SAGA pattern - Kirill Sereda - Medium

Предисловие

By Kirill Sereda

8 мин. на чтение · Посмотреть оригинал

Предисловие

> Один из лучших способов решить проблему распределенных транзакций между микросервисами — это полностью их избежать.
>
> Такое высказывание вы услышите не раз от опытных коллег, которые уже имели дело с сагой. Имел личный опыт Саги, скажу “Ну ее нафиг”.

> Но если такой возможности нет или распределенные транзакции ну очень уж нужны, тогда читаем дальше :)

Как и многие спикеры, начну с вступления и такого понятия как CAP теорема.

CAP теорема

Эти 3 правила лежат в основе CAP теоремы.

CAP теорема говорит нам о том, что в любой реализации распределенных вычислений возможно обеспечить лишь 2 из этих 3 свойств!

Таким образом может быть либо CP (Consistency и Partition tolerance), либо AP (Availability и Partition tolerance), либо AC (Availability и Consistency).

ACID

Теперь давайте вспомним что такое ACID из понятия реляционных БД.

Единая БД

Как мы знаем, в MSA (микросервисной архитектуре, сокращенно MSA) не должно быть такого, чтобы одна общая БД обслуживалась разными микросервисами. Ну такое себе, если честно, по ряду причин. Это самый настоящий антипаттерн — Shared DB (одна общая БД).

В MSA можно забыть про транзакции. Переход в MSA подразумевает отказ от привычных старых основополагающих вещей, к которым все привыкли.

Если у вас единая БД на все микросервисы — то это ужасный подход! Вы не сможете дальше расти и расширяться по всем правилам и канонам MSA.

Как делать правильно ? Очень просто: каждому сервису своя БД. Этот подход называется “**BataBase per service/Polyglot Persistence”.**

> Связей между несколькими БД быть не должно, только между сервисами.

Если рассматривать CAP теорему, то при нормальном подходе к MSA у нас по умолчанию используется подход **Partition tolerance** (устойчивость к разделению). Значит, согласно ее определению, нам надо жертвовать либо **Consistency** (согласованность данных) либо **Availability** (доступность).

Но как же так ? Нам надо и то и другое!

Значит надо придумать что-то к нашей БД, которая будет позволять и то и другое и третье. Звучит весьма круто, но выполнимо ли это?

2-фазный коммит

Есть такое понятие как 2-фазный коммит.

Двухфазный коммит помогает нам решить **проблему согласованности данных.**

Чтобы понять что такое 2-фазный коммит, вначале стоить рассмотреть такое понятие, как **распределенный коммит**.

Распределенный коммит — это процесс принятия каких-то атомарных изменений между как минимум двумя участниками транзакции.

Это коммит, который состоит из двух фаз.

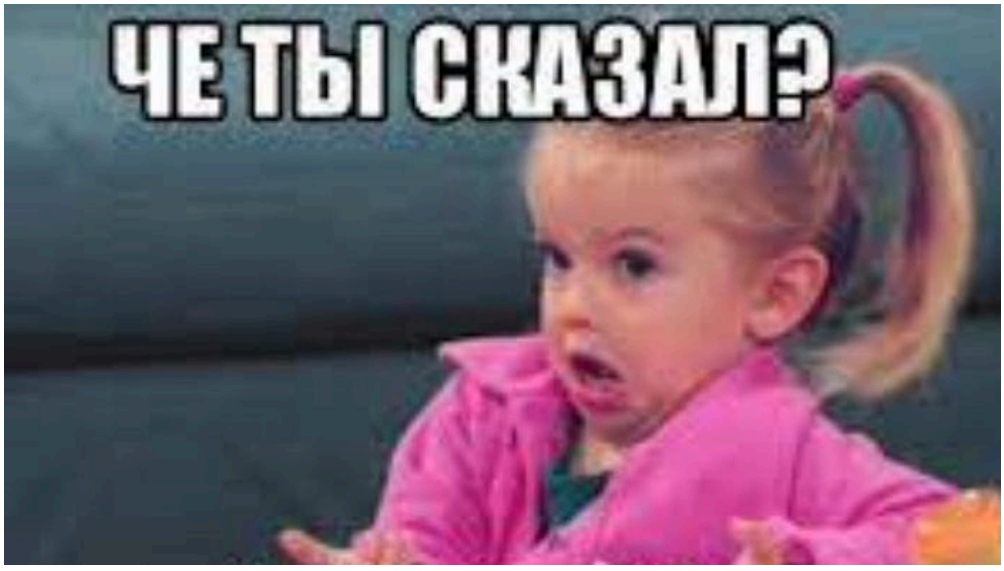
Первая фаза — это атомарная операция, которая производит проверку по возможности начала транзакции и затем происходит блокировка участников этого коммита.

