重温快速实用拜占庭容错算法

Ittai Abraham 著

Guy Gueta 著

Dahlia Malkhi, VMware Research 著

cr025, EthFans China, rchuqiao@gmail.com 译

2018 年 12 月 15 日

摘要

在这份说明中，我们发现了一个象鼻虫系统 (Zyzzyva) 的安全漏洞 [7, 9, 8] 以及 FaB 系统中的活跃度违规 [14,15]。为了展示这些问题，我们设定了一些简单的场景，包括四个复制和一到两个检视 (view) 变化。在如上所有的场景中我们能从第一个日志时间段中发现问题

1 引言

Castro 和 Liskov 提出的实用拜占庭容错算法 (PBFT)[3, 4] 是一个里程碑式的方法，它使得拜占庭容错算法可以有复制 (replication)。在 PBFT 提出之后，很多学者都致力于提高 PBFT 协议的效率。其中一个分支围绕着乐观性 (optimism) 展开 [10, 14, 15, 7, 9, 8, 5, 2]。在这条分支上，焦点在于如何提供一个快速的常用例（比如没有链接或者服务器错误）。在其他情况下，乐观解法倒退回一些有强大进程保证的替补方案。

在这份说明中，我们发现在乐观分支下的一些重要研究并没有正确的处理乐观性 (optimism)。我们在第二部分展示了象鼻虫系统 (Zyzzyva) 的安全漏洞 [7, 9, 8]。我们在第三部分展示了 FaB 系统 [14, 15] 如何被其“过分安全”的特征卡住。为了展示这些问题，我们设定了一些简单的场景，包括

四个复制和一到两个检视 (view) 变化。在如上所有的场景中我们能从第一个日志时间段中发现问题我们如下也发现了在其他快速拜占庭复制方案中，乐观的分支没有完全和正常的协议关联，因此他们不是很快速。

因此在 [12] 中发布的关于提供拜占庭快速 Paxos 算法的挑战依然存在：

> “快速 Paxos 算法可以被扩展到快速拜占庭 Paxos 算法，在没有碰撞的情况下，协议与学习之间只需要两个信息延迟（然而，一个不怀好意的提议者可以自己创建一个碰撞）。”[12]

也就是说，没有一个我们所知的快速拜占庭协议可以提供一个可以同时解决如下四种情况的方案: (1) 优化步骤的复杂度 (2) 优化弹性 (3) 保持系统免于少于系统总量三分之一的故障 (4) 在部分同步的期间保持进程。

我们已经做出了一个完整的解决方案，并会在不久的未来发布后续的文章。

前文

这个文章的重点是为 n 个复制提供复制状态机 (SMR)，这里的 f 是拜占庭错误。一个无界的用户组会发送一系列请求给复制机。我们把系统中不论是复制机还是用户都统称为节点 (node)。节点之间的通讯都是被认证，可信和异步的；换言之，我们假设从一个有效的节点发送给另一个有效节点的信息是经过认证并且最终会到达。

SMR 的核心是一个对用户请求日志裁定的增长施行决策的协议，需要满足如下的特征：

协议 如果两个有效的复制机在同一个日志点 s 施行了决策，那么这个决策一定是一样的

合法性 如果一个有效的复制机在日志的 s 位置做出了决断，那么这个（认证后的）决断一定是某个用户的请求

活跃度 如果一些有效的用户发送了请求，以及这个系统最终是部分同步的 (partially-synchronous)[6]，那么最终复制机会做出一些决断。

检视变化 (View Change)

我们讨论的方案设计到一个经典的框架，它通过检视次数来明确的把提案排名。

所有的复制机都从一个初始的检视开始，从一个检视过渡到另一个。他们仅接受并回复当前检视里的请求。

在每个检视中有且仅有唯一的一个特定的领导者 (leader)。在一个检视中，可能有零个或是多个决定。这样的策略把安全性从活跃度剥离开：即使系统展现出了任意的交流延迟并出现 f 拜占庭错误，它都能保证系统的安全性。它可以在同步的时期提供进展。

如果有足够数量的复制机怀疑领导者出现错误，那么检视就会变化并且一个新的领导者会被选出。这种触发改变到更高的检视的机制不会对安全性造成影响但是却对于活跃度非常重要。一方面，复制机不应该停滞在一个检视中没有进展；另一方面，他们也不应该反复无常的转移到更高的检视中，因为这样会阻碍任何检视的进展。因此，一个复制机应该转移到更高的检视中如果一个本地计时器过期了或者它从 $f+1$ 个复制机处接收到了新检视的建议。活跃度依赖于系统中固定百分比的检视都有一个正确的领导者，并且这个领导者与有效复制机的交流是及时的，这样就防止 $f+1$ 个复制机过期。

