Evergreen OpenILS

Table of contents

1 Overview

This document contains the summary findings of our review of the Evergreen OpenILS software system. This review was conducted in 2013 with the assistance and testing resources of the MassLNC library consortium. Our analysis was separated into three components of the system: the staff client, the messaging layer, and the underlying relational database.

2 Findings and Recommendations

2.1 Evergreen Staff Client

The current Evergreen Staff Client is composed of a XULRunner application deployed via a Firefox extension and run locally on client machines. The Evergreen community has already formed a consensus around moving away from the XULRunner platform and developing a new client based on modern HTML5 and Javascript technologies that are not tied to a specific browser or delivered as a browser extension.

2.1.1 Migrate From XULRunner

While XULRunner in the past offered a method of developing rich client applications that was not possible, or was significantly more cumbersome, to achieve with plain HTML/CSS/JS, that advantage has diminished outside of a few specific features (e.g. client side file I/O and device access through XPCOM).

Deployment of updates to a XULRunner based staff client requires more coordination, and action by end users or the staff responsible for the maintenance of end user machines. This complication is avoided by a web application, for which updates and fixes can be much more easily deployed in a user-transparent manner.

Modern JS libraries like Dojo, YUI, jQuery, Angular, etc. have almost entirely closed this gap in interface functionality, and in nearly all cases unbind an application developed with them from specific browsers and releases.

2.1.2 Reduce API Round Trips

Wherever possible, minimize the number of repeated calls from the client to the OpenSRF messaging layer – particularly for the same type of information. This problem was most evident on screens in the staff client displaying tables of results (e.g. patron and catalog searches) or associations (e.g. holds and histories for a patron).

Providing set-returning variations of methods, or using the ones already present, can greatly cut the number of separate calls the client needs to make. Take advantage of the *flesh* feature already present to

collect and return related data relevant to the particular Evergreen screen or view, instead of issuing separate API calls to collect the data piecemeal.

2.1.3 Interface Testing

With the likely move to a standard HTML5/JS client interface, implementing a comprehensive testing framework in parallel to that effort may be useful to encourage early and full adoption. There are several options, but the open-source Selenium project offers a featureful and relatively language and platform agnostic system for automated testing within a variety of browsers, and provides an API for the creation and management of individual tests. Additionally, the use of Selenium, or similar, can be tied into the BuildBot continuous-integration suite employed by the Evergreen project.

2.1.4 UI Framework Performance

The current staff client makes use of the Dojo JS framework. Dojo is a comprehensive and full featured UI framework. With that extensive feature list does come a larger footprint if using standard/uncustomized builds. Dojo provides a method to package custom builds, containing only the framework functionality (and associated stylesheets and static content) necessary.

Using these customizable builds to prevent the unnecessary transfer of features and plugins not used by Evergreen will reduce the footprint of the framework files downloaded by the user's client, as well as provide a minified package that is bundled into the smallest number of downloads necessary, thereby reducing the number of HTTP requests issued.

2.2 OpenSRF Messaging Layer

The OpenSRF messaging layer is a Jabber/XMPP based JSON-encapsulating method broker.

2.2.1 Framework Overhead

As observed on the test environment provided by MassLNC, the OpenSRF messaging system does not introduce any significant overhead to method calls. Processing of incoming method requests, dispatching to a worker drone, and the return of the method's results to the caller occur in milliseconds. Network latency between the staff client site and the Evergreen servers, in combination with the time spent within the method itself, are much more likely to contribute significantly to the wallclock time of method calls via OpenSRF.

2.2.2 Jabber Server Point-of-Failure

The Jabber/XMPP server exposes a single point of failure for the messaging layer. While any number of drone servers may continue to function without incident and be capable of processing method calls, if the headnode experiences a failure clients will be unable to transmit method requests to the drones, or receive responses back from them.

2.2.3 *Undispatched Method Calls*

While the drone-worker model employed by OpenSRF enables simple horizontal scaling of capacity, the framework does not appear to provide durability of method calls when drones are not present.

