Lab2 Primary-Backup Key-Value Service

6.824 Lab 2: Primary/Backup Key/Value Service

In the MapReduce lab handling failures was relatively easy because the workers did not maintain state. The master did maintain state, but you didn't have to make it fault-tolerant. Lab 2 is a first step towards fault tolerance for servers with state. In the next 4 labs you will build several key/value services.

http://nil.csail.mit.edu/6.824/2015/labs/lab-2.html

参考资料：

- Viewstamped Replication Revisited 论文，decentralized primary backup。非常有价值，应细看、多看。

  https://pmg.csail.mit.edu/papers/vr-revisited.pdf

- mit 6.824 关于这个 lab 的 notes。看最后这一块。

  http://nil.csail.mit.edu/6.824/2015/notes/l-remus.txt

- washington 同一门课的 lecture

  https://courses.cs.washington.edu/courses/cse452/18sp/PrimaryBackup.pdf

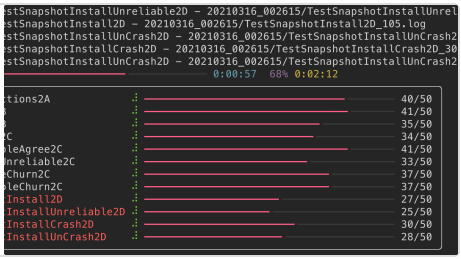
- debugging

https://pdos.csail.mit.edu/6.824/notes/debugging.pdf

Debugging by Pretty Printing

15 Mar 2021 This semester I'm a Teaching Assistant for MIT's 6.824 Distributed Systems class. The class requires students to iteratively implement Raft, a distributed consensus protocol.

https://blog.josejg.com/debugging-pretty/



Goal

实现一个基于 primary backup replication 的 fault-tolerant key-value service。

信息、要求以及假设

lab spec 并没有给出完整的 protocol，只要求实现基于 centralized view service 的 primary backup replication。因此需要自行实现一个在 lab 给定的场景下能够正确执行的 protocol。为此，首先应该构建正确及合适的 mental model。

需要注意 lab 的假设，包括 failure model, consistency model 以及其他 spec 说明了的假设。

需要理解 test cases，明确 test cases 是如何模拟各种场景以覆盖各种测试点，包括 concurrency, unreliable, crash 与 restart, partition 等。

总结信息、要求及假设如下：

- key-value db 作为 state machine，其仅支持 get, put 和 append 三种操作，这些操作都没有副作用，且都是 deterministic。因此 state machine replication 中的常用假设成立：primary 和 backup 从相同的状态开始，以相同的顺序执行相同的操作，必定得到完全一致的状态。
    - 推论：可以使用 state transfer，即 primary 将执行操作后的 state 发送给 backup，backup 再 install 这些 state，实现 replication。也可以使用 state machine replication， primary 根据自己接收或执行操作的顺序，将操作发送给 backup，backup 再执行操作，达到与 primary 执行操作后一致的 state。
- machine failure model: server 会 crash，failure model 为 fail stop（spec 没有明说，但根据测试用例可以得出），client 不会 crash，view server 不会 crash。

- network failure model: 比较通用的 failures 包括 delays, drops, dups, reorders, partitions，但只要一直发送，一定会 eventually delivered。由于 spec 明确说明了一个 client 在任意时刻最多只有一个 outstanding RPC，因此没有 dups 和 reorders（实际上，测试时会出现 dups，见之后的分析）。test 也没有 reorder msgs 的功能。至于 delays，一定是有的。
- consistency model: linearizability（spec 没有明说，但根据测试用例可以得出）。
- 对于一次 get, put 或 append 调用，client 端只有在确认上一个 request 结束后（request 执行成功 或 request 发送失败），才能发送下一个 request。且只有确认一个 request 执行成功后，才会结束此次调用。
    - 推论：client 端不需要进行并发控制。
- 虽然 lab spec 要求实现 at-most-once semantics，即对于一个操作，任意一个 server 最多执行一次。但是根据 test cases 反推，实际需要实现 exactly-once semantics，即对于一个操作，server 必须执行一次且只执行一次。
    - 推论：server 需要具备 filter dups 的能力。
- test cases 同一时刻只会出现最多 3 个 alive servers。
    - 推论：view 中只需要包括一个 primary 和一个 backup，primary 也只需要 forward request 给最多一个 server （即 backup）。
- concurrency: 测试会开多个 clients，每个 client 运行在独立的 goroutine。
    - 测试 primary backup 执行 put 的顺序一致：每个 client 执行一系列 put，put 的 key 和 value 都是随机生成，但是 key 的 range 是确定的。让这些 clients 运行一段时间，然后停止。对于 key range 中的每个 key，收集 primary 中所存的对应的 values（利用 get 调用）。然后 kill primary，则 backup 会被 promote 为 primary。此时再用同样的方法收集 values，则得到原 backup 的所有 key-value pairs。最后比对 primary 和 backup 对于同一个 key 是否给出了同样的 value。测试代码其实有一个潜在的问题，不表。
    - 测试 primary backup 执行 append 的顺序一致，且没有 dup: 每个 client 执行一系列 append, append 的 key 均为某个给定的常量，value 根据一定的规则生成，利用 client id 可以复现这些 values。让这些 clients 运行一段时间，然后停止。根据给定的 key，得到 primary 中所存的 value，这个 value 为一个字符串。根据 value 的生成规则，复现 values。对于每一个所复现的 value，利用字符串匹配算法（golang 内置的库），获取该 value 在 value 字符串中的第一个和最后一个位置。如果两个位置不等，则说明对应的 append 被执行了两次。同时记录

