UDC 004.75; 004.724.2

G. V. Poryev
National Technical University of Ukraine «Kyiv Polytechnic Institute»
Peremohy ave., 37, 02056 Kiev, Ukraine

Binary Trees Approach to Speedup Address Range Lookup in Peer-to-Peer Solutions

It has been analyzed specifics of underlying algorithms for address range lookup. Concerning the storage of IP ranges, the drawbacks and bottlenecks of existing implementations have been determined. A new, faster method based on binary trees for storing and accessing the IP range databases is proposed.

***Key words**: binary trees, distributed networks, peer-to-peer.*

Introduction

The beginning of third millennium is marked by the networking technologies in general and Internet in particular reaching so extreme a level of integration and mobility within our everyday life that it would not be an underestimation to compare it with the importance of electric power. Today, both technologies are so crucial to the progress and the survival of humankind that virtually no activity is done without either one.

Of particular interest is the variety of the so-called geolocation services, now widely popular among content providers. Geolocation is the set of techniques to relate client IP address to his/her geographical location. This kind of information helps focusing the advertisement taking into account cultural specifics and local demand, choose potentially preferable user interface language based on the country, provide search for nearby facilities such as shopping, entertainment, transportation etc.

Since every IP address belongs to the IP range registered with some LIRs (Local Internet Registries), which, in turn, are a part of larger IP range superset allocated by the superior LIR or RIR (Regional Internet Registry) it is possible to track business address of a registree entity and presume that its clientele are likely to be located in the neighboring territory.

The usual ways of doing this involve either querying RIR directly or maintain a local cache of RIR database snapshot, against which to query. The query takes a form of plain IP address and outputs the administrative and technical details, including address which is then parsed to extract country and city.

The information relevant to this simple way of geolocation is stored in two places. One being *delegated* table, which lists primary allocated IP ranges, year of allocation and ISO code of the country it's been assigned to. For the more detailed information, RIR clients either refer to RIR's WHOIS server or look up the *inetnum* database which gives more detailed business address of the entities responsible.

There are, however, other ways in which geolocation techniques can be useful. As was mentioned in our previous work [1], a special type of internetworking called «peer-to-peer» (p2p) is taking hold in the modern Internet, by some estimates already generating more than half of total Internet traffic worldwide.

Being able to efficiently manage the large and worldwide p2p network is a challenge and there were many different approaches [2], including those concerned with optimal clustering of the p2p network. An efficient clustering is impossible without the efficient metric that we call «locality metric», which is a method to estimate the topological distance between the arbitrary nodes of a network.

While there were many attempts to build such a metric, we believe that our approach envisioned in the series of publications [3, 4] is the most suitable for p2p needs. This article, however, is not about improving the estimation method itself but rather about speeding it up, because, as of today, most reference implementations still suffer from performance issues.

On the other hand, geolocation-like methods can help advance topological metric, should we consider that the essence of both is the query for corresponding IP range for a given address.

Previous work

Unlike CARMA, current implementation of the locality metric called «SAMPAN» for Service-oriented Affinity Metric for Peer-to-peer Architecture Networks employs separation of the database component from the business-logic component like to widely known Model-View-Controller scheme. The database consists of three tables: *assets*, *atoms* and *linkage*.

The *assets* table is simple and contain only two columns — numeric identifier and a name for an asset in terms of RIR allocations. This is to maintain asset reference by indexes in another two tables. As of August 2012, there were over 12 thousands registered assets in the Internet.

The atoms table is a primary table containing both delegated ranges for IPv4 and IPv6 address space and Autonomous System Numbers (ASNs). Its declaration in terms of MySQL language is as shown in Fig. 1.

Almost all fields in the Fig. 1 are self-descriptive except the *baseaddr*, *volume* and *refid*. In case of the IPv4 or IPv6 range, the *baseaddr* contain network identifier converted into the unsigned long integer, *volume* lists the total number of addresses allocated in this range (which is not always in the form of 2^n) and *refid* refers to the identifier (*id*) field of the ASN or superset IP range entry in this same table to indicate the linkage relationship. The entities are linked bottom-up, meaning that the *refid* is always pointing to the larger inclusive range or an ASN. In case of ASN, *baseaddr* contains the number of the ASN, while *volume* and *refid* are NULL. As of August 2012 there were

