Реферат

по статье
In Reference to RPC: It's Time to Add Distributed Memory
Stephanie Wang, Benjamin Hindman, Ion Stoica
HotOS '21

Шарафатдинов Камиль БПМИ192

2021-05-25 15:16+0300

Дисклеймеры:

1. в этой статье я буду называть экзекьютором некоторый хост (или группу хостов) в распределенной системе, которая обеспечивает выполнение той или иной процедуры, но не занимается организацией и контролем перемещения данных и выполнения процедур.
2. почему-то у статьи невалидный doi, поэтому вместо цитирования вот ссылка:
   https://sigops.org/s/conferences/hotos/2021/papers/hotos21-s10-wang.pdf
3. Свои предложения по улучшению я буду писать в сносках, потому что их легче понимать в контексте проблемы, а не скопом в конце документа
4. в статье делается сильный акцент на асинхронность вызовов. Я считаю, что асинхронность здесь – дополнительная фича, реализация которой слабо связана с реализацией, собственно, распределённой памяти.

В этой статье предлагается добавить к RPC архитектуре общую память, с помощью которой можно снизить нагрузку на транспорт данных и скорость вызова процедуры за счет оптимизации копирования данных в некоторых случаях.

В популярных фреймворках RPC, таких как gRPC и Apache Thrift, аргументы можно передавать только "по значению". Это не проблема для многих приложений, где объёмы передаваемых данных небольшие. Но для data-intensive приложений, например, пайплайнов машинного обучения, объём передаваемых данных велик, но передача данных часто бывает излишней. Например, пусть есть две функции с такими прототипами:

```
U f(T t);
T g(X x);
```

если приложение хочет выполнить $u = f(g(x))$, ему сначала нужно выполнить $t = g(x)$, а потом $u = f(t)$. При этом t два лишних раза передается по сети (или другому транспорту) от экзекьютора к клиенту и в обратную сторону. При этом f и g могут быть функциями из разных фреймворков, например, Spark и TensorFlow, которые не интегрированы между собой, поэтому единственным транспортом, способным передать данные от одного к другому становится основное приложение.

Тут авторы предлагают ввести механизм ссылок на объекты, чтобы не передавать данные тем, кто явно этого не попросит. То есть в нашем случае прототипы выглядели бы так:

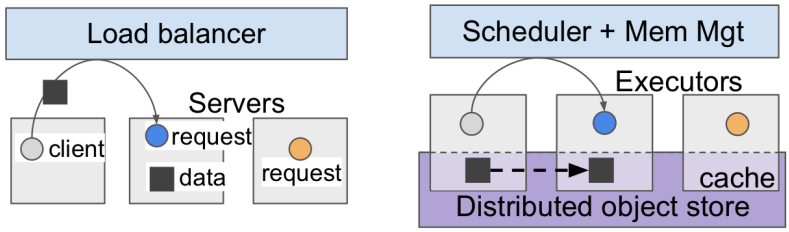
```
Ref<U> f(Ref<T> t);
Ref<T> g(Ref<X> x);
```

а вызов конструкции из клиента выглядел бы так:

```
Ref<X> x_ref(x);
Ref<T> t_ref = g(x_ref);
Ref<U> u_ref = f(t_ref);
u = u_ref.get()
```

Таким образом, мы получаем всего два копирования данных: один раз от клиента в RPC систему и второй раз из RPC-системы к клиенту. При этом RPC-система сама: либо перемещает t от экзекьютора g к экзекьютору g, либо назначает выполнение f на того же экзекьютора, что выполнял g.

На картинке видно, как выглядит общая схема передачи данных:



Теперь чуть более формально, про то, какими свойствами должно обладать API ссылок, которое позволит такие оптимизации и будет удобным с точки зрения разработки приложений:

Аллокация

Для того, чтобы работать с памятью, надо как-то эту память заполнять. В примере я использовал конструктор гипотетического класса `Ref<T>` и это удобство скрывает одну важную особенность: я не указывал куда и когда надо передать x для того, чтобы использовать его в вычислениях, куда и когда передавать данные от клиента решает RPC-система.

