Peer to Peer Systems and Blockchains

2019/2020 Mid Term

Marco Costa
m.costa22@studenti.unipi.it
545144

1 IPFS Swarm Monitoring

All the commented source code with purposes for all the modules and the methods is listed in the Appendix of the paper and hosted on github.com/mcdeldams/p2pmid2020.

The application was implemented using the IPFS's Go HTTP API[1], which allows to communicate with an IPFS daemon running in a separate process. It's important to stay that all the results proposed were took on a fresh installation of IPFS without performing any task in the meantime.

The monitoring was executed for 65 hours using an Intel Core i5-8250U CPU with 8 GB of RAM machine connected to a 100 Mbps FTTC home network. Fedora 31 was used as OS, however all the sources compile as well on Windows. Moreover, the following GO external libraries were used:

- *ipinfo/go*[2] for the country lookup
- *sparrc/go-ping*[3] for inspecting the effective RTT of the peers
- *tidwall/gjson*[4] for JSON parsing

As requested an evaluation of the swarm peers, including the amount of peers belonging to the IPFS swarm, the provenance country, the RTT and the hour-daily behavior of the swarm, was performed every hour from 7 PM of April 7th to 11 AM[5] of April 10th (total of 65 hours). The application outputs a JSON file of the form:

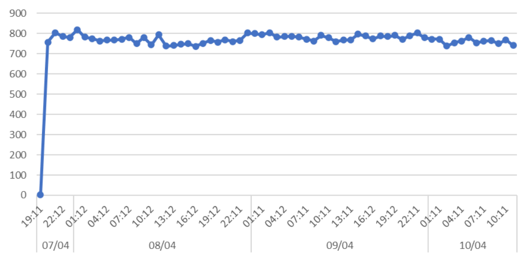
```
[{
    "DayTime": // day of the measurement,
    "HourTime": // time of the measurement,
    "ConnectedPeers": // number of peers in the current swarm
    "Protocol": [
        // peers per protocol used
    ]
    "LocationCount": [
        // peers per location
```

[1]https://github.com/ipfs/go-ipfs-api
[2]https://github.com/ipinfo/go
[3]https://github.com/sparrc/go-ping
[4]https://github.com/tidwall/gjson
[5]All the times referred are GMT+2

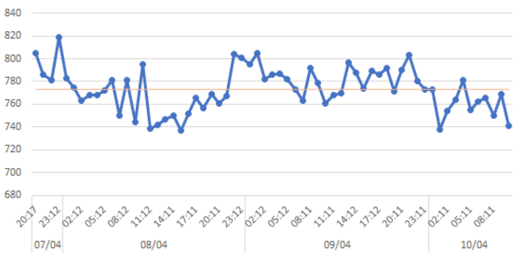
```
    ],
    "Bucket": [
        // last ith-bucket (i is a parameter) [optional exercise]
    ],
    "BucketChurn": // churn of the ith-bucket [optional exercise]
    "BucketNewPeers": // new peers since last test for the ith-
       bucket [optional exercise]
    "MinRTT": // min RTT for the current swarm
    "AvgRTT": // avg RTT           "
    "MaxRTT": // max RTT           "
}, {
    ... // next measurements
}]
```

The results obtained can be seen in Figure 1. Some interesting properties found are:

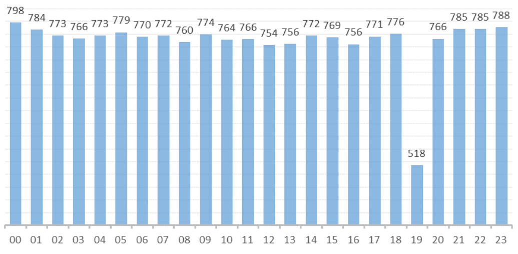
- In the first measurement, executed some seconds after the daemon start, the only swarm peers found are four bootstrap nodes (1a)
- An average number of 772 peers in the swarm was measured (1b)
- After the bootstrap, the number of peers in the swarm remains stable, no drastic changes were recorded (1b)
- The highest concentration of peers in the swarm was found in the 9 PM - 1 AM range with a peak at midnight (1c, 1d)
- The lowest concentration of peers was found in the 12 AM - 1 PM range (1c, 1d)
- The average RTTs of the distinct swarms were all around about 200ms, except for the first measurement where the swarm was formed only by bootstrap nodes and the average was halved (1e)
- At 4 PM was found the only IP using the IPv6 protocol (1f)
- The 67.08% of the IPs monitored was Chinese, followed by Americans (13.47%) and Germans (4.41%) (1g)
- 3945 distinct IPs were encountered, which gives an average of almost 61 distinct IPs per swarm, that is the 7.90% over the average swarm of 772 peers (1a, 1g)
- IPs from 67 different countries were recorded, which covers more than one third of Earth's sovereign states (1g)
- Summoning all the swarms monitored, the Chinese peers take almost the 81% of the set, then the Americans with 8.72% and finally Germans with 2.53% (1h)