Вторая фаза — сбор ответов от всех участников транзакции и применение этой транзакции.

Другими словами двухфазный коммит описывается как “atomic commitment protocol”. Т.е. все участники транзакции видят состояние **либо до совершения транзакции, либо после ее совершения** *без промежуточного состояния!!!*

Узел, выполняющий транзакцию, является **координатором**.

Алгоритм работы



Система будет согласована лишь в **начале** транзакции и в **конце** транзакции.

Минусы

Если координатор вышел из строя (отказал) после первой фазы, остальные участники не имеют информации о том, должна ли транзакция быть

зафиксирована или отменена (им придется ждать устранения сбоя).

Нам нужен механизм, который позволяет работать с различными БД как с единой БД (чтобы решить проблему с целостностью данных в распределённой базе).

> Я участвовал в конференциях талантливого человека Николая Голова, в которых он экспериментально показывает, что 2-фазный коммит не вариант, т.к. **чем больше вы горизонтально масштабируете сервера, тем больше падает производительность.**
>
> Давайте ему верить, он очень толковый парень.

Сразу скажу, следующий пример придумал он, а я просто нагло взял попользоваться, чтобы объяснить вам на пальцах. Николай, без обид :)

Например: представьте старый советский магазин — один продавец и толпа людей. Продавец контролирует все: сколько и чего взял покупатель, пробивает это все на кассе, упаковывает и отдаем ему готовый продукт. Так со следующим покупателем и со следующим и т.д.

Есть такое понятие как оптимистичный подход — это когда ведется контроль только для нарушений, а не за всем подряд (пример: современный супермаркет — магазин самообслуживания: пришел, взял тележку, выбрал товары, пошел на кассу, расплатился или еще лучше самостоятельно оплатил все в бесконтактном терминале оплаты. В итоге очередь гораздо меньше. Такую ситуацию можно заметить в современных магазинах.

Т.е. большинство операций завершаются успешно.

Дополнительные же действия производим уже по факту произошедшего сбоя. Т.е. уменьшаем издержки для большинства операций, что приводит к повышению производительности.

Смотрите, каждая БД в MSA может иметь свою пару в CAP теореме (CA, CP, AP).

Плюс ко всему вышесказанному могут быть проблемы в сети, такие как: ненадежная сеть, большая задержка передачи данных, небезопасная сеть, пропускная способность страдает и т.д.

Здесь и появляются многие проблемы.

Один сервис делает запрос на второй, тот на третий, а тот на четвертый и в итоге он обращается к БД. В результате у вас что-то пошло не так (между третьим и четвертым) и весь ваш запрос будет неуспешным. Я надеюсь вы понимаете почему.

Что же такое сага ?

Сага — паттерн, который представляет собой набор локальных транзакций.

Вся суть состоит в следующем: ***каждая локальная транзакция обновляет БД и публикует сообщение или событие, инициируя следующую локальную транзакцию в саге.***

Т.е. **если транзакция завершилась неудачей, то сага запускает компенсирующие транзакции, которые откатывают изменения, сделанные предшествующими локальными транзакциями.**

Если на каком-то этапе случается ошибка то запускается серия компенсационных транзакций, которая откатит предыдущее изменение.

Обязательные условия Саги

1) Ключ идемпотентности

Каждому шагу назначается ключ идемпотентности. Если что-то случится, вы должны иметь возможность повторить событие по этому ключу.

Каждый сервис, который будет по цепочке получать запрос должен уметь понять, он раньше

обрабатывал этот запрос или нет. Если он его раньше обрабатывал (проверка происходит по этому ключу идемпотентности) — то он игнорирует запрос, а если нет — то выполняет.

2) Надежный канал доставки

Событие должно быть доставлено как минимум 1 раз, возможность подписки, канал должен помнить о событиях некоторое время, если сервис упал он должен потом при возможности перезапросить все события из канала.

3) Компенсация

Должна быть возможность отменить шаги по ключу идемпотентности.

Если, например, 1–2 шаги сделаны успешно а на 3-ем произошел сбой, то Сага должна откатить все предыдущие шаги (1 и 2) для атомарности данных.

Виды Саги

1) По порядку вызова функции

2) По получению результата вызова функции

- **синхронная**: результат функции известен сразу. Сразу хочу отметить здесь минус в том, что сервис саг будет ждать пока не выполнится каждый шаг, а это лишняя нагрузка на производительность.

