Домашнее задание по распределенным транзакциям

Цели

Реализовать сценарий интернет-магазина используя механизм распределенной транзакции

Решение

Реализация будет состоять из четырех сервисов, Склад, Платеж, Доставка и Заказ.

В сервисе заказ, собственно, и будет реализован механизм распределенной транзакции на основе оркестрируемой Саги.

Описание сервисов

Сервис Order – сервис для обработки заказа содержащий Сагу.

Сервис Storage – сервис для хранения и управления резервированием товаров. Все локальные транзакции в сервисе компенсируемые. Компенсирующие транзакции находятся здесь же.

Сервис Payment – сервис для обработки платежа по заказу. Все локальные транзакции в сервисе компенсируемые. Компенсирующие транзакции находятся здесь же.

Сервис Delivery – сервис для планирования доставки. Содержит поворотную транзакцию Саги. Если не удалось успешно запланировать доставку, то откатываются оплата и резервирование товара на складе. В реальных системах возможно поворотной транзакцией будет успешная оплата.

Повторяемых транзакций в моем сценарии нет.

Пользовательские стори для сервисов

Сервис Order

Создание заказа через POST /orders

1. Создание заказа через POST /orders
2. Для поддержки идемпотентности происходит проверка по X-Request-Id. Если запрос есть в хранилище, по нему достается заказ и происходит перенаправление на результаты.
3. Если запроса нет в хранилище создается новый заказ со статусами CREATED и происходит перенаправление на страницу с результатами заказов
4. Если запрос завершился неопределенной ошибкой, возвращается сообщение об ошибке.
5. Далее асинхронно происходит запуск Саги, который начинается с отправки сообщения на бронирование товаров в сервис Storage.
6. Сервис заказа слушает сообщения от сервиса Storage
   1. Если пришло сообщение об ошибке резервировании товара(ов), то сервис меняет статус заказа на STORAGE\_RESERVE\_ERROR
   2. Если пришло сообщение об успешном резервировании товара(ов), то сервис меняет статус заказа на STORAGE\_RESERVE\_SUCCESS, суммирует сколько нужно списать денег и отправляет сообщение в сервис Payment на проведение оплаты для указанного покупателя
7. Сервис заказа слушает сообщения от сервиса Payment
   1. Если пришло сообщение об ошибке оплаты, то сервис меняет статус заказа на PAYMENT\_ERROR и отправляет сообщение в сервис Storage на откат резирвирования товаров.
   2. Если пришло успешное сообщение об оплате, то сервис меняет статус заказа на PAYMENT\_SUCCESS и отправляет запрос в сервис Delivery на формирование доставки
8. Сервис заказа слушает сообщения от сервиса Delivery
   1. Если пришло сообщение об ошибке планирования доставки, то сервис меняет статус заказа на DELIVERY\_ERROR и отправляет сообщение в сервис Payment на откат оплаты и отправляет сообщение в сервис Storage на откат резервирования товаров.
   2. Если пришло успешное сообщение планировании доставки, то сервис меняет статус заказа на DELIVERY\_SUCCESS и завершает Сагу

Отслеживание результатов запросов через GET /results

1. Запрос возвращает выборку заказов из хранилища.

Сервис Storage

1. Слушает сообщения на резервирование от сервиса Order
   1. Получив сообщение пытается зарезервировать товар
      1. Если резервирование успешно, отправляет успешный ответ в сервис заказа
      2. Если резервирование неуспешно, отправляет неуспешный ответ в сервис заказа
2. Слушает сообщения на откат резервирование от сервиса Order
   1. Получив сообщение пытается откатить зарезервированный товар
      1. Если откат успешен, ничего не делает
      2. Если откат неуспешен, то пытается повторить N раз (тут возможны варианты, но не в рамках этого ДЗ, уже слишком объемно получается)

