A Kademlia module for Peersim

Project of the course of *Distributed Systems*

M. Bonani D. Furlan

Academic Year 2009/2010

Abstract

In this work we present an implementation of a Kademlia module for the Peersim simulator.

1 Introduction

In this paper, we explain the implementation of a Kademlia module for Peersim [1].

Peersim is a simulator environment for P2P network topologies, designed to deal with protocol evaluation in challenging scenarios with high number of nodes, which typically join and leave continuously. The simulator allows the evaluation of a protocol distributed over millions of nodes, in each stage of the development process.

Peersim is written in Java and it's released under the GPL open source license. It offers two engines for the simulation: a simplified (cycle-based) model and the event-driven one. The extreme scalability, the high expansiveness with pluggable components and the flexible configuration mechanism allows Peersim to be the perfect tool for evaluating P2P protocols.

1.1 Plan of the report

This document is composed of 5 sections. In Section 2 we present a brief overview of the Kademlia protocol, analyzing the main features and the general functioning. In Section 3 we explain the implementation of the Kademlia module in Peersim, analyzing the code and the choices made in order to

[1] http://peersim.sourceforge.net/

perform simulations as close as possible to real scenarios. In Section 4 we present the actual simulations, presenting the configuration used and the different cases analyzed. In the section 5 we present the results of the simulations. Finally in Section 6 we present the conclusions of our work and the possible development in the future.

2 Kademlia Protocol

2.1 Introduction

Kademlia is a *distributed hash table (DHT)* for peer-to-peer computer systems designed by Petar Maymounkov and David Mazières [1]. Kademlia takes further the functionalities offered previously by other DHT implementations, like CHORD[2], offering a flexible routing table and minimizing the number of configuration messages sent among the nodes.

The participant nodes are identified by IDs, which usually are 160-bits long random numbers. The IDs are used not only to identify uniquely a node, but also to locate the resources (usually in the form of file hashes or keywords). In fact the node ID provides a direct map to file hashes and that node stores information on where the specific resource is located. Kademlia treats nodes as leaves in a binary tree, with each node's position determined by the shortest unique prefix of its ID.

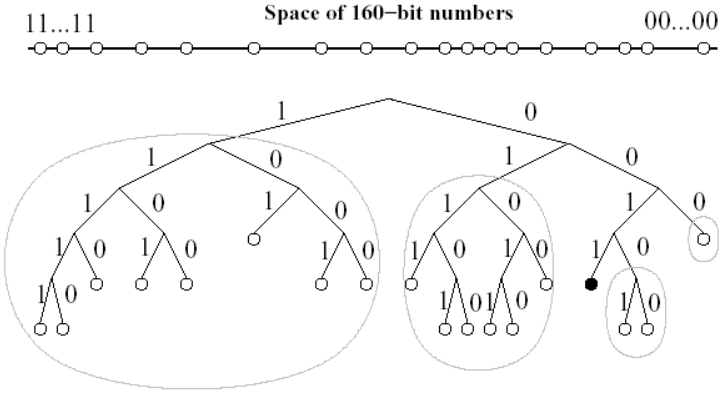

Figure 1: *Network partition for node 110. Grey ovals show the subtrees corresponding to different k-buckets*

To identify the resources in the network the DHTs store a couple <key, value> for each resource. The key is usually a 160-bit identifier and is computed from the value using an hash function. In this way any participating

node can efficiently retrieve the value associated with a given key. The <key, value> pairs are stored on nodes with IDs close to the key. When searching for some specific value, the algorithm needs to know the associated key and explores the network in several steps. Every step will find nodes closer to the key until the contacted nodes returns the value or no more closer nodes are found.

The Kademlia protocol ensures that every node knows of at least one node in each of its subtrees, if that subtree contains a node. With this information it's clear that a node can locate any other node by its ID. To locate nodes near a particular ID, Kademlia uses a single algorithm instead of two, like in other systems.

