1 Введение.

Цель данной работы состоит в том, чтобы на примере распределённой системы Apache Kafka убедиться в том, что различная конфигурация параметров некоторой системы способна менять её положение в рамках CAP модели. То есть, в зависимости от выбранных параметров, система может становиться AP, CP и так далее.

Также, хотелось бы на практике убедиться и эмпирически уяснить, почему для распределенных систем имеет место быть CAP теорема, утверждающая, что распределенная система одновременно не может быть доступной, консистентной и устойчивой к расколам в сети.

2 Apache Kafka.

Apache Kafka - пример распределенной персистентной системы с состоянием, которую я буду исследовать.

Состоянием в ней являются топики, партиции и данные, которые находятся в данных партициях.

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 $org.apache.kafka$ на языке Scala. 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)

5 Принцип работы системы Apache Kafka:

В данной системе присутствуют узлы двух типов: Zookeeper и Kafka Broker. Можно мыслить их как процессы, запущенные на некоторых устройствах программы. В целом такая система называется кластером. В системе есть непересекающиеся топики (логические хранилища данных). Топик состоит из партиций, но в рамках данного исследования я буду считать, что топик = партиция, поскольку тот факт, что топик состоит из нескольких партиций - никак не сказывается на нашем исследовании. Несколько партиций лишь обеспечивают возможность параллельной агрегации событий, параллельной записи (каждая партиция файл, топик можно разбить на несколько файлов, с которыми можно эффективно параллельно работать, но в этом параллелизме нет конкурентности).

Zookeeper является ядром кластера. Именно благодаря нему брокеры узнают друг о друге.

Также важно понятие контроллера. Контроллер выбирается в системе Zookeeper и управляет кластером (определяет, кто будет лидером раздела, если нет других лидеров).

На каждую партицию назначено один или несколько брокеров из всего кластера. Среди назначенных должен быть лидер (должен быть, но его может не быть, что мы увидим позже), а у лидера фоловеры. (если включены репликации).

С системой вообще говоря можно взаимодействовать двумя образами: запись и чтение.

Запись.

Операция записи предполагает два операнда: название топика (в нашем случае из одной партиции) и сообщение. Также операция должна быть ассоциирована с некоторым конкретным брокером (адрес брокера в сети: IP и port).

Далее есть несколько вариантов.

1) Мы обратились к лидеру партиции. Он запишет данные в свой файл и асинхронно отправит фоловерам запрос записать изменения в свои файлы. В зависимости от параметра acks - он отправит продьюсеру ответ.

2) Мы обратились к другому брокеру. Он рекурсивно перенаправит запрос лидеру партиции.

Чтение.

Интерфейс почти как у записи, но мы должны передать offset. Некоторую позицию в топике, сообщения начиная с которой мы хотели бы получить. Для того, чтобы система вернула ответ, один из брокеров, а именно привязанный к данной партиции, должен считать данные из файла. То есть, это должен сделать либо лидер, либо фоловер.

6 Что вообще такое in-sync replica?

С прикладной точки зрения мы можем сказать, что in-sync replica - это брокер, прочитав данные из которого, мы можем быть уверены, что они являются новейшими. То есть между последней записью в брокере и нашим прочтением не было успешных записей в систему. Можно свести это к следующему: лидер всегда обладает самой современной информацией, поскольку любая запись в систему так или иначе проходит через него. Тогда in-sync replica - просто напросто тот брокер, данные в котором совпадают с лидером.

Но встаёт вопрос: в какой момент брокер перестаёт быть in-sync, в какой момент он становится in-sync, и кто вообще решает, является ли он in-sync, когда мы пытаемся прочитать данные из системы.

Скажем, что произойдёт, если между записью лидером в партицию и тем, как он отправит фоловерам запрос на запись - лидер упадёт?

Если бы лидер, получив запрос на запись, сразу попытался вписать в свой файлик все необходимые данные, а лишь потом вызвал бы соответствующую процедуру у реплик, то мы могли бы заметить окно, между записью лидером в файл и отправкой процедуры репликам. В течение этого окна лидер мог упасть, что привело бы к тому, что данные в лидера уже были записаны, а реплики при этом не получили данной информации.