Сервис Payment

1. Слушает сообщения на оплату от сервиса Order
   1. Получив сообщение на оплату проверяет/пробует списать деньги покупателя со счета.
      1. Если списание успешно, отправляет успешный ответ о списании в сервис заказов
      2. Если списание неуспешно, отправляет неуспешный ответ о списании в сервис заказов
2. Слушает сообщения на откат оплаты от сервиса Order
   1. Получив сообщение на откат оплаты возвращает сумму по заказу на счет покупателя.
      1. Если откат успешен, отправляет успешный ответ в сервис заказов
      2. Если откат неуспешен, отправляет неуспешный ответ в сервис заказов

Сервис Delivery

1. Слушает сообщения на формирование доставки от сервиса Order
   1. Получив сообщение на доставку планирует доставку на указанное время
      1. Если успешно, то отправляет успешный ответ в сервис заказов
      2. Если неуспешно, отправляет неуспешный ответ в сервис заказов

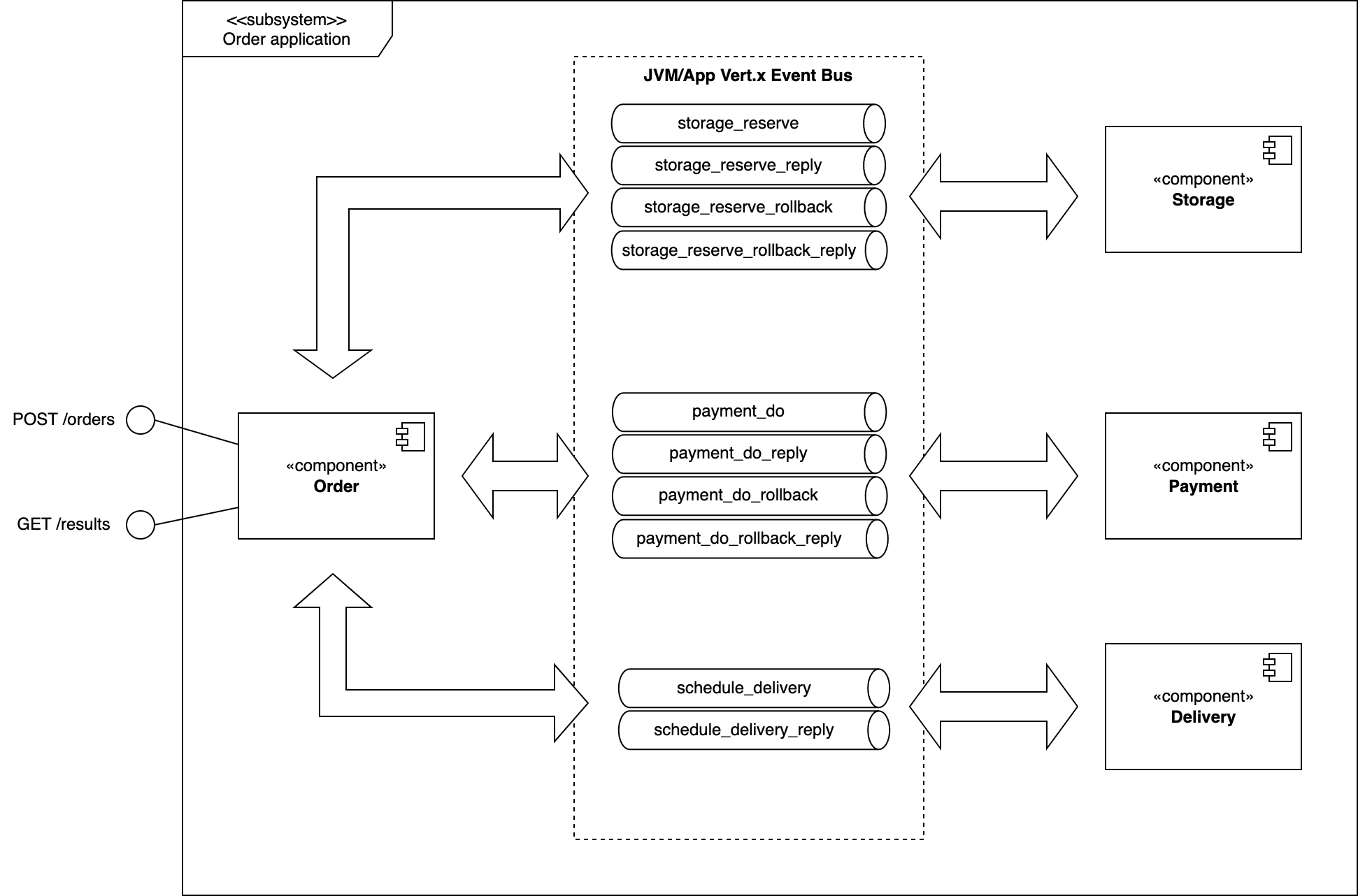
Описание решения

Ради экономии времени решение изначально спроектировано как один сервис-монолит с возможностью пошаговой декомпозиции на микросервисы.

Изначальная архитектура указана далее с возможными описанием шагов по дальнейшему разбиению.

Вариант 1

Сервис-монолит.



Все четыре компонента Заказ, Склад, Платеж и Доставка находятся внутри одного приложения с единым хранилищем данных. При этом ответственность разделена по