# Fast paxos

一、Classic paxos

1、算法过程：

1) 新角色：coordinator:可以开始新的 round，接收 proposer 的 proposal 并转发给 acceptor

2) 具体化：

从一般的 Client/Server 来考虑，Client 其实承担了 Proposer 和 Learner 的作用，而 Server 则扮演 Acceptor 的角色，因此下面重新描述了 Paxos 算法中的几个角色：

Client/Proposer/Learner：负责提案并执行提案

Coordinator：Proposer 协调者，可为多个，Client 通过 Coordinator 进行提案

Leader：在众多的 Coordinator 中指定一个作为 Leader

Acceptor：负责对 Proposal 进行投票表决

就是 Client 的提案由 Coordinator 进行，Coordinator 存在多个，但只能通过其中被选定 Leader 进行；提案由 Leader 交由 Server 进行表决，之后 Client 作为 Learner 学习决议的结果。

Proposer-----Leader（coordinator）-----Acceptor-----Learner

3) 过程描述

- Acceptor (a)的信息：

  rnd(a):a 现在所参与的最大的 round,初始化为 0，（rnd=0 表示 a 还没有参与任何 round）

  vrnd(a):a 进行投票的最大的 round,初始化为 0,（vrnd(a)=0 表示 a 还没有对任何 value 进行过投票，vrnd(a)<=rnd(a)一直为真）

  vval(a):a 在 vrnd(a)中投票了的 value 值。

- Coordinator(c)的信息：

  crnd(c):c 已经开始的最大的 round 值，初始化为 0.

  cval(c):如果 c 在 crnd(c)中已经选择了一个 value，则 cval(c)的值为该 value，或者如果 c 还没有选择 value，则 cval(c)不确定。

- Round I 的过程描述如下，其中 c 为该 round 的 coordinator

  1、(a):如果 crnd(c)<I,则 c 开始 round i 并修改 crnd=i,cval(c)=none,并向每个 acceptor 发送加入 round I 的请求

  (b):如果一个 acceptor 收到了加入 round i 的请求并且 i>rnd(a),则 a 加入 round i 修改 rnd(a)=i,并向 coordinator 发送包含（rnd(a)，vrnd(a)）的 message。

  如果 i< rnd(a),则 a 忽略该请求。

  2、(a):如果 c 收到了来自一个 majority 的 round i 的 1b message，并且 crnd(c)=i , cval(c)=none(即 c 还没有开始更大的 round 并且也没有开始 2a 阶段),则 c 按照以下方法选出 value 值，并向 acceptor 发送接收 round i 的接收 value 的请求。

  （如果 c 收到的所有 1b message 中 vrnd 都等于 0，则可以选择任意一个 proposer 提出的 value 作为 cval，但若不是所有的 1b message 的 vrnd 都等于 0，则从所有的 1b message 中选出 vrnd 最大的值所对应的 vval 作为 cval）

  (b):如果一个 acceptor 收到了一个对 round i 的 value（v）的投票请求，其中 i>=rnd(a) 并且 vrnd(a)!=I,则 a 对 v 进行投票,修改 vrnd(a)=i,rnd(a)=i,vva(a)=v,病向所有的 learner 发送通知消息说明它在 round i 的投票

  如果 i<rnd(a)或者 vrnd(a)=I,(即 a 已经参与了更高的 round 或者 a 在 round i 已经投

票了)那么 a 就可以忽略该请求

2、progress property

1. 算法的 requirements

nonfaulty:一个 agent 被定义为 nonfaulty 当且仅当它最终一定会执行它应该执行的行为，比如说回应一个 message

good：一个 agent 的集合 G 并定义为 good 当且仅当 G 中所有的 agent 都是 nonfaulty 并且如果 G 中的一个 agent 重复的向 G 中的其他任意的 agent 发送消息的话，那么这个消息最终一定会被接收到。

对于任意的 proposer: p,learner:l,coordinator:c 和 acceptor 的 set：Q 定义 LA（p,l,c,Q）满足以下要求。

LA1.{p,l,c}∪Q 是一个 good set

LA2.p 已经提出了一个 value

LA3.c 是唯一一个相信自己是 leader 的 coordinator

则可以保证如下的 progress：

对于任意一个 learner：l,如果存在一个 proposer：p,coordinator:c 和一个 majority set：Q，LA（p,l,c,Q）从一个时刻开始始终成立，则 l 最终一定会 learn 一个 value。

然后对整个算法做一些小的修改使它可以满足 progress property:

