



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

Two-Phase Commit for FUCoin

Michael Kmoch, Yuri Lewash

Matrikelnummer: 4289388, 4293181

michael.kmoch@inf.fu-berlin.de, yuri.lewash@inf.fu-berlin.de

Betreuer: Simon Schmitt

Eingereicht bei: Katinka Wolter

Berlin, 24.07.2015

Abstract

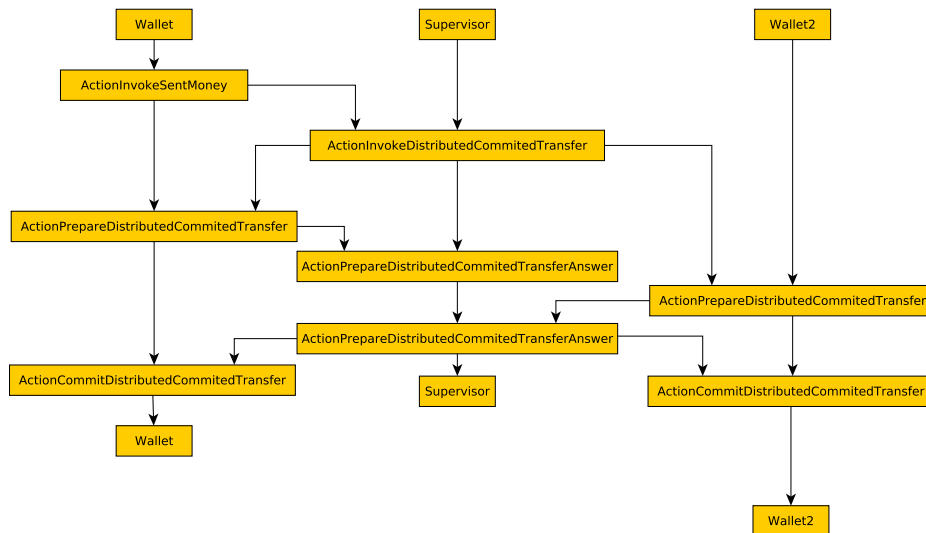
In distributed systems a commit is needed to make changes in the network permanent and visible to other participants. The two-phase commit protocol includes a voting phase and a decision phase to coordinate processes participating in a atomic transaction. We implement a variant of this protocol for the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

Contents

1	Introduction	1
2	ActionInvokeSentMoney	2
3	ActionInvokeDistributedCommittedTransfer	3
4	ActionPrepareDistributedCommittedTransfer	4
5	ActionPrepareDistributedCommittedTransfer Answer	5
6	ActionCommitDistributedCommittedTransfer	6
7	ActionUpdateQueue	7

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 init 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 participants. Otherwise it sends an abort message to all participants that voted YES.



2 ActionInvokeSentMoney

This action is invoked by the graphical user interface of the wallets. The goal is 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 after 200 ms.

```
protected void onAction(ActorRef sender, ActorRef self, UntypedActorContext context,
                        Wallet wallet) {
    log(wallet.getKnownNeighbors()+"");
    if(wallet.getKnownNeighbors().containsKey(name)){
        wallet.getRemoteSuperVisorActor().tell(
            new ActionInvokeDistributedCommittedTransfer(self,
                wallet.getKnownNeighbors().get(name), amount), sender);
    }else{
        ActionSearchWalletReference aswr = new ActionSearchWalletReference(name);
        for(ActorRef neighbor : wallet.getKnownNeighbors().values()){
            neighbor.tell(aswr, self);
        }
        sleep(self, context, 200);
        self.tell(this, self);
    }
}
```