The details of the protocols are unveiled in the next subsections.

2.2 Distance Calculation

The particularity of Kademlia is the use of a novel XOR metric in order to calculate the distance between points (the node IDs) in the key space. Given two 160-bit identifiers, x and y, Kademlia defines the distance between them as their bitwise Exclusive Or (XOR) interpreted as an integer,

$$d(x, y) = x \oplus y$$

Exclusive Or was chosen because it presents some common properties with the geometric distance formula, in particular:

- the distance between a node and itself is zero.
- it is symmetric: the distance from x and y is the same as the distance from y to x.
- it offers the *triangle property*: given three points x, y and z, the distance from x to z is less than or equal to the sum of the distance from x to y and the distance from y to z.

Each cycle of iteration comes one bit closer to the target. A basic Kademlia network with 2^n nodes will take, in the worst case, n steps to find that node.

Exclusive Or permits to use the notion of distance implicit in the binary-tree representation of the network. In a fully-populated tree of 160-bit IDs, the magnitude of the distance between two IDs is the height of the smallest subtree containing them. When a tree is not fully populated, the closest leaf to an ID x is the leaf whose ID shares the longest common prefix of x.

2.3 Routing tables

Every node maintains a routing table which consists of a set of lists, one for each bit of the node ID (e.g. nodes IDs with 160-bit identifiers will keep 160 such lists). Typically a list has many entries, each one defined by a triplet <IP address, UDP port, Node ID> which specifies the necessary information to locate a node. Each list is kept sorted by time last seen. These lists are called *k-buckets*. The value k is called *system-wide replication parameter* and defines the upper bound to which every list can grow.

When a node receives a message from another node, it updates the specific k-bucket for the sender's node ID. If the sending node already exists in the recipient's k-bucket, the recipient moves it to the tail of the list. If the node is not already in the appropriate k-bucket and the bucket has fewer than k entries, then the recipient just inserts the new sender at the tail of the list. If the appropriate k-bucket is full, however, the recipient pings the k-bucket's least-recently seen node to decide what to do. If the least-recently seen node fails to respond, it is removed from the k-bucket and the new sender is inserted at the tail. Otherwise, if the least-recently seen node responds, it is moved to the tail of the list, and the new sender's contact is discarded.

Kademlia tries to exploit the documented [3] fact that the longer a node has been up, the higher the probability to remain up in the future. By keeping in the routing table the oldest live contacts, each k-bucket maximize the probability that the nodes they contain will remain connected.

2.4 Messages

Every message a node transmits includes its node ID, permitting the recipient to record the sender's existence, if necessary.

Kademlia provides four messages (or RPCs):

- PING used to verify if a node is up
- STORE stores a <key, value> pair in one node
- FIND_NODE takes a 160-bit ID as an argument. The recipient returns the <IP address, UDP port, Node ID> triplets for the k nodes it knows about closest to the target ID.
- FIND_VALUE works in the same way as FIND_NODE, but if the recipient of the request has the requested key in its store, it will return the corresponding value.

Kademlia provides some mechanisms in order to assure the sender's network address, but they are not considered in this paper.

2.5 Locating Nodes

The most important procedure of the Kademlia protocol is the location of the k closest nodes to some given node ID. In the Kademlia literature this procedure is called *node lookup*. The initiator picks α nodes from its closest non-empty k-bucket. Then the initiator sends parallel, asynchronous FIND_NODE messages to the α nodes it has chosen. The value of α, which defines the quantity of simultaneous lookups, is typically three. When these recipient nodes receive the request, they will look in their k-buckets and will return the k closest nodes to the desired key that they know. The initiator will update a result list with the IDs of the nodes he receives, keeping the k best ones (obviously the nodes that are closed to the searched key) that respond to the queries. The algorithm will iterate again on the k best results, issuing the request to them, until no nodes are returned that are closer than the previous result. The best k nodes are the ones in the network that are the closest to the desired key.