上一个 value 在 value 字符串中出现的位置。对于当前的 value 所出现的位置，如果其比上一个 value 在 value 字符串中出现的位置更早，则说明 server 执行顺序出现错误。然后 kill primary，则 backup 会被 promote 为 primary。此时再用同样的方法检查 backup 中的 value 字符串。

- unreliable: 对于每一个 request，server 都会新开一个 goroutine 去 handle 它。这个 goroutine 中的代码是这样的。有一定的几率直接关闭 socket，完全不 handle。有一定的几率关闭 socket 的发送功能，即 server 会 handle 这个 request，但 reply 发送不了。如果以上两个事件均不发生，则正常 handle 这个 request。这模拟了 network failure 中的 drops，包括 drop request 和 drop reply。
    - 不知道是 golang 的 socket 机制的问题，还是什么问题，使用以上的方法模拟 drop reply 时，会导致 msg dup。
- server crash: test 会启动一个 goroutine，它会每隔一段固定的时间间隔让随机的一个 server crash，此后该 server 无法接收和发送任何 msg。过一段固定的时间后，再创建一个新的 server，让它去 ping view service。test 还会创建一个 goroutine，它每次随机的产生一个 key-value pair，调用 put 写入到 server 中，然后再调用 get。如果能拿到对应的 value，则判断是否与刚才生成的相等。如果不能，直接跳过。
    - 这里需要注意的是，上述的固定的时间间隔是足够的。lab 设定了 ping interval，以及规定连续 5 次收不到某个 server 的 ping，则 view service 认为它 dead 了，此时会尝试更新 view。上述的固定的时间间隔均为 10 * ping interval，因此一个 server crash 后，cluster 有足够的时间进入新的 view，并且 primary 有足够的时间去 transfer state 给 backup。类似的，一个 server 重新加入集群后，有足够的时间参与到新的 view，例如 promote 为 backup，并得到 primary transfer 过来的 state。
    - 还需要注意的是，因为每次只会 crash 一个 server，因此 quorum intersection property 一定满足。因此 state 一定可以在 view change 过程中 preserve。
- partition: 使用 golang 提供的一些网络机制 block msg 的 delivery。

Part A The view service

view server 的实现并不复杂，关键在于对于 view 的理解。view 表示 cluster 中各 pb server 的 role，即 primary, backup。不在 view 中的 pb server 均被称为 idle server。

view server 端维护了 `currentView` 和 `latestView`，这是实现的关键。前者表示已经被当前的 primary ack 的、view server 允许 pb server 看到的 view，后者表示尚未被 primary ack 的、只有 view server 自己看到的最新的 view。需要注意的是，在 view server 刚启动时以及每次刚刚完成 view change 之后，以上性质不存在。还要注意的是，由于 idle server 可能被 promote 进 latest view，因此讨论 latest view 时通常包括了 idle server。

发送给 pb server 的 view 始终为 current view，而所有 view change 首先作用到 latest view。view num 只是 view 的编号，本身并不驱动 view change，因此只需要考虑 primary 和 backup 的变化。变化只有两种：（1）增：某个 pb server 被 promote 成 primary 或 backup。（2）删：primary 或 backup 被认定 crash 了而被从 view 中删去。

有两种情况会驱动删：（1）一个 pb server 发送了一个 view num 0 的 ping。（2）view server tick 时发现一个 pb server 已经长时间未 ping。这两种情况的处理一致，均是将该 pb server 从 latest view 中删去（如果它在 latest view 中的话）。

