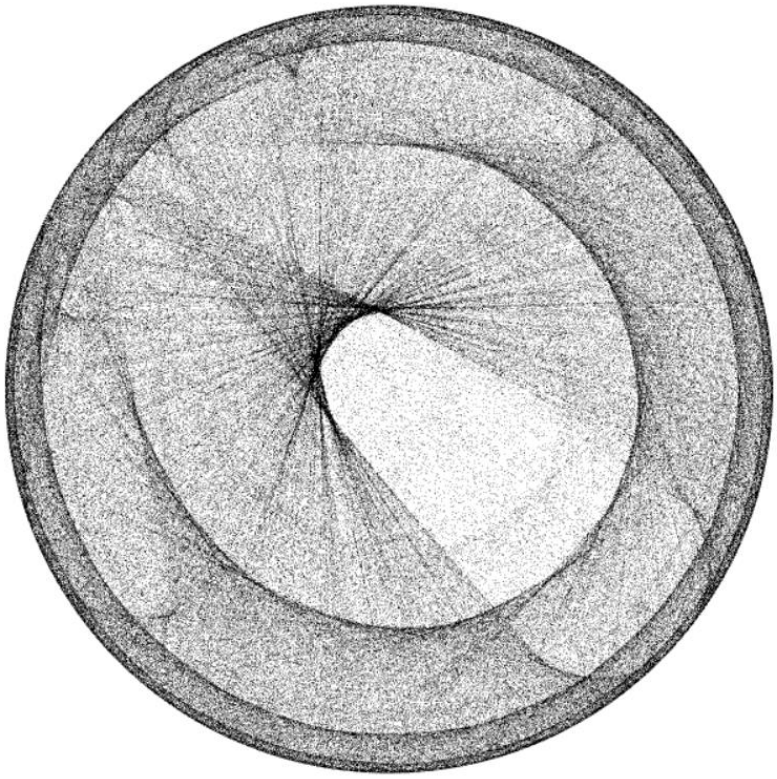


Peer to Peer Systems and Blockchains

A.A. 2017/2018

Report Mid-Term "Analyzing the Chord DHT"

Studente: *Antonio Sisbarra*

Matricola: *518552*

Indice

1. Introduzione

Nello studio delle proprietà della *Distributed HashTable* di Chord può essere di particolare importanza eseguire una simulazione del routing, tra un numero fissato di peer, utilizzando le regole dell'algoritmo.
In quest' elaborato nelle prime due sezioni vengono analizzati, in modo statistico e comparativo rispetto ai risultati standard di Chord, i dati raccolti sulla topologia e il lookup delle simulazioni effettuate.
L'eseguibile, il cui codice sorgente è riportato nell'ultima sezione, prende in input due parametri, m e n, ossia il numero di bit da usare per gli identificatori e il numero di nodi nella rete da simulare.
Variando questi parametri, si è arrivati ad alcuni risultati interessanti.

1.1 Scelte progettuali

Dopo diverse simulazioni si è deciso di utilizzare **al massimo 16 bit per gli id** (valore m dell'input) per la maggior parte delle analisi per via dei tempi di attesa, e di studiare diversi parametri statistici al variare di m e n; l'eseguibile può lavorare anche con numeri più alti, ma i tempi di esecuzione crescono a causa delle operazioni tra BigInteger.
Per quanto riguarda la **gestione delle collisioni** nello spazio degli identificatori si è adottata la seguente politica:

1. Calcolo il modulo sullo SHA1 di un IP generato a random
2. Se l'identificatore è già presente nella rete -> vai al punto 1 (incrementando il contatore delle collisioni)

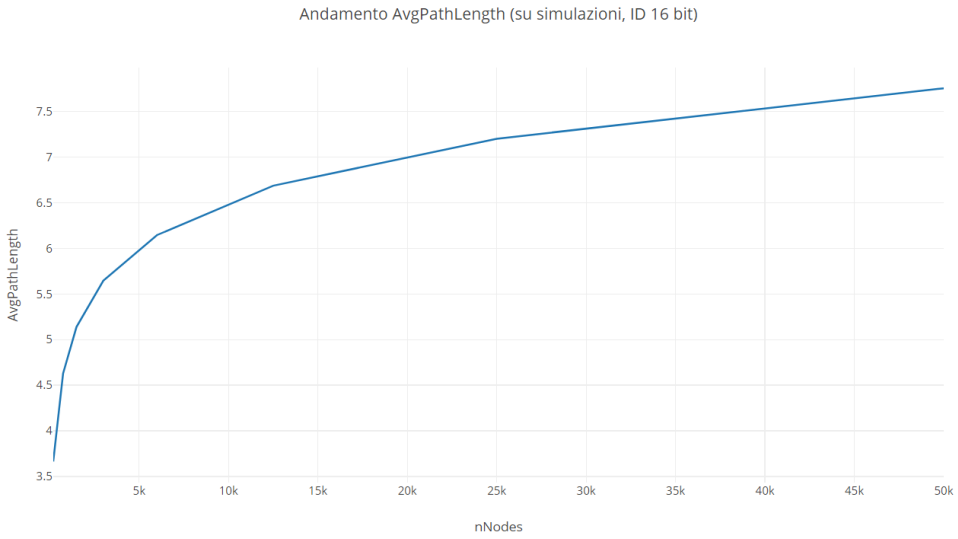
Per effettuare le **analisi sul routing** si è deciso di simulare **un milione di query**, scegliendo ogni volta un nodo a random tra i peer della rete, da cui far partire il lookup di una chiave. In questa maniera è stato possibile analizzare in modo più accurato la distribuzione di alcune variabili, ad es. *EndPoint Distribution* e *Query Distribution*.

2. Analisi statistiche su routing ChordSim

Come già anticipato nell'introduzione tutte le statistiche che riguardano il routing sono state effettuate su una base di un milione di routing op.

2.1 Hops per lookup

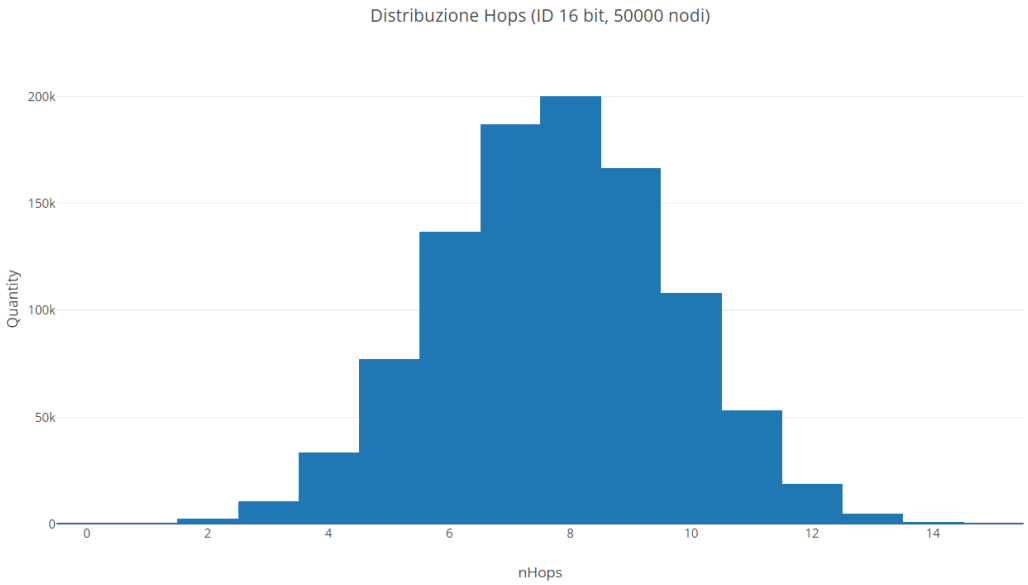
Il primo studio riguarda l'andamento del numero di Hop medio nei lookup, al variare del numero dei nodi nella rete.



