Design and Implementation of the bowline DNS Full Resolver

At IIJ, we develop our own DNS full resolver (also known as a caching DNS server; we refer to this simply as a full resolver below). We named our software bowline after the bowline knot, often called the king of knots. At this point, we have finished implementing most of the functions necessary for ISP operations, such as logging and monitoring, and are verifying stability through trial operations. This article describes the design and implementation of bowline.

IIJ provides caching DNS servers using multiple full resolver implementations. One of bowline's aims is to increase the number of independent implementations and improve attack resilience. Another key point is that IIJ has complete control over it. Having it under our control means we should be able to quickly implement countermeasures against new attack methods and replace running servers.

We implemented bowline in Haskell, and we have released it as open source. Our reasons for using Haskell are explained in "3.2 Why Implement it in Haskell?" in our previous article "Implementing QUIC in Haskell"[*1]. The most important point is that Haskell provides lightweight threads (simply referred to as "threads" below), which makes it possible to structure software more flexibly and with better clarity than with event-driven programming.

2.1 Development History and Library Structure

In 2010, as an anti-spam initiative, the author implemented a framework in Haskell that integrates SPF, Sender ID, DomainKeys, and DKIM. To use these technologies, DNS lookup functionality is essential. Initially, I used a well-known DNS stub resolver library written in C through a foreign function interface (FFI), but I discovered that it did not work well under Haskell's advanced concurrency due to frequent assertion failures.

I therefore stopped using that library and developed a DNS stub resolver library written entirely in Haskell (named dns). Because of the language's features, writing everything in Haskell makes it much easier to achieve high concurrency. Indeed, Haskell's practical applicability has been demonstrated in multiple Internet services.

I started developing bowline in 2022 together with my colleague Hibino. Hibino handled the key functions of a full resolver: iterative resolution, caching, and DNSSEC validation. I have been developing libraries in Haskell for HTTP/2, TLS 1.3, QUIC, and HTTP/3 since 2013, and I was interested in applying them to DNS, so I primarily developed the transport layer.

The prototype of bowline used the dns library, but in our experience, dns lacked extensibility and also had memory fragmentation issues. To resolve these shortcomings without worrying about backward compatibility, we decided to develop a new suite of DNS libraries. The libraries are divided by function and all begin with the prefix dnsext. They provide the following functionality.

- dnsext-types: Extensible, non-fragmenting basic data types and encoders/decoders for basic RRs (Resource Records)
- dnsext-dnssec: Encoders/decoders for DNSSEC-related RRs, DNSSEC validator
- dnsext-svcb: Encoders/decoders for the recently standardized SVCB (Service Binding) RR
- dnsext-utils: Utility functions such as logging and caching
- dnsext-do53: Client-side DNS over UDP and TCP
- dnsext-dox: Client-side DNS over HTTP/2, HTTP/3, TLS, and QUIC
- dnsext-iterative: Iterative resolution algorithm; server-side DNS over UDP, TCP, HTTP/2, HTTP/3, TLS, and QUIC
- dnsext-bowline: The bowline full resolver, the dug DNS lookup command[*2], and the ddrd daemon[*3], which enables discovery of encrypted DNS servers

dnsext-dnssec and dnsext-svcb serve as examples of how dnsext-types can be extended.

*1 Internet Infrastructure Review (IIR) Vol. 52, "Implementing QUIC in Haskell" (https://www.iij.ad.jp/en/dev/iir/052.html).
*2 IIJ Engineers Blog: "Introducing the DNS lookup command dug" (https://eng-blog.iij.ad.jp/archives/27527, in Japanese).
*3 IIJ Engineers Blog: "Discovering encrypted DNS servers" (https://eng-blog.iij.ad.jp/archives/31843, in Japanese).

2.2 Thread Structure

Generally, a full resolver is expected to operate as follows.

