1 Введение.

Цель данной работы состоит в том, чтобы исследовать вопрос доставки в распределенных системах на примере Apache Kafka. Хотелось бы понять, какие трудности возникают, как они решаются с помощью конфигурации.

2 Apache Kafka. Краткое описание.

Для данной работы нам будет достаточно знать, что Kafka - система, в которую можно писать, из которой можно читать. Будем называть записываемые данные сообщениями. Мы ожидаем, что каждое сообщение, которое попало в Кафку, может быть впоследствии прочитано.

Кафка кластер - множество брокеров Кафки, которые соединяются друг с другом с помощью системы ZooKeeper и начинают работать вместе.

Брокеры разделяют между собой топики, своего рода таблицы, если сравнивать с реляционными базами. Топики состоят из партиций, но в данной работе для простоты будем считать, что топик=партиция, поскольку в исследуемой теме партиции - не критично. У каждого топика есть реплики, брокеры, которые ассоциированы с ним. Подразумевается, что реплики должны синхронизироваться между собой. То есть нет какого-то конкретного места, где лежат сообщения в топике, они распределены между всеми репликами.

Среди реплик топика есть главная реплика, называемая лидером. Остальные называются фолловерами. Когда мы пишем в топик, запись идет на лидера. Он сразу записывает к себе, больше лидер ничего не делает.

Синхронизацию выполняет каждый фолловер независимо от других. Он идет на лидера, спрашивает у него о наличии новых изменений, и если таковые имеются, записывает их к себе.

Здесь важным будет сказать о понятии ISR(In-Sync Replica).

ISR реплика определяется очень легко.

После создания топика на него назначаются реплики. Изначально они все ISR.

Реплика теряет свойство быть ISR, если слишком долго не синхронизируется с лидером.

Реплика обретает его вновь, если успешно синхронизируется с ним.

Чтение всегда происходит из ISR.

Если лидер выходит из строя, происходят перевыборы. Выбирается новый лидер среди ISR.

Записью в кафку и чтением из нее занимаются продьюсеры и консьюмеры (producer, consumer). Они тоже обладают конфигурацией, которая будут важны нам чуть позже.

3 Базовая конфигурация.

Во всех следующих тестах будем использовать такую конфигурацию: 3 брокера и 3 zookeeper-а.

Это позволит обеспечить некоторую базовую доступность.

4 План тестов:

Тестирование проводилось с кафкой, которую я развернул в kubernetes кластере (minikube), 3 zookeeper-а и 3 kafka брокера.

Запись и чтение производилось с использованием библиотеки на языке Java. Конфигурация продьюсеров и консьюмеров, соответственно, описывалась с помощью этой библиотеки.

Падение отдельных узлов системы я эмулировал путем ручного отключения подов. (они спустя указанное время оживали, будучи частью деплоймента)

4.1 Правило acks.

Основной параметр конфигурации, с которым нам придется работать - acks. Это параметр конфигурации продьюсера. Он важен тем, что именно он позволит нам определить и понять, что мы будем считать успешной записью. Вернее, успешной записью мы будем считать запись, при которой лидер ответил нам, что запись прошла успешно. Но вот что именно этот ответ лидера значит - очень сильно зависит от параметра acks.

Всего у него может быть три значения, фактически это три режима работы продьюсера.

acks = 0 - успешной считаем вообще любую запись, поскольку не ожидаем ответа от лидера. acks = 1 - успешной считается лишь та запись, при которой лидер смог записать данные в себя. acks = 2 - запись, при которой лидер записал в себя данные и все ISR отреплецировали их.

Разумно будет считать, что параметр *acks* будет определять, находятся ли данные в системе. Причём, полную уверенность в том, что дан-

ные пришли, мы можем получить лишь если отправка производилась с параметром $acks = all$ и сервер успешно ответил нам о том, что данные были записаны и отреплецированы.

Будем считать настоящим data loss (true data loss) тот случай, когда отправитель получил ответ от сервера об успешной записи, но при этом, при последующем чтении, мы обнаружили, что эти данные отсутствуют.