- **асинхронная**: вначале возвращается статус (например все успешно), а результат функции возвращается потом (например через обратный вызов API “сервиса саг” из клиентского сервиса).

Проблемы в Саге

Например вы читаете данные из БД, потом кто-то другой обновит эти данные а ваша исходная транзакция перепишет эти изменения.

Например в процессе выполнения Саги вы что-то записали в БД, кто-то другой эти изменения уже прочитал, а ваша Сага ведь еще не закончилась, и вы еще раз что-то запишите в БД, измените данные, и другая транзакция прочитает неверное состояние этих данных.

Например в течении одной Саги вы можете получать разное состояние данных.

Ну и как же с этими проблемами бороться ?

Можно каждый раз перепроверять, не изменилось ли значение.

Сработает ? Да, сработает. Но это же костыль. Да, это костыль :) Но работает же! :)

Свойства ACID хороши, когда речь идет о реляционной БД или, например, монолите.

Но что делать если транзакцию надо растянуть на несколько сервисов ?

Вот здесь и поджидает нас самое интересное и опасное.

Мы с вами выяснили, что каждый сервис имеет свою БД. Отлично. Значит нам важно сделать так, чтобы весь процесс был как единый транзакция!

Пфф, легче простого. Поехали.

Реализация саг: способы

Существует два способа координации саг:

А теперь более детально про каждый из них.

1) Хореография (Choreography)

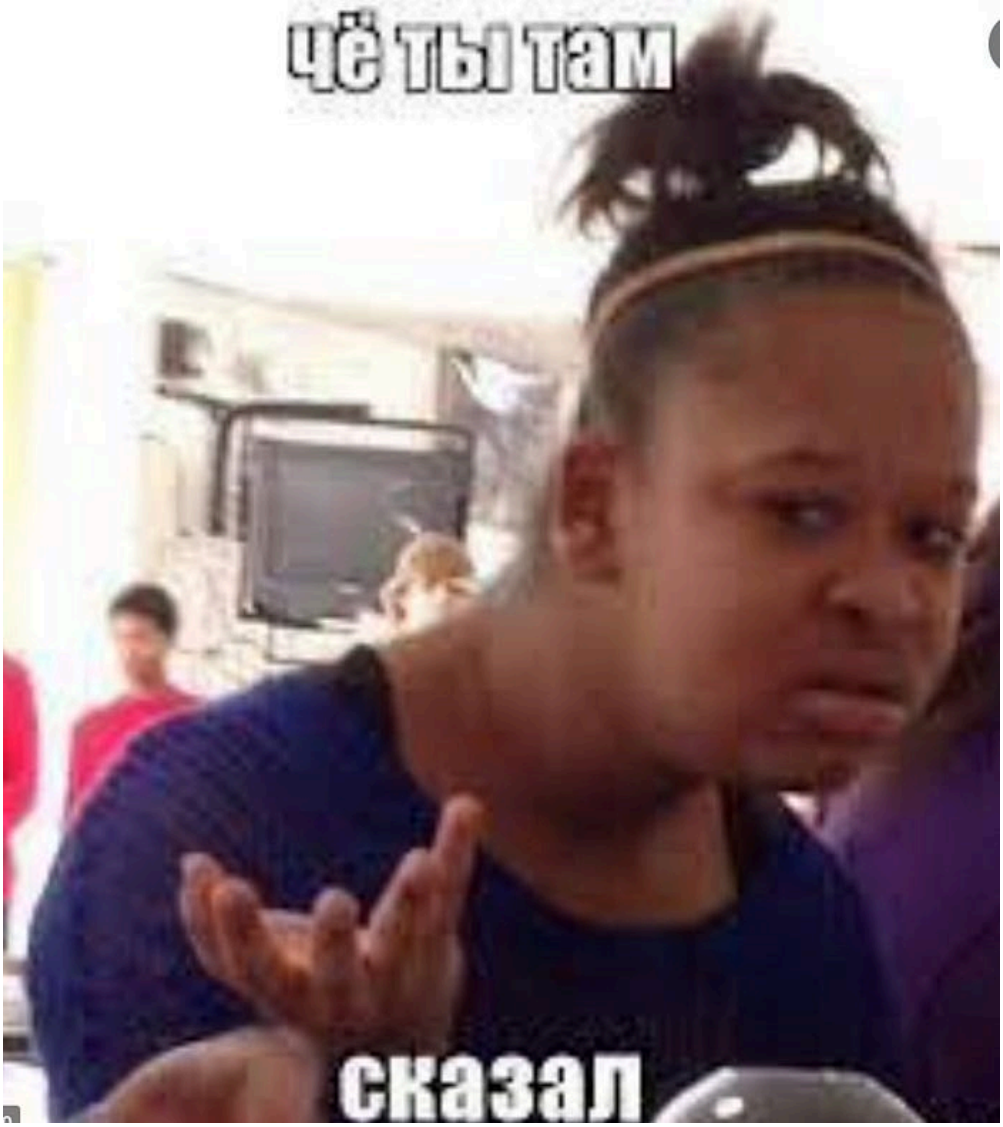
Например, используем канал доставки — любой брокер.

