CYBERNETICA
Institute of Information Security

Kademlia-based distributed hash tables implementation for VirtualLife

Jaak Ristioja

T-4-7 / 2009

Jaak Ristioja[1].
[1] AS Cybernetica, Institute of Information Security

The research reported here was supported by:

1. Estonian Science foundation, grant(s) No. 6944,
2. the target funded theme SF0012708s06 “Theoretical and Practical Security of Heterogenous Information Systems”,
3. the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS, and the Software Technology and Applications Competence Centre, STACC,
4. EU FP7-ICT project VirtualLife (contract no. 216064).

Cybernetica research reports are available online at
`http://research.cyber.ee/`
Mailing address:
AS Cybernetica
Akadeemia tee 21
12618 Tallinn
Estonia

Kademlia-based distributed hash tables implementation for VirtualLife

Jaak Ristioja

January 25, 2010

1 Motivation

DHT or Distributed Hash Table[1] is a decentralized distributed system, that provides a lookup service. In its essence, it's similar to a hash table, as it stores key-value pairs as its content, allowing clients to lookup values, using a key. The content is distributed among all participating nodes in a way that causes minimal disruption from node additions, removals, failures etc.

VirtualLife is a 3D virtual world running on several servers (zone servers), to which clients connect. The virtual world spans over all the zone servers, i.e. each zone server handles objects in a particular area in the virtual world. The client application connects to a single zone server at the time. It retrieves all required objects from that zone server. Given this design, the need for the client to access objects from other zone servers arises.

For example each identified client might have an inventory which contains objects located in different zones of the virtual world. The client can still only connect to one zone server at a time, but still needs to access objects in his inventory from different zones. Since the client only retrieves objects from the zone server it is currently connected to, that zone server has to provide those objects. To do so, the zone server first needs to know from which zone server to retrieve an object, then retrieve that object and pass it to the client.

Since all the zone servers in the virtual world are connected to the same nation server, the easiest way would be for the zone servers to query the nation server for the locations of such objects. This requires the nation server to provide an indexing service for storing the locations of these objects. However, putting too much load on the single nation server will be inefficient. Therefore a more distributed means for providing such index is required.

[1] http://en.wikipedia.org/wiki/Distributed_hash_table

The objects are stored on the zone servers, hence likely the best solution to this problem is for the zone servers themselves to provide some kind of distributed indexing service. For this purpose, a distributed hash tables (DHT) solution called Kademlia[2] was implemented for the zone servers in VirtualLife. For indexing purposes, Kademlia provides both fault-tolerance and performance. When several zone servers participating in the DHT go offline, this will not cause a serious data loss in the index itself, since the index data is distributed on multiple zone servers. For the same reason lookups from the index do not require many messages to be sent between the zone servers, resulting in fast lookups.

[2] http://en.wikipedia.org/wiki/Kademlia

2 Summary of DHT in VirtualLife

The foremost purpose of the DHT in VirtualLife is to provide a mapping of asset GUID's to their location in the system, i.e. it stores pairs of (GUID, GUID), since the locations where assets are stored also have a GUID. DHT does NOT provide any means to retrieve the assets from their location.

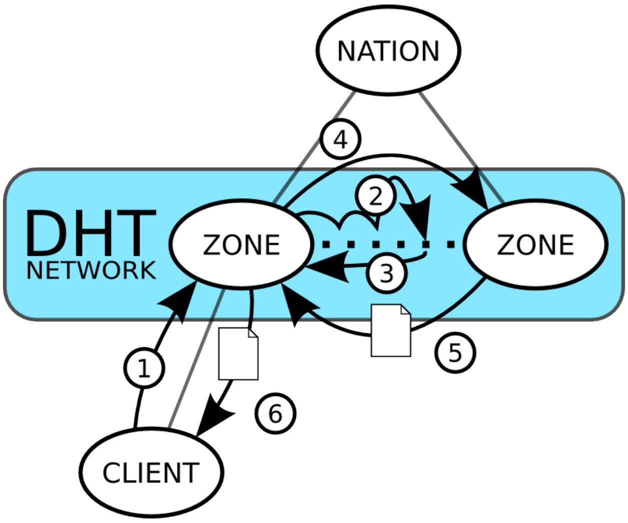

Figure 1: Example of handling an asset request by a client. DHT is used in 2-3.

1. Client requests an asset from a zone.
2. The zone queries the DHT network.
3. The location of the asset is returned by the DHT query.
4. The zone requests the asset from the location returned by the DHT query.
5. A copy of the asset is returned to the zone.
6. The zone returns a copy of the asset to the client.