2.6 Join the network

To join a network, a node n must have a contact with an already participating node. The bootstrap process explicitly needs to know the IP address and the port of the node that is already part of the Kademlia network. The joining node inserts the bootstrap node into one of its k-buckets. The new node then does a NODE_LOOKUP of his own ID against the only other node it knows. This process (called simply "self-lookup") will populate the k-buckets of the other nodes with the new node ID, and at the same time it will populate the new node's k-buckets with the nodes in the path between it and the bootstrap node.

Finally the new node refreshes all the k-buckets further away than its closest neighbor. This refresh is just a lookup of a random key that is within that k-bucket range.

3 Implementation

The implementation of the Kademlia module in Peersim follows the event-driven model. The module implemented is slightly different than the standard Kademlia protocol. In particular we have skipped the implementation of the search and store of values, focusing on the operation of finding a node in the network that is at the basis of all the other operations. The events that drive the protocol are substantially the exchange of messages plus some

other timeout events that handle the failure of nodes in the network. In this session we will describe in detail the messages and how the protocol works.

3.1 Messages

The events associated to Peersim are the messages exchanged in the Kademlia protocol. Every message in the implementation is an instance of the class `Message` which extends the `SimpleEvent` peersim class.
The messages are:

- `MSG_FINDNODE` the message used to start a find node operation.
- `MSG_ROUTE` the message sent to the neighbors to query about a node to find.
- `MSG_RESPONSE` the type of message used to respond to a `MSG_ROUTE`. It contains the K closest node to the destination peer.

3.2 Code comments

3.2.1 Bootstrap process

In addition to the class `WireKOut` contained in the Peersim core, two classes has been implemented to provide network's initialization and bootstrap: `CustomDistribution` and `StateBuilder`.
At the beginning the `WireKOut` class builds randomly the interconnections between nodes creating a virtual network overlay.
Then the whole network is initialized by the class `CustomDistribution`, assigning to every node a unique ID, randomly generated in the space $0..2^{BITS}$, where BITS is a parameter from the kademlia protocol (usually 160). The BITS parameter, as we will see, is configurable in the configuration file.

Finally the `StateBuilder` class performs the node's bootstrap process filling the k-buckets of all initial nodes. In particular the initial bootstrap process has been implemented adding, to each peer, the IDs of 50 random nodes and its 50 nearest nodes. This naïve procedure permits to fill the various routing tables in a simple manner, granting to every peer the knowledge of its neighbors and some other random nodes, distributed among the network.

3.2.2 Find a node

The process of finding a node is initiated upon the receipt of a `MSG_FINDNODE` message (i.e. generated by the class `TrafficGenerator`). The first opera-

tion performed is the creation of a `FindOperation` object that contains all the information connected to the specific search operation. In particular it contains the closest neighbor set as well as the information about the nodes queried and the outstanding requests. This class also contains the variables `timestamp` and `nrHops` that counts respectively the duration and the total number of hops of the operation.

As described before, at the beginning the K closest nodes are retrieved from the node k-buckets. Now the ALPHA nearest nodes to the destination are queried in parallel sending them a `MSG_ROUTE` message. When a response having type `MSG_RESPONSE` arrives, the nearest nodes in the `FindOperation` object are updated and, if there are nodes not already queried, a new route request can be send. It's important to remark that it's not necessary to wait all the ALPHA responses before proceed to query other nodes. The important issue is that the maximum number of parallel requests is ALPHA.

When no more nodes are available to query and there aren't outstanding request the operation ends.

3.2.3 Joining the network

The joining on the network works almost in the same way described before in the paper. A new Peersim node is created and assigned to a new ID. Initially his routing table is empty, so an existing random node is added to its k-buckets. The node proceeds with a search operation of itself, sending a `MSG_ROUTE` message to the random bootstrap node. The new node also performs another random search to enrich its routing table.

