Easing gently into OpenSRF

Dan Scott <dscott@laurentian.ca>

Revision History

Revision 1.0	June 2010	DS

Table of Contents

The Evergreen open-source library system serves library consortia composed of hundreds of branches with millions of patrons - for example, the Georgia Public Library Service PINES system [http://www.georgialibraries.org/statelibrarian/bythenumbers.pdf]. One of the claimed advantages of Evergreen over alternative integrated library systems is the underlying Open Service Request Framework (OpenSRF, pronounced "open surf") architecture. This article introduces OpenSRF, demonstrates how to build OpenSRF services through simple code examples, and explains the technical foundations on which OpenSRF is built.

1. Introducing OpenSRF

OpenSRF is a message routing network that offers scalability and failover support for individual services and entire servers with minimal development and deployment overhead. You can use OpenSRF to build loosely-coupled applications that can be deployed on a single server or on clusters of geographically distributed servers using the same code and minimal configuration changes. Although copyright statements on some of the OpenSRF code date back to Mike Rylander’s original explorations in 2000, Evergreen was the first major application to be developed with, and to take full advantage of, the OpenSRF architecture starting in 2004. The first official release of OpenSRF was 0.1 in February 2005 (http://evergreen-ils.org/blog/?p=21), but OpenSRF’s development continues a steady pace of enhancement and refinement, with the release of 1.0.0 in October 2008 and the most recent release of 1.2.2 in February 2010.

OpenSRF is a distinct break from the architectural approach used by previous library systems and has more in common with modern Web applications. The traditional "scale-up" approach to serve more transactions is to purchase a server with more CPUs and more RAM, possibly splitting the load between a Web server, a database server, and a business logic server. Evergreen, however, is built on the Open Service Request Framework (OpenSRF) architecture, which firmly embraces the "scale-out" approach of spreading transaction load over cheap commodity servers. The initial GPLS PINES hardware cluster [http://evergreen-ils.org/blog/?p=56], while certainly impressive, may have offered the misleading impression that Evergreen requires a lot of hardware to run. However, Evergreen and OpenSRF easily scale down to a single server; many Evergreen libraries run their entire library system on a single server, and most OpenSRF and Evergreen development occurs on a virtual machine running on a single laptop or desktop image.

Another common concern is that the flexibility of OpenSRF’s distributed architecture makes it complex to configure and to write new applications. This article demonstrates that OpenSRF itself is an extremely simple architecture on which one can easily build applications of many kinds – not just library applications – and that you can use a number of different languages to call and implement OpenSRF methods with a minimal learning curve. With an application built on OpenSRF, when you identify a bottleneck in your application’s business logic layer, you can adjust the number of the processes serving that particular bottleneck on each of your servers; or if the problem is that your service is resource-hungry, you could add an inexpensive server to your cluster and dedicate it to running that resource-hungry service.

1.1. Programming language support

If you need to develop an entirely new OpenSRF service, you can choose from a number of different languages in which to implement that service. OpenSRF client language bindings have been written for C, Java, JavaScript, Perl, and Python, and service language bindings have been written for C, Perl, and Python. This article uses Perl examples as a lowest common denominator programming language. Writing an OpenSRF binding for another language is a relatively small task if that language offers libraries that support the core technologies on which OpenSRF depends:

- Extensible Messaging and Presence Protocol [http://tools.ietf.org/html/rfc3920] (XMPP, sometimes referred to as Jabber) - provides the base messaging infrastructure between OpenSRF clients and services
- JavaScript Object Notation [http://json.org] (JSON) - serializes the content of each XMPP message in a standardized and concise format

- memcached [http://memcached.org] - provides the caching service
- syslog [http://tools.ietf.org/html/rfc5424] - the standard UNIX logging service

Unfortunately, the OpenSRF reference documentation [http://evergreen-ils.org/dokuwiki/doku.php?id=osrf-devel:primer], although augmented by the OpenSRF glossary [http://evergreen-ils.org/dokuwiki/doku.php?id=osrf-devel:terms], blog posts like the description of OpenSRF and Jabber [http://evergreen-ils.org/blog/?p=36], and even this article, is not a sufficient substitute for a complete specification on which one could implement a language binding. The recommended option for would-be developers of another language binding is to use the Python implementation as the cleanest basis for a port to another language.

2. Enough jibber-jabber: writing an OpenSRF service

Imagine an application architecture in which 10 lines of Perl or Python, using the data types native to each language, are enough to implement a method that can then be deployed and invoked seamlessly across hundreds of servers. You have just imagined developing with OpenSRF – it is truly that simple. Under the covers, of course, the OpenSRF language bindings do an incredible amount of work on behalf of the developer. An OpenSRF application consists of one or more OpenSRF services that expose methods: for example, the `opensrf.simple-text` demonstration service [http://svn.open-ils.org/trac/OpenSRF/browser/trunk/src/perl/lib/OpenSRF/Application/Demo/SimpleText.pm] exposes the `opensrf.simple-text.split()` and `opensrf.simple-text.reverse()` methods. Each method accepts zero or more arguments and returns zero or one results. The data types supported by OpenSRF arguments and results are typical core language data types: strings, numbers, booleans, arrays, and hashes.

To implement a new OpenSRF service, perform the following steps:

1. Include the base OpenSRF support libraries
2. Write the code for each of your OpenSRF methods as separate procedures
3. Register each method
4. Add the service definition to the OpenSRF configuration files

For example, the following code implements an OpenSRF service. The service includes one method named `opensrf.simple-text.reverse()` that accepts one string as input and returns the reversed version of that string:

```
#!/usr/bin/perl

package OpenSRF::Application::Demo::SimpleText;

use strict;

use OpenSRF::Application;
use parent qw/OpenSRF::Application/;
```

```
sub text_reverse {
    my ($self , $conn, $text) = @_;
    my $reversed_text = scalar reverse($text);
    return $reversed_text;
}

__PACKAGE__->register_method(
    method    => 'text_reverse',
    api_name  => 'opensrf.simple-text.reverse'
);
```

Ten lines of code, and we have a complete OpenSRF service that exposes a single method and could be deployed quickly on a cluster of servers to meet your application’s ravenous demand for reversed strings! If you’re unfamiliar with Perl, the `use OpenSRF::Application; use parent qw/OpenSRF::Application/;` lines tell this package to inherit methods and properties from the `OpenSRF::Application` module. For example, the call to `__PACKAGE__->register_method()` is defined in `OpenSRF::Application` but due to inheritance is available in this package (named by the special Perl symbol `__PACKAGE__` that contains the current package name). The `register_method()` procedure is how we introduce a method to the rest of the OpenSRF world.

2.1. Registering a service with the OpenSRF configuration files

Two files control most of the configuration for OpenSRF:

- `opensrf.xml` contains the configuration for the service itself, as well as a list of which application servers in your OpenSRF cluster should start the service.
- `opensrf_core.xml` (often referred to as the "bootstrap configuration" file) contains the OpenSRF networking information, including the XMPP server connection credentials for the public and private routers. You only need to touch this for a new service if the new service needs to be accessible via the public router.

Begin by defining the service itself in `opensrf.xml`. To register the `opensrf.simple-text` service, add the following section to the `<apps>` element (corresponding to the XPath `/opensrf/default/apps/`):

```
<apps>
  <opensrf.simple-text>                                                       <!-- 1 -->
    <keepalive>3</keepalive>                                                  <!-- 2 -->
    <stateless>1</stateless>                                                  <!-- 3 -->
    <language>perl</language>                                                 <!-- 4 -->
    <implementation>OpenSRF::Application::Demo::SimpleText</implementation>  <!-- 5 -->
    <max_requests>100</max_requests>                                          <!-- 6 -->
    <unix_config>
      <max_requests>1000</max_requests>                                       <!-- 7 -->
      <unix_log>opensrf.simple-text_unix.log</unix_log>                       <!-- 8 -->
      <unix_sock>opensrf.simple-text_unix.sock</unix_sock>                    <!-- 9 -->
      <unix_pid>opensrf.simple-text_unix.pid</unix_pid>                      <!-- 10 -->
      <min_children>5</min_children>                                         <!-- 11 -->
      <max_children>15</max_children>                                        <!-- 12 -->
      <min_spare_children>2</min_spare_children>                             <!-- 13 -->
      <max_spare_children>5</max_spare_children>                             <!-- 14 -->
    </unix_config>
  </opensrf.simple-text>

  <!-- other OpenSRF services registered here... -->
</apps>
```

1 The element name is the name that the OpenSRF control scripts use to refer to the service.
2 The `<keepalive>` element specifies the interval (in seconds) between checks to determine if the service is still running.
3 The `<stateless>` element specifies whether OpenSRF clients can call methods from this service without first having to create a connection to a specific service backend process for that service. If the value is `1`, then the client can simply issue a request and the router will forward the request to an available service and the result will be returned directly to the client.
4 The `<language>` element specifies the programming language in which the service is implemented.
5 The `<implementation>` element pecifies the name of the library or module in which the service is implemented.
6 (C implementations only): The `<max_requests>` element, as a direct child of the service element name, specifies the maximum number of requests a process serves before it is killed and replaced by a new process.
7 (Perl implementations only): The `<max_requests>` element, as a direct child of the `<unix_config>` element, specifies the maximum number of requests a process serves before it is killed and replaced by a new process.
8 The `<unix_log>` element specifies the name of the log file for language-specific log messages such as syntax warnings.
9 The `<unix_sock>` element specifies the name of the UNIX socket used for inter-process communications.
10 The `<unix_pid>` element specifies the name of the PID file for the master process for the service.
11 The `<min_children>` element specifies the minimum number of child processes that should be running at any given time.
12 The `<max_children>` element specifies the maximum number of child processes that should be running at any given time.
13 The `<min_spare_children>` element specifies the minimum number of idle child processes that should be available to handle incoming requests. If there are fewer than this number of spare child processes, new processes will be spawned.
14 The`<max_spare_children>` element specifies the maximum number of idle child processes that should be available to handle incoming requests. If there are more than this number of spare child processes, the extra processes will be killed.