If a client asks for an asset from a zone by the GUID and the zone does not possess it, nor have it cached, it uses the DHT to find the location of the asset (see Figure **??**). The zone does this by querying the DHT using the GUID given by the client. In return, the DHT query either returns the location of the asset, or a failure. If the DHT query succeeds and returns the location of the asset, the zone attempts to retrieve the asset from that location. In case the asset cannot be retrieved in an acceptable manner (e.g. location does not respond or possess the asset any more), then the asset is found to be unavailable. On successful retrieval, the asset is forwarded to the client, and may optionally be cached in the zone. In case the DHT query returns a failure, or if the asset can not be retrieved from the location received from the DHT query, a failure is returned to the client.

2.1 Participants

The nodes of the DHT consist only of Virtual Zones. The DHT network itself is kept private (clients will have no direct access), as it's only directly used by the nodes (the Virtual Zones) themselves.

2.2 Storage

The DHT is used to store the locations of various assets, and not the assets themselves. This means that in case the DHT contains the location for a given GUID, the asset may still be unavailable from that location (e.g. network failure). As each location is also identified by a GUID, we store (GUID-GUID) pairs.

The contents of the DHT (GUID-GUID pairs) will be distributed among all participating nodes, whereas multiple nodes may contain the unique GUID-GUID pairs (to counter node failures and provide faster lookup). The current implementation tries to stores any pair on at least 10 (parameter DHT_K, see Kademlia paper) participating nodes. In case of many DHT nodes, the more popular a GUID-GUID pair is (i.e. frequent enough lookups for that pair key are done), the more nodes that pair will be stored on.

2.3 Types assets in the DHT

Since the DHT will be able to store any GUID→GUID mappings, it does not actually matter what you put into the DHT. You could, in theory, store whatever key-value pairs of type GUID-GUID you like. So from the viewpoint of developing the DHT, we actually won't have to consider what abstract information those pairs hold. For simplicity, in the current DHT API implementation you are only allowed to issue a store request on a DHT node where in the key-value pair the value is the GUID of the same node.

3 DHT description

The DHT implementation is an overlay network running on top of the message system, using the Kademlia[3] protocol. The nodes participating in the DHT overlay are identified by the GUID of their entity manager. That GUID will also be used when addressing messages.

3.1 Differences from the Kademlia paper

To better suit the DHT needs of VirtualLife, the Kademlia protocol was not implemented in the exact manner described in the Kademlia paper. This section describes the most important aspects of Kademlia that were implemented with changes.

3.1.1 Node distance metric

We have slightly modified the XOR metric described in the Kademlia paper. According to Kademlia the distance between 2 nodes with the GUID's x and y would be $d(x, y) = x \oplus y$, whereas in the DHT implementation in VirtualLife the distance is actually $d(x, y) = 128 - f(x \oplus y)$, where 128 is the number of bits in the GUID, and the function f returns the index (starting from 0) of the first set bit (1) in its argument or 128 if no bit is set. In comparison to the distance metric used in Kademlia, our distance metric is less accurate, but fulfills our DHT needs and fits nicely into a CPU register, since $0 \leq d(x, y) \leq 128$. Our distance metric can immediately be used as an index of a k-bucket, without requireing any conversion.

Just as Kademlia's XOR metric, it satisfies the following properties:

- $d(x, x) = 0$
- $d(x, y) > 0$ if $x \neq y$
- $\forall x, y : d(x, y) = d(y, x)$
- $\forall x, y, z : d(x, y) + d(y, z) \geq d(x, z)$

[3] `http://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf`

3.1.2 System-wide parameters

Here's a descriptive list of numeric parameters present in the Kademlia paper and their counterparts in the VirtualLife DHT implementation.

Value in Kademlia	**Value in VirtualLife**	**Description**
160 bits	128 bits	length of ID's / GUID's.
$k = 20$	DHT_K = 10	system-wide replication parameter.
$\alpha = 3$	DHT_A = 3	system-wide concurrency parameter.
$24h$	DHT_VALUE_TIMEOUT = $24h$	key-value pair expiration time.
$1h$	DHT_VALUE_REPUBLISH = $1h$	how often to republish key-value pairs.
$1h$	DHT_BUCKET_REFRESH = $1h$	if we have not done any lookups for the GUID range of a specific bucket for the specified time, we have to refresh this bucket.
5	DHT_STALE_COUNT = 5	number of failed requests to a peer to consider it stale.
not described	DHT_REQUEST_TIMEOUT = $2s$	timeout for a request.
assuming 1	DHT_PUSH_VALUES_THRESHOLD = 3	only store a value on a new peer if we are one of the DHT_PUSH_VALUES_THRESHOLD closest nodes to that value.