компонентам ровно так как описано в пользовательских сценариях, общаются компоненты между собой по средствам событий через шину событий. По сути, тут используется частично модель акторов по средствам ее частичной реализации в используемом фреймворке Vert.x - четыре компонента это акторы, которые обмениваются сообщениями. Механизм распределенной транзакции в виде Саги полностью реализован в рамках компонента Заказ.

Все компоненты разделяют одно хранилище, но данные логически разделены, каждый компонент использует свой набор таблиц/сущностей.

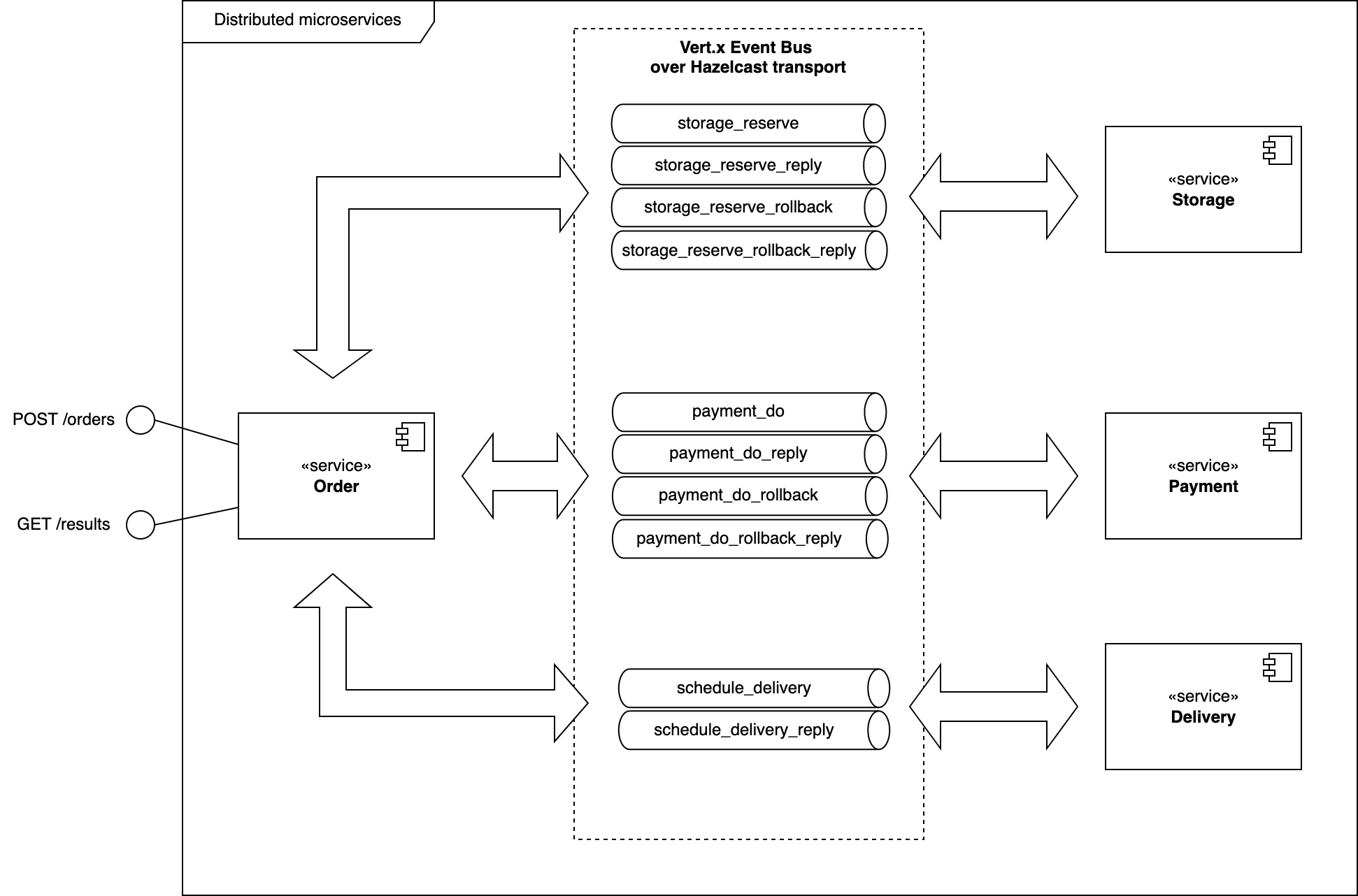
Каждый компонент/актор может горизонтально масштабироваться, подстраиваясь под кол-во ядер процессора.

