Sistema de leilões com registos públicos

Pedro Fernando Moreira Silva Antunes
Departamento de Ciências de Computadores
Faculdade de Ciências da Universidade do Porto
Porto, Portugal
up201507254@edu.fc.up.pt

I. Introdução

Este trabalho está inserido na unidade curricular de Segurança de Sistemas e Dados do Mestrado de Segurança Informática da Faculdade de Ciências da Universidade do Porto. Consiste na construção de um sistema de leilões descentralizado dando ênfase à segurança dos dados do sistema.

Os utilizadores podem iniciar um leilão sobre um produto que queiram vender definindo um valor inicial. Os compradores podem entrar no leilão de um produto fazendo uma oferta superior ao valor momentâneo do produto. Quando um produto teve uma oferta superior ao valor inicial do leilão, e essa oferta consegue ser a melhor oferta durante um dia, o leilão desse produto acaba e definimos que o autor do leilão conseguiu vender o produto com o preço definido da última oferta ao comprador que foi o autor da última oferta. Quando isto acontece, criamos uma transação entre o comprador e o vendedor.

Os registos das transações do sistema são públicos. Para isto foi desenvolvido uma blockchain das transações do sistema. A blockchain foi desenvolvida sobre uma rede ponto-a-ponto de modo que nos garanta descentralização da informação, ou seja que todos os pontos de rede tenham os registos das transações do nosso sistema. A rede ponto-a-ponto também é responsável pela construção registos. Há pontos na rede chamados de ”miners” apenas para calcular as hashes de cada bloco da blockchain. Só depois de os miners calcularem as hashes de um novo bloco é que os registos das transações entram para a blockchain. A partir do momento em que entram na blockchain, a informação das transações fica pública e descentralizada.

Ao longo deste relatório vamos introduzir breves conceitos teóricos do que foi utilizado no nosso sistema, seguindo o design do sistema e os resultados obtidos.

II. Descrição do sistema e dos conceitos utilizados

Como foi introduzido, o sistema de leilões não tem um ponto fixo chamado de servidor que fornece os serviços necessários para o funcionamento do sistema. Por outro lado, utilizando uma rede ponto-a-ponto, todos os clientes podem servir de servidor para fornecer os serviços a outros clientes.

O sistema de leilões utiliza o padrão de publish-subscriber onde cada criador de leilão é chamado de publisher e categoriza o leilão por um ou mais tipos. Os utilizadores que querem entrar em leilões são chamados de subscribers. Similar ao processo dos publishers, os subscribers também categorizam os tipos de leilões que têm interesse. Desta forma, se as categorizações coincidirem existe a possibilidade da realização de ofertas para os leilões de um certo produto.

A informação dos produtos, leilões e categorizações são espalhadas pelos nós da rede. Assim um nó pode sair da rede à vontade e quando quiser pertencer à rede, os nós que permaneceram na rede partilham as informações que têm guardadas do sistema.

A. *Kademlia:*

A rede ponto-a-ponto foi estruturada para implementar a rede descentralizada Kademlia. O kademlia é uma tabela de hash distribuída que cria uma rede sobreposta formada pelos nós pertencentes ao sistema de leilões. Cada nó (para além de dados como ips, certificados, portas e dados do utilizador) tem associado um identificador ID que identifica esse nó na rede sobreposta. Este ID é depois utilizado para localizar os nós na rede.

Quando queremos pesquisar um nó na rede exploramos a rede em várias etapas. O kademlia calcula as distâncias a que estamos de cada nó que pertence à nossa rede através do cálculo da métrica XOR sobre os IDs de cada nó na rede. Cada etapa localizará os nós que estão mais próximos sendo estes os que tiverem as distâncias iguais 1,2,4 e 8. Os nós que estão a estas distãncias são os nós que estamos virtualmente ligados.