安全性和活跃度的巅峰是如何处理领导者的替换。在失败后重新获得一致意见最核心的方面是新的领导者如何安全的继承上一个领导者的数据。原因很简单，前一任领导者可能已经做出了一个裁决，那么最安全的做法就是继承这个决定。

在良性环境中占优势的解决方案 (DLS [6], Paxos [11], VR [16], Raft [17]) 中，从法定上选出的 $n-f$ 个复制机中选出有最多次检视变化的人来替换现有领导者。注意在这里的 $n-f$ 个复制机所代表的法定人数应该和前面所有 (不仅仅是最近的几个) 检视中的领导者的法定人数有交集。更重要的是我们要考虑到前几次检视的领导者法定人数是如何相互影响的。选择具有最大检视次数的决定是非常重要的，因为从几个相互冲突的决定中随机性的选择一个不一定永远是安全的。

PBFT [3, 4] 有一个相似的范例。新的领导者需要从 $n-f$ 个法定复制机上读数据并选择一个有最多检视的值。和良性环境不同的是，在拜占庭环境下，独特性是由一个更大的拜占庭法定人数 [13] 实现的。拜占庭法定人数保证了任意节点不只是有效节点之间的交集。

在拜占庭设定下，一个有效节点也需要向新的领导者证明决策。在 PBFT 中这个过程是在形成决策前加入一个新的阶段。第一阶段是通过在 $n-f$ 个节点上准备信息 (prepare message) 的方式保证独特性。在第二个

阶段，节点发送一个由 $n-f$ 个准备信息 (prepare message) 组成的保证证书 (commit-certificate)。当 $n-f$ 个节点发送了保证证书，决定就形成了。一旦一个决定形成了，这样的双阶段模式保证了一定会有一个有效节点发送了保证证书到下一个检视。

牺牲弹性

PBFT 额外的阶段也许可以由牺牲弹性并用 $n = 5f + 1$ 来避免，就像在 FaB[14, 15]，象鼻虫 [5, 7]，和 Q/U[1] 一样。这里，一个潜在决定的法定人群和一个检视变更的法定人群的交集有 $2f + 1$ 个有效节点，足够用来提供独特性和价值转移。

Kursawe 的解决方案

Kursawe 在 2002 年提供了一个简单的黑箱技能在有限的范围里把任何异步拜占庭协议 (有足够强的合法性性质) 转化成一个有最快速优化路线的共识协议。它是这么运作的：

在系统中有两个可能的达成一致的路线，他们有可能被合并 (有一些节点加入快速的路线有一些不)。在快速路线中，如果所有的节点准备了同样的数值那么一个节点就能形成决策。在备用路线上，任何拜占庭协议被调用时节点用它们的准备值作为起始输入值。协议的唯一需求是它需要满足如下的合法性属性：

拜占庭合法性 如果所有的有效节点都从一个输入址 v 开始，那么最后的决定值一定是 v。

这个简洁的解决方案是 (近似于毋庸置疑) 正确的。然而，在恢复的阶段我们并没有用到快速路线中已经完成的准备步骤。因此，虽然快速路线是快速的，备用路线缺不是最优化的。

此外，如我们提到的，这种解决方案仅局限在一个有限的范围里：它仅仅解决了一次性的共识，却完全没有解决复制状态机 (的执行) 问题。

FaB

FaB[14, 15] 在多个方面扩展了 Kursawe 的方法。首先，快速路线中的准备信息传送到了恢复阶段，因此减少了恢复模式中的步数。由此，FaB 的恢复模式与 PBFT 有相同的总成本。其次，FaB 扩展了对一个参数化错误

模型 $n = 3f + 2t + 1$ 的处理。因此，如果我们合理的增加系统的大小，即使有 t 个非领导者拜占庭错误，快速终止依然可以完成，并对 f 保证了安全性。

为了达到这样的增强效果，FaB 不能把在恢复阶段的拜占庭协议当作一个黑箱子。不幸的是，开放恢复协议以及把共识步骤加入到 FaB 框架导致了我们接下来会提及的疏忽 (参照 §3)。

象鼻虫系统

象鼻虫系统从 FaB 系统借鉴了把优化快速路线和恢复路线有效融合的方法。在此基础上它提高了各个方面。相比于 FaB 仅仅是一次的共识解决方案，象鼻虫系统提供了一个状态复制机协议。在象鼻虫系统中状态更新使用了投机形式，使得状态复制机复制的通道可以实现高吞吐量，这个在 FaB 系统中是没有的。最后，在象鼻虫系统中一个新的领导者不能像在 FaB 中一样卡在选择“安全”值的步骤。不幸的是，在象鼻虫系统中的检视变更协议无法阻止有缺陷的领导者所造成的安全隐患，这点会在 §2 中详细阐述。