La funzione risultante è un logaritmo del numero dei nodi, quindi in linea con la teoria vista a lezione, nello specifico:

$$\text{AvgPathLength(nNodes)} = \Theta(\frac{1}{2}(\log(\text{nNodes})))$$

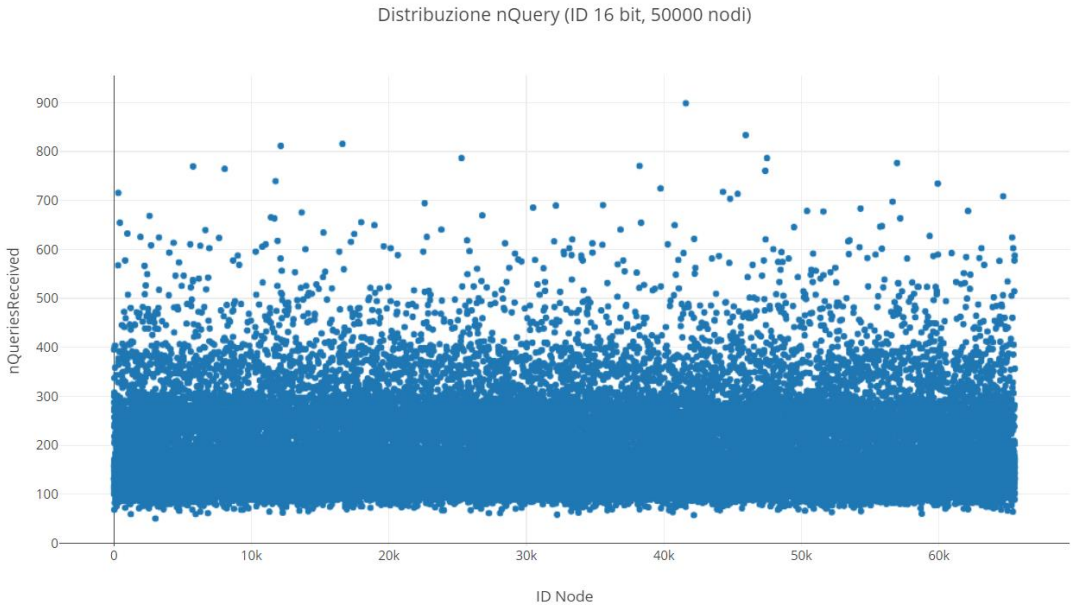
Il secondo grafico riguarda invece la distribuzione del valore del PathLength in una rete di 50.000 nodi.



La distribuzione richiama fortemente una Gaussiana, centrata sul valore medio calcolato in precedenza. Anche questo risultato è in linea con i risultati di Chord nella realtà.

2.2 Numero di Query per nodo

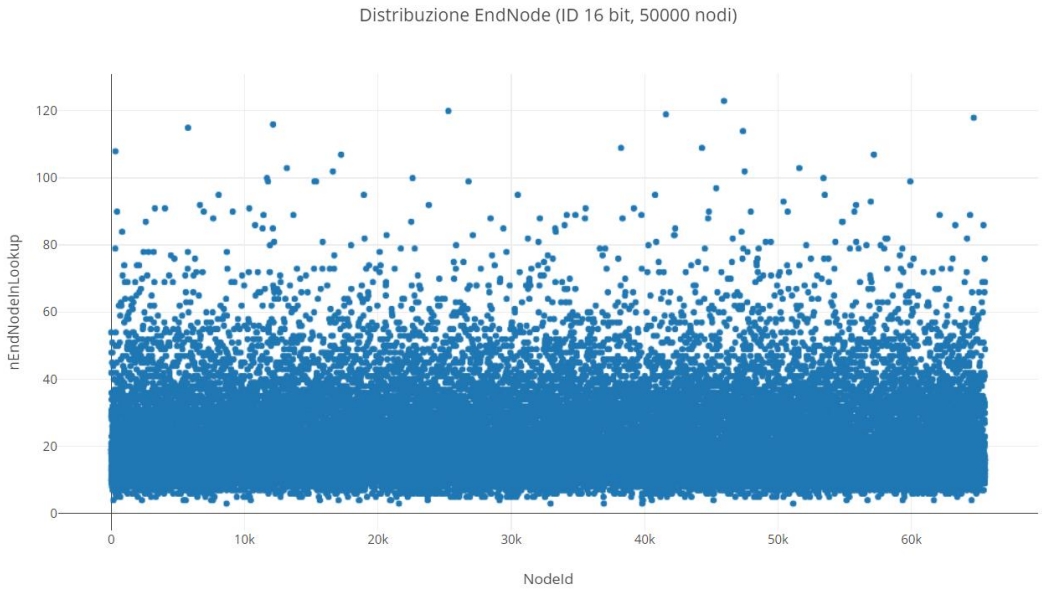
Ai fini dello studio dell'uniformità della distribuzione degli identificatori può essere utile analizzare il numero di Query (intese come richieste di una chiave provenienti da un nodo) ricevute.



Ogni punto nel grafico proietta un nodo e il numero di query ricevute. Si evince una discreta uniformità, con una forte maggioranza di nodi che hanno un numero di query comprese nell'intervallo [100, 400], mentre fino a 900 troviamo una minoranza di nodi.

2.3 EndNode Distribution

Oltre al numero di query ricevute, può aiutare nello studio dell'uniformità degli identificatori il numero di volte in cui un nodo risulta nodo finale in un lookup. Infatti, da una distribuzione non uniforme degli identificatori nello spazio, ne consegue una forte diversità nel valore nTimesEndNode.



Anche in questo caso risulta una discreta uniformità nel valore analizzato. La maggior parte dei nodi è EndNode tra le 5 e le 58 volte, con picchi che possono arrivare a 120.

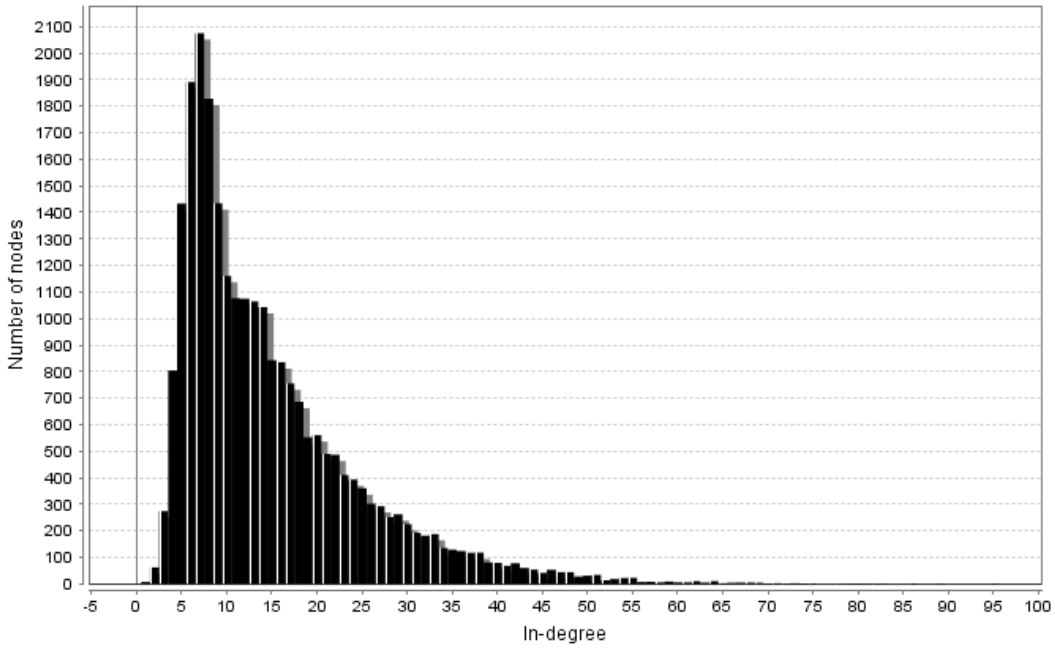
3. Analisi statistiche su topologia ChordSim

Date le limitazioni di memoria imposte dalla macchina su cui si è eseguita la simulazione, si è deciso di analizzare, dopo varie prove, la topologia di ChordSim su due reti con 16 bit per gli identificatori. Il primo studio riguarda una rete con uno spazio degli identificatori ancora libero per metà, utilizzando 25.000 nodi; il secondo studio invece con 50.000 nodi; il terzo con confronti statistici tra reti di dimensioni diverse.

