Изучение консенсуса с помощью python.(24 марта 2018)

Вступление

Как заставить разные компьютеры, входящие в более крупный кластер, договориться о том, каково значение чего-либо?
Предположим, клиент отправил запрос на сервер 1, чтобы установить значение ключа `foo` в строку значений. Однако сервер 1 является частью кластера из N серверов (есть веские причины для того, чтобы серверы были частью кластера, но мы не будем вдаваться в это)

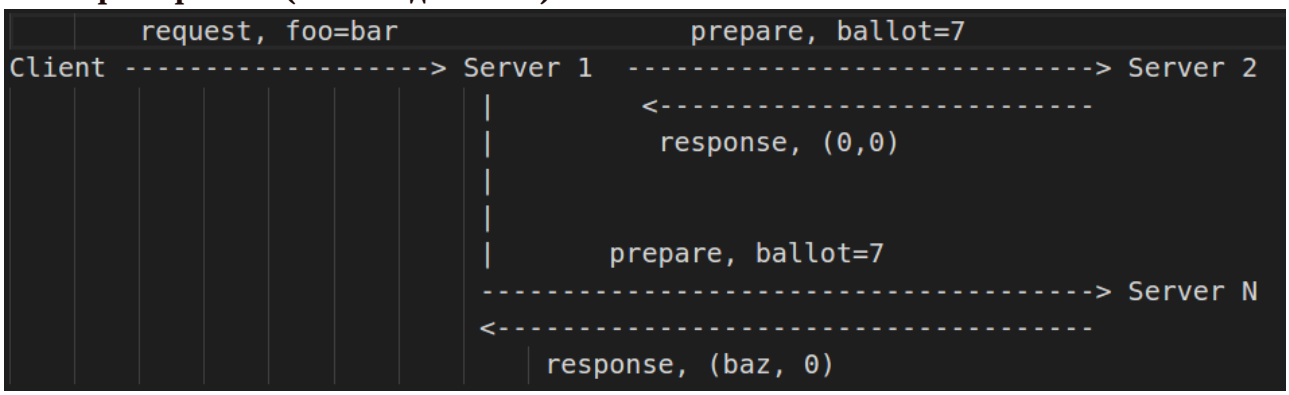
запрос клиента. Таким образом, запрос; `set foo=bar` должен быть распространен на все серверы. Наивный подход выглядел бы примерно так;
Когда сервер 1 получает запрос, он также отправляет этот запрос серверу 2, который, в свою очередь, отправляет запрос серверу 3...
Однако при таком подходе возникает пара проблем;

- Что произойдет, если где-нибудь в этой цепочке запросов возникнет какая-то сетевая ошибка?
- Должен ли клиент ждать завершения всех этих обходов запросов, прежде чем он сможет получить ответ?
- Что, если правительственное агентство с именем из 3 букв перехватит запрос и изменит его с foo=bar на foo=baz?
- И так далее

То, как мы заставляем компьютеры соглашаться со значением некоторых данных, достигается с помощью консенсуса.
Существует ряд консенсусных протоколов с различной степенью использования в реальном мире. Вот кроличья нора из Википедии, чтобы вы могли начать.
В оставшейся части этого поста мы рассмотрим один из таких алгоритмов - CASPaxos.

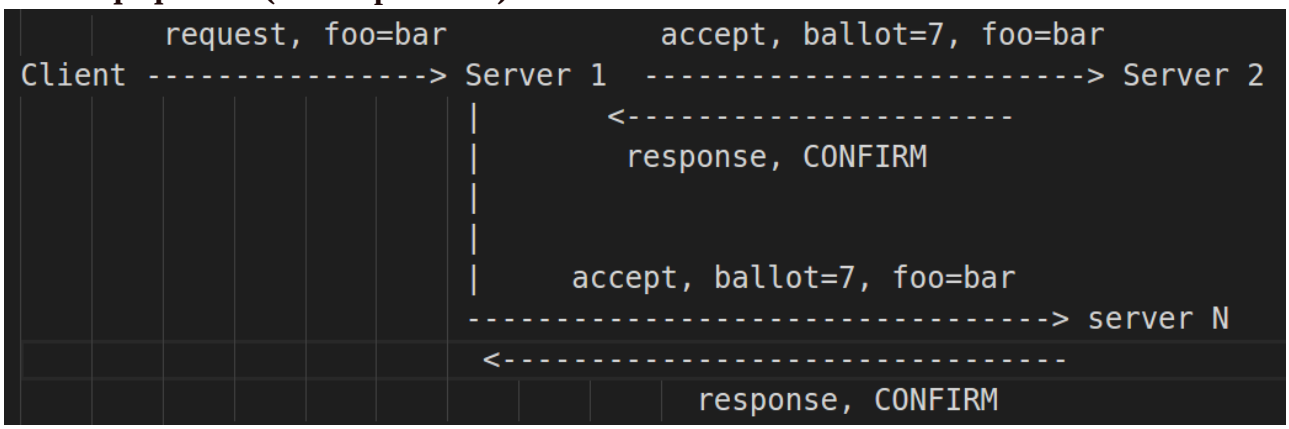
CASPaxos (это относительно новый способ)
CASPaxos - это консенсусный алгоритм Дениса Рысцова, технический документ доступен на сайте arxiv.
Статья на удивление проста и к тому же легка для понимания.
В двух словах, алгоритм выглядит следующим образом;