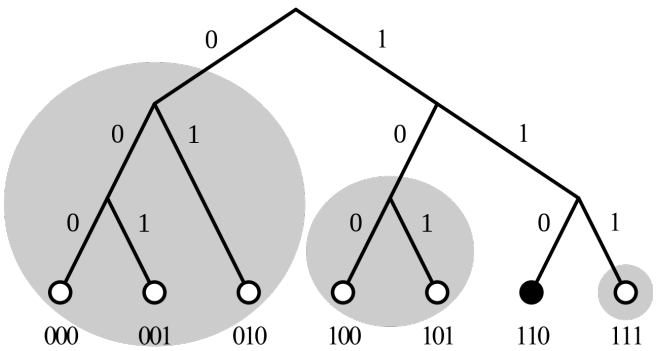

Fig. 1. Partição da rede para o nó 110

Com a escolha destas distâncias estamos a criar partições na rede de modo que cada nó tem referências para os nós a essas distâncias. Assim estamos a definir que cada nó que estamos em contacto, é a raiz de uma sub-árvore de pesquisa

da nossa rede, sobre os IDs kademlia. Posto isto, conseguimos descobrir um nó na rede em log(n) passos quando a rede tem n nós.

Posto isto, depois de encontrarmos o nó que estavamos à procura a comunicação com ele é feita através de sockets. A comunicação através dos sockets tem de ter meios criptográficos, como implementação de certificados para a utilização do SSL, que nos garantam confidencialidade, integridade, disponibilidade e não-repúdio nas comunicações.

A rede kademlia é construída pelo nó inicial (bootstrap node) e todos os novos nó que entrem para a rede têm de ter mecanismos para comunicar com o nó inicial. O nó inicial vai fornecer um ID para identificar o novo nó na rede kademlia. O processo para que o novo nó fique a funcionar na rede é contactar o nó inicial para que ele faça uma pesquisa na árvore do nó que o está a contactar. Assim e através das distâncias, o nó inicial irá obter as tabelas de encaminhamento do novo nó. Assim que tiver a tabela de encaminhamento, que contém os nós mais próximos do novo nó, passa essa informação para o novo nó e atualiza as tabelas de encaminhamento de todos os nós.

A rede também tem nós fixos apenas para o processo de ”mining” dos blocos da blockchain. Assim, garantimos que a rede tem sempre nós com as informações guardadas do nosso sistema e que temos o suporte da rede descentralizada para auxiliar a construção da blockchain.

*B. **BlockChain:***

Uma blockchain é simplesmente uma lista de blocos. Cada bloco terá a sua própria identidade (hash), contém e é construído através da identidade(hash) do bloco anterior e também terá os dados referentes às transações do sistema de leilões. Na prática cada bloco não contém apenas as hashes da bloco anterior, mas sim a sua própria hash é calculada através da hash do bloco anterior.

Caso os dados do bloco anterior mudarem a sua hash será diferente. Isto irá afetar todas as hashes dos blocos que virão a seguir. Calculando assim e comparando as hashes dos blocos, permite-nos que possamos verificar se a blockchain é inválida. Ou seja, modificar qualquer dado na lista de blocos, irá mudar a identidade e irá quebrar toda a cadeia de blocos.

É através desta metodologia que vamos guardar a informação referente às transações provenientes dos leilões. Os blocos serão construídos através da data em que aconteceu a transação, quem comprou, quem vendeu e por quanto. Cada bloco é constituido por 4 transações. Sempre que tivermos mais transações que estas, calculamos a hash do bloco, colocamos o bloco na blockchain e criamos um novo bloco para ser preenchido.

O processo de minerar as hashes de cada bloco chama-se de ”proof-of-work” onde tentamos diferentes valores inteiros para que este valor entre na identidade do bloco e a hash de cada bloco só é valida caso comece com um certo números de 0’s no prefixo. Normalmente, é este o desafio lançado para que um bloco seja validado e a dificuldade do desafio é influenciado pela tamanho do prefixo de 0’s. Quantos mais zeros temos como prefixo de cada hash, mais tempo demora a arranjar um número para que o resultado da hash passe o desafio.

Finalmente, a hash de cada bloco é influenciada pela hash do bloco anterior, mais as hashes os dados referentes a cada transação inserida neste bloco, a data da criação do bloco e o número aleatório que permitiu que o resultado final da hash passasse o desafio de começar com um certo número de 0’s.

Apesar de as transações serem um tema sensível quando falamos de segurança, construíndo uma blockchain e ter as funcionalidade prontas para minerar os blocos, estamos a garantir a seguranças dos dados de forma que um atacante só poderá falsificar os dados da blockchain caso tenha um poder computacional superior à combinação de todos os nós que pertencem à nossa rede. Com esse poder computacional, o atacante terá de ser capaz de criar uma nova e inteira blockchain válida no tempo que os nós da nossa rede estão a calcular a identidade do último bloco.

