



Kafka Without Zookeeper

Apache Kafka Committer

邓子明



Andor Molnar - Tuesday, August 13, 2019 11:18:03 PM GMT+8

Subscriber API

<https://issues.apache.org/jira/browse/ZOOKEEPER-153>

Is it supposed to be something like a generic Observer API on the client side?

Observers essentially consume ordered updates of ZAB, so we would need to provide a way for users to implement their own "observers". They should be able to filter for path to be more convenient.

Andor



Jordan Zimmerman - Tuesday, August 13, 2019 11:20:35 PM GMT+8

Also see <https://issues.apache.org/jira/browse/ZOOKEEPER-1416> <<https://issues.apache.org/jira/browse/ZOOKEEPER-1416>>

There are many use cases where a client wants to see all events from a given parent path down. The semantics of setting one-time watches on a single node in ZK are cumbersome for these use cases. FWIW I had a working PR a few years ago but it's fallen far behind 3.6 now.

-Jordan



Ted Dunning - Wednesday, August 14, 2019 12:10:04 AM GMT+8

If you want to see all events, use Kafka.



Patrick Hunt - Wednesday, August 14, 2019 12:38:13 AM GMT+8



heh - I was thinking the same thing. :-)

Patrick



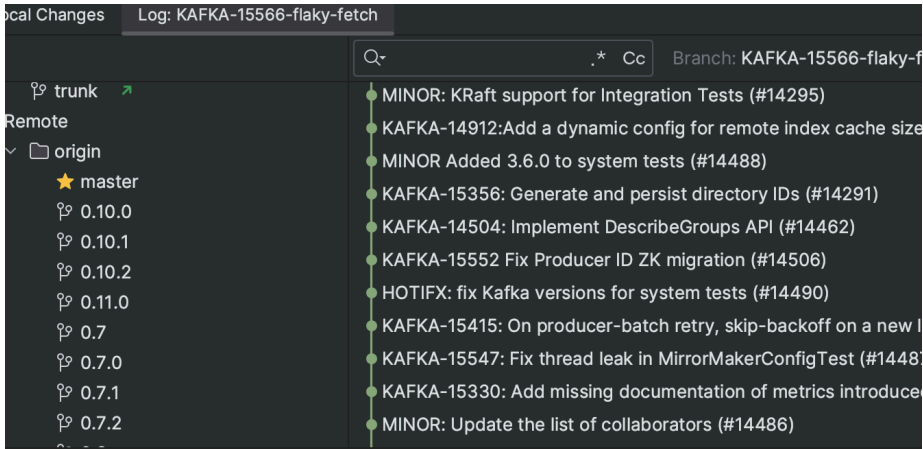
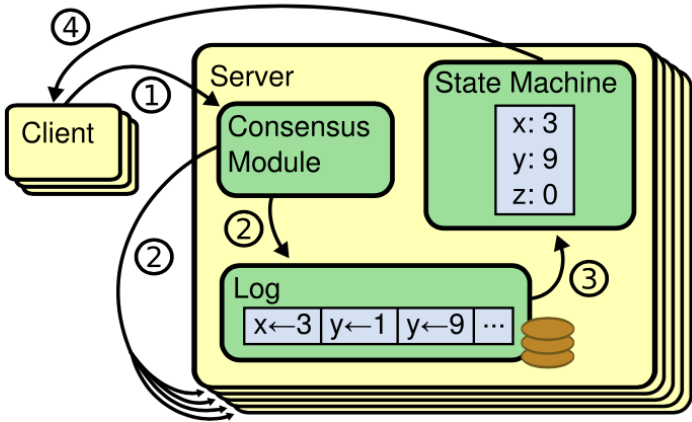
当 KIP-500 通过投票要移除 Zookeeper 时，Apache ZooKeeper 讨论过 Watches 不能够看到所有事件，而是 单次触发的语义是否需要改进。

Ted Dunning 给了一个非常经典的回应:If you want to see all events, **use Kafka**. 这个回应也得到了项目作者 Patrick Hunt 的认可。

