# P2P over Netsukuku

NTK RFC 0014

http://netsukuku.freaknet.org

August 3, 2007

**Abstract**

This text describes how is it possible to create a distributed P2P service over the Netsukuku network. As example, a distributed P2P Bittorrent service is presented.

## 1 Introduction

Netsukuku is a distributed, collaborative network of nodes. For this reason, the development of P2P applications over Netsukuku is rather easy.

A P2P application over Ntk can directly access the information regarding every part of the network by reading the maps and can known immediately its dynamic changes by listening to QSPN packets. In order to ease the development of such applications, a "ntkp2p" library will be developed.

## 2 P2P structure

The P2P architecture is a Distributed Hash Table.

**Definition 2.1.** **KEY** is the key space, i.e the set of all the keys.
**IP** is IP space, i.e. the set of all the IPs of the network. In the ipv4 case, we have $\mathbf{IP} = ipv4 = \{n \in \mathbb{N} \mid 0 \leq n \leq 2^{32} - 1\}$, in the ipv6 case we have $\mathbf{IP} = ipv6 = \{n \in \mathbb{N} \mid 0 \leq n \leq 2^{128} - 1\}$.
Taking an IP $m \in \mathbf{IP}$, we can write it as a compound of gnode IDs of different levels: $m = m_l.m_{l-1}.\ldots.m_1.m_0$, where $l$ is the highest level[1]. $m_i$ is the ID of the gnode of level $i$ where $m$ belongs. For more information, see the topology document [4].

**Definition 2.2.** The Netsukuku network can host up to $2^{16}$ different P2P services. Each registered service has a unique identification number called PID (P2P ID). The set of all PIDs is $\mathbf{PID} = \{n \in \mathbb{N} \mid 0 \leq n \leq 2^{16} - 1\}$.
A node is a *participant* to a P2P service $p \in \mathbf{PID}$ if it exists in the network and if it announced its partecipation to the service (we'll see later how).
$\mathbf{IP}^*$ is the set of all the IPs of the participant nodes of the network, i.e. $\forall x \in \mathbf{IP}^* \ \exists_1$ a participant node X: ip(X)=x.

[1] for example, in ipv4 we have $m = m_3 * 256^3 + m_2 * 256^2 + m_1 * 256^1 + m_0$

**Definition 2.3.** The function $h : \mathbf{KEY} \longrightarrow \mathbf{IP}$ maps a key $k$ to an IP $x$. If the keys have the same bit length of the IPs, then h can be simply defined as the identity function, for example, if **KEY** is the md5 hashes set and $\mathbf{IP} = ipv6$[2].
The function $H(x) : \mathbf{IP} \longrightarrow \mathbf{IP}^*$, is defined as follow

$$H(x) = \max \{y \in \mathbf{IP}^* \mid \forall t \in \mathbf{IP}^* \ \ |y - x| \leq |y - t|\}$$

in simple words, $H(x)$ is the closest existent IP to $x$.
The function $\overline{h} : \mathbf{KEY} \longrightarrow \mathbf{IP}^*$ is $\overline{h}(k) = H(h(k))$.
The node having the IP $\overline{h}(k)$, is called hash node of $k$.

## 2.1 Becoming a participant

A node $g$, in order to become an active participant of a $p \in \mathbf{PID}$, sends a CTP inside its gnode $G$. This CTP is considered interesting by a node $g' \in G$, if $g'$ didn't know that $g$ is a participant of $p$ or if the CTP is interesting as described in the QSPN document[3].
This same procedure is reitered in all the higher levels: $G$ becomes an active participant of $p$, because it has at least one participant node. $G$ sends a CTP in level 1, informing all the gnodes it is a participant. Etcetera.

Note: at the end of the above procedures, all the nodes of the network will know what gnode and nodes are participants to $p$. Note 2: it isn't necessary to become a participant to store or retrieve data from $p$.