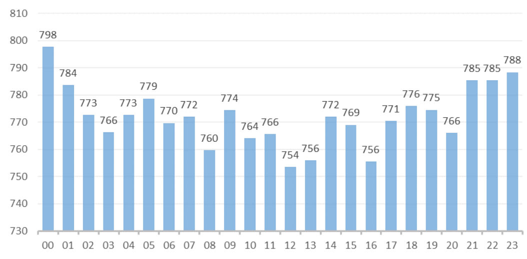
(a) Number of peers in the swarm (cold start)



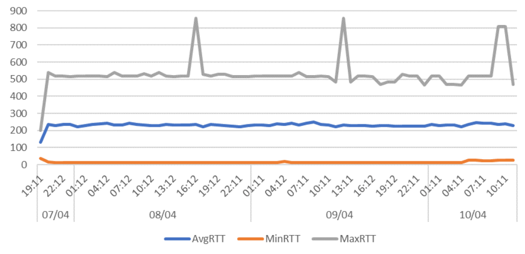
(b) Number of peers in the swarm (hot start)



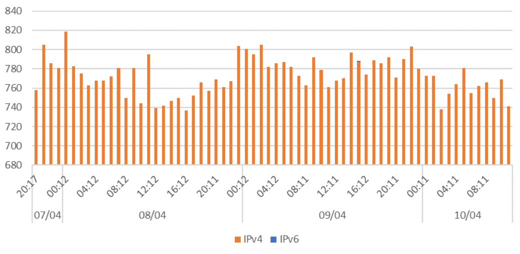
(c) Hour-based average of peers in the swarm (cold start)



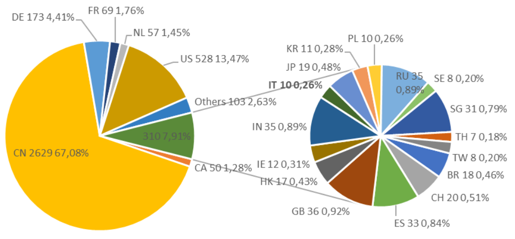
(d) Hour-based average of peers in the swarm (hot start)



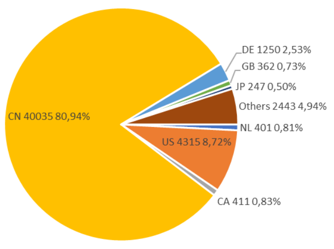
(e) Average, minimum and maximum RTT of the swarm (ms)



(f) IP protocol of the swarm peers



(g) Country of each distinct IP found in the swarm. **Note:** the "Others" label groups 45 distinct states



(h) Country distribution for all the swarms monitored

Figure 1: Results for the mandatory part of the midterm

1.1 Optional

Both the optional exercise proposed during the lectures were implemented, however the first one hasn't been successfully terminated due to lack of computational power.

Exercise 1: *a (smart?) brute-forcing of the Kademlia routing table.*

The algorithm implemented works as follows: it starts testing all the `PeerIDs` in the swarm, when a new identifier *id* that covers a new bucket is found[6], the command `ipfs dht query id` is performed using a long timeout and all the identifiers returned are tested recursively. This method hopes to find new `PeerIDs` in near buckets. Once those IDs are finished, the same routine is performed using random-generated CIDv0. The source code can be found in the *dht.go* file[7].

Exercise 2: *inspecting a single bucket of the Kademlia routing table.*

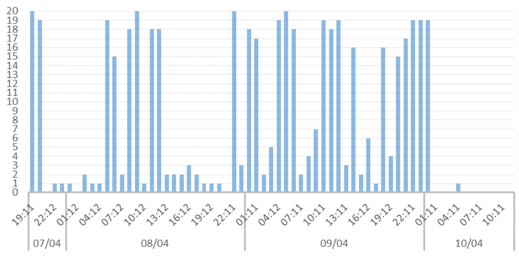
Regarding the second exercise, the relative script was executed with the same frequency as the mandatory one and then, for a more accurate result, every 5 minutes for 6 hours; for every bucket measured, the parameters inspected were:

1. the number of replaced peers from the previous bucket
2. the number of replaced peers from the previous bucket which left the IPFS network