Upright

象鼻虫系统的检视变更协议用的是 UpRight[5]，它和 FaB 一样包含 $n = 3f + 2t + 1$ 失误模型。UpRight 的目标是创造出一种以工程为优势的 BFT 引擎。UpRight 论文中并没有对算法做出一个具体的介绍，它只是指出 UpRight 借鉴了前两种解决方案。

下一个 700BFT 协议

在“下一个 700BFT 协议”中，Aublin 及他人 [2] 提出了一个在 BFT 协议中关于检视变更的原则性方案。他们的方案不仅转换了领导者们，也转换了整个组织方式，使它们都能回应系统的自适性状态。700BFT 协议家族中的一个节点是 AZyzzyva，它结合了象鼻虫系统中的投机（快速）路线，Zlight 协议以及一个恢复协议，比如 PBFT。如果 Zlight 没有进度，这个系统就会切换到新的检视，运行一个固定 k 个 PBFT 日志时段。因此 AZyzzyva 有 Kursawe 作为替补的同时也增加了众多状态机命令以及实现一个复制的状态机。Azyzzyva 实际上是简单且原则性的，并且它并没有我们将会在 §2 提到的安全违规。同时，Azyzzyva 的恢复路线比有两个阶段的象鼻虫协议

需要更多的步骤。除此之外，Azyzzyva 需要等到一个 (有 k 个时段的) 承诺决定 (commit decision) 来从 PBFT 切换到 Zlight。

2 再谈象鼻虫系统的检视变更

2.1 前言

象鼻虫系统 [7, 9, 8] 有两个执行路线：一个是如同 PBFT 一样的双阶段路线，另一个是快速路线。快速路线没有决策信息，客户如果看到 $3f+1$ 个准备信息即可执行决策。最佳模式和恢复模式的融合保障了错误中的进展。恢复模式把 PBFT 的双阶段步骤融入了协议。

[8] 中是这样阐述的:" 快速达成共识和投机性的执行对象鼻虫系统的检视变化属协议产生了深远的影响"。

事实上，在象鼻虫系统中，有两种可能的方式在检视中传送决策值，他们对应了协议中的两个决策路线（快速和双阶段）：在快速路线中，一个可能的决策值被表示为 $f+1$ 个准备信息。在双阶段路线中，它被表示为一个执行证书 (commit-certificate)，像在 PBFT 中一样。这两种路线中，比起 $f+1$ 个准备信息，象鼻虫系统更加偏好执行证书。如果有两个执行证书，系统就会更偏好用更长请求日志的那个。

这里我们将要展示这两个规则都有可能造成安全隐患。

这里被省略的地方不是很重要，因为除非一个领导者含糊其辞，执行证书是不会与更高检视的快速路线相冲突。

相似的，除非一个领导者含糊其辞，日志的增长是从一个检视到下一个。因此，在良性运行中，更高的检视有更长的（或者至少不是负增长的）命令序列。所以更高检视和更长的请求日志在表示上是一样的。

然而，我们在这里展示了两种策略都不能提供安全保证，因为他们允许我们在这里描述的会打破安全的场景。

2.2 象鼻虫系统的概括

我们从象鼻虫系统的概括开始。我们仅仅描述了象鼻虫系统的大体架构，忽略了详细的工程细节。我们假设所有的信息都被签署并在转发的时候携带着签名。我们忽略了检查点的机制以及空间回收，以及我们不优化信息大小和加密操作。这些细节都在象鼻虫的论文中有所涉及，这里为了简化我

们忽略了他们。

像原始的象鼻虫论文一样，我们把象鼻虫协议分成三个子协议：一个快速路线子协议，一个双阶段子协议和一个检视变化子协议。

信息 因为我们从 PBFT 借用了很多概念和术语，所以我们从参考指引开始，把象鼻虫的信息类型对应到 PBFT 上。

用户请求：一个从用户到领导者的用户请求（REQUEST）包含了一系列操作 o，为了这里讨论用途，它的语义是完全不透明的。

命令请求：一个领导者的前准备信息被称作命令请求 (ORDER-REQ)，它包括领导者的用户请求日志 $OR_n = (o_1, \ldots, o_n)$。（在现实中，领导者仅发送最后的请求和先前操作历史的一个哈希值。一个节点可以请求领导者重新发送任何缺失的操作。）