- Receive recursive query requests (Recursion Desired flag on) from stub resolvers
- Search the cache for each request, and if a "positive response" or "negative response" exists, return it to the stub resolver
- If not found, repeatedly send iterative queries (Recursion Desired flag off) to authoritative servers, obtain a positive or negative response, return it to the stub resolver, and store a new entry in the cache

Iterative resolution using the network takes much more time than cache lookups, which are a memory operation. So the software must be designed so that iterative resolution does not adversely affect the processing of other requests and responses.

Historically, UDP has been the primary transport for DNS, and when multiple requests arrive from the same stub resolver, the full resolver returns responses as soon as each RR is resolved. With connection-oriented transports, meanwhile, the full resolver receives requests in the order the stub resolver sends them. If the resolver tries to return responses in that same order, iterative resolution for one request can block responses to subsequent requests. To avoid this, connection-oriented transports require the resolver to return responses as they become available (pipelining). In other words, even with connection-oriented transports, the resolver must behave in the same way it does over UDP.

Suppose we designed the system such that a single thread handles both cache lookups and iterative resolution. That thread may block while performing iterative resolution. To process subsequent requests without delay, the system would then need to spawn a new thread for every request. With this design, the number of threads responsible for core functionality could become huge, making the system fragile.

We therefore decided to use separate worker threads for cache lookups ("cache-lookup workers") and iterative resolution ("iterative-resolution workers"). On a cache miss, a cache-lookup worker delegates iterative resolution to an iterative-resolution worker. Cache-lookup workers do not block, but iterative-resolution workers may block. A fixed number of cache-lookup workers and iterative-resolution workers are created at server startup. Smooth pipelining can be achieved if the number of iterative-resolution workers is sufficiently larger than the number of cache-lookup workers.

Figure 1 shows the thread structure we designed given the above considerations. The dashed rectangle represents bowline as a whole, and the gray rectangles represent threads. The receiver and sender handle the transport. For UDP, one pair stays resident per network interface. For

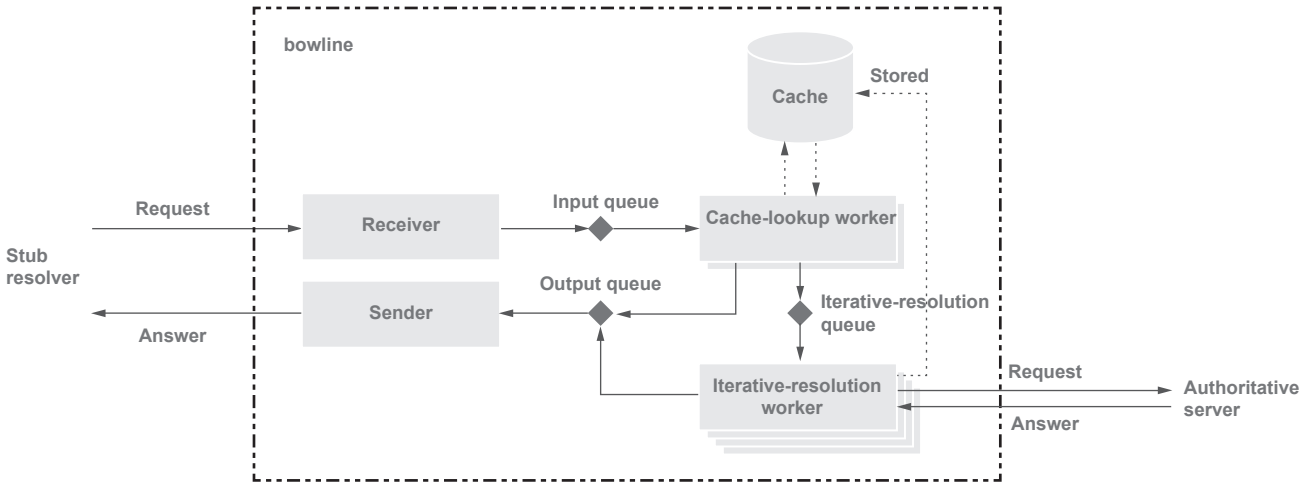

Figure 1: bowline's Thread Structure

connection-oriented transports, one listener thread stays resident per network interface, and a receiver/sender pair is spawned each time a connection is created. For HTTP/2, QUIC, and HTTP/3, auxiliary threads are also started.