Em termos de eficiência, e para não termos todas as hashes de cada transação colocadas num bloco, desenvolveu-se uma estrutura de dados Merkle Tree que nos permite efetuar verificações eficientes em que os dados são reduzidos e que pertencem a um enorme conjunto de dados. Ou seja, reduziu-se todas as hashes das transações referentes a um bloco para apenas um valor de hash. Conseguiu-se isto através da combinação de duas hashes, calcula-se o valor desta combinação e vamos recursivamente fazendo este processo até que por fim tenhamos apenas um valor de hash que será o valor da raiz da nossa estrutura Merkle Tree.

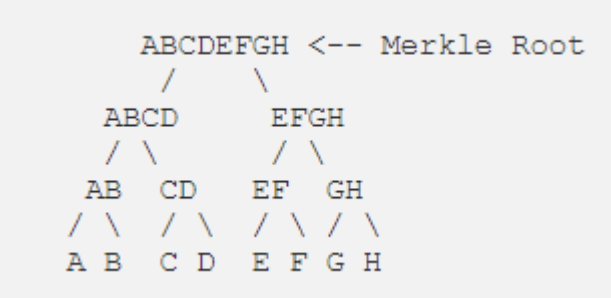

Fig. 2. Exemplo abstrato da construção da raiz da Merkle Tree

III. Análise de Segurança e Resultados

Assumiu-se que a parte de publish/subscriber que controla os pedidos de leilões já estava implementada. Portanto, assumimos que utilizaríamos essa parte através de outra implementação que não é nossa. Assumimos que seja a implementação que usarmos proveniente de terceiros, tem que nos garantir autenticação, identificação, autorização e controlo de acessos sobre os utilizadores do serviço. Se não tivermos estas garantias não conseguimos garantir nenhuma propriedade de segurança para a frente.

A rede ponto-a-ponto também não está funcional, mas tentamos implementar alguns dos métodos do Kademlia. Para isto utilizou-se a framework gRPC que foi desenvolvida pela Google que nos garante eficientes chamadas remotas através

de sockets. As mensagens Kademlia a serem implementadas para esta comunicação são:

- ***PING:***
  Utilizado para verficar se um nó ainda está ativo na rede.
- ***STORE:***
  Armazena informação em um nó através do par (chave, valor) onde a chave é o identificar do nó.
- ***FIND_NODE:***
  O destinatário da solicitação retornará os K nós da tabela de encaminhamento que são os mais próximos do identificardor do nó solicitado.
- ***FIND_VALUE:***
  O mesmo princípio da função FIND_NODE mas neste caso, o destinatário da solicitação retorna o valor armazenado correspondente ao identificar solicitado.

Destas funções implementamos a FIND_NODE. Fazendo para isto, uma procura recursiva do nó nas tabelas de encaminhamento que íamos obtendo dos nós que tocavamos. Caso não encontrássemos o valor numa primeira fase, víamos qual era o nó mais próximo (de todos os que já tínhamos visualizado) do nó que queremos encontrar.

Os restantes métodos não serão muito diferentes de implementar do método FIND_NODE.

Com a implementação do Kademlia estamos vulneráveis a alguns ataques que podem colocar o nosso sistema em perigo. Faltou implementar a resistência a Sybil e Eclipse attacks. Para a nossa rede ser resistente a estes ataques teríamos de implemntar a nossa rede kademlia de forma a que nos criasse uma rede sobreposta em que tinhamos a certeza de que confiávamos em mais de 50% dos nós que pertencem à rede. Só assim conseguiríamos ter a maioria dos nós na rede resistentes a qualquer tipo de corrupção quando fosse necessário tomar medidas para o funcionamento da rede e do sistema.

Quando tivermos implementados todos os pontos referidos a cima, podemos admitir que o nosso sistema é seguro e que temos uma rede descentralizada que suporte o nosso método de blockchain de guardar-mos os dados referentes às transações dos utilizadores do nosso sistema.

A parte da blockchain foi implementada com sucesso. A forma de estabelecermos consenso de verificação dos dados introduzidos na blockchain e de resistirmos a casos como o gasto duplo (quando um utilizador consegue gastar as mesmas moedas mais que uma vez) foi através do proof-of-work (referido em cima). No entanto, este não é o melhor método para este efeito. Isto porque consome muita energia no cálculo das hashes dos blocos e isto não é bom nem sustentável para o Planeta.