命令响应：当一个复制机接受了一个有效前准备信息，它会投机性的运行这个信息并把结果以被称作命令响应 (SPEC-RESPONSE) 的准备信息形式发送回来。

执行请求：从用户到复制机的执行请求 (COMMIT) 包括一个执行证书 CC，一组 $2f + 1$ 个有签名的复制机响应 (SPEC-RESPONSE) 于一个 (同样的) 命令请求 OR_n。

执行回应：当一个复制机从 OR_n 收到一个有效的执行证书 CC，它用一个被称作执行回应 (LOCAL-COMMIT) 的执行信息回应用户在 OR_n 里面的请求。

检视变化：从一个复制机到新检视的领导者的一个检视变化 (VIEW-CHANGE) 信息包含复制机的本地状态。

新的检视：从新的检视的领导者发出的新检视信息 (NEW-VIEW) 包括了领导者收集的一个集合 P 的检视变化信息，它被用作新检视的证明。它包括了新的顺序请求 $G_n = (o_1, \ldots, o_n)$。

快速路线子协议 象鼻虫包括了一个快速路线协议，在这个协议里用户只需要用三个信息的延迟就能得到请求的结果，并且只有一个线性增长数量的加密操作。它的运行过程如下：

一个用户给现在的领导者发送了一个请求 o。现在的领导者会扩展它的本地日志，把 o 加入到 OR_n，并发送一个携带 OR_n 的前准备 (pre-prepare)（命令请求）。我们并没有说领导者的日志时如何被初始化的。下面我们会讨论在开始新的检视的时候领导者如何拾起一个初始日志的协议。

一个复制机从现有检视的领导者接受一个前准备 (pre-prepare) 如果它是有效的格式，并且拓展从这个领导者输出的任何原先的前准备。在复制机接受一个前准备的时候，它拓展了它的本地日志以涵盖 OR_n，投机性的运行它，并把结果嵌入准备信息中直接发送回用户。当 $3f+1$ 个互不相同的复制机发送了准备信息的时候在检视 v 上对于 OR_n 的决定就生成了。

双阶段子协议 如果进程停滞了，那么一个用户将会等待收集一个执行证书，也就是 OR_n 的 $2f+1$ 个准备回应的集合。然后用户发送一个执行请求给复制机们，在这个执行请求上载有执行证书。一个复制机用一个执行消息回复一个有效的执行请求。在双阶段路线当 $2f+1$ 个不同的复制机发了一个执行消息那么在检视 v 里 OR_n 的决定就达成了。

检视变化协议 在象鼻虫中，检视间转移安全值的核心机制是新的象鼻虫领导者从 $2f+1$ 个法定复制机中收集一个检视变化消息的合集 P。每一个复制机发送一个载有复制机本地状态的检视变化消息：它的本地请求日志，以及它回应的有最高检视数的执行证书以及可能包括的执行信息。领导者按如下步骤处理集合 P:

1. 先开始，它把底层日志 G 设为空日志。

2. 如果任何的检视变更消息包括一个有效的执行证书，那么它选择一个有最长请求日志的 OR_n，并把 OR_n 复制给 G。

3. 如果 $f+1$ 个检视变化信息包含相同的请求日志 OR'_m，那么它用 OR'_m 拓展 G 的尾部。(如果有两个 OR'_m 日志同时满足，那么我们随机选择一个)。

4. 最后，系统根据任何一个有效准备中的最长日志的长度用空请求填满 G。

领导者发送一个新的检视信息给所有的复制机。这个信息包括了新的检视数 $v+1$，领导者收集的检视变化信息集合 P 作为新检视 $(v+1)$ 的证明，以及一个请求日志 G。一个复制机接受一个新的检视信息如果它是有效的，并且采用领导者的日志。它可能需要回溯并投机性的运行请求，并且运行新的请求。

2.3 违反安全的第一个场景

我们现在演示在象鼻虫中检视变化的机制是无法保证安全性。我们如上提供的象鼻虫的概括应该足够让读者理解如下的场景。对于精确的细节

以及象鼻虫协议的符号，读者请参见 [9]。

我们第一个场景演示了结合快速路线的决定和双阶段决定的标准可能导致安全违规。尤其是，像在象鼻虫系统中，让执行证书优先于 $f+1$ 个准备的情形，并不是经常正确的。