Можем ли мы после смерти лидера избрать нового? Мы обязаны это сделать, иначе какой вообще смысл в репликах! Но данные, которые поступили на лидера, будут потеряны.

И всё таки перед тем как дальше копать этот вопрос, попытаемся понять, какие гарантии нам способна предоставлять Kafka, когда мы производим запись.

Ведь если нет гарантий, то мы не можем вести речь о потере каких-то данных. Ведь Kafka не дает обязательств, что данные окажутся в системе, после того, как мы сделаем запрос на запись. Однако, Kafka способна

в ответном сообщении уведомить нас о том, что запись прошла успешно. И вообще говоря, такое уведомление можно посчитать *КРИТЕРИЕМ* записи, что означает, что отсутствие уведомления должно означать отсутствие данных в системе, а наличие положительного ответа - наличие их там.

Потерей же данных назовем обнаружение отсутствия данных в Kafka, при условии, что между записью этих данных и последним чтением не происходило явной попытки их удалить, и в момент записи этих данных Kafka сообщила нам о том, что данные были успешно записаны.

В следующем разделе попробуем точнее определить, что будем считать потерей данных. (речь идет о консистентности)

6.1 Правило acks.

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

Как я сказал ранее, 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. Возможно, наш пакет вообще не дошёл до сервера, или сервер каким-то образом его упустил. Могло произойти всё что угодно. 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. Из моих предыдущих рассуждений, вообще говоря может быть так, что acks=1 недостаточно, чтобы быть уверенным в успешном размещении данных. Ведь, вообще говоря, есть два варианта, которые я привёл выше.

Рассмотрим подробнее, как может работать система при параметре acks=1.

1) Клиент отправляет запрос на запись

2) Лидер пишет к себе в файл обновляет свои данные.

3) Лидер уведомляет другие реплики о том, что их данные устарели.

4) Лидер отправляет ответ об успешной записи данных.

5) Лидер начинает репликацию.

В данном случае лидер мог упасть между 4 и 5 шагом. То есть, клиент был уведомлён о том, что данные записаны, но при этом они становятся недоступными. Ведь единственный брокер, который знал об этих данных, а именно лидер - пал. То есть, мы потеряли availability на ровном месте (хотя, казалось бы, репликации должны обеспечивать доступность!!!). Такой распорядок вполне можно считать недопустимым, а потому я не ожидаю увидеть его в тестах.

Есть альтернативная последовательность шагов, намного более разумная и вероятная:

1) Клиент отправляет запрос на запись

2) Лидер пишет к себе в файл обновляет свои данные.

3) Лидер отправляет ответ об успешной записи данных.

4) Лидер начинает репликацию.

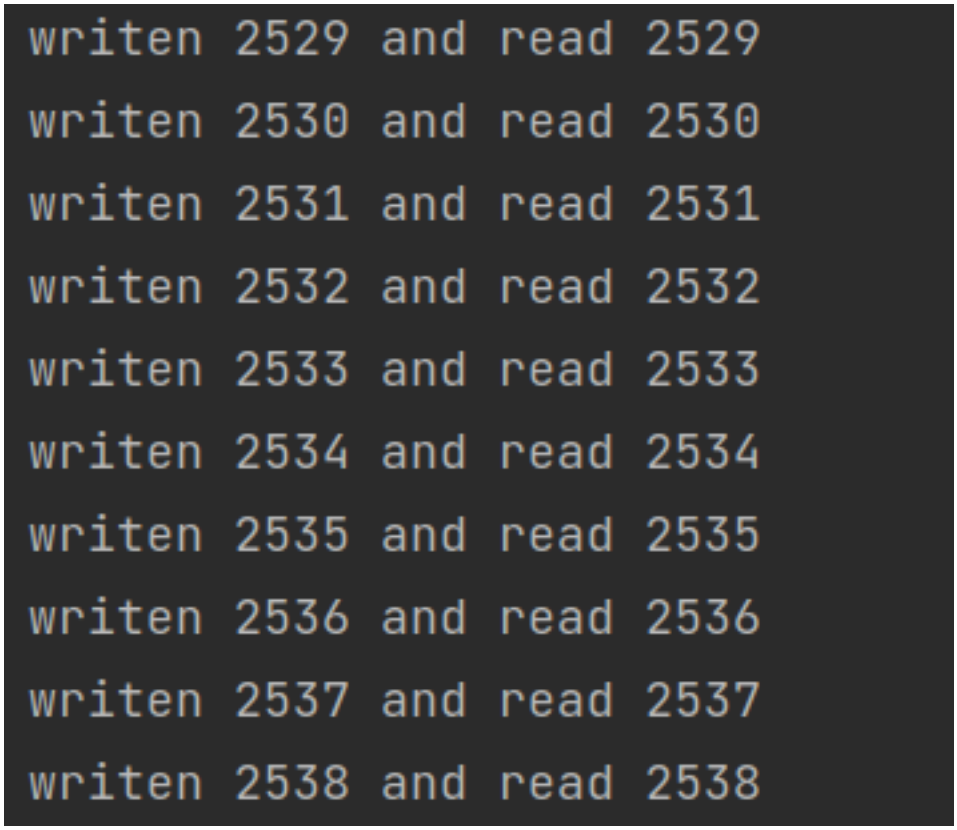
Что происходит здесь? На самом деле, я просто исключил шаг с объявлением данных устаревшими. Когда происходит объявление данных устаревшими? Об этом позже. Но пока остановимся на том, что оно происходит не в момент записи.