To make the service accessible via the public router, you must also edit the `opensrf_core.xml` configuration file to add the service to the list of publicly accessible services:

```
<router>                                                                <!-- 1 -->
    <!-- This is the public router. On this router, we only register applications
     which should be accessible to everyone on the opensrf network -->
    <name>router</name>
    <domain>public.localhost</domain>                                   <!-- 2 -->
    <services>
        <service>opensrf.math</service>
        <service>opensrf.simple-text</service>                          <!-- 3 -->
    </services>
</router>
```

1. This section of the `opensrf_core.xml` file is located at XPath `/config/opensrf/routers/`.
2. `public.localhost` is the canonical public router domain in the OpenSRF installation instructions.
3. Each `<service>` element contained in the `<services>` element offers their services via the public router as well as the private router.

Once you have defined the new service, you must restart the OpenSRF Router to retrieve the new configuration and start or restart the service itself.

Complete working examples of the opensrf_core.xml and opensrf.xml configuration files are included with this article for your reference.

2.2. Calling an OpenSRF method

OpenSRF clients in any supported language can invoke OpenSRF services in any supported language. So let’s see a few examples of how we can call our fancy new `opensrf.simple-text.reverse()` method:

2.2.1. Calling OpenSRF methods from the srfsh client

`srfsh` is a command-line tool installed with OpenSRF that you can use to call OpenSRF methods. To call an OpenSRF method, issue the `request` command and pass the OpenSRF service and method name as the first two arguments; then pass one or more JSON objects delimited by commas as the arguments to the method being invoked.

The following example calls the `opensrf.simple-text.reverse` method of the `opensrf.simple-text` OpenSRF service, passing the string `"foobar"` as the only method argument:

```
$ srfsh
srfsh # request opensrf.simple-text opensrf.simple-text.reverse "foobar"

Received Data: "raboof"

=------------------------------------
Request Completed Successfully
Request Time in seconds: 0.016718
=------------------------------------
```

2.2.2. Getting documentation for OpenSRF methods from the srfsh client

The `srfsh` client also gives you command-line access to retrieving metadata about OpenSRF services and methods. For a given OpenSRF method, for example, you can retrieve information such as the minimum number of required arguments, the data type and a description of each argument, the package or library in which the method is implemented, and a description of the method. To retrieve the documentation for an opensrf method from `srfsh`, issue the `introspect` command, followed by the name of the OpenSRF service and (optionally) the name of the OpenSRF method. If you do not pass a method name to the `introspect` command, `srfsh` lists all of the methods offered by the service. If you pass a partial method name, `srfsh` lists all of the methods that match that portion of the method name.

Note

The quality and availability of the descriptive information for each method depends on the developer to register the method with complete and accurate information. The quality varies across the set of OpenSRF and Evergreen APIs, although some effort is being put towards improving the state of the internal documentation.

```
srfsh# introspect opensrf.simple-text "opensrf.simple-text.reverse"
--> opensrf.simple-text

Received Data: {
  "__c":"opensrf.simple-text",
  "__p":{
    "api_level":1,
    "stream":0,                                                       \ # 1
    "object_hint":"OpenSRF_Application_Demo_SimpleText",
    "remote":0,
    "package":"OpenSRF::Application::Demo::SimpleText",               \ # 2
    "api_name":"opensrf.simple-text.reverse",                         \ # 3
    "server_class":"opensrf.simple-text",
    "signature":{                                                     \ # 4
      "params":[                                                      \ # 5
        {
          "desc":"The string to reverse",
          "name":"text",
          "type":"string"
        }
      ],
      "desc":"Returns the input string in reverse order\n",           \ # 6
      "return":{                                                      \ # 7
        "desc":"Returns the input string in reverse order",
        "type":"string"
      }
    },
    "method":"text_reverse",                                          \ # 8
    "argc":1                                                          \ # 9
  }
}
```

1. `stream` denotes whether the method supports streaming responses or not.
2. `package` identifies which package or library implements the method.
3. `api_name` identifies the name of the OpenSRF method.
4. `signature` is a hash that describes the parameters for the method.
5. `params` is an array of hashes describing each parameter in the method; each parameter has a description (`desc`), name (`name`), and type (`type`).
6. `desc` is a string that describes the method itself.
7. `return` is a hash that describes the return value for the method; it contains a description of the return value (`desc`) and the type of the returned value (`type`).
8. `method` identifies the name of the function or method in the source implementation.
9. `argc` is an integer describing the minimum number of arguments that must be passed to this method.

2.2.3. Calling OpenSRF methods from Perl applications

To call an OpenSRF method from Perl, you must connect to the OpenSRF service, issue the request to the method, and then retrieve the results.

```
#/usr/bin/perl
use strict;
use OpenSRF::AppSession;
use OpenSRF::System;

OpenSRF::System->bootstrap_client(config_file => '/openils/conf/opensrf_core.xml'); # 1

my $session = OpenSRF::AppSession->create("opensrf.simple-text");                  # 2

print "substring: Accepts a string and a number as input, returns a string\n";
my $result = $session->request("opensrf.simple-text.substring", "foobar", 3);      # 3
my $request = $result->gather();                                                    # 4
print "Substring: $request\n\n";

print "split: Accepts two strings as input, returns an array of strings\n";
$request = $session->request("opensrf.simple-text.split", "This is a test", " ");   # 5
my $output = "Split: [";
my $element;
while ($element = $request->recv()) {                                               # 6
    $output .= $element->content . ", ";                                            # 7
}
$output =~ s/, $/]/;
print $output . "\n\n";

print "statistics: Accepts an array of strings as input, returns a hash\n";
my @many_strings = [
    "First I think I'll have breakfast",
    "Then I think that lunch would be nice",
    "And then seventy desserts to finish off the day"
];

$result = $session->request("opensrf.simple-text.statistics", @many_strings);      # 8
$request = $result->gather();                                                       # 9
print "Length: " . $result->{'length'} . "\n";
print "Word count: " . $result->{'word_count'} . "\n";

$session->disconnect();                                                             # 10
```

1 The `OpenSRF::System->bootstrap_client()` method reads the OpenSRF configuration information from the indicated file and creates an XMPP client connection based on that information.

2 The `OpenSRF::AppSession->create()` method accepts one argument - the name of the OpenSRF service to which you want to want to make one or more requests - and returns an object prepared to use the client connection to make those requests.

3 The `OpenSRF::AppSession->request()` method accepts a minimum of one argument - the name of the OpenSRF method to which you want to make a request - followed by zero or more arguments to pass to the OpenSRF method as input values. This example passes a string and an integer to the `opensrf.simple-text.substring` method defined by the `opensrf.simple-text` OpenSRF service.

4 The `gather()` method, called on the result object returned by the `request()` method, iterates over all of the possible results from the result object and returns a single variable.

5 This `request()` call passes two strings to the `opensrf.simple-text.split` method defined by the `opensrf.simple-text` OpenSRF service and returns (via `gather()`) a reference to an array of results.

6 The `opensrf.simple-text.split()` method is a streaming method that returns an array of results with one element per `recv()` call on the result object. We could use the `gather()`

method to retrieve all of the results in a single array reference, but instead we simply iterate over the result variable until there are no more results to retrieve.

7 While the `gather()` convenience method returns only the content of the complete set of results for a given request, the `recv()` method returns an OpenSRF result object with `status`, `statusCode`, and `content` fields as we saw in the HTTP results example.

8 This `request()` call passes an array to the `opensrf.simple-text.statistics` method defined by the `opensrf.simple-text` OpenSRF service.

9 The result object returns a hash reference via `gather()`. The hash contains the `length` and `word_count` keys we defined in the method.

10 The `OpenSRF::AppSession->disconnect()` method closes the XMPP client connection and cleans up resources associated with the session.