我们的场景需要四个复制机 i_1, i_2, i_3, i_4，其中 i_1 是拜占庭。它在三个检视中进行，并在第一个日志的位置达到了一个互相冲突的决定。

检视 1: 对于 (a) 创建一个执行证书

1. 两个用户 c_1, c_2 分别提供一个符合规则的请求 (REQUEST) 给检视 v_1 的领导者 i_1，分别表示为 a 和 b。

2. 在检视 1 中，领导者 i_1 给复制机 i_2 和 i_3 对于 a 发送一个前准备 (ORDER-REQ)。

3. 领导者 i_1(拜占庭) 含糊其辞并发给复制机 i_4 一个冲突的对于 b 的前准备。

4. 复制机 i_2 和 i_3 接受了领导者的符合规则的前准备，并投机性的运行 a。他们获得了一个投机性的结果并把它通过一个准备回应 (SPEC-RESPONSE) 的方式发送给 c_1。

5. 用户 c_1 对于请求日志 (a) 从 i_1, i_2 和 i_3 收集准备。这些回应构成一个执行证书，用 $cert$ 表示。

然后用户等待更多回应的时间到期。它对于 (a) 发送了一个执行请求 (COMMIT)，包括了执行证书 $cert$。执行请求只能到达 i_1。

检视 2: 决定 (b)

1. 所有的进一步消息都被延迟，迫使系统经过一个检视变化。

2. 在检视 2 中，领导者 i_2 从它自己，i_1 和 i_4 种收集检视变化信息 (VIEW-CHANGE)，如下所示：

- 复制机 i_2 发送它的本地日志 (a)
- 复制机 i_4 发送它的本地日志 (b)
- 复制机 i_1（拜占庭）结合 i_4 冰发送一个请求日志 (b)。

根据这些检视变化信息，i_2 建立起了一个由 (b) 组成的新的请求日志 G，并把它放在新检视信息 (NEW-VIEW) 中发送给复制机。

3. 每一个复制机从领导者 i_2 处接受符合规则的新检视信息 (NEW-VIEW)。接受以后，每一个复制机清空它的本地日志（如果需要的话，撤销 a）。所有的复制机都接纳了领导者的请求日志 (b) 并投机性的运行 b。他们取得了一个投机性的结果并把这些回应 (SPEC-RESPONSE) 发送给 c_2。

4. b 的用户 c_2 从所有的复制机处收集投机性的回应，然后 b **成功在日志的位置 1 被执行**。

检视 3: 选择了错误的执行证书

1. 所有的进一步消息都被延迟，迫使系统经过一个检视变化。

2. 在检视 3 中，领导者 i_3 从它自己，i_1 以及 i_4 处收集检视变化信息 (VIEW-CHANGE)，如下所示:

- 拜占庭复制机 i_1 隐藏了它为检视 2 准备的值，并对于 (a) 发送了一个执行证书 $cert$（见下）
- 复制机 i_3 和 i_4 发送了他们的本地日志 (b)

根据这些检视变化信息，i_3 选择执行证书 $cert$，并接受了它。它建立起一个新的由请求 (a) 组成的请求日志 G，并把它放在一个新检视消息 (NEW-VIEW) 中发送给复制机们。

3. 每一个复制机都从领导者 i_3 接受符合规范的新检视信息。接受完后，复制机清空他们的本地日志，如果需要的话撤销 b。然后他们投机性的运行 a，发送结果，然后 **a 在日志位置 1 被成功执行**。

2.4 违反安全的第二个场景

第二个场景演示了从不同的检视结合两个阶段决定的标准可能导致安全违例。尤其是，像象鼻虫系统里那样，优先选择最长的执行证书的做法，不一定永远是正确的。

我们的第二个场景再一次需要四个复制机 i_1, i_2, i_3, i_4，其中 i_1 是拜占庭。经过三个检视以后，我们在第一个日志位置达成了互相冲突的决定。为了能建立不同长度的执行证书，我们用到四个操作请求，用户 c_1 的 a_1，c_2 的 a_2，c_3 的 b_1 以及 c_4 的 b_2。

检视 1: 对于 (a_1, a_2) 创建一个执行证书

1. 四个用户 $c_1, \ldots, c_4$ 为检视 1 的领导者 i_1 提供了格式规范的请求 (REQUEST)，分别为 a_1, a_2, b_1 和 b_2。

2. 在检视 1 中，领导者 i_1 给复制机 i_2 和 i_3 发送了两个前准备信息 (ORDER-REQ)。第一个是为了在日志位置 1 的 a_1，第二个是为了在日志位置 2，排在 a_1 后面的 a_2。

3. 领导者 i_1（拜占庭）含糊其辞并发送了两个互相冲突的前准备请求给复制机 i_4。第一个是为了日志位置 1 的 b_1，第二个是为了日志位置 2，排在 b_1 后面的 b_2。

4. 复制机 i_2 和 i_3 接受了相关领导者的格式规范的前准备，并投机性的运行 a_1，随后运行 a_2。他们投机性的收到结果并把结果放在对应的准备回应 (SPEC-RESPONSE) 里发送给请求的用户。