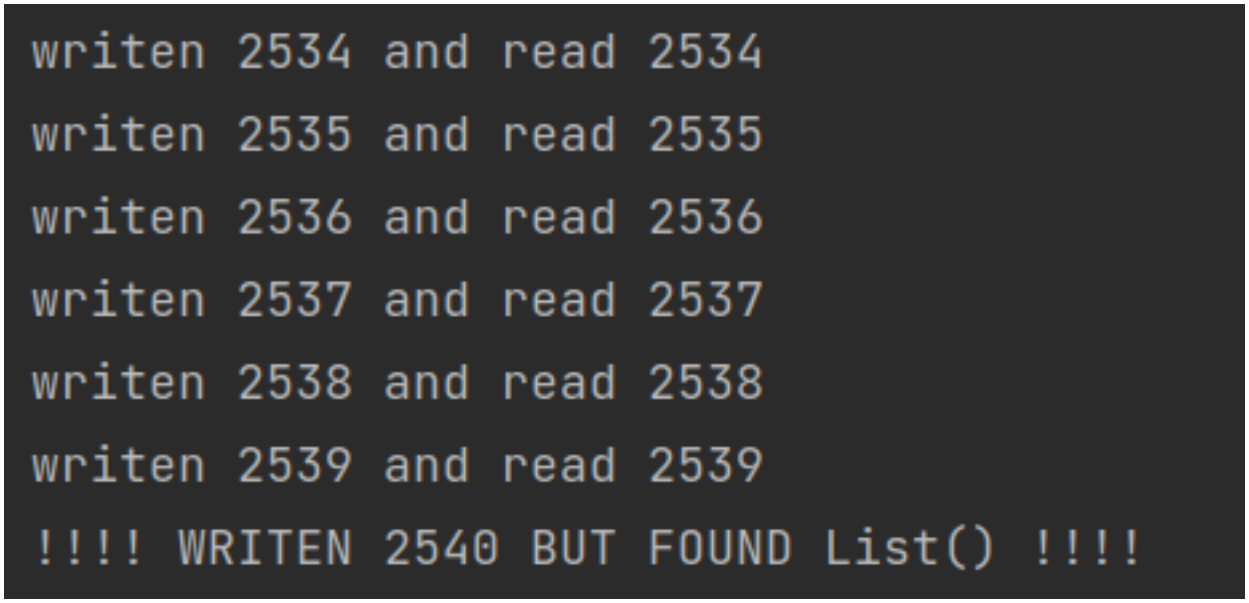
С acks=0 вроде бы все понятно. Здесь мы вообще не можем ничего гарантировать. Поэтому, такой случай действительно не интересен. Здесь нельзя говорить о потере данных. Такую систему сложно обвинить в неконсистентности, но легко в бестолковости.

Теперь что касается acks=1. Здесь мы можем получить true data loss. Алгоритм очень прост. Если мы записали на лидера сообщение, лидер ответил нам успехом, а реплики пока не торопятся синхронизироваться, может случиться так, что лидер упадет. Поскольку прошло не более $replica.lag.time.max.ms$ времени (этот конфигурационный параметр брокера отвечает за то, сколько времени должно пройти без синхронизаций, прежде чем лидер перестанет считать фолловера ISR) фолловеры остались ISR, можем выбрать нового лидера (несмотря на то, что на этих фолловерах лежат устаревшие данные, не хватает нашего сообщения)

5 TEST1

ЦЕЛЬ: смоделировать падение лидера при записи с параметром acks=1, получить true data loss. СПОСОБ ДОСТИЖЕНИЯ: спамить записи в кластер, потом резко уронить лидера, затем прочитать данные с кластера и убедиться в том, что данные, которые там должны быть, не совпадают с теми, которые там находятся. В первом тесте безостановочно выполняем в кафку 10000 записей.