3.1.3 Routing table

Instead of having a dynamic tree for a routing table we just keep 128 buckets for simplicity. Since the number of participating DHT nodes is quite low, we also do not limit node data kept in memory.

3.1.4 Optimizations

We have not implemented the optimization of avoiding a node lookup before republishing keys. Nor have we implemented any optimizations regarding the routing table as a tree, since we don't keep it as a tree.

4 API

4.1 DHT subsystem initialization

4.1.1 DHT initialization in the zone server

The general steps to initialize DHT in the zone server are as follows:

1. Initialize the entity manager
2. Create a `ZoneDHTManager`
3. Create a `ZoneDHTComponent` and attach it to the message queue.
4. To start the bootstrapping process for DHT, pass the GUID of the nation to `ZoneDHTManager`

Listing 1: Example of initializing the DHT in the zone server

```
// Presuming the entity manager has already been set up, we
// initialize the DHT manager by passing it a reference to
// the network protocol:
ZoneDHTManager &DHTManager(*ZoneDHTManager::create());
DHTManager.initNetworkProtocol(protocol);

// Setup the DHT component and attach it to the message queue:
EntityManager &em(EntityManager::getSingleton());
ZoneDHTComponent *DHTComponent = new ZoneDHTComponent();
em.addComponent(*DHTComponent, EntityManager::PRIORITY_NORMAL);

// After all network connections have been setup we need to
// bootstrap the DHT:
ZoneDHTManager::getSingleton().init(nationGuid);

// At this point the DHT in the zone has started bootstrapping
// itself and will be ready for general use after it has finished
// bootstrapping.
```

4.1.2 DHT initialization in the nation server

To initialize DHT in the nation

1. Create a `NationDHTManager` singleton
2. Create a `NationDHTComponent` and attach it to the message queue of the nation's entity manager.

Listing 2: Example of initializing the DHT in the nation server

```
// Create the DHT manager:
NationDHTManager::create();

// Create the NationDHTComponent and attach it to the message
// queue:
EntityManager &em(EntityManager::getSingleton());
NationDHTComponent* pDht = new NationDHTComponent();
em.addComponent(*pDht, EntityManager::PRIORITY_LOW);

// Now the DHT in the nation will receive and answer DHT
// bootstrap queries from zones.
```

4.2 Other bindings

4.2.1 Zone bindings

When the zone receives a message from the nation (especially when its a DHT message), the message must explicitly be passed to `ZoneDHTManager`. This is because the nation messages are currently not forwarded to the general message loop, and will not be received by the `ZoneDHTComponent`.

Listing 3: Passing on messages from the nation

```
ZoneDHTManager::getSingleton().processNationMessage(message);
```

4.2.2 Nation bindings

When a zone goes offline for any reason, the nation should pass that zone's GUID to the DHT manager.

Listing 4: Example of notifying the DHT that a zone has left

```
NationDHTManager::getSingleton().zoneLeft(zoneGuid);
```

4.3 DHT Access

4.3.1 Publishing objects

Publishing a key means that the zone will tell DHT, that it has made the object in question available. That object is uniquely identified by its GUID, the key. Publishing an object therefore comes down to storing a key-value pair on DHT: the key is the GUID of the object in question, and the value is the GUID of the zone that provides the object.

WARNING! If any zone publishes a key that has already been published by another zone, the original value will mostly prevail, but it cannot be guaranteed that lookups for that key will always return the original value. However, it is possible to modify the DHT subsystem in the future so that it stores multiple GUID values per key GUID, i.e. GUID→[GUID] pairs.

Listing 5: Method for telling the DHT that an object is available from this zone, by passing the GUID of the object

```
void ZoneDHTManager::publish(const Guid &key);
```

This starts an asynchronous storage process. So this does not guarantee that the key-value pair was or will actually be stored properly (right away), because of the peer-to-peer nature of DHT.

Listing 6: Example of publishing an object

```
ZoneDHTManager &dm(ZoneDHTManager::getSingleton());
if (dm.getState() == ZoneDHTManager::RUNNING) {
    ZoneDHTManager::getSingleton().publish(objectGuid);
}
```

If a published key-value pair has not been (re)published within 24 hours, it will disappear from DHT. `ZoneDHTManager` will automatically republish all keys it has not (re)published within an hour. When the zone is restarted, the zone would need to republish all keys (the ones it is the original publisher of) manually by using `ZoneDHTManager::publish(const Guid &key)`, because `ZoneDHTManager` does not save it's state to disk. If the zone does not republish its keys, they will disappear from DHT after the 24 hours have elapsed.

