MaidSafe Distributed Hash Table

David Irvine, email: david.irvine@maidsafe.net.

September, 2010 ***Abstract*—An effective distributed network requires an addressing mechanism and distribution of data that is designed to overcome churn (the unmanageable outage, node failure or unforeseen communication fault). There are several implementations of such a system, but this paper proposes a system that not only provides an efficient churn-resistant DHT, but also has fast awareness of infrastructure changes. This allows the implementation of a locking system, thereby enabling amendment of data from multiple sources.**

***Index Terms*—security, freedom, privacy, DHT, encryption**

CONTENTS

I. INTRODUCTION

Distributed hash tables are a method of storing data in a $key/value$ system in a distribution that should be set to allow for multiple node failures. This particular DHT implementation is based upon Kademlia as described in *Kademlia: A Peer-to-peer Information System Based on the XOR Metric* [5].

Data validity is achieved in Kadmelia by selecting a value for the replication factor (κ) that is the number defined as the optimum node count in a network that will not fail between refresh times $f(t)$. This $f(t)$ is generally set to a system wide constant and obeyed by all nodes. In an improved version of the refresh as identified in *Improving the Performance and Robustness of Kademlia-based Overlay Networks [6]* this $f(t)$ is not fixed but distributed over a period where nodes should refresh in a more random fashion. This reduces refresh traffic on the network and allows all the κ nodes to reset their refresh timers. This paper presents a system that removes this refresh timer altogether in favour of a solution that is closer to real time.

In addition the paper *Improving the Performance and Robustness of Kademlia-based Overlay Networks [6]* also promotes a system of down list, whereby a node that is found to be down is reported by the node searching for it back to the node that provided this dead nodes details in a search iteration. This is a significant improvement in the Kadmelia network and with a κ of 20 is reportedly takes the current accuracy measurement of a κ bucket from $13/20$ to $19.8/20$ good nodes. This is a dramatic improvement and will again save on network traffic and time-outs wasted on dead nodes. The system presented here will improve on this performance significantly.

II. KADEMLIA OBSERVATIONS

Some issues with Kademlia and DHTs in general is the ability to amend (or delete) values after publishing. This is due to the inherent distributed nature of the values (as the name suggests). As values are distributed in a pseudo random manner (mathematical predictability becomes an issue due to the large amount of variables (routes, nodes, node names, locations etc.) and random[1] human input).

Searches on Kademlia networks are very fast and iterative (*not recursive*). This means that each node will ask some nodes a question, get answers (hopefully) and from the answers construct a new node list to ask. If this were recursive, then a failed node would halt the process; as this is iterative then failed nodes simply time-out whilst the search continues (this is the β (required replies to continue) requirement, from

[1]Random here is used to indicate not computationally recognisable with today's resources.

the α (loosely parallel level) searches). As Kademlia uses a binary tree and XOR searching the search time per iteration is $\mathcal{O} \ln n$ (where $\mathcal{O}$ is network latency and a factor of XOR distance between node IDs (holes in address space) and n is number of nodes, not number of possible nodes). This figure can be improved upon by the additions shown so far. Further examination of the $\mathcal{O}$ constant may lead to further improvements in search completion times by not only using decreasing distance but also decreasing distance with path protection (multiple routes protected) by examination of XOR distance between parts of the network, particularly where there is binary imbalance in early days.

A. Value Handling

In Kademlia there are several methods of ensuring values are consistent and protected at the same time. The initial value to be considered is K and this should be chosen to be the number of random nodes that can be allowed to go offline in a refresh time (REFRESH). There is also a republish time (REPUB) which is the initial store of the value and sets a time to live (TTL) value on the data. It should be noted in the MAIDSAFE_DHT this TTL may be set at (-1), representing do not time-out. In cases where a TTL is set, then the publisher must store the data again (effectively resetting the TTL) should it wish to maintain the data for an extended period.

1) Caching: In Kademlia, values are cached distant from their natural location of the κ closest nodes. The values are cached using a simple mechanism, whereby the last node that did not have the value, is given if for future searches. This increases efficiency of search and reduced the iteration count. There is currently no mechanism in Kademlia to use these cached values to improve data consistency. See section IV-A for an improvement based upon this omission.