3.1 Rete con 25.000 nodi

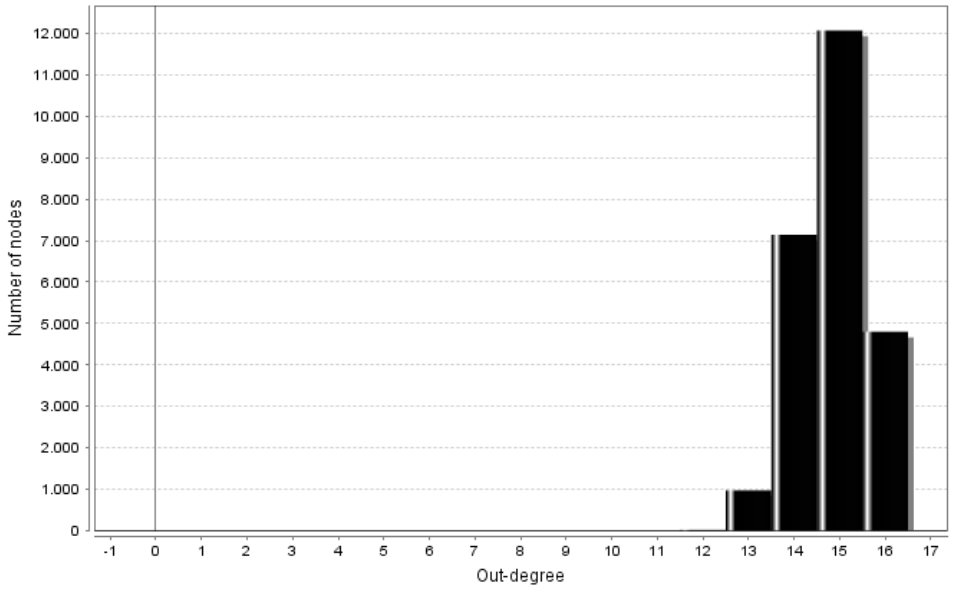
Segue una serie di analisi sulla simulazione di Chord su una rete con circa metà dello spazio degli identificatori ancora libero.

In-Degree Distribution



L'In-Degree misura il numero di archi entranti in ciascun nodo, e può essere descritto da una curva esponenziale (decrescente). Questo introduce un problema di load balancing, dato che i nodi con un In-Degree alto gestiranno tanti messaggi. La distribuzione osservata è in linea con quanto visto a lezione.

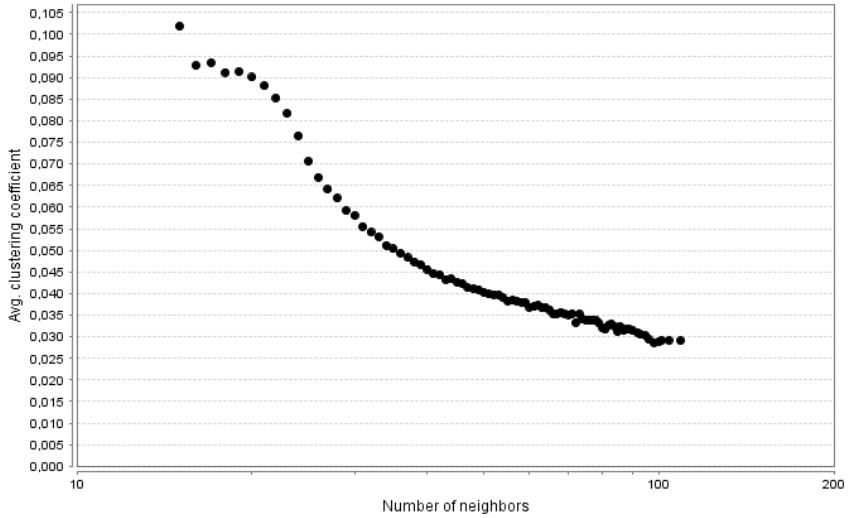
Out-Degree Distribution



Dall'analisi dell'Out-Degree (numero di archi uscenti da un nodo) viene fuori una simmetria rispetto al numero di bit utilizzati per la simulazione, in linea con quanto visto per il routing reale di Chord.

AvgClustering Coefficient in base a numero di vicini

Studiare la distribuzione del Clustering Coefficient è importante per capire se la rete che stiamo analizzando è organizzata, secondo delle regole, oppure se si è in presenza di una Random Network.

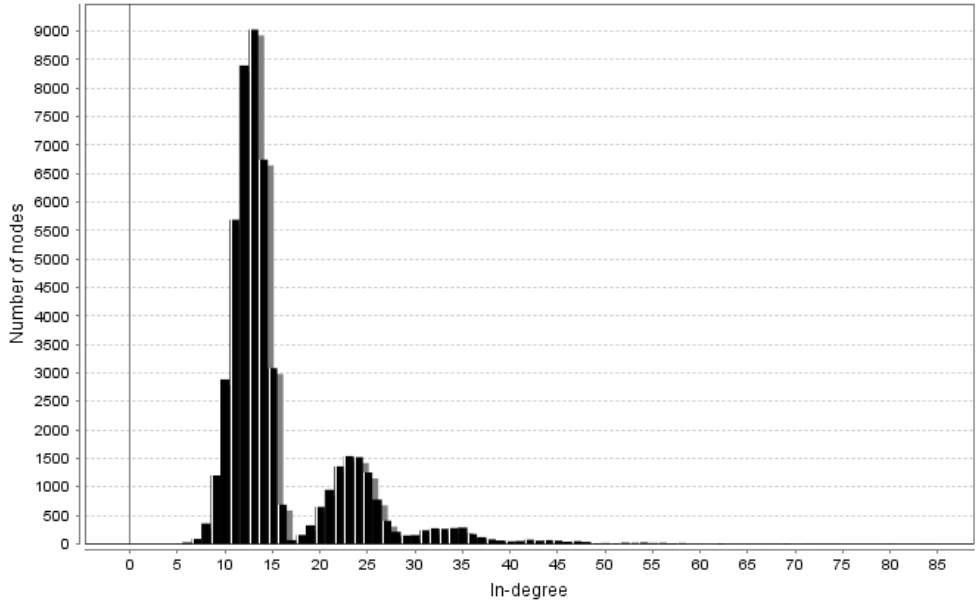


Dal grafico si evince una curva decrescente, in proporzione al numero di vicini, e si può notare che in generale il coefficiente non è basso (se messo in relazione con quello di una rete random).

