

A Netsukuku Domain Name Architecture

<http://netsukuku.freaknet.org>
AlpT (@freaknet.org)

July 3, 2008

Abstract

In this document, we present the ANDNA system. ANDNA is the distributed, non hierarchical and decentralised system of hostname management used in the Netsukuku network.

This document is part of Netsukuku.

Copyright ©2007 Andrea Lo Pumo aka AlpT <alpt@freaknet.org>. All rights reserved.

This document is free; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this document; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Contents

1	Preface	1
2	Introduction	1
3	ANDNA	1
3.1	Hash gnode	2
3.2	Approximated hash gnode	2
3.3	Hook	2
3.4	Cryptographic ANDNA	2
3.5	Hostnames limit	2
3.5.1	Implementation of the limit	2
3.6	Registration step by step	3
3.7	Hostname hibernation	3
3.8	Hash gnodes mutation	4
3.9	Yet another queue	4
3.10	Distributed cache for hostname resolution	4
3.11	noituloser emantsoh esreveR	5
3.12	DNS wrapper	5
4	Scattered Name Service Disgregation	5
4.1	Service, priority and weight number	6
4.1.1	Service number	6
4.1.2	Priority	6
4.1.3	Weight	6
4.2	SNSD Registration	7
4.2.1	Zero SNSD IP	8
4.2.2	SNSD chain	8

1 Preface

We're assuming that you already know the basics of the Netsukuku topology. If not, read the topology document first: [2].

2 Introduction

ANDNA is the distributed, non hierarchical and decentralised system of host-name management in Netsukuku. It substitutes the DNS.

The ANDNA database is scattered inside all Netsukuku network. In the worst case, every node will have to use few hundred kilobytes of memory.

3 ANDNA

ANDNA is based on the principle that every hostname h is a string of maximum 512 bytes. By taking the hash $H(h)$ of the hostname, we obtain a number. We consider this number an IP and we call *hash node* the node which owns the same IP. Every hash node will keep a small portion of the distributed database of ANDNA.

```
Node n
ip: 123.123.123.123
hash( hostname: "netsukuku" ) == 11.22.33.44
    ||
    ||
    Node i
    ip: 11.22.33.44
    { [ Andna database of the node i ] }
    { hash("netsukuku") ---> 123.123.123.123 }
```

The registration of the hostname h , by the node n proceeds as follow:

1. n computes $i = H(h)$
2. n sends a registration request to i
3. i associates in its database $H(h)$ to the IP of n , which is $IP(n)$.

The resolution of the hostname h by the node m :

1. m computes $i = H(h)$
2. m sends a resolution query to i
3. i sends back to m the IP associated to $H(h)$, that in this example is $IP(n)$.

3.1 Hash gnode

It is highly probable that the hash $H(h)$ doesn't corresponds to any active node: if there are a total of 2^{30} active nodes and we are using IP of 32 bits, then by taking the 32 bit hash of h the probability is equal to 25%. Moreover, even the actives node can also exit from the network.

For this reasons, the registration of a hostname will be kept by all the nodes of a single gnodes. This ensures the distribution and the redundancy of the database. The gnode associated to $H(h)$ is the *hash gnode*, and it is simply the gnode of level 1 of the hash node $H(h)$. We will denote the former with $gH(h)$ and the latter with $nH(h)$. In symbols we can write: $nH(h) \in gH(h)$.

3.2 Approximated hash gnode

Since even the hash gnodes cannot exist, an approximation strategy is utilised: the nearest active gnode to the hash gnode becomes the *rounded hash gnode*, which will substitute in full the original (non existent) hash gnode. The rounded hash gnode is denoted with $rgH(h)$.

Generally, when we are referring to the gnode, which has accepted a registration, there is no difference between the two kind of gnodes, they are always called hash gnodes.

3.3 Hook

When a node hooks to Netsukuku, it becomes automatically part of a hash gnode, thus it will also perform the *andna hook*. By hooking, it will get from its rnodes all the caches and databases, which are already distributed inside its hash gnode.