4 Simulations using the module

The Kademlia module is customizable through a simple-text configuration file. This file allows to define the parameters of the whole network and to set the various Controls. In this section we will describe briefly the various components of the module, explaining its characteristics.

4.1 The protocol

The Kademlia protocol can be customized through the following parameters:

- BITS specifies the length of the ID (default is 160)
- K called also *system-wide replication parameter*, defines the dimension of k-buckets (default is 20).

- ALPHA is the number of simultaneous lookup actions allowed by the protocol (default is 3).

The module, with the default configuration, relies over two transport layers, which defines how messages can be sent among the nodes of the network: the `UniformRandomTransport` layer reliably delivers messages with a random delay (which minimum and maximum values are defined in the configuration file), while the `UnreliableTransport` class simulates message losses, based on the configured probability.

4.2 The controls

The module provides two different Controls classes: `TrafficGenerator` and `Turbolence`. The first one generates find node messages from random sources with a random IDs to search. The second, according to a given probability, simulates the addition or the removal (failure) of a node. The probabilities are configurable from the parameters:

- `p_idle` (default = 0): probability that the current execution does nothing (i.e. no adding and no failures).
- `p_add` (default = 0.5): probability that a new node is added in this execution.
- `p_rem` (default = 0.5): probability that this execution will result in a failure of an existing node.

The two controls act according with a configurable frequency given by the STEP parameter.

4.3 The observer

The module provides also an observer (called `KademliaObserver`). This class is used to gather simple statistics about the behavior of the protocol. It keeps track of the number of hops, the time and the number of messages delivered, together with the number of find operations performed by the protocol.

The only allowed parameter is STEP, which defines the frequency of the output of the measure on the *stderr*.

4.4 Compiling the module

The module, together with the Peersim library, can be compiled using the provided Makefile (located in the main directory kademlia/ of the archive).

However, a manual compilation of the software can be performed with the following steps: execute the command:

```
javac -g \ -classpath .:jep-2.3.0.jar:djep-1.0.0.jar
\'find . -name "*.java"'
```

then execute:

```
rm -f peersim-1.0.jar
```

to remove the old jar archive and then run the script:

```
./make jar
```

to create a new jar distribution of Peersim.

4.5 Running the simulation

```
java -Xmx500M -cp \"peersim-1.0.jar:jep-2.3.0.jar:
djep-1.0.0.jar" \peersim.Simulator example.cfg
```

The parameter `-Xmx500M` is used to increase the amount of memory dedicated to the simulation (i.e. 500 Mb are sufficient for a network with 25000 nodes).

5 Results

In this section we present some simulation's results obtained with the Kademlia module. We have analyzed the performance in doing search operations, changing the network size and analyzing the behavior in presence of churn. In particular we will report the average number of messages exchanged in each operation and the average duration of the operation. In the kademlia paper is presented a very qualitative value regarding the expected execution time for most operation which is:
$\lceil log\, n \rceil + c$
for a small constant c.

In the graphs, apart from the obtained values, it's also indicated the theoretical behavior together with the actual value of the constant c used for the plotting.

5.1 Simulation parameters

We have chosen to use a reliable channel (indeed an `UnreliableTransport` with $drop = 0$ on a `UniformRandomTransport` with 100 ms path delay) as network model, in order to simulate a connection oriented transport protocol like TCP.

The simulation has been done varying the network size from 128 to 65536 nodes, doubling the number of nodes at each round. A round lasts 1 hour (total simulation time). The number of search operations depends on the number of nodes. In practice, on average, every node starts an equal number of find operations during the simulation. In particular in the example configuration file the traffic step is:

```
TRAFFIC_STEP (SIM_TIME)/SIZE
```

That corresponds to a find operation per node.