2.3. Accepting and returning more interesting data types

Of course, the example of accepting a single string and returning a single string is not very interesting. In real life, our applications tend to pass around multiple arguments, including arrays and hashes. Fortunately, OpenSRF makes that easy to deal with; in Perl, for example, returning a reference to the data type does the right thing. In the following example of a method that returns a list, we accept two arguments of type string: the string to be split, and the delimiter that should be used to split the string.

```
sub text_split {
    my $self = shift;
    my $conn = shift;
    my $text = shift;
    my $delimiter = shift || ' ';

    my @split_text = split $delimiter, $text;
    return \@split_text;
}

__PACKAGE__->register_method(
    method    => 'text_split',
    api_name  => 'opensrf.simple-text.split'
);
```

We simply return a reference to the list, and OpenSRF does the rest of the work for us to convert the data into the language-independent format that is then returned to the caller. As a caller of a given method, you must rely on the documentation used to register to determine the data structures - if the developer has added the appropriate documentation.

2.4. Accepting and returning Evergreen objects

OpenSRF is agnostic about objects; its role is to pass JSON back and forth between OpenSRF clients and services, and it allows the specific clients and services to define their own semantics for the JSON structures. On top of that infrastructure, Evergreen offers the fieldmapper: an object-relational mapper that provides a complete definition of all objects, their properties, their relationships to other objects, the permissions required to create, read, update, or delete objects of that type, and the database table or view on which they are based.

The Evergreen fieldmapper offers a great deal of convenience for working with complex system objects beyond the basic mapping of classes to database schemas. Although the result is passed over the wire as a JSON object containing the indicated fields, fieldmapper-aware clients then turn those JSON objects into native objects with setter / getter methods for each field.

All of this metadata about Evergreen objects is defined in the fieldmapper configuration file (`/openils/conf/fm_IDL.xml`), and access to these classes is provided by the `open-ils.cstore`, `open-ils.pcrud`, and `open-ils.reporter-store` OpenSRF services which parse the fieldmapper configuration file and dynamically register OpenSRF methods for creating, reading, updating, and deleting all of the defined classes.

```
<class id="mous" controller="open-ils.cstore open-ils.pcrud"
 oils_obj:fieldmapper="money::open_user_summary"
 oils_persist:tablename="money.open_usr_summary"
 reporter:label="Open User Summary">                                <!-- 1 -->
    <fields oils_persist:primary="usr" oils_persist:sequence="">    <!-- 2 -->
        <field name="balance_owed" reporter:datatype="money" />     <!-- 3 -->
        <field name="total_owed" reporter:datatype="money" />
        <field name="total_paid" reporter:datatype="money" />
        <field name="usr" reporter:datatype="link"/>
    </fields>
    <links>
        <link field="usr" reltype="has_a" key="id" map="" class="au"/>     <!-- 4 -->
    </links>
    <permacrud xmlns="http://open-ils.org/spec/opensrf/IDL/permacrud/v1">  <!-- 5 -->
        <actions>
            <retrieve permission="VIEW_USER">                          <!-- 6 -->
                <context link="usr" field="home_ou"/>                  <!-- 7 -->
            </retrieve>
        </actions>
    </permacrud>
</class>
```

1 The `<class>` element defines the class:

- The `id` attribute defines the *class hint* that identifies the class both elsewhere in the fieldmapper configuration file, such as in the value of the `field` attribute of the `<link>` element, and in the JSON object itself when it is instantiated. For example, an "Open User Summary" JSON object would have the top level property of `"__c":"mous"`.
- The `controller` attribute identifies the services that have direct access to this class. If `open-ils.pcrud` is not listed, for example, then there is no means to directly access members of this class through a public service.
- The `oils_obj:fieldmapper` attribute defines the name of the Perl fieldmapper class that will be dynamically generated to provide setter and getter methods for instances of the class.
- The `oils_persist:tablename` attribute identifies the schema name and table name of the database table that stores the data that represents the instances of this class. In this case, the schema is `money` and the table is `open_usr_summary`.
- The `reporter:label` attribute defines a human-readable name for the class used in the reporting interface to identify the class. These names are defined in English in the fieldmapper configuration file; however, they are extracted so that they can be translated and served in the user’s language of choice.

2 The `<fields>` element lists all of the fields that belong to the object.

- The `oils_persist:primary` attribute identifies the field that acts as the primary key for the object; in this case, the field with the name `usr`.
- The `oils_persist:sequence` attribute identifies the sequence object (if any) in this database provides values for new instances of this class. In this case, the primary key is defined by a field that is linked to a different table, so no sequence is used to populate these instances.

3 Each `<field>` element defines a single field with the following attributes:

- The `name` attribute identifies the column name of the field in the underlying database table as well as providing a name for the setter / getter method that can be invoked in the JSON or native version of the object.
- The `reporter:datatype` attribute defines how the reporter should treat the contents of the field for the purposes of querying and display.
- The `reporter:label` attribute can be used to provide a human-readable name for each field; without it, the reporter falls back to the value of the `name` attribute.

4 The `<links>` element contains a set of zero or more `<link>` elements, each of which defines a relationship between the class being described and another class.

- The `field` attribute identifies the field named in this class that links to the external class.
- The `reltype` attribute identifies the kind of relationship between the classes; in the case of `has_a`, each value in the `usr` field is guaranteed to have a corresponding value in the external class.
- The `key` attribute identifies the name of the field in the external class to which this field links.
- The rarely-used `map` attribute identifies a second class to which the external class links; it enables this field to define a direct relationship to an external class with one degree of separation, to avoid having to retrieve all of the linked members of an intermediate class just to retrieve the instances from the actual desired target class.
- The `class` attribute identifies the external class to which this field links.

5 The `<permacrud>` element defines the permissions that must have been granted to a user to operate on instances of this class.

6 The `<retrieve>` element is one of four possible children of the `<actions>` element that define the permissions required for each action: create, retrieve, update, and delete.

- The `permission` attribute identifies the name of the permission that must have been granted to the user to perform the action.
- The `contextfield` attribute, if it exists, defines the field in this class that identifies the library within the system for which the user must have prvileges to work. If a user has been granted a given permission, but has not been granted privileges to work at a given library, they can not perform the action at that library.

7 The rarely-used `<context>` element identifies a linked field (`link` attribute) in this class which links to an external class that holds the field (`field` attribute) that identifies the library within the system for which the user must have privileges to work.

When you retrieve an instance of a class, you can ask for the result to *flesh* some or all of the linked fields of that class, so that the linked instances are returned embedded directly in your requested instance. In that same request you can ask for the fleshed instances to in turn have their linked fields fleshed. By bundling all of this into a single request and result sequence, you can avoid the network overhead of requiring the client to request the base object, then request each linked object in turn.

You can also iterate over a collection of instances and set the automatically generated `isdeleted`, `isupdated`, or `isnew` properties to indicate that the given instance has been deleted, updated, or created respectively. Evergreen can then act in batch mode over the collection to perform the requested actions on any of the instances that have been flagged for action.

2.5. Returning streaming results

In the previous implementation of the `opensrf.simple-text.split` method, we returned a reference to the complete array of results. For small values being delivered over the network, this is perfectly acceptable, but for large sets of values this can pose a number of problems for the requesting client. Consider a service that returns a set of bibliographic records in response to a query like "all records edited in the past month"; if the underlying database is relatively active, that could result in thousands of records being returned as a single network request. The client would be forced to block until all of the results are returned, likely resulting in a significant delay, and depending on the implementation, correspondingly large amounts of memory might be consumed as all of the results are read from the network in a single block.

OpenSRF offers a solution to this problem. If the method returns results that can be divided into separate meaningful units, you can register the OpenSRF method as a streaming method and enable the client to loop over the results one unit at a time until the method returns no further results. In addition to registering the method with the provided name, OpenSRF also registers an additional method with `.atomic` appended to the method name. The `.atomic` variant gathers all of the results into a single block to return to the client, giving the caller the ability to choose either streaming or atomic results from a single method definition.

In the following example, the text splitting method has been reimplemented to support streaming; very few changes are required:

```
sub text_split {
    my $self = shift;
    my $conn = shift;
    my $text = shift;
    my $delimiter = shift || ' ';

    my @split_text = split $delimiter, $text;
    foreach my $string (@split_text) {           # 1
        $conn->respond($string);
    }
    return undef;
}

__PACKAGE__->register_method(
    method    => 'text_split',
    api_name  => 'opensrf.simple-text.split',
    stream    => 1                              # 2
);
```

1 Rather than returning a reference to the array, a streaming method loops over the contents of the array and invokes the `respond()` method of the connection object on each element of the array.

2 Registering the method as a streaming method instructs OpenSRF to also register an atomic variant (`opensrf.simple-text.split.atomic`).