3.4 Cryptographic ANDNA

Before making a request to ANDNA, a node generates a couple of RSA keys, i.e. a public one (*pub key*) and a private (*priv key*). The size of the pub key will be limited due to reasons of space. The request of a hostname made to ANDNA will be signed with the private key and in the same request the public key will be attached. In this way, the node will be able to certify the true identity of its future requests.

3.5 Hostnames limit

The maximum number of hostnames, which can be registered is 256. This limit is necessary to prevent the massive registration of hostnames, which may lead to a overload of the ANDNA system. One its side effect is to slow down the action of spammers, which registers thousands of common words to resell them.

3.5.1 Implementation of the limit

The *counter node* is a node with an ip equal to the hash of the public key of the node, which has registered the hostname h . We'll denote the counter node

associated to the node n with $cr(n)$. Thus, for every node n , which has registered a hostname h , exists its corresponding counter node $cr(n)$.

The counter node $cr(n)$ will keep the number of hostnames registered by n .

When a hash gnode receives the registration request sent by n , it contacts $cr(n)$, which replies by telling how many hostnames have been already registered by the n . If the n has not exceeded its limit, then $cr(n)$ will increment its counter and will allow the hash gnode to register the hostname.

$cr(n)$ is activated by the check request the hash gnode sends. n has to keep the $cr(n)$ active following the same rules of the hibernation (see the chapter below). Practically, if $cr(n)$ doesn't receives any more check requests, it will deactivate itself, and all the registered hostnames will become invalid.

The same distribution technique used for the hash gnode is used for the counter node: there is a whole gnode of counter nodes, which is called, indeed, counter gnode.

3.6 Registration step by step

1. The node n wants to register its hostname h ,
2. it finds the rounded hash gnode $rgH(h)$ and contacts, randomly, one of its node. Let this node be y .
3. n sends the registration request to y . The request includes the public key of the node n . The pkt is also signed with the private key of n .
4. The node y verifies to be effectively the nearest gnode to the hash gnode, on the contrary it rejects the request. The signature validity is also checked.
5. The node y contacts the counter gnode $cr(n)$ and sends to it the IP, the hash of th hostname to be registered and a copy of the registration request itself.
6. $cr(n)$ checks the data and gives its ok.
7. The node y , after the affirmative reply, accepts the registration request and adds the entry in its database, storing the date of registration,
8. y forwards to its gnode the registration request.
9. The nodes, which receive the forwarded request, will check its validity and store the entry in their db.

3.7 Hostname hibernation

The hash gnode keeps the hostname in an hibernated state for about 30 days since the moment of its registration or last update. The expiration time is very long to stabilise the domains. In this way, even if someone attacks a node to steal its domain, it will have to wait 30 days to fully have it.

When the expiration time runs out, all the expired hostnames are deleted and substituted with the others in queue.

A node has to send an update request for each of its hostnames, each time it changes ip and before the hibernation time expires, in this way it's hostname won't be deleted.

The packet of the update request has an id, which is equal to the number of updates already sent. The pkt is also signed with the private key of the node to warrant the true identity of the request. The pkt is sent to any node of the hash gnode, which will send a copy of the request to the counter gnode, in order to verify if it is still active and that the entry related to the hostname being updated exists. On the contrary, the update request is rejected.

3.8 Hash gnodes mutation

If a generical rounded gnode is overpassed by a new gnode, which is nearer to the hash gnode, it will exchange its role with that of the second one, i.e. the new gnode will become the rounded hash gnode.

This transition takes place passively. When the register node will update its hostname, it will directly contact the new rounded gnode. Since the hostname stored inside the old rounded gnode won't be updated, it will be dropped.

While the hostname has not been updated yet, all the nodes trying to resolve it, will find the new rounded gnode as the gnode nearest to the hash gnode and so they'll send their resolution queries to the new gnode.

When the new rounded gnode receives the first query, it will notice that it is indeed a rounded gnode and to reply to the query, it will retrieve from the old hash gnode the andna cache related to the hostname to resolve.