5. 对应 a_2 的用户 c_2 从 i_1, i_2 以及 i_3 处对于请求日志 (a_1, a_2) 收集准备。这些回应用一个执行证书构成，表示为 $cert_1$。

然后用户等待更多回应的时长过期。它对于 (a_1, a_2) 发送一个包含有执行证书 $cert_1$ 的执行请求 (COMMIT)。执行请求只能送达 i_3。

检视 2: 决定 (b_1)

1. 所有的进一步信息都被延迟，迫使系统经过一个检视变化。

2. 在检视 2 中，领导者 i_2 从它自己，i_1 以及 i_4 收集检视变化信息 (VIEW-CHANGE)，如下所示：

- 复制机 i_2 把它的本地日志 (a_1, a_2) 发送出去。
- 复制机 i_4 把它的本地日志 (b_1, b_2) 发送出去。
- 复制机 i_1（拜占庭）结合 i_4 并发送请求日志 (b_1, b_2)。

根据这些检视变动信息，i_2 构造了一个由 (b_1, b_2) 组成的新的请求日志 G，并且把它通过一个新检视信息 (NEW-VIEW) 发送给所有的复制机。

3. 在 i_1, i_2 和 i_4 中每个复制机都从领导者 i_2 处接受了格式规范的新检视信息。在接收后，复制机 i 清空它的本地日志（如果需要的话撤销 a_1, a_2），并采用领导者的请求日志 (b_1, b_2)。它首先投机性的运行 b_1，获得一个投机性的结果，并把它放入回应 (SPEC-RESPONSE) 发送给 c_3。

4. b_1 的用户 c_3 从 i_1, i_2 和 i_4 处收集对于请求日志 (b_1) 的投机性的回应。这些回应组成一个执行证书，表示为 $cert_2$。然后用户不再等待更多的回应。它对于 (b_1) 发送了一个载有执行证书 $cert_2$ 的执行请求 (COMMIT)。

5. 在接收到格式规范的执行请求后，复制机 i_1, i_2 和 i_4 用一个执行消息 (LOCAL-COMMIT) 回应用户 c_3。

6. 用户收集到这些执行消息并且**在日志位置 1，b_1 被成功执行**。

检视 3: 选择了错误的最长执行证书

1. 所有的进一步信息都被延迟，迫使系统经过一个检视变化。

2. 在检视 3 里面，领导者 i_3 从它自己，i_1 以及 i_4 处收集检视变化信息 (VIEW-CHANGE)，如下所示：

- 复制机 i_3 对于 (a_1, a_2) 发送执行证书 $cert_1$（如上所述）。
- 复制机 i_4 对于 (b_1) 以及本地日志 (b_1, b_2) 发送执行证书 $cert_2$（如上所述）。
- 复制机 i_1（拜占庭）可以加入上面的任意一个，或者甚至发送一个载有空日志的检视变化信息。

根据这些检视变化信息，i_3 选择了载有最长请求日志的执行证书 $cert_1$，并采用了它。它创建了一个由 (a_1, a_2) 组成的新的请求日志 G，并把新检视信息 (NEW-VIEW) 发送给所有的复制机。

3. 每一个复制机从领导者 i_3 处接受了一个格式规范的新检视信息。接受后，复制机**清空了它们的本地日志，如果需要的话撤销了** b_1。然后它们投机性的运行了 a_1，发送了结果，**在日志位置 1，a_1 成功被执行**。

3 再谈 FaB 的检视变更

3.1 前言

象鼻虫协议借鉴了一个早期的名为 FaB（快速拜占庭共识）的成果。FaB 介绍了一整个系列的异步拜占庭协议解决方案，比起同步系统降低了延迟。尤其是，它建立了一个对于 $n \geq 3f + 2t + 1$ 当 $t \leq f$ 的参数化变种，当不超过 t 个非领导者成员失败，它有最优化的同步延迟系统。把 t 设为 0，我们得到一个和象鼻虫系统相似的设置，和一个在没有失败的运行中快速执

行的保证。在这篇文章中，为了简便，我们聚焦在确定 f 和 t 情况下 n 的最小值（从而达到 $n = 3f + 2t + 1$）。

简单来讲，把安全值从一个检视传送到另一个的核心机制围绕着一个“进程证书”。这个证书由签名过的从 $n - f$ 个法定复制机发送到新检视的领导者的新检视信息组成。从复制机发送的一个新检视信息包含这个复制机接受的最新的提名前 (pre-proposed) 信息，和它收到的最新的执行证书。一个进程证书“担保”了一个值 v 如果新检视的领导者可以安全的提名前 (pre-propose) 值 v。