CA1.如果一个 acceptor : a 收到了一个来自 round i 的 1a 或者 2a message,而 i<rnd(a), 并且 rnd(a)和 round I 不是同一个 coordinator，那么 acceptor：a 就会向 round I 发送一个特殊的 message 告诉它 rnd(a)已经开始了（如果 i<rnd(a), 并且 rnd(a)和 round I 不是同一个 coordinator，则忽视 round I 的 message）

CA2.一个 coordinator 执行一个动作当且仅当她相信自己是当前的 leader，并且它只会在以下两种情况下开始一轮新的 round（i）:

1） 当 crnd(c)=0 的时候

2） 当它发现一个 round（j）开始了的时候，其中 crnd(c )<j<i

因为 message 很有可能会丢失，以 agent 就需要去重传 message 来保证最终 message 会被送达，为此对算法稍作以下的修改：

CA3.每个 acceptor 都持续重发它发送的 1b 和 2b 信息，每个相信自己是 leader 的 coordinator 都会持续重发它发送的 1a 和 2a 信息，每个已经提出 value 的 proposer 都会持续向 coordinator 发送他提出的 proposal。

CA4.一个 nonfaulty agent 最终会执行任何一个它应该执行的动作。

2. Proof of progress

假设从 T0 时刻开始，LA（p,l,c,Q）都一直成立，则 l 最终一定会 learn 到一个 value

1） 从 T0 时刻开始，除 C 之外没有其他的 coordinator 会执行其他的行为

Proof：by LA3 & CA2

2） 存在一个时刻 T1 >T0,和一个 round number：i,即从 T1 时刻开始，crnd(c)=i

Proof:通过 LA1,LA3 和 CA4，c 最终会开始一个 round，由 1）可知，从 T0 开始，不会有其他的 coordinator 开始新的 round。而在 T0 的时候只会有有限的 round 被开始，则最终不再会有比 crnd（c）大的 round number，则由 CA2 可知 c 不会再开始新的 round，则存在一个时刻 T1 >T0,和一个 round number：i,即从 T1 时刻开始，使得 crnd(c)=i 一直成立。

3） 从 T1 时刻开始，对所有的 Q 中的 acceptor，都有 rnd(a)<=i

Proof:假设 T1 之后 Q 中存在一个 acceptor 使得 rnd(a)>i,由 2），LA1,LA3 和 CA3 可知，c 会持续向 a 发送 1a message，通过 LA1,CA1 和 CA4,a 会持续向 c 发送 1b message，而由 LA1,LA3 和 CA4 可知，c 最终会收到这个 1b message，从而 c 会知道有比 i 更大的 round number，则 c 会开始新一轮更大的 round，这会违反 2)，所以此处矛盾，所以得出结论从 T1 时刻开始，对所有的 Q 中的 acceptor，都有 rnd(a)<=i

4） 在某个时刻 T2>T1,coordinator 会开始 round i 的 phase 2

Proof：由 2)可知从 T1 时刻开始 c 开始了 round i.假设 round i 的 phase 2 阶段永远不会开始。

由 2），LA1,LA3 和 CA3 可知，c 必须持续向 acceptor 发送 phase 1a message。由 LA1 和 CA4 可知，Q 中的每个 acceptor 都最终会收到这个 phase 1a message，再由 3）可知，这些 acceptor 会向 coordinator 会发 phase 1b message。再由 LA1,CA3 和 1）可知，这些 acceptor 会回应 coordinator phase 1b message 并且最终会被 c 收到,当 c 收到来自一个 majority 的 acceptor 的 phase 1b message 时，则由 LA3 可知 c 总会开始它的 phase 2 阶段，则与假设矛盾，则得出结论在某个时刻 T2>T1,coordinator 会开始 round i 的 phase 2

5） 最终 l 会 learn 一个 value

Proof：由 4）,LA1,LA3 和 CA3 可知，c 会持续向 Q 中的 acceptor 发送 round i 的 phase 2 a message（包含已经选好的 value），由 LA1 可知，Q 中的每个 acceptor 都会最终收到这条信息，再由 CA4 和 2）可知，每个 acceptor 都会向 learner l 发送 phase 2b message,再由 CA3 和 3)可知，每个 acceptor 都会向 learner l 持续重发已经发送的 phase 2b message,则最终 l 会 learn 这个 value

二、Fast paxos

1、 在 Phase2a 的第一种情况下，即如果 Leader 可以自由决定一个 Value，则可以让 Proposer 提交这个 Value，自己则退出通信过程。只要之后的过程运行正常，Leader 始终不参与通信，一直有 Proposer 直接提交 Value 到 Acceptor，从而把 Classic Paxos 的三阶段通信减少为两阶段，这便是 Fast Paxos 的由来。因此，我们形式化下 Fast Paxos 的几个阶段：