The TTL is used in this instance to calculate how long a value can live and the further away from the κ closest determines the TTL value that each value has. This value is chosen as the reciprocal of the distance from κ of the value (this distance is the XOR ($\oplus$) distance). This distance figure may be further analysed to proved a more efficient mechanism such as the $1/(\oplus distance/current\,height\,of\,tree)$ to be more effective and accurate.

This can present issues when the refresh time (usually 60 mins but configurable) is due. In this case the node must check it is in the κ closest and if so refresh (i.e. send a store_value) the κ closest nodes. β refresh, as described in [6] is used to vary in an evenly distributed way the 60 minute intervals to reduce potential for race conditions in republishing values.

2) Time To Live Values: The TTL can be different for any different type of file, from seconds to a figure indicating do not delete ever (i.e. -1). For fast changing data the TTL should be small, and for unchanging data (like digital keys) the TTL should be very large. This value is only reset by the storing node unless another algorithm for that value type is known by the Kademlia node. Extending kademlia nodes to validate rules based on data types is explain in section IV-B.

B. Routing table architecture

An empirical and difficult to understand part of Kademlia is the routing table. This routing table in a very clever, yet misunderstood part of XOR based routing. It all starts with a single κ bucket which *does not* contain the nodes own address. As the network starts up and nodes are added to this bucket it *splits*. As more nodes are added the 1st bucket continues to split up to the point where there are $n-1$ buckets. Nodes will continue to be added, which if no node vanished would mean the buckets would fill quickly to a maximum node count. As each bucket is created it covers an area of the network which is 2^n in size. In this case n represents the bucket value starting from 0 all the way to $n-1$ therefore the last bucket is 2^{511} in this case is a 512 bit address size. In many cases it is simpler to think of the buckets address as the number of most significant bits in common with bucket having $n-1$ bit in common and bucket 0 (the bucket the node should be in) as the bucket with $n-1$ bits in common (i.e. leaving space of just 2 entries, a 0 and a 1). This number represents the number of nodes covered by this bucket, another way to think of this is that the bucket is responsible for routing in that area of the network.

It then becomes obvious that the last bucket covers half of the network address space therefore 50% of all nodes will exist in a bucket containing κ nodes. In the case under discussion we assumed a $\kappa = 4$. Then 4 nodes are all that is locally known about for information for the other half of the network from the address of the routing tables owner.

This is vital to understand that as the average distance between all the nodes in the last bucket is $2^n/(2 * \kappa)$ which is a significant distance. More importantly though the 1^{st} bucket will contain nodes extremely close and in fact in a full network the first bucket would contain a single node in the last leaf of the binary tree shared by the nodes own address.

In essence the distribution of node distance known by each bucket increases logarithmically and therefore the knowledge of each segment represented by a bucket decreases logarithmically. This presents a difficult to comprehend system that is generally missed by a casual reader of a binary tree based XOR routing algorithm.

III. MANAGED CONNECTIONS

Unlike other algorithms such as chord where each node maintains $num\,keys/num\,nodes$ in Kademlia based networks each node maintains the keys that are between his ID and the ID of the κ closest nodes. This combined with the fact that it is the nodes closest to any node that are more significant in terms of knowledge of the network. This basically states the closer a node is to you then the more you should know about it, for the network to operate correctly this is fundamental.

To achieve this Kademlia uses a notion of κ buckets as described in *Kademlia: A Peer-to-peer Information System Based on the XOR Metric* [5]. The size of the routing table when expressed as κ buckets is $n-1$ where n is the size of the network address in bits. For a SHA1 based addressing scheme this is 159 buckets. Taking a traditional set-up for κ of 20 this means the number of nodes in a routing table is a maximum value of $159 * 20 = 3180$. The MAIDSAFE_DHT default address

space size is 512 (SHA512) which would equate to a routing table node count of $511 * 20 = 10,220$, reducing κ to perhaps 4 would dramatically reduce the routing table size to a more manageable $511 * 4 = 2044$. A κ value of 4 though is perhaps dangerous unless the refresh value was very small indeed, creating lots of traffic in the balance between integrity and network congestion.

The traffic due to down-list and refresh is reasonably high and exists in an attempt to ensure routing tables are maintained with live nodes as well as maintaining data on the network in a reliable fashion.

A. DHT transport

In this DHT implementation, unlike pure Kademlia, we use a reliable transport mechanism, although like Kademlia, the underlying transport is User Datagram Protocol (UDP). In this case a version of reliable overlay was chosen and is currently UDT[2], although this is open to change.