over 3.8 million of IPv4 ranges, over 41 thousands of IPv6 ranges and over 55 thousands ASNs in the Internet.

```
CREATE TABLE `atoms` (
   `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT COMMENT 'Atom ID',
   `isocc` CHAR(2) NULL DEFAULT NULL COMMENT 'ISO country code' COLLATE 'utf8_unicode_ci',
   `type` ENUM('ipv4','ipv6','asn') NOT NULL COMMENT 'Atom type' COLLATE 'utf8_unicode_ci',
   `baseaddr` BIGINT(20) UNSIGNED NOT NULL COMMENT 'Atom base address',
   `volume` BIGINT(20) UNSIGNED NULL DEFAULT NULL COMMENT 'Atom range volume',
   `refid` INT(10) UNSIGNED NULL DEFAULT NULL COMMENT 'Reference id',
   PRIMARY KEY (`id`),
   INDEX `k_baseaddr_volume` (`baseaddr`, `volume`),
   INDEX `k_isocc_refid` (`refid`, `isocc`),
   INDEX `k_isocc` (`isocc`)
)
COMMENT='Contain ranges, subranges, ASes and their linkage'
COLLATE='utf8_unicode_ci'
ENGINE=MyISAM
AUTO_INCREMENT=3975803;
```

Fig. 1. Declaration of the atoms table

The third *linkage* table contains information concerning the linkage of ASN into assets and assets into each other. Its declaration in terms of MySQL language is as shown in Fig. 2.

```
CREATE TABLE `linkage` (
   `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT COMMENT 'Link ID',
   `type_a` ENUM('asn','asset') NOT NULL COMMENT 'Type of Endpoint A' COLLATE 'utf8_unicode_ci',
   `node_a` INT(10) UNSIGNED NOT NULL COMMENT 'ID of Endpoint A',
   `type_b` ENUM('asn','asset') NOT NULL COMMENT 'Type of Endpoint B' COLLATE 'utf8_unicode_ci',
   `node_b` INT(10) UNSIGNED NOT NULL COMMENT 'ID of Endpoint B',
   PRIMARY KEY (`id`),
   UNIQUE INDEX `type_a_node_a_type_b_node_b` (`type_a`, `node_a`, `type_b`, `node_b`)
)
COMMENT='Contain link mesh with ASNs and ASSETs'
COLLATE='utf8_unicode_ci'
ENGINE=MyISAM
AUTO_INCREMENT=157636;
```

Fig. 2. Declaration of the linkage table

The SAMPAN, like CARMA, takes the two IP addresses as input, but, unlike CARMA, uses external MySQL storage engine to perform all relevant queries. These may include the following:

— query for *id* in the *atoms* table to determine all IP ranges the input address belongs to; check if any of the range cover both addresses;

— among those ranges found, leave only those with non-NULL *refid* field, meaning only those ranges that are assigned to the ASN;

— for each range found and filtered in the previous query, retrieve all entries of all ASNs; check again to see if both input addresses belong to the same ASN;

— retireve all entries from the *linkage* table which contain any of the thus found ASN in either of the node fields, indicating that the ASN are a part of larger asset;

— at this point, the algorithm has two lists of assets which are related to two input addresses and those two lists are checked for common entries;

— if the previous point yielded no common asset, another query is made to determine if there exists an asset, which is linked to any two of the assets in the previous lists;

— if no common asset is found, one final query is made to verify whether both ranges are registered to the same country.

The experiments undertaken on the SAMPAN reference implementation have shown the performance to be suboptimal and largely inacceptable for p2p deployment scenarios where quick estimation decisions are required. For instance, on the hardware such as four-core AMD Phenom 9550 processor with 4GB RAM and the most favorable running conditions for MySQL Server Community Edition it took almost 10 seconds to pass all calculations for all flavors for both IP addresses. Since most p2p network clients have peer lists on the order of tens, usually around 50–100, that would slow down the network significantly, requiring more than 10 minutes to just process and sort peer list.

The apparent bottleneck was determined to occur while executing the first pair of queries to find all IP ranges a given addresses are belong to. The query involved is following: «*SELECT * FROM atoms WHERE INET_ATON(ipaddr) BETWEEN baseaddr AND baseaddr+volume ORDER BY volume LIMIT 16*».

Despite properly configured indexing (namely, *k-baseaddr-volume* index specifically for address lookup) the average query execution time is 5 seconds. Even though EXPLAIN modifier shows that the index is actually being used, the number of processed rows remain large, usually one to two millions. This is believed to be caused by the use of BETWEEN clauses, since MySQL server is not particularly good in queries for inexact matches like this one.

