t &tRecordMayWithValueIDOnly);

int AcceptOnMajorityPromise(uint32_t iLocalAcceptorID, const PaxosValue_t &tValue,
                            bool &bAcceptPreparedValue);

void SetStoredValueID(uint32_t iLocalAcceptorID);

// For readonly cmd.
void ResetAllCheckedEmpty();
void SetCheckedEmpty(uint32_t iAcceptorID);
```

```
  bool IsLocalEmpty();
  bool IsReadOK();

  bool IsRemoteCompeting();

  string ToString();

  uint32_t CalcSize();
};
```

m_atRecord 中存储了自身的状态和其他节点的视图状态，可以通过 iAcceptorID 访问。整个 clsEntryStateMachine 构成一个状态机，接收到消息时通过执行 Update （对应算法描述中的 **UpdateStates**）更新自身的实际状态和其他节点的视图状态，进而推进 Paxos 的流程。当前 Paxos 执行的阶段使用 m_iEntryState 表示，初始化阶段是 kEntryStateNormal， 最终 Chosen 的阶段是 kEntryStateChosen 。下面是状态机代码的实现：

```
#include "EntryState.h"

namespace Certain {

uint32_t clsEntryStateMachine::s_iAcceptorNum = 0;
uint32_t clsEntryStateMachine::s_iMajorityNum = 0;

// 从 ValueID 中解析 AcceptorID
uint32_t clsEntryStateMachine::GetAcceptorID(uint64_t iValueID) {
  uint32_t iProposalNum = iValueID & 0xffffffff;
```

```
  AssertLess(0, iProposalNum);
  return (iProposalNum - 1) % s_iAcceptorNum;
}

// 初始化，设定 Acceptor 和 Majority 的数量
int clsEntryStateMachine::Init(clsConfigure *poConf) {
  s_iAcceptorNum = poConf->GetAcceptorNum();
  s_iMajorityNum = (s_iAcceptorNum >> 1) + 1;

  return 0;
}

// 获取 Acceptor[i].EntryRecord
const EntryRecord_t &clsEntryStateMachine::GetRecord(uint32_t iAcceptorID) {
  return m_atRecord[iAcceptorID];
}

// 获取下一个 PreparedNum
uint32_t clsEntryStateMachine::GetNextPreparedNum(uint32_t iLocalAcceptorID) {
  if (m_iMaxPreparedNum == 0) {
    m_iMaxPreparedNum = iLocalAcceptorID + 1;
  } else {
    m_iMaxPreparedNum += s_iAcceptorNum;
  }

  return m_iMaxPreparedNum;
}

// 清空所有 CheckedEmpty 状态
void clsEntryStateMachine::ResetAllCheckedEmpty() {
```

```
  for (uint32_t i = 0; i < s_iAcceptorNum; ++i) {
    m_atRecord[i].bCheckedEmpty = false;
  }
}

// 设定 CheckedEmpty 状态
void clsEntryStateMachine::SetCheckedEmpty(uint32_t iAcceptorID) {
  m_atRecord[iAcceptorID].bCheckedEmpty = true;
}

// 检查本地是否是 Normal 状态
bool clsEntryStateMachine::IsLocalEmpty() {
  // (TODO)rock: use tla to check
  return m_iEntryState == kEntryStateNormal;
}

// 设定本地 Record 的 ValueID
void clsEntryStateMachine::SetStoredValueID(uint32_t iLocalAcceptorID) {
  EntryRecord_t &tLocalRecord = m_atRecord[iLocalAcceptorID];
  if (tLocalRecord.tValue.iValueID > 0) {
    tLocalRecord.iStoredValueID = tLocalRecord.tValue.iValueID;
  }
}

bool clsEntryStateMachine::IsReadOK() {
  uint32_t iCount = 0;

  for (uint32_t i = 0; i < s_iAcceptorNum; ++i) {
    if (m_atRecord[i].bCheckedEmpty && m_atRecord[i].iPromisedNum == 0) {
      iCount++;
```

```
    }
  }

  CertainLogDebug("iCount %u", iCount);
  return iCount >= s_iMajorityNum;
}

// 统计 Accepted 数量
uint32_t clsEntryStateMachine::CountAcceptedNum(uint32_t iAcceptedNum) {
  uint32_t iCount = 0;

  for (uint32_t i = 0; i < s_iAcceptorNum; ++i) {
    if (m_atRecord[i].iAcceptedNum == iAcceptedNum) {
      iCount++;
    }
  }
  return iCount;
}