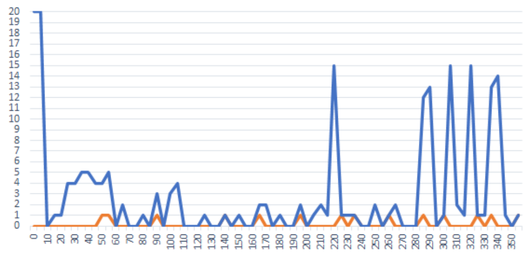
The results are presented in Figure 2. As we can see from Figure 2a, the periodic refreshment of the Kademlia routing table does not always coincide with an almost-full replacement of the Peers in the bucket; in fact, this is coherent with what can be seen in Figure 2b, where no drastic changes happen from the 100th to the 220th minute. Also, small changes in the bucket are often related to peers quitting the network (represented by the orange line) which may indicate a refresh based on new information received by the node through IPFS queries. Finally, in Figure 2c is what we obtain if we do not consider the measurements under 2 new peers and the ones when a replaced peer has left the network, which is more similar to the periodic refreshment expected.

[6] According to the `libp2p` documentation: *"`dXOR(sha256(id1), sha256(id2))` is the number of common leftmost bits between SHA256 of each peer IDs. The `dXOR` between us and a peer X designates the bucket index that peer X will take up in the Kademlia routing table"*

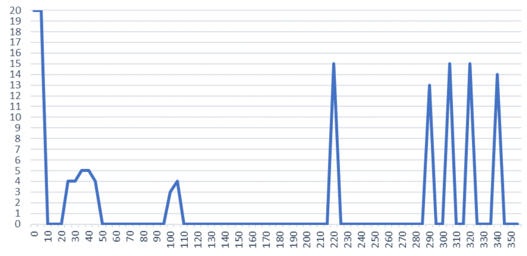
[7] The external library `https://www.github.com/ipfs/go-cid` was used to verify the CIDs correctness



(a) New peers in the bucket since the previous measurement



(b) In blue the number of peers replaced and in orange how many of them has left the network after x minutes. No more than one peer in 5 minutes left the IPFS network.



(c) Number of replaced peers considering only the measurements > 2 where nodes didn't left the network

Figure 2: Results for the inspection of a Kademlia bucket

2 The Kademlia DHT

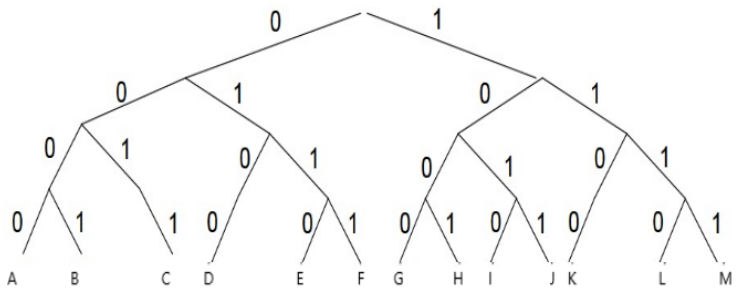


Figure 3: Labelling used for the nodes identifiers

Index	Bucket
0	M J

(a) C's routing table

Index	2-bucket	
1	G	H
2	K	-
3	L	-

(b) M's routing table

Figure 4: Content of the referred routing tables. **Note**: not relevant buckets are omitted.

Assumptions: we assume that all the nodes in the tree remain alive and reachable and that the content of routing tables are the same as Figure 4.
Notation: in addition to the labelling described in Figure 3, we'll use "X.KAD[i]" to refer to the i-th k-bucket of the routing table of the node X and X.id to refer to the `PeerID` of the node X.

Question 1.
The node responsible for the identifier `01000001` is the node D (`01001101`) because it's the closest one to the identifier according to the `XOR` metric.
Question 2.
Let's label as N the node with `PeerID` `11010101` and C the bootstrap node `00110011`. The node join routine works as follows:

1. The initial routing table of N is a single 2-bucket containing *at least* `(PeerID, IP, UDP port)` of N and C. (Figure 5a)
2. N executes `C.FIND_NODE(N.id)` with an RPC.
3. C replies with his *entire* "N.id's common prefix" index of the routing table. Because they differ since the first bit, their common prefix is 0 and, since $k-|\text{C.KAD}[0]| = 0$, only C.KAD[0] is returned.

4. The common prefixes of the `PeerIDs` of J and M with the node N are respectively 1 and 2, so they are added to N.KAD[1] and N.KAD[2] respectively. Now, the initial single bucket can be splitted up, the node C goes into N.KAD[0] while the self-node N is removed. (Figure 5b)

5. At this point, the node N will start to enrich the further buckets, from the one which contains the bootstrap node, using `FIND_NODE` with random-generated identifiers as a parameter. As an example of a first iteration, N generates the id `11010110` and executes the `FIND_NODE` RPC through the closest node to `11010110`. Thus, because M is the closest node according to the `XOR` metric[8], `M.FIND_NODE(11010110)` is executed[9].