The turbulence step is configured in almost the same manner to have around 5% of node turnover during the simulation:

```
TURBULENCE_STEP (SIM_TIME*20)/SIZE
```

5.2 Node search performance in a stable environment

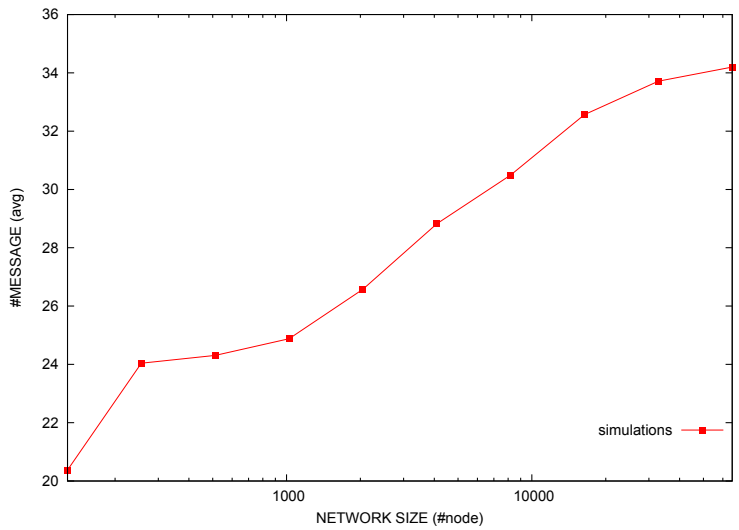

Figure 2: *Number of messages per search operation compared to network size in absence of churn*

The first test has been done using the basic configuration described above. The turbolence class in this first simulation is not used.

In the picture 2 is presented the average number of messages per search operation and the average duration of the operation in milliseconds.

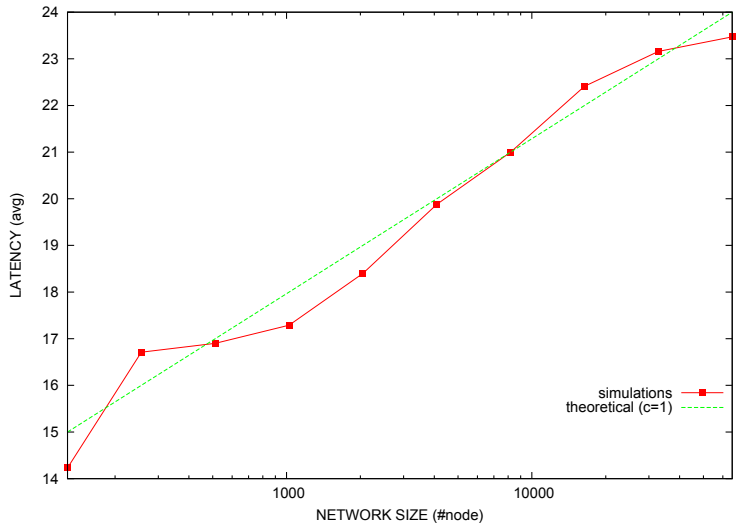

Figure 3: *Duration of search operation with respect to network size in presence of churn. The theoretical line corresponds to* $\lceil \log n \rceil + c$ *with* n *the number of nodes and* $c = 1$

From the picture can be evinced that the implemented protocol shows performance similar to the theoretical one. The more representative measures although are those with a fairly high number of nodes because Kademlia is suited for network with up to millions of nodes.

5.3 Node search performance in presence of churn

The simulation with churn is almost identical to the previous one. The only difference is the presence of the `Turbulence` control activated. The turbulence control is tuned with equal probability of adding and removing nodes. The idle probability has been set to 0.

In the picture 4 is presented, like previously, the number of messages exchanged in each search operation and the average duration of the operation.

From the graphs can be evinced that Kademlia support churn very well. In fact the performance are only a little bit worse than without churn (the constant c has been increased by 1). This fact derives from the number of nodes memorized in the k-buckets which is very high.