<https://lists.apache.org/thread/fz9bkndvbntfjwxm952clh9vky3nwyd5>

探讨一下该话题-状态机和数据系统

数据系统 = 日志+状态机，下面我们举几个例子：



Rank			DBMS	Database Model	Score		
Apr 2023	Mar 2023	Apr 2022			Apr 2023	Mar 2023	Apr 2022
1.	1.	1.	Oracle +	Relational, Multi-model f	1228.28	-33.01	-26.54
2.	2.	2.	MySQL +	Relational, Multi-model f	1157.78	-25.00	-46.38
3.	3.	3.	Microsoft SQL Server +	Relational, Multi-model f	918.52	-3.49	-19.94
4.	4.	4.	PostgreSQL +	Relational, Multi-model f	608.41	-5.41	-6.05
5.	5.	5.	MongoDB +	Document, Multi-model f	441.90	-16.89	-41.48
6.	6.	6.	Redis +	Key-value, Multi-model f	173.55	+1.10	-4.05
7.	7.	8.	IBM Db2	Relational, Multi-model f	145.49	+2.57	-14.97
8.	8.	7.	Elasticsearch	Search engine, Multi-model f	141.08	+2.01	-19.76
9.	9.	10.	SQLite +	Relational	134.54	+0.72	+1.75
10.	10.	9.	Microsoft Access	Relational	131.37	-0.69	-11.41
11.	12.	11.	Cassandra +	Wide column	111.81	-1.98	-10.19
12.	11.	14.	Snowflake +	Relational	111.12	-3.27	+21.68
13.	13.	12.	MariaDB +	Relational, Multi-model f	95.93	-0.90	-14.38
14.	14.	13.	Splunk	Search engine	85.44	-2.54	-9.81
15.	16.	15.	Microsoft Azure SQL Database	Relational, Multi-model f	79.06	+1.62	-6.72
16.	15.	16.	Amazon DynamoDB +	Multi-model f	77.45	-3.32	-5.46
17.	17.	17.	Hive	Relational	71.65	+0.74	-9.77
18.	18.	18.	Teradata	Relational, Multi-model f	61.59	-2.14	-5.98
19.	19.		Databricks	Multi-model f	60.97	+0.11	
20.	21.	24.	Google BigQuery +	Relational	53.32	-0.12	+5.34
21.	20.	19.	Neo4j +	Graph	51.60	-1.91	-7.92
22.	23.	21.	SAP HANA +	Relational, Multi-model f	51.08	+0.24	-4.71
23.	22.	22.	FileMaker	Relational	50.00	-1.14	-2.91
24.	24.	20.	Solr	Search engine, Multi-model f	48.22	+0.94	-9.52
25.	25.	23.	SAP Adaptive Server	Relational, Multi-model f	42.87	-0.08	-5.49
26.	26.	25.	HBase	Wide column	37.79	+0.17	-6.54
27.	27.	26.	Microsoft Azure Cosmos DB +	Multi-model f	35.08	-1.03	-5.26
28.	28.	27.	PostGIS	Spatial DBMS, Multi-model f	29.41	-0.81	-2.64
29.	29.	28.	InfluxDB +	Time Series, Multi-model f	28.59	-0.56	-1.43
30.	31.	29.	Couchbase +	Document, Multi-model f	23.75	+0.39	-5.30