У нас есть объект транзакции, который обращается к сервисам.

> Сервисы сами вписываются в Сагу, т.е. сервис вписывает логику запуска в класс саги.

Каждый сервис контролирует окружение вокруг самого себя. Т.е. нету какого-то объекта отдельно, который мониторит, все ли там в порядке.

Сага мониторит всю последовательность шагов и может как-бы “добавлять” события в шину (например: отмена, retry, ping, health, check status или еще что).



Плюсы

Минусы

Какой мы можем сделать из этого вывод ?

Если мы создаем MSA с нуля без легаси кода — Сага это отличный выбор, нежели переписывать монолит и внедрять это вот все.

С монолитом можно, по крайней мере не запрещено, но опасно, в Сагу внедрять сверхзагруженную логику из старого легаси (очень сложно будет отлаживать это все дело, поверьте). Я один раз столкнулся с этим, больше не хочу (вместе с тестировщиками плачу в сторонке).

Важный момент:

Например выполняется большая цепочка вызовов и в конце что-то пошло не так, есть смысл добавить новую операцию “отмена”, чтобы не откатывать всю Сагу до самого начала.

Каждый сервис как-бы внедряется в сагу и указывает, обязательный ли он в Саге или нет и от кого он зависит (после кого он идет). Т.е. например если сервис1 указывает что он обязательный в Саге и он будет идти после выполнения Сервис0, то, если что-то пошло не так на этом шаге, Сага должна будет откатить изменения, а если сервис2 указывает что он не обязательный и будет идти после выполнения Сервис1, то в случае ошибки на этом этапе Сага пройдет дальше, отката здесь не будет.

Сервис1 не знает что за ним будет идти Сервис2, ему это не нужно. Он знает что он сам обязательный или нет и после кого он будет выполняться и все, на остальное ему плевать.

Следующий, например, Сервис3 указывает, что он обязательный и он знает что он идет после Сервис2, но он не знает про Сервис1 и Сервис0 ничего, ему это не нужно.

Как уже говорилось выше в случае сбоя транзакции **происходит другая транзакция, которая компенсирует предыдущую успешную транзакцию** до ее исходного состояния.

Здесь возможны два варианта использования сбоя транзакции:

1. **Сбой транзакции в сервисе 1 очень просто обработать**. В этом случае сервис 1 просто откатит свою локальную транзакцию и не опубликует никаких событий.

Например, транзакция в сервисе1 и сервисе2 прошла успешно, а транзакция в сервисе3 завершилась неудачно. На этом этапе сервис3 опубликует события сбоя транзакции, а предыдущие службы сервис1 и сервис2 будут прослушивать это событие и начнут компенсирующую транзакцию, которая откатит предыдущую транзакцию. Тот же процесс применяется на последней транзакции в каком-то по счету сервисе.

2) Второй вариант, например, когда где-то произошла беда, но **компенсировать действие саги мы не можем** (например отправка письма на почту). Что делать в таком случае ? В этом случае мы можем использовать второй тип рализации Саги — это **Оркестровка (Orchestration)**.

2) Оркестровка (Orchestration)

Оркестратор сообщает участникам, какие локальные транзакции выполнять. Оркестратор также называется “координатором саги” (сервис “владелец Саг”).

Если что-то пошло не так, то будет отрабатывать compensator. Если сервис не обязательный (как говорилось выше в первом подходе), то будем пытаться его компенсировать.

Как работает компенсация транзакций ?

Для каждой положительной транзакции мы должны описать обратные действия: бизнес-сценарий шага на случай, если что-то пойдет не так.

Но ведь может быть так, что компенсировать также не получится. Конечно может. Что же делать в таком случае ?

Компенсацию транзакций надо делать также как и положительные транзакции: retry + идемпотентные ключи (про которые писалось выше).