Existem outras formas de estabelecermos consenso, tais como o proof-of-stake, e será um dos trabalhos a realizar no futuro. O proof-of-stake o ”indivíduo” que cria o próximo bloco é escolhido em função de quantas ”stacks” é que produziu. Estas ”stacks” são baseadas no número de moedas que a pessoa tem relacionadas com a blockchain que estão a tentar minerar. Quantas mais moedas, mais probabilidade tem de ser escolhido para verificar o bloco, calcular a hash (sem o prefixo do proof-of-work) e inserir o bloco na blocochain.

Utilizamos o algoritmo sha256 para produzirmos as hashes necessárias para a blockchain.

```java
package blockchain;

import java.security.MessageDigest;

public class Hash_helper {
    public static String sha256(String input) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-256");
            byte[] hash = digest.digest(input.getBytes( charsetName: "UTF-8"));
            StringBuffer hexString = new StringBuffer();
            for (int i = 0; i < hash.length; i++){
                String hex = Integer.toHexString( i: 0xff & hash[i]); // '0xff & hash[i] set bits result and then converted to decimal'
                if (hex.length() == 1)  // add leading zero if needed (all hex conversion sets with 2 chars)
                    hexString.append('0');
                hexString.append(hex);
            }

            return hexString.toString();
        }
        catch(Exception e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }
    }
}
```

Fig. 3. Método de Hash de uma string.

Foi feito um objeto Transaction para lidar com os dados referentes às transações e calcular a hash de cada transação.

```java
public class Transaction extends Object{
    private String buyer ;
    private String seller ;
    private int amount ;
    private String hash;
    private long timestamp;

    public Transaction(String buyer, int amount, String seller) {
        this.buyer = buyer ;
        this.seller = seller ;
        this.amount = amount ;
        this.timestamp = new Date().getTime();;


        String digest = timestamp + ":" + buyer + "," + String.valueOf(amount) + "," + seller;

        this.hash = Hash_helper.sha256(digest);
    }
    //
    // GETTERS
    //
or
    public String getHash() { return hash; }

    public String getBuyer() { return buyer; }

    public String getSeller() { return seller; }

    public int getAmount() { return amount; }


}
```

Fig. 4. Classe Transaction.

Criou-se uma classe Block que constrói os blocos a colocar na blockchain.

```java
public class Block  {
    private final int NUMBER_OF_NONCE_BYTES = 4;

    public String hash ;
    public String prev_hash ;
    private long timestamp ;
    private List<Transaction> transactions;
    private double nonce;
    private String root_merkle_hash;


    public Block(String prevHash) {
        this.prev_hash = prevHash ;
        this.timestamp = new Date().getTime();
        this.hash = calculateHash();

        this.transactions = new ArrayList<>();
        this.nonce = 0;
        this.root_merkle_hash= "";
    }

    private void setRoot_merkle_hash() {
        List<String> tree = merkleTree();
        this.root_merkle_hash = tree.get(tree.size()-1);
    }
```

Fig. 5. Construtor e variáveis da classe Block.

Apenas colocamos a hash da raiz da estrutura de dados Merkle Tree no cálculo da hash do bloco.

```java
private List<String> merkleTree(){
    List<String> tree = new ArrayList<>();

    // adicionar hashes das transacoes à árvore
    for ( Transaction t : transactions)
        tree.add(t.getHash());

    int levelOffSet = 0; // contador de nos visitados. offset para saber onde começa cada nivel.

    for ( int size_level = transactions.size(); size_level > 1; size_level = (size_level+1) / 2 ) { // nmr de nos por nivel. prox nivel tem o nmr de nos
        for ( int left = 0 ; left < size_level ; left +=2 ) { //percorre todos os nos da esquerda de cada nivel
            int right = Math.min( left+1 , size_level-1 );   // right=min(left+1, size_level-1) (caso o nmr de transactions seja impar, right=left)
            String left_hash = tree.get(levelOffSet + left);
            String right_hash = tree.get(levelOffSet + right);
            tree.add( Hash_helper.sha256(  input: left_hash + right_hash )); //add num novo nivel H(left_hash+right_hash)
        }
        levelOffSet += size_level; // nos lidos neste nivel. para que o proximo nivel saber que começa a partir deste valor.
    }

    return tree;
}
```