6. N uses the results of those methods to populate its routing table. As described in Point 3, since the common prefix between M.id and `11010110` is 2, `M.KAD[2]` is returned to N. However, since $k - |M.KAD[2]| = 1$ also the node L, which has the second closest ID to N in the M's routing table, is returned. The new routing table of N can be seen in Figure 5c.

7. Furthermore, other peers in the network will start discovery N and will add the node in their routing table, and so will do N adding peers known through IPFS queries to its routing table.

Index	Bucket	
-	N	C

(a)

Index	2-bucket	
0	C	-
1	J	-
2	M	-

(b)

Index	2-bucket	
0	C	-
1	J	-
2	M	L
3	K	-

(c)

Figure 5: Representation of N.KAD over time

[8]Trivial according to the indexes, since $2^k > \sum_{i=0}^{i<k} 2^i$

[9]**Note:** another possible implementation could execute a *node lookup* for the random-generated identifier in order to discover new peers to insert in the routing table, in that case α `FIND_NODE(id)` would be executed parallelly through the α contacts closest to the key (in this case M and J), once obtained the k closest nodes the procedure will be executed iteratively until no closer node to the target is returned.

Appendix

Sources

Listing 1: config.go

```
package main

import "time"

// This module provides the configuration options for the application
//
// Author: Marco Costa

// The IPFS HTTP API address, needed for contacting the IPFS daemon instance
const ipfsAPI string = "localhost:5001"

// The API token for https://ipinfo.io/ is needed for the country IP lookup
const apiToken string = "2ea5eaba35e3dd"

// Default parameters for the routine interval and the max number of executions
// can be overwritten using command line arguments
// @see usage
var routineInterval time.Duration = 1 * time.Hour
var maxExecutions int = 72

// If this is true and the script is run in privileged mode the pings
    statistics
// are enabled
var enablePings bool = true

// If this is true the RTT for every known IP is saved and not recalculated in
    case
// of future uses
const cachePings bool = true

// Default number of ping iterations and timeout
// higher values give a more accurate response
const pingIterations int = 1
const pingTimeout = 1 * time.Second

// Default filename for the json output
// Format: HHHH-MM-DD HH-MM.json
var dataFilename string = time.Now().Format("2006-01-02 15-04") + ".json"

// Default format for the timestamps
// Day: HHHH-MM-DD
// Time: HH:MM:SS
const jsonDayFormat string = "2006-01-02"
const jsonHourFormat string = "15:04:05"

// The format of the RTT statistics
const rttFormat int64 = int64(time.Millisecond)
```