Теперь, если лидер упадёт между 3 и 4 шагом - у нас останутся in-sync реплики, которые, правда, не будут знать о новых данных, которые лидер уже записал себе, даже уведомил об этом продьюсера. Но, увы, данные были потеряны вместе с лидером. Поскольку мы оставили в in-sync наши реплики, среди них найдётся новый лидер, правда, без тех самых данных. На лицо true data loss.

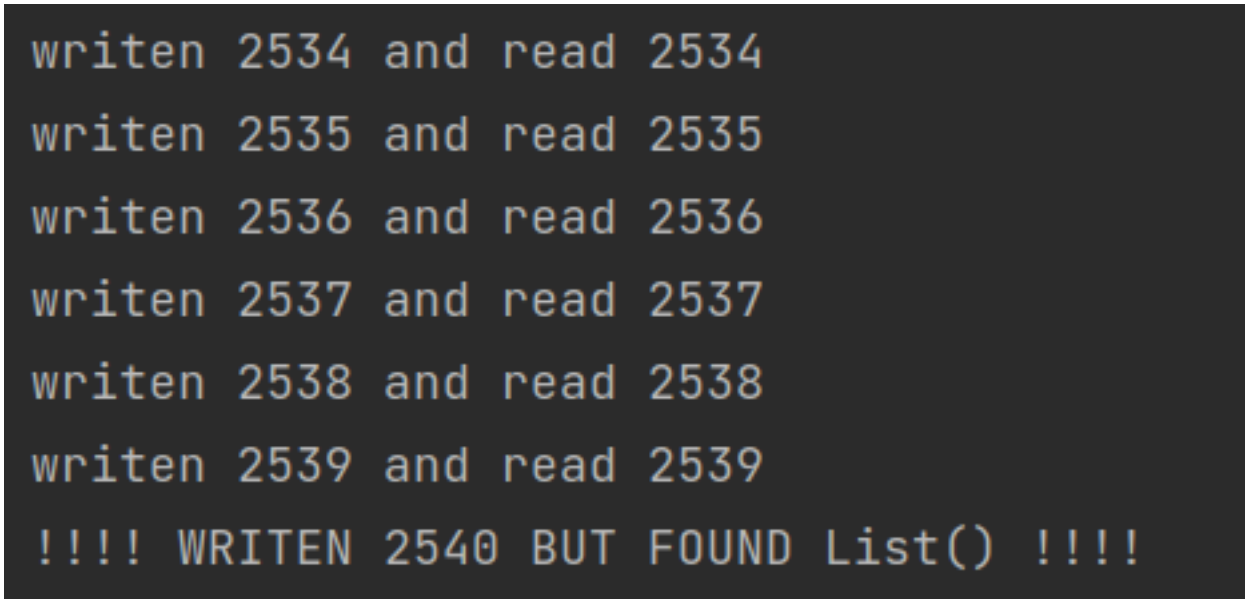
7 TEST1

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 я безостановочно делал в кафку 10000 записей. После каждой записи я отправлял консьюмера читать, что же лежит в этом топике.