2.2.4 *Testing and Regression Suite*

Regular performance/regression testing against the OpenSRF messaging layer can be employed to identify methods with poor performance profiles. The Evergreen project formerly used Constrictor for some testing, but notes have indicated that Constrictor was replaced by a more continuous-integration focused tool, BuildBot, and that the previous system may not have survived the migration to new version control and issue tracking workflows.

While a CI framework like BuildBot is invaluable for ensuring the correctness of changes, the commentary indicates Constrictor was specifically designed for load/performance testing. Including full coverage of the OpenSRF-exposed methods in a unit test like suite of Constrictor scripts can help to expose the most problematic methods, and protect against what could otherwise be performance regressions during future development.

2.2.5 *Non-performant Methods*

Because the general overhead of the OpenSRF messaging framework is low, the performance of individual methods exposed by OpenSRF can vary quite significantly. It is therefore important at the OpenSRF layer to identify which methods contribute the most to user-perceived performance issues.

Methods which are called more frequently than necessary for different instances of the same type of data may benefit from the addition of set-returning variants or, perhaps, more conscientious use from the client layer; analysis of any database queries or access patterns; reduction in expensive computations that may be candidates for caching, either through direct caching of the method's return values based on inputs, or at a lower level such as selective use of materialized views on complex relational data that has a low to moderate rate of change.

2.3 PostgreSQL Database

The Evergreen OpenILS system uses PostgreSQL as its underlying relational database.

2.3.1 *Query and Stored Procedure Optimizations*

Based on database log performance reporting performed against the production logs of two deployments of Evergreen OpenILS, OmniTI identified the most time-consuming queries. A closer inspection was performed and several optimizations were suggested.

Proposed changes to the function evergreen.ranked_volumes() included inlining nested function calls, removing function calls from the main SELECT that were only referenced inside of the WINDOW, and

converting it to a basic PL/PGSQL functional to provide access to variables, allowing a more efficient call to determining the org unit depth value.

```
CREATE FUNCTION evergreen.ranked_volumes(
    p_bibid bigint,
    p_ouid integer,
    p_depth integer DEFAULT NULL::integer,
    p_slimit public.hstore DEFAULT NULL::public.hstore,
    p_soffset public.hstore DEFAULT NULL::public.hstore,
    p_pref_lib integer DEFAULT NULL::integer,
    p_includes text[] DEFAULT NULL::text[]
) RETURNS TABLE(
    id bigint,
    name text,
    label_sortkey text,
    rank bigint
)
LANGUAGE plpgsql
STABLE
AS $_$
DECLARE
    v_depth int4;
BEGIN
    v_depth := coalesce(
        p_depth,
        (
            SELECT depth
            FROM actor.org_unit_type aout
                INNER JOIN actor.org_unit ou ON ou_type = aout.id
            WHERE ou.id = p_ouid
        ),
        p_pref_lib
    );

    RETURN QUERY
    WITH RECURSIVE descendant_depth AS (
        SELECT  ou.id,
                ou.parent_ou,
                out.depth
        FROM  actor.org_unit ou
                JOIN actor.org_unit_type out ON (out.id = ou.ou_type)
                JOIN anscestor_depth ad ON (ad.id = ou.id)
        WHERE ad.depth = v_depth
            UNION ALL
        SELECT  ou.id,
                ou.parent_ou,
                out.depth
```

```
    FROM  actor.org_unit ou
            JOIN actor.org_unit_type out ON (out.id = ou.ou_type)
            JOIN descendant_depth ot ON (ot.id = ou.parent_ou)
), anscestor_depth AS (
    SELECT  ou.id,
            ou.parent_ou,
            out.depth
    FROM  actor.org_unit ou
            JOIN actor.org_unit_type out ON (out.id = ou.ou_type)
    WHERE ou.id = p_ouid
        UNION ALL
    SELECT  ou.id,
            ou.parent_ou,
            out.depth
    FROM  actor.org_unit ou
            JOIN actor.org_unit_type out ON (out.id = ou.ou_type)
            JOIN anscestor_depth ot ON (ot.parent_ou = ou.id)
), descendants as (
    SELECT ou.* FROM actor.org_unit ou JOIN descendant_depth USING (id)
)

SELECT ua.id, ua.name, ua.label_sortkey, MIN(ua.rank) AS rank FROM (
    SELECT acn.id, aou.name, acn.label_sortkey,
        RANK() OVER w
    FROM asset.call_number acn
        JOIN asset.copy acp ON (acn.id = acp.call_number)
        JOIN descendants AS aou ON (acp.circ_lib = aou.id)
    WHERE acn.record = p_bibid
        AND acn.deleted IS FALSE
        AND acp.deleted IS FALSE
        AND CASE WHEN ('exclude_invisible_acn' = ANY(p_includes)) THEN
            EXISTS (
                SELECT 1
                FROM asset.opac_visible_copies
                WHERE copy_id = acp.id AND record = acn.record
            ) ELSE TRUE END
    GROUP BY acn.id, acp.status, aou.name, acn.label_sortkey, aou.id
    WINDOW w AS (
        ORDER BY
            COALESCE(
                CASE WHEN aou.id = p_ouid THEN -20000 END,
                CASE WHEN aou.id = p_pref_lib THEN -10000 END,
                (SELECT distance - 5000
                    FROM actor.org_unit_descendants_distance(p_pref_lib) as x
                    WHERE x.id = aou.id AND p_pref_lib IN (
                        SELECT q.id FROM actor.org_unit_descendants(p_ouid) as q)),
                (SELECT e.distance FROM actor.org_unit_descendants_distance(p_ouid)
```

```
as e WHERE e.id = aou.id),
                    1000
                ),
                evergreen.rank_cp_status(acp.status)
        )
    ) AS ua
    GROUP BY ua.id, ua.name, ua.label_sortkey
    ORDER BY rank, ua.name, ua.label_sortkey
    LIMIT (p_slimit -> 'acn')::INT
    OFFSET (p_soffset -> 'acn')::INT;
END;
$_$;
```

