Flexible LDAP load balancing

Ondřej Kuzník, Symas Corp.

October 2017

Why do this?

- Some applications cannot do failover nor load balancing on their own
- Most load balancers on the market connect to backend servers and just pass the stream on
- Backends going unavailable temporarily might not receive enough traffic after they rejoin

You can do better if you decouple the client and backend connections and distribute on a per request basis.

Where we are now

- Prototype has gone through initial load testing at a client
- Round-robin load distribution only
- Work on making it more complete under way
- Plan is to have the full feature set available mid 2018

Where we are now

- Prototype has gone through initial load testing at a client
- Round-robin load distribution only
- Work on making it more complete under way
- Plan is to have the full feature set available mid 2018

Current code at `ITS#8747` awaiting merge

Architecture

- Expect all backends to be practically indistinguishable
- Never resends operations that fail or time out

Architecture

- Expect all backends to be practically indistinguishable
- Never resends operations that fail or time out
- Implement the bare minimum of LDAP:
  - Can see PDUs, only extracts msgid, op type, rest stays opaque
  - Handles Abandon, Unbind, Notice of Disconnection internally
  - Extracts identity from Bind - proxyauthz

Architecture

- Expect all backends to be practically indistinguishable
- Never resends operations that fail or time out
- Implement the bare minimum of LDAP:
  - Can see PDUs, only extracts msgid, op type, rest stays opaque
  - Handles Abandon, Unbind, Notice of Disconnection internally
  - Extracts identity from Bind - proxyauthz
- Rooted in the OpenLDAP project
  - Based in large part on `slapd`
  - Heavy user of `liblber`
  - Infrastructure from `libldap_r`, `liblutil`
  - I/O and timeouts provided by `libevent` (unlike `slapd`)

Upstream connections

- Managed independently of client connections
- Maintain preconfigured amount of connections for each backend
  - Regular connections
  - Bind connections (support for VC Exop available)
- At most 1 connection per backend being set up at any time:

Upstream connections

- Managed independently of client connections
- Maintain preconfigured amount of connections for each backend
  - Regular connections
  - Bind connections (support for VC Exop available)
- At most 1 connection per backend being set up at any time:
  - Name resolution, connect()
  - StartTLS/ldaps
  - Bind
- If we fail, wait before next attempt
- Fully non-blocking