A. Prepare phase: (Этап подготовки)



1. клиент отправляет запрос на сервер 1 (proposer = предлагатель). Сервер 1 генерирует номер бюллетеня и отправляет этот бюллетень всем остальным серверам (акцепторам)
2. Другие серверы (акцепторы оперируют данными) вернут сообщение о конфликте (на сервер 1), если они уже увидели большее количество бюллетеней. Сохраняет номер бюллетеня и возвращает подтверждение либо с пустым значением (если они еще не приняли какое-либо значение), либо с кортежем принятого значения и его номером для голосования.
3. Сервер 1 ожидает ответа большинства серверов с подтверждением (кворум ответов).

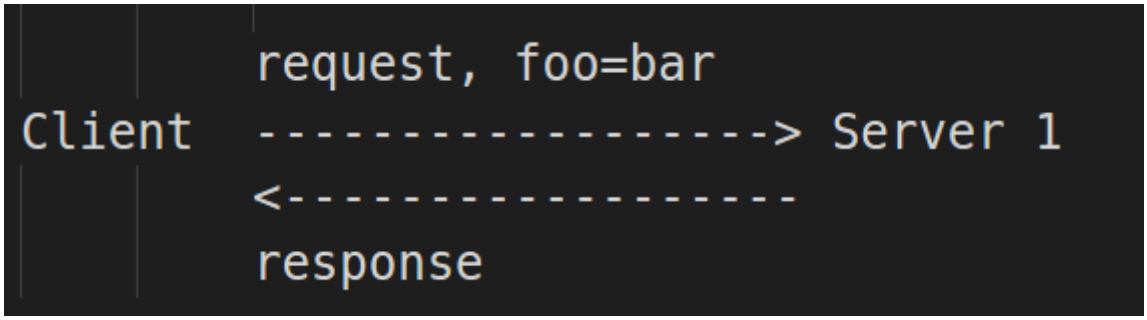
B. Accept phase: (Этап принятия)



4. Сервер 1 отправляет ранее сгенерированный номер и значение бюллетеня (и значение foo=bar) на все остальные серверы.

5. Серверы вернут сообщение о конфликте, если они уже увидели большее значение номера бюллетеня (за время между prepare и accept по какой то причине применилось другое значение?). Они помечают номер и значение полученного бюллетеня как принятые значения и возвращают подтверждение.

C. End (Конец):



6. Сервер 1 ожидает ответа большинства серверов с подтверждением (кворум).
7. Возвращает ответ клиенту.

Отказ от ответственности: Приведенное выше описание не является правильным на 100%. Я опустил ряд важных деталей (например, в статье клиент не отправляет запрос типа set foo =bar, вместо этого он отправляет функцию изменения). Вам следует ознакомиться с самим документом для получения более подробной информации. Однако на данный момент приведенного выше описания будет достаточно.

python

Давайте рассмотрим, как алгоритм достигает консенсуса, мы будем использовать python, чтобы попытаться продемонстрировать это.

Мы собираемся создать прототип распределенной системы хранения данных и будем использовать CASPaxos для достижения консенсуса между различными серверами в нашей системе хранения данных.