Практически с первого раза был получен следующий результат:



5.1 Объяснение теста:

Из-за параметра acks=1, брокер поспешил сообщить нам о том, что сообщение было записано. Мы это запомнили, однако сам лидер брокер 1 не успел отреплецировать данные и умер. Очевидным образом произошло переназначение лидера, мы пошли дальше считывать данные с брокера 3, но ничего там не обнаружили.

5.2 Вывод из теста:

В вопросе репликации кафка предпочитает доступность, жертвуя консистентностью. Мы могли бы положить всю партицию, исходя из того, что мертвый лидер обладает современными данными, которые не известны другим брокерам, и мы будем ждать, когда он воскреснет, но тогда мы потеряли бы доступность. Однако здесь мы предпочли сохранить доступность, пожертвовав консистентностью. Пусть лучше одна запись потеряется, чем мы вообще потеряем возможность писать в кафку.

6 А что если acks=all?

Что касается acks=all, здесь намного сложнее придумать ситуацию, при которой что-то могло бы пойти не так.

В крайнем случае, лидер просто не вернёт нам никакого ответа, что будет значить, что что-то пошло не так, сообщение не записано. Но всё ли так просто? На самом деле нет. И в этот раз нужно подумать вот о чём.

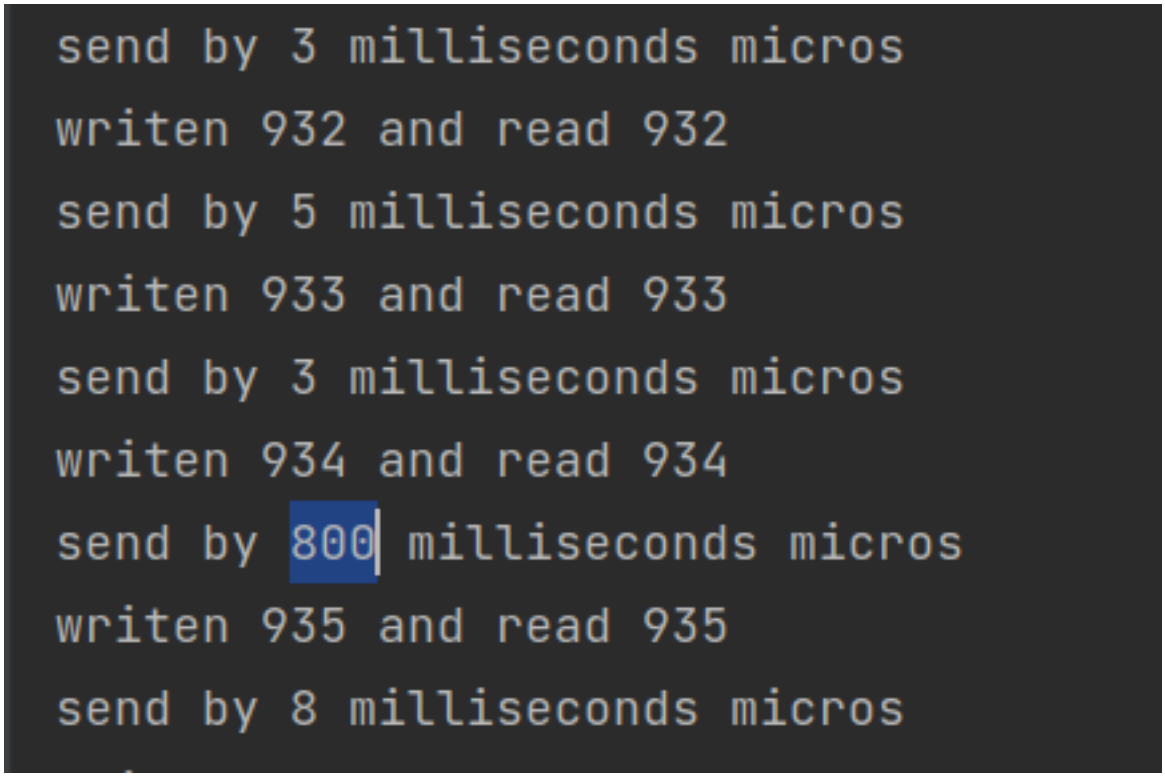
Если в случае acks=1 нам пришёл ответ, мы едва ли можем быть уверенны в том, что сообщение по-настоящему записалось в систему, что было подтвержено предыдущим тестом.

