Министерство науки и высшего образования Российской Федерации
ФГАОУ ВО «УрФУ имени первого Президента России Б.Н. Ельцина»
Кафедра «школы бакалавриата (школа)»

Оценка работы _____
Руководитель от УрФУ Кошелев Г.Н.

Тема задания на практику

Исследование проблем доставки в распределенных системах на примере Apache Kafka

ОТЧЕТ
Вид практики Учебная практика
Тип практики Учебная практика,
научно-исследовательская работа (получение первичных
навыков научно-исследовательской работы)

Руководитель практики от предприятия (организации) Кошелев Г. Н

Студент Башкаров И.А.

Специальность (направление подготовки) 02.03.01 Математика и компьютерные науки

Группа МЕН - 290206

Екатеринбург 2021

1 Введение.

Распределенные системы имеют множество проблем и тонкостей. В данной работе хотелось бы сосредоточиться на одной из них: на проблеме доставки сообщений.

Проблема доставки состоит в том, что клиент, пытающийся изменить содержание распределенной системы, не всегда может быть уверен в том, каким состоянием будет обладать распределенная система после произведенной записи, основываясь лишь на отчете данной системы.

Могут возникать фантомные сообщения, то есть те, которых, по отчету распределенной системы в момент записи, быть не должно.

Могут также пропадать сообщения, которые, если судить по ответу систему, должны были стать её частью в момент записи.

Цель данной работы состоит в том, чтобы исследовать вопрос доставки в распределенных системах на примере Apache Kafka. Хотелось бы исследовать то, какие проблемы доставки встречаются в Apache Kafka, как они решаются с помощью конфигурирования системы.

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.

Система ZooKeeper позволяет брокерам обнаружить друг друга, а также выбрать контроллера.

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.

Лидер топика выбирается контроллером.

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-a.

Это позволит обеспечить некоторую базовую доступность.

4 План тестов:

Тестирование проводилось с кафкой, которая была развернута в kubernetes кластере (minikube), 3 zookeeper-a 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 эмулировалось 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 = all - 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 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.

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)

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.

НАБЛЮДЕНИЯ: Практически с первого раза был получен следующий результат: лидер ответил, что запись в Кафку прошла успешно, но потом, при попытке прочитать, сообщение не было обнаружено.

ОБЪЯСНЕНИЕ: Реплики приходят на лидера собирать данные с некоторой периодичностью. Когда лидеру пришло сообщение от продьюсера, он сделал запись к себе в файл, отправил продьюсеру успешный ответ. Но реплики еще не пришли к лидеру, чтобы скачать новые данные, это произойдет спустя некоторое время. Они остаются ISR все это время. И когда лидер внезапно падает, происходит переизбрание, мы выбираем нового лидера среди реплик, которые являются ISR, но при этом не содержат того сообщения, которое мы ранее записали в лидера.

4.2 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.

4.3 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.

5 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.

6 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.

НАБЛЮДЕНИЯ: Не было потерь сообщений. Однако встретилось дублирование сообщений.

ОБЪЯСНЕНИЕ: Есть гипотеза, что дублирование сообщений возникло из-за ретраев. Мы пишем на лидера, он записывает к себе, фолловеры приходят, чтобы загрузить себе свежие обновления, но лидер не успевает отправить сообщение об успехе продьюсеру, падает. Продьюсер из-за включенных ретраев идет на нового лидера (где, кстати, есть новое сообщение), пытается сделать туда запись повторно. В итоге в новом лидере два одинаковых сообщения.

7 ТЕСТ 3

ЦЕЛЬ: убедиться в том, что параметр message.send.max.retries приводит к дублированию сообщений в топике.

СПОСОБ ДОСТИЖЕНИЯ: изменим параметр ретраев на 0 и попробую повторить встреченное дублирование.

НАБЛЮДЕНИЯ: 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

ОБЪЯСНЕНИЕ: Лидер записал сообщение, реплики синхронизировались, он упал. Из-за отсутствия ретраев мы не пытаемся писать второй раз. Но это не решает проблему, ибо сообщение все же могло оказаться в новом лидере.

P.S.: Кстати, что еще интересно, так это то, что при acks=1 мы не можем получить два одинаковых сообщения в топике. То есть, если сообщение вообще хоть как-то оказалось в лидере, мы узнаем об этом раньше, чем произойдет синхронизация. А потому, если лидер упадет (не ответит нам успехом), мы можем быть уверены, что синхронизация не произошла.

