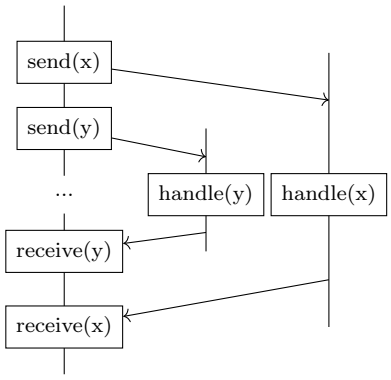
Promises: асинхронные вызовы в распределённых системах

Лев Хорошанский

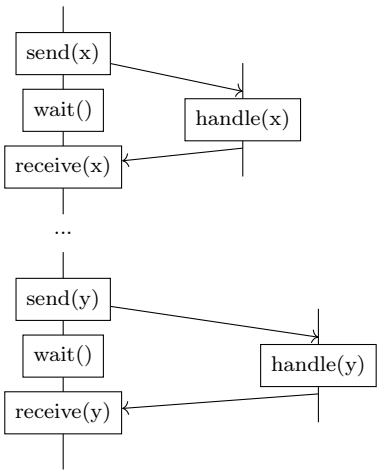
Проблемы коммуникации

Параллельные вычисления и удалённые вызовы (remote procedure calls / RPC) являются одними из важнейших составляющих распределённых систем (зачастую подобные системы включают в себя программы, написанные на разных языках и/или исполняющиеся на различных устройствах).

Если в качестве средств коммуникации внутри системы используются сообщения, то определить порядок действий становится довольно затруднительно. Ввиду этого сообщениям предпочитают RPC, которые, однако, обладают недостатком: вызывающая сторона обязана дождаться завершения вызова перед тем, как продолжить выполнение, – чем не страдает система обмена сообщениями.



Message passing



RPC

В конечном счёте, система может либо обладать гарантией того, что вызовы будут исполняться по порядку, и снижать собственную производительность из-за ожидания, либо поддерживать быстроту в ущерб последовательности выполнения.

Call-streams

Для решения данной проблемы был придуман новый способ коммуникации, в некотором смысле объединяющий сообщения и RPC. Чтобы ввести его, необходимо взглянуть на распределённую систему под определённым углом.

Рассмотрим сеть, в которой:

- в каждой вершине находится произвольное количество отправителей и получателей;
- каждый отправитель или получатель целиком располагается в одной вершине;
- отправитель и получатель могут быть соединены, используя stream (что именно из себя представляет stream будет оговорено дальше);
- отправитель может совершать вызовы к получателю при помощи stream;
- вызовы делятся на обыкновенные RPC и stream calls, отличие которых заключается в том, что они не ждут завершения вызова (то есть можно совершить несколько stream calls перед тем, как придёт ответ первому вызову);
- система отправки и получения вызовов гарантирует, что полученные вызовы будут обработаны в том порядке, в котором они были отправлены.

Таким образом, stream calls имеют два преимущества перед RPC: параллельная отправка вызовов без необходимости ожидания ответа и уменьшение стоимости отправки вызовов за счёт буфферизации (тогда как система RPC обязана отправить вызов сразу).

Каждый получатель имеет один или более портов – строго типизированные идентификаторы процедур, которые может запустить получатель на своей стороне.

Например:

```
double Sqrt(int number) throw(Failure);
```

Порт Sqrt считает квадратный корень числа, также может быть брошено исключение Failure.

Также порты могут быть объединены в группу, чтобы использовать их последовательно. В одну группу могут попасть только порты одного и того же получателя. Группы портов определяют один из концов stream, отвечающих получателю.

Пример:

```
Window CreateWindowWithText(std::string text) throw(Failure) {
    Window window;
    DrawRectangular(&window); // port
    DrawText(&window, text); // port
    ChangeColor(&window, Color::BLACK); // port
    return window;
}
```

Отправитель и получатель могут содержать concurrent события, однако подобные события имеют различные stream. Каждому событию соответствует агент, который определяет конец stream, отвечающий отправителю. Агент обладает уникальным именем и принадлежит отправителю, причём одному отправителю могут принадлежать несколько агентов одновременно.

Агент и группа портов вместе образуют пару stream = (agent, port group). Каждый stream гарантирует корректную доставку сообщений (единоразовую и строго по порядку), где под сообщением понимается вызов или ответ.

Система также обеспечивает функционал, соответствующий поломке и починке stream в разных ситуациях. Кроме этого, отправителю доступны два примитива синхронизации:

- flush() – очищает буффер вызовов и ответов у обеих сторон;
- sync() – вызывает flush() и ждёт завершения текущих вызовов.

Promises

Сразу после вызова stream call отправителю возвращается объект типа promise, который позже можно будет использовать, чтобы запросить (claim) результат вызова. Однако в promise может также лежать исключение в ситуации, когда что-то пошло не так – оборвалось соединение, некорректный запрос и так далее.

У promise объекта есть два состояния: “заблокирован” и “готов”. В момент создания promise переходит в состояние “заблокирован”, после получения ответа меняет состояние на “готов” и остаётся таким, сохраняя результат вызова до момента уничтожения самого объекта (в некотором смысле promise является “одноразовым”).

Метод claim() ждёт до тех пор, пока promise не станет готов, после чего возвращает результат как обыкновенная функция или же бросает исключение в случае неуспешного вызова. Также имеется метод ready(), который просто возвращает true, если promise готов, и false иначе.