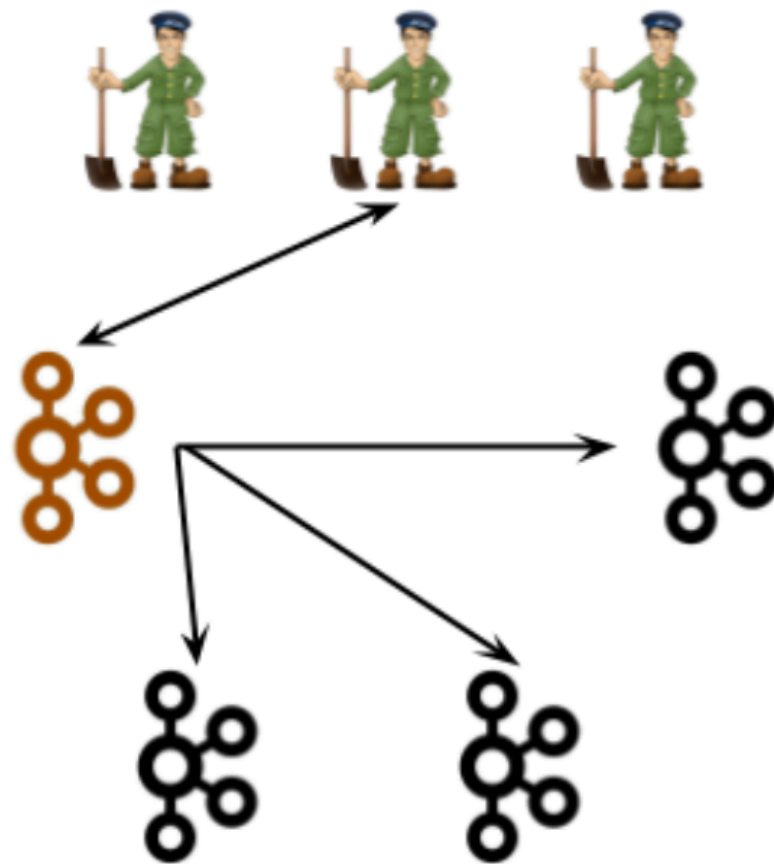
Replicated state machines



1. 从上面介绍我们总结出状态机的核心就是日志，日志在不同的状态机有不同的名字，比如 binlog、wal、commit log、Transaction log 等，所以理解数据系统必须理解 Log。
2. 在分布式系统中，Log 从保证 ACID 特性的一种实现，发展成了一种数据库之间数据复制的手段。因为从单机到分布式、我们只需要复制 Leader 即可，保证多个 replica 的 log 是一致的，得到的状态机一定能保证最终一致性。
3. 所谓的状态机复制原理 (State Machine Replication Principle)，假设两个确定的处理过程，从同样的状态开始，依照同样的顺序，收同样的输入，那么它们将会产生同样的输出，并以同样的状态结束。
4. 而 Replicated state machines，就是多个 server 保存一份 log 的多个 replication，每个 server 的 StateMachine 依次执行 log 得到一个一致的最终状态。

将数据业务抽象成状态机

1. 再次看 Ted Dunning 的发言
2. Zookeeper 和 KRaft?