3.2 Rete con 50.000 nodi

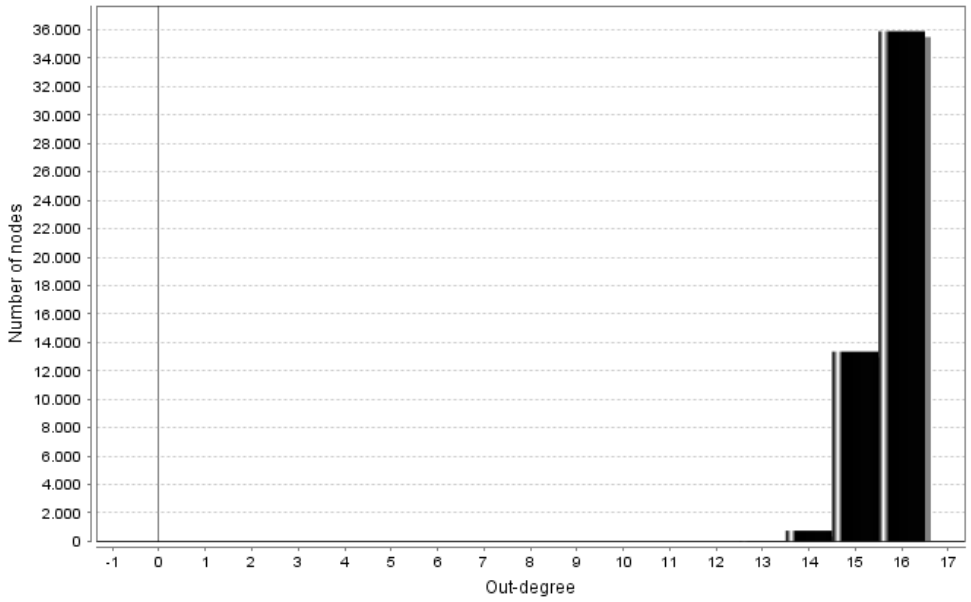
Segue una serie di analisi su una simulazione di rete Chord con spazio degli identificatori quasi pieno.

In-Degree Distribution



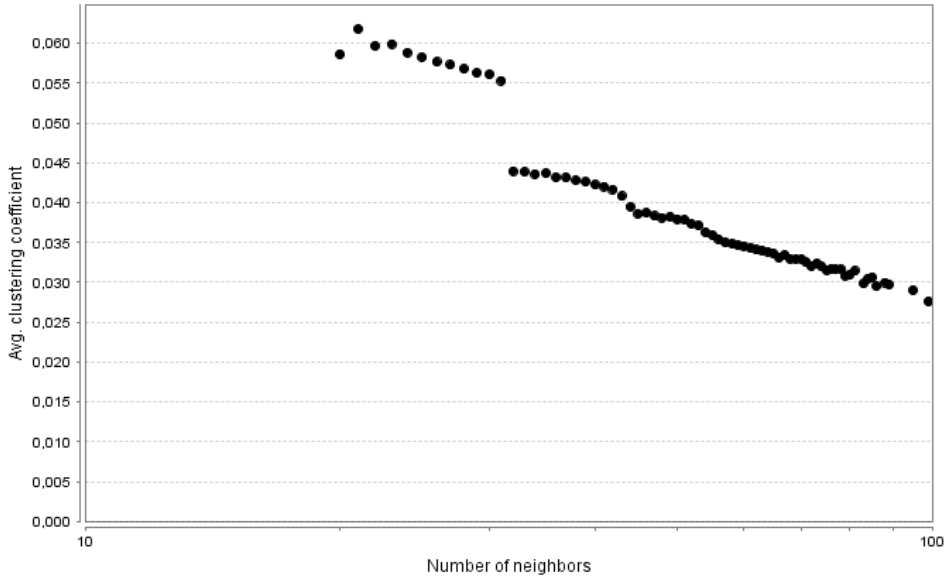
Similmente a quanto visto per la rete a 25.000 nodi, si riconosce una curva esponenziale (decrescente), fatta eccezione per valori di In-Degree compresi tra 23 e 27. Dal punto di vista del load balancing questo è un comportamento migliore rispetto alla rete precedente, perché sta a significare che nella rete c'è una distribuzione livellata meglio (un pochino) dei messaggi in entrata, e sono presenti meno nodi che dominano (a livello di In-Degree).

Out-Degree Distribution



Come era prevedibile la distribuzione dell'Out-Degree è centrata sul numero di bit usati per gli identificatori, con un numero maggiore di nodi con Out-Degree uguale a 16 rispetto alla rete precedente.

AvgClustering Coefficient in base a numero di vicini



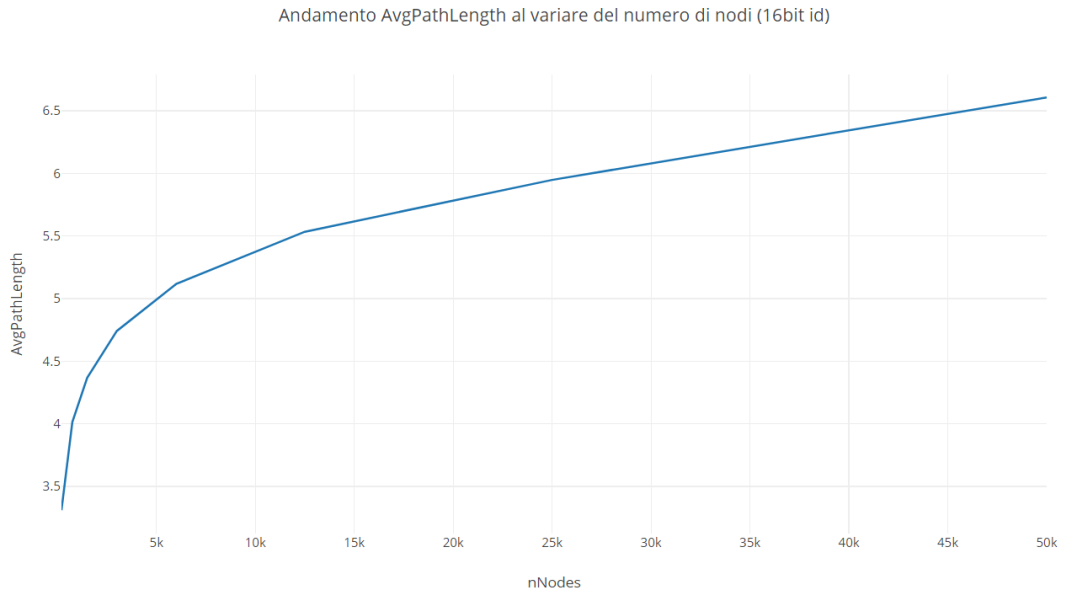
Per quanto riguarda la distribuzione del Clustering Coefficient tra i nodi risalta una situazione differente rispetto alla rete precedente: intorno al valore 50 di vicini si crea "un vuoto" nella curva (una retta decrescente). Inoltre, rispettando quanto visto nella teoria, il coefficiente decresce all'aumentare del numero di nodi (vedi par **3.3** per ulteriori dettagli).

3.3 Confronti tra reti

Scopo di quest'ultimo paragrafo è quello di mettere in evidenza alcune differenze osservate durante le simulazioni del programma, andando ad osservare diversi parametri statistici al variare del numero dei nodi nella rete Chord.

Adamant AvgShortestPath Length

Dalla topologia delle reti (dirette) simulate è stato possibile studiare la media dei cammini minimi al variare del numero dei nodi. Questo studio è differente da quello effettuato nel par. **2.1**, in quanto in quel caso venivano studiate le lunghezze dei cammini durante le simulazioni (eseguite dal programma) del routing.