Аргументы передаются по значению (а если быть точным – сериализуются, потому что получатель может быть запущен в другом адресном пространстве и быть написанным на другом языке), причём не каждый тип можно передать (к примеру, сами promise). В случае ошибки процесса сериализации promise бросит исключение при попытке запросить результат.

Понятно, что claim() различных promise могут быть вызваны в любом требуемом порядке, однако стоит понимать, что если promise $(n+1)$-го вызова готов, то готов и promise для n-го вызова.

Простой пример

Рассмотрим программу, в которой студентам записываются оценки за один конкернтый курс в базу данных, после чего на экран печатается средняя оценка каждого из них за все курсы.

```
std::vector<Grade> course_grades(student_count);
std::vector<Promise<Average>> averages(student_count);

for (std::size_t i = 0; i < course_grades.size(); ++i) {
    auto serialized_grade = Serialize<Grade>(course_grades[i]);
    try {
        averages[i] = StreamCall(database_agent, RecordGrade(serialized_grade));
    } catch (Failure failure) { // handle exception }
}
averages.back().flush();

Promise<void> last_print;
for (std::size_t i = 0; i < averages.size(); ++i) {
    Average promised_average;
    try {
        promised_average = averages[i].claim();
    } catch (Failure failure) { // handle exception }

    try {
        last_print = StreamCall(print_agent, Print(promised_average));
    } catch (Failure failure) { // handle exception }
}
last_print.sync();
```

У примера выше есть как достоинства (значительное число параллельных событий – вызовы RecordGrade() поступают равномерно, не дожидаясь завершения предыдущих, равно как и Print()), так и недостатки (Print() можно позвать только после того, как все вызовы RecordGrade() дойдут до другой системы).

Легко видеть, что в приведённом примере не требуется, чтобы агенты находились в различных вершинах сети – таким образом, мы можем воспроизвести подобную программу на одном устройстве, используя fork() или std::thread.

Композиции stream calls

Что если мы хотим организовать работу программы таким образом, что результат stream call под номером i подаётся как входные данные для stream call под номером $(i+1)$? Одним из возможных решений может выступить распараллеливание управляющей программы – один процесс/тред в цикле запускает RecordGrade(), добавляя полученные promise в очередь, в то время как другой в цикле достаёт новые promise из очереди и отправляет их Print().

Однако и у такого подхода есть недостатки – в случае, если один из вызовов RecordGrade() завершится исключением, соответствующий ему Print() будет бесконечно ждать. Для решения данной проблемы можно рассматривать последовательные stream calls как группу, после чего реализовывать, в случае брошенного исключения, остановку последующих вызовов.

Argus

Для языка Argus придумали решение: выражение coenter, которое, помимо всего прочего, позволяет писать код для fork()-ed процессов, не вынося его в отдельную функцию.

Выражение coenter состоит из произвольного числа arms, каждая из которых определяет то, как должен работать процесс, соответствующий определённой arm. Проводя аналогию, можно сказать, что это своего рода switch-case выражение, где каждой условной ветке ставится в исполнение отдельный процесс.

Выполнение всего coenter переводит программу в состояние ожидания до тех пор, пока не:

- либо все подпроцессы успешно завершатся (иными словами, все stream calls вернут promise, которые, в свою очередь, позволят корректно получить результат);
- либо какая-либо из arm бросит исключение, что заставит остальные arms аварийно завершить работу.

Внимательный читатель может заметить, что прерывание работы подпроцесса может навредить системе: оставить данные в некорректном виде, не разблокировать заблокированный мьютекс и так далее. Чтобы не допускать подобных ситуаций, Argus ждёт до тех пор, пока процесс не выйдет из всех критических секций, параллельно не давая ему зайти в новые. Также Argus поддерживает атомарность масштабных транзакций – в примере про студентов либо все оценки будут записаны, либо ни одна из них не попадёт в базу данных.

Кратко

Проблема коммуникации процессов внутри распределённой системы привела к появлению нового способа коммуникации call-streams (совместивший в себе два известных метода), а также типа данных promise, подчиняющегося определённой политике использования.

Однако такой подход повлёк за собой определённые трудности в плане архитектуры программы и способов их реализовать, причём настолько, что языку программирования, возможно, придётся добавить новый механизм, поддерживающий возможность привычного интерфейса вызовов обычных функций и не жертвующий производительностью.

Но достоинства тоже есть: грубо говоря, все stream calls являются асинхронными RPC, которые, в отличие от сообщений, в некотором роде предсказуемы в плане порядка исполения.

Комментарии

В общем и целом, предложенное авторами решение можно назвать удачным – подобные вещи реализованы в современных языках наподобие C++, Scala или же JavaScript (хотя и частично). Однако, изначальная задумка кажется слишком требовательной – особенно это касается гарантии порядка обработки системы отправки и получения вызовов.

Также, на мой взгляд, статье вредит жёсткая привязка к одному конкретному языку Argus (который перестал поддерживаться в 1988), так как не всегда удаётся сходу придумать аналог тому или иному функционалу в других языках (особенно это касается coenter).

Но стоить заметить, что, при должных внимательности и уровне знаний об особенностях языка, на Argus можно написать асинхронное приложение, отличающееся по производительности в лучшую сторону, однако авторами чётко не объявлена цена атомарности операций (вполне может быть, что последствия брошенного исключения могут нивелировать выигрыш в производительности).

На момент написания, такой язык как C++ обладает широкими возможностями асинхронных вызовов функций, начиная с std::promise и std::future, продолжая обычными тредами и заканчивая std::async, ввиду чего можно сказать, что более удачные подходы к решению уже были найдены.