// 统计 PromisedNum 数量
uint32_t clsEntryStateMachine::CountPromisedNum(uint32_t iPromisedNum) {
  uint32_t iCount = 0;

  for (uint32_t i = 0; i < s_iAcceptorNum; ++i) {
    if (m_atRecord[i].iPromisedNum == iPromisedNum) {
      iCount++;
    }
  }
  return iCount;
}
```

```
// 根据 AcceptedNum 获取 Value
bool clsEntryStateMachine::GetValueByAcceptedNum(uint32_t iAcceptedNum, PaxosValue_t &tVa
  for (uint32_t i = 0; i < s_iAcceptorNum; ++i) {
    if (m_atRecord[i].iAcceptedNum == iAcceptedNum) {
      tValue = m_atRecord[i].tValue;
      return true;
    }
  }

  return false;
}

// 更新最多 Accepted 的 Num
void clsEntryStateMachine::UpdateMostAcceptedNum(const EntryRecord_t &tRecord) {
  if (m_iMostAcceptedNum == tRecord.iAcceptedNum) {
    m_iMostAcceptedNumCnt++;
    return;
  }

  uint32_t iCount = CountAcceptedNum(tRecord.iAcceptedNum);

  if (m_iMostAcceptedNumCnt < iCount) {
    m_iMostAcceptedNum = tRecord.iAcceptedNum;
    m_iMostAcceptedNumCnt = iCount;
  }
}

// 计算当前的状态
int clsEntryStateMachine::CalcEntryState(uint32_t iLocalAcceptorID) {
```

```
EntryRecord_t &tLocalRecord = m_atRecord[iLocalAcceptorID];

m_iEntryState = kEntryStateNormal;

if (tLocalRecord.bChosen) {
  m_iEntryState = kEntryStateChosen;
  return 0;
}

if (m_iMostAcceptedNumCnt >= s_iMajorityNum) {
  m_iEntryState = kEntryStateChosen;

  if (tLocalRecord.iAcceptedNum != m_iMostAcceptedNum) {
    if (!GetValueByAcceptedNum(m_iMostAcceptedNum, tLocalRecord.tValue)) {
      return -1;
    }
    tLocalRecord.iAcceptedNum = m_iMostAcceptedNum;
  }

  tLocalRecord.bChosen = true;
  return 0;
}

if (tLocalRecord.iPromisedNum > tLocalRecord.iPreparedNum) {
  m_iEntryState = kEntryStatePromiseRemote;
  if (tLocalRecord.iAcceptedNum >= tLocalRecord.iPromisedNum) {
    m_iEntryState = kEntryStateAcceptRemote;
  }
  return 0;
}
```

```
  if (tLocalRecord.iPromisedNum != tLocalRecord.iPreparedNum) {
    return -2;
  }

  // iPromisedNum == 0 means null.
  if (tLocalRecord.iPromisedNum > 0) {
    m_iEntryState = kEntryStatePromiseLocal;

    uint32_t iLocalPromisedNum = tLocalRecord.iPromisedNum;
    uint32_t iPromisedNumCnt = CountPromisedNum(iLocalPromisedNum);

    if (iPromisedNumCnt >= s_iMajorityNum) {
      m_iEntryState = kEntryStateMajorityPromise;
    }

    if (tLocalRecord.iAcceptedNum == tLocalRecord.iPromisedNum) {
      m_iEntryState = kEntryStateAcceptLocal;
    }
  }

  if (tLocalRecord.iAcceptedNum > tLocalRecord.iPromisedNum) {
    m_iEntryState = kEntryStateAcceptRemote;
  }

  return 0;
}

// 通过 ValueID 获取 tRecord.tValue
int clsEntryStateMachine::MakeRealRecord(EntryRecord_t &tRecord) {
```

```
  for (uint32_t i = 0; i < s_iAcceptorNum; ++i) {
    EntryRecord_t &tRealRecord = m_atRecord[i];

    if (tRecord.tValue.iValueID != tRealRecord.tValue.iValueID) {
      continue;
    }