Another query identified in the log analysis was reviewed and a conversion to a function call proposed. The query in question used asset.call_number and inner joined against asset.copy. The proposed function conversion does away with the inner join and instead looks initially at relevant records in asset.copy, and using a PL/PGSQL cursor steps through them and queries against asset.call_number individually. In the test environment provided by MassLNC, the performance was observed to change from about 13 seconds to under 3 milliseconds.

```
CREATE OR REPLACE FUNCTION asset.some_clever_name( IN p_limit INT4, VARIADIC
p_circ_libs INT4[] )
    RETURNS TABLE ( record INT8, create_date timestamptz ) as $$
DECLARE
    v_results  public.hstore := '';
    v_seen     public.hstore := '';
    v_records  public.hstore := '';
    v_oldest   timestamptz   := NULL;
    v_c_oldest timestamptz   := NULL;
    v_found    INT4 := 0;
    v_circ_lib INT4;
    v_record   int8;
    v_temprec  record;
    v_iter     INT4;
    v_cursor   REFCURSOR;
BEGIN
    FOREACH v_circ_lib IN ARRAY p_circ_libs LOOP
        v_iter := 0;
        v_seen := '';
        v_c_oldest := NULL;
        open v_cursor NO SCROLL FOR
            SELECT c.call_number, c.create_date
            FROM asset.copy c
            WHERE c.circ_lib = v_circ_lib AND NOT c.deleted
            ORDER BY c.create_date DESC;

        LOOP
```

```
            FETCH v_cursor INTO v_temprec;
            EXIT WHEN NOT FOUND;

            v_iter := v_iter + 1;

            -- If we already have better data than current row (newer records in
            EXIT WHEN v_oldest IS NOT NULL AND v_oldest >= v_temprec.create_date;

            -- Ignore if we've seen given call number in current query (for current
circ_lib)
            CONTINUE WHEN v_seen ? v_temprec.call_number::TEXT;
            v_seen := v_seen || public.hstore( v_temprec.call_number::TEXT, '1' );

            -- If we don't have yet record for given call_number, we need to get it
            IF v_records ? v_temprec.call_number::TEXT THEN
                v_record := v_records -> v_temprec.call_number::TEXT;
            ELSE
                SELECT cn.record INTO v_record FROM asset.call_number cn WHERE cn.id =
v_temprec.call_number;
                CONTINUE WHEN NOT FOUND;
                CONTINUE WHEN v_record <= 0;
                v_records := v_records || hstore( v_temprec.call_number::TEXT,
v_record::TEXT );
            END IF;

            -- If results already contain "better" date for given record, next row
            IF v_results ? v_record::TEXT THEN
                CONTINUE WHEN ( v_results -> v_record::TEXT )::timestamptz >
v_temprec.create_date;
            END IF;

            v_found := v_found + 1;
            v_results  := v_results || hstore( v_record::TEXT,
v_temprec.create_date::TEXT );

            IF v_c_oldest IS NULL OR v_c_oldest > v_temprec.create_date THEN
                v_c_oldest := v_temprec.create_date;
            END IF;

            EXIT WHEN v_found = p_limit;
        END LOOP;

        CLOSE v_cursor;

        -- Update oldest information based on oldest row added in current loop
        IF v_oldest IS NULL OR v_oldest < v_c_oldest THEN
            v_oldest := v_c_oldest;
```

```
        END IF;

    END LOOP;
    RETURN QUERY SELECT KEY::INT8, value::timestamptz FROM each(v_results) ORDER BY
value::timestamptz DESC LIMIT p_limit;
RETURN;
END;
$$ language plpgsql STRICT;
```