In this way, the registration of that hostname is automatically transferred from the old gnode into the new one.

If an attacker is able to send a registration request to the new hash gnode, before that the rightful owner of the hostname updates it or before that the passive transfer occurs, it will be able to steal it.

To prevent this, the new hash gnode will always double check a registration request by contacting the old hash gnode.

3.9 Yet another queue

Every node is free to choose any hostname, even if the hostname has been already chosen by another node.

The andna cache will keep a queue of `MAX_ANDNA_QUEUE` (5) entries. When the hostname on top of the queue expires, it will be automatically substituted by the next hostname.

3.10 Distributed cache for hostname resolution

In order to optimise the resolution of a hostname, a simple strategy is used: a node, each time it resolves a hostname, stores the result in a cache. For each next resolution of the same hostname, the node has already the result in its cache.

Since the resolution packet contains the latest time when the hostname has

been registered or updated, an entry in the cache will expires exactly at the same time of the ANDNA hostname expiration.

The *resolved hnames* cache is readable by any node.

A node x , exploiting this feature, can ask to any node y , randomly chosen inside its same gnode, to resolve for itself the given hostname. The node y , will search in its resolved cache the hostname and on negative result the node will resolve it in the standard way, sending the result to the node x . These technique avoids the overload of the hash gnodes keeping very famous hostnames.

3.11 noituloser emantsoh esreveR

If a node wants to know all the hostnames associated to an ip, it will directly contact the node which owns that ip.

3.12 DNS wrapper

The DNS wrapper listen to the resolution queries sent by localhost and wraps them: it sends to the ANDNA daemon the hostnames to resolve and receives the their resolutions.

Thanks to the wrapper it is possible to use the ANDNA without modifying any preexistent programs.

See the ANDNS RFC [3] and the ANDNA manual [4].

4 Scattered Name Service Disgregation

The updated SNSD RFC can be found here: [5]

The Scattered Name Service Disgregation is the ANDNA equivalent of the SRV Record[6] of the Internet Domain Name System.

SNSD isn't the same of the "SRV Record", in fact, it has its own unique features.

With the SNSD it is possible to associate IPs and hostnames to another hostname.

Each assigned record has a service number, in this way the IPs and hostnames which have the same service number are grouped in an array. In the resolution request the client will specify the service number too, therefore it will get the record of the specified service number which is associated to the hostname. Example:

1. The node X has registered the hostname "angelica". The default IP of "angelica" is "1.2.3.4".
2. X associates the "depausceve" hostname to the 'http' service number (80) of "angelica".

3. X associates the "11.22.33.44" IP to the 'ftp' service number (21) of "angelica".

When the node Y resolves normally "angelica", it gets 1.2.3.4, but when its web browser tries to resolve it, it asks for the record associated to the 'http' service, therefore the resolution will return "depauseve". The browser will resolve "depauseve" and will finally contact the server. When the ftp client of Y will try to resolve "angelica", it will get the "11.22.33.44" IP.

The node associated to a SNSD record is called "SNSD node". In this example "depauseve" and 11.22.33.44 are SNSD nodes.

The node which registers the records and keeps the registration of the main hostname is always called "register node", but it can also be named "Zero SNSD node", in fact, it corresponds to the most general SNSD record: the service number 0.

Note that with the SNSD, the NTK_RFC 0004 will be completely deprecated.

4.1 Service, priority and weight number

4.1.1 Service number

The service number specifies the scope of a SNSD record. The IP associated to the service number 'x' will be returned only to a resolution request which has the same service number.

A service number is the port number of a specific service. The port of the service can be retrieved from `/etc/services`.

The service number 0 corresponds to a normal ANDNA record. The relative IP will be returned to a general resolution request.

4.1.2 Priority

The SNSD record has also a priority number. This number specifies the priority of the record inside its service array.

The client will contact first the SNSD nodes which have the higher priority, and only if they are unreachable, it will try to contact the other nodes which have a lower priority.