#if CERTAIN_DEBUG
    // For check only.
    if (tRecord.tValue.strValue.size() > 0) {
      if (tRecord.tValue.strValue != tRealRecord.tValue.strValue ||
          (tRecord.tValue.vecValueUUID.size() > 0 && tRealRecord.tValue.vecValueUUID.size
           tRecord.tValue.vecValueUUID != tRealRecord.tValue.vecValueUUID)) {
        CertainLogFatal("CRC32(%u, %u) BUG record: %s lrecord[%u]: %s",
                        CRC32(tRecord.tValue.strValue), CRC32(tRealRecord.tValue.strValue
                        EntryRecordToString(tRecord).c_str(), i,
                        EntryRecordToString(tRealRecord).c_str());

        Assert(false);
      }
    }
#endif
    tRecord.tValue = tRealRecord.tValue;
    break;
  }

  if (tRecord.iAcceptedNum > 0) {
    if (tRecord.tValue.iValueID > 0) {
      return 0;
    }
```

```
    return -1;
  }

  return 1;
}

// 转为字符串
string clsEntryStateMachine::ToString() {
  string strState;
  for (uint32_t i = 0; i < s_iAcceptorNum; ++i) {
    if (i > 0) {
      strState += " ";
    }
    EntryRecord_t &tRecord = m_atRecord[i];
    strState += EntryRecordToString(tRecord);
  }
  return strState;
}

int clsEntryStateMachine::Update(uint64_t iEntityID, uint64_t iEntry, uint32_t iLocalAcce
                                 uint32_t iAcceptorID,
                                 const EntryRecord_t &tRecordMayWithValueIDOnly) {
#if CERTAIN_DEBUG
  RETURN_RANDOM_ERROR_WHEN_IN_DEBUG_MODE();
#endif

  int iRet;

  if (m_iEntryState == kEntryStateChosen) {
```

```
    return -1;
  }

  if (iLocalAcceptorID >= s_iAcceptorNum || iAcceptorID >= s_iAcceptorNum) {
    return -2;
  }

  m_iEntryState = kEntryStateNormal;
  EntryRecord_t &tLocalRecord = m_atRecord[iLocalAcceptorID];
  EntryRecord_t &tRemoteRecord = m_atRecord[iAcceptorID];

  // Make record into real when it comes in machine state.
  EntryRecord_t tRecord = tRecordMayWithValueIDOnly;
  iRet = CheckEntryRecordMayWithVIDOnly(tRecord);
  if (iRet != 0) {
    CertainLogFatal("CheckEntryRecordMayWithVIDOnly ret %d", iRet);
    return -3;
  }

  iRet = MakeRealRecord(tRecord);
  if (iRet < 0) {
    CertainLogFatal("MakeRealRecord ret %d", iRet);
    return -4;
  }

  if (!tRecord.tValue.bHasValue && tRecord.tValue.iValueID > 0) {
    if (tLocalRecord.iAcceptedNum > tRecord.iAcceptedNum) {
      // The remote acceptor supposed that this acceptor know the V.
      // But the V stored in tLocalRecord has been overriden,
```

```
      // Ignore the accepted message.
      tRecord.iAcceptedNum = 0;
      tRecord.tValue.iValueID = 0;
    } else {
      return -5;
    }
  }

  iRet = CheckEntryRecord(tRecord);
  if (iRet != 0) {
    CertainLogFatal("CheckEntryRecord ret %d", iRet);
    return -6;
  }

  if (tRecord.bChosen) {
    // For check only.
    if (tRemoteRecord.iAcceptedNum >= tRecord.iAcceptedNum) {
      if (tRemoteRecord.tValue.iValueID != tRecord.tValue.iValueID) {
        return -7;
      }
    }

    tLocalRecord.iAcceptedNum = tRecord.iAcceptedNum;
    tLocalRecord.tValue = tRecord.tValue;
    tLocalRecord.bChosen = true;

    tRemoteRecord.iAcceptedNum = tRecord.iAcceptedNum;
    tRemoteRecord.tValue = tRecord.tValue;
    tRemoteRecord.bChosen = true;
```

```
    m_iEntryState = kEntryStateChosen;

    return m_iEntryState;
  }

  // 1. update m_iMaxPreparedNum
  uint32_t iGlobalMaxPreparedNum = max(tRecord.iPreparedNum, tRecord.iPromisedNum);
  if (m_iMaxPreparedNum < iGlobalMaxPreparedNum) {
    uint32_t iNextPreparedNum = m_iMaxPreparedNum;

    while (iNextPreparedNum <= iGlobalMaxPreparedNum) {
      m_iMaxPreparedNum = iNextPreparedNum;

      if (iNextPreparedNum == 0) {
        iNextPreparedNum = iLocalAcceptorID + 1;
      } else {
        iNextPreparedNum += s_iAcceptorNum;
      }
    }
  }