The receiver decodes DNS requests from the stub resolver, converts them to a Haskell data type called Input, and places them in the global input queue. The Input contains a reference to the output queue of the sender corresponding to that receiver.

A cache-lookup worker receives an Input from the global input queue and searches the cache. This operation does not block. If the lookup succeeds, it represents the result in a data type called Output and stores it in the output queue referenced by the Input. If it fails, it relays the Input to the global iterative-resolution queue. By default, bowline has four cache-lookup workers.

An iterative-resolution worker reads an Input from the iterative-resolution queue and performs iterative resolution. This operation may block. If all iterative-resolution workers are blocked, the entire iterative-resolution subsystem cannot make progress, so the number of iterative-resolution workers must be sufficiently large. Each iterative-resolution worker represents the result of iterative resolution as an Output and stores it in the output queue. By default, bowline uses 128 iterative-resolution workers.

The sender reads Output from its own output queue, encodes it into a DNS response, and sends it to the stub resolver.

2.3 Iterative Resolution Algorithm

Authoritative DNS servers behave as follows in response to client queries.

- If the queried domain name is in a zone the server manages and an RR of the queried RR type exists, it returns that value
- If the queried domain name is in a zone the server manages and the query name does not exist or an RR of the queried RR type does not exist, it returns the SOA RR to provide the TTL for caching the negative response
- If the queried domain name is in a zone the server manages and delegation information for a child zone exists, it returns the delegation information (NS RR) and glue (A RR or AAAA RR) for the child zone so that the resolver can continue iterative resolution

In iterative resolution, the full resolver acts as a client when querying authoritative servers. In the RFC 1034 iterative algorithm, the query name and query type remain fixed to the values specified by the stub resolver.

Suppose, for example, that the stub resolver requests resolution of the TXT RR for www.example.jp. The full resolver ultimately queries the authoritative servers for "example.jp." for the TXT RR of www.example.jp, and similarly uses the same request when querying the root (".") authoritative servers and the intermediate "jp." authoritative servers.

This mechanism arguably gives Internet eavesdroppers many opportunities to observe the original query name and query RR type. So in the interests of protecting privacy, QNAME minimization was proposed. In QNAME minimization, authoritative name servers are queried using only the necessary portions of the original domain name. And for the intermediate query RR type, RFC 7816 uses NS RRs, while RFC 9156 uses A or AAAA RRs.

Table 1 summarizes the query names and query RR types used when resolving the TXT RR for www.example.jp.; bowline only uses QNAME minimization as defined in RFC 9156 for iterative resolution. The RFC 9156 column in Table 1 is explained in detail below.

1. Query the "." authoritative server for the A RR of "jp." to obtain the authoritative servers for "jp."
2. Query the "jp." authoritative server for the A RR of "example.jp." to obtain the authoritative servers for "example.jp."
3. Query the "example.jp." authoritative server for the

A RR of “www.example.jp.” and receive an answer, indicating there is no further delegation.

4. Query the “example.jp.” authoritative server for the TXT RR of “www.example.jp.”

The seemingly redundant query at the end occurs to check for delegation while hiding the original query RR type. If the original query RR type happens to match the RR type used by the algorithm, the number of queries is reduced by one.

If the query name exists as expected, one or more authoritative server names are obtained at each stage, and each authoritative server has one or more A/AAAA RRs. As mentioned above, A/AAAA RRs may be returned as glue along with NS RRs.

2.3.1 Implementation of Iterative Resolution

When bowline receives a request from a stub resolver, it first performs root priming before beginning iterative resolution against authoritative servers as a client. Root priming involves querying the built-in candidate authoritative servers for “.” (hint information) for the latest NS RR of “.” and simultaneously resolving the A/AAAA RRs. The result is cached. So as long as it is within its validity period, subsequent requests will use the cached value.

Once this special processing completes, bowline repeats the following steps until it obtains the final answer. First, it forms two groups: authoritative servers whose IP addresses are known and those whose IP addresses are unknown.