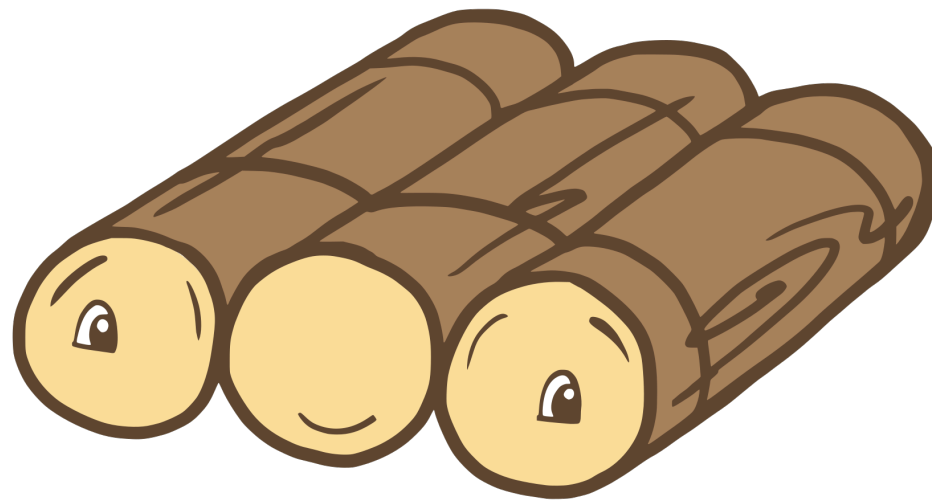


Raft-分布式日志

Raft 就是维护了一个 Log？是的，raft 的核心就是这么简单。

这个抽象是分布式系统的一大进步。

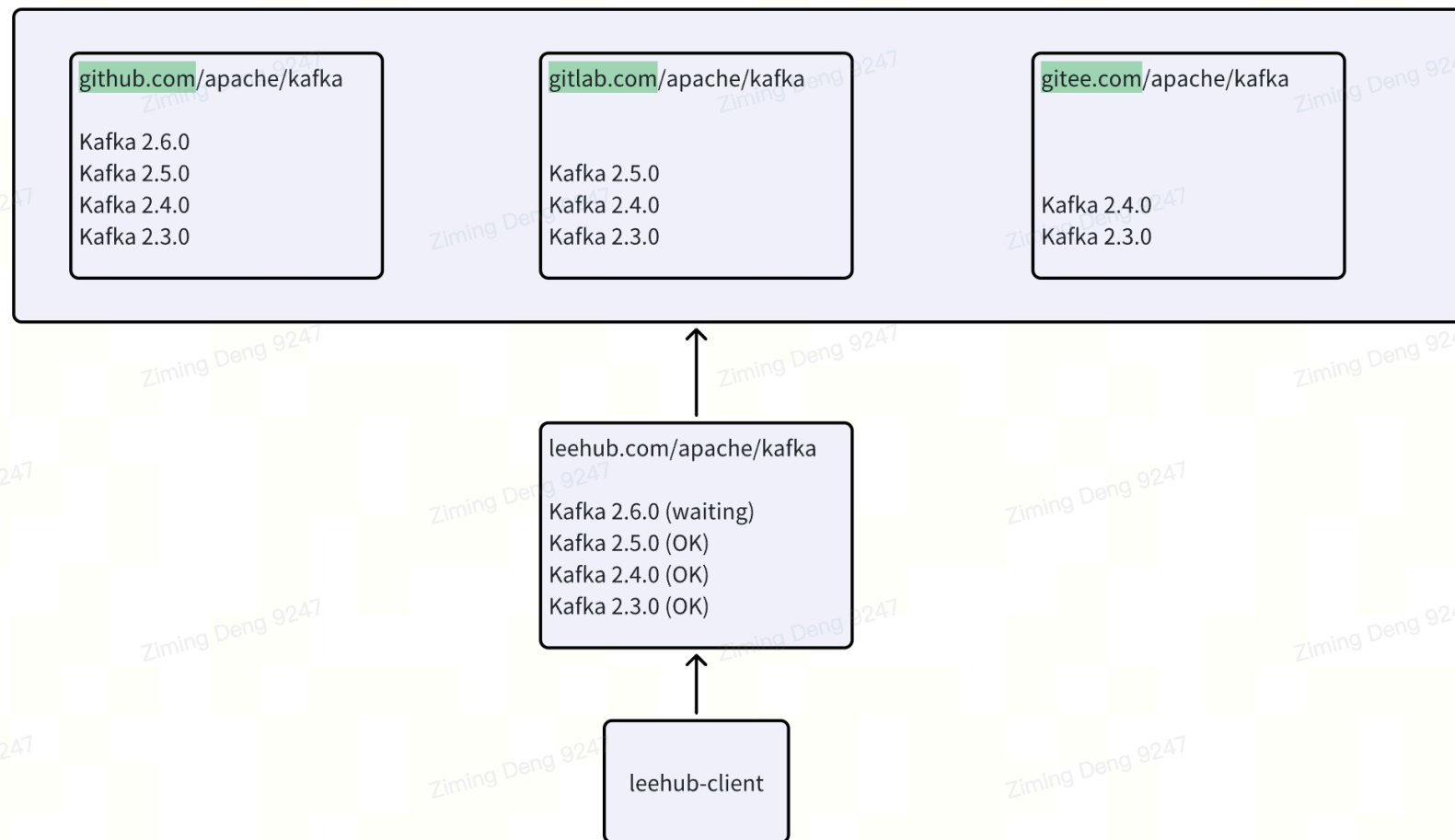
但是，维护一个日志并不是我们想象的那么简单。



Raft-分布式日志

git 就是一个日志系统，commit log 就是日志。

考虑我们要做一个分布式的 git 服务器中间件维护 Kafka 的源码 commit log，借助 gitlab、gitee、github 三个平台(看成服务器)，通过 Raft 来实现。leehub(server, client)