Such transports, like Transport Control Protocol (TCP), sequence data being transferred to allow the receiver to notify the sender on message segment failure, which can be re-transmitted. UDT uses a system of transmitting the acknowledgements of sequences received every δ times and increasing the value of δ on success. This allows a faster throughput that TCP, which acknowledges every packet sent.

UDT allows the network to create connections between nodes and keep these alive by sending very small single bit transmissions every γ seconds. These connections (sockets) if made on the same sending port can allow a demultiplexer to handle many connections with little effort, reducing system overhead.

Using this system a managed connection, or a connection that is not closed transmits a very small packet every γ seconds with a count of failures δ. The minimum time to find a node dead is approximately 30 milliseconds for nodes that gracefully leave the network or $\gamma * \delta$ in the worst case where a node crashes or becomes disconnected without notice.

B. Overhead per connection

In tests the memory overhead per connection is approximately 500Kb. This can be further reduced by removing some unused attributes of a socket structure. The memory overhead is likely to be reduced to between 80 and 150Kb. This paper will assume a connection memory overhead of 150Kb. It is possible that an efficient serialisation mechanism could further reduce this with in memory structures and on disk structures to complete all the socket requirements, allowing either complete socket structures or partial structures to exist on disk, making more efficient use of Random Access Memory (RAM).

The bandwidth costs per managed connection is less than 0.5 bytes per second, given a value of $\gamma = 2$.

[2]A UDP transport also allows traversal of Network Address Translation (NAT) devices allowing nodes to traverse routers. This explanation is extended in the paper *maidsafe_dht NAT traversal* [3].

C. Additional logic required for $1^{st} \kappa Bucket$

In the first bucket, where the values are held, it is imperative that values are shared between the nodes. A value must be shared by all the κ nodes. Ordinarily a refresh carries out this process, but with managed connections, refresh is not called any more. Therefore some logic has to be put in place to ensure values are indeed distributed.

An additional RPC is required:

GIVE_VALUE[VALUES] -> this RPC is executed when a node joins the κ closest to any node. The function is to pass all values between the nodes address and the address of the connecting node.

This RPC should now be sent on joining a κ group, or on finding a node is now part of your κ. This RPC should also be sent to each node in the κ closest to any value held by a node in the event of *any* amendments to *any* values.

The refresh logic of Kademlia is now no longer required as long as a managed connection is maintained for the κ closest nodes at least.

D. Using Managed connections for all routing table entries (requires simulation - VERY IMPORTANT)

If κ could be reduced to a suitably small number (say 4 or even 2, see below) then the number of connections required may be small enough to hold many hundreds connections, particularly through a single port and multiplexer point. In this case the complete network would be relatively current and there would be no requirement for down-list logic (another reduction in code).

1) Incoming connections: In the case where all routing table entries are in fact managed connections, it is not only the routing table entries that require to be managed by the node. Other nodes who add the current node to their routing table will create connections back to this node. This increases the number of structs to be maintained and could grow to a very significant number.

In such network the nodes in one routing table should not be the same nodes in another routing table, this is also a logarithmically possible situation and the higher buckets should increasingly follow this rule, with the closer buckets being likely to have two way nodes that exists in each, particularly the κ bucket. The bucket $n-1$ for instance should certainly not share contacts between nodes. To ensure such a system may work effectively the number of connections required is in fact $2 * \kappa * n$. This allows each bucket to maintain its κ outgoing connections and accept κ incoming connections. This may slow down the population of all κ buckets per node as many may refuse the incoming connections as its κ bucket is full.

2) Replica value count: In standard Kademlia the replica number is κ which suffices and perhaps even improves algorithmic efficiency (through simplicity). There is no reason for this to remain the case and with managed connections and much lower values of κ there is no reason whatsoever for the value replicant count to be $\kappa * 2$ (or any arbitrary number for that matter).

In cases where κ is a low value (such as 2) then using a replicant count of $\kappa * 2$ allows a further increase in data retention, this should be a consideration when reducing the size of the routing table for obvious reasons.

This presents no issues for the DHT and does not impact adversely on any algorithmic complexity.

E. Recursive lookups

It is very important in distributed computing not to hold state on remote actions. This is because remote actions are just that, remote and therefore out of your control. Kademlia handles this well with iterative searches carried out in a loosely parallel fashion as described in II.