Next, for authoritative servers with known IP addresses, it randomly shuffles the IP addresses and selects two such that the authoritative server names do not overlap. It then queries those servers in parallel with requests following the QNAME minimization algorithm.

In our implementation, this client functionality is provided by the dnsext-do53 library, which can spawn query threads for multiple servers to race against each other and take the first response returned.

If the queries fail, it moves on to the next two candidates. If all queries to authoritative servers with known IP addresses fail, it moves on to the authoritative servers with unknown IP addresses. At this stage, it randomly shuffles the authoritative servers and takes the first one, selects either A or AAAA at random, and performs a new iterative resolution for the authoritative server name. If resolution fails, it proceeds to the next name. If resolution succeeds, it performs the intended query for this step against one of the obtained IP addresses.

If all queries fail, the overall lookup fails and an error is returned to the stub resolver. If any attempt succeeds and it obtains an exact match for the target RR, iterative resolution is complete. Otherwise, because it has obtained the names of lower-level authoritative servers, it repeats this step using a query name one label longer.

Table 1: Examples of QNAME Minimization

	Authoritative server	RFC 1034	RFC 7816	RFC 9156
1	.	www.example.jp. TXT	jp. NS	jp. A
2	jp.	www.example.jp. TXT	example.jp. NS	example.jp. A
3	example.jp.	www.example.jp. TXT	www.example.jp. NS	www.example.jp. A
4	example.jp.		www.example.jp. TXT	www.example.jp. TXT

2.3.2 Iterative Resolution Error Cases

NODATA is an error where the domain name exists but there is no value for the queried RR type (values for other RR types exist) and there is no next authoritative server to query. In the RFC 1034 algorithm, which sends the full query name, if NODATA is returned, that is the final result. But with QNAME minimization, the resolver must extend the query name and continue searching. For example, querying the ".jp." authoritative server for "ad.jp." yields NODATA, but querying for "iij.ad.jp." yields authoritative server information. This is because "jp." and "ad.jp." are the same zone.

NXDOMAIN is an error indicating that the domain name does not exist. If an intermediate domain name results in NXDOMAIN, RFC 8020 specifies that no domain names exist beneath it. In practice, however, looking up a lower-level name may still succeed. So even if an intermediate name returns NXDOMAIN, bowline continues querying with longer domain names. Only when the queried name ultimately returns NXDOMAIN does it report this error.

SERVFAIL indicates a server failure, REFUSED that the query was rejected for some reason, and FORMERR a format error. When these errors occur, bowline falls back to the next candidate IP address and continues searching.

2.3.3 Visualization with dug

The aforementioned dug is a DNS lookup command with two modes. In one mode, it acts as a simple stub resolver and requests recursive resolution from a full resolver. It can also query authoritative servers with the Recursion Desired flag turned off.

