TinyCoin: Simulating mining strategies in a simplified Bitcoin Network:

Project Code of Francesco Balzano

I initially report the Java classes of the project and finally the Python scripts used to automatize the simulation.

Java classes of the project

Package *it.unipi.p2p.tinycoin*

Block

```
package it.unipi.p2p.tinycoin;
import java.util.ArrayList;
import java.util.List;

public class Block {

	private final String bid;
	private final String parent;
	private final TinyCoinNode miner;
	private final double reward;
	private final List<Transaction> transactions;

	public Block(String bid, String parent, TinyCoinNode miner, List<Transaction> trans,
			double fixedFee)
	{
		this.bid = bid;
		this.parent = parent;
		this.miner = miner;
		transactions = new ArrayList<>();
		double fees = 0;
		for (Transaction t : trans) {
			transactions.add(t);
			fees += t.getFee();
		}
		reward = fixedFee + fees;
	}

	public String getBid() {
		return bid;
	}

	public String getParent() {
		return parent;
	}

	public TinyCoinNode getMiner() {
		return miner;
	}

	public double getReward() {
		return reward;
	}

	public List<Transaction> getTransactions() {
		return transactions;
	}

	/** Gets the revenue for the block, defined as the fixed reward plus the fees for all
	 *  the transactions
	 */
	public double getRevenueForBlock() {
		double revenue = reward;
```

```
                List<Transaction> trans = transactions;
                for (Transaction t: trans)
                        revenue += t.getFee();
                return revenue;
        }

        /** Gets the amount of coins destined to the TinyCoinNode tn in the transactions of the
Block.
         */
        public double getTransactionsAmountIfRecipient(TinyCoinNode tn)
        {
                int amount = 0;
                for (Transaction t: transactions) {
                        if (t.getOutput() == tn)
                                amount += t.getAmount();
                }
                return amount;
        }
}
```

MinerType

```
package it.unipi.p2p.tinycoin;

public enum MinerType
{
        CPU,
        GPU,
        FPGA,
        ASIC
}
```

NodeType

```
package it.unipi.p2p.tinycoin;

public enum NodeType {
        NODE,
        MINER,
        SELFISH_MINER
}
```

TinyCoinNode

```
package it.unipi.p2p.tinycoin;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import peersim.core.GeneralNode;

public class TinyCoinNode extends GeneralNode{

    private NodeType nodeType;
    private MinerType minerType;
    private double balance;
    private List<Block> blockchain;
    private Map<String, Transaction> transPool;

    public TinyCoinNode(String prefix) {
        super(prefix);
        transPool = new HashMap<>();
        blockchain = new ArrayList<>();
    }

    @Override
    public Object clone()
    {
        TinyCoinNode clone = (TinyCoinNode)super.clone();
        clone.setTransPool(new HashMap<>());
        clone.setBlockchain(new ArrayList<>());
        return clone;
    }
```

```
    public void setTransPool(Map<String, Transaction> transPool) {
            this.transPool = transPool;
    }

    public MinerType getMtype() {
            return minerType;
    }

    public void setMtype(MinerType mtype) {
            this.minerType = mtype;
    }

    public boolean isNode() {
            return nodeType==NodeType.NODE;
    }

    public boolean isMiner() {
            return nodeType==NodeType.MINER;
    }

    public boolean isSelfishMiner() {
            return nodeType==NodeType.SELFISH_MINER;
    }

    public List<Block> getBlockchain() {
            return blockchain;
    }

    public void setBlockchain(List<Block> blockchain) {
            this.blockchain = blockchain;
    }

    public double getBalance() {
            return balance;
    }

    public void setBalance(double balance) {
            this.balance = balance;
    }

    public void increaseBalance(double amount) {
            balance += amount;
    }

    public void decreaseBalance(double amount) {
            balance -= amount;
    }

    public void setNodetype(NodeType ntype) {
            this.nodeType = ntype;
    }

    public Map<String, Transaction> getTransPool() {
            return transPool;
    }
}
```

Transaction

```
package it.unipi.p2p.tinycoin;

public class Transaction {

        private final String tid;
        private final TinyCoinNode input;
        private final TinyCoinNode output;
        private final double amount;
        private final double fee;

        public Transaction(String id, TinyCoinNode input, TinyCoinNode output, double amount,
                        double fee) {
                tid = id;
                this.input = input;
                this.output = output;
                this.amount = amount;
                this.fee = fee;
        }
```

```
	public double getAmount() {
		return amount;
	}

	public TinyCoinNode getOutput() {
		return output;
	}

	public double getFee() {
		return fee;
	}

	public String getTid() {
		return tid;
	}

	@Override
	public String toString() {
		return "Transaction " + tid + ": Source = " + input.getID() + ", Destination = " +
				output.getID() + ", amount = " + amount + ", fee = " + fee;
	}
}
```

Subpackage *controls*

Oracle

```
package it.unipi.p2p.tinycoin.controls;
import java.util.Random;

import it.unipi.p2p.tinycoin.MinerType;
import it.unipi.p2p.tinycoin.TinyCoinNode;
import it.unipi.p2p.tinycoin.protocols.MinerProtocol;
import it.unipi.p2p.tinycoin.protocols.SelfishMinerProtocol;
import peersim.config.Configuration;
import peersim.core.Control;
import peersim.core.Network;
import peersim.core.Node;

public class Oracle implements Control {

	private static final String PAR_P2 = "prob_2_miners";
	private static final String PAR_HRCPU = "hr_cpu";
	private static final String PAR_HRGPU = "hr_gpu";
	private static final String PAR_HRFPGA = "hr_fpga";
	private static final String PAR_HRASIC = "hr_asic";
	private static final String PAR_MINER_PROT = "miner_protocol";
	private static final String PAR_SMINER_PROT = "self_miner_protocol";

	private final String prefix;
	private final double p2;
	private double pcpu;
	private double pgpu;
	private double pfpga;
	private double pasic;
	private Random r;
	private final int minerPid;
	private final int selfMinerPid;

	public Oracle(String prefix)
	{
		p2 = Configuration.getDouble(prefix + "." + PAR_P2);
		minerPid = Configuration.getPid(prefix + "." + PAR_MINER_PROT);
		selfMinerPid = Configuration.getPid(prefix + "." + PAR_SMINER_PROT);
		pcpu = pgpu = pfpga = pasic = -1.0;
		this.prefix = prefix;
		r = new Random(0);
	}

	@Override
	public boolean execute() {
	/* Each miner has a given probability of being selected by the oracle. For each type of
	 * miner, I define the probability of the type as P = total_hash_rate_miner_type /
	 * total_hash_rate. So I initialize the probabilities the first time that execute() is
	 * invoked and not in the constructor, because I must be sure that the network has been
	 * initialized */

		if (pcpu == -1.0)
```

```
        {
                boolean initSuccess = initializeProb();
                if (!initSuccess)
                        return true;
        }

        MinerType m1, m2;
        m1 = getMinerType();       // Always choose one miner
        TinyCoinNode mn1 = (TinyCoinNode)chooseMinerNode(m1);
        if (mn1.isMiner())
                ((MinerProtocol)mn1.getProtocol(minerPid)).setSelected(true);
        else //selfish miner
                ((SelfishMinerProtocol)mn1.getProtocol(selfMinerPid)).setSelected(true);
        double rd = r.nextDouble();
        if (rd < p2) {              // two miners solved PoW concurrently
                m2 = getMinerType();
                TinyCoinNode mn2 = (TinyCoinNode)chooseMinerNode(m2);
                if (mn2.isMiner())
                    ((MinerProtocol)mn2.getProtocol(minerPid)).setSelected(true);
                else
                    ((SelfishMinerProtocol)mn2.getProtocol(selfMinerPid)).setSelected(true);
        }
        return false;
}

private boolean initializeProb() {
        int hrcpu = Configuration.getInt(prefix + "." + PAR_HRCPU);
        int hrgpu = Configuration.getInt(prefix + "." + PAR_HRGPU);
        int hrfpga = Configuration.getInt(prefix + "." + PAR_HRFPGA);
        int hrasic = Configuration.getInt(prefix + "." + PAR_HRASIC);
        if (hrcpu < 0 || hrgpu < 0 || hrfpga < 0 || hrasic < 0) {
                System.err.println("Hash rates cannot be negative!");
                return false;
        }
        int ncpu, ngpu, nfpga, nasic;
        ncpu = ngpu = nfpga = nasic = 0;

        for (int i=0; i< Network.size(); i++) {
                TinyCoinNode n = (TinyCoinNode) Network.get(i);
                if (!n.isNode()) {
                        switch(n.getMtype()) {
                        case CPU :  ncpu++;
                                    break;
                        case GPU :  ngpu++;
                                    break;
                        case FPGA : nfpga++;
                                    break;
                case ASIC : nasic++;
                                    break;
                        }
                }
        }
        // I get the probabilities of choosing cpu/gpu/fpga/asic miner
        int thr = (ncpu*hrcpu + ngpu*hrgpu + nfpga*hrfpga + nasic*hrasic);
        pcpu = ((double) hrcpu * ncpu) / ((double)  thr);
        pgpu = ((double) hrgpu * ngpu) / ((double)  thr);
        pfpga = ((double) hrfpga * nfpga) / ((double)  thr);
        pasic = ((double) hrasic * nasic) / ((double)  thr);
        return true;
}


private MinerType getMinerType()
{
        double rd = r.nextDouble();
        if (rd < pcpu)
                return MinerType.CPU;
        else if (rd < pcpu + pgpu)
                return MinerType.GPU;
        else if (rd < pcpu + pgpu + pfpga)
                return MinerType.FPGA;
        else
                return MinerType.ASIC;
}

/** One miner of the given type is chosen randomly. The randomness is achieved by shuffling
 *  the nodes in the network and then taking the first miner node with appropriate type.
 *  @return the miner node which has mined the block
 */
private Node chooseMinerNode(MinerType m) {
```

```
                Network.shuffle();
                for (int i=0; i< Network.size(); i++) {
                        TinyCoinNode n = (TinyCoinNode) Network.get(i);
                                if (n.getMtype() == m)
                                        return n;
                }
                return null;
        }
}
```