Listing 2: script.go

```
package main

// This module provides the script routine for the mandatory part and the
    bucket churn optional
// the results are encoded in a json file with timestamp as name.
// The script can be closed in a sae way using an interrupt signal
// IMPORTANT: the IPFS daemon must be executing on "localhost:5001" for
    changing this parameter
//          @see config.go
//
// NOTE: for executing the ping the script must be executed in privileged
    mode, otherwise RTT 0 is returned
//
// Author: Marco Costa

import (
   "encoding/json"
   "fmt"
   "log"
   "os"
   "os/signal"
   "strconv"
   "syscall"
   "time"

   "github.com/ipinfo/go-ipinfo/ipinfo"
   "github.com/sparrc/go-ping"
)

var client *ipinfo.Client

// Defines a backup matrix used for next evaluations purposes
type backupDB struct {
   Swarm [][]string
}

type rtt struct {
   AvgRTT int64
   MaxRTT int64
   MinRTT int64
}
```

```go
// The JSON output data format
type outputData struct {
   DayTime            string
   HourTime           string
   ConnectedPeers     int
   ProtocolCount      *map[string]int
   LocationCount      *map[string]int
   Bucket             *[bucketSize]string
   BucketChurn        int
   BucketNewPeers     int
   BucketChurnNewPeers int
   RTTInfos           rtt
}

var db backupDB

// Initializes the JSON output structs
func (d *outputData) initData(i *Ipfs) {
   t := time.Now()
   d.DayTime = t.Format(jsonDayFormat)
   d.HourTime = t.Format(jsonHourFormat)
   d.ConnectedPeers = len(i.sw.lastSwarm)
   d.LocationCount = &i.sw.lastSwarmLocationCount
   d.ProtocolCount = &i.sw.lastSwarmProcolCount
   d.Bucket = &i.bu.lastBucket
   d.BucketChurn = i.bu.lastBucketOffline
   d.BucketChurnNewPeers = i.bu.lastBucketNewPeersOffline
   d.BucketNewPeers = i.bu.lastBucketNewPeers
   if enablePings {
      d.RTTInfos.AvgRTT = i.sw.avg / rttFormat
      d.RTTInfos.MaxRTT = i.sw.max / rttFormat
      d.RTTInfos.MinRTT = i.sw.min / rttFormat
   }
}

// Writes data to a file in APPEND | CREATE mode
func writeDataFile(filename string, data []byte) {
   f, err := os.OpenFile(filename, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
   if err != nil {
      log.Fatal(err)
   }
   if _, err := f.Write(data); err != nil {
      f.Close() // ignore error; Write error takes precedence
      log.Panicln(err)
   }
   if err := f.Close(); err != nil {
      log.Panicln(err)
   }
}
```

```go
// Routine execution
func routine(ipfs *Ipfs, i int) {
   /* ipfs.GetSwarmPeers()
   ipfs.GetSwarmInfos(i) */
   ipfs.GetBucket(ipfs.selfID, bucketSize)

   var d outputData
   d.initData(ipfs)

   jsonString, _ := json.MarshalIndent(d, "", " ")

   jsonString = append(jsonString, byte(','))
   writeDataFile(dataFilename, jsonString)
   log.Printf("#%v/%v read successful - #swarm peers %v\n", i, maxExecutions,
       d.ConnectedPeers)
}

// This is the script method which handles the routine execution
func script() {
   var ipfs Ipfs
   ipfs.InitIPFS()
   authTransport := ipinfo.AuthTransport{Token: apiToken}
   httpClient := authTransport.Client()
   client = ipinfo.NewClient(httpClient)

   db.Swarm = make([][]string, maxExecutions)

   log.Println("PeerID: ", ipfs.selfID)

   writeDataFile(dataFilename, []byte("["))

   // Listening to signals
   sig := make(chan os.Signal)
   signal.Notify(sig, syscall.SIGTERM)
   // First execution before using the ticker
   ticker := time.NewTicker(routineInterval)
   routine(&ipfs, 1)
   // Routine
R:
   for i := 2; i <= maxExecutions; i++ {
      select {
      case z := <-sig:
         log.Printf("received %v, graceful closing started", z)
         break R
      case <-ticker.C:
         routine(&ipfs, i)
      }
   }
```

```
    // End of routine
    ticker.Stop()
    writeDataFile(dataFilename, []byte("]"))

    // Saving the db for next evaluations
    dbJSON, err := json.MarshalIndent(db, "", " ")
    if err != nil {
        log.Panic(err)
    }
    writeDataFile("db.json", dbJSON)

    log.Println("script successfully terminated")
    os.Exit(0)

}

func printUsage(name string) {
    fmt.Printf("*******************************************\n")
    fmt.Printf("USAGE:\n\t%v --script [TICKER] [ITERATIONS] // executes the
        routine every [TICKER] minutes per [ITERATIONS]\n", name)
    fmt.Printf("\t%v --dht // tries to reconstruct the DHT table of the
        Peer\n", name)
    fmt.Printf("DEFAULT VALUES:\n\tTICKER = %v\n\tITERATIONS = %v\n",
        routineInterval, maxExecutions)
}

func main() {
    args := os.Args

    if len(args) <= 1 || (len(args) > 1 && !(args[1] == "--script" || args[1]
        == "--dht")) {
        printUsage(args[0])
        os.Exit(0)
    }

    if args[1] == "--script" && enablePings {
        p, _ := ping.NewPinger("8.8.8.8")
        if !p.Privileged() {
            fmt.Printf("*******************************************\n")
            fmt.Printf("If you need the ping statistics the application must run
                in privileged mode\n")
            fmt.Printf("Disabling pings and starting the application\n")
            fmt.Printf("*******************************************\n")
            enablePings = false
        }
    }
    if args[1] == "--script" && len(args) != 4 {
        script()
```

```
  } else if args[1] == "--script" && len(args) == 4 {
    temp, err := time.ParseDuration(args[2])
    if err != nil {
      printUsage(args[0])
      os.Exit(0)
    }
    routineInterval = temp
    t, err := strconv.ParseInt(args[3], 10, 64)
    if err != nil {
      printUsage(args[0])
      os.Exit(0)
    }

    maxExecutions = int(t)
    script()
  } else {
    dht()
  }
}
```