Fig. 6. Método que constrói a merkle Tree.

```java
// calculate hash of this block
public String calculateHash() {

    // hash do cabecalho do bloco = prev_hash + nonce + timestamp + root_merkle_hash
    String hash_block = Hash_helper.sha256(
            input: prev_hash +
            timestamp +
            nonce +
            root_merkle_hash
re  );

    return hash_block;
}
```

Fig. 7. Método que calcula a hash do bloco.

```java
// nonce challenge to calculate hash of this block
public void mineBlock(int difficulty) throws NoSuchAlgorithmException {

    //Create a string with difficulty * "0"
    String hash_prefix = new String(new char[difficulty]).replace( oldChar: '\0', newChar: '0');

    while ( !hash.substring(0,difficulty).equals(hash_prefix)) {
        // strong RNG generates random numbers with NO_OF_NONCE_BYTES bytes length
        SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
        random.setSeed(System.currentTimeMillis() * 1000);
        byte[] ranBytes = new byte[NUMBER_OF_NONCE_BYTES];
        random.nextBytes(ranBytes);

        for(int i=0; i<ranBytes.length; i++) {
            int number = 0xff & ranBytes[i];
            nonce += (double) number;
            // o ciclo para caso receba a hash calculdada de outro miner. depois verifica se a hash esta correta
        }

        hash = calculateHash();
    }
    System.out.println("Block Mined!! : " + hash);
}
```

Fig. 8. Consenso proof-of-work que calcula o nonce para a hash do bloco ter um prefixo com vários 0's.

Podemos verificar as transactions através da função isTransactionValid() que constrói novamente a estrutura merkleTree e compara a raiz guardada com a raiz da nova árvore. Se forem diferentes é porque houve falsificação nos dados.

```java
public boolean isTransactionsValid() {
    List<String> tree = merkleTree();

    int size = tree.size();
    if(size == 0)
        return true;    // genesis_block case
    else {
        String root = tree.get( size - 1 );
        return root.equals(this.getRootMerkleHash());
    }
}

public boolean isFull() {
    //if ( transactions == null )
        //return false; // genesis_block case

    if ( transactions.size() < 5 )
        return false;

    return true;
}

// true if t is added successfully ;
// false tells that this block is ready to be mined
public boolean addTransaction(Transaction t) {
    if ( !isFull() ) {
        transactions.add(t);
        setRoot_merkle_hash();
        return true;
    }
    return false;
}
```

Fig. 9. Funções relativas às transações de cada bloco.

A nossa blockchain é simplesmente uma lista de objetos Block. Para verificarmos se a nossa lista de blocos está válida, chamamos a função isChainValid() sempre que inserimos um novo bloco na lista de blocos.

```
public static boolean isChainValid(){
    Block curr_block;
    Block prev_block;

    //Create a string with difficulty * "0"
    String hash_prefix = new String(new char[difficulty]).replace( oldChar: '\0', newChar: '0');
    //check hashes of each block of blockchain
    for ( int i=1; i < blockchain.size(); i++ ) {
        curr_block = blockchain.get(i);
        prev_block = blockchain.get(i-1);

        //compare previous hash and registered previous hash
        if ( !prev_block.hash.equals(curr_block.prev_hash) ) {
            System.out.println("Previous Hashes not equal");
            return false;
        }
        // check if hash has been mined
        if ( !curr_block.hash.substring(0, difficulty).equals(hash_prefix) ) {
            System.out.println("This block hasn't been mined");

            //calculate hash again and compare with registered hash
            if ( !curr_block.hash.equals(curr_block.calculateHash()) ){
                System.out.println("Current Hashes not equal");
                return false;
            }
            return false;
        }
        // check if transaction have integrity
        if ( !curr_block.isTransactionsValid() )
            return false;
    }
    return true;
}
```

Fig. 10. Validação da blockchain.