## 2.2 Storing information in $p$

Suppose a node $n$ wants to store some data $d$ in the distributed P2P service $p \in \mathbf{PID}$. It will operate as follow:

1. It computes the key $k$ from the data $d$.
2. It computes $m = \overline{h}(k)$:
   (a) $m' := h(k)$
   (b) $m := H(m')$ is computed with the use of the maps of $n$ as follow:

$m := undef$
**for** $i = 0, 1, 2, \ldots, l$
    *let* $m_{l-i}$ be the closest participant (g)node ID to $m'_{l-i}$
    [the search of $m_{l-i}$ is restricted inside the gnode $n_{l-i+1}$]
    **if** $m_{l-i} \neq n_{l-i}$
        *break*
**if** $m = undef$
    There are no participants to $p$

Notes:

i. below a certain point, some levels of $m$ may be left undefined. This happens when the break istruction is executed.

[2] An md5 hash has 128 bit, an ipv6 IP too

ii. if $n$ itself is a participant node, then $n_{l-i}$ surely is a participant.

3. $n$ sends to $m$ the *store request* $r$:
   (a) If all the levels of $m$ are defined, then $m \in n_1$. Thus, $n$ can directly contact $m$.
   (b) otherwise, if the last defined level of $m$ is $i$ , then $r$ will be routed by the ntk nodes to a node $\overline{m} \in m_i$.
   The node $\overline{m}$ will then recompute $m = \overline{h}(k)$. This time, more levels will be defined, because $\overline{m}$ is nearer to the $m$ and its map can give more accurate information (regarding $m$).
   (c) This same procedure is reiterated: $\overline{m}$ forwards $r$ to a node $\overline{\overline{m}} \in m_j$, where $j < i$. This node will compute $m = h(k)$, and so on.
   (d) Finally, $m$ will receive the request $r$.
4. At this point, the $p$-protocol will dictate what $m$ must do with $r$ , f.e. $m$ can store the data in its cache.
5. To ensure redundancy, if $m$ has accepted the request, it will send it to 31 nodes, which have the closest IP to $m$. In this way, when $m$ dies, they will automatically replace it.

$m$, and all the backup nodes, will automatically delete the stored data after a time limit specified by the $p$-protocol. For this reason, $n$ has to repeat the storing procedure periodically. This also has the benefit of restoring dead hash nodes: if $m$ and all the backup nodes dies simultaneusly, and if $m \notin n_1$, then $n$ won't be able to know their death, however, with the next periodic storing procedure, a new hash node $m$ will be elected.

Suppose a node $b$ enters in the gnode $m_1$. Let $B$ be the set containing the hash node $m$ and all its backup nodes. If the IP of $b$ is closer to $\overline{h}(k)$ than at least one node $b' \in B$, then $m$ will send the request $r$ to $b$, in this way $b$ will become the new hash node or will substitute a backup node.
This same procedure is applied to any level, f.e. suppose that a new gnode $g$ joins in $m_2$. If its ID is closer to $\overline{h}(k)$ than the ID of $m_2$, then $m$ will send to $g$ the request $r$. In this way, $g$ will substitute $m_1$.
An optimisation to this procedure is based on the following observation: if a (g)node $g$ joins the network and if its IP is closer to $\overline{h}(k)$ than the actual hash gnode, then all the successive P2P requests will be routed to $g$. Since $g$ knows it is a new (g)node, it has to simply forward the received requests to the old hash node, that is $m$, to see if it knows the answer to the requests. This method has the advantage that there will be no delay when a new, closer, hash node joins the network.

# 3 Examples of P2P over Ntk

1. We'll now present as example a simple P2P Bittorrent[5] service.

Suppose a node $n$ wants to share a file $f$. $n$ will calculate the key $k$ from the filename of $f$. $n$ will send a store request to $\overline{h}(k)$, where the data of the request is the *.torrent* metafile generated from $f$. The node $\overline{h}(k)$, after having accepted the request, will start up a bittorrent tracker. Finally, $n$ will connect to the tracker $\overline{h}(k)$, becoming a seeder.

The node $m$ that wants to download the file, has to simply calculate $k$ and connect to the tracker $\overline{h}(k)$.

2. As a further example consider ANDNA[6]. In ANDNA there are two different P2P services in one: the service that stores the IPs and the service of the counter nodes, that serves to store the number of hostnames registered by a node.

# References


[1] Netsukuku website: http://netsukuku.freaknet.org/

[2] Netsukuku RFCs: NTK RFCs

[3] QSPN document: qspn.pdf

[4] Netsukuku topology document: topology.pdf

[5] Bittorrent: Bittorrent

[6] ANDNA document: andna.pdf

^_^