Root Server Selection in Recursive DNS Software

No Author Given

No Institute Given

Through dynamic debugging, we review the root query policies in four types of popular recursive DNS software: BIND 9 [3], Unbound [5], Knot Resolver [1] and PowerDNS Recursor [6]. All software reuses *authoritative server (NS) selection algorithms* for root queries, and in the case of the DNS root, root servers are their NSes. Below we list pseudo-code and detailed analysis of each type of DNS software.

1 BIND 9

Algorithm 1 shows the selection algorithm of BIND 9. At startup, BIND launches priming queries [4] to find the information of all root servers. For a set of NSes BIND maintains their *adjusted RTTs*, which are dynamically updated using the Exponentially Weighted Moving Average (EWMA) [2], taking the response status (e.g., error and timeout rate), latency and lameness of each NS as factors. From this set BIND will select and query the NS with the **lowest adjusted RTT**. NSes that have not been queried will be given a default adjusted RTT value ranging from 1 to 32 microseconds, such that in early stages they will quickly be tried and have their RTTs updated. After each round BIND also decreases the adjusted RTTs for NSes that are not selected, giving them chances to be selected in less cases. IPv6 addresses are preferred to IPv4.

2 Knot Resolver

Algorithm 2 shows the selection algorithm of Knot Resolver. Knot Resolver launches priming queries at startup. It uses the **ϵ-Greedy algorithm** to select from a list of NSes, giving priorities to NSes that support IPv6, are never tried, with less query failures and lower delay. In around 1/20 cases, Knot Resolver will randomly select one from all untried NSes.

3 Unbound

Algorithm 3 shows the selection algorithm of Unbound. Unlike other software, Unbound does not launch prime queries at startup and only queries root servers when necessary. An adjusted RTT is also maintained for each NS, and NSes with bad status (e.g., lame DNSSEC, long latency or timeout) will receive penalty. Each untried NS is given an initial adjusted RTT of 376ms. Unbound **randomly**

selects from a list of valid NSes and removes an NS from consideration only when its adjusted RTT is over 400ms longer than the least-latent. However, in the case of DNS root, only few locations across the globe witness a delay of over 400ms thus Unbound will tend to select root servers randomly.

Note that the algorithm described here is enabled by default. Through custom configuration, Unbound can also select the best NS from a valid list at a given probability.

4 PowerDNS Recursor

Algorithm 4 shows the selection algorithm of PowerDNS Recursor. PowerDNS Recursor launches prime queries at startup. Simliar to BIND, PowerDNS Recursor maintains the adjusted RTT for each NS and dynamically update them using the EWMA. The default adjusted RTT value is 0ms, thus in early stages PowerDNS Recursor selects NSes randomly. If a query to an NS fails, the server will be set as throttled and removed from consideration temporarily. **It selects NS with the lowest adjust RTT and decays the adjusted RTT for all servers with the same factor. The longer the query interval, the lower the decay factor and adjusted RTTs.** It also removes the RTT record of a NS periodically to ensure every record is up-to-date. Because root servers are typically not frequently queried, the interval will be long, resulting in low decay factor and adjusted RTTs for servers not selected in the current round.

References

1. CZ.NIC: Knot resolver (2021), `https://www.knot-resolver.cz/`
2. Hunter, J.S.: The exponentially weighted moving average. Journal of quality technology **18**(4), 203–210 (1986)
3. ISC: Bind 9 (2021), `https://www.isc.org/bind/`
4. Koch, P., Larson, M., Hoffman, P.E.: Initializing a DNS Resolver with Priming Queries. RFC 8109 (Mar 2017). https://doi.org/10.17487/RFC8109, `https://rfc-editor.org/rfc/rfc8109.txt`
5. Labs, N.: Unbound (2021), `https://www.nlnetlabs.nl/projects/unbound/about/`
6. PowerDNS: Powerdns recursor (2020), `https://www.powerdns.com/recursor.html`

Algorithm 1 BIND 9: Select the best NS candidate

```
1: function SortAllNSes(L_NS)
2:     for each ns ∈ L_NS do
3:         if ns is an IPv4 address then
4:             tmp_srtt of ns ← _srtt of ns + penalty value
5:         end if
6:     end for
7:     bubble sort L_NS by tmp_srtt                          ▷ ascending
8:     return L_NS
9: end function
10:
11: function FindTheBestNS(L_NS)
12:     for each ns ∈ L_NS do
13:         if ns isn't tried then
14:             set ns is tried
15:             return ns
16:         end if
17:     end for
18:     return no more NS
19: end function
20:
21: function AfterQuery(ns, rtt, L_NS)
22:     for each ns ∈ L_NS do
23:         if ns isn't tried then
24:             _srtt of ns ← (_srtt of ns · 2^9 − _srtt of ns)/2^9
25:         end if
26:     end for
27:     if query successed then
28:         v ← rtt
29:     else
30:         v ← a penalty value
31:     end if
32:     _srtt of ns ← _srtt of ns ·(1 − a) + a · rtt      ▷ a is depended on status of ns
33: end function
```