Como teste instância-mos algumas transações da classe Transaction e tentamos simular a construção da blockchain. O procedimento para o funcionamento é verificar as transações do bloco criado e quando o bloco estiver cheio passa para a fase de minerar o bloco. Após isto verificamos se a nossa blockchain ainda se mantém válida e continuamos com o procedimento.

```
for ( Transaction t : transactions_list ) {
    //Block current_block = blockchain.get(curr);

    boolean success_transaction_block = current_block.addTransaction(t);

    // check if block is ready to mine
    if ( current_block.isFull() || !success_transaction_block ) {
        if ( current_block.isTransactionsValid() ) {    // can i trust in this block transactions?

            // mine current block
            System.out.println("Trying to Mine block " + (curr + 1) + "... ");
            current_block.mineBlock(difficulty);
            blockchain.add(current_block);

            if (!BlockChain.isChainValid()) {
                System.out.println("\n\n\nFRAUD DETECTED ON BLOCKCHAIN");
                break;
            }

            // create new block
            Block new_block = new Block(current_block.hash);
            curr++;
            current_block = new_block;
        }
    }
}
```

Fig. 11. Ciclo para transformar as trasações em blocos.

Posto isto, obtemos o resultado final com a criação da nossa blockchain

```
Trying to Mine block 1...
Block Mined!! : 0000d24728a7025e6b0730a763692cc40254a82fa4eb0e47c24bb32d61fcc9f4
Trying to Mine block 2...
Block Mined!! : 0000420234c3399d660b2fb8e09d2e35ff5876ddeaf5bcde05709ec5b5e42e24

The block chain:
[
  {
    "NUMBER_OF_NONCE_BYTES": 4,
    "hash": "0000d24728a7025e6b0730a763692cc40254a82fa4eb0e47c24bb32d61fcc9f4",
    "prev_hash": "0",
    "timestamp": 1623186969030,
oftware    "once": 1.43784049E8,
    "root_merkle_hash": "fbbc94026d415531482e9a759520068943281a368fb605d6312f198b1b23b77e",
    "transactions": [
      {
        "buyer": "Bob",
        "seller": "Pedro",
        "amount": 1,
        "hash": "91ae71c0b499243cff3c20fc520e44f53d29ad21e628a404aab796e8c507559c",
        "timestamp": 1623186968947
      },
      {
        "buyer": "Bob",
        "seller": "Alice",
        "amount": 1,
        "hash": "79015a35840b4f3dd027374e615436f3e1b8874622e19f31449116ba32ff27cc",
        "timestamp": 1623186969026
```

Fig. 12. Testagem da blockchain.

IV. Conclusões e trabalhos futuros

Começamos este artigo por identificar o sistema e os requisitos de segurança. Vimos de forma geral os conceitos aplicados tecnológicos aplicados no trabalho para garantirmos a segurança do sistema e dos dados do sistema. Introduzimos conceitos da rede Kademlia mas a implementação desta tecnologia não ficou completa. Demos mais ênfase à parte da segurança dos dados, estudando e implementando a fundo uma estrutura de blockchain que nos garante que os dados não podem sofrer fraudes que não damos conta delas.

Como existem vulnerabilidades a cada dia, temos de estar constantemente a atualizar o nosso conhecimento de ameaças e vulnerabilidades encontradas pela comunidade de segurança para que possamos detetar qualquer falha que possa colocar o nosso sistema em risco.

Como trabalhos futuros, tenciona-se acabar a implementação da rede kademlia e garantir que a rede é resistente a ataques como os Sybil ou Eclipse. Tenciona-se também realizar a implementação da parte da interação dos utilizadores com os leiloes utilizando o padrão publisher/subscriber. Depois de acabarmos os trabalhos futuros podemos dizer que temos um sistema de leilões com os registos das transações públicos.

References

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[2] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper, 151:1–32, 2014.

[3] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems (TOCS), 20(4):398–461, 2002.

[4] João Sousa, Eduardo Alchieri, and Alysson Bessani. State machine replication for the masses with bft-smart. 201315

[5] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In International Workshop on Peer-to-Peer Systems, pages 53–65. Springer, 2002.

[6] Ingmar Baumgart and Sebastian Mies. S/kademlia: A practicable approach towards secure keybased routing. In Parallel and Distributed Systems, 2007 International Conference on, pages 1–8. IEEE, 2007.

[7] Francis N. Nwebonyi, Rolando Martins, and Manuel E. Correia. Reputation based approach for improved fairness and robustness in p2p protocols. Peer-to-Peer Networking and Applications, Dec 2018.