3 ActionInvokeDistributedCommittedTransfer

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).

```
protected void onAction(ActorRef sender, ActorRef self, UntypedActorContext context,
                        Supervisor supervisor) {
    log("invoke transaction "+source.path().name()+" sends "+amount+" to "
        +target.path().name());
    long timeout = System.currentTimeMillis()+500;
    DistributedCommittedTransferRequest ds =
        new DistributedCommittedTransferRequest(source, target, timeout);
    supervisor.addDistributedCommittedTransferRequest(ds);
    ActionPrepareDistributedCommittedTransfer apdct =
        new ActionPrepareDistributedCommittedTransfer(source, target, amount, timeout,
                                                    ds.getId());
    for(ActorRef neighbor : supervisor.getKnownNeighbors().values()){
        neighbor.tell(apdct, self);
    }
}
```

4 ActionPrepareDistributedCommittedTransfer

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.

```
protected void onAction(ActorRef sender, ActorRef self, UntypedActorContext context,
                        Wallet wallet) {
    // sender is supervisor (bank) and always has funds
    boolean granted = sender.compareTo(source) == 0
    // sender is unknown, might be valid
    || (wallet.amounts.containsKey(source)
    // sender have enough money
    && wallet.amounts.getOrDefault(source, 0) >= amount);
    sender.tell(new ActionPrepareDistributedCommittedTransferAnswer (source, target,
                                                                    amount, timestamp, granted, id), self);
}
```

5 ActionPrepareDistributedCommittedTransferAnswer

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.

```
protected void onAction(ActorRef sender, ActorRef self, UntypedActorContext context,
                        Supervisor supervisor) {
    log(""+supervisor.getKnownNeighbors());
    log("granted?" + granted);
    DistributedCommittedTransferRequest request = supervisor.getRequest(id);
    if(granted){
        if(request==null) //unknown DistributedCommittedTransferRequest ignore
            return;
        int newCount = request.addPositiveAnswer(sender);
        if(newCount == supervisor.getKnownNeighbors().size()){
            ActionCommitDistributedCommittedTransfer acdct =
                new ActionCommitDistributedCommittedTransfer(source, target, amount, true,
                                                              timestamp, id);

            for(ActorRef neighbor : request.getAnswers()){
                neighbor.tell(acdct, self);
            }
            supervisor.deleteRequest(request);
        }
    }else{
        // A client wants to rollback
        if(request!=null){
            ActionCommitDistributedCommittedTransfer acdct =
                new ActionCommitDistributedCommittedTransfer(source, target, amount, false,
                                                              timestamp, id);

            for(ActorRef neighbor : request.getAnswers()){
                neighbor.tell(acdct, self);
            }
        }
    }
}
```

6 ActionCommitDistributedCommittedTransfer

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.

```
protected void onAction(ActorRef sender, ActorRef self, UntypedActorContext context,
                        Wallet wallet) {
    log("ActionCommitDistributedCommittedTransfer is granted?" + granted);
    if(granted){
        Integer sourceAmount = wallet.amounts.getOrDefault(source,0);
        Integer targetAmount = wallet.amounts.getOrDefault(target,0);
        wallet.amounts.put(source,sourceAmount-amount);
        wallet.amounts.put(target,targetAmount+amount);
        if(source.compareTo(self)==0)
            wallet.amount-=amount;
        else if(target.compareTo(self)==0)
            wallet.amount+=amount;
        wallet.log("have now "+wallet.amounts.get(self)+" Fucoins");
    }else{
        log("abort transaction with id"+id);
    }
    log("wallet.amounts:"+wallet.amounts);
}
```


7 ActionUpdateQueue

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.

```
protected void onAction(ActorRef sender, ActorRef self, UntypedActorContext context,
                        Supervisor supervisor) {
    List<DistributedCommittedTransferRequest> deletes = supervisor.updateList();
    for(DistributedCommittedTransferRequest outdatedRequest : deletes){
        ActionCommitDistributedCommittedTransfer acdct =
            new ActionCommitDistributedCommittedTransfer(outdatedRequest);
        for(ActorRef neighbor : supervisor.getKnownNeighbors().values()){
            neighbor.tell(acdct, self);
        }
    }
    sleep(self, context, 1000);
    self.tell(this, self);
}
```