TinyObserver

```
package it.unipi.p2p.tinycoin.controls;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.List;
import it.unipi.p2p.tinycoin.Block;
import it.unipi.p2p.tinycoin.TinyCoinNode;
import it.unipi.p2p.tinycoin.protocols.NodeProtocol;
import it.unipi.p2p.tinycoin.protocols.SelfishMinerProtocol;
import peersim.config.Configuration;
import peersim.core.Control;
import peersim.core.Network;

public class TinyObserver implements Control{

    private static final String PAR_NODE_PROT = "node_protocol";
    private static final String PAR_MINER_PROT = "miner_protocol";
    private static final String PAR_SMINER_PROT = "selfish_miner_protocol";
    private static final String PAR_REPETITION = "repetition";
    private static final String PAR_SMINER = "p_self_miner";
    private static final String PAR_HRCPU = "hr_cpu";
    private static final String PAR_HRGPU = "hr_gpu";
    private static final String PAR_HRFPGA = "hr_fpga";
    private static final String PAR_HRASIC = "hr_asic";
    private static final String PAR_ONLYLATENCY = "only_latency";
    private static final String PAR_DELAY = "delay";

    private final int npid;
    private final int mpid;
    private final int smpid;
    private final int repetition;
    private final double psm;
    private final int hrcpu;
    private final int hrgpu;
    private final int hrfpga;
    private final int hrasic;
    private final boolean onlyLatency;
    private final int delay;
    private int cycle;
    private TinyCoinNode node;
    private final String prefix;

    public TinyObserver(String prefix) {
        npid = Configuration.getPid(prefix + "." + PAR_NODE_PROT);
        mpid = Configuration.getPid(prefix + "." + PAR_MINER_PROT);
        smpid = Configuration.getPid(prefix + "." + PAR_SMINER_PROT);
        repetition = Configuration.getInt(prefix + "." + PAR_REPETITION);
        psm = Configuration.getDouble(prefix + "." + PAR_SMINER);
        hrcpu = Configuration.getInt(prefix + "." + PAR_HRCPU);
        hrgpu = Configuration.getInt(prefix + "." + PAR_HRGPU);
        hrfpga = Configuration.getInt(prefix + "." + PAR_HRFPGA);
        hrasic = Configuration.getInt(prefix + "." + PAR_HRASIC);
        onlyLatency = Configuration.getBoolean(prefix + "." + PAR_ONLYLATENCY);
        delay = Configuration.getInt(prefix + "." + PAR_DELAY);
        cycle = 0;
        this.prefix = prefix;
    }

    @Override
    public boolean execute()
    {
        int forks=0;
        int sminers=0;
        FileWriter forkStats = null;
        FileWriter blockchainStats = null;
        FileWriter hashrateStats = null;
```

```
FileWriter rewardStats = null;
FileWriter latencyStats = null;
BufferedWriter bw = null;
cycle++;

try
{
    if (cycle == 1)  // Initialization
    {
        for (int i =0; i< Network.size(); i++) {
                if (((TinyCoinNode)Network.get(i)).isSelfishMiner()) {
                        node = (TinyCoinNode)Network.get(i);
                        break;
                }
        }
        if (onlyLatency == true)
        {
                latencyStats = new FileWriter("docs/statistics/latency_R" +
                    repetition + "_D" + delay + ".dat", false);
                bw = new BufferedWriter(latencyStats);
                bw.write("# Mined_Blocks" + " "  + "Cycle \n");
                bw.close();
        }
        else
        {
                forkStats = new FileWriter("docs/statistics/forks_R" + repetition +
                    "_P" + psm + ".dat", false);
                bw = new BufferedWriter(forkStats);
                bw.write("# Forks_number" + " " + "Cycle \n");
                bw.close();

                blockchainStats = new FileWriter("docs/statistics/blockchain_R" +
                    repetition + "_P" + psm + ".dat", false);
                bw = new BufferedWriter(blockchainStats);
                bw.write("# Honest_blocks" + " " + "Fraudolent_blocks" + " " +
                    "Cycle \n");
                bw.close();

                rewardStats = new FileWriter("docs/statistics/reward_R" + repetition
                    + "_P" + psm + ".dat", false);
                bw = new BufferedWriter(rewardStats);
                bw.write("# Reward_honest" + " " + "Reward_selfish" + " " + "Cycle\n");
                bw.close();

                int hrsminers = 0;
                int hrhonests = 0;
                TinyCoinNode n = null;
                for (int i=0; i< Network.size(); i++) {
                        n = (TinyCoinNode)Network.get(i);
                        if (n.isSelfishMiner())
                                hrsminers += getHashRate(n);
                        else if (n.isMiner())
                                 hrhonests += getHashRate(n);
                }
                hashrateStats = new FileWriter("docs/statistics/hashrate_R" +
                    repetition + "_P" + psm + ".dat", false);
                bw = new BufferedWriter(hashrateStats);
                bw.write("# Honest_HR" + " " + "Fraudolent_HR" + " " +
                   "Probability(SelfishMiner) \n");
                bw.write(hrhonests + "            " + hrsminers + "            " + psm);
                bw.close();
        }
    }

    TinyCoinNode n = null;
    for (int i=0; i< Network.size(); i++) {
            n = (TinyCoinNode)Network.get(i);
            if (n.isSelfishMiner())
                    sminers++;
            else
                    forks+=((NodeProtocol)n.getProtocol(npid)).getNumForks();
    }

    //Statistics about blockchain
    int honestBlocks, fraudolentBlocks, honestReward, fraudolentReward;
    honestBlocks = fraudolentBlocks = honestReward = fraudolentReward = 0;
    List<Block> blockchain = node.getBlockchain();
    for (Block b : blockchain) {
            if (b.getMiner().isSelfishMiner()) {
                fraudolentBlocks++;
```

```
                    fraudolentReward += b.getRevenueForBlock();
                }
                else {
                    honestBlocks++;
                    honestReward += b.getRevenueForBlock();
                }
            }
            // Add the fraudolent blocks (with the associated reward) that are in the private
               blockchain but not yet in the public one, if any. This is an optimistic
               assumption, indeed they could never end up in the blockchain
            List<Block> privateBlockchain =
                ((SelfishMinerProtocol)node.getProtocol(smpid)).getPrivateBlockchain();
            int diff = privateBlockchain.size() - blockchain.size();
            if (diff > 0) {
                fraudolentBlocks += privateBlockchain.size() - blockchain.size();
                for (int i = blockchain.size(); i < blockchain.size() + diff -1 ; i++) {
                    fraudolentReward += privateBlockchain.get(i).getRevenueForBlock();
                }
            }

            if (onlyLatency == true) {
                int totalBlocks = honestBlocks + fraudolentBlocks;
                latencyStats = new FileWriter("docs/statistics/latency_R" + repetition
                    "_D" + delay + ".dat", true);
                bw = new BufferedWriter(latencyStats);
                bw.write(totalBlocks + "            "  + cycle + "\n");
                bw.close();
            }
            else {
                blockchainStats = new FileWriter("docs/statistics/blockchain_R" + repetition
                    + "_P" + psm + ".dat", true);
                bw = new BufferedWriter(blockchainStats);
                bw.write(honestBlocks + "            " +
                    fraudolentBlocks + "            "  + cycle + "\n");
                bw.close();

                // Statistics about forks
                int honests = Network.size()- sminers;
                System.out.println("Honest nodes and miners are " + honests);
                try {
                    forks = forks / honests;  // take the avg
                }
                catch(ArithmeticException e) {
                    forks = ((NodeProtocol)node.getProtocol(npid)).getNumForks();
                }
                System.out.println("Forks are " + forks + " at cycle " + cycle);
                forkStats = new FileWriter("docs/statistics/forks_R" + repetition +
                    "_P" + psm + ".dat", true);
                bw = new BufferedWriter(forkStats);
                bw.write(forks + "            " + cycle + "\n");
                bw.close();

                // Statistics about reward
                rewardStats = new FileWriter("docs/statistics/reward_R" + repetition +
                    "_P" + psm + ".dat", true);
                bw = new BufferedWriter(rewardStats);
                bw.write(honestReward + "            " +
                    fraudolentReward + "            "  + cycle + "\n");
                bw.close();
            }
        }
        catch (IOException e) {
            System.err.println(e);
        }
            return false;
    }

    public int getHashRate(TinyCoinNode n) {
        switch (n.getMtype()) {
                case CPU: return hrcpu;
                case GPU: return hrgpu;
                case FPGA: return hrfpga;
                case ASIC: return hrasic;
                default: return 0;
        }
    }
}
```