Therefore, a new approach is needed, one that will be specifically tailored for IP address range lookups.

Using custom binary trees for performing range lookups

Let's consider the nature of IP range lookups in this particular scenario. For any given IP address, it is very common to find more than one range it belongs to. Ranges appear to be not overlapping in the sense that every shorter range is a subrange of the larger one and every shorter range «belongs» to only one larger range, that do not overlap too.

Moreover, the ranges are finite. IPv4 address space takes no more than 2^{32} addresses, but various kinds of reserved ranges lower that value by approximately $5.8 \cdot 10^8$. Since minimal allocation slot is of 4 addresses, it is reasonable to estimate the upper number of possible IPv4 ranges as $\frac{2^{32} - 5.8 \cdot 10^8}{4} \approx 9.2 \cdot 10^8$. The actual volume of allocated IPv4 ranges, as mentioned earlier, is two orders of magnitude less due to utilization of the large allocation ranges.

It seems that binary trees may be very useful to organize IPv4 ranges due to their natural adherence to power of two — a descent from the root node to the lowermost layer with the capacity to contain 4 million nodes would take about 22 steps.

Let's estimate the total memory footprint required to hold the extensive database for IPv4 ranges. For every range, the following data should be stored: a) Start address, which also serve as tree node identifier; for IPv4 it needs 32 bits; b) Range volume, number of nodes in that range; also 32 bits; c) Reference ID, number of ASN associated with the range; this requires 32 bits.

If the tree node is not terminating (i.e. containing the range data) but a range lookup fork, it will need two fields of 32 bits each — pointer (branch) to the left and right nodes. Therefore the byte width of the lowermost layer $WN = 12$, for all other layers $W = 8$.

Hence the total memory footprint for the binary tree for all ranges consisting of N layers from the root node downward except the latest N-th layer is the following

$$M_s = \sum_{i=1}^{N-1} M_i W + M_N W_N = -\left(1 - 2^{N-1}\right) W + 2^N W_N . \quad (1)$$

For the proposed scenario, assuming $N = 22$ we get $M_s = 2^{26} - 2$ or 64 MBytes. Again this is considerably less memory than required using SAMPAN approach (whose atoms table alone require 512 MBytes and above), and therefore is more affordable for mobile or embedded applications with memory constraints.

Processing range information from *delegated* and *inetnum* tables yields unsorted sequence, so to build sorted binary tree we have to devise the proper filling algorithm.

For every triplet of «range», «volume» and «refid» that enters the tree, we convert «range» from IPv4 standard dotted representation into the 32-bit unsigned integer using standard BSD sockets function *inet_addr()* so that tree can be sorted by this «index».

The root node is exactly in the middle of all IPv4 range, which equals 2^{31}. Depending on the numeric value of the given IPv4 range, a left or right branch is initialized and the corresponding node of next level is created (if it was not initialized and created prior to that). The process repeats itself on each of subsequent nodes, creating ones and initializing branches if necessary, until the tree depth reaches preset $N = 22$. At this point, a data (non-forking) node is created, holding all three values.

When the complete tree is built, the lowermost node layer will contain sorted range array, with ascending start addresses. The rough schematic layout of such tree is shown on Fig. 3.

It should be noted, however, that the estimation given by equation (1) is very optimistic, assuming that the address space tree never splits further down below 2^{22} and the ranges are distributed uniformly at the bottom level of the tree.

This is, however, not the case. Taking into account that a multiple registered range may share single *baseaddr*, let us estimate the pessimistic required volume of the tree in terms of number of nodes, having established the following additional considerations.

First, the complete 31-tier tree spanning the entire IPv4 address space is not necessary, nor is it technically viable. The actual number K of allocated IPv4 ranges is ap-

proximately 4 million. To accommodate K entries, the lowermost tier of the «cap» part of the tree (that spawns two branches from each node) should be $N = 2^n > K$.

Second, once the tree population algorithm reaches the lowermost layer, the node ID there will most likely not match the specified input ID, so the pseudo-branching will continue to the 31-st layer — «pillar» part of the tree. In this context, we call pseudo-branching a situation where each node spawns only one branch to lower layer.