Raft-分布式日志

Leader 选举

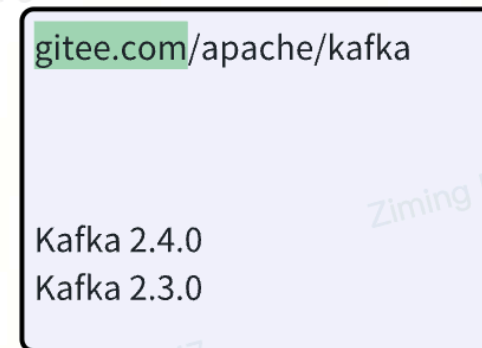
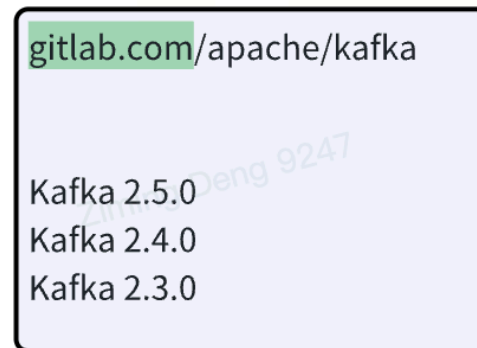
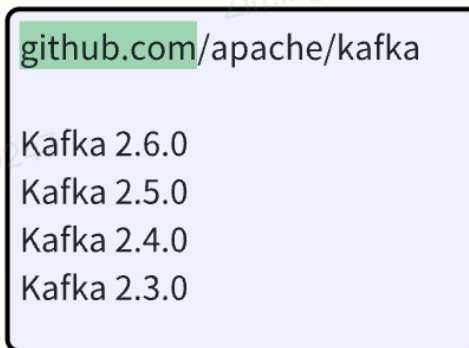
每个节点初始是 follower，通过心跳超时发现当前没有 leader，转变为 Candidate 发起投票，成为下一届的 leader

Follower：只投比自己代码新(后面介绍新的比较条件)，每一届任期内只有一张票，先到先得

Candidate：获得半数就成为这一届的 leader

证明：一个任期只有一个 leader

推演：split brain 现象



情况1：github 或者 gitlab 成为 term 1 的 leader

情况2：term 1 选举冲突，开始第二轮选举

Raft-分布式日志

日志复制

leader 只 write

follower 只 backup/truncate

证明：日志一致性

github.com/apache/kafka

Kafka 2.6.0, term 1

Kafka 2.5.0, term 1

Kafka 2.4.0, term 1

Kafka 2.3.0, term 1

gitlab.com/apache/kafka

Kafka 2.5.0, term 1

Kafka 2.4.0, term 1

Kafka 2.3.0, term 1

gitee.com/apache/kafka

Kafka 2.4.0, term 2

Kafka 2.3.0, term 1

每种情况下 truncate/backup 情况？

情况1：github 成为 leader，

情况2：gitlab 成为 leader

情况3：gitee 成为 leader

Raft-分布式日志

安全保证: more up to date

证明: 这一届的 leader 提交成功后就一定会出现在以后所有的 leader 上

问题: gitee 代码更旧, 但是 more up to date?

github.com/apache/kafka

Kafka 2.6.0, term 1

Kafka 2.5.0, term 1

Kafka 2.4.0, term 1

Kafka 2.3.0, term 1

gitlab.com/apache/kafka

Kafka 2.5.0, term 1

Kafka 2.4.0, term 1

Kafka 2.3.0, term 1

gitee.com/apache/kafka

Kafka 2.4.0, term 2

Kafka 2.3.0, term 1

Raft-分布式日志

1. Gitlab 一直落后，Github 和 Gitee 出现脑裂，轮流成为 leader，写各自的数据
2. Gitlab 同步 GitHub/Gitlab 上上个任期以前的代码，此时已经保存到 majority
3. 每次 Leader 切换，都会删掉 Gitlab 的数据