Clients

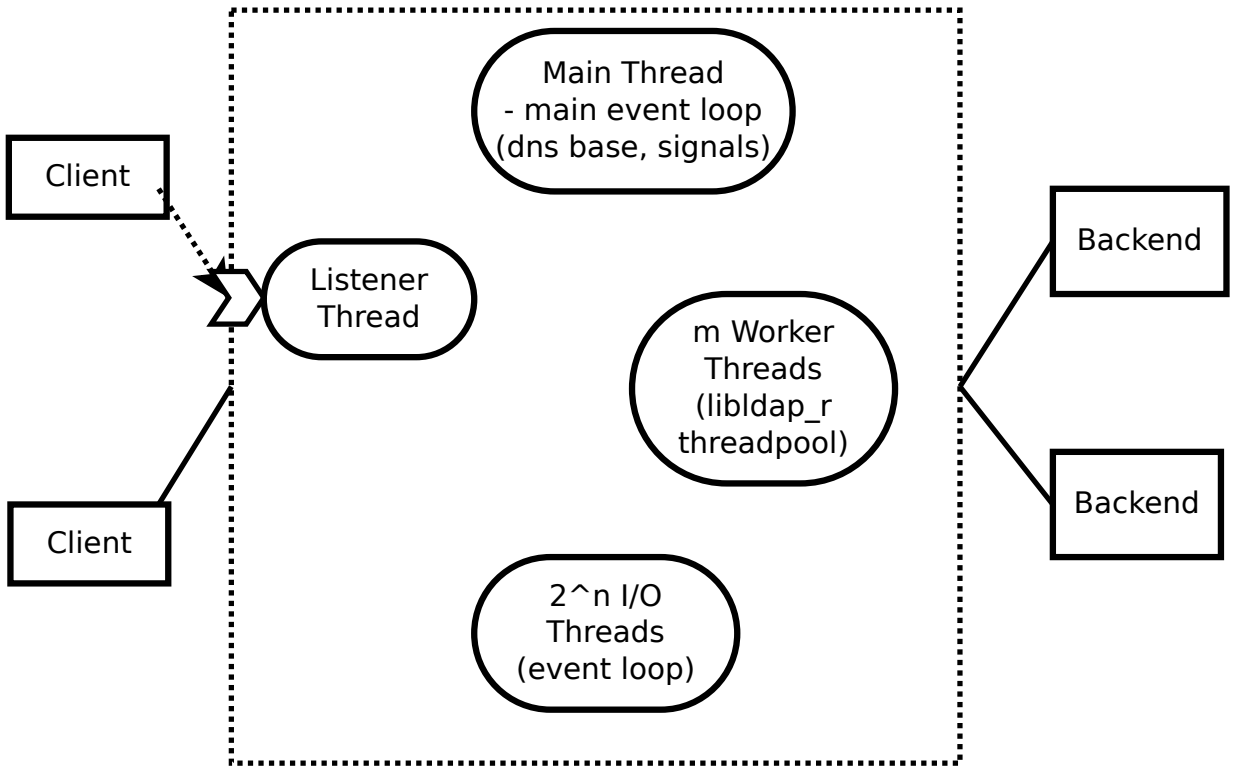
For each connection, we maintain:

- Connection state
- Bind identity
- Pending operations

On a socket read:

- Pull the PDU off the stream
- Defer as a task for further processing
- Suspend read activity on socket

Intermission - Threads



We have a request

A worker thread picks up the task

- Lift Operation data (request, controls)
- Decide whether it needs processing internally
- Otherwise pick an upstream
- Pick a msgid and assemble new PDU

Can add ProxyAuthz control to forwarded operations.

Next, try reading another PDU off the stream - but preserve fairness.

Load management

At the moment, backends are selected

- In a round-robin fashion
- Subject to pending operation limits

An upstream connection is chosen (again round-robin) if

- It is not marked busy (binding, . . .)
- Pending operation count is within limits
- There is no data pending to be written

If no eligible backend/connection is found (overload, downtime), we return `LDAP_UNAVAILABLE` to the client.