Subpackage initializer

NodesInitializer

```java
package it.unipi.p2p.tinycoin.initializer;
import java.util.Random;

import it.unipi.p2p.tinycoin.MinerType;
import it.unipi.p2p.tinycoin.NodeType;
import it.unipi.p2p.tinycoin.TinyCoinNode;
import peersim.config.Configuration;
import peersim.core.Control;
import peersim.core.Network;

public class NodesInitializer implements Control
{

    private static final String PAR_PMINER = "pminer";
    private static final String PAR_PSMINER = "p_self_miner";
    private static final String PAR_PCPU = "pcpu";
    private static final String PAR_PGPU = "pgpu";
    private static final String PAR_PFPGA = "pfpga";
    private static final String PAR_PASIC = "pasic";
    private static final String PAR_MAX_BALANCE = "max_balance";

    // Probability that a network node is a miner.
    private double pminer;
    private double psminer;

    // If the node is a miner, then it has different probabilities of mining through CPU, GPU, FPGA
       or ASIC
    private double pcpu;
    private double pgpu;
    private double pfpga;
    private double pasic;
    private double maxBalance;

    public NodesInitializer(String prefix)
    {
            pminer = Configuration.getDouble(prefix + "." + PAR_PMINER);
            psminer = pminer = Configuration.getDouble(prefix + "." + PAR_PSMINER);
            pcpu = Configuration.getDouble(prefix + "." + PAR_PCPU);
            pgpu = Configuration.getDouble(prefix + "." + PAR_PGPU);
            pfpga = Configuration.getDouble(prefix + "." + PAR_PFPGA);
            pasic = Configuration.getDouble(prefix + "." + PAR_PASIC);
            maxBalance = Configuration.getDouble(prefix + "." + PAR_MAX_BALANCE);
    }

    /** Initializes the nodes in the network based on the probability values received from
     *  Configuration file.
     */
    @Override
    public boolean execute()
    {
        if (pcpu + pgpu + pfpga + pasic != 1) {
            System.err.println("The sum of the probabilities of the mining  HW must be equal to
                100");
            return true;
        }

        TinyCoinNode n = null;
        Random r = new Random(0);
        for (int i=0; i< Network.size(); i++) {
            n = (TinyCoinNode)Network.get(i);
            double b = Math.random()*maxBalance;
            n.setBalance(b);
            double drandom = r.nextDouble();
            if (drandom < pminer) { // the node is a miner
                drandom = r.nextDouble();
                if (drandom < psminer) //Node is a selfish miner
                    n.setNodetype(NodeType.SELFISH_MINER);
                else
                    n.setNodetype(NodeType.MINER);
                drandom = r.nextDouble();
                if (drandom < pcpu)
                    n.setMtype(MinerType.CPU);
                else if (drandom < pcpu + pgpu)
                    n.setMtype(MinerType.GPU);
                else if (drandom < pcpu + pgpu + pfpga)
```

```
                    n.setMtype(MinerType.FPGA);
                else
                    n.setMtype(MinerType.ASIC);
            }
            else
            {
                n.setNodetype(NodeType.NODE);
                n.setMtype(null);
            }
        }
        return false;
    }
}
```