额外条件：leader 只能提交当前任期的记录

分析：比较的是最后一条日志，最后一条日志可能是上一个任期写的

github.com/apache/kafka

Kafka 2.5.0, term 3
Kafka 2.4.0, term 1

gitlab.com/apache/kafka

gitee.com/apache/kafka

Kafka 2.3.1, term 2

github.com/apache/kafka

Kafka 2.5.0, term 3
Kafka 2.4.0, term 1

gitlab.com/apache/kafka

Kafka 2.4.0, term 1

gitee.com/apache/kafka

Kafka 2.3.1, term 2

github.com/apache/kafka

Kafka 2.5.0, term 3
Kafka 2.4.0, term 1

gitlab.com/apache/kafka

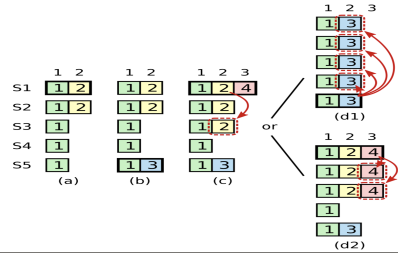
Kafka 2.4.0, term 1

gitee.com/apache/kafka

Kafka 2.3.2, term 4
Kafka 2.3.1, term 2

Dive into KRaft implementation

	KRaft	Raft	Kafka logs	
RPCs	Fetch, BeginQuorum, EndQuorum, RequestVote	AppendEntries, RequestVote	Fetch/AlterPartition	参考 KIP-595
Replica States	Leader、candidate、follower、unattached、voted	Leader、follower、candidate	leader、isr、all、removing、adding	
Persistent State	Log, epoch, voted, leader, voters	Log,term,voted	Log, isr, epoch	KRaft 保存了更多字段, 和准确性无关
Volatile State	highWatermark, listeners	Committed, applied	highWatermark	KRaft 设计了 listeners 类比 raft 的状态机, 每个 listener 维护各自 applied, 例如 kafka-metadata-shell.sh(类似 zk-cli.sh)
Snapshot	FetchSnapshot	InstallSnapshot	Log compaction/Tired storage	
membership changes	暂不支持	joint consensus		KRaft 已经有提案

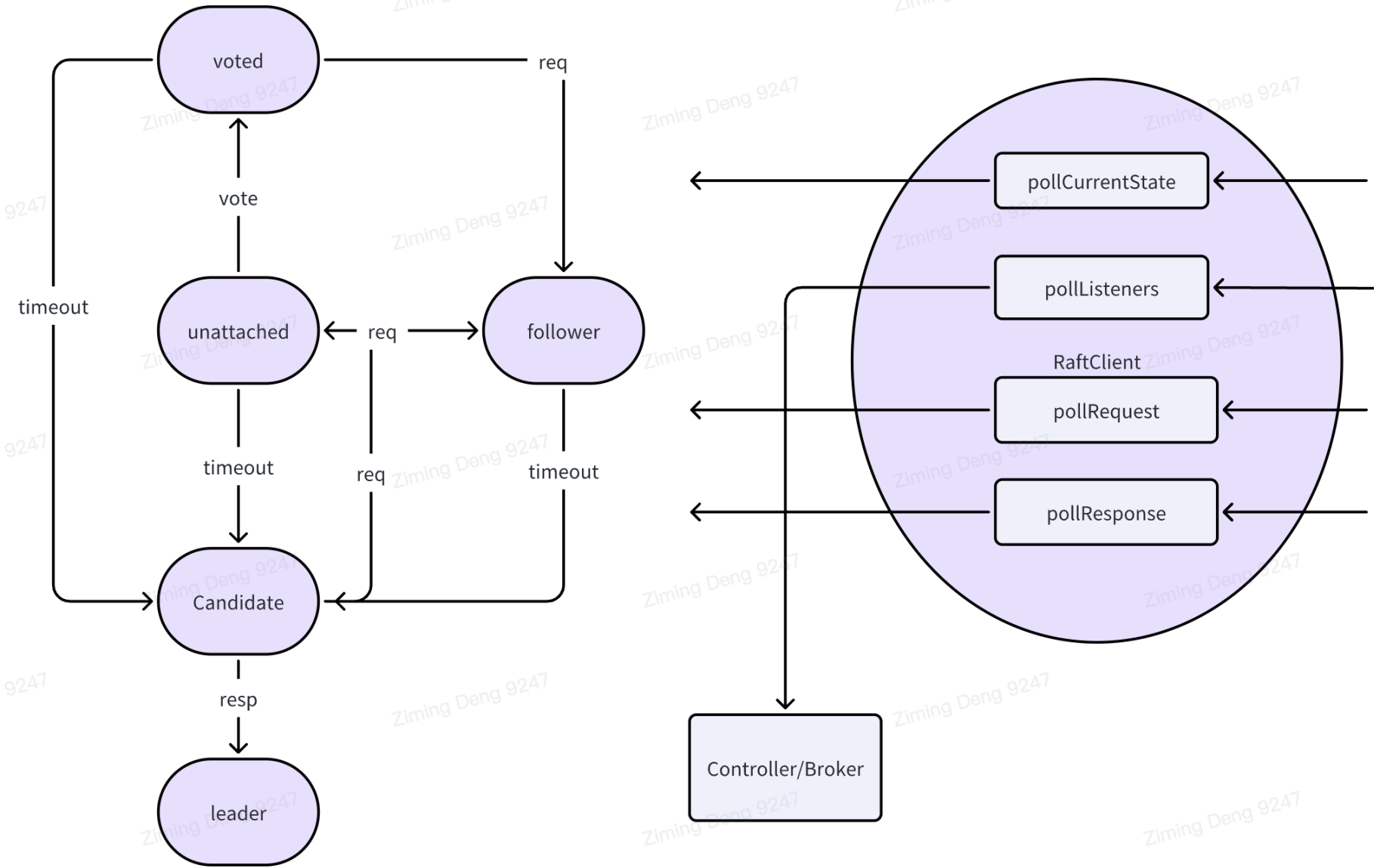
	KRaft	Raft	Kafka logs	Desc
Consistency check	Leader	Follower	Leader	Leader 不需要一次一次去找到不一致的分歧点, KRaft 更简单
Commit sementics	Quorum	Quorum	ISR	由于 pull/push 的区别, KRaft 需要下一次 rpc 才能 commit, Raft 更简单
Extra commit conditions	LeaderChange message	does not commit entries that are not from its current epoch	同 KRaft	
Leader election	Self managed	Self managed	很多策略、zookeeper 等 (整个集群通过 Zookeeper ephemeral node)	
Leader discovery	BeginQuorum	AppendEntries	Zookeeper/AlterPartition	Raft 更简单, KRaft 复杂度也不高