Oh, a response

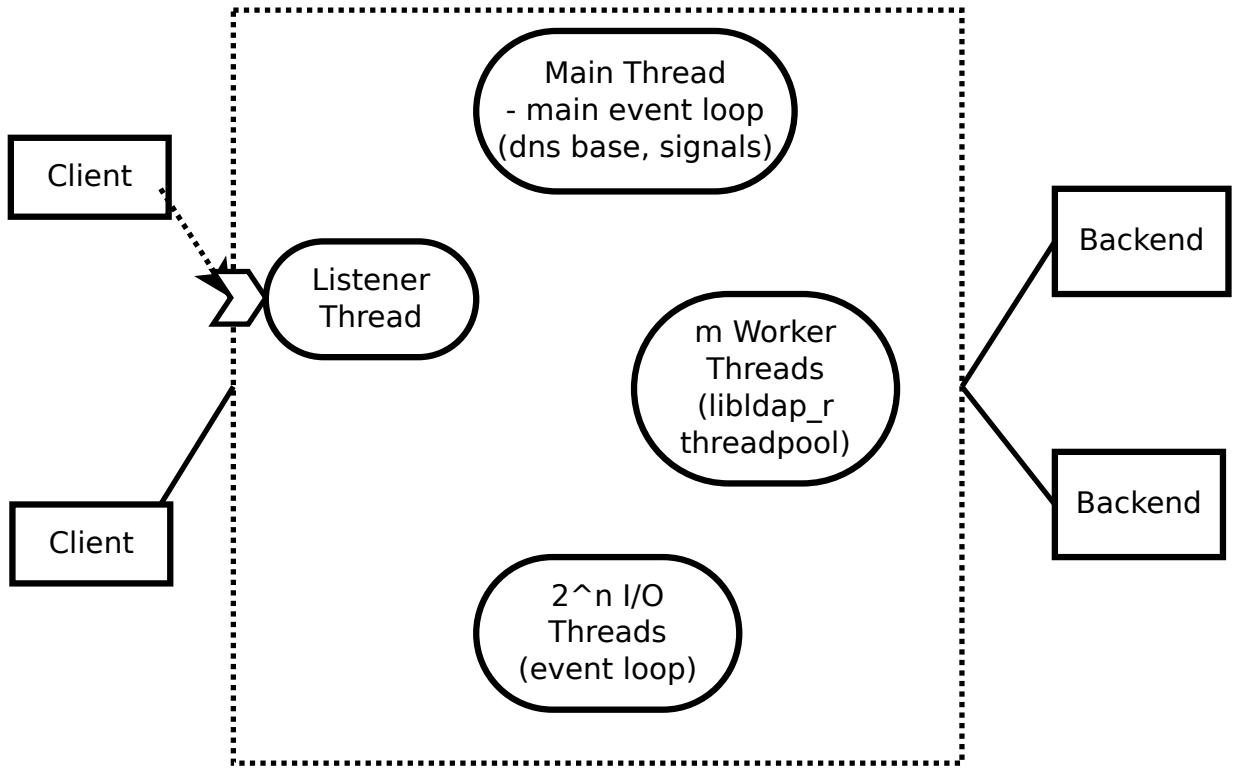
Again, we get a PDU and queue a task:

- It is a response, forward to relevant client
- A final response triggers disposal of `struct Operation`
- Bind response, Unsolicited response

`struct Operation`:

- Links Client and Upstream

Intermission - Running a race course



Intermission - Running a race course

Lifetime management, locking fun

```
struct Connection {
        Sockbuf *c_sb;
        ber_socket_t c_fd;

        ldap_pvt_thread_mutex_t c_mutex;
        unsigned long c_connid;
        int c_refcnt, c_live;
        TAvlnode *c_ops; /* Operation * */

        ldap_pvt_thread_mutex_t c_write_mutex;
        ber_int_t c_next_msgid;
        BerElement *c_pendingber;
        ...
};
```

```
struct Operation {
    ber_int_t o_client_msgid, o_upstream_msgid;
    unsigned long o_client_connid,
                  o_upstream_connid;
    int o_client_refcnt, o_client_live;
    int o_upstream_refcnt, o_upstream_live;
    Connection *o_client, *o_upstream;
    mutex o_link_mutex, o_mutex;
    enum op_state o_freeing;
    ...
};
```

```
struct Operation {
    ber_int_t o_client_msgid, o_upstream_msgid;
    unsigned long o_client_connid,
                  o_upstream_connid;
    int o_client_refcnt, o_client_live;
    int o_upstream_refcnt, o_upstream_live;
    Connection *o_client, *o_upstream;
    mutex o_link_mutex, o_mutex;
    enum op_state o_freeing;
    ...
};

enum op_state {
    SLAP_OP_FREEING_UPSTREAM = 1 << 0,
    SLAP_OP_FREEING_CLIENT = 1 << 1,
    SLAP_OP_DETACHING_UPSTREAM = 1 << 2,
    SLAP_OP_DETACHING_CLIENT = 1 << 3,
};
```

Where next?

- Tiered load balancing
- LDAP Transactions (RFC 5805)
- Online reconfiguration
- `cn=monitor`
- SASL binds (kind of)
- Portability
- Optimise for performance (concurrent resource management, zero-copy, . . .)

Find this useful?

Join us and with your help, we can make it sooner, better.