4.3.2 Lookup

To asynchronously look up the location of an object on DHT, the `ZoneDHTManager` provides the following method:

Listing 7: Asynchronous lookup method for DHT

```
template <class F, class G>
bool ZoneDHTManager::lookup(const Guid &key,
                            const F &failureCallback,
                            const G &successCallback);
```

The lookup method returns true if the asynchronous lookup was properly initiated, and false otherwise, e.g. when the DHT has not yet been properly bootstrapped. One of the two callbacks provided is called respectively either when the lookup fails or succeeds.

Listing 8: Example of doing an asynchronous DHT lookup

```
class MyLookup {
    bool doLookup(const Guid &key) {
        return ZoneDHTManager::getSingleton().lookup(
            key,
            boost::bind(&MyLookup::onFailToFind, this, _1),
            boost::bind(&MyLookup::onFound, this, _1, _2)
        );
    }
    void onFailToFind(const Guid &key);
    void onFound(const Guid &key, const Guid &value);
}
```

5 DHT internals

The DHT implementation consists of mainly 2 modules for both the Nation and the Zone: the DHT manager singleton (classes ZoneDHTManager and NationDHTManager), and the DHT Component (classes ZoneDHTComponent and NationDHTComponent), which is attached to the message queue of the zone's (or nation's) entity manager. DHT messages are received from the entity managers message loop via the DHT Component (except for messages in the zone received from the nation, which are directly passed to the ZoneDHTManager from ZoneNetComponent). For sending messages, the NetworkManager singleton is used directly.

5.1 DHT in the nation

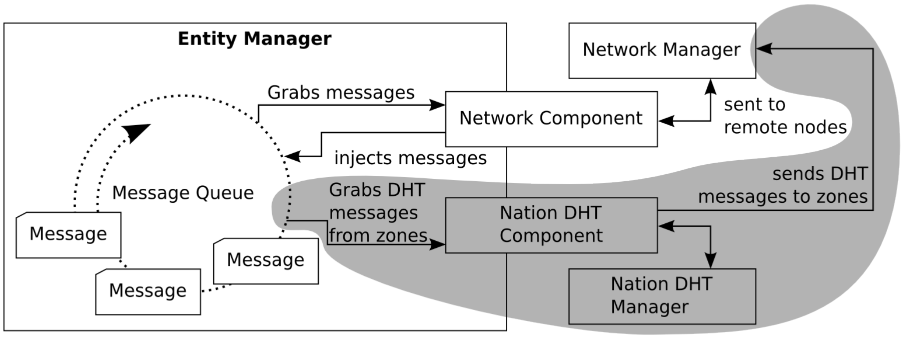

Figure 2: DHT in the nation

5.1.1 NationDHTComponent

The nation DHT component handles DHT bootstrap messages (i.e. the GET_NODES request) from DHT nodes, and replies with a list (gotten from NationDHTManager) of other DHT nodes that are already online. It also notifies the NationDHTManager when a nodes comes online.

5.1.2 NationDHTManager

The only role of the nation DHT manager is to keep a list of DHT nodes that are currently online (i.e. connected to the nation). A DHT node is considered being online by the nation when the nation receives a GET_NODES request from the DHT node when the latter bootstraps itself. For a reply to the GET_NODES request, the nation DHT manager returns a list of DHT nodes "closest" to the bootstrapping node.

5.2 DHT in the zone

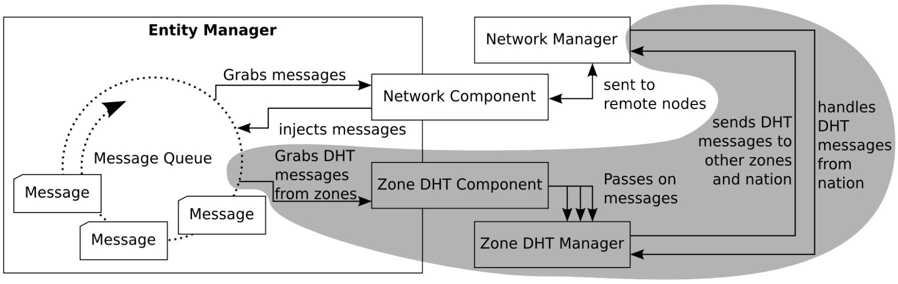

Figure 3: DHT in the zone

5.2.1 ZoneDHTComponent

The zone DHT component is the first to handle messages received from the entity manager. Based on the message type, it takes the message components apart from their container, and passes them to the respective ZoneDHTManager method.