Subpackage protocols

NodeProtocol

```
package it.unipi.p2p.tinycoin.protocols;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import it.unipi.p2p.tinycoin.Block;
import it.unipi.p2p.tinycoin.TinyCoinNode;
import it.unipi.p2p.tinycoin.Transaction;
import peersim.cdsim.CDProtocol;
import peersim.config.Configuration;
import peersim.config.FastConfig;
import peersim.core.Linkable;
import peersim.core.Network;
import peersim.core.Node;
import peersim.edsim.EDProtocol;
import peersim.transport.Transport;

public class NodeProtocol implements CDProtocol, EDProtocol{

    private static final String PAR_P_TRANS = "transaction_prob";
    private static final String PAR_SMINER = "self_miner_prot";

    private double transProb;
    private int numTrans;
    private int smpid;
    private boolean fork;
    private Block forked;
    private int numForks;
    private List<Block> missedBlocks;
    private int limit;

    public NodeProtocol(String prefix)
    {
        transProb = Configuration.getDouble(prefix + "." + PAR_P_TRANS);
        numTrans = 0;
        smpid = Configuration.getPid(prefix + "." + PAR_SMINER);
        fork = false;
        forked = null;
        numForks = 0;
        missedBlocks = new ArrayList<>();
        limit = 20;
    }

    @Override
    public Object clone() {
        NodeProtocol np = null;
        try {
            np = (NodeProtocol)super.clone();
            np.setTransProb(transProb);
            np.setNumTrans(0);
            np.setSmpid(smpid);
            np.setFork(false);
            np.setForked(null);
            np.setNumForks(0);
            np.setMissedBlocks(new ArrayList<>());
            np.setLimit(limit);
        }
        catch(CloneNotSupportedException  e) {
            System.err.println(e);
        }
```

```
        return np;
    }

    @Override
    public void nextCycle(Node node, int pid)
    {
        TinyCoinNode tnode = (TinyCoinNode) node;
        double balance = tnode.getBalance();
        // I assume that if a node has less than 1 coin cannot make a transaction
        // (Substitutes the test for empty balance, and allows to avoid very small, fractional
        // transactions)
        if (balance < 1) {
                return;
        }
        double r = Math.random();
        // At each cycle, each node generates a transaction with a given probability
        if (r < transProb) {
            String tid = node.getID() + "@" + numTrans;
            numTrans++;
            // Randomly choose one recipient
            Network.shuffle();
            TinyCoinNode recipient = (TinyCoinNode) Network.get(0);
            double totalSpent = Math.random() * balance;
            double amount = totalSpent * (9.0/10.0);
            double fee = totalSpent - amount;
            Transaction t = new Transaction(tid, tnode, recipient, amount, fee);
            System.out.println(t.toString());
            // Transaction has been created, so update balance and insert into local pool of
               unconfirmed transactions
            tnode.getTransPool().put(tid, t);
            balance -= totalSpent;
            // Send the transaction to all neighbor nodes
            sendTransactionToNeighbors(node, pid, t);
        }
    }

    @Override
    public void processEvent(Node node, int pid, Object event)
    {
        if (event instanceof Transaction) {
            Transaction t = (Transaction) event;
            Map<String, Transaction> transPool = ((TinyCoinNode) node).getTransPool();
            String tid = t.getTid();
            // If never received the transaction, broadcast it to the neighbors
            if (!transPool.containsKey(tid)) {
                transPool.put(tid, t);
                sendTransactionToNeighbors(node, pid, t);
            }
        }
        else if (event instanceof Block) {
            TinyCoinNode tnode = (TinyCoinNode)node;
            List<Block> blockchain = tnode.getBlockchain();
            Block b = (Block)event;
            String last = blockchain.size()==0 ? null : blockchain.get(blockchain.size()-
                1).getBid();
            if (tnode.isSelfishMiner()) {  //Selfish miner receives a new block from a honest node
                SelfishMinerProtocol smp = (SelfishMinerProtocol)node.getProtocol(smpid);
                List<Block> privateBlockchain = smp.getPrivateBlockchain();
                int privateBranchLength = smp.getPrivateBranchLength();
                int prevDiff = privateBlockchain.size() - blockchain.size();
                if ( last == b.getParent()) {
                    blockchain.add(b);
                    if (!missedBlocks.isEmpty())
                        attachMissedBlocksToBlockchain(tnode);
                    tnode.increaseBalance(b.getTransactionsAmountIfRecipient(tnode));
                    if (b.getMiner() == tnode) // Added this check, should be redundant
                        tnode.increaseBalance(b.getRevenueForBlock());
                        switch (prevDiff) {
                        case(0):
                            if (onlyAddTheBlock(privateBlockchain, blockchain))
                                privateBlockchain.add(b);   //simply add one block
                            else
// also delete last block of private blockchain to make the two exactly equal
                                smp.copyPublicBlockchain(tnode);
                            smp.setPrivateBranchLength(0);
                            sendBlockToNeighbors(node, pid, b);
                            break;
                        case(1):
                            Block sb = privateBlockchain.get(privateBlockchain.size() - 1);
                            sendBlockToNeighbors(node, pid, sb);
```

```
                        break;
                    case(2):
                        for (int i = privateBranchLength; i > 0; i--) {
                            sb = privateBlockchain.get(privateBlockchain.size() - i);
                            sendBlockToNeighbors(node, pid, sb);
                        }
                        smp.copyPrivateBlockchain(tnode);
                        smp.setPrivateBranchLength(0);
                        break;
                    default:
                        if (prevDiff > 2 && privateBranchLength > 0)
                        {
                            sb = privateBlockchain.get(privateBlockchain.size() -
                                privateBranchLength);
                            sendBlockToNeighbors(node, pid, sb);
                            smp.setPrivateBranchLength(privateBranchLength - 1);
                        }
                    }
                }
                else
                    addMissedBlockToPool(b);
            }
            else {
            // If the parent field of the block is valid, then the honest node adds the block
            // to its blockchain and removes the transactions inside the block from the pool.
                if ( last == b.getParent() ||
                    (fork == true && forked.getBid() == b.getParent())) {
                    if (fork == true) {
                        if (forked.getBid() == b.getParent()) {
                            Block lastb = blockchain.get(blockchain.size()-1);
                            blockchain.remove(lastb);
                            addTransactionsToPool(tnode, lastb);
                            tnode.decreaseBalance(lastb.getTransactionsAmountIfRecipient(
                                tnode));
                            if (tnode == lastb.getMiner())
                                tnode.decreaseBalance(lastb.getRevenueForBlock());
                            blockchain.add(forked);
                            // No need to add the revenue for mining the block, because a honest
                            // miner always takes the revenue as soon as it mines the block
                            tnode.increaseBalance(forked.getTransactionsAmountIfRecipient(
                                tnode));
                        }
                        fork = false;  // Fork is resolved, whichever is the extended branch
                        forked = null;
                    }
                    blockchain.add(b);
                    if (!missedBlocks.isEmpty())
                        attachMissedBlocksToBlockchain(tnode);
                    tnode.increaseBalance(b.getTransactionsAmountIfRecipient(tnode));
                    removeTransactionsFromPool(tnode, b);
                    // Finally (if block is valid) send the block to all  the neighbor nodes
                    sendBlockToNeighbors(node, pid, b);
                }
                else if (blockchain.size() >= 2 &&
                    blockchain.get(blockchain.size()-2).getBid() == b.getParent() &&
                    blockchain.get(blockchain.size()-1).getBid() != b.getBid() &&
                    fork == false) {
                    fork = true;
                    forked = b;
                    numForks++;
                    sendBlockToNeighbors(node, pid, b);
                    solveForkWithMissedBlocks(tnode);
                }
                else if (last != b.getParent())
                    addMissedBlockToPool(b);
            }
        }
    }

    public void removeTransactionsFromPool(TinyCoinNode tn, Block b) {
            Map<String, Transaction> transPool = tn.getTransPool();
            for (Transaction t : b.getTransactions()) {
                transPool.remove(t.getTid());
            }
    }

    public void addTransactionsToPool(TinyCoinNode tn, Block b) {
            Map<String, Transaction> transPool = tn.getTransPool();
            for (Transaction t : b.getTransactions()) {
                transPool.putIfAbsent(t.getTid(), t);
```

```
        }
}

public void setNumForks(int numForks) {
    this.numForks = numForks;
}

/** Sends a transaction t to the protocol pid of all the neighbor nodes
 *
 * @param sender The sender node
 * @param pid The id of the protocol the message is directed to
 * @param t The transaction to be sent
 */
 public void sendTransactionToNeighbors(Node sender, int pid, Transaction t) {
     int linkableID = FastConfig.getLinkable(pid);
     Linkable linkable = (Linkable) sender.getProtocol(linkableID);
     for (int i =0; i<linkable.degree(); i++) {
         Node peer = linkable.getNeighbor(i);
         ((Transport)sender.getProtocol(FastConfig.getTransport(pid)))
             .send(sender, peer, t, pid);
     }
 }


 /** Sends a block b to the protocol pid of all the neighbor nodes
  *
  * @param sender The sender node
  * @param pid The id of the protocol the message is directed to
  * @param b The block to be sent
  */
 public void sendBlockToNeighbors(Node sender, int pid, Block b) {
     int linkableID = FastConfig.getLinkable(pid);
     Linkable linkable = (Linkable) sender.getProtocol(linkableID);
     for (int i =0; i<linkable.degree(); i++) {
         Node peer = linkable.getNeighbor(i);
         ((Transport)sender.getProtocol(FastConfig.getTransport(pid)))
             .send(sender, peer, b, pid);
     }
 }

 public boolean solveForkWithMissedBlocks(TinyCoinNode tn) {
     List <Block> blockchain = tn.getBlockchain();
     for (int i=0; i< missedBlocks.size(); i++) {
         if (missedBlocks.get(i).getParent() == forked.getBid()) {
             Block lastb = blockchain.get(blockchain.size()-1);
             blockchain.remove(lastb);
             tn.decreaseBalance(lastb.getTransactionsAmountIfRecipient(tn));
             addTransactionsToPool(tn, lastb);
             blockchain.add(forked);
             Block head = missedBlocks.remove(i);
             blockchain.add(head);
             removeTransactionsFromPool(tn, head);
             tn.increaseBalance(head.getTransactionsAmountIfRecipient(tn));
             fork = false;
             forked = null;
             attachMissedBlocksToBlockchain(tn);
             return true;
         }
     }
     return false;
}

/** Scans the list of missed blocks trying to find some blocks that can be attached to the head
 *  of the blockchain
 */
public void attachMissedBlocksToBlockchain(TinyCoinNode tn)
{
    List <Block> blockchain = tn.getBlockchain();
    Block head = blockchain.get(blockchain.size()-1);
    int i=0;
    while ( i < missedBlocks.size()) {
        if (missedBlocks.get(i).getParent() == head.getBid()) {
            head = missedBlocks.remove(i);
            blockchain.add(head);
            removeTransactionsFromPool(tn, head);
            tn.increaseBalance(head.getTransactionsAmountIfRecipient(tn));
            i = 0;  // The head of the blockchain changed,so restart scanning the missed blocks
        }
        else
            i++;
```

```
        }
    }
    public void addMissedBlockToPool(Block missed) {
        if (missedBlocks.size() == limit)       // If reached the limit, empty it
            missedBlocks.removeAll(missedBlocks);
        if (!missedBlocks.contains(missed))
            missedBlocks.add(missed);
    }

    public int getNumForks() {
        return numForks;
    }

    public void setSmpid(int smpid) {
        this.smpid = smpid;
    }

    public void setNumTrans(int numTrans) {
        this.numTrans = numTrans;
    }

    public void setFork(boolean fork) {
        this.fork = fork;
    }

    public void setForked(Block forked) {
        this.forked = forked;
    }

    public double getTransProb() {
        return transProb;
    }

    public void setTransProb(double transProb) {
        this.transProb = transProb;
    }

    public void setMissedBlocks(List<Block> missedBlocks) {
        this.missedBlocks = missedBlocks;
    }

    public void setLimit(int limit) {
        this.limit = limit;
    }

    private boolean onlyAddTheBlock(List<Block> privateBlockchain , List<Block> blockchain )
    {
        if (privateBlockchain.size() == 0 ||
            blockchain.get(blockchain.size() -1).getParent() ==
            privateBlockchain.get(privateBlockchain.size()-1).getBid())
            return true;
        else
            return false;
    }
}
```