2.6. Error! Warning! Info! Debug!

As hard as it may be to believe, it is true: applications sometimes do not behave in the expected manner, particularly when they are still under development. The service language bindings for OpenSRF include integrated support for logging messages at the levels of ERROR, WARNING, INFO, DEBUG, and the extremely verbose INTERNAL to either a local file or to a syslogger service. The destination of the log files, and the level of verbosity to be logged, is set in the `opensrf_core.xml` configuration file. To add logging to our Perl example, we just have to add the `OpenSRF::Utils::Logger` package to our list of used Perl modules, then invoke the logger at the desired logging level.

You can include many calls to the OpenSRF logger; only those that are higher than your configured logging level will actually hit the log. The following example exercises all of the available logging levels in OpenSRF:

```
use OpenSRF::Utils::Logger;
my $logger = OpenSRF::Utils::Logger;
# some code in some function
{
    $logger->error("Hmm, something bad DEFINITELY happened!");
    $logger->warn("Hmm, something bad might have happened.");
    $logger->info("Something happened.");
    $logger->debug("Something happened; here are some more details.");
    $logger->internal("Something happened; here are all the gory details.")
}
```

If you call the mythical OpenSRF method containing the preceding OpenSRF logger statements on a system running at the default logging level of INFO, you will only see the INFO, WARN, and ERR messages, as follows:

Example 1. Results of logging calls at the default level of INFO

```
[2010-03-17 22:27:30] opensrf.simple-text [ERR :5681:SimpleText.pm:277:] Hmm, something
[2010-03-17 22:27:30] opensrf.simple-text [WARN:5681:SimpleText.pm:278:] Hmm, something
[2010-03-17 22:27:30] opensrf.simple-text [INFO:5681:SimpleText.pm:279:] Something happe
```

If you then increase the the logging level to INTERNAL (5), the logs will contain much more information, as follows:

Example 2. Results of logging calls at the default level of INTERNAL

```
[2010-03-17 22:48:11] opensrf.simple-text [ERR :5934:SimpleText.pm:277:] Hmm, something 
[2010-03-17 22:48:11] opensrf.simple-text [WARN:5934:SimpleText.pm:278:] Hmm, something 
[2010-03-17 22:48:11] opensrf.simple-text [INFO:5934:SimpleText.pm:279:] Something happe
[2010-03-17 22:48:11] opensrf.simple-text [DEBG:5934:SimpleText.pm:280:] Something happe
[2010-03-17 22:48:11] opensrf.simple-text [INTL:5934:SimpleText.pm:281:] Something happe
[2010-03-17 22:48:11] opensrf.simple-text [ERR :5934:SimpleText.pm:283:] Resolver did no
[2010-03-17 22:48:21] opensrf.simple-text [INTL:5934:Cache.pm:125:] Stored opensrf.simpl
[2010-03-17 22:48:21] opensrf.simple-text [DEBG:5934:Application.pm:579:] Coderef for [O
[2010-03-17 22:48:21] opensrf.simple-text [DEBG:5934:Application.pm:586:] A top level Re
[2010-03-17 22:48:21] opensrf.simple-text [DEBG:5934:Application.pm:190:] Method duratio
[2010-03-17 22:48:21] opensrf.simple-text [INTL:5934:AppSession.pm:780:] Calling queue_w
[2010-03-17 22:48:21] opensrf.simple-text [INTL:5934:AppSession.pm:769:] Resending...0
[2010-03-17 22:48:21] opensrf.simple-text [INTL:5934:AppSession.pm:450:] In send
[2010-03-17 22:48:21] opensrf.simple-text [DEBG:5934:AppSession.pm:506:] AppSession send
[2010-03-17 22:48:21] opensrf.simple-text [DEBG:5934:AppSession.pm:506:] AppSession send
...
```

To see everything that is happening in OpenSRF, try leaving your logging level set to INTERNAL for a few minutes - just ensure that you have a lot of free disk space available if you have a moderately busy system!

2.7. Caching results: one secret of scalability

If you have ever used an application that depends on a remote Web service outside of your control — say, if you need to retrieve results from a microblogging service — you know the pain of latency and dependability (or the lack thereof). To improve the response time for OpenSRF services, you can take advantage of the support offered by the `OpenSRF::Utils::Cache` module for communicating with a local instance or cluster of `memcache` daemons to store and retrieve persistent values. The following example demonstrates caching by sleeping for 10 seconds the first time it receives a given cache key and cannot retrieve a corresponding value from the cache:

```
use OpenSRF::Utils::Cache;                                       # 1
sub test_cache {
    my $self = shift;
    my $conn = shift;
    my $test_key = shift;
    my $cache = OpenSRF::Utils::Cache->new('global');            # 2
    my $cache_key = "opensrf.simple-text.test_cache.$test_key";  # 3
    my $result = $cache->get_cache($cache_key) || undef;         # 4
    if ($result) {
        $logger->info("Resolver found a cache hit");
        return $result;
    }
    sleep 10;                                                    # 5
    my $cache_timeout = 300;                                     # 6
    $cache->put_cache($cache_key, "here", $cache_timeout);       # 7
    return "There was no cache hit.";
}
```

1. The OpenSRF::Utils::Cache module provides access to the built-in caching support in OpenSRF.
2. The constructor for the cache object accepts a single argument to define the cache type for the object. Each cache type can use a separate `memcache` server to keep the caches separated. Most Evergreen services use the `global` cache, while the `anon` cache is used for Web sessions.

3 The cache key is simply a string that uniquely identifies the value you want to store or retrieve. This line creates a cache key based on the OpenSRF method name and request input value.
4 The `get_cache()` method checks to see if the cache key already exists. If a matching key is found, the service immediately returns the stored value.
5 If the cache key does not exist, the code sleeps for 10 seconds to simulate a call to a slow remote Web service or an intensive process.
6 The `$cache_timeout` variable represents a value for the lifetime of the cache key in seconds.
7 After the code retrieves its value (or, in the case of this example, finishes sleeping), it creates the cache entry by calling the `put_cache()` method. The method accepts three arguments: the cache key, the value to be stored ("here"), and the timeout value in seconds to ensure that we do not return stale data on subsequent calls.

2.8. Initializing the service and its children: child labour

When an OpenSRF service is started, it looks for a procedure called `initialize()` to set up any global variables shared by all of the children of the service. The `initialize()` procedure is typically used to retrieve configuration settings from the `opensrf.xml` file.

An OpenSRF service spawns one or more children to actually do the work requested by callers of the service. For every child process an OpenSRF service spawns, the child process clones the parent environment and then each child process runs the `child_init()` process (if any) defined in the OpenSRF service to initialize any child-specific settings.

When the OpenSRF service kills a child process, it invokes the `child_exit()` procedure (if any) to clean up any resources associated with the child process. Similarly, when the OpenSRF service is stopped, it calls the `DESTROY()` procedure to clean up any remaining resources.

2.9. Retrieving configuration settings

The settings for OpenSRF services are maintained in the `opensrf.xml` XML configuration file. The structure of the XML document consists of a root element `<opensrf>` containing two child elements:

- The `<default>` element contains an `<apps>` element describing all OpenSRF services running on this system — see Section 2.1, “Registering a service with the OpenSRF configuration files” --, as well as any other arbitrary XML descriptions required for global configuration purposes. For example, Evergreen uses this section for email notification and inter-library patron privacy settings.

- The `<hosts>` element contains one element per host that participates in this OpenSRF system. Each host element must include an `<activeapps>` element that lists all of the services to start on this host when the system starts up. Each host element can optionally override any of the default settings.

OpenSRF includes a service named `opensrf.settings` to provide distributed cached access to the configuration settings with a simple API:

- `opensrf.settings.default_config.get` accepts zero arguments and returns the complete set of default settings as a JSON document.

- `opensrf.settings.host_config.get` accepts one argument (hostname) and returns the complete set of settings, as customized for that hostname, as a JSON document.

- `opensrf.settings.xpath.get` accepts one argument (an XPath [http://www.w3.org/TR/xpath/] expression) and returns the portion of the configuration file that matches the expression as a JSON document.

For example, to determine whether an Evergreen system uses the opt-in support for sharing patron information between libraries, you could either invoke the `opensrf.settings.default_config.get` method and parse the JSON document to determine the value, or invoke the `opensrf.settings.xpath.get` method with the XPath `/opensrf/default/share/user/opt_in` argument to retrieve the value directly.

In practice, OpenSRF includes convenience libraries in all of its client language bindings to simplify access to configuration values. C offers osrfConfig.c, Perl offers `OpenSRF::Utils::SettingsClient`, Java offers `org.opensrf.util.SettingsClient`, and Python offers `osrf.set`. These libraries locally cache the configuration file to avoid network roundtrips for every request and enable the developer to request specific values without having to manually construct XPath expressions.

3. Getting under the covers with OpenSRF

Now that you have seen that it truly is easy to create an OpenSRF service, we can take a look at what is going on under the covers to make all of this work for you.