Phase1a：Leader 提交 proposal 到 Acceptor

Phase1b：Acceptor 回应已经参与投票的最大 Proposer 编号和选择的 Value

Phase2a：Leader 收集 Acceptor 的返回值

Phase2a.1:如果 Acceptor 无返回值,则发送一个 Any 消息给 Acceptor,之后 Acceptor 便等待 Proposer 提交 Value

Phase2a.2：如果有返回值，则根据规则选取一个

Phase2b：Acceptor 把表决结果发送到 Learner（包括 Leader）

算法主要变化在 Phase2a 阶段，即：

若 Leader 可以自由决定一个 Value，则发送一条 Any 消息，Acceptor 便等待 Proposer 提交 Value

若 Acceptor 有返回值，则 Acceptor 需选择某个 Value

先不考虑实现，从形式上消息仅需在

Proposer-----Acceptor-----Learner

之间传递即可，也即仅需 2 个通信步骤。

2、 一些新的定义

- Quorum : classic paxos 中一直通过多数派（majority）来保证算法的正确性，对多数派再进一步抽象化，称为”Quorum”,要求任意两个 quorum 之间有交集（从而间接表达了 majority 的含义）
- i-Quorum：在 classic paxos 中执行过程中，一般不会明确区分每次 round 执行的 quorum。而在 fast paxos 中会通过 i-Quorum 明确指定 round i 需要的 Quorum
- Any message：在正常情况下，Leader 若可以自由决定一个 Value，应该发生一条 Phase2a 消息，其中包含了选择的 Value，但此时却发送了一条无 Value 的 Any 消息。Acceptor 在接收到 Any 消息后可做一些开始 Fast Round 的初始化工作，等待 Proposer 提交真正的 Value。Any 消息的意思是 Acceptor 可以做任意的处理。
  因此，一个 Fast Round 包括两个阶段：由 Any 消息开始的阶段和由 Proposer 提交 Value 的结束阶段，而 Leader 只是起到一个初始化过程的作用，如果没有错误发生，Leader 将退出之后的通信中过程。
- classic round : 执行 classic paxos 的 round 称为 classic round，若 leader 没有发送 Any message，则后续行为仍是 Classic round ，会和以前一样通信。
- fast round : 如果 leader 发送了 Any message,则认为后续通信是一个 fast round ；
  Acceptor 在接收到 Any 消息后可做一些开始 Fast Round 的初始化工作，等待 Proposer 提交真正的 Value。如果一个 coordinator 在一个 fast round 提交了一个 Any message，每个 acceptor 都可以独立的决定选择哪个 proposal 作为 phase 2a message ，这样就无法保证一致性。
  在 fast paxos 中，round number 被分为 fast round number 和 classic round number 这两种，classic round 和 fast round 可以通过 round number 来进行区分。

3、冲突：

在 Classic Paxos 中，Acceptor 投票的 value 都是 Leader 选择好的，所以不存在同一 Round 中投票多个 Value 的场景，从而保证了一致性。但在 Fast Round 中因为允许多个 Proposer 同时提交不同的 Value 到 Acceptor，这将导致在 Fast Round 中没有任何 value 被作为最终决议，这也称为“冲突”（Collision）

Proposer 提交的 Round 是全序的，不同的 Proposer 提交的 Round 肯定不一样，同一 Proposer 不可能在同一 Round 中提交不同的 Value，那为什么还会有同一 Fast Round 中有多个 Value 的情况？原因在于 Fast Round 与 Round 区别，当 Fast Round 开始后，会被分配一个唯一的 Round Number，之后无论多少个 Proposer 提交 Value 都是基于这个 Round Number，而不管 Proposer 提交的 Round 是否全序。比如，Fast Round Number 为 10，Proposer1 提交了（11，1），Proposer2 提交了（12，2），但对 Fast Round 来说存在（10，1，2）两个 Value。

因为冲突的存在，会导致 Phase2a.2 的选择非常困难，原因是：

在 Classic Paxos 中，如果 Acceptor 返回多个 Value，只要排序，选择最高的编号对应的 Value 即可，因为 Classic Paxos 中的 Value 都是有 Leader 选择后在 Phase2a 中发送的，因此最高编号的 Value 肯定只有一个。但在 Fast Paxos 中，最高编号的 Value 会发现多个，比如（10，1，2）。假如当前 Leader 正在执行第 i 个 Classic Round(i-Quorum 为 Q) ，得到 Acceptor 反馈的最高编号为 k，有两个 value：v、w，说明 Fast Round k 存在两个 k-Quorum，Rv,Rw。