Деаллокация

Память не бесконечная, поэтому её надо иногда чистить. Для этого можно использовать систему отказоустойчивой сборки мусора, основанную на reference counting: например, если случились неполадки у клиента, держащего ссылку на какой-то объект, нужно

как-то обрабатывать сборку мусора. В общем случае, можно уменьшать количество ссылок на ссылку при явном запросе клиента: либо явным вызовом функции клиентского API, либо неявным вызовом из деструктора `Ref`. Последнее предпочтительнее – в этом случае RPC-система (в том числе её клиентское API) берет на себя все обязанности по управлению общей памятью.

Перемещение

RPC-система может автоматически обрабатывать перемещение либо экзекьюторов к данным, либо наоборот. Для того, чтобы эффективно перемещать данные, вводится механизм `Ref` vs `SharedRef`. Зачем нужны два типа ссылок? для того, чтобы система понимала, как перемещать данные: экзекьютор может не использовать свои зависимости (аргументы) напрямую, а передать их в другую процедуру. Для того, чтобы не гонять данные к такому прокси-экзекьютору, экзекьютор может указать, что он ожидает `SharedRef`. Таким образом он проинформирует систему о том, что ему, может быть, не пригодятся сами данные и их можно не доставлять на хост. `Ref` тогда обозначает, что данные будут использоваться на хосте экзекьютора и данные можно предзагрузить поближе к хосту.

Так как клиент не знает, где находятся данные, а у него есть только ссылка на них, для него все происходит абсолютно прозрачно. Такую же прозрачность могут, соответственно, иметь и экзекьюторы. [1]

Memory pressure

Физическая память, как впрочем и любой другой ресурс, бывает ограничена и в пределах одного хоста. Для того, чтобы утилизировать больше ресурсов одновременно, RPC-система может планировать несколько процедур на один хост. Соответственно, чтобы планировать выполнение по хостам, RPC-система хочет знать хотя бы примерные затраты на выполнение, чтобы ресурсов на хосте хватило для выполнения и процедуры не падали и не перезапускались (если, конечно, требуется хотя бы at-least-once). И тут она может получить некоторые подсказки по использованию памяти на основе зависимостей (аргументов) процедуры: ей понадобится хотя бы столько памяти, сколько требуют её прямые зависимости (`Ref`).

Иммутабельность

Для упрощения всей концепции, данные иммутабельны. Это упрощает имплементацию data locality (перемещение данных поближе к экзекьюторам), упрощает разработку клиентских приложений, упрощает отказоустойчивость – иммутабельные данные проще, например, реплицировать. К сожалению, иммутабельность накладывает и ряд очевидных ограничений, таких как скорость обработки данных.

В итоге, мы получаем систему, которая должна:

[1] можно планировать клиенты в той же системе, что и экзекьюторы. Тогда, потенциально, клиент может попасть на один хост с экзекьютором и данные вообще не поедут за пределы хоста.

1. собирать мусор, причем отказоустойчиво. Любой хост может упасть, и нужно обеспечить удаление данных, которые остались без владельца (на которые никто не ссылается). [2]

2. заниматься перемещением данных по хостам. Необходимо некое key-value хранилище, в котором будут лежать все данные, передаваемые экзекьюторам. Одной из оптимизаций может быть совместное размещение данных и экзекьюторов на хосте. Это позволит данным быть еще ближе к экзекьюторам, но привнесет сложность в реализации

3. заниматься планированием ресурсов. Распределять запросы по хостам, при этом учитывая количество ресурсов (памяти), необходимых для обработи запроса.

Отдельная проблема – отказоустойчивость. Вдобавок ко всем точкам отказа моделей текущих RPC фреймворков, тут добавляется перемещение и хранение данных. И тут возникают сложности с сохранением данных в случае отказа: если какую-то процедуру нужно перезапустить (для exactly/at least once семантик) нужно, чтобы данные, от которых эта процедура зависит, не уничтожились.

Ну и главная задача состоит в реализации фреймворка, который интегрировал бы существующие фреймворки, например, те же TensorFlow и Spark, и использовал бы новый подход передачи значений по ссылке.

[2] Можно, например, реализовать что-то вроде keepalive соединения. Тогда, если какой-то хост не отвечает на несколько keepalive запросов подряд, он считается упавшим.