Listing 3: ipfs.go

```
package main

// This module provides the structures and methods for the interaction with
    the IPFS daemon
// including the methods for getting the swarm, bucket, etc.
//
// Author: Marco Costa

import (
  "bytes"
  "context"
  "log"
  "math"
  "net"
  "os/exec"
  "strings"
  "time"

  shell "github.com/ipfs/go-ipfs-api"
  "github.com/tidwall/gjson"
)

// The Kademlia bucket size
const bucketSize int = 20

var timeout time.Duration = 500 * time.Millisecond
```

```go
type ipInfo struct {
   nation  string
   protocol string
}

type swarm struct {
   lastSwarm             []shell.SwarmConnInfo // Last swarm array
   lastSwarmLocationCount map[string]int    // Nation -> #Peer
   lastSwarmProcolCount map[string]int      // Protocol -> #Peer
   min                  int64
   avg                  int64
   max                  int64
}

type bucket struct {
   lastBucket              [bucketSize]string
   lastBucketNewPeers      int
   lastBucketNewPeersOffline int
   lastBucketOffline       int
}

// Ipfs is the main structure for the IPFS node
type Ipfs struct {
   selfID string          // The ID of the local peer
   sh     *(shell.Shell)  // The shell for daemon interaction
   ipInfo map[string]ipInfo // IP -> ipInfo
   sw     swarm
   bu     bucket
}

// Given an IPFS address returns the IP address and the transport protocol used
func trimIpfsAddress(addr string) (string, string) {
   /* ["", ip version protocol, ip address, transport protocol, port] */
   split := strings.Split(addr, "/")
   return split[2], split[1]
}

// getIPCountry returns the ip country relative to "ip"
// Argument: the ip to check
func getIPCountry(ip string) string {
   res, err := client.GetCountry(net.ParseIP(ip))
   if err != nil {
      return "Undefined"
   }

   return strings.TrimRight(res, "\n")
}

// InitIPFS initializes a new Ipfs struct
// Arguments: address of the local API
func (i *Ipfs) InitIPFS() {
   i.sh = shell.NewShell(ipfsAPI)
   cmd := exec.Command("ipfs", "id")
   stdout, err := cmd.Output()
   if err != nil {
      log.Fatalf("The daemon isn't started on %v\nPlease start the daemon or
          modify \"ipfsAPI\" in config.go\n", ipfsAPI)
   }
   i.selfID = gjson.Get(string(stdout), "ID").String()
   i.ipInfo = make(map[string]ipInfo)
}

// GetSwarmInfos construct the infos (location, ipversion, rtt) for the last
    swarm
// Argument: n the iteration number
func (i *Ipfs) GetSwarmInfos(n int) {
   // Resetting memory
   // this is the faster way proposed by Golang docs
   i.sw.lastSwarmLocationCount = nil
   i.sw.lastSwarmLocationCount = make(map[string]int)
   i.sw.lastSwarmProcolCount = nil
   i.sw.lastSwarmProcolCount = make(map[string]int)
   i.sw.avg = int64(0)
   i.sw.min = int64(math.MaxInt64)
   i.sw.max = int64(0)
   found := 0

   /* saving backup */
   db.Swarm[n-1] = make([]string, len(i.sw.lastSwarm))

   for j := 0; j < len(i.sw.lastSwarm); j++ {
      ipAddr, ipProto := trimIpfsAddress(i.sw.lastSwarm[j].Addr)
      curr, ok := i.ipInfo[ipAddr]
      if ok {
         // Location known previously
      } else { // Location unknown
         curr.nation = getIPCountry(ipAddr)
         curr.protocol = ipProto
         i.ipInfo[ipAddr] = curr
      }
      // Add the location to the total count
      i.sw.lastSwarmLocationCount[curr.nation]++
      i.sw.lastSwarmProcolCount[curr.protocol]++

      // If enabled ping the address to add his RTT to the stats
      if !enablePings {
         continue
      }
    rtt, err := getRTT(ipAddr)
    if err != nil || rtt == 0 {
      log.Printf("[!!] can't get %v RTT", ipAddr)
      continue
    }
    // Compute the MIN, MAX, AVG stats
    rttNano := rtt.Nanoseconds()
    i.sw.avg += rttNano
    found++
    if rttNano < i.sw.min {
      i.sw.min = rttNano
    }
    if rttNano > i.sw.max {
      i.sw.max = rttNano
    }
  }

  if found > 0 {
    i.sw.avg = i.sw.avg / int64(found)
  }
}

// GetSwarmPeers gets the current swarm peers and saves it in the lastSwarm
     array
func (i *Ipfs) GetSwarmPeers() {
  info, err := i.sh.SwarmPeers(context.Background())
  if err != nil {
    log.Fatal(err)
  }
  i.sw.lastSwarm = info.Peers
}

// GetBucket inserts in i.lastBucket the Kademlia bucket relative to peerID
//          using the command ipfs dht query
// Argument the peerID, the dimension of the maximum returned array
// Returns an error if the command wasn't executed, nil otherwise
func (i *Ipfs) GetBucket(peerID string, dim int) ([]string, error) {
  // Initializing the exec structures for the ipfs dht query command
  cmd := exec.Command("ipfs", "dht", "query", peerID)
  // Redirecting the output on the out Buffer
  var out bytes.Buffer
  cmd.Stdout = &out
  // Starting the command and waiting for the process termination by timeout
      or normal ending
  if err := cmd.Start(); err != nil {
    log.Fatal(err)
  }
  done := make(chan error, 1)
  go func() {
        done <- cmd.Wait()
    }()
    select {
    case <-time.After(timeout):
        if err := cmd.Process.Kill(); err != nil {
            log.Fatal("failed to kill process: ", err)
        }
    case err := <-done:
        if err != nil {
            return nil, err
        }
    }

    // Dividing the output in a string array, one for every PeerID
    t := strings.Split(out.String(), "\n")

    var newBucket []string
    if len(t) < bucketSize {
        newBucket = t[:]
    } else {
        newBucket = t[:bucketSize]
    }

    // Statistic computation of the new bucket found
    // Checking
    //      1. number of new peers in the bucket
    //      2. number of old peers no more reachables
    i.bu.lastBucketNewPeers = 0
    i.bu.lastBucketNewPeersOffline = 0
    i.bu.lastBucketOffline = 0
OuterLoop:
    for _, v := range i.bu.lastBucket {
        if v == "" {
            continue
        }
        for _, k := range newBucket {
            if v == k {
                // This peer was in the old bucket
                i.bu.lastBucketNewPeers++
                /* continue OuterLoop */
                break
            }
        }
        // Peer v isn't in the new bucket, still reachable?
        // if the node isn't reachable function ID does not have a public ip
             address related
        a, err := i.sh.ID(v)
        i.bu.lastBucketNewPeersOffline++
        if err != nil || a == nil {
            continue
        }
        for _, ipfsAddr := range a.Addresses {
            if ipfsAddr == "" {
                continue
            }
            ipAddr, _ := trimIpfsAddress(ipfsAddr)
            netIPAddr := net.ParseIP(ipAddr)
            if netIPAddr != nil && !isPrivateIP(netIPAddr) {
                i.bu.lastBucketNewPeersOffline--
                continue OuterLoop
            }
        }
        i.bu.lastBucketOffline++
    }

    i.bu.lastBucketNewPeers = 20 - i.bu.lastBucketNewPeers

    // Copy the new bucket in the structure and return
    copy(i.bu.lastBucket[:], newBucket)
    return t, nil
}
```

