Параллельные и распределенные вычисления. Задание 5 “Диспетчер-рабочие”.

Старченко Владимир

Основная часть

Поставленная задача

Усовершенствуйте разобранную на занятиях реализацию схемы «Управляющий-рабочие» (см. ниже код к заданию):

1. Реализуйте в диспетчере поддержку одновременного обслуживания (обработки заданий) нескольких клиентов в порядке поступления их заданий;

2. Реализуйте в диспетчере обнаружение и обработку отказов, связанных с рабочими процессами («падение» или временное «зависание» процесса, отказ сети между диспетчером и рабочим);

3. Реализуйте в рабочем обработку отказов сети между диспетчером и рабочим (рабочий должен пытаться заново соединиться с диспетчером);

4. Реализуйте в клиенте обработку отказов сети между диспетчером и клиентом (клиент должен пытаться заново соединиться с диспетчером).

Таким образом, усовершенствованная реализация должна функционировать корректно (все отправленные клиентами задания в конце концов выполняются и доставляются клиентам) в условиях отказов сети и рабочих.

Помимо корректности, реализация должна быть эффективной, минимизируя времена обнаружения отказов и ожидания результатов клиентами. При этом следует соблюдать разумный баланс между эффективностью и накладными расходами (нагрузкой на сеть).

Наконец, реализация должна хорошо работать не только на тестах, но и в произвольных условиях, не делая тем самым априорных предположений о временах поступления и обработки заданий.

Обоснование решения

В данной задаче нужно было раздать усовершенствовать существующее решение.

1. **Реализуйте в диспетчере поддержку одновременного обслуживания (обработки заданий) нескольких клиентов в порядке поступления их заданий;**

   Основная идея заключается в разделении общей очереди результатов на отдельные очереди, предназначенные для каждого подключающегося клиента. Очередь с заданиями можно не разделять, так как для клиента не имеет разницы, какой именно воркер будет обрабатывать задание.

2. **Реализуйте в диспетчере обнаружение и обработку отказов, связанных с рабочими процессами («падение» или временное «зависание» процесса, отказ сети между диспетчером и рабочим);** Здесь нужно было реализовать механизм heartbeat. С его помощью, диспетчер может мониторить состояние воркеров. В случае обнаружения проблемы, он должен заново дать работу новому воркеру, чтобы обрабатываемый item не потерялся и client получил полностью сделанную работу.

3. **Реализуйте в рабочем обработку отказов сети между диспетчером и рабочим (рабочий должен пытаться заново соединиться с диспетчером);** В этом пункте требовалось не завершать работу воркера при падении сети, а пытаться повторить попытку соединения. При этом диспетчер может подумать, что воркер

упал на всегда и дать данный item другому воркеру. При зависании воркера данный item будет доставлен диспетчеру дважды, а client должен получить его лишь 1 раз. Это нужно отдельно обрабатывать

4. **Реализуйте в клиенте обработку отказов сети между диспетчером и клиентом (клиент должен пытаться заново соединиться с диспетчером).** В этом пункте требовалось не завершать работу клиента при падении сети, а пытаться повторить поптку соединения.

Так же в реализации нужно было следить за возможностью появления состояния гонки в разных потоках диспетчера.

Описание реализации

Реализация обхода состояния гонки может быть тривиальным, если использовать mutex-ы в диспетчере. Действительно, все race conditions могут быть убраны таким образом. При этом сильно будет страдать время выполнения. Хотелось сделать решение с наименьшим количеством mutex-ов, используя все возможности потокобезопасных структур типа defaultdict и queue.

Обработку отказов сети и в клиенте и в воркере можно сделать, обернув вызовы функций диспетчера в try-except. Это и было сделано, только разными способами. В начале появилась реализация воркера. Честно сказать, она довольно некрасивая. try-except-ы стоят в разных местах, обрабатывая каждый конкретный случай. Это убрано в реализации клиента. Одна обертка позволяет пытаться выполнить запрос до тех пор, пока сеть не наладится.