删腾出了 latest view 中的空位，因此驱动增。增的唯一方法就是 promotion，包括：（1）初始时将第一个 ping 的 pb server promote 成 primary。（2）primary fail，将 backup promote 成 primary。（3）backup fail，将 idle promote 成 backup。

在初始阶段之后，禁止将 idle 直接 promote 成 primary。这是显然的，因为 idle 不含任何 state。仅能将 backup promote 成 primary，这是为了保证 quorum intersection property，即连续两个 view 之间一定至少要有一个重合的 pb server。根据上述的 promotion 规则，这个重合的 pb server 在上一个 view 一定是 backup，在当前 view 一定是 primary。如此即可 preserve state over failures。

lab 还设定了一个限制：当且仅当 current view 的 primary ack 了，才能 switch 到 latest view。lab 给出的理由如下：

> The acknowledgment rule prevents the view service from getting more than one view ahead of the key/value servers. If the view service could get arbitrarily far ahead, then it would need a more complex design in which it kept a history of views, allowed key/value servers to ask about old views, and garbage-collected information about old views when appropriate. The downside of the acknowledgement rule is that if the primary fails before it acknowledges the view in which it is primary, then the view service cannot ever change views again.

我还没有理解为什么如果不确认 primary ack，就需要维护 a history of views。不过有关于这个限制的测试，因此在代码中要体现出来。这个体现在于：一是要每次收到 ping 时，检查是否 primary ack。二是在 promote 和 switch view 之前检查是否 primary ack 了。增和删的时候都不需要考虑 primary ack。

实现的时候还有一些点需要注意：

- backup promote 成 primary 后，可以尝试继续 promote idle 成 backup。
- 一个 pb server 以 ping 0 时，先把它从 view 中删除，给 promotion 腾出位置。待执行完 promotion 和 view switch 之后，再把它作为 idle。如果它之后还是 ping 0，则它永远不会参与进 view。这是显然的，因为它一直在反复的 crash。如果它之后不 ping 0 了，则会通过 promotion 参与进 view。
    - 在我的实现中，如果不这样做最后一个测试会过不了。这个测试的目的是保证 uninitialized server cannot become primary。这个测试是这样的：初始时 S3 和 S1 分别为 primary 和 backup，S2 为 idle。然后让 S3 和 S1 都 fail。理论上 S2 不应该被 promote 成 backup 或 primary，因为没有其他 server transfer state 给它。但是根据我打印的日志来看，S3 和 S1 并不是严格同步 fail 的，这就留出了一个 gap。即存在一个很小的时间段，S3 fail 了，但是 S1 没有。因此 S1 被 promote 成 primary，S2 被 promote 成 idle。当 S1 fail 后，S2 被 promote 成 primary。这个测试其实就表明了，一个 idle server 被 promote 的前提是：当前存在 primary。测试代码的意图是同步 fail S3 和 S1，导致没有 primary，因此无法 initialize idle。但是测试代码实际上没有保证这个同步性。
- 当且仅当存在 view change，才尝试 switch view。
- switch view 之后，注意 reset primary ack。

100次测试结果：

Test	Failed	Total	Time
TestViewService	0	100	6.89 ± 0.40

Part B The primary-backup key-value service

本部分主要讨论协议是如何解决几个重点问题的。关于协议具体是如何实现的，此处不表。

第一个重点问题是 split brain。由于这是一个 centralized system，即所有的 pb servers 通过 contact view service 达成谁是 primary 的 agreement，因此 split brain 仅可能因为 primary 和 view server partition 导致。由于 primary 无法 ping view server，因此一段时间后 view service 会认为 primary dead 了而 promote backup 和 idle，从而使得 cluster 中同时存在两个 primary。

解决这个问题的关键是保证只有一个 pb server acts as primary，即保证只有一个 pb server reply client requests。为此，需要在 reply 之前确保路径执行成功。这样一条路径是这样的：一个 client request 首先到达 primary，再被 forward 到 backup。要使得这条路径执行成功，应该验证路径上的所有节点的 view，即 client cached，primary cached 和 backup cached 的 view 应该完全一致。有任何不一致，都应该 abort。因为一旦发生 split brain，旧 primary 和 新 primary 的 view 必定不一致，因此通过验证 view 即可避免多个 brain acts as primary。

第二个重点问题是 filter dups。server 端维护了一个 `nextOpId` field，其存储着该 server expect 收到的、来自某个 client 的、下一个 request 的 op id。当接收到来自某个 client 的 request 时，考虑其 op id 与 next op id，无非三种情况：