4.1.3 Weight

The weight number, associated to each SNSD record, is used when there are more than one records which have the same priority number. In this case, this is how the client chooses which record using to contact the servers:

The client asks ANDNA the resolution request and it gets, for example, 8 different records.

The first record which will be used by the client is chosen in a pseudo-random

manner: each record has a probability to be picked, which is proportional to its weight number, therefore the records with the heavier weight are more likely to be picked.

Note that if the records have the same priority, then the choice is completely random.

It is also possible to use a weight equal to zero to disable a record.

The weight number has to be less than 128.

4.2 SNSD Registration

The registration method of a SNSD record is similar to that described in the NTK.RFC 0004.

It is possible to associate up to 16 records to a single hostname. The maximum number of total records which can be registered is 256.

The registration of the SNSD records is performed by the same register node. The hash node which receives the registration won't contact the counter node, because the hostname is already registered and it doesn't need to verify anything about it. It has only to check the validity of the signature.

The register node can also choose to use an optional SNSD feature to be sure that a SNSD hostname is always associated to its trusted machine. In this case, the register node needs the ANDNA pubkey of the SNSD node to send a periodical challenge to the node. If the node fails to reply, the register node will send to ANDNA a delete request for the relative SNSD record.

The registration of SNSD records of hostnames which are only queued in the andna queue is discarded.

Practically, the steps necessary to register a SNSD record are:

1. Modify the `/etc/netsukuku/snsd_nodes` file.

```
register_node# cd /etc/netsukuku/
register_node# cat snsd_nodes
#
# SNSD nodes file
#
# The format is:
# hostname:snsd_hostname:service:priority:weight[:pub_key_file]
# or
# hostname:snsd_ip:service:priority:weight[:pub_key_file]
#
# The 'pub_key_file' parameter is optional. If you specify it, NetsukukuD will
# check periodically 'snsd_hostname' and it will verify if it is always the
# same machine. If it isn't, the relative snsd will be deleted.
#
```

```
depausceve:pippo:http:1
depausceve:1.2.3.4:21:0
```

```
angelica:frenzu:ssh:1:/etc/netsukuku/snsd/frenzu.pubk
```

```
register_node#
register_node# scp frenzu:/usr/share/andna_lcl_keyring snsdfrenzu.pubk
```

2. Send a SIGHUP to the NetsukukuD of the register node:

```
register_node# killall -HUP ntkd
# or, alternatively
register_node# rc.ntk reload
```

4.2.1 Zero SNSD IP

The main IP associated to a normal hostname has these default values:

```
IP = register_node IP # This value can't be changed
service = 0
priority = 16
weight = 1
```

It is possible to associate other SNSD records in the service 0, but it isn't allowed to change the main IP. The main IP can only be the IP of the register_node.

Although it isn't possible to set a different association for the main IP, it can be disabled by setting its weight number to 0.

The string used to change the priority and weight value of the main IP is:

```
hostname:hostname:0:priority:weight
```

For example:

```
register_node# echo depausceve:depausceve:0:23:12 >> /etc/netsukuku/snsd_nodes
```

4.2.2 SNSD chain

Since it is possible to assign different aliases and backup IPs to the zero record, there is the possibility to create a SNSD chain. For example:

```
depausceve registers: depausceve:80 --> pippo
pippo registers:      pippo:0 --> frenzu
frenzu registers:     frenzu:0 --> angelica
```

However the SNSD chains are ignored, only the first resolution is considered valid. Since in the zero service there's always the main IP, the resolution is always performed.

In this case ("depausceve:80 -i pippo:0") the resolution will return the main IP of "pippo:0".

The reply to a resolution request of service zero, returns always IPs and not hostnames.

References

- [1] Netsukuku website: <http://netsukuku.freaknet.org/>
- [2] Netsukuku topology document: [topology.pdf](#)
- [3] ANDNS RFC: [Andna and dns](#)
- [4] ANDNA manual page: [andna\(8\)](#)
- [5] SNSD RFC: [SNSD](#)
- [6] SRV record: [RFC 2782](#)
[SRV on wikipedia](#)