MinerProtocol

```
package it.unipi.p2p.tinycoin.protocols;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

import it.unipi.p2p.tinycoin.Block;
import it.unipi.p2p.tinycoin.TinyCoinNode;
import it.unipi.p2p.tinycoin.Transaction;
import peersim.cdsim.CDProtocol;
import peersim.config.Configuration;
import peersim.config.FastConfig;
import peersim.core.Linkable;
import peersim.core.Node;
import peersim.transport.Transport;


public class MinerProtocol implements CDProtocol
{
    private static final String PAR_MAX_TRANS_BLOCK = "max_trans_block";
```

```java
    private static final String PAR_REWARD = "reward";
    private static final String PAR_NODE_PROT = "node_protocol";
    private int minedBlocks;
    private boolean selected;
    private int maxTransPerBlock;
    private double reward;
    private int nodeProtocol;

    public MinerProtocol(String prefix) {
        minedBlocks = 0;
        maxTransPerBlock = Configuration.getInt(prefix + "." + PAR_MAX_TRANS_BLOCK);
        reward = Configuration.getDouble(prefix + "." + PAR_REWARD);
        nodeProtocol = Configuration.getPid(prefix + "." + PAR_NODE_PROT);
    }

    @Override
    public Object clone() {
        MinerProtocol mp = null;
        try {
            mp = (MinerProtocol)super.clone();
            mp.setMinedBlocks(0);
            mp.setSelected(false);
            mp.setMaxTransPerBlock(maxTransPerBlock);
            mp.setReward(reward);
            mp.setNodeProtocol(nodeProtocol);
        }
        catch(CloneNotSupportedException  e) {
            System.err.println(e);
        }
            return mp;
    }

    public boolean isSelected() {
        return selected;
    }

    public void setSelected(boolean selected) {
        this.selected = selected;
    }

    public void setMaxTransPerBlock(int maxTransPerBlock) {
        this.maxTransPerBlock = maxTransPerBlock;
    }

    public void setReward(double reward) {
        this.reward = reward;
    }

    public void setNodeProtocol(int nodeProtocol) {
         this.nodeProtocol = nodeProtocol;
    }

    public int getMinedBlocks() {
        return minedBlocks;
    }

    public void setMinedBlocks(int minedBlocks) {
        this.minedBlocks = minedBlocks;
    }

    @Override
    public void nextCycle(Node node, int pid)
    {
        TinyCoinNode tnode = (TinyCoinNode)node;
        if (!tnode.isMiner())
            return;
        if (isSelected())
        {
             setSelected(false);
             Map<String, Transaction> transPool = tnode.getTransPool();
             List<Block> blockchain = tnode.getBlockchain();
             // Create a new block and announce it to all the neighbors
             Block b = createBlock(transPool, tnode, blockchain);
             blockchain.add(b);
//the reward for mining the block is given to the miner
             tnode.increaseBalance(b.getRevenueForBlock());
             tnode.increaseBalance(b.getTransactionsAmountIfRecipient(tnode));
             sendBlockToNeighbors(node, nodeProtocol, b);
             System.out.println("Mined a block!");
        }
```

```
    }

    /** Sends a block b to the protocol pid of all the neighbor nodes
     *
     * @param sender The sender node
     * @param pid The id of the protocol the message is directed to
     * @param b The block to be sent
     */
    public void sendBlockToNeighbors(Node sender, int pid, Block b) {
        int linkableID = FastConfig.getLinkable(pid);
        Linkable linkable = (Linkable) sender.getProtocol(linkableID);
        for (int i =0; i<linkable.degree(); i++) {
            Node peer = linkable.getNeighbor(i);
            ((Transport)sender.getProtocol(FastConfig.getTransport(pid)))
                .send(sender, peer, b, pid);
        }
    }

    private Block createBlock(Map<String, Transaction> transPool, TinyCoinNode tnode,
        List<Block> blockchain) {
        minedBlocks++;
        int transInBlock = Math.min(transPool.size(), maxTransPerBlock);
        String bid = "B" + tnode.getID() + minedBlocks;
        String parent = blockchain.size()== 0
            ? null : blockchain.get(blockchain.size()-1).getBid();
        List<Transaction> trans = new ArrayList<>(transInBlock);
        Iterator<String> iter = tnode.getTransPool().keySet().iterator();
        for (int i=0; i< transInBlock; i++) {
            String key = iter.next();
            Transaction t = transPool.get(key);
            iter.remove();
            trans.add(t);
        }
        return new Block(bid, parent, tnode, trans, reward);
    }
}
```