- op id < next op id，说明该 request 已经成功执行。对于 Put, Append，可以直接 reject。对于 Get，需要返回 cached reply。注意只需要 cache 最新的 Get 的 reply，同时也只需要在 op id = next op id - 1 时，回复 cached reply。因为再往前的 Get，client 必定已经拿到 reply 了，否则按照 client 端不收到当前 request reply 不发送下一个 request 的逻辑，这个 request 是不会被发送的。
- op id = next op id，说明该 request 就是 expected。
- op id > next op id，说明该 request 是一个超前的 request。这可能是由于该 server 是一个刚启动的 backup，尚未 catch up primary。此时为了保证 backup 与 primary 按照统一的顺序执行 requests，应该 reject。

第三个重点问题是 state transfer。对于 state transfer，其数据流通方向必定是从 primary 到 backup。但是通知方式有两种选择：push based 和 pull based。前者表示 primary 向 backup 推送 state，后者表示 backup 向 primary 拉取 state。

如果是 push based，那么当更新 view 时，如果存在一个 backup，那么必定需要 transfer state，因为既然更新了 view，那么只有两种情况：（1）primary fail, backup promoted, idle server promoted。（2）backup fail, idle server promoted。根据 lab 的假设，不可能出现 primary backup 同时 fail 的情况。对于以上两种情况，当前的 primary 必定需要 transfer state。

如果是 pull based，那么当更新 view 时，当前的 backup 需要向 primary 发送一个 state pull 请求。

可以看出来，pull based 的思考更直接，因为只要一个 server 被 promote 为 backup，它一定之前是 idle server。则一定需要 state transfer。在代码的实现上也会省略很多判断。

primary 和 backup sync 之前会出现一个时间差，因为 primary 和 backup 更新 view 的时刻不一致。这个时间差是一定存在的，因为是分布式环境。这个时间差实际上类似于 split brain，只要验证 primary 和 backup 的 view 即可。

不管是哪种实现方式，都需要在 view 更新之后，transfer 完成之前，拒绝 receive 和 reply 任何 request ，不管是 client request，还是 forwarded request。这是因为包括 filter dups 在内的许多重要操作都与 state，即 next op id, cached reply 等紧密相关。为防止出问题，应该在 transfer 完成之前 reject。

其他还需要说明的点：

- 为了辨明 request 来自哪个 client，需要设计一个 client id。使用 starter code 所提供的 `nrand` 函数随机生成 unique 的 client id。
- 有不变量：the backup is *at most one* request ahead of the primary
- 参考了 viewstamped replication 和 raft，在每次 forward request 后，检查 primary 和 backup 的同步性。如果有任何不同步，直接 state transfer。
- 参考 viewstamped replication 和 raft，一个更好的做法是：primary forward 给 backup 的 request 带上 primary 的 view num。这样可以仅凭 view num 验证 view。
- 不要用 `hasPendingtransfer` 这样的 bool 量，应该用 `transferee` 这样有明确指向的量。因为这样可以对 transferee 做出更多判断和验证，打印日志时也更明确。

- 在这个 lab，最需要考虑的就是 server 两个 crash 情况：discard request 和 discard reply。要仔细思考 discard reply 可能发生在什么位置，以及有什么后果，以及需要做哪些调整。
- primary 执行 request 的顺序可以在 forward request 给 backup 之前或之后，只要保证 primary reply client 时，primary 和 backup 都成功执行了该 request。
- 目前只实现了 view service declare server as dead 的逻辑，但是没有实现反向的逻辑。这并不是说 view service crash 了，而是说可能出现 server 与 view service 之间的 partition。可能的一个做法是，当 server 连续一定次数（可以和 view service declare server as dead 的判断逻辑一致，连续 5 次 ping）ping 不到 view service 时（通过检查 `Ping` 是否返回任何错误），server 可以合理认为自己与 view service 之间产生了 partition，且 view service 很有可能已经 declare 自己为 dead，因此会 switch view。虽然目前的实现可以应对这种情况，但我认为更 robust 的做法是，让 server 拒绝任何 request，不管是 client request 还是 forwarded request，直到可以 ping 到 view service 为止。

100 次测试结果：

Test	Failed	Total	Time
TestBasicFail	0	100	9.05 ± 0.33
TestAtMostOnce	0	100	5.22 ± 0.20
TestFailPut	0	100	7.94 ± 0.16
TestConcurrentSame	0	100	24.78 ± 0.21
TestConcurrentSameAppend	0	100	5.11 ± 0.20
TestConcurrentSameUnreliable	0	100	10.23 ± 0.19
TestRepeatedCrash1	0	100	24.57 ± 0.17
TestRepeatedCrashUnreliable	0	100	24.67 ± 0.20
TestPartition1	0	100	7.89 ± 0.20
TestPartition2	0	100	9.12 ± 0.25