设计出发点

1. 设计上尽量保证不依赖其他项目模块，可以通过 sdk 的方式应用在其他项目
2. 实现是 single-raft，但是保留了 multi-raft 的能力
3. 将 raft 的整个实现设计成一个状态机，状态包括 leader/follower/...，驱动事件就是 timeout、write、RPC

核心类

1. RaftClient 主要包括 init() write(message) register(listener) 方法
2. Listener 主要包括 handleCommit(message) handleSnapshot(snapshot) handleLeaderChange()
3. ReplicatedLog 是日志接口
4. QuorumState 则是 raft 状态机所处状态，包括 resigned/unattached/voted/candidate/leader/follower/observer
5. ThresholdPurgatory 配置 TimingWheel 用来做延迟处理 RPC
6. KafkaRaftClient 是维护状态机的变化，入口方法 poll() 具体会调用 pollListeners() pollXXXState()



典型时序

1. leader: initialize() -> UnattachedState -> poll() -> timeout -> CandidateState -> poll() -> RequestVoteReq-> poll() -> RequestVoteResp -> Leader
2. Follower: initialize() -> UnattachedState -> poll() -> BeginQuorumReq
3. Observer: initialize() -> UnattachedState -> poll() -> FetchReq
4. Failover : leader -> poll() -> XXXRpc -> higher epoch -> follower
5. Replication: follower -> poll() -> FetchReq -> poll() -> FetchResp , update timeout -> poll() -> Fetch
6. Reelection: follower -> poll() -> poll() -> timeout -> candidate -> poll() -> ...