Listing 4: ping.go

```
package main

// This module implements the RTT check for the queried IPs
// See config.go for the default parameters like the number of pings
//
// Author Marco Costa

import (
    "fmt"
    "log"
    "net"
    "time"

    "github.com/sparrc/go-ping"
)

var privateIPBlocks []*net.IPNet
var isInit bool = false
var pingCache map[string]time.Duration

// Initializes the ping module used in a manner similar to a singleton
func initPing() {
    for _, cidr := range []string{
        "127.0.0.0/8",  // IPv4 loopback
        "10.0.0.0/8",     // RFC1918
        "172.16.0.0/12", // RFC1918
        "192.168.0.0/16", // RFC1918
        "169.254.0.0/16", // RFC3927 link-local
        "::1/128",        // IPv6 loopback
        "fe80::/10",      // IPv6 link-local
        "fc00::/7",       // IPv6 unique local addr
    } {
        _, block, err := net.ParseCIDR(cidr)
        if err != nil {
            panic(fmt.Errorf("parse error on %q: %v", cidr, err))
        }
        privateIPBlocks = append(privateIPBlocks, block)
    }
    if cachePings {
        pingCache = make(map[string]time.Duration)
    }
}

// Checks if an IP address is a private network IP or not
// Returns true if it is, false otherwise
func isPrivateIP(ip net.IP) bool {
    if !isInit {
        initPing()
        isInit = true
    }

    if ip.IsLoopback() || ip.IsLinkLocalUnicast() || ip.IsLinkLocalMulticast() {
        return true
    }
    for _, block := range privateIPBlocks {
        if block.Contains(ip) {
            return true
        }
    }
    return false
}

// getRTT returns the average RTT of the "ip" IP
// An error is returned if the ping didn't succeed
//
// NOTE: the script must be executed in privileged mode
//       otherwise duration 0 is returned
func getRTT(ip string) (time.Duration, error) {
    if !isInit {
        initPing()
        isInit = true
    }
    if cachePings {
      if rtt, present := pingCache[ip]; present {
         return rtt, nil
      }
   }

   p, err := ping.NewPinger(ip)
   if err != nil {
      return 0, err
   }
   p.SetPrivileged(true)
   if !p.Privileged() {
      log.Printf("Can't ping without privileged mode")
      return 0, nil
   }

   p.Count = pingIterations
   p.Timeout = pingTimeout
   p.Run()
   stat := p.Statistics()

   if cachePings {
      pingCache[ip] = stat.AvgRtt
   }

   return stat.AvgRtt, nil
}
```