SelfishMinerProtocol

```
package it.unipi.p2p.tinycoin.protocols;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

import it.unipi.p2p.tinycoin.Block;
import it.unipi.p2p.tinycoin.TinyCoinNode;
import it.unipi.p2p.tinycoin.Transaction;
import peersim.cdsim.CDProtocol;
import peersim.config.Configuration;
import peersim.config.FastConfig;
import peersim.core.Linkable;
import peersim.core.Node;
import peersim.edsim.EDProtocol;
import peersim.transport.Transport;

public class SelfishMinerProtocol implements CDProtocol, EDProtocol {

    private static final String PAR_MAX_TRANS_BLOCK = "max_trans_block";
    private static final String PAR_REWARD = "reward";
    private static final String PAR_NODE_PROT = "node_protocol";

    private int minedBlocks;
    private boolean selected;
    private int maxTransPerBlock;
    private double reward;
    private int nodeProtocol;
    private List<Block> privateBlockchain;
    private int privateBranchLength;

    public SelfishMinerProtocol(String prefix) {
        minedBlocks = 0;
        maxTransPerBlock = Configuration.getInt(prefix + "." + PAR_MAX_TRANS_BLOCK);
        reward = Configuration.getDouble(prefix + "." + PAR_REWARD);
        nodeProtocol = Configuration.getPid(prefix + "." + PAR_NODE_PROT);
        privateBlockchain = new ArrayList<>();
        privateBranchLength = 0;
    }
```

```
    @Override
    public Object clone() {
        SelfishMinerProtocol smp = null;
        try {
            smp = (SelfishMinerProtocol)super.clone();
            smp.setMinedBlocks(0);
            smp.setSelected(false);
            smp.setMaxTransPerBlock(maxTransPerBlock);
            smp.setReward(reward);
            smp.setNodeProtocol(nodeProtocol);
            smp.setPrivateBlockchain(new ArrayList<>());
            smp.setPrivateBranchLength(0);
        }
        catch(CloneNotSupportedException  e) {
            System.err.println(e);
        }
            return smp;
    }

    public boolean isSelected() {
        return selected;
    }

    public void setSelected(boolean selected) {
        this.selected = selected;
    }

    public void setPrivateBranchLength(int privateBranchLength) {
        this.privateBranchLength = privateBranchLength;
    }

    public void setPrivateBlockchain(List<Block> privateBlockchain) {
        this.privateBlockchain = privateBlockchain;
    }

    public void setMaxTransPerBlock(int maxTransPerBlock) {
        this.maxTransPerBlock = maxTransPerBlock;
    }

    public void setReward(double reward) {
        this.reward = reward;
    }

    public void setNodeProtocol(int nodeProtocol) {
        this.nodeProtocol = nodeProtocol;
    }

    public int getMinedBlocks() {
        return minedBlocks;
    }

    public void setMinedBlocks(int minedBlocks) {
        this.minedBlocks = minedBlocks;
    }

    @Override
    public void nextCycle(Node node, int pid)
    {
        if (isSelected())
        {
            setSelected(false);
            TinyCoinNode tnode = (TinyCoinNode)node;
            Map<String, Transaction> transPool = tnode.getTransPool();
            // Create a new block
            Block b = createBlock(transPool, tnode);
            String last = privateBlockchain.size()==0 ? null :
                privateBlockchain.get(privateBlockchain.size()-1).getBid();
            if (isValidBlock(last, b)) {
// Announce the block either to the selfish miners or to all the neighbor nodes based on convenience
                List<Block> blockchain = tnode.getBlockchain();
                int prevDifference = privateBlockchain.size() - blockchain.size();
                privateBlockchain.add(b);
                privateBranchLength ++;
// If there was a fork, publish both blocks of the private branch to win the tie break
                if (prevDifference == 0 && privateBranchLength == 2 ) {
                    copyPrivateBlockchain(tnode);
                    for (int i = privateBranchLength; i > 0; i--)
                        sendBlockToNeighbors(node, nodeProtocol,
```

```
                        privateBlockchain.get(privateBlockchain.size()-i));
                privateBranchLength = 0;
                }
                else
                    sendBlockToSelfishMiners(node, pid, b);
                System.out.println("Mined a block!" );
            }
        }
    }

    public boolean isValidBlock(String last, Block toBeAdded) {
        if ( last != toBeAdded.getParent()) {
            try {
                throw new Exception("Parent node of the new block is different from the last"
                    + "node of the blockchain");
            } catch (Exception e) {
                return false;
            }
        }
        return true;
    }

    /** Sends a block b to the protocol pid of all the neighbor nodes
     *
     * @param sender The sender node
     * @param pid The id of the protocol the message is directed to
     * @param b The block to be sent
     */
    public void sendBlockToNeighbors(Node sender, int pid, Block b) {
        int linkableID = FastConfig.getLinkable(pid);
        Linkable linkable = (Linkable) sender.getProtocol(linkableID);
        for (int i =0; i<linkable.degree(); i++) {
            Node peer = linkable.getNeighbor(i);
            ((Transport)sender.getProtocol(FastConfig.getTransport(pid)))
                .send(sender, peer, b, pid);
        }
    }

    /** Sends a block b to the protocol pid of the neighbor nodes which are selfish miners
     *
     * @param sender The sender node
     * @param pid The id of the protocol the message is directed to
     * @param b The block to be sent
     */
    public void sendBlockToSelfishMiners(Node sender, int pid, Block b) {
        int linkableID = FastConfig.getLinkable(pid);
        Linkable linkable = (Linkable) sender.getProtocol(linkableID);
            for (int i =0; i<linkable.degree(); i++) {
                TinyCoinNode peer = (TinyCoinNode)linkable.getNeighbor(i);
                if ( peer.isSelfishMiner())
                    ((Transport)sender.getProtocol(FastConfig.getTransport(pid)))
                        .send(sender, peer, b, pid);
                }
            }

    @Override
    public void processEvent(Node node, int pid, Object event) {
    // This protocol only receives blocks sent by selfish miners. The blocks have to be added to the
       private blockchain
        TinyCoinNode tnode = (TinyCoinNode)node;
        List<Block> blockchain = tnode.getBlockchain();
        int prevDifference = privateBlockchain.size() - blockchain.size();
        Block b = (Block)event;
        String last = privateBlockchain.size()==0 ? null :
            privateBlockchain.get(privateBlockchain.size()-1).getBid();
        if (isValidBlock(last, b)) {
            privateBlockchain.add(b);
            privateBranchLength++;
            if (prevDifference == 0 && privateBranchLength == 2 ) {
                copyPrivateBlockchain(tnode);
                for (int i = privateBranchLength; i > 0; i--)
                    sendBlockToNeighbors(node, nodeProtocol,
                        privateBlockchain.get(privateBlockchain.size()-i));
                privateBranchLength = 0;
            }
            sendBlockToSelfishMiners(node, pid, b);
        }
    }

    private Block createBlock(Map<String, Transaction> transPool, TinyCoinNode tnode) {
```

```
        minedBlocks++;
        int transInBlock = Math.min(transPool.size(), maxTransPerBlock);
        String bid = "B" + tnode.getID() + minedBlocks;
        String parent = privateBlockchain.size()== 0
            ? null : privateBlockchain.get(privateBlockchain.size()-1).getBid();
        List<Transaction> trans = new ArrayList<>(transInBlock);
        Iterator<String> iter = tnode.getTransPool().keySet().iterator();
        for (int i=0; i< transInBlock; i++) {
            String key = iter.next();
            Transaction t = transPool.get(key);
            iter.remove();
            trans.add(t);
            if (t.getOutput() == tnode) {
                tnode.increaseBalance(t.getAmount());
            }
        }
        return new Block(bid, parent, tnode, trans, reward);
    }

    public List<Block> getPrivateBlockchain() {
        return privateBlockchain;
    }

    public int getPrivateBranchLength() {
        return privateBranchLength;
    }

    /** Update the public blockchain to be a copy of the private one, discarding the last item of
     *  the public one
     */
    public void copyPrivateBlockchain(TinyCoinNode tnode) {
        List<Block> blockchain = tnode.getBlockchain();
        Block last = blockchain.get(blockchain.size() - 1);
        blockchain.remove(last); //remove last item
        tnode.decreaseBalance(last.getTransactionsAmountIfRecipient(tnode));
        ((NodeProtocol)tnode.getProtocol(nodeProtocol)).addTransactionsToPool(tnode, last);
        if (tnode == last.getMiner())
            tnode.decreaseBalance(last.getRevenueForBlock()); //remove block reward
        for (int i = privateBranchLength; i > 0; i--) {
            Block b = privateBlockchain.get(privateBlockchain.size() - i);
            blockchain.add(b);
            ((NodeProtocol)tnode.getProtocol(nodeProtocol)).removeTransactionsFromPool(tnode, b);
            tnode.increaseBalance(b.getTransactionsAmountIfRecipient(tnode));
            if (tnode == b.getMiner())
                tnode.increaseBalance(b.getRevenueForBlock()); //get reward for the added block
        }
    }

    /** Update the private blockchain to be a copy of the public one, discarding the last item of
      * the private one
      */
    public void copyPublicBlockchain(TinyCoinNode tnode) {
        List<Block> blockchain = tnode.getBlockchain();
        if (privateBlockchain.size() != 0)
            privateBlockchain.remove(privateBlockchain.size() - 1); //remove last item
        for (int i = privateBranchLength; i >= 0; i--) {
             Block b = blockchain.get(blockchain.size() - (i+1));
             privateBlockchain.add(b);
        }
    }
}
```

Package test.*it.unipi.p2p.tinycoin*

TestRunner

```
package test.it.unipi.p2p.tinycoin;

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner
{
        public static void main(String[] args) {
                Result result = JUnitCore.runClasses(MissedBlocksTest.class);
                 for (Failure failure : result.getFailures()) {
                 System.out.println(failure.toString());
```

```
            }
            System.out.println(result.wasSuccessful());
        }
}
```