KRaft 使用案例：QuorumController

每个 Controller Node 启动一个 Controller，包含一个 RaftClient，并注册 RaftClient 的 Listener 监听 metadata log 的更新

RaftClient 状态为 leader 就是 active 的 controller，其余为 standby

所有生成 log 的 RPC 都会发送到 active controller，active controller 需要进行验证并生成

Active controller 调用 RaftClient 生成 metadata log，需要实现 MVCC 算法，支持多版本、回退

standby controller 需要实现 Listener 接口，监听到 log 的消息变化以后更新，只能监听到 committed log，不需要实现 MVCC

Log 消费者采用单线程设计，避免复杂的并发问题

QuorumController 一个写时序

启动 controller 后，某一个 controller 成为 active controller

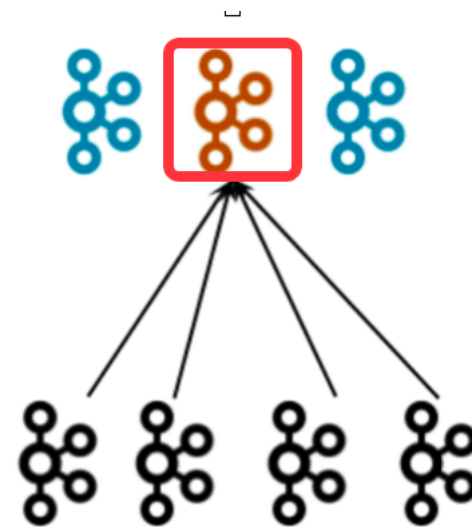
客户端发送 CreateTopic 的 RPC，最终转发到 Controller，验证以后生成包含 topic 和 partition 信息的 metadata log

active controller 将消息应用到底层的状态，需要考虑回滚的可能性，并调用 RaftClient 的 append 方法保存 metadata log

standby controller 的 raftClient 会同步日志，最终触发 Listener 的逻辑，将消息应用到底层的状态，不需要考虑回滚的可能性

当消息确定会被提交以后(备份到 majority)，不需要考虑回滚的可能性，active controller 会删除对应的版本信息

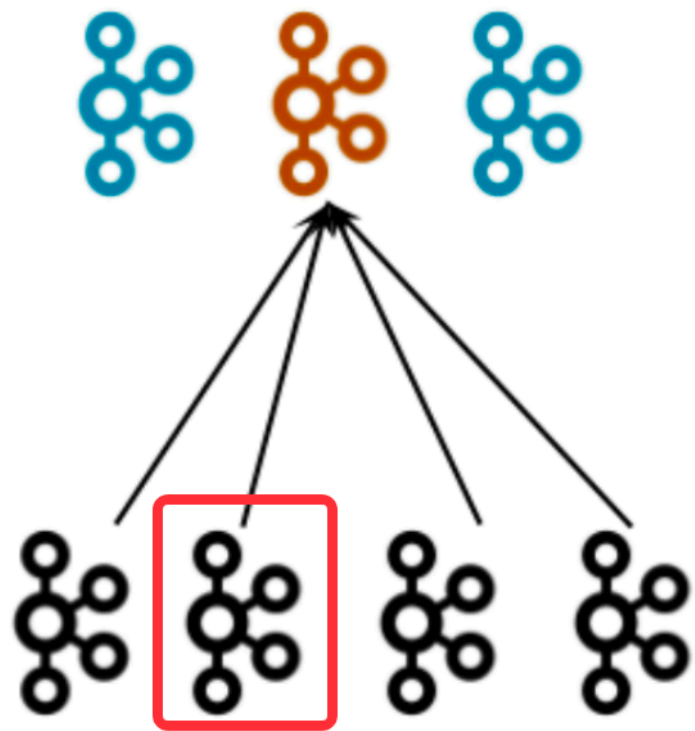
假设没有提交(少于 majority) 就发生了 election，旧的 active controller 回滚消息成为 standby controller，新的 active controller 直接接替，没有 $O(n)$ 的 failover



KRaft 使用案例：BrokerServer

KRaft 使用案例：BrokerServer

每个Broker Node 启动一个 BrokerServer，包含一个 RaftClient，并注册 RaftClient 的 Listener 监听 metadata log 的更新
RaftClient 状态都是 observer，只能监听到 committed log，应用到自己的 metadata 中BrokerServer



KRaft 待完善

KIP-856: KRaft Disk Failure Recovery

KIP-642: Dynamic quorum reassignment

kafka-metadata-shell.sh 只支持读本地日志，不支持网络连接

Multi-raft

KIP-500 后续工作

去 zookeeper, kafka-4.0

JBOD 支持

Zookeeper migration

Thanks

邓子明

dengziming1993@gmail.com