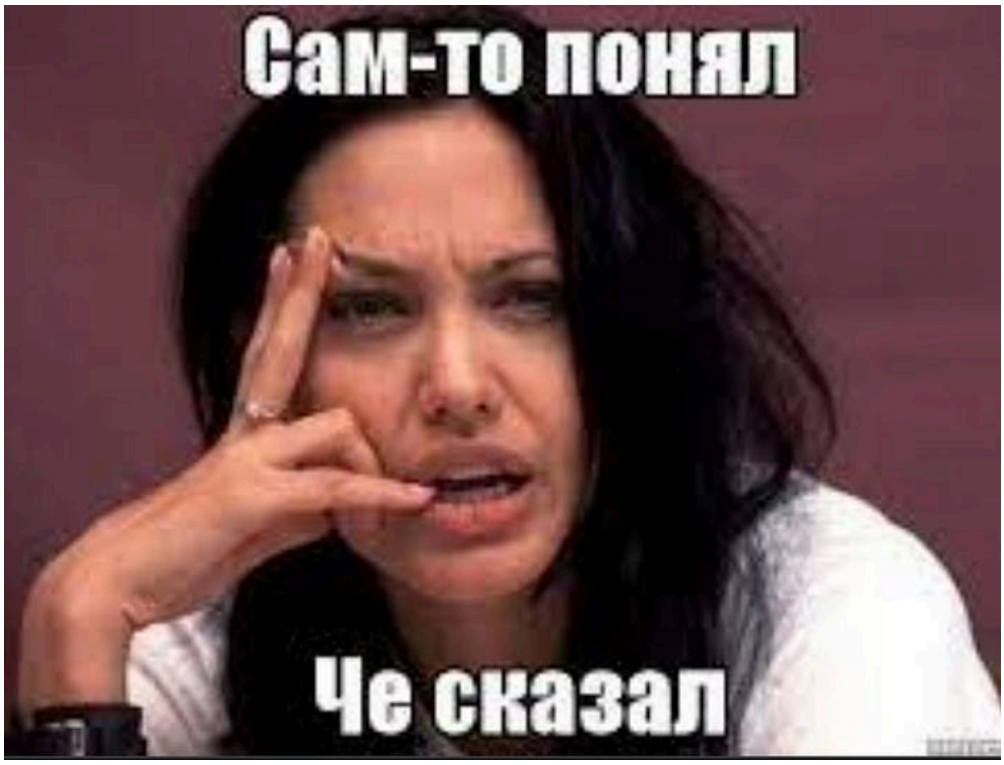
Если компенсация все же не удалась, можем вызвать владельца сервиса Саг и уведомить его о том, что Сага неудачная.

Например, у нас 5 сервисов в цепочке и **в первом сервисе (при первой транзакции) произошел сбой**. В таком случае сервис1 отправит ответ об ошибке транзакции в оркестратор. Оркестратор распознает, что транзакция не удалась конкретно для сервис1 и не будет предпринимать никаких дальнейших действий.

Если произошла **ошибка где-то в цепи посередине** (например в сервисе3): на этом этапе сервис3 отправит сообщение об ошибке транзакции в оркестратор. Оркестратор распознает, что транзакция не удалась для сервиса3. Также оркестратор знает, что предыдущие транзакции (для сервис1 и сервис2) были успешными, и отправляет команду отката на сервис1 и сервис2 для отката предыдущей успешной транзакции.

Пример оркестрации:

1. ClientService создает MovieOrder (Заказ) и создает CreateOrderSagaOwner (координатор Саги или сервис “владелец Саг”)
2. координатор Саги CreateOrderSagaOwner отправляет команду BuyTicket во второй сервис OrderMovieService.
3. OrderMovieService выполняет какую-то логику и отправляет обратно ответ.
4. координатор Саги CreateOrderSagaOwner получает ответ и отправляет BuyTicketApprove (одобрение заказа) или BuyTicketReject (отмена заказа) команду в ClientService
5. ClientService изменяет состояние заказа в *approved (подтвержден)* или *cancelled (отменен)*



Плюсы

Отличие Хореографии от Оркестрации

Если сервис Саг вдруг сломался то у нас все по прежнему будет работать. Он нужен лишь в том случае, когда надо выполнять откат действий.

Чем меньше логики в нем — тем быстрее и проще он будет работать.

Плюсы/минусы Саги

Еще раз хочу напомнить важное правило: если вы можете сделать свою логику без Саги, в таком случае не нужно делать Сагу, это будут лишние расходы (как говорится, не изобретайте велосипед).

Также немаловажным минусом является то, что модель программирования при реализации паттерна Сага становится более сложной.

Из существенных плюсов можно выделить то, что Сага позволяет приложению поддерживать согласованность данных между сервисами без использования распределенных транзакций. Но разве нужны эти плюсы когда есть настолько большие минусы и дикая головная боль ? Мое мнение — нет, но решать именно вам.