3.1. Get on the messaging bus - safely

One of the core innovations of OpenSRF was to use the Extensible Messaging and Presence Protocol (XMPP, more colloquially known as Jabber) as the messaging bus that ties OpenSRF services together across servers. XMPP is an "XML protocol for near-real-time messaging, presence, and request-response services" (http://www.ietf.org/rfc/rfc3920.txt) that OpenSRF relies on to handle most of the complexity of networked communications. OpenSRF requres an XMPP server that supports multiple domains such as ejabberd [http://www.ejabberd.im/]. Multiple domain support means that a single server can support XMPP virtual hosts with separate sets of users and access privileges per domain. By routing communications through separate public and private XMPP domains, OpenSRF services gain an additional layer of security.

The OpenSRF installation documentation [http://evergreen-ils.org/dokuwiki/doku.php?id=opensrf:1.2:install] instructs you to create two separate hostnames (`private.localhost` and `public.localhost`) to use as XMPP domains. OpenSRF can control access to its services based on the domain of the client and whether a given service allows access from clients on the public domain. When you start OpenSRF, the first XMPP clients that connect to the XMPP server are the OpenSRF public and private *routers*. OpenSRF routers maintain a list of available services and connect clients to available services. When an OpenSRF service starts, it establishes a connection to the XMPP server and registers itself with the private router. The OpenSRF configuration contains a list of public OpenSRF services, each of which must also register with the public router.

3.2. OpenSRF communication flows over XMPP

In a minimal OpenSRF deployment, two XMPP users named "router" connect to the XMPP server, with one connected to the private XMPP domain and one connected to the public XMPP domain. Similarly, two XMPP users named "opensrf" connect to the XMPP server via the private and public XMPP domains. When an OpenSRF service is started, it uses the "opensrf" XMPP user to advertise its availability with the corresponding router on that XMPP domain; the XMPP server automatically assigns a Jabber ID (*JID*) based on the client hostname to each service's listener process and each connected drone process waiting to carry out requests. When an OpenSRF router receives a request to invoke a method on a given service, it connects the requester to the next available listener in the list of registered listeners for that service.

Services and clients connect to the XMPP server using a single set of XMPP client credentials (for example, `opensrf@private.localhost`), but use XMPP resource identifiers to differentiate themselves in the JID for each connection. For example, the JID for a copy of the `opensrf.simple-text` service with process ID `6285` that has connected to the `private.localhost` domain using the `opensrf` XMPP client credentials could be `opensrf@private.localhost/opensrf.simple-text_drone_at_localhost_6285`. By convention, the user name for OpenSRF clients is `opensrf`, and the user name for OpenSRF routers is `router`, so the XMPP server for OpenSRF will have four separate users registered: * `opensrf@private.localhost` is an OpenSRF client that connects with these credentials and which can access any OpenSRF service. * `opensrf@public.localhost` is an OpenSRF client that connects with these credentials and which can only access OpenSRF services that have registered with the public router. * `router@private.localhost` is the private OpenSRF router with which all services register. * `router@public.localhost` is the public OpenSRF router with which only services that must be publicly accessible register.

All OpenSRF services automatically register themselves with the private XMPP domain, but only those services that register themselves with the public XMPP domain can be invoked from public OpenSRF clients. The OpenSRF client and router user names, passwords, and domain names, along with the list of services that should be public, are contained in the `opensrf_core.xml` configuration file.

3.3. OpenSRF communication flows over HTTP

In some contexts, access to a full XMPP client is not a practical option. For example, while XMPP clients have been implemented in JavaScript, you might be concerned about browser compatibility and processing overhead - or you might want to issue OpenSRF requests from the command line with `curl`. Fortunately, any OpenSRF service registered with the public router is accessible via the OpenSRF HTTP Translator. The OpenSRF HTTP Translator implements the OpenSRF-over-HTTP proposed specification [http://www.open-ils.org/dokuwiki/doku.php?id=opensrf_over_http] as an Apache module that translates HTTP requests into OpenSRF requests and returns OpenSRF results as HTTP results to the initiating HTTP client.

```
# curl request broken up over multiple lines for legibility
curl -H "X-OpenSRF-service: opensrf.simple-text"                        \ # 1
    --data 'osrf-msg=[                                                  \ # 2
        {"__c":"osrfMessage","__p":{"threadTrace":0,"locale":"en-CA",   \ # 3
            "type":"REQUEST","payload": {"__c":"osrfMethod","__p":      \
                {"method":"opensrf.simple-text.reverse","params":["foobar"]}    \
            }}                                                          \
        }]'                                                             \
http://localhost/osrf-http-translator                                   \ # 4
```

1 The `X-OpenSRF-service` header identifies the OpenSRF service of interest.

2 The POST request consists of a single parameter, the `osrf-msg` value, which contains a JSON array.

3 The first object is an OpenSRF message (`"__c":"osrfMessage"`) with a set of parameters (`"__p":{}`).

- The identifier for the request (`"threadTrace":0`); this value is echoed back in the result.
- The message type (`"type":"REQUEST"`).
- The locale for the message; if the OpenSRF method is locale-sensitive, it can check the locale for each OpenSRF request and return different information depending on the locale.
- The payload of the message (`"payload":{}`) containing the OpenSRF method request (`"__c":"osrfMethod"`) and its parameters (`"__p:"{}`).
  - The method name for the request (`"method":"opensrf.simple-text.reverse"`).
  - A set of JSON parameters to pass to the method (`"params":["foobar"]`); in this case, a single string `"foobar"`.

4 The URL on which the OpenSRF HTTP translator is listening, `/osrf-http-translator` is the default location in the Apache example configuration files shipped with the OpenSRF source, but this is configurable.

```
# HTTP response broken up over multiple lines for legibility
[{"__c":"osrfMessage","__p":                                                \ # 1
    {"threadTrace":0, "payload":                                            \ # 2
        {"__c":"osrfResult","__p":                                          \ # 3
            {"status":"OK","content":"raboof","statusCode":200}             \ # 4
        },"type":"RESULT","locale":"en-CA"                                  \ # 5
    }
},
{"__c":"osrfMessage","__p":                                                 \ # 6
    {"threadTrace":0,"payload":                                             \ # 7
        {"__c":"osrfConnectStatus","__p":                                   \ # 8
            {"status":"Request Complete","statusCode":205}                  \ # 9
        },"type":"STATUS","locale":"en-CA"                                  \ # 10
    }
}]
```

1 The OpenSRF HTTP Translator returns an array of JSON objects in its response. Each object in the response is an OpenSRF message (`"__c":"osrfMessage"`) with a collection of response parameters (`"__p":`).

2 The OpenSRF message identifier (`"threadTrace":0`) confirms that this message is in response to the request matching the same identifier.

3 The message includes a payload JSON object (`"payload":`) with an OpenSRF result for the request (`"__c":"osrfResult"`).

4 The result includes a status indicator string (`"status":"OK"`), the content of the result response - in this case, a single string "raboof" (`"content":"raboof"`) - and an integer status code for the request (`"statusCode":200`).

5 The message also includes the message type (`"type":"RESULT"`) and the message locale (`"locale":"en-CA"`).

6 The second message in the set of results from the response.

7 Again, the message identifier confirms that this message is in response to a particular request.

8 The payload of the message denotes that this message is an OpenSRF connection status message (`"__c":"osrfConnectStatus"`), with some information about the particular OpenSRF connection that was used for this request.

9 The response parameters for an OpenSRF connection status message include a verbose status (`"status":"Request Complete"`) and an integer status code for the connection status (`"statusCode":205).