Algorithm 2 Knot Resolver: Select the best NS candidate

```
1: function FindHigherPriorityNS(Two NS candidates)
2:     if one NS has IPv6 address, the other one don't and IPv6 network enabled then
3:         ns ← the NS has IPv6 address
4:     else if one NS is never tried before, the other one has tried then
5:         ns ← the NS which is never tried before
6:     else if one NS has less error in previous probes then
7:         ns ← the NS has less error
8:     else
9:         if one NS has lower _srtt then
10:            ns ← the NS has lower _srtt
11:        else
12:            return equal priority
13:        end if
14:    end if
15:    return ns
16: end function
17:
18: function FindTheBestNS(L_NS)
19:    shuffle L_NS randomly
20:    quick sort L_NS by FindHigherPriorityNS                    ▷ descending
21:    r ← a random value, r ∈ [0, 1)                             ▷ ϵ-Greedy Selection
22:    if r < 1/20 then                                           ▷ Explore
23:        ns ← a random untried NS in L_NS
24:    else                                                       ▷ Exploit
25:        ns ← L_NS[0]
26:    end if
27:    if ns has errors before and still has untried NS in L_NS then
28:        ns ← a random untried NS in L_NS
29:    end if
30:    return ns
31: end function
32:
33: function AfterQuery(ns, rtt)
34:    if ns is never tried before then
35:        _srtt of ns ← 400ms
36:        set ns has tried
37:    else
38:        if query successed then
39:            use rtt to update _srtt of ns according to section 3 in RFC6298
40:        else
41:            error of ns ← error of ns + 1
42:        end if
43:    end if
44: end function
```

Algorithm 3 Unbound: Select the best NS candidate

```
1: function FindTheBestNS(L_NS)
2:     for each ns ∈ L_NS do
3:         if ns is bogus or lame or in unsupported network or not allowed to be queried then
4:             remove ns from L_NS
5:             continue
6:         else if ns is never tried before then
7:             tmp_srtt of ns ← 376ms
8:         else if ns is in a bad status (e.g., dnssec lame, huge timeout) then
9:             tmp_srtt of ns ← a corresponding penalty value
10:        else
11:            tmp_srtt of ns ← _srtt of ns
12:        end if
13:        best_srtt ← Min(best_srtt, tmp_srtt of ns)
14:    end for
15:    for each ns ∈ L_NS do
16:        if tmp_srtt of ns > best_srtt + 400ms then
17:            remove ns from L_NS
18:        end if
19:    end for
20:    return a random NS in L_NS
21: end function
22:
23: function AfterQuery(ns, rtt)
24:    set ns is tried in this turn of query
25:    if query successed then
26:        use rtt to update _srtt of ns according to section 3 in RFC6298
27:    else
28:        record corresponding status(as described in FindTheBestNS) of ns
29:    end if
30: end function
```

Algorithm 4 PowerDNS: Select the best NS candidate

```
1: now ← current time (seconds)
2:
3: function SortAllNSes(L_NS)
4:     for each ns ∈ L_NS do
5:         if ns isn't set throttled then
6:             d ← last of ns − now
7:             _srtt of ns ← _srtt of ns · exp(d/60)
8:             last of ns ← now
9:         end if
10:    end for
11:    shuffle L_NS randomly
12:    stable sort L_NS by _srtt                          ▷ ascending order
13:    return L_NS
14: end function
15:
16: function FindTheBestNS(L_NS)
17:    for each ns ∈ L_NS do
18:        if ns isn't tried then
19:            return ns
20:        end if
21:    end for
22:    return no more NS
23: end function
24:
25: function AfterQuery(ns, rtt)
26:    set ns is tried in this turn of query
27:    if query successed then
28:        if ns doesn't have _srtt record then
29:            _srtt of ns ← rtt
30:        else
31:            d ← last of ns − now
32:            a ← exp(d)/2
33:            _srtt of ns ← _srtt of ns ·a + (1 − a) · rtt
34:            last of ns ← now
35:        end if
36:    else
37:        set ns is throttled.
38:    end if
39: end function
40:
41: function HouseKeeping(L_NS) ▷ An independent thread to remove the status of
    NSes periodically
42:    for each ns ∈ L_NS do
43:        every 5 seconds, remove the throttled status of ns
44:        every 200 seconds, remove _srtt of ns, whose now − last>300
45:    end for
46: end function
```