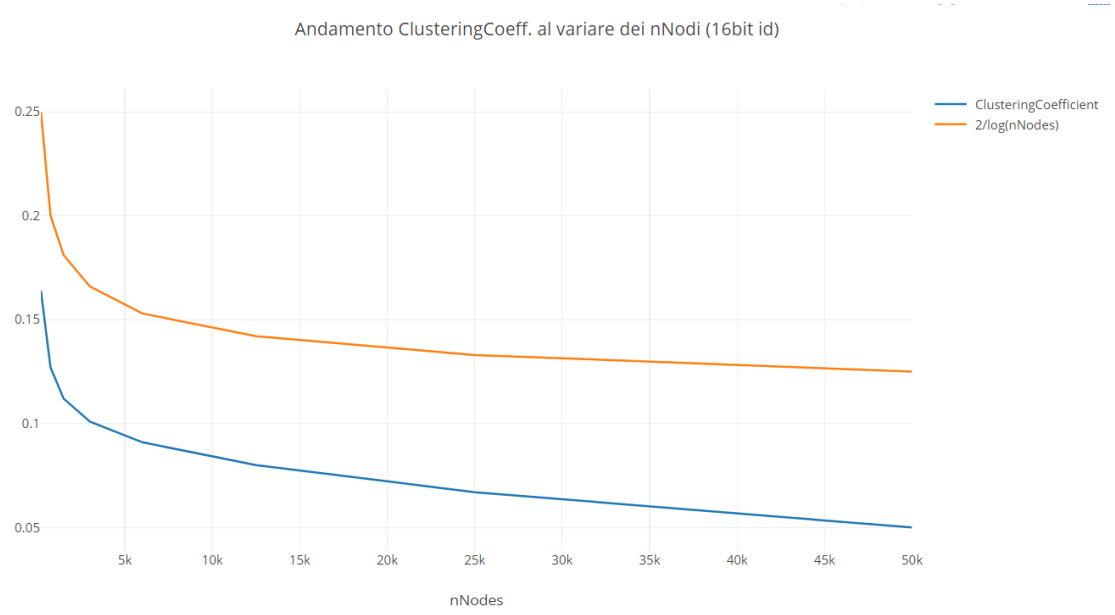


L'andamento evidenziato è quello di una curva logaritmica, e questo risultato è quindi in linea con quanto visto sulla teoria per Chord. La scalabilità di conseguenza è molto buona.

Andamento Clustering Coefficient

Secondo il teorema visto a lezione il Clustering Coefficient di una rete Chord è descritto da una funzione logaritmica (decrescente) in base al numero di nodi, in particolare:

$$CC(G) = \frac{2}{\log_2(nNodes)}$$



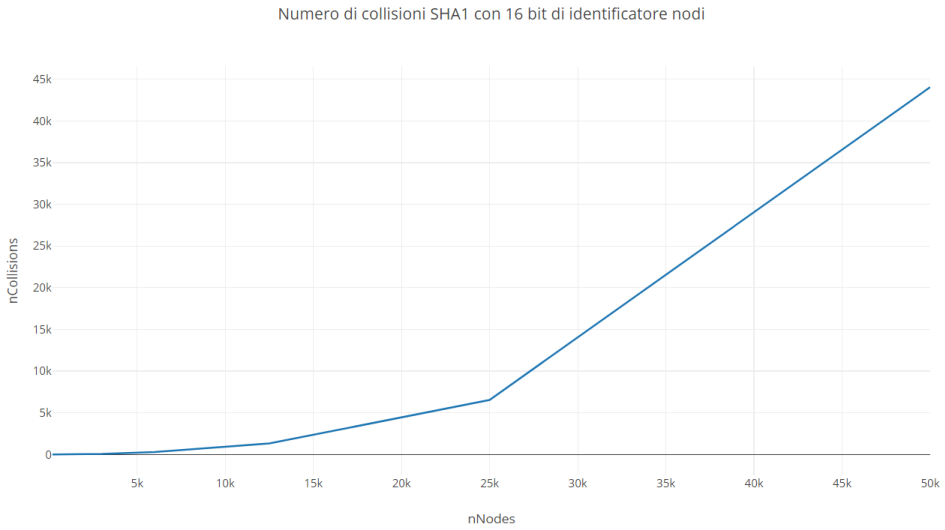
Nel grafico vengono riportate due curve, la funzione che dovrebbe essere restituita secondo il teorema e il Clustering Coefficient analizzato nelle simulazioni effettuate. Risulta un andamento delle due funzioni quasi identico, che però differisce lievemente sui valori. In sostanza potremmo definire la funzione risultante come:

$$CC_{sim}(G) \cong \frac{2}{\log_2(nNodes)} - 0.1 = \Theta\left(\frac{2}{\log_2(nNodes)}\right)$$

Andamento collisioni SHA-1

Arrivati alla fine di questa serie di studi sulle proprietà di Chord nelle simulazioni effettuate, viene proposto uno studio sul numero di collisioni registrate utilizzando il calcolo della funzione modulo, applicata al risultato della funzione hash SHA-1 sugli indirizzi IP generati.

Infatti, anche se si è deciso di riprovare in loop indirizzi IP diversi fin quando non si sono avute collisioni, è chiaro che il numero di collisioni è un dato importante da tenere in considerazione nella progettazione di una rete p2p.



Il numero delle collisioni registrate è direttamente proporzionale al numero dei nodi. In questo caso particolare non si sono avute numerose collisioni fino al riempimento della metà dello spazio degli identificatori (25.000 nodi), ma da questo punto in poi le collisioni sono aumentate in modo consistente.

Questo comportamento è conseguenza della non-uniformità della funzione modulo sullo SHA1, usate per avere un numero variabile di bit di output, e non è quindi un difetto della funzione hash che, per definizione, restituisce un identificatore a 160 bit.

4. Codice Sorgente

Di seguito, diviso per classi java, è mostrato il codice sorgente della simulazione.