MissedBlocksTest

```
package test.it.unipi.p2p.tinycoin;

import static org.junit.Assert.assertEquals;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import org.junit.Test;
import it.unipi.p2p.tinycoin.Block;
import it.unipi.p2p.tinycoin.TinyCoinNode;
import it.unipi.p2p.tinycoin.Transaction;
import peersim.core.Node;

public class MissedBlocksTest {

    private boolean fork;
    private Block forked;
    private List<Block> missedBlocks;
    private int limit;
    List<Block> blockchain;

    public MissedBlocksTest()
    {
        fork = false;
        forked = null;
        missedBlocks = new ArrayList<>();
        limit = 20;
        blockchain = new ArrayList<>();
    }

    @Test
    public void testWrongArrivalOrder() {
        // only the order of blocks is under test
        Block first = new Block("b1", null, null, new ArrayList<Transaction>(), 0);
        Block second = new Block("b2", "b1", null, new ArrayList<Transaction>(), 0);
        Block third = new Block("b3", "b2", null, new ArrayList<Transaction>(), 0);
        Block fourth = new Block("b4", "b3", null, new ArrayList<Transaction>(), 0);
        Block fifth = new Block("b5", "b4", null, new ArrayList<Transaction>(), 0);

        processEvent(null, 0, first);
        processEvent(null, 0, third);
        processEvent(null, 0, fourth);
        processEvent(null, 0, second);
        processEvent(null, 0, fifth);
        assertEquals(blockchain.size(),5);
    }

    @Test
    public void testForkResolutionWithMissedBlocksPool() {
        Block b1 = new Block("b1", null, null, new ArrayList<Transaction>(), 0);
        Block b2 = new Block("b2", "b1", null, new ArrayList<Transaction>(), 0);
        Block b3 = new Block("b3", "b1", null, new ArrayList<Transaction>(), 0);
        Block b4 = new Block("b4", "b3", null, new ArrayList<Transaction>(), 0);
        Block b5 = new Block("b5", "b4", null, new ArrayList<Transaction>(), 0);

        processEvent(null, 0, b4);
        processEvent(null, 0, b5);
        processEvent(null, 0, b1);
        processEvent(null, 0, b2);
        processEvent(null, 0, b3);
        assertEquals(blockchain.size(),4);
    }

    /** Scans the list of missed blocks trying to find some blocks that can be attached to the head
        of the blockchain
         */
    public void attachMissedBlocks()
    {
        Block head = blockchain.get(blockchain.size()-1);
        int i=0;
        while (i < missedBlocks.size()) {
            if (missedBlocks.get(i).getParent() == head.getBid()) {
                head = missedBlocks.remove(i);
```

```
                blockchain.add(head);
                i = 0;
            }
            else
                i++;
        }
    }

    public void addMissedBlock(Block missed) {
        if (missedBlocks.size() == limit)
            missedBlocks.removeAll(missedBlocks);
        if (!missedBlocks.contains(missed))
            missedBlocks.add(missed);
    }

    public void removeTransactionsFromPool(TinyCoinNode tn, Block b) {
        Map<String, Transaction> transPool = tn.getTransPool();
        for (Transaction t : b.getTransactions()) {
            transPool.remove(t.getTid());
        }
    }

    public void processEvent(Node node, int pid, Object event)
    {
        if (event instanceof Block) {
            Block b = (Block)event;
            String last = blockchain.size()==0 ? null : blockchain.get(blockchain.size()-
                1).getBid();
            if ( last == b.getParent() ||
               (fork == true && forked.getBid() == b.getParent())) {
               if (fork == true) {
                   if (forked.getBid() == b.getParent()) {
                       Block lastb = blockchain.get(blockchain.size()-1);
                       blockchain.remove(lastb);
                       blockchain.add(forked);
                   }
                   fork = false;
                   forked = null;
               }
               blockchain.add(b);
               if (!missedBlocks.isEmpty())
                   attachMissedBlocks();
           }
           else if (blockchain.size() >= 2 &&
               blockchain.get(blockchain.size()-2).getBid() == b.getParent() &&
               blockchain.get(blockchain.size()-1).getBid() != b.getBid() &&
               fork == false)
               {
                   fork = true;
                   forked = b;
                   solveForkWithMissedBlocks();
               }
               else if (last != b.getParent())
                   addMissedBlock(b);
           }
    }

    public boolean solveForkWithMissedBlocks() {
        for (int i=0; i< missedBlocks.size(); i++) {
            if (missedBlocks.get(i).getParent() == forked.getBid()) {
                Block lastb = blockchain.get(blockchain.size()-1);
                blockchain.remove(lastb);
                blockchain.add(forked);
                Block head = missedBlocks.remove(i);
                blockchain.add(head);
                fork = false;
                forked = null;
                attachMissedBlocks();
                return true;
            }
        }
        return false;
    }
}
```

Python scripts

start_simulation.py

```
import os
import sys

if len(sys.argv) != 2:
    repetitions = 1
else:
    repetitions = int(sys.argv[1])

# Set parameters values
size = '1000'
cycles = '30'
cycle_length = '1000'
drop = '0'
max_trans_per_block = '50'
reward = '10'
trans_prob = '0.15'
prob_cpu = '0.10'
prob_gpu = '0.30'
prob_fpga = '0.30'
prob_asic = '0.30'
prob_miner = ['0.30']
prob_sminer = [ '0.10', '0.30', '0.50', '0.70','0.90', '1.00']
prob_2miners = ['0.05']
delay = ['0', '10', '50', '90']


# Read in the file
with open('tinycoin_config_for_script.txt', 'r') as file :
  config = file.read()

# Substitute in the configuration file the parameters that have a single value
config = config.replace('SIZE', size)
config = config.replace('CYCLES', cycles)
config = config.replace('CYCLE_LENGTH', cycle_length)
config = config.replace('DROP', drop)
config = config.replace('MAX_TRANS_PER_BLOCK', max_trans_per_block)
config = config.replace('REWARD', reward)
config = config.replace('TRANS_PROB', trans_prob)
config = config.replace('PROB_CPU', prob_cpu)
config = config.replace('PROB_GPU', prob_gpu)
config = config.replace('PROB_FPGA', prob_fpga)
config = config.replace('PROB_ASIC', prob_asic)
config = config.replace('PROB_MINER', prob_miner[0])
config = config.replace('PROB_2MINERS', prob_2miners[0])
configd = config                                          # Use configd to configure
simulations with various values of delay
configd = configd.replace('ONLYLATENCY', 'true')
configd = configd.replace('PROB_SMINER', '0.10')
config = config.replace('DELAY', '0')
config = config.replace('ONLYLATENCY', 'false')


repetitions = range(1,repetitions+1)
for count in repetitions:
# Replace the parameters in the configuration file and run the simulation
    config_overwrite = config
    config_overwrite = config_overwrite.replace('REPETITION', str(count))
    for p in prob_sminer:
        config_overwrite2 = config_overwrite
        config_overwrite2 = config_overwrite2.replace('PROB_SMINER', p)
        # Write the file out again
        with open('tinycoin_config_overwritten.txt', 'w+') as file:
            file.write(config_overwrite2)
        os.system('java -cp lib/jep-2.3.0.jar:lib/djep-1.0.0.jar:lib/peersim-
1.0.5.jar:lib/tinycoin.jar peersim.Simulator tinycoin_config_overwritten.txt ')

    # Run the simulation also with different values of delay
    for d in delay:
```

```
        config_overwrite = configd
        config_overwrite = config_overwrite.replace('REPETITION', str(count))
        config_overwrite = config_overwrite.replace('DELAY', d)
        with open('tinycoin_config_overwritten.txt', 'w+') as file:
            file.write(config_overwrite)
        os.system('java -cp lib/jep-2.3.0.jar:lib/djep-1.0.0.jar:lib/peersim-
1.0.5.jar:lib/tinycoin.jar peersim.Simulator tinycoin_config_overwritten.txt ')


# Make the averages of the various statistics
for p in prob_sminer:
    os.system('python build_avg_statistics.py ' + p)

for d in delay:
    os.system('python build_avg_statistics.py d' + d)

# Plot the averaged statistics
os.system('python plot_avg_statistics.py')
```