Third, it is possible that ASN reference might occur in the upper layers (above n); it is also possible that true branching occur below n, so the margin is rather provisional. What is certain, though, is that below n-th layer the total number of nodes per layer do not increase. Therefore given K we can estimate the maximum possible volume of nodes required to build such a tree by using the following equation:

$$M_s(n,K) = M_{cap}(n) + M_{pillar}(n,K) = \left(2^{n+1} - 1\right) + K(31 - n). \qquad (2)$$

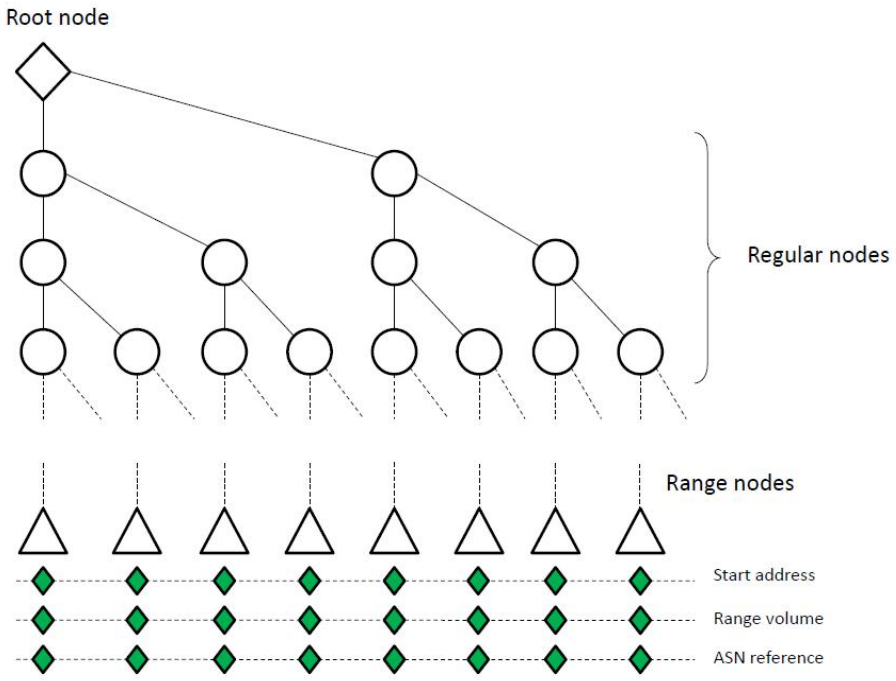

Fig. 3. Schematic layout of the custom binary tree to hold IP ranges database

For instance, by mid-December 2012 the number of allocated IPv4 ranges was K = 3,898,098, which we safely approximate as $4 \cdot 10^6$, hence «cap» depth n = 22, «pillar» depth $p = 31 - n = 9$, number of nodes in the lower layer of «cap» is $N = 2^{22} =$ = 4,194,304, total number of nodes in the «cap» part $M_{cap}(n) = (2^{22} + 1 - 1) =$ 8,388,607, total number of nodes in the «pillar» part $M_{pillar}(n, K) = 3{,}898{,}098(31 - 22) =$ = 35,082,882, total number of nodes in the entire tree thus being $M_s(n, K)$ = 43,471,489. The schematic representation of the tree is given in Fig. 4.

It should be noted, that this estimation of $M_s(n, K)$ gives only the maximum possible number of nodes for the given K, as worst-case scenario. Assuming the same upper

bound for the amount of 12 bytes necessary per each node, we can safely estimate the upper bound for the total memory footprint to be about 500 MBytes.