Если же нам пришёл положительный ответ в случае acks=all, то будем считать, что ничего не предвещает опасности. (система гарантирует, что данные отреплецированы по всем репликам и не осталось реплики, в которой они были бы старыми, за исключением тех реплик, которые перестали быть in-sync в силу того, что не смогли быть отреплецированы, но эти реплики более не являются участниками топика, то есть не способны оказать вредное влияние на нашу систему. Мы не читаем из этих реплик, пока они не синхронизируются с лидером и не станут in-sync)

Все реплики знают о наших данных, лидер знает о наших данных - всё чудесно. Но что значит отсутствие ответа при acks=all? Да и вообще, какой вывод мы можем сделать, если нам не пришёл ответ? Стоит подумать об этом.

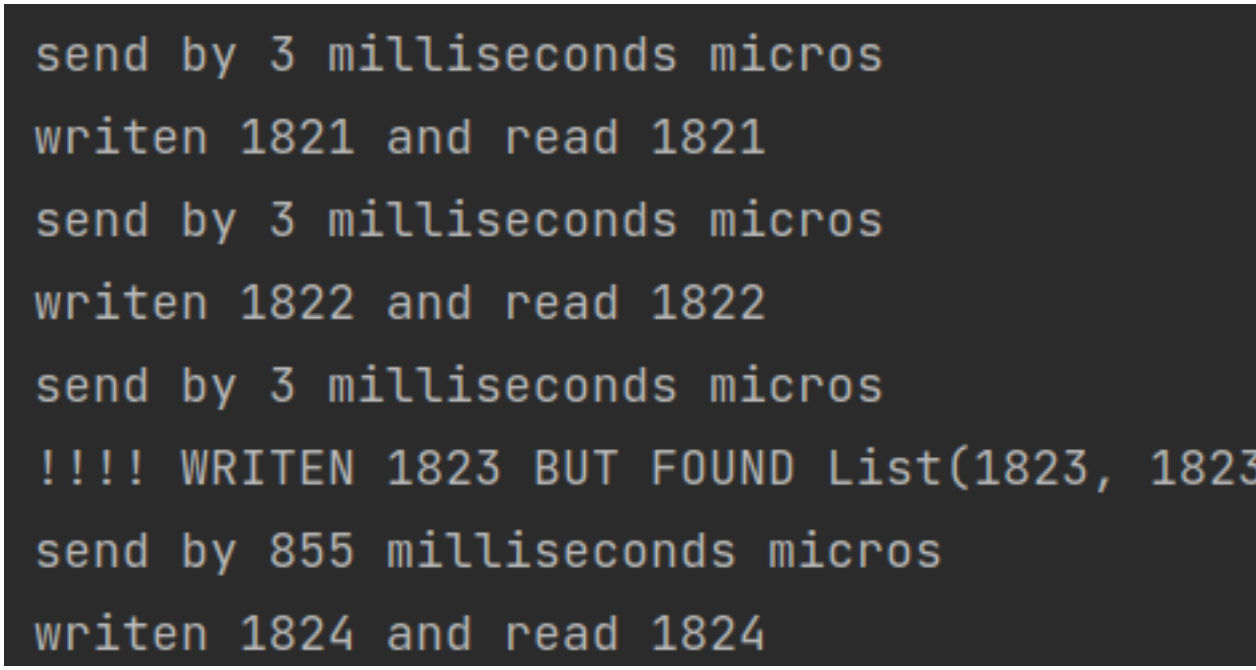
7 TEST2

ЦЕЛЬ: смоделировать падение лидера при записи с параметром acks=all. Убедиться, что true dataloss нельзя получить тем же способом при множественных попытках. СПОСОБ ДОСТИЖЕНИЯ: такой же как и в первом тесте.