  // 2. update iPreparedNum
  if (tRemoteRecord.iPreparedNum < tRecord.iPreparedNum) {
    tRemoteRecord.iPreparedNum = tRecord.iPreparedNum;
  }

  // 3. update old remote record
  if (iAcceptorID != iLocalAcceptorID && tRecord.iAcceptedNum > tRemoteRecord.iAcceptedNu
    tRemoteRecord.iAcceptedNum = tRecord.iAcceptedNum;
    if (tRemoteRecord.tValue.iValueID != tRecord.tValue.iValueID) {
```

```
      tRemoteRecord.tValue = tRecord.tValue;
    }

    UpdateMostAcceptedNum(tRecord);
  }

  // 4. update value for remote accept first when use PreAuth
  if (tRecord.iAcceptedNum == 0) {
    if (tRecord.tValue.iValueID > 0) {
      if (0 == tRecord.iPromisedNum) {
        return -8;
      }

      if (iLocalAcceptorID == iAcceptorID) {
        if (tLocalRecord.iAcceptedNum != 0 || tLocalRecord.tValue.iValueID != 0) {
          return -9;
        }
        tLocalRecord.tValue = tRecord.tValue;
      } else if (tRecord.iPromisedNum <= s_iAcceptorNum) {
        if (tRecord.iPromisedNum == 0 || tRecord.iPreparedNum != tRecord.iPromisedNum) {
          return -10;
        }
        tRecord.iAcceptedNum = tRecord.iPromisedNum;
      }
    }

    // store the newest status of remote record
    if (iLocalAcceptorID != iAcceptorID && tRemoteRecord.iAcceptedNum == 0) {
      if (tRemoteRecord.tValue.iValueID > 0 && tRecord.tValue.iValueID > 0 &&
          (tRemoteRecord.tValue.iValueID != tRecord.tValue.iValueID ||
```

```
            tRemoteRecord.tValue.strValue != tRecord.tValue.strValue)) {
          return -11;
        }
        tRemoteRecord.tValue = tRecord.tValue;
      }
    }

    // 5. update old local record
    if (tRecord.iAcceptedNum > tLocalRecord.iAcceptedNum &&
        tRecord.iAcceptedNum >= tLocalRecord.iPromisedNum) {
      tLocalRecord.iAcceptedNum = tRecord.iAcceptedNum;
      if (tLocalRecord.tValue.iValueID != tRecord.tValue.iValueID) {
        tLocalRecord.tValue = tRecord.tValue;
      }

      UpdateMostAcceptedNum(tRecord);
    }

    // 6. update iPromisedNum
    if (tRemoteRecord.iPromisedNum < tRecord.iPromisedNum) {
      tRemoteRecord.iPromisedNum = tRecord.iPromisedNum;
    }
    if (tLocalRecord.iPromisedNum < tRecord.iPromisedNum) {
      tLocalRecord.iPromisedNum = tRecord.iPromisedNum;
    }

    iRet = CalcEntryState(iLocalAcceptorID);
    if (iRet < 0) {
      CertainLogFatal("CalcEntryState ret %d", iRet);
      return -12;
```

```
  }

  // For check only.
  iRet = CheckEntryRecord(tRecord);
  if (iRet != 0) {
    CertainLogFatal("CheckEntryRecord ret %d", iRet);
    return -13;
  }

  if (tLocalRecord.iPreparedNum > 0) {
    if ((tLocalRecord.iPreparedNum - 1) % s_iAcceptorNum != iLocalAcceptorID) {
      return -14;
    }
  }
  if (tLocalRecord.iPromisedNum > tLocalRecord.iPreparedNum) {
    if ((tLocalRecord.iPromisedNum - 1) % s_iAcceptorNum == iLocalAcceptorID) {
      return -15;
    }
  }

  if (tLocalRecord.iPromisedNum == iLocalAcceptorID + 1 && tLocalRecord.iAcceptedNum == 0
      tLocalRecord.tValue.iValueID == 0) {
    return -16;
  }

  return m_iEntryState;
}

int clsEntryStateMachine::AcceptOnMajorityPromise(uint32_t iLocalAcceptorID,
                                                  const PaxosValue_t &tValue,
```