4.1 Classe ChordNode

Classe che incapsula un nodo Chord e le sue strutture dati fondamentali.

```java
package com.sisbarra;

import java.math.BigInteger;
import java.util.Collection;

class ChordNode {

    private int mQueryReceivedCounter;
    private int mEndNodeCounter;
    private boolean isLast; //true if id is maximum in network
    private Integer mIntIdentifier;
    private BigInteger mBigIdentifier;
    private ChordFingerTable mFingerTable;
    private ChordNode mPredecessor;
    private ChordNode mSuccessor;

    //Constructor for int identifier
    ChordNode(Integer id){
        mIntIdentifier = id;
        mFingerTable = new ChordFingerTable(false);
        mQueryReceivedCounter = 0;
        mEndNodeCounter = 0;
    }

    //Constructor for bigint identifier
    ChordNode(BigInteger id){
        mBigIdentifier = id;
        mFingerTable = new ChordFingerTable(true);
        mQueryReceivedCounter = 0;
        mEndNodeCounter = 0;
    }

    int getmEndNodeCounter() {
        return mEndNodeCounter;
    }

    Integer getmIntIdentifier() { return mIntIdentifier; }

    BigInteger getmBigIdentifier() {
        return mBigIdentifier;
    }
```

```
43      void setmPredecessor(ChordNode mPredecessor) {
44          this.mPredecessor = mPredecessor;
45      }
46
47      void setmSuccessor(ChordNode mSuccessor) {
48          this.mSuccessor = mSuccessor;
49
50          if(mIntIdentifier != null)
51              isLast = mSuccessor.getmIntIdentifier() < this.mIntIdentifier;
52          else
53              isLast = mSuccessor.getmBigIdentifier().compareTo(mBigIdentifier) < 0;
54      }
55
56      //Add a row to the FingerTable, given node target
57      ChordNode addIntFingerTableRow(Integer target, ChordNode node){
58          return mFingerTable.putIntNode(target, node);
59      }
60
61      //Add a row to the FingerTable, given node target (BigInt)
62      ChordNode addBigIntFingerTableRow(BigInteger target, ChordNode node){
63          return mFingerTable.putBigIntNode(target, node);
64      }
65
66      //Returns a StringBuffer with the topology (source, destination) of node
67      StringBuilder getTopology(){
68          StringBuilder res = new StringBuilder();
69          Collection<ChordNode> neighbors = mFingerTable.getNeighbors();
70
71          //Distinguish Int and BigInt case
72          if(mIntIdentifier != null) {
73              Integer lastNeighbor = null;
74              //For every neighbor print a line (source, dest) in result
75              for (ChordNode n : neighbors) {
76                  //Avoid duplicates
77                  if(n.getmIntIdentifier().equals(lastNeighbor))
78                      continue;
79
80                  res.append(mIntIdentifier).append(", ").append(n.getmIntIdentifier());
81                  res.append(System.getProperty("line.separator"));
82                  lastNeighbor = n.getmIntIdentifier();
83              }
84          }
```

```
85          else {
86              BigInteger lastNeighbor = null;
87              //For every neighbor print a line (source, dest) in result
88              for (ChordNode n : neighbors) {
89                  //Avoid duplicates
90                  if(n.getmBigIdentifier().equals(lastNeighbor))
91                      continue;
92
93                  res.append(mBigIdentifier).append(", ").append(n.getmBigIdentifier());
94                  res.append(System.getProperty("line.separator"));
95                  lastNeighbor = n.getmBigIdentifier();
96              }
97          }
98
99          return res;
100     }
101
102     //Returns the number of Query Received from the node
103     int getQueryReceivedCounter() {
104         return mQueryReceivedCounter;
105     }
106
107     //Increment number of Query Received
108     private void incQueryReceivedCounter(){
109         mQueryReceivedCounter++;
110     }
111
112     //Increment End Node counter
113     private void incEndNodeCounter(){
114         mEndNodeCounter++;
115     }
116
117     //Returns the value of flag isLast (in the ring)
118     private boolean isLast(){
119         return isLast;
120     }
121
```

```
122     //Returns the successor of node id (Int type)
123     ChordNode findIntSuccessor(Integer id, MyCounter hopCounter){
124         incQueryReceivedCounter();
125 
126         Integer idPrec = mPredecessor.getmIntIdentifier(), idSucc = mSuccessor.getmIntIdentifier();
127 
128         //I'm the successor -> stop lookup
129         if((id > idPrec && id <= mIntIdentifier) ||
130                 (mPredecessor.isLast() && (id <= mIntIdentifier || id > idPrec))) {
131             System.out.println("Successor found! It's the "+mIntIdentifier);
132             incEndNodeCounter();
133             return this;
134         }
135 
136         //If id is between me and successor
137         if((id > mIntIdentifier && id <= idSucc) || (isLast && (id > mIntIdentifier || id <= idSucc ))){
138             hopCounter.incValue();
139             mSuccessor.incQueryReceivedCounter();
140             mSuccessor.incEndNodeCounter();
141             System.out.println("Successor found! It's the "+mSuccessor.getmIntIdentifier());
142             return mSuccessor;
143         }
144 
145         //Forward the query around the circle
146         ChordNode next = mFingerTable.getIntLastPredecessor(id);
147         hopCounter.incValue();
148         System.out.println("Forwarding the query to node "+next.getmIntIdentifier());
149         return next.findIntSuccessor(id, hopCounter);
150     }
151 
152     //Returns the successor of node id (Int type)
153     ChordNode findBigIntSuccessor(BigInteger id, MyCounter hopCounter){
154         incQueryReceivedCounter();
155 
156         BigInteger idPrec = mPredecessor.getmBigIdentifier(), idSucc = mSuccessor.getmBigIdentifier();
157 
158         //I'm the successor -> stop lookup
159         if((id.compareTo(idPrec)>0 && id.compareTo(mBigIdentifier)<=0) ||
160                 (mPredecessor.isLast() && (id.compareTo(mBigIdentifier)<=0 || id.compareTo(idPrec)>0))) {
161             System.out.println("Successor found! It's the "+mBigIdentifier);
162             incEndNodeCounter();
163             return this;
164         }
166         //If id is between me and successor
167         if((id.compareTo(mBigIdentifier)>0 && id.compareTo(idSucc)<=0)
168                 || (isLast && (id.compareTo(mBigIdentifier)>0 || id.compareTo(idSucc)<=0 ))){
169             hopCounter.incValue();
170             mSuccessor.incQueryReceivedCounter();
171             mSuccessor.incEndNodeCounter();
172             System.out.println("Successor found! It's the "+mSuccessor.getmBigIdentifier());
173             return mSuccessor;
174         }
175 
176         //Forward the query around the circle
177         ChordNode next = mFingerTable.getBigIntLastPredecessor(id);
178         hopCounter.incValue();
179         System.out.println("Forwarding the query to node "+next.getmBigIdentifier());
180         return next.findBigIntSuccessor(id, hopCounter);
181     }
182 }
183 
```

4.2 Classe ChordFingerTable

Classe contenente metodi e struttura dati di una FingerTable di Chord.

```
8  class ChordFingerTable {
9
10     private TreeMap<Integer, ChordNode> mIntFingerTableMap;
11     private TreeMap<BigInteger, ChordNode> mBigIntFingerTableMap;
12
13     //Construct a FingerTable with Int or BigInt ids
14     ChordFingerTable(boolean isBigInt){
15         if(isBigInt) mBigIntFingerTableMap = new TreeMap<>();
16         else mIntFingerTableMap = new TreeMap<>();
17     }
18
19     //Put a Node with BigInt Id in the FingerTable
20     ChordNode putBigIntNode(BigInteger target, ChordNode node){
21         return mBigIntFingerTableMap.put(target, node);
22     }
23
24     //Put a Node with Int Id in the FingerTable
25     ChordNode putIntNode(Integer target, ChordNode node){
26         return mIntFingerTableMap.put(target, node);
27     }
28
29     //Returns a collection of neighbors node
30     Collection<ChordNode> getNeighbors(){
31         if(mIntFingerTableMap!=null) return mIntFingerTableMap.values();
32         else return mBigIntFingerTableMap.values();
33     }
34
35     //Given a target, returns the last predecessor to it, if target is not in the network (Int identifiers)
36     ChordNode getIntLastPredecessor(Integer id) {
37         TreeSet<Integer> keys = new TreeSet<>(mIntFingerTableMap.keySet());
38
39         if (keys.contains(id)) return mIntFingerTableMap.get(id);
40
41         Integer next = keys.lower(id);
42         //It is null if it is the minimum (the predecessor is the maximum)
43         if (next == null) next = keys.last();
44
45         return mIntFingerTableMap.get(next);
46     }
```

```
48     //Given a target, returns the last predecessor to it, if target is not in the network (BigInt identifiers)
49     ChordNode getBigIntLastPredecessor(BigInteger id) {
50         TreeSet<BigInteger> keys = new TreeSet<>(mBigIntFingerTableMap.keySet());
51
52         if (keys.contains(id)) return mBigIntFingerTableMap.get(id);
53
54         BigInteger next = keys.lower(id);
55         //It is null if it is the minimum (the predecessor is the maximum)
56         if (next==null) next = keys.last();
57
58         return mBigIntFingerTableMap.get(next);
59     }
60
61 }
```