10 The message also includes the message type (`"type":"RESULT"`) and the message locale (`"locale":"en-CA"`).

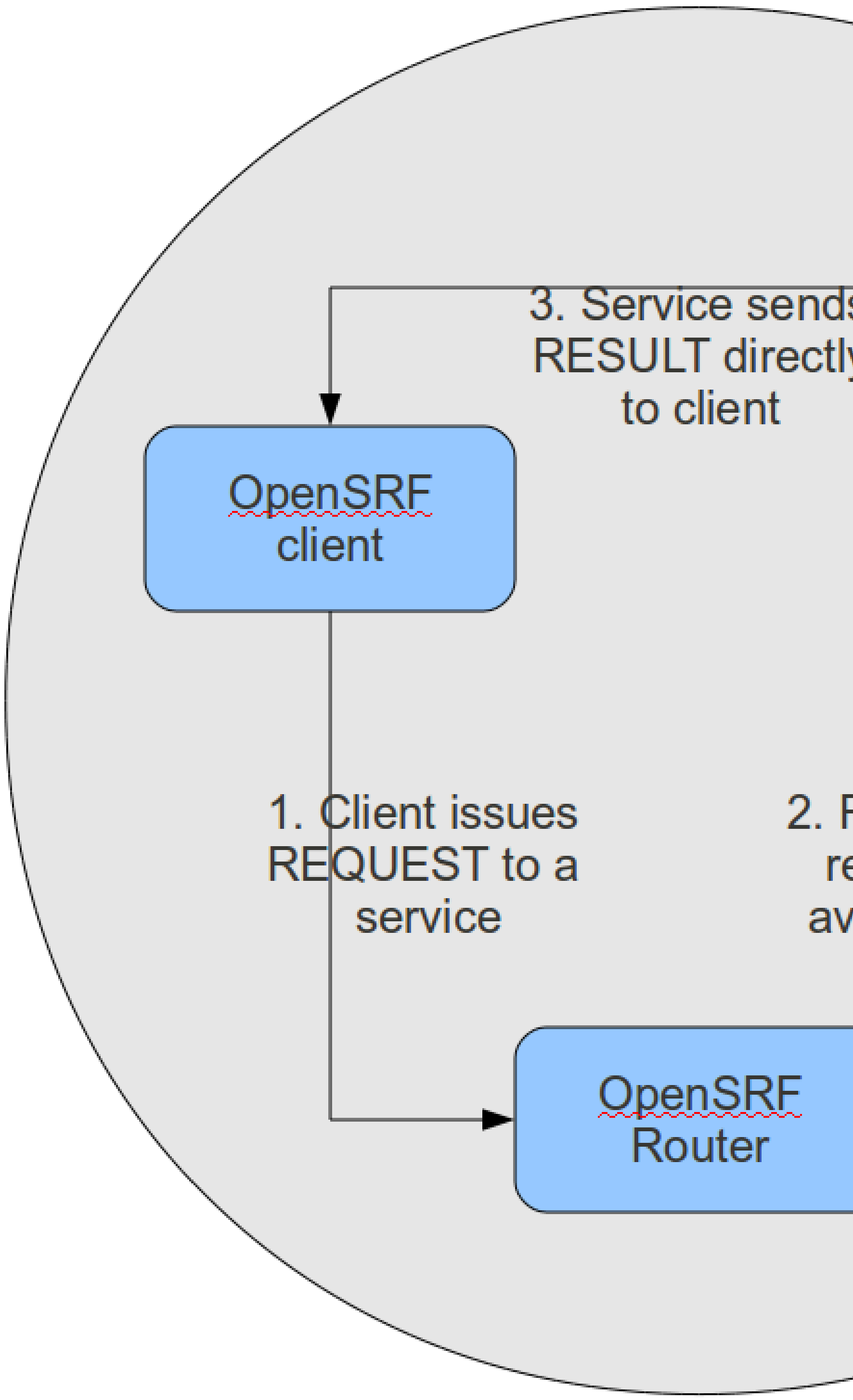
Tip

Before adding a new public OpenSRF service, ensure that it does not introduce privilege escalation or unchecked access to data. For example, the Evergreen `open-ils.cstore` private service is an object-relational mapper that provides read and write access to the entire Evergreen database, so it would be catastrophic to expose that service publicly. In comparison, the Evergreen `open-ils.pcrud` public service offers the same functionality as `open-ils.cstore` to any connected HTTP client or OpenSRF client, but the additional authentication and authorization layer in `open-ils.pcrud` prevents unchecked access to Evergreen's data.

3.4. Stateless and stateful connections

OpenSRF supports both *stateless* and *stateful* connections. When an OpenSRF client issues a `REQUEST` message in a *stateless* connection, the router forwards the request to the next available service and the service returns the result directly to the client.

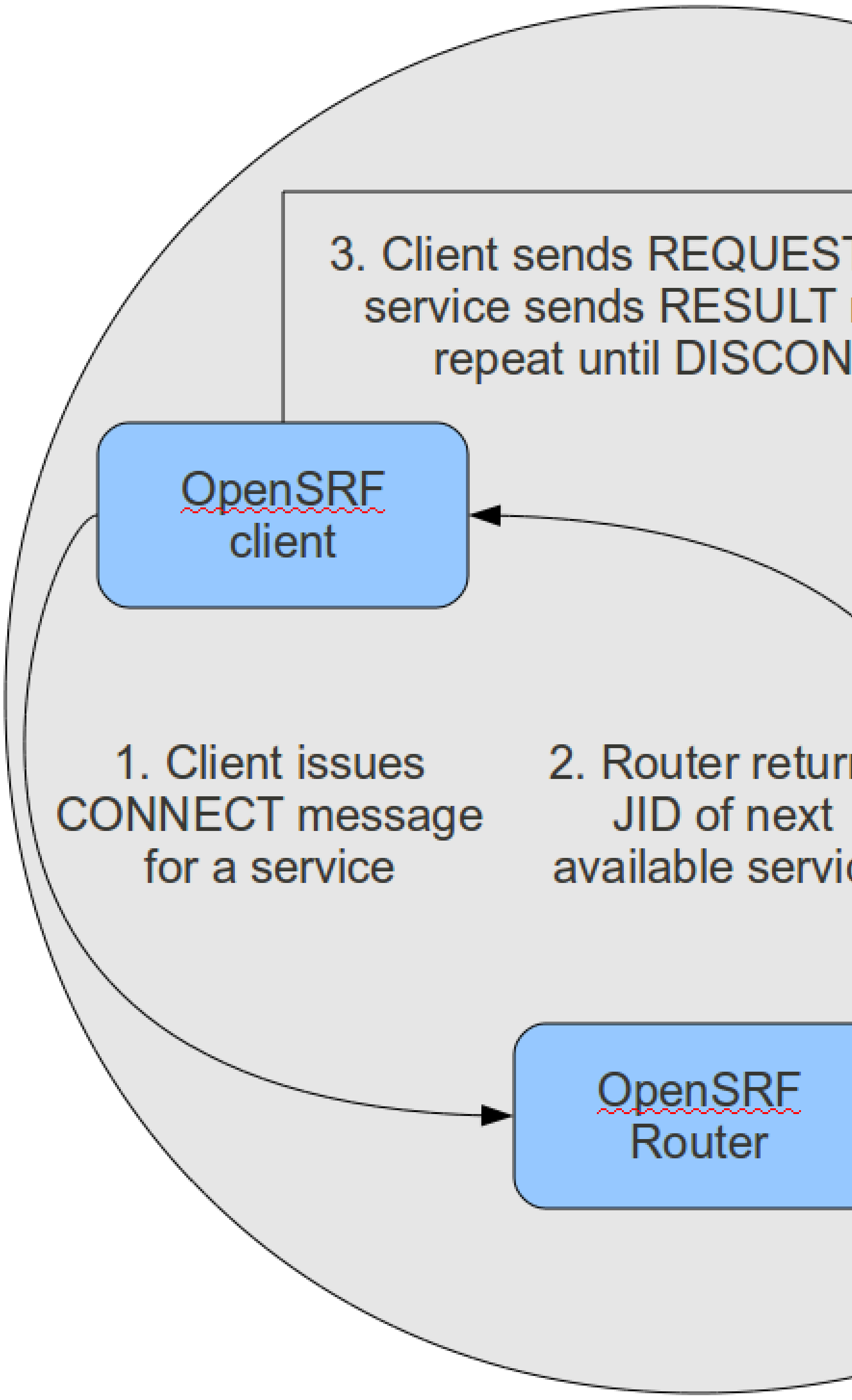
REQUEST flow in a stateless



connection.

When an OpenSRF client issues a `CONNECT` message to create a *stateful* conection, the router returns the Jabber ID of the next available service to the client so that the client can issue one or more `REQUEST` message directly to that particular service and the service will return corresponding `RESULT` messages directly to the client. Until the client issues a `DISCONNECT` message, that particular service is only available to the requesting client. Stateful connections are useful for clients that need to make many requests from a particular service, as it avoids the intermediary step of contacting the router for each request, as well as for operations that require a controlled sequence of commands, such as a set of database INSERT, UPDATE, and DELETE statements within a transaction.

CONNECT, REQUEST, and DISCONNECT flow in a stateful



connection.

