TD 10 – Protocole de validation à deux phases

1. Protocole de validation à deux phases

Question 1 :
Dans un protocole de validation a deux phases, il y a tout d'abord une phase de vote qui consiste à faire valider pour chaque membre de la transaction que celle ci est valide et possible, puis une phase d'engagement ou d'abandon.

Question 2 :

```
service TransactionItf {
        i32 beginTransaction(),
        bool commit(1:i32 transacID),
        bool rollback(1:i32 transacID)
}

service GestionItf {
        void register(1:i32 transacID, 2:Participant participant),
        void remove(1:i32 transacID, 2:Participant participant)
}

struct Participant {}

service ParticipantItf {}
```

Question 3 :
C implémente GestionItf et TransactionItf.
P1, P2, ... Pn implémentent ParticipantItf.
U n'implémente aucune interface.

Question 4 :
Soit id l'id de transaction (transacID) retournait par l'appel C.beginTransaction()

```
// Debut de la transaction et enregistrement des participants
U -> C.beginTransaction()
U -> C.register(id, P1)
…
U -> C.register(id, Pn)
C -> P1.beginTransaction(id)
…
C -> Pn.beginTransaction(id)
// Chaque participant enregistre son vote
U -> P1.credit(x) ou P1.debit(x)
...
U -> Pn.credit(x) ou Pn.debit(x)
U -> C.commit(id)
// Chaque participant retourne son vote
C -> P1.vote(id)
…
C -> Pn.vote(id)
```

```
// Si la transaction est validé et possible, phase d'engagement
C -> P1.commit(id)
...
C -> Pn.commit(id)
// Sinon, phase d'abandon
C -> P1.rollback(id)
…
C -> Pn.rollback(id)
// Dans tous les cas, on enlève les participants de la transaction à la fin
U -> C.remove(id, P1)
…
U -> C.remove(id, Pn)
```

Question 5 :

```
public class Coordinateur {
	private HashMap<Integer, List<Participant>> map = new HashMap <Integer, new
                                                                List<Participant>>;

	public synchronized int beginTransaction () {
		int id = map.size();
		map.put(id, new ArrayList<Participant>());
		return id;
	}

	public void register (int id, Participant p) {
		List<Participant> list_p = map.get(id);
		if(!list_p.contains(p)){
			list_p .add(p);
		}
	}

	public void remove (int id, Participant p) {
		List<Participant> list_p = map.get(id);
		if(list_p.contains(p)){
			list_p .remove(p);
		}
	}
}

public class Participant implement BanqueItf {
	private txCurrent = -1;
	private account;
	private nextAccount;

	public synchronized int beginTransaction (int transacId) {
		if (txCurrent != -1){
			txCurrent = transacId;
			nextAccount = account;
```

```
                return 1;
            }
            return 0;
        }

        @Override
        public void credit (int i){
            nextAccount+= i;
        }

        @Override
        public void debit(int i){
            nextAccount-= i;
        }

        public synchronized boolean vote (int id)throws InvalidId {
            if (currentTransaction != transactionId){
                throw new InvalidId();
            }
            return nextAccount >= 0;
        }


        public synchronized void commit (int id) throws InvalidId {
            if (currentTransaction != transactionId){
                throw new InvalidId();
            }
            account = nextAccount;
            txCurrent = -1;
        }

        public synchronized void rollback (int id) throws InvalidId {
            if (currentTransaction != transactionId){
                throw new InvalidId();
            }
            txCurrent = -1;
        }
}
```

Question 6 :

Cette version permet de réduire le nombre d'invocations d'opérations en effectuant la transaction en une seule phase. En effet, lors du vote, chaque participant transmet à son successeur l'état de son vote && l'état du vote qu'il a reçu de son prédécesseur. Le dernier participant sait ainsi directement s'il faut annuler ou faire la transaction. De plus, en répondant, les autres participants en ont connaissance également.

Question 7 :

// Début de la transaction et enregistrement des participants suivant le modèle linéaire
U -> C.beginTransaction()
U -> P1.register(id, P2)
…
U -> Pn-1.register(id, Pn)
U -> C.register(id, P1)
C -> P1.beginTransaction(id)
P1 -> P2.beginTransaction(id)
…
Pn-1 -> Pn.beginTransaction(id)
// Chaque participant enregistre son vote
U -> P1.credit(x) ou P1.debit(x)
...
U -> Pn.credit(x) ou Pn.debit(x)
U -> C.commit(id)
// Chaque participant vote
C -> P1.vote(id, vote1)
P1 -> P2.vote(id, vote1&&vote2)
…
C -> Pn.vote(id, voteN-1 && voteN)
// Si la transaction est validé et possible, phase d'engagement
Pn -> this.commit(id)
...
P1 -> this.commit(id)
// Sinon, phase d'abandon
Pn -> this.rollback(id)
…
P1 -> this.rollback(id)
// Dans tous les cas, on enlève les participants de la transaction à la fin
U -> P1.remove(id, P2)
…
U -> Pn-1.remove(id, Pn)
U -> C.remove(id, P1)

Question 8 :

Dans la version Distributed 2PC, chaque participant donne le résultat de son vote à tous les autres. Ainsi, un participant peut, une fois qu'il a connaissance de tous les votes des autres ou qu'un des autres participant à voter false, annuler ou faire la transaction.

Question 9 :

// Debut de la transaction et enregistrement des participants suivant le modèle Distributed 2PC
U -> C.beginTransaction()
// Chaque participant enregistre tous les autres
U -> P1.register(id, C)
U -> P1.register(id, P2)
…
U -> P1.register(id, Pn)

U -> P2.register(id, C)
U -> P2.register(id, P1)
…
U -> P2.register(id, Pn)
....
U -> Pn.register(id, C)
U -> Pn.register(id, P1)
…
U -> Pn.register(id, Pn-1)
U -> C.register(id, P1)
…
U -> C.register(id, Pn)
C -> P1.beginTransaction(id)
C -> P2.beginTransaction(id)
…
C -> Pn.beginTransaction(id)
// Chaque participant enregistre son vote
U -> P1.credit(x) ou P1.debit(x)
...
U -> Pn.credit(x) ou Pn.debit(x)
U -> C.commit(id)
// Chaque participant vote et le transmet à tous le monde
P1 -> P2.vote(id)
...
P1 -> Pn.vote(id)
…
Pn -> P1.vote(id)
...
Pn -> Pn-1.vote(id)
// Si la transaction est validé et possible, phase d'engagement
P1 -> this.commit(id)
…
Pn -> this.commit(id)
// Sinon, phase d'abandon
P1 -> this.rollback(id)
…
Pn -> this.rollback(id)
// Dans tous les cas, on enlève les participants de la transaction à la fin
U -> P1.remove(id, C)
U -> P1.remove(id, P2)
…
U -> P1.remove(id, Pn)
U -> P2.remove(id, C)
U -> P2.remove(id, P1)
…
U -> P2.remove(id, Pn)
....
U -> Pn.remove(id, C)
U -> Pn.remove(id, P1)

…
U -> Pn.remove(id, Pn-1)
U -> C.remove(id, P1)
…
U -> C.remove(id, Pn)