При этом, если лидером топика являлся брокер 1, я взаимодействовал только с брокером 3. В обычное время я имел следующие логи:



Это означало, что после записи числа x в кафка кластер через брокера 3, я выполнял чтение через брокера 3 и видел, что в кафке есть записанное мной число. Однако, затем я открывал консоль кубернетеса и дропал под, на котором работал лидер этой партиции. Практически с первого раза я получил в логах следующее:



7.1 Объяснение теста:

Судя по всему, 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.

7.2 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.

Также можно сделать вывод, что информация о том, являются ли реплики in-sync, хранится ну уж точно не у лидера. И судя по всему, рассинхронизация реплик происходит именно в тот момент, когда они не отвечают на запрос о реплицировании (в данном случае, вероятно, такого запроса даже не было)

8 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.

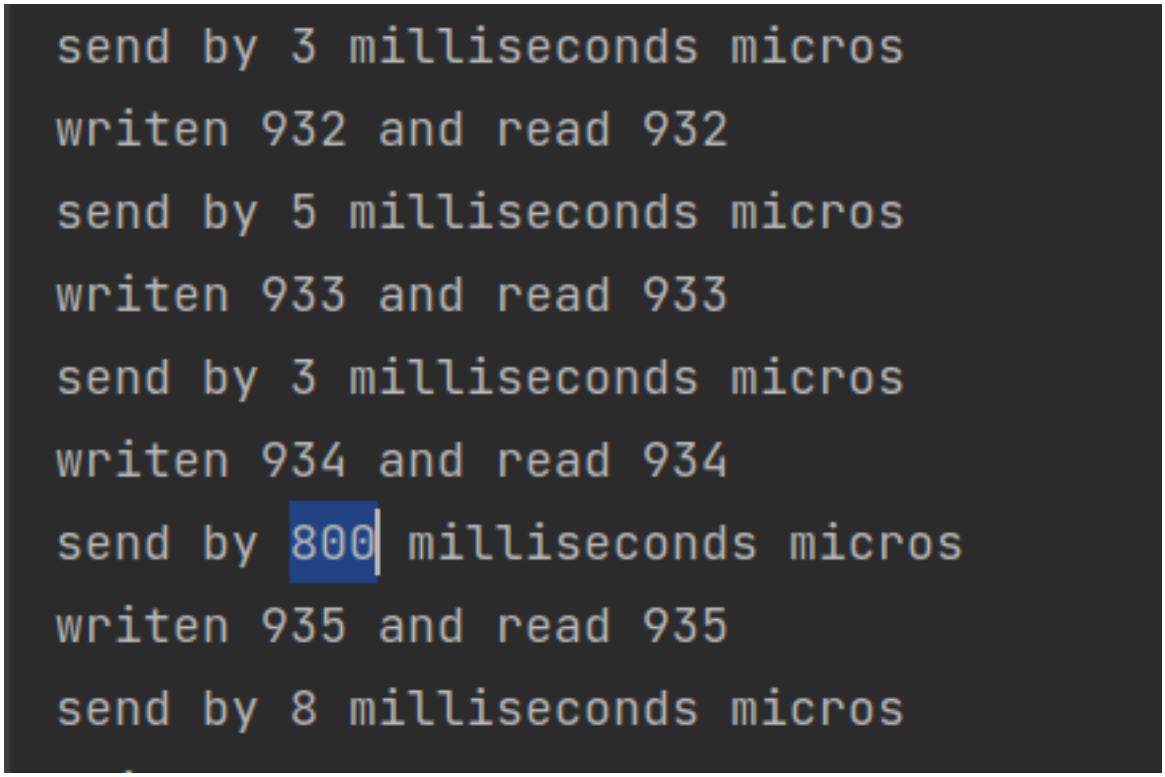
Если в случае acks=1 нам пришёл ответ, мы едва ли можем быть уверенны в том, что сообщение по-настоящему записалось в систему, что было подтвержено предыдущим тестом.

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.