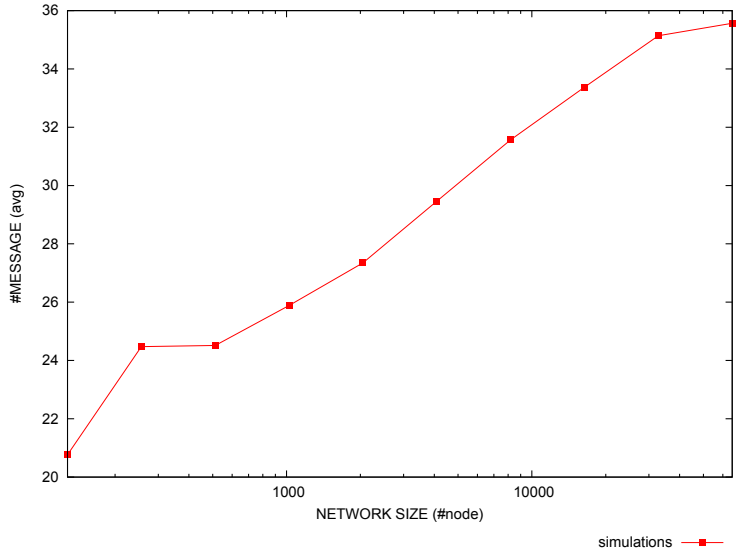

Figure 4: *Number of message per search operation with respect to network size in presence of churn*

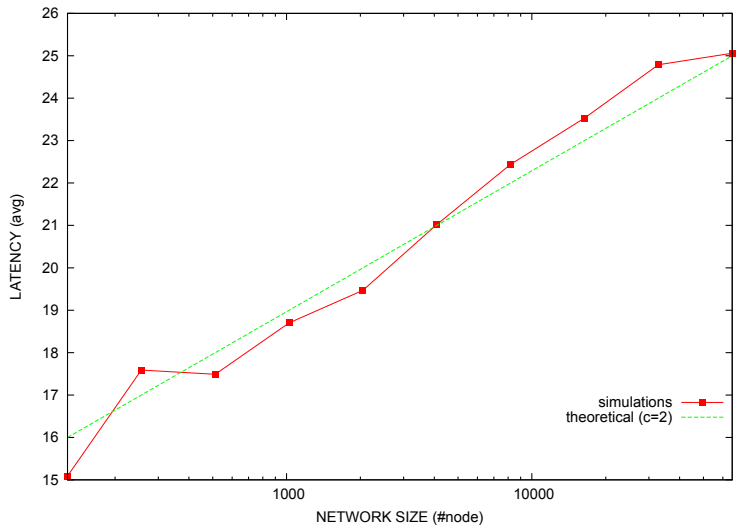

Figure 5: *Duration of search operation with respect to network size in presence of churn. The theoretical value is* $\lceil \log n \rceil + c$ *with* n *the number of peer in the network and* $c = 2$

6 Conclusions and Future Work

In this paper we have initially described how the Kademlia protocol works in theory. Then we have provided a description of the implementation of a basic Kademlia module for Peersim, presenting its functioning and the possible existing configurations that can be used to customize the module. Finally we have made some simple simulations analyzing the results to verify the correctness of our implementation with respect to the theoretical results.

Due to the simplicity of the module, further work can be moved in the direction of completing the module itself, adding the item store and search processes and maybe implementing some optimization described in the paper. It will be also interesting to develop a new turbulence class that takes into account the time that a node was up before deciding which node to kill. With this new module it would be possible to do some comparisons between Kademlia and other DHT protocols in situations with non-uniform probability of peer faults.

References

[1] Petar Maymounkov and David Maziï¿½res. Kademlia: A peer-to-peer information system based on the xor metric. pages 53–65, 2002.

[2] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 149–160, San Diego, CA, 2001. ACM, ACM Press.

[3] Stefan Saroiu P. Krishna Gummadi Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. Technical report, UW-CSE-01-06-02, University of Washington, Department of Computer Science and Engineering, July 2001.