```
                                        bool &bAcceptPreparedValue) {
#if CERTAIN_DEBUG
  RETURN_RANDOM_ERROR_WHEN_IN_DEBUG_MODE();
#endif
  int iRet;

  bAcceptPreparedValue = false;
  if (m_iEntryState != kEntryStateMajorityPromise) {
    return -1;
  }
  EntryRecord_t &tLocalRecord = m_atRecord[iLocalAcceptorID];
  if (tLocalRecord.iPreparedNum != tLocalRecord.iPromisedNum ||
      tLocalRecord.iAcceptedNum >= tLocalRecord.iPromisedNum) {
    return -2;
  }

  // 取得最大的 AcceptedNum 及其对应的 Value
  for (uint32_t i = 0; i < s_iAcceptorNum; ++i) {
    if (tLocalRecord.iAcceptedNum < m_atRecord[i].iAcceptedNum) {
      tLocalRecord.iAcceptedNum = m_atRecord[i].iAcceptedNum;
      tLocalRecord.tValue = m_atRecord[i].tValue;
    }
  }

  if (tLocalRecord.iAcceptedNum == 0) {
    tLocalRecord.tValue = tValue;
    bAcceptPreparedValue = true;
  }

  if (tLocalRecord.iAcceptedNum > tLocalRecord.iPromisedNum) {
```

```
    return -3;
  }
  tLocalRecord.iAcceptedNum = tLocalRecord.iPromisedNum;

  iRet = CheckEntryRecord(tLocalRecord);
  if (iRet != 0) {
    CertainLogFatal("CheckEntryRecord ret %d", iRet);
    return -4;
  }

  iRet = CalcEntryState(iLocalAcceptorID);
  if (iRet < 0) {
    CertainLogFatal("CalcEntryState ret %d", iRet);
    return -5;
  }

  if (m_iEntryState != kEntryStateAcceptLocal) {
    return -6;
  }

  if ((tLocalRecord.iPromisedNum - 1) % s_iAcceptorNum != iLocalAcceptorID) {
    return -7;
  }

  return 0;
}

bool clsEntryStateMachine::IsRemoteCompeting() {
  if (m_iEntryState == kEntryStatePromiseRemote || m_iEntryState == kEntryStateAcceptRemo
    return true;
```

```
  }

  return false;
}

// 统计自身数据大小
uint32_t clsEntryStateMachine::CalcSize() {
  uint32_t iSize = sizeof(clsEntryStateMachine);

  for (uint32_t i = 0; i < s_iAcceptorNum; ++i) {
    if (m_atRecord[i].tValue.strValue.size() == 0) {
      continue;
    }

    bool bToAdd = true;
    for (uint32_t j = 0; j < i; ++j) {
      if (m_atRecord[i].tValue.iValueID == m_atRecord[j].tValue.iValueID) {
        bToAdd = false;
        break;
      }
    }

    if (bToAdd) {
      iSize += m_atRecord[i].tValue.strValue.size();
    }
  }

  return iSize;
}
```

```
}  // namespace Certain
```

算法描述中其他部分则主要实现于 src/EntityWorker.cpp 。该文件还实现了 CatchUp / Recovery 等功能，代码非常长，这里摘录出 Paxos 流程的核心代码：

```
// A 节点发起 Paxos 流程
int clsEntityWorker::DoWithClientCmd(clsClientCmd *poCmd) {
  uint64_t iEntityID = poCmd->GetEntityID();
  EntryInfo_t *ptInfo = m_poEntryMng->FindEntryInfo(iEntityID, iEntry);
  uint32_t iLocalAcceptorID = ptEntityInfo->iLocalAcceptorID;

  if (ptInfo == NULL) {
    // 创建 Entry，包含了 clsEntryStateMachine
    ptInfo = m_poEntryMng->CreateEntryInfo(ptEntityInfo, iEntry);
  }

  clsEntryStateMachine *poMachine = ptInfo->poMachine;
  int iEntryState = poMachine->GetEntryState();

  // 生成可用的 ProposalNum
  uint32_t iProposalNum = poMachine->GetNextPreparedNum(iLocalAcceptorID);

  // 创建一个新的状态
  EntryRecord_t tTempRecord;
  InitEntryRecord(&tTempRecord);