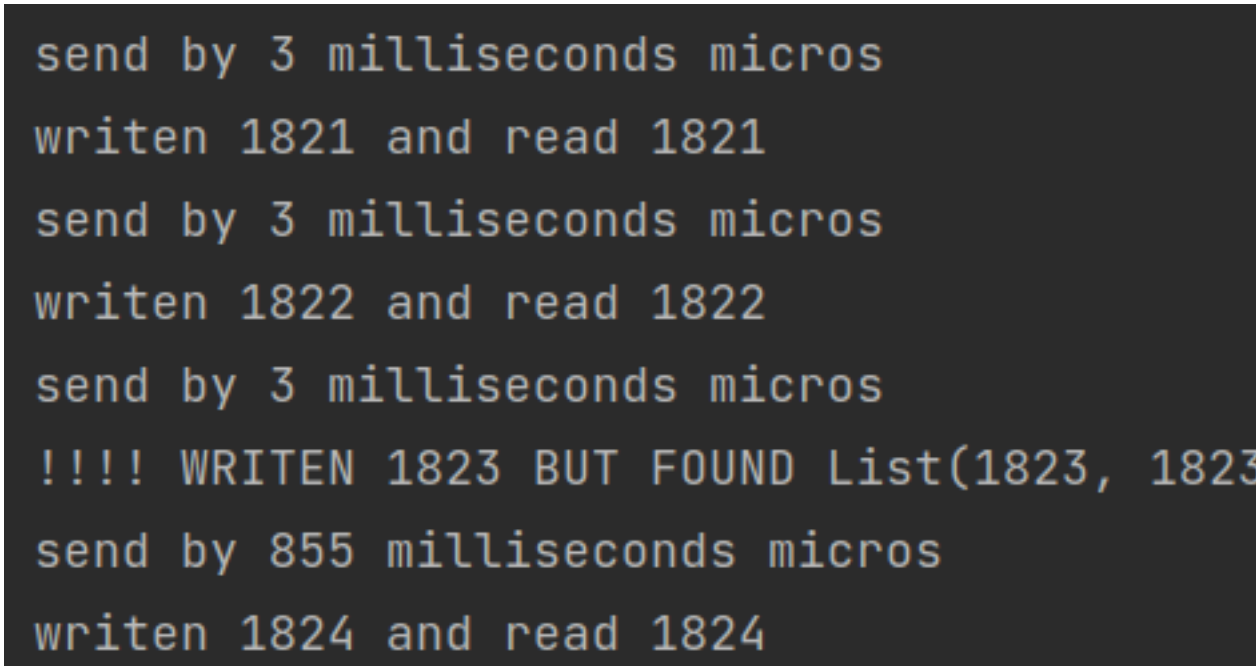
9 TEST2

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. Я модифицировал логи, добавив время, которое ушло на запись + чтение.



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 я получил один интересный результат:



Это не случайность. Мне удалось поймать такое не один раз. Каким-то чудесным образом после смерти лидера мы обнаруживали в кафке сразу два сообщения!

Довольно сразу сложно было понять, что происходит. Пришлось искать на stackoverflow похожие проблемы. Там я нашел ответ. Дело в том, что я положил под в тот момент, когда он уже отреплецировал данные и был готов отправить мне сообщение о том, что все прошло успешно. Не поймав такого сообщения, мой продьюсер решил попробовать отправить сообщение ещё раз. Так называемый retry. Речь идет о следующем параметре:

message.send.max.retries	3

10 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. Я изменил параметр на 0, многократно повторил предыдущий эксперемент (10 раз) и ни разу не поймал 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.

11 Exactly Once Delivery

Вообще говоря, порядок доставки сообщений - тоже данные, некоторое ординальное число, которое может быть искажено. Но в рамках данной работы я не хотел 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, как я написал ранее, 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. Просто пакет не дошёл.

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 TEST4

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.

14.1 Результаты:

Когда у нас есть три брокера, которые знают друг о друге, разделяют некоторую партицию, являясь репликами - мы можем не бояться падения лидера или падения любой другой ноды. А при параметрах acks=all, ретраях и включенной идемпотентности можем гарантировать то, что одно сообщение, которое сгенерировалось на стороне клиента, будет обработано ровно один раз на той стороне.

Таким образом Kafka может быть как минимум CA. То есть сохранять как доступность, так и консистентность, при этом не гарантируя устойчивость к расколам сети.