With managed connections, however, there is a different situation as we are working with a very current network of nodes who are all in communication. In such a case a recursive lookup may prove significantly faster and also with much less network traffic.

This recursive lookup can now be a single message to the closest node in the routing table, who recursively passes on to their closest node and so on. On any failure the recursion would continue from the previous node to the failure, who has an open RPC that will fail to the failed node and can easily select the next closest node.

On finding a node or value the requester is passed the contact tuple of the node in question from the last node in the chain (not the actual node who has the answer) and then continues with normal kademlia logic, which may involve getting the κ closest nodes in a find node situation or simply getting a value in the get value situation. Caching and last node requests in addition to caching (in future) can then also cache the value in a find value request and do so without being requested.

IV. Additional improvements

A. Resilient caching

As explained in section II-A1, caching can be used to improve data reliability at the DHT level with an improved caching algorithm. As described cache data will expire faster than actual data, this is by design to remove the possibility of stale data. This is a wise stance to take, as stale data may prove to be extremely cumbersome and pollute the data store.

As described in section II-A, only the publisher should republish information. This makes perfect sense and the responsibility to do so should lie with the publisher. This does not, however, mean the network should not attempt to maintain the data up to the TTL expiration time[3] has elapsed.

The additional rules in this case, are relatively simple, a node easily can detect it is holding cache data, it may even be stored in a different location from current data. When the TTL is close to expiring the node should check the κ closest nodes supposedly holding the data still have a copy and if not republish it to them. If a node that should have the data receives such a store command from a cache node it should calculate the distance the node is from it and multiply the reciprocal time used to calculate caching in the first instance, thereby increasing the TTL from nearly zero back to the value closer to the actual TTL the value should be set at. This process should take place a reasonable time before delete such as 30 mins. This can be improved on by using the original TTL and dividing by distance and use half the remaining TTL as the trigger in the remote node. Each time half the TTL remaining comes around again the remote node can trigger a consistency check and store the data again if required by the network. A suitable minimum time can be set by the system (say 180 secs) where this consistency check stops (this is the kCONSISTENCYSTOP setting). In nodes with a -1 TTL (infinity) this consistency check is settable by the designer (kCONSISTENCYCHECK).

If a value is deleted the closest nodes with the value (k closest) should be updated with a -1 value and this should not be cached, this prevents cached copies actually republishing values that are due to be removed anyway (i.e. ignoring or overriding the delete instruction).

[3]it should be noted here that all time is relative to an event and never time from a date that is used. This allows the network to refrain from time synchronisation services in order to calculate events.

B. Extending validation checks based on data types

In standard Kademlia all data is treated equally, in that all key value pairs are just that. The MAIDSAFE_DHT allows signed values and uses an extension of this system to cope with different rules for a value. These rules allow manipulation of data based on pre programmed actions. This allows for a very flexible and very secure network to be created.

Some inbuilt rules are:

1) Only the signatory of a value can delete or amend the value with a remaining TTL
2) If signed data exists all values must be signed (in this case a node is configured as secure and will only talk with other secure nodes and store secure data)
3) Multiple values may exist for any key, and all or none may be independently signed.

In MAIDSAFE_DHT there is an interface which allows the user to extend the functionality of store/delete/amend data. This allows applications using such a system to implement different data (or more correctly value) types and associate functions with those data types. In it's simplest for these rules are based on the identified actions *store, delete and amend*. There are infinitely more possibilities to this process with the inclusion of an unknown RPC message, given and unknown (or not recognised in our rule set) RPC the upper layer applications can act on these RPC's, allowing multiple action types to be executed based on the RPC sent, in this case the DHT layer will signal upper layers with a numbered signal message that the upper layer can parse and associate the correct logic, which may include amend a value including append data to the value, in such cases the DHT layer will in turn be signalled to carry out a refresh (or more accurately resynchronise the value in the network, which in this case is permeate the new value (which requires signatures in a signed system)).

C. Delete or modify values

1) Description: As previously stated, modifying a value in a distributed hash table is notoriously difficult. This is endemic in the very nature of the pseudo random distribution of data in such a network (the DHT strength is one of the largest weaknesses here). If a single value is altered via any method, the value itself may exist in many places and it's finding these places that's the problem. We have already seen the distribution as random, therefore this implies you would be required to search every node in the network (not only closest) for copies and make sure they are also altered. To make this even more difficult, another node may also be altering the value at the same time, thereby causing chaos in the synchronising of data on the network, eventually leading to total collapse.