3.5. Message body format

OpenSRF was an early adopter of JavaScript Object Notation (JSON). While XMPP is an XML protocol, the Evergreen developers recognized that the compactness of the JSON format offered a significant reduction in bandwidth for the volume of messages that would be generated in an application of that size. In addition, the ability of languages such as JavaScript, Perl, and Python to generate native objects with minimal parsing offered an attractive advantage over invoking an XML parser for every message. Instead, the body of the XMPP message is a simple JSON structure. For a simple request, like the following example that simply reverses a string, it looks like a significant overhead: but we get the advantages of locale support and tracing the request from the requester through the listener and responder (drone).

```
<message from='router@private.localhost/opensrf.simple-text'
  to='opensrf@private.localhost/opensrf.simple-text_listener_at_localhost_6275'
  router_from='opensrf@private.localhost/_karmic_126678.3719_6288'
  router_to='' router_class='' router_command='' osrf_xid=''
>
  <thread>1266781414.366573.12667814146288</thread>
  <body>
[
  {"__c":"osrfMessage","__p":
    {"threadTrace":"1","locale":"en-US","type":"REQUEST","payload":
      {"__c":"osrfMethod","__p":
        {"method":"opensrf.simple-text.reverse","params":["foobar"]}
      }
    }
  }
]
  </body>
</message>
```

```
<message from='opensrf@private.localhost/opensrf.simple-text_drone_at_localhost_6285'
  to='opensrf@private.localhost/_karmic_126678.3719_6288'
  router_command='' router_class='' osrf_xid=''
>
  <thread>1266781414.366573.12667814146288</thread>
  <body>
[
  {"__c":"osrfMessage","__p":
    {"threadTrace":"1","payload":
      {"__c":"osrfResult","__p":
        {"status":"OK","content":"raboof","statusCode":200}
      } ,"type":"RESULT","locale":"en-US"}
  },
  {"__c":"osrfMessage","__p":
    {"threadTrace":"1","payload":
      {"__c":"osrfConnectStatus","__p":
        {"status":"Request Complete","statusCode":205}
      },"type":"STATUS","locale":"en-US"}
  }
]
  </body>
</message>
```

The content of the `<body>` element of the OpenSRF request and result should look familiar; they match the structure of the OpenSRF over HTTP examples that we previously dissected.

3.6. Registering OpenSRF methods in depth

Let’s explore the call to `__PACKAGE__->register_method()`; most of the members of the hash are optional, and for the sake of brevity we omitted them in the previous example. As we have seen in the results of the introspection call, a verbose registration method call is recommended to better enable the internal documentation. Here is the complete set of members that you should pass to `__PACKAGE__->register_method()`:

- The `method` member specifies the name of the procedure in this module that is being registered as an OpenSRF method.
- The `api_name` member specifies the invocable name of the OpenSRF method; by convention, the OpenSRF service name is used as the prefix.
- The optional `api_level` member can be used for versioning the methods to allow the use of a deprecated API, but in practical use is always 1.
- The optional `argc` member specifies the minimal number of arguments that the method expects.
- The optional `stream` member, if set to any value, specifies that the method supports returning multiple values from a single call to subsequent requests. OpenSRF automatically creates a corresponding method with ".atomic" appended to its name that returns the complete set of results in a single request. Streaming methods are useful if you are returning hundreds of records and want to act on the results as they return.
- The optional `signature` member is a hash that describes the method’s purpose, arguments, and return value.
  - The `desc` member of the `signature` hash describes the method’s purpose.
  - The `params` member of the `signature` hash is an array of hashes in which each array element describes the corresponding method argument in order.
    - The `name` member of the argument hash specifies the name of the argument.
    - The `desc` member of the argument hash describes the argument’s purpose.
    - The `type` member of the argument hash specifies the data type of the argument: for example, string, integer, boolean, number, array, or hash.
  - The `return` member of the `signature` hash is a hash that describes the return value of the method.
    - The `desc` member of the `return` hash describes the return value.
    - The `type` member of the `return` hash specifies the data type of the return value: for example, string, integer, boolean, number, array, or hash.

4. Evergreen-specific OpenSRF services

Evergreen is currently the primary showcase for the use of OpenSRF as an application architecture. Evergreen 1.6.1 includes the following set of OpenSRF services:

- The `open-ils.actor` service supports common tasks for working with user accounts and libraries.
- The `open-ils.auth` service supports authentication of Evergreen users.
- The `open-ils.booking` service supports the management of reservations for bookable items.
- The `open-ils.cat` service supports common cataloging tasks, such as creating, modifying, and merging bibliographic and authority records.
- The `open-ils.circ` service supports circulation tasks such as checking out items and calculating due dates.
- The `open-ils.collections` service supports tasks that assist collections agencies in contacting users with outstanding fines above a certain threshold.
- The `open-ils.cstore` private service supports unrestricted access to Evergreen fieldmapper objects.
- The `open-ils.ingest` private service supports tasks for importing data such as bibliographic and authority records.
- The `open-ils.pcrud` service supports permission-based access to Evergreen fieldmapper objects.
- The `open-ils.penalty` penalty service supports the calculation of penalties for users, such as being blocked from further borrowing, for conditions such as having too many items checked out or too many unpaid fines.
- The `open-ils.reporter` service supports the creation and scheduling of reports.
- The `open-ils.reporter-store` private service supports access to Evergreen fieldmapper objects for the reporting service.
- The `open-ils.search` service supports searching across bibliographic records, authority records, serial records, Z39.50 sources, and ZIP codes.
- The `open-ils.storage` private service supports a deprecated method of providing access to Evergreen fieldmapper objects. Implemented in Perl, this service has largely been replaced by the much faster C-based `open-ils.cstore` service.
- The `open-ils.supercat` service supports transforms of MARC records into other formats, such as MODS, as well as providing Atom and RSS feeds and SRU access.
- The `open-ils.trigger` private service supports event-based triggers for actions such as overdue and holds available notification emails.
- The `open-ils.vandelay` service supports the import and export of batches of bibliographic and authority records.

Of some interest is that the `open-ils.reporter-store` and `open-ils.cstore` services have identical implementations. Surfacing them as separate services enables a deployer of Evergreen to ensure that the reporting service does not interfere with the performance-critical `open-ils.cstore` service. One can also direct the reporting service to a read-only database replica to, again, avoid interference with `open-ils.cstore` which must write to the master database.

There are only a few significant services that are not built on OpenSRF in Evergreen 1.6.0, such as the SIP and Z39.50 servers. These services implement different protocols and build on existing daemon architectures (Simple2ZOOM for Z39.50), but still rely on the other OpenSRF services to provide access to the Evergreen data. The non-OpenSRF services are reasonably self-contained and can be deployed on different servers to deliver the same sort of deployment flexibility as OpenSRF services, but have the disadvantage of not being integrated into the same configuration and control infrastructure as the OpenSRF services.

5. Evergreen after one year: reflections on OpenSRF

Project Conifer [http://projectconifer.ca] has been live on Evergreen for just over a year now, and as one of the primary technologists I have had to work closely with the OpenSRF infrastructure during that time. As such, I am in a position to identify some of the strengths and weaknesses of OpenSRF based on our experiences.

5.1. Strengths of OpenSRF

As a service infrastructure, OpenSRF has been remarkably reliable. We initially deployed Evergreen on an unreleased version of both OpenSRF and Evergreen due to our requirements for some functionality that had not been delivered in a stable release at that point in time, and despite this risky move we suffered very little unplanned downtime in the opening months. On July 27, 2009 we moved to a newer (but still unreleased) version of the OpenSRF and Evergreen code, and began formally tracking our downtime. Since then, we have achieved more than 99.9% availability - including scheduled downtime for maintenance. This compares quite favourably to the maximum of 75% availability that we were capable of achieving on our previous library system due to the nightly downtime that was required for our backup process. The OpenSRF "maximum request" configuration parameter for each service that kills off drone processes after they have served a given number of requests provides a nice failsafe for processes that might otherwise suffer from a memory leak or hung process. It also helps that when we need to apply an update to a Perl service that is running on multiple servers, we can apply the updated code, then restart the service on one server at a time to avoid any downtime.

As promised by the OpenSRF infrastructure, we have also been able to tune our cluster of servers to provide better performance. For example, we were able to change the number of maximum concurrent processes for our database services when we saw a performance bottleneck with database access. To go live with a configuration change, we restart the `opensrf.setting` service to pick up the change, then restart the affected service on each of our servers. We were also able to turn off some of the less-used OpenSRF services, such as `open-ils.collections`, on one of our servers to devote more resources on that server to the more frequently used services and other performance-critical processes such as Apache.

The support for logging and caching that is built into OpenSRF has been particularly helpful with the development of a custom service for SFX holdings integration into our catalogue. Once I understood how OpenSRF works, most of the effort required to build that SFX integration service was spent on figuring out how to properly invoke the SFX API to display human-readable holdings. Adding a new OpenSRF service and registering several new methods for the service was relatively easy. The support for directing log messages to syslog in OpenSRF has also been a boon for both development and debugging when problems arise in a cluster of five servers; we direct all of our log messages to a single server where we can inspect the complete set of messages for the entire cluster in context, rather than trying to piece them together across servers.

5.2. Weaknesses

The primary weakness of OpenSRF is the lack of either formal or informal documentation for OpenSRF. There are many frequently asked questions on the Evergreen mailing lists and IRC channel that indicate that some of the people running Evergreen or trying to run Evergreen have not been able to find documentation to help them understand, even at a high level, how the OpenSRF Router and services work with XMPP and the Apache Web server to provide a working Evergreen system. Also, over the past few years several developers have indicated an interest in developing Ruby and PHP bindings for OpenSRF, but the efforts so far have resulted in no working code. The lack of a formal specification, clearly annotated examples, and portable unit tests for the major OpenSRF communication use cases is a significant hurdle for a developer seeking to fulfill a base set of expectations for a working binding. As a result, Evergreen integration efforts with popular frameworks like Drupal, Blacklight, and VuFind result in the best practical option for a developer with limited time — database-level integration — which has the unfortunate side effect of being much more likely to break after an upgrade.