In the other mode, it uses the same iterative resolution implementation as bowline and provides a visualization of the process. The block on the right-hand side of the page shows an example of iterative resolution using dug. The -i flag specifies iterative mode, -vv increases verbosity, and +cdflag (check disabled) disables DNSSEC validation.

```
% dug -i -vv iij.ad.jp. txt +cdflag
...
root-priming: query "." NS
    query @2001:7fe::53#53/UDP "." NS
    query @198.97.190.53#53/UDP "." NS
    query @2001:7fe::53#53/UDP "." NS: win
root-priming: verification success - RRSIG of NS: "."
    "a.root-servers.net." 198.41.0.4#53 2001:503:ba3e::2:30#53
    "b.root-servers.net." 170.247.170.2#53 2801:1b8:10::b#53
    "c.root-servers.net." 192.33.4.12#53 2001:500:2::c#53
    "d.root-servers.net." 199.7.91.13#53 2001:500:2d::d#53 (※1)
    "e.root-servers.net." 192.203.230.10#53 2001:500:a8::e#53
    "f.root-servers.net." 192.5.5.241#53 2001:500:2f::f#53
    "g.root-servers.net." 192.112.36.4#53 2001:500:12::d0d#53
    "h.root-servers.net." 198.97.190.53#53 (※2) 2001:500:1::53#53
    "i.root-servers.net." 192.36.148.17#53 2001:7fe::53#53
    "j.root-servers.net." 192.58.128.30#53 2001:503:c27::2:30#53
    "k.root-servers.net." 193.0.14.129#53 2001:7fd::1#53
    "l.root-servers.net." 199.7.83.42#53 2001:500:9f::42#53
    "m.root-servers.net." 202.12.27.33#53 2001:dc3::35#53
iterative: query "jp." A
    query @2001:500:2d::d#53/UDP "jp." A (※1)
    query @198.97.190.53#53/UDP "jp." A (※2)
    query @2001:500:2d::d#53/UDP "jp." A: win
delegation - no DS, check disabled: "." -> "jp."
zone: "jp.":
    "a.dns.jp." 203.119.1.1#53 2001:dc4::1#53
    "b.dns.jp." 202.12.30.131#53 2001:dc2::1#53 (※4)
    "c.dns.jp." 156.154.100.5#53 (※5) 2001:502:ad09::5#53
    "d.dns.jp." 210.138.175.244#53 2001:240::53#53
    "e.dns.jp." 192.50.43.53#53 2001:200:c000::35#53
    "f.dns.jp." 150.100.6.8#53 (※3) (※6) 2001:2f8:0:100::153#53
    "g.dns.jp." 203.119.40.1#53
    "h.dns.jp." 161.232.72.25#53 2a01:8840:1bc::25#53
iterative: query "ad.jp." A
    query @150.100.6.8#53/UDP "ad.jp." A (※3)
    query @2001:dc2::1#53/UDP "ad.jp." A (※4)
    query @150.100.6.8#53/UDP "ad.jp." A: win
delegation - no delegation: "jp." -> "ad.jp."
cache-soa: no verification - check-disabled: "jp."
iterative: query "iij.ad.jp." A
    query @156.154.100.5#53/UDP "iij.ad.jp." A (※5)
    query @150.100.6.8#53/UDP "iij.ad.jp." A (※6)
    query @156.154.100.5#53/UDP "iij.ad.jp." A: win
delegation - no DS, check disabled: "jp." -> "iij.ad.jp."
zone: "iij.ad.jp.":
    "dns0.iij.ad.jp." 210.130.0.5#53 2001:240::105#53 (※8)
    "dns1.iij.ad.jp." 210.130.1.5#53 (※7) 2001:240::115#53
resolve-exact: query "iij.ad.jp." TXT
    query @210.130.1.5#53/UDP "iij.ad.jp." TXT (※7)
    query @2001:240::105#53/UDP "iij.ad.jp." TXT (※8)
    query @2001:240::105#53/UDP "iij.ad.jp." TXT: win
no verification - check-disabled: "iij.ad.jp."
;; HEADER SECTION:
;Standard query, NoError, id: 0
;Flags: Recursion Available

;; QUESTION SECTION:
;iij.ad.jp.               IN          TXT

;; ANSWER SECTION:
iij.ad.jp.   3600(1 hour) IN          TXT
"20f10da4-fb66-42ac-941e-133d9c6c09ba"
iij.ad.jp.   3600(1 hour) IN          TXT
"v=spf1 include:spf.iij.ad.jp include:spf.dox.jp -all"
iij.ad.jp.   3600(1 hour) IN          TXT
"_globalsign-domain-verification=qSNEOr9rmlIgx-CLJgTN-uypossXQTKD4GuWN-Jwp0"

;; AUTHORITY SECTION:

;; ADDITIONAL SECTION:

;; 172usec
```

Of the candidate authoritative server IP addresses, those actually used are annotated with "(※number)." "win" indicates which of the two competing lookups won. If there is delegation, the IP addresses are listed; otherwise, "no delegation" is displayed.

2.4 DNSSEC Iterative Resolution Algorithm

DNSSEC uses digital signatures, a type of public-key cryptography, to establish an authentication chain for delegation information. There are two DNSSEC-related RRs within a zone.