  // 自己当然直接 Promise
```

```
  tTempRecord.iPreparedNum = iProposalNum;
  tTempRecord.iPromisedNum = iProposalNum;

  // 更新状态机
  iEntryState =
      poMachine->Update(iEntityID, iEntry, iLocalAcceptorID, iLocalAcceptorID, tTempRecor

  // 获取自身状态
  const EntryRecord_t &tUpdatedRecord = poMachine->GetRecord(iLocalAcceptorID);

  // 构造消息
  clsPaxosCmd *poPaxosCmd =
      new clsPaxosCmd(iLocalAcceptorID, iEntityID, iEntry, &tUpdatedRecord, NULL);

  // 加入 PLog Req 队列进行持久化，可以暂时忽略
  iRet = clsPLogWorker::EnterPLogReqQueue(poPaxosCmd);
  return eRetCodePtrReuse;
}

// A 节点经过 PLogWorker 持久化后，会广播 Message
void clsEntityWorker::BroadcastToRemote(EntryInfo_t *ptInfo, clsEntryStateMachine *poMach
                                        clsClientCmd *poCmd) {
  EntityInfo_t *ptEntityInfo = ptInfo->ptEntityInfo;

  uint64_t iEntityID = ptEntityInfo->iEntityID;
  uint64_t iEntry = ptInfo->iEntry;

  uint32_t iLocalAcceptorID = ptEntityInfo->iLocalAcceptorID;

  // 遍历所有节点
```

```
  for (uint32_t i = 0; i < m_iAcceptorNum; ++i) {
    // 忽略自己
    if (i == iLocalAcceptorID) {
      continue;
    }
    // 获取自身状态和目标节点的视图状态
    const EntryRecord_t &tSrc = poMachine->GetRecord(iLocalAcceptorID);
    const EntryRecord_t &tDest = poMachine->GetRecord(i);
    // 构造消息
    clsPaxosCmd *po = new clsPaxosCmd(iLocalAcceptorID, iEntityID, iEntry, &tSrc, &tDest)
    // 设定发送目标
    po->SetDestAcceptorID(i);
    // 使用 IO Worker 发送消息
    m_poIOWorkerRouter->GoAndDeleteIfFailed(po);
  }
}

// X 节点通过网络接收到消息后
int clsEntityWorker::UpdateRecord(clsPaxosCmd *poPaxosCmd) {
  uint32_t iAcceptorID = poPaxosCmd->GetSrcAcceptorID();
  uint64_t iEntityID = poPaxosCmd->GetEntityID();
  uint64_t iEntry = poPaxosCmd->GetEntry();

  EntityInfo_t *ptEntityInfo = m_poEntityMng->FindEntityInfo(iEntityID);

  uint32_t iLocalAcceptorID = ptEntityInfo->iLocalAcceptorID;

  EntryInfo_t *ptInfo = m_poEntryMng->FindEntryInfo(iEntityID, iEntry);

  // 获取状态机
```

```
clsEntryStateMachine *poMachine = ptInfo->poMachine;
// 读取当前自己的状态
const EntryRecord_t tOldRecord = poMachine->GetRecord(iLocalAcceptorID);

// 获取消息中节点 A 发送的实际状态 S_A^A 和视图状态 S_X^A
const EntryRecord_t &tSrcRecord = poPaxosCmd->GetSrcRecord();
const EntryRecord_t &tDestRecord = poPaxosCmd->GetDestRecord();

// 更新 X 自身的状态机
int iRet = poMachine->Update(iEntityID, iEntry, iLocalAcceptorID, iAcceptorID, tSrcReco

// 获取实际状态 S_X
const EntryRecord_t &tNewRecord = poMachine->GetRecord(iLocalAcceptorID);
// 获取视图状态 S_A^X
const EntryRecord_t &tRemoteRecord = poMachine->GetRecord(iAcceptorID);

// 判断视图状态 S_X^A 与 S_X 是否一致
bool bRemoteUpdated = IsEntryRecordUpdated(tDestRecord, tNewRecord);

// 判断实际状态 S_X 是否存在更新
bool bLocalUpdated = IsEntryRecordUpdated(tOldRecord, tNewRecord);