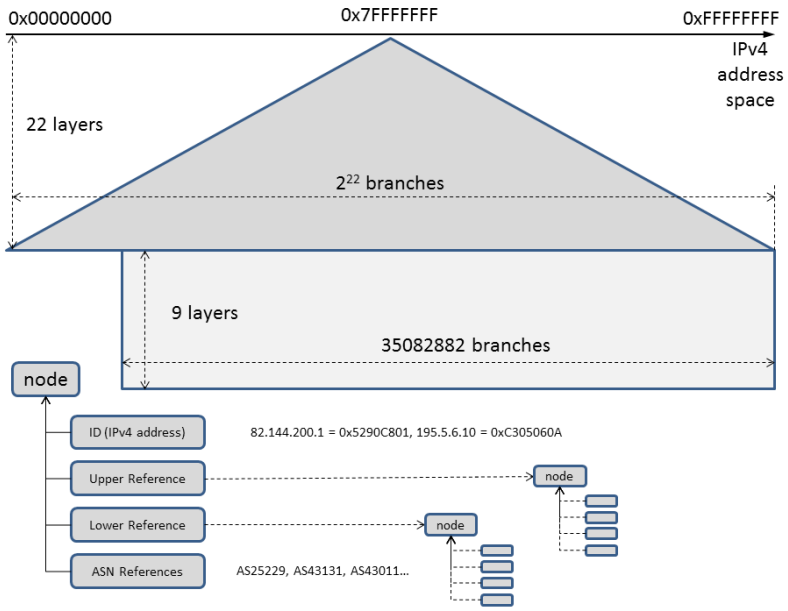

Fig. 4. Customized binary tree layout and volume estimation

Range lookup method

Let's now consider how the range lookups should be done for any range holding tree. To locate IPv4 range, converted input address is compared to the single node on each layer, starting from the root node on the topmost. If the number of input node is greater or lower than that of the sequence node and there exist corresponding branches, another node is chosen for the same lookup based on this branch. If there is no branch, the algorithm stops and returns to the previous level. If the lowermost layer is reached, the algorithm also stops and returns the corresponding node.

The tree-walking algorithm can be expressed as follows. Let $c(I, i, j)$ be a function that return a j-th node Mj on the lowermost layer, and I is the input address. This way, a stored range $M(I)$ can be calculated by the recursive function

$$M(I) = c(I,1,1), \tag{3}$$

where

$$c(I,i,j) = \begin{cases} I < N_i, \exists N_{i+1,2j-1} : c(I, i+1, 2j-1), \\ I > N_i, \exists N_{i+1,2j} : c(I, i+1, 2j), \\ I = N_i : M_j. \end{cases} \tag{4}$$

Since every tree building iteration eventually lands some range at the lowermost layer, there should be no hanging nodes — those with neither branch initialized and not on the lowermost layer.

However, there is a chance that an input IP range would require to choose a non-initialized branch. For instance, if an IP address is allocated right before the registered range starts on the right branch, the algorithm will be stopped because there's no left branch from the previous node. In this case the algorithm should perform a «backtrack» to find next downward branching to the existing range. This backtrack is done by repeatedly checking the previous nodes until the algorithm reaches a node with two existing branches, which it had already passed, turning to the right branch. This time the left branch is chosen and the lookup occurs again from this node. Hence the lookup should end at the next adjacent or nearby range to the left from where the previous backtracking occurred.

Because IP ranges are not necessary adjacent, the lookup of an IP address may not land at the second range even after tree backtracking. In this case the backtracking is repeated moving generally in left direction (since ranges are «extending» to the right) until a suitable range is found.

Conclusion

Although it is generally accepted that the computational complexity for the binary tree lookup varies from $O(\log n)$ to $O(n)$ our test runs have confirmed the observed speedup in comparison to the SQL approach corresponds to $O(\log n)$, since only 3,1 % of the IP addresses from the testing BitTorrent swarm required tree backtracking at all, and 1,7 % required tree backtracking performed more than once to complete range lookup. On the same hardware as mentioned earlier, the average lookup timeout was 120 msec.

Thus we have conclusively demonstrated that using the specially designed binary tree may not only decrease the memory footprint requirements for IP address range lookup routines but, more importantly, significantly increase the lookup speed.

1. *Poryev G.V.* CARMA: A Distance Estimation Method for Internet Nodes and Its Usage in P2P Networks / G.V. Poryev, H. Schloss, R. Oechsle // International Journal on Advances in Telecommunications. — 2010. — Vol. 3, N 3,4. — P. 114–128.

2. *iPlane:* An Information Plane for Distributed Services / Madhyastha H., Isdal T., Piatek M., Dixon C. // In Proceedings of the 7-th USENIX Symposium on Operating Systems Design and Implementation. — USENIX, 2006. — P. 367–380.

3. *Poryev G.* Improved CARMA Locality Estimation Model for Peer List Reordering and Its Experimental Validation / G. Poryev, V. Poryev // Data Rec., Storage & Processing. — 2011. — Vol. 13, N 4. — P. 3–11.

4. *Poryev G.V.* Direct Observation of Performance Improvement in Filesharing p2p Networks Using CARMA Techniques // Journal of Qafqaz University. — 2011. — N 31.— P. 16–22.

Receive 14.01.2013