- DNSKEY: The public key used to verify digital signatures (RRSIGs) provided by the zone (DNSKEYs include KSKs and ZSKs, but we do not distinguish them in this article)
- RRSIG: A digital signature over RRs managed by the zone

Verifying an RRSIG RR using a DNSKEY RR allows us to confirm that the data has not been tampered with. However, we cannot tell whether the claimed domain name is truly legitimate. We therefore need to prove that it is delegated from the parent domain. The following RR is provided for this purpose.

- DS: An RR for registering the cryptographic hash value of a zone's DNSKEY in the parent zone

For iterative resolution requests with the DO (DNSSEC OK) flag on, authoritative servers behave as follows.

- If there is delegation to a child zone: when returning delegation information such as NS RRs, return DS RRs as well
- If there is no delegation to a child zone: If the returned RRs have corresponding RRSIG RRs, return those as well

The validation of delegation combined with QNAME minimization ends up being somewhat complex, so rather than providing a general explanation, we use Figure 2 to briefly explain how bowline resolves the A RR for "www.example.jp". Note that we assume the DS RR for "." is provided in advance as a trust anchor.

- Query a built-in candidate "." authoritative server for the "." DNSKEY RR (b), and select the DNSKEY RR that matches the hash value (a) for "." provided as a trust anchor. The response also includes the RRSIG RR over the DNSKEY RR (c). Using the public key contained in the "." DNSKEY RR, verify the signature in the RRSIG RR. If verification succeeds, trust the "." DNSKEY RR.

