

Frei

Seminararbeit am Institut für Informatik der Freien Universität Berlin,
Arbeitsgruppe Technische Informatik

Two-Phase Commit for FUCoin

Michael Kmoch
Matrikelnummer: 4289388
michael.kmoch@inf.fu-berlin.de

Yuri Lewash
Matrikelnummer: 4293181
yuri.lewash@inf.fu-berlin.de

Betreuer: S. Schmitt
Eingereicht bei: Prof. Dr. K. Wolter

Berlin, 24.07.2015

Abstract

In distributed systems a commit is needed to make changes in the network permanent and visible to other participants. The two-phase commit protocol includes a voting phase and a decision phase to coordinate processes participating in a atomic transaction. We implement a variant of this protocol for the FUCoin system.

1 Introduction

For a transaction in the FUCoin system we have to make sure, that either all operational participants commit the transaction or none of them. Each participant has one of two votes for the possible transaction: **YES** or **NO**. For the transaction to happen all participants have to vote **YES**, so a decision cannot be reversed. The initial participant acts as the coordinator and sends a 'vote request' to all other participants. After a participant receives this request, it responds with either **YES** or **NO**. In case of **NO** the decision is already 'Abort'. If all participants vote **YES**, the coordinator decides 'Commit' and sends a commit message to all the other participants. Otherwise it sends an abort message to all participants that voted **YES**.

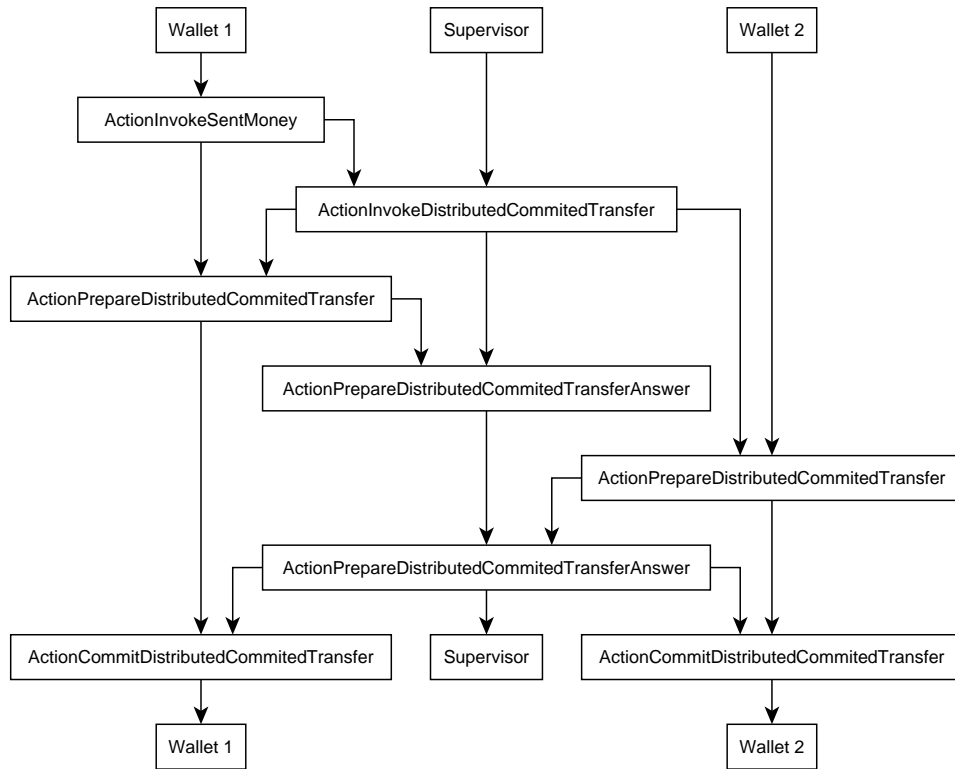


Figure 1: Flowchart of a complete transaction

2 Implementation

The first action `ActionInvokeSentMoney` is invoked by the graphical user interface of the wallets with the purpose to transfer an amount of FUCoins to a given wallet ID. If this ID already has a mapping to an `ActorRef`, an `ActionInvokeDistributedCommittedTransfer` will be send to the supervisor. Otherwise a gossip is invoked using the `ActionSearchWalletReference` following by `ActionInvokeSentMoney` again after 200 ms.

Next the `ActionInvokeDistributedCommittedTransfer` creates a `DistributedCommittedTransferRequest`, which contains a random generated ID on the server with a timeout of 500 ms and stores this to a map `Long -> Request` according to the ID. The timeout is handled by the `ActionUpdateQueue` explained later. The request will spread out to all clients that can answer with an acknowledgement or an abort afterwards (see below).

The clients will reply to a request with an acknowledgment if one of two cases occur. In the first case the supervisor (bank) will send FUCoins to a user. Otherwise the client would need to know the sender, who also needs to have sufficient funds.

After an answer from a client reached the server, the server tries to find the corresponding request. If the answer was a acknowledgement, the request will get another positive answer. When the same amount of positive answers equals the count of known neighbors received on the server, all client will be informed to commit the change and the request will be deleted. If the answer was an abort, all clients will be informed to abort the transaction. If a client has to commit the given changes, it will perform the transaction on the amounts map. Otherwise it will just print an abort transaction message.

The `ActionUpdateQueue` event will be invoked each second on the server and remove all outdated request. If a request is deleted, all clients will be informed to abort this transaction.

Concerning the ACID properties, atomicity is complied with the two-phase commit, which is an atomic commitment protocol. Consistency is also not a problem here, since the only variables are the amount of FUCoins on each node as well as the temporary holder of a wallet. None of them can invalidate the state of the system. Isolation might be a problem while processing two or more transactions simultaneously. Without a serialization responses within a transaction can be read by another transaction. Durability was not a subject for this implementation and therefore not met at all.

3 Development

The Implementation was made with Akka, an open source toolkit for highly concurrent, distributed Java applications and part of Typesafe's "Reactive Platform".

To get started, Typesafe recommends their "Typesafe Activator", a kind of IDE and hub running in a web browser. While it's quite easy to build a first "Hello World" example within the Activator, it's not very flexible and things get more complicated when trying to transfer a project to Eclipse. Furthermore high hardware standards, like at least 4 GB of RAM on modern desktop environments, are needed to avoid poor performance.

So, setting up a standalone distribution for use with Eclipse would seem the thing to do. Maven is the weapon of choice for this job. After creating a "Maven Project" in Eclipse, a Project Object Model like the following can be used to add Akka as an external library.

```
<project>
~
  <!-- project coordinates -->
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-actor_2.11</artifactId>
  <version>2.4-M1</version>
  ~
</project>
```

4 Conclusion

Akka is a useful library for distributed programming with a steep learning curve (in terms of both possible initial difficulties as well as a high rate of acquiring skills). It is easy to develop distributed systems by dividing the logic over multiple actors. Akka also provides support for networking, but we didn't use it in this application.