2.3.2 Trigger Optimizations

Aside from the triggers employed by Slony-I for replication purposes, Evergreen itself makes extensive use of row-level triggers. At least one of these fires off frequently enough that it was reported as a significant time consumer in the production log performance reports. Upon further inspection it was seen to be a PL/PerlU function, and most invocations were quite fast, but the trigger function incurred an additional startup cost the first time it was used on a newly started PostgreSQL backend process due to external Perl module dependencies. This startup cost doubled the trigger function's run time and led to an inconsistent performance profile.

Proposed remedies were to either accept the initial cost per database backend; to allow the use of a connection pooler like PgBouncer to potentially greatly reduce the number of times the cost is incurred; or to pre-load these Perl module dependencies in the PostgreSQL server configuration.

2.3.3 Connection Pooling

Encourage the use of a database connection pooler, such as PgBouncer. This can help to reduce the initialization cost of PL/Perl or similar functions which need to be compiled before use in each new database backend, particularly when those functions contain external module dependencies. A pooler such as PgBouncer also opens up more transparent options for scaling read-only database loads horizontally.

2.3.4 Tracking PostgreSQL Releases

MassLNC production installations are currently two releases behind the stable version of PostgreSQL. Testing Evergreen compatibility with the latest stable releases and providing straightforward upgrade paths/tools for Evergreen administrators would permit the performance and stability enhancements in most new versions of PostgreSQL to be more rapidly adopted.

2.3.5 Evaluate Alternate Database Replication Options

Slony-I is currently in production at some consortium members. While it is a robust, and in some cases (e.g. minimal downtime upgrades between major releases) ideal, replication system, the fact that it is entirely trigger-based can negatively impact performance; as well as introduce additional maintenance

and upgrade burdens on administrators when any schema changes occur, as those require special handling with Slony-I's trigger model.

Modern PostgreSQL releases, including those in production within MassLNC, support streaming replication which imposes very minimal overhead on the primary/read-write database server, and transparently ships schema modifications without the need for additional tools, processes, or configuration, as is the case with Slony-I.

2.3.6 Performance Monitoring and Tuning

A review of the production PostgreSQL configurations for MassLNC consortium members revealed some tuning had already been performed. However, query logging was disabled, which makes it difficult to measure the actual impact of any changes. It is recommended that regular monitoring and performance reporting be performed, particularly when new releases are deployed. This monitoring should not be limited to test or staging installs, but should be performed whenever possible on live, production systems to best capture the performance profiles of the Evergreen system under load.

OmniTI provided several recommended configuration changes which result in PostgreSQL server logs that can be parsed accurately by tools such as pgbadger.