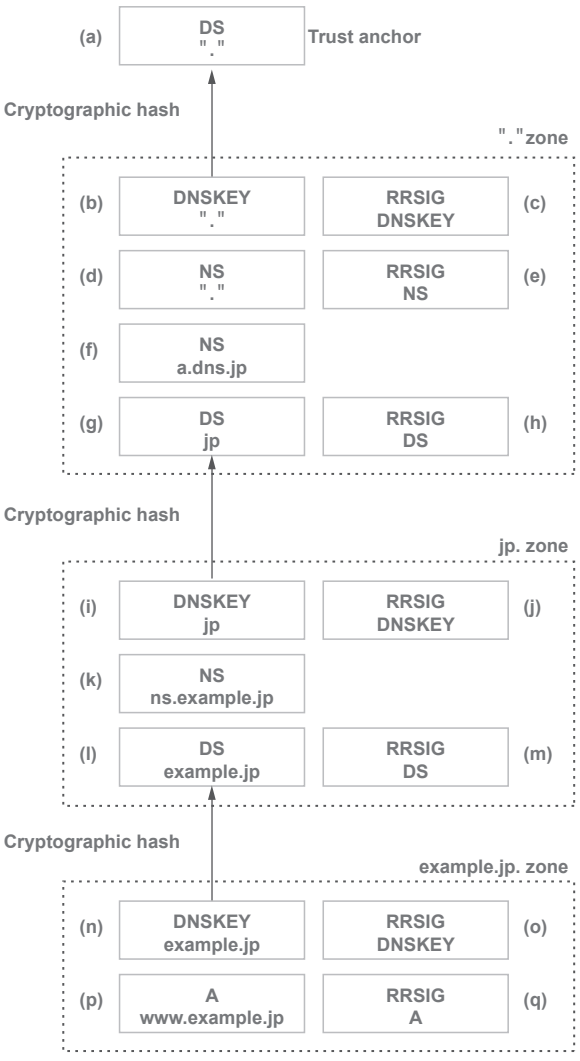

Figure 2: DNSSEC Authentication Chain

- Query a built-in candidate "." authoritative server for the NS RR of "." (d) to obtain the list of "." authoritative servers. Verify the signature in the accompanying RRSIG RR (e) with the "." public key, and trust the "." authoritative servers (root priming).
- Query a "." authoritative server for the A RR of "jp." This yields NS RRs (f) indicating "jp." authoritative servers and A/AAAA RRs, but there is no corresponding RRSIG because this is not information managed by the "." zone. It also yields the DS RR (g) containing the hash of the DNSKEY RR of "jp." with an accompanying RRSIG RR (h). Verify this signature with the "." public key. If successful, trust the delegation from "." to "jp."
- Query a "jp." authoritative server for the DNSKEY RR of "jp." (i), and extract the DNSKEY matching the hash for "jp." The response includes the DNSKEY RR and its RRSIG RR (j). If verification of the signature using the "jp." public key contained in the DNSKEY RR succeeds, trust the "jp." DNSKEY RR.
- Query a "jp." authoritative server for the A RR of "example.jp." This yields NS RRs (k) and A/AAAA RRs indicating "example.jp." authoritative servers. It also yields the DS RR (l) containing the hash of the DNSKEY RR of "example.jp." along with its RRSIG RR (m). Verify this signature with the "jp." public key. If successful, trust the delegation from "jp." to "example.jp."
- Query an "example.jp." authoritative server for the DNSKEY RR of "example.jp." (n), and extract the DNSKEY matching the hash for "example.jp." The response includes the RRSIG RR (o) over the DNSKEY. Using the public key contained in the "example.jp." DNSKEY, verify the signature in the RRSIG RR. If verification succeeds, trust the "example.jp." DNSKEY RR.
- Query an "example.jp." authoritative server for the A RR of "www.example.jp." (p), which also yields the RRSIG RR (q). Verify this signature and trust the final answer.

Since not all zones support DNSSEC, the chain of trust may be broken mid-way. In bowline, if the chain of trust is broken, the DO flag is turned off for subsequent iterative resolution.

2.4.1 Parent–Child Coexistence Problem

As described above, authoritative servers return DS RRs in response to queries for names in a child zone so that the resolver can continue the lookup. So in most cases it is not necessary to explicitly query for DS RRs. But there are cases in which DS RRs may not be returned even though the chain of trust has not been broken.

This occurs when the parent zone and the child zone are managed by the same authoritative server. As an example, let the parent zone be "a." and the child zone be "b.a.", and assume that both zones are managed by the same authoritative name server.

Suppose we query the authoritative server for "a." requesting the A RR of "b.a." If the parent and child are not co-located, the server returns delegation information for "b.a.", namely NS RRs and DS RRs. But in this example, the parent and child are co-located, and because a DNS query does not specify which zone it targets, the longest-match rule applies and the "b.a." zone is taken to be the target. If an A RR exists, it is returned; otherwise an SOA RR is returned, and no DS RR is obtained.

In this case, we need to determine that delegation exists even though a DS RR was not returned, and explicitly query for the DS RR. The method bowline uses is as follows.

- If an A RR for "b.a." exists, then an RRSIG RR is returned along with it. If the Signer's Name field in the RRSIG RR is "b.a.", then the parent and child are co-located.
- If an A RR for "b.a." does not exist, an SOA RR is returned. Extract its domain name (the RR's NAME field). If it is "b.a.", the parent and child are co-located.

2.4.2 DO Flag Specified by Stub Resolver

The DO flag is also used by the stub resolver to tell the full resolver whether it supports DNSSEC. The full resolver behaves as follows.

If the DO flag is off: Validate DNSSEC as much as possible, strip DNSSEC-related RRs from the response, and do not set the response's AD (Authenticated Data) flag.

If the DO flag is on: Return all DNSSEC-related RRs as well. If all validations succeed, set the response's AD flag.

If delegation for DNSSEC disappears partway through, do not set the response's AD flag. If validation fails anywhere, return SERVFAIL.

2.4.3 Proof of Non-Existence

DNSSEC also includes proof of non-existence (NSEC/NSEC3 RRs). The technical details are beyond the scope of this article, but for details, see "Integrating DNSSEC into a DNS Full Resolver Implementation—Proving Negative Responses with NSEC/NSEC3"[*4].

2.5 Cache Data Structure

For cache data structure, we use a PSQ (Priority Search Queue), which has properties of both a search tree and a priority queue. The key is the request (domain name, RR type, etc.), the priority is the TTL (Time To Live), and the value is the "iterative resolution result." In other words, we can efficiently look up an iterative resolution result from a request and also delete cache entries according to TTL.