Прежде всего, код для proposer(server1 в нашем описании CASPaxos выше).

```
import time
import random

class Proposer(object):
    def __init__(self, acceptors):
        if not isinstance(acceptors, list):
            raise ValueError(
                "acceptors ought to be a list of child classes of
Acceptor object")
```

```
        self.acceptors = acceptors
        # since we need to have 2F+1 acceptors to tolerate F
failures, then:
        self.F = (len(self.acceptors) - 1) / 2
        self.state = 0

    def receive(self, state):
        """
        receives request from client. The request will be a
dictionary like; state={'foo': 'bar'}
        """
        ballot_number = self.generate_ballot_number()
        self.send_prepare(ballot_number=ballot_number)
        result = self.send_accept(state, ballot_number)
        return result

    def generate_ballot_number(self, notLessThan=0):
        # we should never generate a random number that is equal to
zero
        # since Acceptor.promise defaults to 0
        ballot_number = random.randint(notLessThan + 1, 100)
        return ballot_number

    def send_prepare(self, ballot_number):
        # list of tuples conatining accepted (value,
ballotNumOfAcceptedValue)
        confirmations = []
        for acceptor in self.acceptors:
            confirmation =
acceptor.prepare(ballot_number=ballot_number)
            if confirmation[0] == "CONFLICT":
                # CONFLICT, do something
                pass
            else:
                confirmations.append(confirmation)

        # Wait for the F + 1 confirmations
        while True:
            if len(confirmations) >= self.F + 1:
```

```
                break
            else:
                # sleep then check again
                time.sleep(5)

        total_list_of_confirmation_values = []
        for i in confirmations:
            total_list_of_confirmation_values.append(i[0])

        if sum(total_list_of_confirmation_values) == 0:
            # we are using 0 as PHI
            self.state = 0
        else:
            highest_confirmation =
self.get_highest_confirmation(confirmations)
            self.state = highest_confirmation[0]

    def get_highest_confirmation(self, confirmations):
        ballots = []
        for i in confirmations:
            ballots.append(i[1])
        ballots = sorted(ballots)
        highestBallot = ballots[len(ballots) - 1]

        for i in confirmations:
            if i[1] == highestBallot:
                return i

    def send_accept(self, state, ballot_number):
        self.state = state
        acceptations = []
        for acceptor in self.acceptors:
            acceptation = acceptor.accept(
                ballot_number=ballot_number, new_state=self.state)
            if acceptation[0] == "CONFLICT":
                # CONFLICT, do something
                pass
            else:
                acceptations.append(acceptation)
```

```
        # Wait for the F + 1 confirmations
        while True:
            if len(acceptations) >= self.F + 1:
                break
            else:
                # sleep then check again
                time.sleep(5)

        # Returns the new state to the client.
        return self.state
```

И код для акцепторов (другие серверы в нашем описании CASPaxos выше).

```
class Acceptor(object):
    promise = 0  # ballot number
    accepted = (0, 0)

    def __init__(self, name):
        self.name = name

    def prepare(self, ballot_number):
        if self.promise > ballot_number:
            return ("CONFLICT", "CONFLICT")

        # this ought to be flushed to disk
        self.promise = ballot_number
        return self.accepted

    def accept(self, ballot_number, new_state):
        if self.promise > ballot_number:
            return ("CONFLICT", "CONFLICT")
        elif self.accepted[1] > ballot_number:
            return ("CONFLICT", "CONFLICT")

        # these two ought to be flushed to disk
        # http://rystsov.info/2015/09/16/how-paxos-works.html
        self.promise = 0
```

```
        self.accepted = (new_state, ballot_number)
        return ("CONFIRM", "CONFIRM")
```

Мы собираемся использовать экземпляры класса python для представления (вместо) реальных серверов.
Мы собираемся создать пять серверов с именами a1 .... a5 типа Acceptor.
Мы также создадим еще один сервер типа Proposer.
Мы попросим клиента отправить запрос заявителю, после чего алгоритм CASPaxos вступит в силу.
В конце всего этого запрос клиента будет сохранен на всех 5 серверах.

```
a1 = Acceptor(name='a1')
a2 = Acceptor(name='a2')
a3 = Acceptor(name='a3')
a4 = Acceptor(name='a4')
a5 = Acceptor(name='a5')

acceptorsList = [a1, a2, a3, a4, a5]
p = Proposer(acceptors=acceptorsList)

response = p.receive(state={'foo': 'bar'})
print "response:", response

for acceptor in acceptorsList:
    print "value persisted by acceptor={0} is
{1}".format(acceptor.name, acceptor.accepted[0])
```

И результат:

```
response: {'foo': 'bar'}
value persisted by acceptor=a1 is {'foo': 'bar'}
value persisted by acceptor=a2 is {'foo': 'bar'}
value persisted by acceptor=a3 is {'foo': 'bar'}
value persisted by acceptor=a4 is {'foo': 'bar'}
value persisted by acceptor=a5 is {'foo': 'bar'}
```

Полный код можно найти на github.

Вывод

CASPaxos может помочь вам достичь консенсуса в ваших распределенных системах и немного упростить обработку сбоев.
Протокол также содержит разделы, которые охватывают другие аспекты системного администрирования распределенных систем, такие как добавление и удаление серверов и другие.
Я приглашаю вас пойти почитать исходную статью.
Его автор, Денис Рысцов, находится в твиттере @rystsov, и я благодарен ему за то, что он отвечал на мои вопросы всякий раз, когда я обращался к нему. (не мне, я перевожу)

Однако CASPaxos не поможет вам, если агентства с названиями из трех букв решат подделать ваши запросы. (я как бы могу и эту проблему решить. переводчик)