5.2.2 ZoneDHTManager

The zone DHT manager is the central component of the DHT subsystem. It provides storage for DHT key-value pairs, keeps track of other DHT nodes, handles most communication between with other DHT nodes etc.

Value storage Values stored in DHT are stored in ZoneDHTManager, in a map storing key-value pairs where the key is the GUID of the object, and value is a structure containing the value and other mostly DHT-specific metadata. Any key-value pairs that expire 24 hours after being republished by the original publisher, are removed from the node. Any key-value pairs that need to be republished will be republished.

Peer storage For keeping track of other DHT nodes, the ZoneDHTManager keeps a mapping of DHT node's GUID's to pointers to structures containing node metadata. ZoneDHTManager also contains a distance table (or a routing table), where pointers to node metadata are kept in a specific sorted order in buckets in order to optimize DHT-specific communication between nodes.

Pending activities To keep track of all ongoing asynchronous operations (such as RPC's etc), the zone DHT manager keeps a list of these operations. Each such pending activity inherits from the class `Activity`, and therefore has an unique ID. Whenever such an operation is added to the list, it is activated via the `start()` method. DHT requests sent by these activities may also carry the ID if the activity, the request ID. Whenever a message with such ID is received by the DHT, it is first parsed by the zone DHT manager, and relevant information in the message is forwarded to the activity expecting the message. Activities may also choose to time out when their onTick() method is called, and remove themselves from the list of pending activities.

The class `Activity` has the following subclasses:

- `Bootstrap`
    1. Sends a `GET_NODES` request to the nation to retrieve GUID's of some other DHT nodes.
    2. Performs a node lookup (using `NodeLookup`) on this node's GUID.
- `Lookup`, a base class inherited by `NodeLookup` and `ValueLookup`
    - `NodeLookup`
        1. Performs a node lookup for the given key.
        2. Returns a list of nodes.
    - `ValueLookup`
        1. Performs a value lookup for the given key.
        2. Returns the found value or failure.
- `Refresh`
    1. Generates a random GUID in the range of the given bucket.
    2. Does a node lookup (using `NodeLookup`) on that GUID.
- `Request`, a base class inherited by `FindNodeRequest` and `FindValueRequest`
    - `FindNodeRequest`
        1. Does a `FIND_NODE` request on the given node.
        2. Returns a list of nodes or failure.

  - `FindValueRequest`
    1. Does a `FIND_VALUE` request on the given node.
    2. Returns the found value, a list of nodes or failure.

- `Store`
  1. Does a node lookup (using `NodeLookup`) on the given key.
  2. Sends a `STORE` messages to all the nodes returned by that lookup to store the given value.

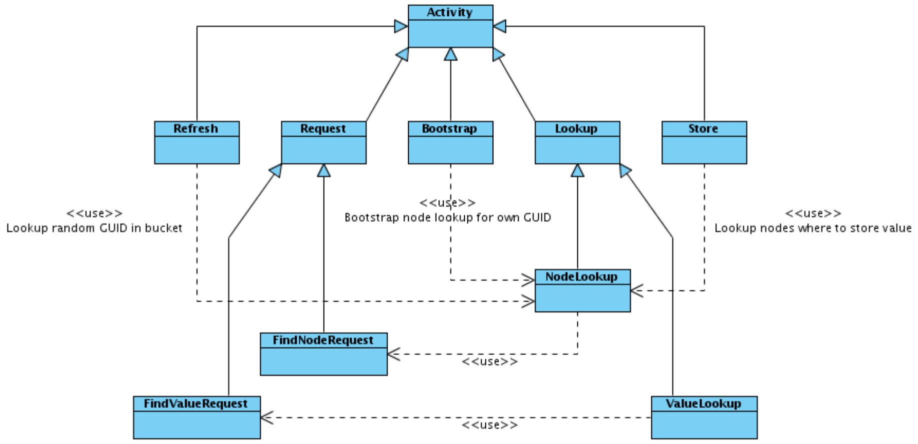

Figure 4: Dependancies between DHT activity classes.

6 DHT Monitor

In addition to implementing DHT in VirtualLife, an interactive tool with a graphical user interface was implemented for monitoring DHT activity in VirtualLife zone servers. Its was written mostly for testing purposes. It also allows to perform certain tests using the value injection and lookup methods provided.

The DHT monitor is a client, which connects to the running DHT nodes. The DHT nodes send the DHT monitor messages about their changing states and the DHT monitor displays these in a human-readable way. The DHT monitor allows to publish (*inject*) GUID's on a connected DHT node, and observe these being propagated to other DHT nodes. It is also possible to perform lookup queries for the keys that were previously injected.