This would imply that DHTs are only good read only, however, in the case of MAIDSAFE_DHT, this negative attribute is turned into a positive and essential aspect. As noted trying to alter the content of a packet of data that hashes to the key is an illegal instruction in some situations (LIFESTUFF and MSSAN use this extensively).

2) Locks: To achieve reliability of data where multiple copies may exist and be altered from multiple locations is a problem that requires addressing, before changing data. Using the above scenario a node intending to alter a piece of data tries to get all the k-closest nodes via an search. On success each node is sent a lock request which it answers with (Acknowledge) ACK or (Negative Acknowledge) NACK. Each node can pass this request to its known k-closest nodes. On receipt of κ ACKs, the node then alters the value. On receiving an altered value the holding nodes release the lock. Failure to receive a value update in η seconds (settable) seconds should auto release the lock.

Two locks in collision (which can still happen) are both sent a NACK and back off for a random period. This may be a defined algorithm based on network address at a future date. With managed connections this is less likely as the response will be very quick.

In a secured versions of MAIDSAFE_DHT this process has the further check of signature validation which should prevent all collisions.

V. CONCLUSIONS

The MAIDSAFE_DHT presented here is very loosely based on Kademlia. The addition of managed connections is very interesting and makes great use of a connections based protocol where it would appear more obvious to use a connectionless protocol for speed and network efficiency. The move from multiple iterative searches to single recursive searches should prove to be vastly more efficient and significantly faster. The disconnection of κ bucket size from number of replicants values is also significant in allowing a smaller number of routing table entries, but improving data retention and validity.

This represents a use of a DHT system that is exceptionally powerful in distributing network knowledge (Kademlia) and extending this with a system of improvements that transform the DHT into a network that can cope with increasingly quickly altering data. This should represent an improvement in distributed networking and distributed processing.

VI. FUTURE WORK

A. Consider equal distribution of nodes per bucket

To improve performance of searches the distribution of nodes per bucket is important, particularity as the number of buckets increases. This will also ensure a more efficient calculation of tree height or number of nodes currently on the network based on distance between nodes in a routing table. Currently this is most effective in the $n = 0$ bucket, in fact this is due to the fact that bucket should be most closely true in a randomly distributed network, it is also the highest error margin due to the lack of proper scale of measure.

B. Consider pre populating buckets

An algorithm to pre populate buckets using a distance request may assist in mapping the network out quickly. This can be implemented using a system of asking for a node in bucket n asking for the nodes $n-1$ bucket of the same number. This can be easily expanded into a very efficient algorithm.

REFERENCES

[1] As described by Van Jacobson in this link below, August 30, 2006 http://video.Google.com/videoplay?docid=-6972678839686672840

[2] Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan,Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

[3] David Irvine, DHT-based NAT Traversal, david.irvine@maidsafe.net

[4] David Irvine, maidsafe: A new networking paradigm, david.irvine@maidsafe.net

[5] PETAR MAYMOUNKOV AND DAVID MAZI'RESE Kademlia: A Peer-to-peer Information System Based on the XOR Metric {petar,dm}@cs.nyu.edu http://Kademlia.scs.cs.nyu.edu

[6] ANDREAS BINZENH¨ FER AND HOLGER SCHNABEL. Improving the Performance and Robustness of Kademlia-based Overlay Networks o University of W¨ rzburg, Institute of Computer Science u Chair of Distributed Systems, W¨ rzburg, Germany u Email: binzenhoefer@informatik.uni-wuerzburg.de

David Irvine is a Scottish Engineer and innovator who has spent the last 12 years researching ways to make computers function in a more efficient manner.

He is an Inventor listed on more than 20 patent submissions and was Designer / Project Manager of one of the World's largest private networks (Saudi Aramco, over $300M). David is an experienced Project Manager and has been involved in start up businesses since 1995 and has provided business consultancy to corporates and SMEs in many sectors. He has presented technology at Google (Seattle), British Computer Society (Christmas Lecture) and many others. David has spent many years as a lifeboat Helmsman and is a keen sailor when time permits.

Thanks to Yanick Vézina who provided great assistance in proof reading this paper.