8 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.

9 ТЕСТ 4

ЦЕЛЬ: Показать, что некоторым образом выбранные приведенные параметры могут позволить нам получить в системе фантомные сообщения.

СПОСОБ ДОСТИЖЕНИЯ: Мы должны получить ситуацию, когда время ожидания ответа от лидера на продьюсере закончится прежде чем все реплики успеют синхронизироваться, и лидер сообщит нам об удачной записи. Это достигается при следующих условиях: 1) replica.fetch.wait.max.ms > producer timeout Если это будет не так, реплики синхронизируются до истечения времени ожидания, и мы получим успешный ответ. 2) replica.lag.time.max.ms > replica.fetch.wait.max.ms Если это будет не так, метрика перестанет быть ISR, прежде чем данные отреплецируются, и лидер вернет нам успешный ответ. 3) replica.fetch.min.bytes > размер сообщения Если это будет не так, реплика, получив слишком большое сообщение, сразу же пойдет реплецироваться. Поэтому, параметр replica.fetch.min.bytes необходимо сделать достаточно большим. true НАБЛЮДЕНИЯ: 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.

ОБЪЯСНЕНИЕ: Записали на лидера, он не отвечает, пока реплики не синхронизируются. Реплики не синхронизируются очень долго, происходит таймаут. Мы считаем, что записи не было. Спустя некоторое время реплики все же синхронизируются. Мы обнаруживаем эти сообщения в каждой из реплик.

10 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 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.

11 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 было сказано 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.

12 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 $enable.idempotence = true$.

13 TEST5

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

СПОСОБ ДОСТИЖЕНИЯ: включить идемпотентность, ретраи и поставить acks=all

НАБЛЮДЕНИЯ: В тесте не удалось получить дублирования при включенном параметре enable.idempotence = true. Также не удалось обнаружить отсутствие. Можно сделать вывод, что флаг idempotence=true с acks=all и с множеством retry обеспечивает высокую устойчивость kafka к аномалиям доставки сообщений при падении лидера партиции.

14 Недостатки идемпотентности.

В статье 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.

14.1 Результаты:

В табличке:

LAG = replica.lag.time.max.ms
WAIT = replica.fetch.wait.max.ms
BYTES = replica.fetch.min.bytes
ФЗ = Фантомные сообщения
ДД = Дублирование сообщений
ПД = Потерянные данные
С = Стоимость отправки сообщений

	acks	retries	WAIT	BYTES	LAG	И	ФЗ	ДД	ПД	С
TEST1	1	true	> timeout	low	> timeout	false	false	true	true	low
TEST2	all	true	> timeout	low	> timeout	false	true	false	false	normal
TEST3	all	false	> timeout	low	> timeout	false	false	true	false	normal
TEST4	all	false	< timeout	very high	< timeout	false	false	true	false	normal
TEST4	all	false	> timeout	low	> timeout	true	false	false	false	high

Таким образом, можно заключить, что проблема доставки сообщений не является такой уж тривиальной, имеет множество деталей, которые следует учитывать. Конфигурация способна кардинальным образом повлиять на то, как будет проходить доставка сообщений. Например подходы at-least-once и at-most-once фактически дают нам две принципиально разных системы. В первой у нас есть гарантия того, что сообщение будет доставлено, во второй такой гарантии нет. Это меняет совершенно все.

И что немаловажно, не существует идеально конфигурации, такого набора параметров, который бы не имел недостатков. В процессе настройки распределенной системы перед нами всегда будет вставать выборы между двумя несовместимыми свойствами.

Список литературы

[1] https://mvnrepository.com/artifact/org.apache.kafka

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

[3] https://kafka.apache.org/documentation/

[4] https://github.com/apache/kafka

[5] https://www.confluent.io/blog/hands-free-kafka-replication-a-lesson-in-operational-simplicity/

[6] https://aphyr.com/posts/293-jepsen-kafka

[7] https://www.cloudkarafka.com/blog/apache-kafka-idempotent-producer-avoiding-message-duplication.html

[8] https://www.youtube.com/watch?v=M3HTM81P-Sg

[9] https://www.youtube.com/watch?v=A_yUaPARv8U

[10] https://www.youtube.com/watch?v=zMLfxztAVlo