Предыдущий тест, но при acks=all. Мы обнаружили все сообщения, которые писали, хоть и ушло 800 милисекунд на то, чтобы кафка перестроилась после смерти лидера и наш продьюсер продолжил писать.

Но в процессе данного теста был получен один интересный результат:



Это не случайность. Такое происходило систематично. Каким-то чудесным образом после смерти лидера мы обнаруживали в кафке сразу два сообщения!

Дело в том, что я лидер был убит в тот момент, когда он уже отреплецировал данные и был готов отправить мне сообщение о том, что все прошло успешно. Не поймав такого сообщения, продьюсер решил попробовать отправить сообщение ещё раз (retry при неуспешной записи). Речь идет о следующем параметре:

message.send.max.retries	3

8 ТЕСТ 3

ЦЕЛЬ: убедиться в том, что параметр message.send.max.retries приводит к дублированию сообщений в топике. СПОСОБ ДОСТИЖЕНИЯ: выкручу параметр в 0 и попробую повторить встреченное дублирование. Выставляем параметр на 0, многократно повторяем предыдущий эксперемент (10 раз) и ни разу не обнаруживаем дублирования данных. (до этого дублирование случалось крайне часто. Примерно 50

Кстати, что еще интересно, так это то, что при acks=1 мы не можем получить два одинаковых сообщения в топике.

Предположим, что лидер упал перед отправкой нам ответа. Но мы знаем, что он еще не отреплецировал данные. Поэтому нашего сообщения X не будет в других брокерах в тот момент, когда лидер упадет. Далее, поскольку ответа мы не получили, делаем ретрай, но уже не на бывшего лидера, который упал, а на нового. Там появляется сообщение X в единственном экземпляре, все ок.

9 Проблема таймаутов.

Одна из существенных проблем, которая возникает при параметре acks=all - необходимость постоянной синхронизации на каждом шаге записи. То есть, мы не можем отправлять сообщения быстрее, чем реплики ходят на лидера, чтобы синхронизироваться.

Здесь интересны следующие параметры брокеров: $replica.lag.time.max.ms$ - максимальное время, которое реплика может не синхронизироваться, оставаясь ISR. $replica.fetch.min.bytes$ - минимальный объем байт изменений, который должен произойти на лидере, чтобы реплика синхронизировалась. $replica.fetch.wait.max.ms$ - спустя это время, даже если байт изменений недостаточно, реплика все равно пойдет синхронизироваться.

10 ТЕСТ 4

ЦЕЛЬ: Показать, что при низком таймауте на запись у продьюсера, если приведенные выше 3 параметра будут очень большими (или вообще, если реплика долго не синхронизируется), можем получить дублирование данных.

Выставляем три параметра очень высокими. Видим следующую картину: из 10 записей 10 failed. При этом, спустя $replica.fetch.wait.max.ms$ обнаруживаем все 10 записей в системе. Если бы мы использовали retry=2, обнаружили бы их там 20.

11 Exactly Once Delivery

Вообще говоря, порядок доставки сообщений - тоже данные, некоторое ординальное число, которое может быть искажено . Но в рамках данной работы хотелось бы заострять внимание на порядке доставки сообщений, а сосредоточиться на содержимом. Будем считать ради простоты, что топик - неупорядоченное множество. Намного интереснее мне было бы сосредоточить свое внимание на проблеме доставки.

Проблема состоит в том, что наличие положительного ответа о записи от системы является критерием, а acks=all обеспечивает лишь достаточность наличия данных (то есть с параметром acks=all нам достаточно положительного ответа, чтобы быть уверенными, что данные оказались в системе, но наличие такого ответа, вообще говоря, не является необходимостью). То есть, отсутствие ответа о записи должно означать, что

данных в системе нет.