Обработка запроса внутри компонента/актора потокобезопасна.

Там, где необходима согласованность данных между разными экземплярами компонентов/акторов они синхронизируются через хранилище.

Вариант 2

Разбиение на микросервисы



Так как внутри сервиса-монолита взаимодействие между компонентами уже происходит по средствам сообщений, то можно относительно легко выделить обозначенные компоненты в отдельные сервисы, а в качестве транспорта для шины событий Vert.x использовать любое коробочное решение для кластеризации [https://vertx.io/docs/#clustering](https://vertx.io/docs/" \l "clustering), например при помощи IMDG Hazelcast.

Общее хранилище разбивается на отдельные для каждого сервиса.

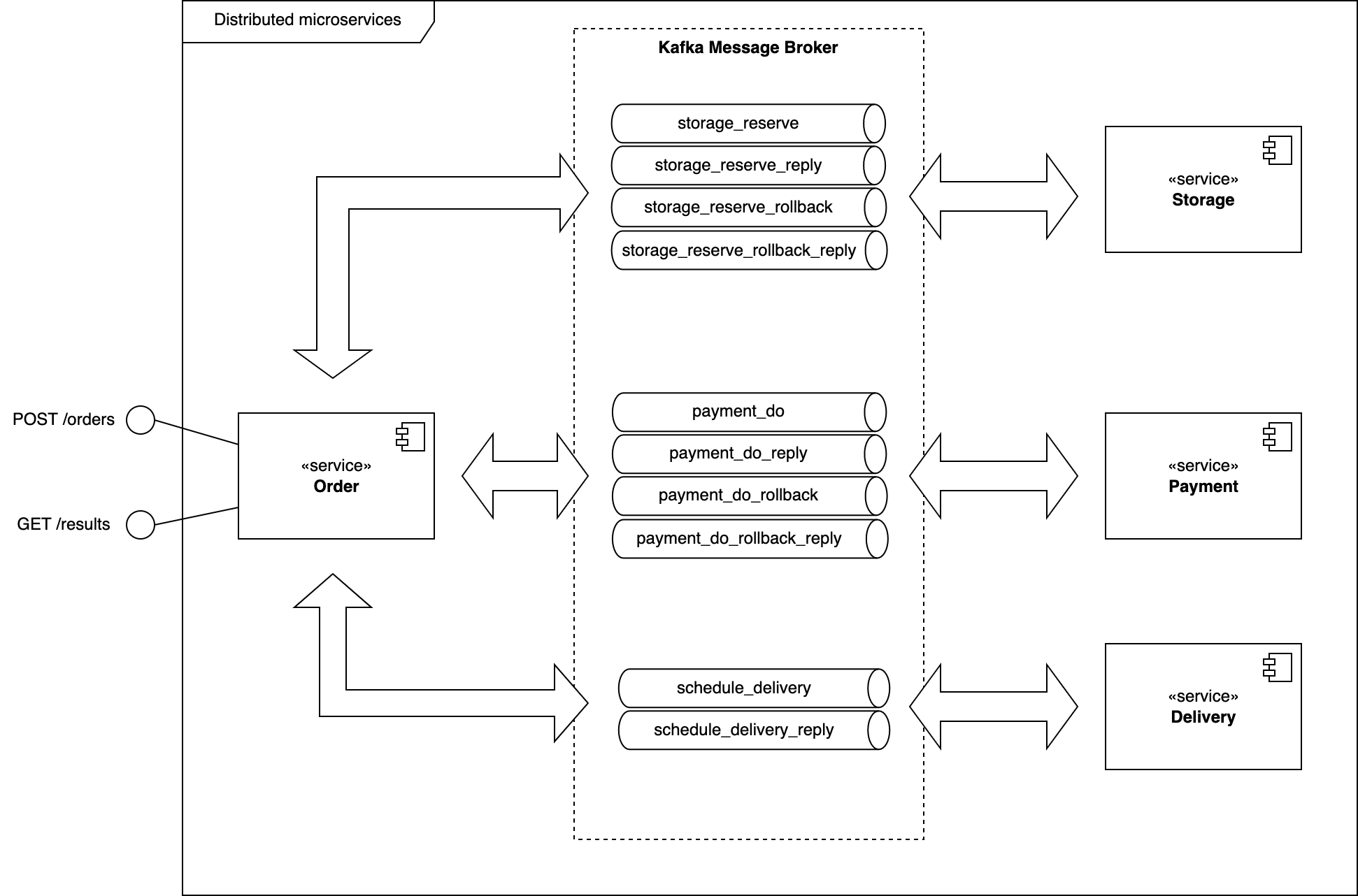
Сервисы могут быть горизонтально масштабированы. Там, где требуется согласованность данных между экземплярами сервисов при обработке запроса, данные синхронизируются через хранилище.

В первом варианте для простоты реализации некоторые модели переиспользовались между компонентами. В этом варианте предполагается так же и разделения моделей по сервисам, которые, по сути, должны являться ограниченными контекстами. То есть если сущность Заказ используется несколькими сервисами, то эта сущность должна быть продублирована и представлена в каждом сервисе своим набором данных. Общая модель должна быть только для сообщений которыми обмениваются сервисы.

Вариант 3 см. на след. стр.

Вариант 3

Разбиение на микросервисы



Коробочные решения для кластеризации на основе IMDG хороши своей быстротой и дешевизной в том плане что все данные хранятся в оперативной памяти, но на мой взгляд плохо подходят для системы, где требуется надежная доставка между узлами в случае их недоступности и т.д. Конечно же доставку сообщений между узлами можно отслеживать при помощи статусов, которые бы сохранялись в хранилищах узлов, и предпринимать какие-то компенсирующие действия, но лучше постараться улучшить надежность доставки сообщений. Собственно третий вариант предполагает заменить шину событий Vert.x на, брокер сообщений Kafka, с возможностью хранения и повторного чтения сообщений.