Результаты

Программа запускалась несколько раз на одних и тех же файлах, для чистоты эксперимента. https://everest.distcomp.org/jobs/5a2c48b1330000d580a3c852

Анализ решения

Самое интетресно происходило при попытках избавиться от race conditions. Рассмотрим пример:

```
# self.processeditems --- defaultdict
processeditems = self.processeditems.get(item.client, None)
if processeditems is None:
  return
```

Этот не хитрый трюк позволяет уберечь от случая, когда объект по ключу item.client удаляется в другом потоке из словаря. Действительно, если удаление было выполнено, дальше пойдет работа с processeditems, который больше не нужен в программе. Причем работа будет выполнена, по сути, в холостую, но без возникновения race condition (если код дальше только изменяет очередь, а не состояние диспетчера на основе состояния processeditems).

Этот трюк позволяет сделать всё хорошо, кроме функции putResult. Действительно, если нам нужно положить результаты в очередь, и есть 2 воркера, которые хотят это сделать (такое может быть). Мы сталкиваемся с проблемой. Не понятно, какой результат будет записан. Нам на помощь придёт метод setdefault.

```
cur_item = processeditems.setdefault(item.itemId, item)
# set item as new processed item
if item.processedBy is cur_item.processedBy:
  clientqueue.put(item)
```

Он атомарно присвоит значению словаря значение item, только если оно не было уже выставлено до него. И вернет значение из словаря. Если возвращенное значение действительно было переданно из данного процесса, оно положиться в очередь результатов. Сделано это будет ровно 1 раз.

Можно заметить, что 1 мьютекс всё равно пришлось использовать.

```
if time() - timestemp > HEARTBEAT_MAX_DELAY:
  ...
  self.lock.acquire()
  if time() - timestemp > HEARTBEAT_MAX_DELAY:
    self.activeworks.pop(worker, None)
    self.heartbeats.pop(worker, None)
  self.lock.release()
```

Действия *if time() - timestemp > HEARTBEAT_MAX_DELAY* нужно делать в единственном потоке. Однако делать lock на каждый heartbeat очень долго. Поэтому мы используем прием, когда 2 раза проверяем условия. Тогда и condition race не возникнет, и lock будет выполняться только если условие выполнено (что намного реже).

Контрольные вопросы

Вопрос 1.

Является ли реализация диспетчера потокобезопасной? Почему? (вес 0.25) Да, является. Все структур данных потокобезопасны, а приемы, описанные выше, и мьютексы позволяют избежать condition race.

Вопрос 2.

Какая стратегия используется для обнаружения и обработки отказов рабочих? Почему? (вес 0.25)

Вычисление factorize вынесено в отдельный поток, а в основном выполняется функция heartbeat. Это позволяет понимать диспетчеру, что рабочий всё ещё жив и успешно функционирует. Как только ответ от воркера задерживается, его задача отдается другому рабочему. Таким образом, если что-то произошло с сетью или самим рабочим, задача не будет потеряна.

Вопрос 3.

Как обрабатывается потеря связи с диспетчером в клиенте/рабочем? Почему? (вес 0.25)

Здесь достаточно просто пытаться заново соединиться. Клиент не выполняет никакой работы и диспетчеру не обязательно знать, что с сетью проблемы. Как только сеть восстановится, клиент снова сможет продолжть нормальную работу и забрать у диспетчера данные. (Кроме случая оговоренного ниже)

Вопрос 4.

Может ли происходить потеря заданий/результатов? Почему? (вес 0.25) Потерь, связанных с отказом рабочего быть не может, как было показано выше.

Если диспетчер умрет, очевидно, задания будут утеряны. Однако этот случай не рассматривается в данном задании.

Остается одна проблема. Допустим, рабочий успешно вызвал метод getResult диспетчера. Этот метод начал выполняться. Допустим, мы в этом методе делаем return. В этот момент данные у диспетчера уже отобраны. Если при return сеть порушится, результат работы потеряется. Чтобы этого избежать, можно сохранять результаты работы несколько раз, дублируя их и удалять копию только при подтверждении клиентом целостности полученных данных. Однако для этого нужно добавлять новые функции (новые именно по своей концепции, а не просто вспомогатеьные функции). Более того, данный случай зависит от реализации Pyro. Если судить по документации, начиная с какой-то версии он поддерживается из коробки. Именно поэтому я не стал заново реализовывать эту функциональность отдельно.