Лидеру приходят данные, нужно записать. Он записал, отреплецировал, отправил ответ, но, к сожалению, ответ потерялся. Мы думаем - наверное что-то пошло не так, отправляем данные ещё раз. Но на самом деле данные вполне таки записались, даже отреплецировались.

Или другая ситуация: сервер записал данные, отреплецировал. Хотел было отправить пакет, но произошла какая-то непредвиденная ситуация, сервер упал - и продьюсер понятия не имеет о том, что пакет был успешно записан и отреплецирован.

В общем, отсутствие ответа от сервера - не может гарантировать ничего. Мы не можем знать, записались ли данные или нет. Именно поэтому, как только мы не получили от сервера ответа, мы не можем принять решение о том, как действовать дальше: попробовать записать данные ещё раз, или ничего не делать. Можно, конечно, пойти и проверять самостоятельно, оказались данные на сервере или нет, но это оверхед.

Эта проблема называется Exactly once delivery. В противовес exactly once стратегии имеются стратегии at-least-once и at-most-once.

При at-least-once мы отправляем на брокера сообщения до тех пор, пока не получим удовлетворительный ответ. Таким образом там может накопиться сколь угодно много сообщений, но хотя бы одно будет, что для нас особенно важно. Это отражено в случае с acks=all и ретраями.

При at-most-once мы просто один раз отправляем сообщение. Оно может не дойти, но мы уверены в том, что в нашем хранилище не окажется более двух таких одинаковых сообщений. Это имеет смысл, когда полное отсутствие каких-то данных в кафке намного лучше, чем несколько таких реквестов (например если речь идет о банковских операциях, намного лучше, если операция вообще не пройдет, чем если пройдет дважды). Здесь речь об acks=1 и ретраями.

12 Как решается проблема с Exactly Once Delivery.

Очевидным образом, Exactly Once Delivery - невозможно.

Доказательство здесь крайне простое. Базируется на той идее, что при записи в систему мы ожидаем получить от нее сообщение, ибо без такого сообщения не можем сделать вывод о том, что запись произошла, и остановиться. Но отсутствие такого сообщения не может значить ничего, как я уже писал ранее. Если такое происходит, мы уже не можем придумать алгоритм, который бы позволил принять решение, как действовать дальше. Этот алгоритм во всяком случае требовал бы от нас

спросить у системы, дошло ли до неё сообщение. Но будем считать, что субъектом доставки является доставщик, который лишь умеет генерировать новые сообщения и получать ответы. Как решить проблему?

Решение на поверхности. Если отправка сообщения это некоторое отображение состояния системы, то почему бы не сделать это отображение идемпотентным, а затем исходить из кейса at-least-once delivery, то есть acks=all и ретраи.

13 Идемпотентность в кафке.

На самом деле есть некоторое продолжение идеи Exactly Once Delivery. Называется Exactly Once Processing.

Мы едва ли можем обеспечить точно одну доставку, но у нас есть возможность сказать системе то, что ей не следует обрабатывать дважды одно и то же сообщение. Именно это позволяет получить флаг $enable.idempotence = true$.

14 TEST5

ЦЕЛЬ: проверить, что идемпотентность позволяет избежать дублирования при ретраях и acks=all

Мне опять таки не удалось получить дублирования при включенном параметре enable.idempotence = true. И тем более не удалось обнаружить отсутствие. Готов сделать вывод, что флаг idempotence=true с acks=all и множеством retry обеспечивает высокую устойчивость kafka к аномалиям доставки сообщений при падении лидера партиции.

15 Недостатки идемпотентности.

В статье [1] приводятся причины, почему идемпотентность в Кафке - это не всегда хорошо. Дело в том, что из-за идемпотентности сообщения могут стать крайне тяжеловесными. Это может быть незаметно, относительно, если сообщения большие. Но если система заточена на обработку большого числа маленьких сообщений, то идемпотентность может существенно понизить пропускную способность системы.

15.1 Результаты:

Список литературы

k1 https://mvnrepository.com/artifact/org.apache.kafka

[0] [1] https://hevodata.com/blog/kafka-exactly-once-semantics/