4.3 Classe ChordSim

Contiene i metodi in cui viene eseguita l'intera simulazione.

```
11  class ChordSim {
12
13      //Number of queries to make
14      private static final int NQUERIES = 1000000;
15
16      private int mBits;
17      private int nNodes;
18
19      private TreeMap<Integer, ChordNode> mIntNodeMap;
20      private TreeMap<BigInteger, ChordNode> mBigIntNodeMap;
21
22      //To calculate SHA1 function
23      private MessageDigest mMD;
24
25      //To write network topology on file
26      private FileWriter mTopologyFW;
27      private PrintWriter mTopologyPW;
28      private String TOPOLOGYFILENAME = "Topology";
29
30      //To write number of collisions
31      private FileWriter mCollisionsFW;
32      private PrintWriter mCollisionsPW;
33      private String COLLISIONSFILENAME = "Collisions";
34
35      //To write number of hops for each lookup
36      private FileWriter mHopsFW;
37      private PrintWriter mHopsPW;
38      private String HOPSFILENAME = "Hops";
39
40      //To write number of queries received by every node
41      private FileWriter mQueriesFW;
42      private PrintWriter mQueriesPW;
43      private String QUERIESFILENAME = "Queries";
44
45      //To write EndNode distribution
46      private FileWriter mEndNodeFW;
47      private PrintWriter mEndNodePW;
48      private String ENDNODEFILENAME = "EndNodeDistribution";
49
50      //Generate random IP Address
51      private String calculateRandIP(){
52          Random r = new Random();
53          return r.nextInt(256) + "." + r.nextInt(256) + "." + r.nextInt(256) +
54                  "." + r.nextInt(256);
```

```
57      //Calculate Integer SHA1 id (with mBits)
58      private Integer calculateIntSHA1(String IP){
59          //Update the input for digest
60          mMD.update(IP.getBytes());
61
62          Integer id = (new BigInteger(mMD.digest())).intValue();
63
64          //Modulo Transformation
65          id =  Math.floorMod(id, (int) Math.pow(2, mBits));
66          return id;
67      }
68
69      //Calculate BigInteger SHA1 id (with mBits)
70      private BigInteger calculateBigIntSHA1(String IP){
71          //Update the input for digest
72          mMD.update(IP.getBytes());
73
74          BigInteger id = new BigInteger(mMD.digest());
75
76          //Modulo Transformation
77          id = id.mod(BigInteger.valueOf(2).pow(mBits));
78          return id;
79      }
80
81      //Initialize identifiers in the network
82      private void initializeIDs(){
83          String randIP; int collisionCounter = 0;
84
85          System.out.println("Starting initialization of nodes...");
86
87          //I'm dealing with Int identifiers
88          if(mIntNodeMap!=null) {
89              //For every node
90              for (int i = 0; i < nNodes; i++) {
91                  randIP = calculateRandIP();
92
93                  System.out.println("Calculating SHA-1 for IP Address: " + randIP);
94
95                  Integer id = calculateIntSHA1(randIP);
96                  while((mIntNodeMap.containsKey(id))){
97                      System.out.println("Collision while calculating SHA for "+randIP);
98                      collisionCounter++;
99                      id = calculateIntSHA1(calculateRandIP());
100                 }
```

```
                mIntNodeMap.put(id, new ChordNode(id));
            }
        }
        else{
            //For every node
            for (int i = 0; i < nNodes; i++) {
                randIP = calculateRandIP();

                System.out.println("Calculating SHA-1 for IP Address: " + randIP);

                BigInteger id = calculateBigIntSHA1(randIP);
                while((mBigIntNodeMap.containsKey(id))){
                    System.out.println("Collision while calculating SHA for "+randIP);
                    collisionCounter++;
                    id = calculateBigIntSHA1(calculateRandIP());
                }
                mBigIntNodeMap.put(id, new ChordNode(id));
            }
        }

        //Print the collision counter to file
        mCollisionsPW.println(collisionCounter);
        mCollisionsPW.flush();
    }

    //Build the FingerTable for a node (Int identifiers)
    private void buildIntFingerTable(Integer id, TreeSet<Integer> ids){
        Integer target;

        for(int i=0; i<mBits; i++){
            target = Math.floorMod((int) (id+Math.pow(2, i)), (int) Math.pow(2, mBits));

            //Verify if it exists a node with exactly target id
            if(ids.contains(target)){
                //Add a row at the id node
                mIntNodeMap.get(id).addIntFingerTableRow(target, mIntNodeMap.get(target));
                System.out.println("Added row at finger table of node "+id+", for target "+target+", "+target);
            }
            else{
                //Get the first successor to target
                Integer firstSucc = ids.higher(target);
                //It is null if it is the maximum (the successor is the minimum)
                if (firstSucc==null) firstSucc = ids.first();

                //Add a row for the target
                mIntNodeMap.get(id).addIntFingerTableRow(target, mIntNodeMap.get(firstSucc));
                System.out.println("Added row at finger table of node "+id+", for target "+target+", "+firstSucc);
            }
        }
    }

    //Save the FingerTable (Int identifiers)
    private void saveIntFingerTable(Integer id){
        mTopologyPW.print(mIntNodeMap.get(id).getTopology());
        mTopologyPW.flush();
    }

    //Build the FingerTable for a node (BigInt identifiers)
    private void buildBigIntFingerTable(BigInteger id, TreeSet<BigInteger> ids){
        BigInteger target;

        for(int i=0; i<mBits; i++){
            target = (id.add(BigInteger.valueOf((long) Math.pow(2, i)))).mod(BigInteger.valueOf(2).pow(mBits));

            //Verify if it exists a node with exactly target id
            //Add a row at the id node
            if(ids.contains(target)) {
                mBigIntNodeMap.get(id).addBigIntFingerTableRow(target, mBigIntNodeMap.get(target));
                System.out.println("Added row at finger table of node "+id+", for target "+target+", "+target);
            }
            else{
                //Get the first successor to target
                BigInteger firstSucc = ids.higher(target);
                //It is null if it is the maximum (the successor is the minimum)
                if (firstSucc==null) firstSucc = ids.first();

                //Add a row for the target
                mBigIntNodeMap.get(id).addBigIntFingerTableRow(target, mBigIntNodeMap.get(firstSucc));
                System.out.println("Added row at finger table of node "+id+", for target "+target+", "+firstSucc);
            }
        }
    }
```

```
184     //Set Pred And Succ for node id (Int identifiers)
185     private void buildIntPredSucc(Integer id, TreeSet<Integer> ids){
186         ChordNode node = mIntNodeMap.get(id);
187 
188         Integer pred = ids.lower(id);
189         if(pred==null) pred = ids.last();
190 
191         Integer succ = ids.higher(id);
192         if(succ == null) succ = ids.first();
193 
194         node.setmPredecessor(mIntNodeMap.get(pred));
195         node.setmSuccessor(mIntNodeMap.get(succ));
196     }
197 
198     //Set Pred And Succ for node id (BigInt identifiers)
199     private void buildBigIntPredSucc(BigInteger id, TreeSet<BigInteger> ids){
200         ChordNode node = mBigIntNodeMap.get(id);
201 
202         BigInteger pred = ids.lower(id);
203         if(pred==null) pred = ids.last();
204 
205         BigInteger succ = ids.higher(id);
206         if(succ == null) succ = ids.first();
207 
208         node.setmPredecessor(mBigIntNodeMap.get(pred));
209         node.setmSuccessor(mBigIntNodeMap.get(succ));
210     }
211 
212     //Save the FingerTable (BigInt identifiers)
213     private void saveBigIntFingerTable(BigInteger id){
214         mTopologyPW.print(mBigIntNodeMap.get(id).getTopology());
215         mTopologyPW.flush();
216     }
```

```
218    //Initialize for every node its FingerTable
219    private void initializeFingerTables(){
220        //We're working with Int
221        if(mIntNodeMap!=null){
222            //Sorted identifiers
223            TreeSet<Integer> ids = new TreeSet<>(mIntNodeMap.keySet());
224 
225 
226            //For every node
227            for(Integer id: ids){
228                buildIntPredSucc(id, ids);
229                buildIntFingerTable(id, ids);
230                saveIntFingerTable(id);
231            }
232        }
233        //We're working with BigInt
234        else{
235            //Sorted identifiers
236            TreeSet<BigInteger> ids = new TreeSet<>(mBigIntNodeMap.keySet());
237 
238 
239            //For every node
240            for(BigInteger id: ids){
241                buildBigIntPredSucc(id, ids);
242                buildBigIntFingerTable(id, ids);
243                saveBigIntFingerTable(id);
244            }
245        }
246    }
```

```
248     //Create the writers for the csv files (parameters included in filename)
249     private void initializeFiles() throws IOException {
250         //Modify filename with parameters
251         TOPOLOGYFILENAME = TOPOLOGYFILENAME.concat(mBits+"bit"+nNodes+"nodes"+".csv");
252         COLLISIONSFILENAME = COLLISIONSFILENAME.concat(mBits+"bit"+nNodes+"nodes"+".txt");
253         HOPSFILENAME = HOPSFILENAME.concat(mBits+"bit"+nNodes+"nodes"+".csv");
254         QUERIESFILENAME = QUERIESFILENAME.concat(mBits+"bit"+nNodes+"nodes"+".csv");
255         ENDNODEFILENAME = ENDNODEFILENAME.concat(mBits+"bit"+nNodes+"nodes"+".csv");
256 
257         //Topology file
258         mTopologyFW = new FileWriter(TOPOLOGYFILENAME);
259         mTopologyPW = new PrintWriter(mTopologyFW);
260         mTopologyPW.println("Source, Target");
261 
262         //Collisions file
263         mCollisionsFW = new FileWriter(COLLISIONSFILENAME);
264         mCollisionsPW = new PrintWriter(mCollisionsFW);
265         mCollisionsPW.println("Number of Collisions");
266 
267         //Hops file
268         mHopsFW = new FileWriter(HOPSFILENAME);
269         mHopsPW = new PrintWriter(mHopsFW);
270         mHopsPW.println("Number of Hops for each lookup");
271 
272         //Queries file
273         mQueriesFW = new FileWriter(QUERIESFILENAME);
274         mQueriesPW = new PrintWriter(mQueriesFW);
275         mQueriesPW.println("NodeId, nQueriesReceived");
276 
277         //EndNodeDistribution file
278         mEndNodeFW = new FileWriter(ENDNODEFILENAME);
279         mEndNodePW = new PrintWriter(mEndNodeFW);
280         mEndNodePW.println("NodeId, nEndNodeInLookup");
281     }
282 
283     //Execute the routing between two nodes (Int identifiers)
284     private void routeIntFromTo(Integer startId, Integer endId){
285         MyCounter hopCounter = new MyCounter(0);
286         mIntNodeMap.get(startId).findIntSuccessor(endId, hopCounter);
287 
288         //Prints occurred hops in this routing
289         mHopsPW.println(hopCounter.getmCountValue());
290     }
```

```
292     //Execute the routing between two nodes (Int identifiers)
293     private void routeBigIntFromTo(BigInteger startId, BigInteger endId){
294         MyCounter hopCounter = new MyCounter(0);
295         mBigIntNodeMap.get(startId).findBigIntSuccessor(endId, hopCounter);
296 
297         //Prints occurred hops in this routing
298         mHopsPW.println(hopCounter.getmCountValue());
299     }
300 
301     //Simulate routing with n starting nodes
302     private void simulateRouting(){
303         Random randomizer = new Random();
304 
305         System.out.println("Starting routing simulation");
306 
307         //Int case
308         if(mIntNodeMap != null) {
309             //KeySet
310             List<Integer> ids = new ArrayList<>(mIntNodeMap.keySet());
311 
312             Integer startId, targetId;
313 
314             //Choose n times a node at random to start the lookup
315             for (int i = 0; i < NQUERIES; i++) {
316                 //Generate random start and target
317                 startId = ids.get(randomizer.nextInt(ids.size()));
318                 targetId = calculateIntSHA1(calculateRandIP());
319                 System.out.println("Routing from "+startId+" to target "+targetId);
320 
321                 //Execute the routing
322                 routeIntFromTo(startId, targetId);
323             }
324         }
325         //BigInt case
326         else{
327             //KeySet
328             List<BigInteger> ids = new ArrayList<>(mBigIntNodeMap.keySet());
329 
330             BigInteger startId, targetId;
331 
332             //Choose n times a node at random to start the lookup
333             for (int i = 0; i < NQUERIES; i++) {
334                 //Generate random start and target
335                 startId = ids.get(randomizer.nextInt(ids.size()));
```

```
                targetId = calculateBigIntSHA1(calculateRandIP());
                System.out.println("Routing from "+startId+" to target "+targetId);

                //Execute the routing
                routeBigIntFromTo(startId, targetId);
            }
        }
    }

    //Print to files the EndNode distribution and the query distribution
    private void printRoutingStatistics(){
        //Integer case
        if(mIntNodeMap!=null){
            Collection<ChordNode> nodes = mIntNodeMap.values();
            for(ChordNode n: nodes){
                //Print the EndNode Counter
                mEndNodePW.println(n.getmIntIdentifier()+", "+n.getmEndNodeCounter());
                //Print the QueryReceived Counter
                mQueriesPW.println(n.getmIntIdentifier()+", "+n.getQueryReceivedCounter());
            }
        }
        else{
            Collection<ChordNode> nodes = mBigIntNodeMap.values();
            for(ChordNode n: nodes){
                //Print the EndNode Counter
                mEndNodePW.println(n.getmBigIdentifier()+", "+n.getmEndNodeCounter());
                //Print the QueryReceived Counter
                mQueriesPW.println(n.getmBigIdentifier()+", "+n.getQueryReceivedCounter());
            }
        }

        //Flushes the buffers
        mEndNodePW.flush();
        mQueriesPW.flush();
    }

    //Close the opened files
    private void closeFiles() throws IOException {
        System.out.println("Closing files for statistics...");

        mTopologyPW.close();
        mTopologyFW.close();
        mCollisionsPW.close();
        mCollisionsFW.close();
        mHopsPW.close();
        mHopsFW.close();
        mQueriesPW.close();
        mQueriesFW.close();
        mEndNodePW.close();
        mEndNodeFW.close();
    }


    ChordSim(int m, int n) throws NoSuchAlgorithmException {
        mBits = m;
        nNodes = n;

        //Verify if BigInt is needed
        if (mBits>=32){
            mIntNodeMap = null;
            mBigIntNodeMap = new TreeMap<>();
        }
        else{
            mBigIntNodeMap = null;
            mIntNodeMap = new TreeMap<>();
        }

        //Initialize the Digest for SHA1
        mMD = MessageDigest.getInstance("SHA1");

        //Initialize writing files
        try {
            initializeFiles();
        } catch (IOException e) {
            System.err.println("Error in opening files, exiting from the program...");
            return;
        }

        //Initialize ChordNode IDs
        initializeIDs();
```

```
414         //Initialize ChordNode IDs
415         initializeIDs();
416
417         //Initialize ChordNode FingerTables
418         initializeFingerTables();
419
420         //Simulate Routing, looking up of n nodes
421         simulateRouting();
422
423         printRoutingStatistics();
424
425         //Close files opened for statistics
426         try {
427             closeFiles();
428         } catch (IOException e) {
429             System.err.println("Error in closing files, exiting from the program...");
430         }
431     }
432 }
```