像我们下面解释的，在参数化 FaB 中存在一个漏洞，这个漏洞到这一个进程证书可能不担保任何值，从而导致协议卡住。

象鼻虫从 FaB 系统借鉴了最优化快速路线的想法，并从各个维度加强了这个方法。象鼻虫提供了一个状态复制机协议，而另一方面 FaB 是一个一次性的共识方案。象鼻虫在状态更新的执行中利用了投机性，允许一个高吞吐量的状态复制机的流水线 (pipeline)，但是这确实 Fab 考虑范围之外的。最后，象鼻虫在检视变化协议中包含了检视序数，从而预防了我们如下讨论的 Fab 协议卡住的情况。

3.2 FaB 协议系列概括

«««< Updated upstream Martin 和 Alvisi 在 [14, 15] 中提出了快速拜占庭共识 (FaB)，一个系列的参数化协议，建立在许多弹性假设下。文章使用了 Paxos 术语来塑造职责：提出者，接受者和学习者。它用提案序数来遍历提案。我们继续使用象鼻虫（和 PBFT）的术语，并把这些翻译成领导者，复制机和检视序数。

FaB 有两个变种。第一个 FaB 的变种有 $n = 5f + 1$ 个复制机，在它们中选出领导者来驱动检视中的协定。我们把这个变种称为 FaB5。第二个变种是参数化的 $n = 3f + 2t + 1$，我们用 PFaB 来指代。

$5f + 1$ FaB. 最基本的 FaB5 协议是一个简单的两步协议。一个领导者先行向复制机提出一个值 (pre-propose)，每个复制机在每个检视接受一个值并回复一个准备信息 (prepare message)。当 $4f + 1$ 个复制机发送了准备回复一个决定的时候，FaB5 达成了一个决定。在同步的时候，FaB5 保证完成这两个简单的步骤，即使甚至高达 f 个随机（拜占庭）非领导者失败。

如果进程停滞了，复制机会选出一个新的领导者并进入新的检视。在 FaB5 中检视间传送安全值的核心机制是进程认证。一个进程认证由从法定

$4f+1$ 个复制机发送到新检视领导者的签名过的新检视信息 (REP) 组成。一个从复制机发送的新检视信息饱含这个复制机发送的准备信息中的值。

一个进程认证担保一个值 v，如果对于一组 $2f+1$ 个新检视信息，不存在一个相同的被接受的值 v'，使得 $v' \neq v$。

直觉上，FaB5 安全的原因是因为如果在一个检视中达成了一个决定，那么 $3f+1$ 个正确的复制机应该准备了这个决定。如果下一个检视被激活，那么在每一个进程认证的法定人群中，$2f+1$ 个法定复制机会避免担保任何冲突的提案。因此，不存在任何正确的复制机会覆盖一个已经被接受的值。

FaB5 是活跃的原因是不存在任何两组 $2f+1$ 复制机会担保对方的值。因此，永远存在一个可以提出的安全值。

参数化 FaB. 第二个 FaB 的变种被称为参数化 FaB（简称 PFaB）。PFaB 从一个历史悠久的早期停止共识 (early-stopping consensus) 处借鉴了最优快速执行路线的想法，尤其是从 [10] 中一个称为 Kursawe 的最优异步拜占庭协议合约。

PFaB 被 $n=3f+2t+1$ 参数化，这里 $t \leq f$。它有两个路线，快速路线和恢复路线。快速路线和 FaB5 一样，如果 $n-t$ 个复制机接受了领导者的提议，那么决定就会在两步中形成。如果有一个正确的领导者和不超过 t 个拜占庭复制机，快速路线可以保证在同步时期完成。

和 FaB5 不同的是，如果参数 t，失败的临界值，被超过，参数化 FaB 不能保证快速进程，即使是在同步时期。换言之，即使 PFaB 在不超过 f 个拜占庭失败的情况下依然是安全的，但它不一定总是快速的。只有在真实的拜占庭失败总数不超过 t 的情况下，快速路线才可以保证在同步时期在两步中被完成。

如果进程暂停了，PFaB 允许进程运行恢复协议，恢复协议本质上就是 PBFT（调整适应了 $n=3f+2t+1$）。

更精确的说，在 PFaB 中，恢复路线四处寻求达成一个执行证书，称为执行证明 (commit-proof)。当复制机接受了一个领导者的提议，在发送准备信息（ACCEPTED）给领导者的同时，复制机也发送签名过的准备信息给其他的复制机。当一个复制机在一个检视中接受到了 $n-f-t$ 个包含有同一个值的准备信息，它达成一个执行证书，并把它经由执行信息的方式 (COMMITPROOF) 发送给其他复制机。