Positive responses are divided into (1) those without DNSSEC signatures, (2) those with signatures that are not validated, and (3) those for which signature validation succeeds. The cache lifetime is basically the RR's TTL. But for DNSSEC it is also constrained by the RRSIG RR's TTL and the signature validity period.

Negative responses (NODATA and NXDOMAIN) are not divided up; they are cached using the TTL value obtained from the SOA RR. If the response is DNSSEC-signed, NSEC/NSEC3 RRs and RRSIG RRs are also returned. These serve as proof of non-existence for NODATA or NXDOMAIN, so they are cached together with the SOA.

For SERVFAIL, REFUSED, and FORMERR, an SOA RR cannot be obtained. Since we want to obtain a positive response or a NODATA/NXDOMAIN response if possible, we fall back to the next candidate IP address. If the candidates are exhausted, we cache these error responses to prevent attacks that repeatedly send the same query. For the TTL, we use the value specified in the configuration file.

Responses are assigned a priority called ranking. Glue information obtained from responses with the AA (Authoritative Answer) flag off has lower priority than information obtained

*4 IIJ Engineers Blog: "Integrating DNSSEC into a DNS Full Resolver Implementation—Proving Negative Responses with NSEC/NSEC3" (https://eng-blog.iij.ad.jp/archives/24512, in Japanese).

from authoritative responses (AA flag on). So the cache entries for the former are overwritten once the latter are obtained. Even if glue information exists in the cache, the full resolver does not return it to the stub resolver but instead returns results only after obtaining authoritative information.

2.6 How Much Control Did We Have?

One of bowline's goals was to enable us to respond quickly when problems were identified. In this section, we present examples of rapid responses that show we have achieved this.

First, let's discuss vulnerabilities. During bowline's development, the operations team told us about attacks related to the number of domain name compression pointers, and we also learned of the KeyTrap attack from external sources. As these vulnerabilities were present in bowline, we immediately fixed them.

As for issues discovered while testing bowline, the operations team reported that communication sometimes failed when a stub resolver used QUIC and HTTP/3 as transport. The Haskell quic library we were using at the time used connected UDP sockets to improve performance. With connected sockets, if an intermediate NAT changes the port number, packets no longer arrive. The quic library has a migration feature enabling it to maintain the connection using a new connected socket if the port changes after a QUIC connection has been established.

However, the NAT that caused this phenomenon was changing the port during the process of establishing a connection. To address this, we added major modifications to make use of unconnected sockets, which are common with UDP usage, although this does result in lower performance.

Further, and this really concerns dug rather than bowline, some encrypted DNS servers were returning multiple TLS session tickets. At the time, the Haskell tls library was implemented on the assumption that there would only be one session ticket. We discovered that when one of the session tickets returned by encrypted DNS servers was selected, session resumption would sometimes fail. Presumably, there are multiple TLS endpoints that all return their session tickets, such that when only one is selected, it may not be what the receiving TLS endpoint expects. So we modified the tls library to use all session tickets.

2.7 Conclusion

The bowline site can be found at the following URL. It includes links to Linux and macOS binaries, instructions for using bowline from Docker Hub, and installation instructions for Debian.

- https://iijlab.github.io/dnsext/bowline.html

The bowline source code can be obtained from GitHub at the following URL. We eventually plan to register it on Hackage, the Haskell package archive, so that users can easily build it using the Haskell build system as well.

- https://github.com/iijlab/dnsext

We provide progress updates on the bowline project at the following URL.

- https://www.iijlab.net/projects/Underpinning/dns.html

Last but not least, I would like to thank my colleagues at IIJ for their insightful comments on draft versions of this article.

Kazuhiko Yamamoto
Development Group Leader, IIJ Research Laboratory
In 2022, Dr. Yamamoto transitioned from full-time to contract employment at IIJ and relocated to his hometown in Yamaguchi Prefecture, where he works remotely. He has recently been chasing Japanese Spanish mackerel and bigfin reef squid in the Seto Inland Sea.