下面定义在 Round k 中 v 被选择的条件：
O4（v）：一个 value（v）已经被选择或者还可能被选择当且仅当存在一个 quorum R 使得每一个 R∩Q 中的 acceptor a 都有 vr(a)=k,vv(a)=v

这个问题也可表述为:R 中的所有 Acceptor 都对 v 作出投票，并且 Q∩R≠ϕ，因为如果 Q∩R=ϕ，则 Round i 将无法得知投票结果
则为了避免同时有两个 value 被选择，提出以下要求：

- 每个 Acceptor 在同一 Fast Round 中仅投票一个 Value
- Q∩Rv∩Rw≠ϕ

第二点可以表达为
任意两个 Classic Quorum 有交集
任意一个 Classic Quorum 与任意两个 Fast Quorum 有交集
不妨设总 Acceptor 数为 N，Classic Round 运行的最大失败 Acceptor 数为 F，Fast Round 允许的失败数为 E，即 N-F 构成 Classic Round 的一个 Quorum，N-E 构成 Fast Round 的一个 Quorum。
上面两个条件等价于：
(a) N>2F
(b) N>2E+F
对于确定的 N,通常确定 E 和 F 的方法是把其中的一个最大化，

把 E 最大化时则得 E=F=⌈N/3⌉−1

在(a)的条件下把 F 最大化则得 F=⌈N/2⌉−1，再将 F 带入(b)可得 E=⌊N/4⌋

Leader 可以通过 nonfaulty 的 acceptor 的数量来决定是要开始 classic round 还是 fast round ，如果有多于 N-E 个 acceptor，则 leader 可以选择开始一轮 fast round，如果之后超过 E 个 acceptor failed，则 leader 可以再切换成 classic round

3、冲突 recovery
作为优化，Acceptor 在投票 Value 时也应该发送到 Leader，这样 Leader 就很容易能发现冲突。Leader 如果在 Round i 发现冲突，可以很容易地开始 Roun i+1，从 Phase1a 开始重新执行 Classic Paxos 过程，但这个其实可以进一步优化，我们首先考虑下面这个事实：
如果 Leader 重启了 Round i+1，并且收到了 i-Quorum 个 Acceptor 发送的 Phase1b 消息，则该消息明确了两件事情：
报告 Acceptor a 参与投票的最大 Round 和对应的 Value
承诺不会对小于 i+1 的 Round 作出投票
假如 Acceptor a 也参与了 Round i 的投票，则 a 的 Phase1b 消息同样明确了上述两件事情，并且会把对应的 Round，Value 在 Phase2b 中发送给 Leader（当然还有 Learner），一旦 Acceptor a 执行了 Phase2b，则也同时表明 a 将不会再对小于 i+1 的 Round 进行投票。
也就是说，Round i 的 Phase2b 与 Round i+1 的 Phase1b 有同样的含义，也暗含着如果 Leader 收到了 Round i 的 Phase2b，则可直接开始 Round i+1 的 Phase2a。经过整理，产生了两种解决冲突(Recovery)的方法：

- 基于协调者的 Recovery*

如果 Leader 在 Round i 中收到了(i+1)-Quorum 个 Acceptor 的 Phase2b 消息，并且发现冲突，则根据 O4(v)选取一个 value，直接执行 Round i+1 的 Phase2a；否则，从 Phase1a 开始重新执行 Round i+1

- 基于非协调的 Recovery

作为基于协调 Recovery 的扩展，非协调要求 Acceptor 把 Phase2b 消息同时发送给其他 Quorum Acceptor，由每个 Acceptor 直接执行 Round i+1 的 Phase2a，但这要求 i-Quorum 与(i+1)-Quorum 必须相同，并且遵循相同选择 value 的规则。

这种方式的好处是 Acceptor 直接执行 Round i+1 的 Phase2a，无需经过 Leader，节省了一个通信步骤，缺点是 Acceptor 同时也作为 Proposer，搞的过于复杂。

4、算法过程总结

Phase1a：Leader 提交 proposal 到 Acceptor

Phase1b：Acceptor 回应已经参与投票的最大 Proposer 编号和选择的 Value

Phase2a：Leader 收集 Acceptor 的返回值

Phase2a.1：如果 Acceptor 无返回值，则发送一个 Any 消息给 Acceptor，之后 Acceptor 便等待 Proposer 提交 Value

Phase2a.2：如果有返回值

2.1 如果仅存在一个 Value，则作为结果提交

2.2 如果存在多个 Value，则根据 O4(v)选取符合条件的一个

2.3 如果存在多个结果并且没有符合 O4(v)的 Value，则自由决定一个

Phase2b：Acceptor 把表决结果发送到 Learner（包括 Leader）