如果包含有同一个值的 $n-t$ 个准备信息被发出，或者 $(n-f-t)$ 个执行信息被发出（有同一个值），那么一个决定就达成了。

就像在 FaB5 中一样，PFaB 发送安全值的核心机制就是从法定数量为 $n-f$ 个复制机发出的包含新检视信息 (REP) 的进程证书 (progress certificate)。不同的是，在 PFaB，从复制机发出的新检视信息包括它在准备信息中发送的最后一个值以及它在执行信息中发出的最后一个执行证书。

在 PFaB 中，一个进程证书担保 (vouch) 一个值如果不存在包含有同一个准备值 v' 的 $f+t+1$ 个新检视信息，且 $v' \neq v$。并且不存在任何执行证书包含有 v'，且 $v' \neq v$。

3.3 被卡住

在这个部分，我们展示了一个进程证书可能有 $f+t+1$ 个新检视信息包含某些值，但一个执行证书包含了另一个值。PFaB 被卡住的原因是证书无法担保任何值，因此新的领导者没有办法给出任何有效的提议。

在这个场景里，我们设置 $f=1, t=0, n=3f+2t+1=4$。我们用 i_1, i_2, i_3, i_4 来表示复制机，其中 i_1 是拜占庭。这个场景中有一个检视变化。

检视 1

1. 领导者 i_1(拜占庭) 提前提出一个值 A 给 i_2, i_3。2. i_1, i_2, i_3 接受了提议并发出准备信息 (ACCEPTED)。他们的准备信息只到达了 i_2，i_2 因此对于 A 达成了一个执行证书 (COMMITPROOF)。3. 同时，领导者 i_1 含糊其辞并向 i_4 提前提出了值 B。4. 所有进一步的准备信息，除却那些发送给 i_1 的，都被延迟了。这个延迟引发了一次检视变化。

检视 2

1. 新的领导者 i_2 从法定人数为 3 个的复制机群（包括它自己）中收集包括新检视信息 (REP) 的进程证书。

- 对于 i_1，新检视信息包括值 B，并没有执行证书。
- 对于 i_2，新检视信息包括值 A，并有一个执行证书。
- 对于 i_4，新检视信息包括值 B, 并没有执行证书。

现在我们卡住了。这个进程证书包含两个有准备值 B 的信息（从 i_1, i_4），因此，证书不会担保 A。与此同时，这个进程证书包含一个有准备值 A 的信息（从 i_2）。因此它也不会担保 B。

PFaB 论文指出所有的进程证书都会至少担保一个值，但是不幸的是这个论证是错的。

参考

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. SIGOPS Oper. Syst. Rev., 39(5):59–74, October 2005.

[2] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knezevic, Vivien Quema, and Marko Vukolic. The next 700 bft protocols. ACM Trans. Comput. Syst., 32(4):12:1–12:45, January 2015.

[3] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In Proceedings of the Third Symposium on Operating Systems Design and Implementation, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[4] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst., 20(4):398–461, November 2002.

[5] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09, pages 277–290, New York, NY, USA, 2009. ACM.

[6] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. J. ACM, 35(2):288–323, April 1988.

[7] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Spec- ulative byzantine fault tolerance. Best paper award. In Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07, pages 45–58, New York, NY, USA, 2007. ACM.

[8] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Spec- ulative byzantine fault tolerance. ACM Trans. Comput. Syst., 27(4):7:1–7:39, January 2010.

[9] Ramakrishna Kotla, Allen Clement, Edmund Wong, Lorenzo Alvisi, and Mike Dahlin. Zyzzyva: Spec- ulative byzantine fault tolerance. Com-

mun. ACM, 51(11):86–95, November 2008.

[10] K. Kursawe. Optimistic byzantine agreement. In Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems, pages 262 – 267, 2002.

[11] Leslie Lamport. The part-time parliament. ACM Trans. Comput. Syst., 16:133–169, May 1998.

[12] Leslie Lamport. Fast paxos. Distributed Computing, 19(2):79 – 103, October 2006.

[13] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. Distrib. Comput., 11(4):203–213, Oc- tober 1998.

[14] Jean-Philippe Martin. Fast byzantine consensus. Paper award. In Proceedings of the 2005 International Conference on Dependable Systems and Networks, DSN '05, pages 402–411, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. IEEE Trans. Dependable Secur. Comput., 3(3):202–215, July 2006.

[16] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC '88, pages 8 – 17, New York, NY, USA, 1988. ACM.

[17] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In Proc. USENIX Annual Technical Conference, pages 305–320, 2014.