// 构造消息
clsPaxosCmd *po =
    new clsPaxosCmd(iLocalAcceptorID, iEntityID, iEntry, &tNewRecord, &tRemoteRecord);
po->SetDestAcceptorID(iAcceptorID);
// 如果视图状态 S_X^A 与 S_X 不一致，那么说明 A 节点需要更新对 X 节点的视图状态，也就需要发送
ptInfo->bRemoteUpdated = bRemoteUpdated;

if (bLocalUpdated) {
```

```
    // 如果自身的实际状态存在更新，则需要使用 PLog 将其持久化
    iRet = clsPLogWorker::EnterPLogReqQueue(po);
  } else {
    // 自身状态不存在更新，直接进入回复阶段
    SyncEntryRecord(ptInfo, po->GetDestAcceptorID(), po->GetUUID());
  }

  return 0;
}

// X 节点持久化完成后，发送回复消息
void clsEntityWorker::SyncEntryRecord(EntryInfo_t *ptInfo, uint32_t iDestAcceptorID,
                                      uint64_t iUUID) {
  EntityInfo_t *ptEntityInfo = ptInfo->ptEntityInfo;

  uint64_t iEntityID = ptEntityInfo->iEntityID;
  uint64_t iEntry = ptInfo->iEntry;

  uint32_t iLocalAcceptorID = ptEntityInfo->iLocalAcceptorID;

  clsEntryStateMachine *poMachine = ptInfo->poMachine;
  const EntryRecord_t &tSrcRecord = poMachine->GetRecord(iLocalAcceptorID);

  // 如果需要发送回复
  if (ptInfo->bRemoteUpdated) {
    const EntryRecord_t &tDestRecord = poMachine->GetRecord(iDestAcceptorID);

    if (!tDestRecord.bChosen) {
      // 构造发送回去的消息
      clsPaxosCmd *po =
```

```
            new clsPaxosCmd(iLocalAcceptorID, iEntityID, iEntry, &tSrcRecord, &tDestRecord)

        // 设定发送目标
        po->SetDestAcceptorID(iDestAcceptorID);

        po->SetMaxChosenEntry(uint64_t(ptEntityInfo->iMaxChosenEntry));
        // 通过 IO Worker 发送回 A 节点
        m_poIOWorkerRouter->GoAndDeleteIfFailed(po);
      }
    }
  }

// 节点 A 收到回复消息后，同样会执行 UpdateRecord
int clsEntityWorker::UpdateRecord(clsPaxosCmd *poPaxosCmd) {
  ...

  // 更新 A 自身的状态机
  int iRet = poMachine->Update(iEntityID, iEntry, iLocalAcceptorID, iAcceptorID, tSrcReco

  // 判断 A 是否已经获得多数派的 Promised
  if (poMachine->GetEntryState() == kEntryStateMajorityPromise) {
    // 构造 Value
    PaxosValue_t tValue;
    tValue.bHasValue = true;
    tValue.iValueID = ptEntityInfo->poClientCmd->GetWriteBatchID();
    tValue.strValue = ptEntityInfo->poClientCmd->GetWriteBatch();
    tValue.vecValueUUID = ptEntityInfo->poClientCmd->GetWBUUID();

    bool bAcceptPreparedValue = false;
    // 尝试在 Accept 请求中使用请求 Value
```

```
    iRet = poMachine->AcceptOnMajorityPromise(iLocalAcceptorID, tValue, bAcceptPreparedVa
    // bAcceptPreparedValue 会返回是否使用该 Value
    // 如果 X 节点的 Promised 回复中已有 Value，则该操作会返回失败
  }

  // 继续进行 PLog 持久化 / 发送回复消息
  ...
}
```

注意这里为了让流程清晰，删减了大量代码。上述代码大概是 Paxos 完整流程的一半，后面还有另一半发起 Accept 请求、确定值的过程，过程和上方基本一致，就不重复了。

4. 总结

PaxosStore 使用自己提出的半对称消息来实现 Paxos 过程，从实现来看确实简化了代码的复杂度，每个节点收到消息时走的都是同一套流程。当然目前只分析了最简单的 Paxos 流程，代码中还有错误处理、数据持久化、预授权优化等内容，这些将会在后续的博文中逐步分析。

References

1. "Paxos Made Simple", *Leslie Lamport*
2. "可靠分布式系统基础：paxos的直观解释", *drdrxp*
3. "6.824: Distributed Systems", *MIT*
4. "PaxosStore: High-availability Storage Made Practical in WeChat", *Jianjun Zheng*

5. "腾讯PaxosStore中共识算法的验证", *黄宇*

0 comments

Write Preview Aa