build_avg_statistics.py

```
import glob
import sys

if sys.argv[1].startswith("d"):  # Building latency statistics
    d = sys.argv[1][1:]
    pattern = 'D' + d

    # Scan files reporting the number of blocks in the blockchain for different delays and compute
the average for each cycle
    files_scanned = 0
    blocks = []
    to_scan = [f for f in glob.glob('docs/statistics/latency_R*') if pattern in f ]
    for filename in to_scan:
        with open(filename) as in_file:
            cycle = 0
            for line in in_file:
                if files_scanned == 0:  # the first time we have to populate the arrays
                    if line.startswith("#"):
                        blocks.append(None)
                    else:
                        a = line.split('            ')   # exactly 12 spaces
                        blocks.append(int(a[0]))
                else:
                    if line.startswith("#"):
                        continue
                    else:                                   # Calculate the average incrementally
                        if cycle != 0:
                            a = line.split('            ')
                            blocks[cycle] = ( (blocks[cycle] * files_scanned) + int(a[0]) ) /
                                (files_scanned + 1)
                cycle+=1
            files_scanned+=1


    with open('docs/statistics/avg/latency_D' + d + '_avg.dat', 'w+') as out_file:
                outs= '# Cycle blocks \n'
                for count in range(1, len(blocks)):
                    outs = outs + str(count) + '     ' + str(blocks[count]) + '\n'
                out_file.write(outs)

else:                           # Building all other statistics
    p = sys.argv[1]
    pattern = 'P'+p[:-1]

    # Scan files reporting the number of honest/fraudolent blocks in the blockchain and compute the
      average for each cycle
    honest_blocks = []
    fraudolent_blocks = []
    files_scanned = 0
    to_scan = [f for f in glob.glob('docs/statistics/blockchain_R*') if pattern in f ]
    for filename in to_scan:
        with open(filename) as in_file:
            cycle = 0
```

```
            for line in in_file:
                if files_scanned == 0:
                    if line.startswith("#"):
                        honest_blocks.append(None)
                        fraudolent_blocks.append(None)
                    else:
                        a = line.split('            ')
                        honest_blocks.append(int(a[0]))
                        fraudolent_blocks.append(int(a[1]))
                else:
                    if line.startswith("#"):
                        continue
                    else:
                        if cycle != 0:
                            a = line.split('            ')
                            honest_blocks[cycle] = ( (honest_blocks[cycle] * files_scanned) +
                                int(a[0]) ) / (files_scanned + 1)
                            fraudolent_blocks[cycle] = ( ( fraudolent_blocks[cycle] * files_scanned
                                ) + int(a[1]) ) / (files_scanned + 1)
                    cycle+=1
            files_scanned+=1

    with open('docs/statistics/avg/blockchain_P' + p + '_avg.dat', 'w+') as out_file:
                outs= '# Cycle Honest_Blocks  Fraudolent_Blocks \n'
                for count in range(1, len(honest_blocks)):
                    outs = outs + str(count) + '     ' + str(honest_blocks[count]) + '     ' +
                        str(fraudolent_blocks[count]) + '\n'
                out_file.write(outs)


# Scan files reporting the number of forks in the blockchain and compute the average for each cycle
    files_scanned = 0
    forks = []
    to_scan = [f for f in glob.glob('docs/statistics/forks_R*') if pattern in f ]
    for filename in to_scan:
        with open(filename) as in_file:
            cycle = 0
            for line in in_file:
                if files_scanned == 0:
                    if line.startswith("#"):
                        forks.append(None)
                    else:
                        a = line.split('            ')
                        forks.append(int(a[0]))
                else:
                    if line.startswith("#"):
                        continue
                    else:
                        if cycle != 0:
                            a = line.split('            ')
                            forks[cycle] = ( (forks[cycle] * files_scanned) + int(a[0]) ) /
                                (files_scanned + 1)
                    cycle+=1
            files_scanned+=1

    with open('docs/statistics/avg/forks_P' + p + '_avg.dat', 'w+') as out_file:
                outs= '# Cycle Forks \n'
                for count in range(1, len(forks)):
                    outs = outs + str(count) + '     ' + str(forks[count]) + '\n'
                out_file.write(outs)


    # Compute the ratio mined_blocks/hash_rate for honest and selfish miners
    honest_hr = []
    selfish_hr = []
    files_scanned = 0
    to_scan = [f for f in glob.glob('docs/statistics/hashrate_R*') if pattern in f ]
    for filename in to_scan:
        with open(filename) as in_file:
            cycle = 0
            for line in in_file:
                if files_scanned == 0:
                    if line.startswith("#"):
                        honest_hr.append(None)
                        selfish_hr.append(None)
                    else:
                        a = line.split('            ')
                        honest_hr.append(int(a[0]))
                        selfish_hr.append(int(a[1]))
                else:
```

```
                    if line.startswith("#"):
                        continue
                    else:
                        if cycle != 0:
                            a = line.split('            ')
                            honest_hr[cycle] = ( (honest_hr[cycle] * files_scanned) + int(a[0]) ) /
                                (files_scanned + 1)
                            selfish_hr[cycle] = ( ( selfish_hr[cycle] * files_scanned ) + int(a[1])
                                ) / (files_scanned + 1)
                cycle+=1
            files_scanned+=1

    with open('docs/statistics/avg/hashrate_P' + p + '_avg.dat', 'w+') as out_file:
                outs= '# P(SMiner) HonestBlocks/Ghr  SelfishBlocks/Ghr \n'
                honest_ratio = float(honest_blocks[len(honest_blocks)-1]) / ((honest_hr[1] /
                    1000000000 ) + 1) # Add 1 to avoid division by zero
                selfish_ratio = float(fraudolent_blocks[len(fraudolent_blocks)-1]) / ((selfish_hr[1]
                    / 1000000000) + 1 )
                outs = outs + p + '        ' + str(honest_ratio) + '               ' +
                    str(selfish_ratio)
                out_file.write(outs)
```

plot_avg_statistics.py

```
import glob
import os

with open ("docs/plots/template.gnu") as filein:
    plot_template = filein.read()

# Plot graphs of blocks mined by honest miners/selfish miners
blockchain_template = plot_template
blockchain_template += 'set xlabel \'Cycles \' \nset ylabel\'Mined Blocks \' \n'
to_plot = [f for f in glob.glob('docs/statistics/avg/blockchain*')]
for filename in to_plot:
    blockchain_plot = blockchain_template
    p = filename.split('P')[1][:4]
# Sets the name of the output file
    blockchain_plot += 'set output \'docs/plots/blockchain_P' + p + '.png\' \n'
    blockchain_plot += 'plot \'' + filename + '\' u 1:2 t \'Honest blocks\' w lp ls 1, \\\n\'\'
        u 1:3 t \'Fraudolent blocks\' w lp ls 2'
    with open('docs/plots/temp.gnu', 'w+') as file:
            file.write(blockchain_plot)
    os.system ('gnuplot docs/plots/temp.gnu')


# Plot graphs of forks
forks_plot = plot_template
forks_plot += 'set xlabel \'Cycles \' \nset ylabel\'Forks \' \n'
forks_plot += 'set output \'docs/plots/forks.png\' \nplot'
to_plot = [f for f in glob.glob('docs/statistics/avg/forks*')]
count = 0
for filename in to_plot:
    p = filename.split('P')[1][:4]
    count += 1
    forks_plot += ' \'' + filename + '\' u 1:2 t \'P(selfish miner) = ' + p + '\' w lp ls ' +
str(count) + ', \\\n'
with open('docs/plots/temp.gnu', 'w+') as file:
        file.write(forks_plot)
os.system ('gnuplot docs/plots/temp.gnu')


# Plot mined_blocks/ hash_rate graph
# Make one file of the several ones
to_merge = [f for f in glob.glob('docs/statistics/avg/hashrate_P*')]
hr_file = ''
for filename in to_merge:
    with open(filename) as in_file:
        hr_file = hr_file + in_file.read() + '\n'
with open('docs/statistics/avg/hashrate_avg_merged.dat', 'w+') as file:
        file.write(hr_file)
hr_plot = plot_template
hr_plot += 'set xlabel \'P(selfish miner) \' \nset ylabel\'Mined blocks / Hash rate \' \n'
hr_plot += 'set output \'docs/plots/blocks_per_hashrate.png\' \n'
hr_plot += 'plot \'docs/statistics/avg/hashrate_avg_merged.dat\' u 1:2 smooth unique t \'Honest
   miners\' w lp ls 1, \\\n\'\'                  u 1:3 smooth unique t \'Selfish miners\' w lp ls 2'
with open('docs/plots/temp.gnu', 'w+') as file:
        file.write(hr_plot)
os.system ('gnuplot docs/plots/temp.gnu')
```

```
# Plot reward / hash_rate graph
# Make one file of the several ones
to_merge = [f for f in glob.glob('docs/statistics/avg/reward_P*')]
rew_file = ''
for filename in to_merge:
    with open(filename) as in_file:
        rew_file = rew_file + in_file.read() + '\n'
with open('docs/statistics/avg/reward_avg_merged.dat', 'w+') as file:
        file.write(rew_file)
rew_plot = plot_template
rew_plot += 'set xlabel \'P(selfish miner) \' \nset ylabel\'Reward / Hash rate \' \n'
rew_plot += 'set output \'docs/plots/reward_per_hashrate.png\' \n'
rew_plot += 'plot \'docs/statistics/avg/reward_avg_merged.dat\' u 1:2 smooth unique t \'Honest
miners\' w lp ls 1, \\\n\'\'                  u 1:3 smooth unique t \'Selfish miners\' w lp ls 2'
with open('docs/plots/temp.gnu', 'w+') as file:
        file.write(rew_plot)
os.system ('gnuplot docs/plots/temp.gnu')


# Plot graphs of mined blocks for different values of message delay
delay_plot = plot_template
delay_plot += 'set xlabel \'Cycles \' \nset ylabel\'Mined Blocks \' \n'
delay_plot += 'set output \'docs/plots/blocks_with_delay.png\' \nplot'
to_plot = [f for f in glob.glob('docs/statistics/avg/latency*')]
count = 0
for filename in to_plot:
    p = filename.split('D')[1][:1]
    count += 1
    delay_plot += ' \'' + filename + '\' u 1:2 t \'Delay = 0.' + p + '0\' w lp ls ' + str(count) +
', \\\n'
with open('docs/plots/temp.gnu', 'w+') as file:
        file.write(delay_plot)
os.system ('gnuplot docs/plots/temp.gnu')
```