4.4 Classe MyCounter

Classe di utility, contatore personalizzato.

```
1  package com.sisbarra;
2
3  class MyCounter {
4
5      private int mCountValue;
6
7      MyCounter(int startValue){
8          mCountValue = startValue;
9      }
10
11     void incValue(){
12         mCountValue++;
13     }
14
15     int getmCountValue() {
16         return mCountValue;
17     }
18 }
```

4.5 Classe Main

Al suo interno viene controllata la correttezza e la semantica dei parametri di input.

```
6  public class Main {
7
8      private static final int MDEFAULT = 160;
9      private static final int NDEFAULT = 1000;
10
11     //Verify correctness of input parameters
12     private static boolean verifyParameters(int mBits, int nNodes){
13
14         //Verify, if nNodes > (2^mBits)-1
15         if((nNodes <= 0) || (mBits < 0) ||
16                 ((BigInteger.valueOf(nNodes).compareTo(BigInteger.valueOf(2).pow(mBits)
17                         .subtract(BigInteger.valueOf(1)))) > 0
18                 || (mBits > 160)){
19             System.err.println("Error in params semantic");
20             return false;
21         }
22
23         return true;
24     }
25
26     public static void main(String[] args) {
27         if(args.length != 2) {
28             System.out.println("Usage: ChordSim m n");
29             System.out.println("Using default params: m=160 n=1000");
30             try {
31                 new ChordSim(MDEFAULT, NDEFAULT);
32             } catch (NoSuchAlgorithmException e) {
33                 System.err.println("Error in SHA1 execution");
34             }
35         }
36
37         try{
38             int m = Integer.parseInt(args[0]), n = Integer.parseInt(args[1]);
39
40             if(verifyParameters(m, n))
41                 new ChordSim(m, n);
42         }
43         catch(NumberFormatException e){
44             System.err.println("Error in params format");
45         } catch (NoSuchAlgorithmException e) {
46             System.err.println("Error in SHA1 execution");
47         }
48     }
49 }
```