Listing 5: dht.go

```
package main

// This module provides the script routine for bruteforcing the 256 PeerIDs
    relative to the Kademlia
// routing table of the local Peer
// IMPORTANT: this module hasn't been fully tested due to lack of
    computational power
//
// Author: Marco Costa

import (
   "crypto/sha256"
   "encoding/json"
   "fmt"
   "log"
   "math/rand"
   "time"

   "github.com/ipfs/go-cid"
)
```

```
const returnedBucket int = 500
const bucketTimeout time.Duration = 1 * time.Minute

// Computes the dXOR function of two [32]byte arrays as defined in the libp2p
    specs
func dXor(id1, id2 [32]byte) int {
   tot := 0

   for i := 0; i < len(id1); i++ {
      val := id1[i] ^ id2[i]
      t := 8
      for val != 0 {
         val = val >> 1
         t--
      }
      tot += t
      if t != 8 {
         return tot
      }
   }
   return tot
}

// Allowed characters for base58
var letters =
    []rune("123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz")

type dhtStruct struct {
   BucketIdentifiers [256]string // Found identifiers per bucket
   foundBucket      int         // Found buckets
   peerIDHash       [32]byte    // Hash of the local peer
   voidCycles       int         // Test since last match
}

// This function calculates the Hash and the dXOR of the generated ID if
// the ID is valid, if it is and it covers a new bucket it's added to
// the struct and all its bucket is tested recursively
func (d *dhtStruct) testString(s []byte, i *Ipfs) {
   d.voidCycles++
   //_, err := i.sh.ID(string(s))
   _, err := cid.Decode(string(s))
   if err != nil { // The ID is not valid
      return
   }
   sHash := sha256.Sum256(s)
   t := dXor(d.peerIDHash, sHash)
   if t >= 256 {
      return
   }
   // Checks if the relative bucket has already a PeerID, otherwise
   // if it's a valid PeerID it executes ipfs dht query recursively
   if d.BucketIdentifiers[t] == "" {
      buck, err := i.GetBucket(string(s), returnedBucket)
      if err == nil {
         d.voidCycles = 0
         d.BucketIdentifiers[t] = string(s)
         d.foundBucket++
         fmt.Printf("#%v/%v -> %v\n", t, (256 - d.foundBucket), string(s))

         for _, v := range buck {
            if v != "" {
               d.testString([]byte(v), i)
            }
         }
      }
   }
}

// Routine for finding the identifiers, stops when they all are found
func (d *dhtStruct) findIdentifiers(i *Ipfs) {
   d.foundBucket = 0
   selfIDBytes := []byte(i.selfID)
   d.peerIDHash = sha256.Sum256(selfIDBytes)

   // Start with the swarm peer know
   i.GetSwarmPeers()
   for _, v := range i.sw.lastSwarm {
      d.testString([]byte(v.Peer), i)
   }

   newID := make([]byte, len(selfIDBytes))
   copy(newID, selfIDBytes)
   rand.Seed(time.Now().UnixNano())
   for d.foundBucket != 256 {
      // Generates a random ID
      for j := 2; j < len(selfIDBytes); j++ {
         if rand.Intn(2) == 1 { // Toss a coin
            newID[j] = byte(letters[rand.Intn(len(letters))])
            d.testString(newID, i)
            if d.voidCycles > 1000 {
               copy(newID, selfIDBytes)
               d.voidCycles = 0
            }
         }
      }
   }
}
```

```go
// Initializes the data and starts the procedure
func dht() {
   var i Ipfs
   i.InitIPFS()
   var d dhtStruct

   timeout = bucketTimeout
   d.findIdentifiers(&i)

   json, err := json.MarshalIndent(d.BucketIdentifiers, "", " ")
   if err != nil {
      log.Fatal(err)
   }
   writeDataFile(i.selfID+".json", json)
   fmt.Printf("PeerIDs printed in %v.json, exiting\n", i.selfID)
}
```