In conjunction with the lack of documentation that makes it hard to get started with the framework, a disincentive for new developers to contribute to OpenSRF itself is the lack of integrated unit tests. For a developer to contribute a significant, non-obvious patch to OpenSRF, they need to manually run through various (undocumented, again) use cases to try and ensure that the patch introduced no unanticipated side effects. The same problems hold for Evergreen itself, although the Constrictor [http://svn.open-ils.org/ILS-Contrib/constrictor] stress-testing framework offers a way of performing some automated system testing and performance testing.

These weaknesses could be relatively easily overcome with the effort through contributions from people with the right skill sets. This article arguably offers a small set of clear examples at both the networking and application layer of OpenSRF. A technical writer who understands OpenSRF could contribute a formal specification to the project. With a formal specification at their disposal, a quality assurance expert could create an automated test harness and a basic set of unit tests that could be incrementally extended to provide more coverage over time. If one or more continual integration environments are set up to track the various OpenSRF branches of interest, then the OpenSRF community would have immediate feedback on build quality. Once a unit testing framework is in place, more developers might be willing to develop and contribute patches as they could sanity check their own code without an intense effort before exposing it to their peers.

6. Summary

In this article, I attempted to provide both a high-level and detailed overview of how OpenSRF works, how to build and deploy new OpenSRF services, how to make requests to OpenSRF method from OpenSRF clients or over HTTP, and why you should consider it a possible infrastructure for building your next high-performance system that requires the capability to scale out. In addition, I surveyed the Evergreen services built on OpenSRF and reflected on the strengths and weaknesses of the platform based on the experiences of Project Conifer after a year in production, with some thoughts about areas where the right application of skills could make a significant difference to the Evergreen and OpenSRF projects.

7. Appendices

7.1. `opensrf_core.xml` complete example

This complete example of `opensrf_core.xml` from a working test system is included for your convenience.

```
<?xml version="1.0"?>
<config>

  <!-- bootstrap config for OpenSRF apps -->
  <opensrf>

    <routers>

      <!-- define the list of routers our services will register with -->

      <router>

        <!-- This is the public router.  On this router, we only register applications
             which should be accessible to everyone on the opensrf network -->
        <name>router</name>
        <domain>public.localhost</domain>
        <services>
            <service>opensrf.math</service>
            <service>opensrf.simple-text</service>
        </services>
      </router>

      <router>
        <!-- This is the private router.  All applications must register with
            this router, so no explicit <services> section is required -->
        <name>router</name>
        <domain>private.localhost</domain>
      </router>
    </routers>


    <!-- Jabber login settings
        Our domain should match that of the private router -->
    <domain>private.localhost</domain>
    <username>opensrf</username>
    <passwd>privosrf</passwd>
    <port>5222</port>
    <!-- name of the router used on our private domain.
        this should match one of the <name> of the private router above -->
    <router_name>router</router_name>

    <!-- log file settings ======================================  -->
    <!-- log to a local file -->
    <logfile>/openils/var/log/osrfsys.log</logfile>

    <!-- Log to syslog. You can use this same layout for
        defining the logging of all services in this file -->
    <!--
    <logfile>syslog</logfile>
    <syslog>local2</syslog>
    <actlog>local1</actlog>
    -->

    <!-- 0 None, 1 Error, 2 Warning, 3 Info, 4 debug, 5 Internal (Nasty) -->
    <loglevel>3</loglevel>

    <!-- config file for the services -->
    <settings_config>/openils/conf/opensrf.xml</settings_config>

  </opensrf>

  <!-- The section between <gateway>...</gateway> is a standard OpenSRF C stack config f
  <gateway>

    <!--
    we consider ourselves to be the "originating" client for requests,
    which means we define the log XID string for log traces
    -->
    <client>true</client>

    <!--  the routers's name on the network -->
    <router_name>router</router_name>

    <!-- jabber login info -->
    <!-- The gateway connects to the public domain -->
    <domain>public.localhost</domain>
    <username>opensrf</username>
    <passwd>pubosrf</passwd>
    <port>5222</port>
    <logfile>/openils/var/log/gateway.log</logfile>
    <loglevel>3</loglevel>

  </gateway>

  <!-- =================================================================================

    <routers>
        <router> <!-- public router -->
            <trusted_domains>
                <!-- allow private services to register with this router
                     and public clients to send requests to this router. -->
                <server>private.localhost</server>
                <!-- also allow private clients to send to the router so it can receive
                <client>private.localhost</client>
                <client>public.localhost</client>
            </trusted_domains>
            <transport>
                <server>public.localhost</server>
                <port>5222</port>
                <unixpath>/openils/var/sock/unix_sock</unixpath>
                <username>router</username>
                <password>pubrout</password>
                <resource>router</resource>
                <connect_timeout>10</connect_timeout>
                <max_reconnect_attempts>5</max_reconnect_attempts>
            </transport>
            <logfile>/openils/var/log/router.log</logfile>
            <!--
            <logfile>syslog</logfile>
```

```
            <syslog>local2</syslog>
            -->
            <loglevel>3</loglevel>
        </router>
        <router> <!-- private router -->
            <trusted_domains>
                <server>private.localhost</server>
                <!-- only clients on the private domain can send requests to this router
                <client>private.localhost</client>
            </trusted_domains>
            <transport>
                <server>private.localhost</server>
                <port>5222</port>
                <username>router</username>
                <password>privrout</password>
                <resource>router</resource>
                <connect_timeout>10</connect_timeout>
                <max_reconnect_attempts>5</max_reconnect_attempts>
            </transport>
            <logfile>/openils/var/log/router.log</logfile>
            <!--
            <logfile>syslog</logfile>
            <syslog>local2</syslog>
            -->
            <loglevel>3</loglevel>
        </router>
    </routers>

  <!-- ================================================================================

</config>
```

7.2. `opensrf.xml` complete example

This complete example of `opensrf.xml` from a working test system is included for your convenience.

```
<?xml version="1.0"?>
<!--
vim:et:ts=2:sw=2:
-->
<opensrf version="0.0.3">
<!--

        There is one <host> entry for each server on the network.  Settings for the
        'default' host are used for every setting that isn't overridden within a given
        host's config.

        To specify which applications a host is serving, list those applications
        within that host's config section.  If the defaults are acceptible, then
        that's all that needs to be added/changed.

        Any valid XML may be added to the <default> block and server components will hav
        acces to it.

-->

  <default>
    <dirs>
```

```
      <!-- opensrf log files go in this directory -->
      <log>/openils/var/log</log>

      <!-- opensrf unix domaind socket files go here -->
      <sock>/openils/var/lock</sock>

      <!-- opensrf pids go here -->
      <pid>/openils/var/run</pid>

      <!-- global config directory -->
      <conf>/openils/conf</conf>
    </dirs>

    <!-- prefork, simple. prefork is suggested -->
    <server_type>prefork</server_type>

    <!-- Default doesn't host any apps -->
    <activeapps/>
    <cache>
      <global>
        <servers>

          <!-- memcached server ip:port -->
          <server>127.0.0.1:11211</server>

        </servers>

        <!-- maximum time that anything may stay in the cache -->
        <max_cache_time>86400</max_cache_time>

      </global>
    </cache>

    <!--
    These are the defaults for every served app.  Each server should
    duplicate the node layout for any nodes that need changing.
    Any settings that are overridden in the server specific section
    will be used as the config values for that server.  Any settings that are
    not overridden will fall back on the defaults
    Note that overriding 'stateless' will break things
    -->

    <apps>
      <opensrf.persist>

        <!--
        How many seconds to wait between server
        requests before timing out a stateful server session.
        -->
        <keepalive>1</keepalive>

        <!--
        if 1, then we support stateless sessions (no connect required),
        if 0 then we don't
        -->
        <stateless>1</stateless>

        <!--
        Tells the servers which language this implementation is coded in
        In this case non "perl" servers will not be able to load the module
        -->
        <language>perl</language>

        <!-- Module the implements this application -->
        <implementation>OpenSRF::Application::Persist</implementation>

        <!-- max stateful requests before a session automatically disconnects a client -
        <max_requests>97</max_requests>

        <!-- settings for the backend application drones.  These are probably sane defau
        <unix_config>

          <!-- unix socket file -->
          <unix_sock>opensrf.persist_unix.sock</unix_sock>

          <!-- pid file -->
          <unix_pid>opensrf.persist_unix.pid</unix_pid>

          <!-- max requests per process backend before a child is recycled -->
          <max_requests>1000</max_requests>

          <!-- log file for this application -->
          <unix_log>opensrf.persist_unix.log</unix_log>

          <!-- Number of children to pre-fork -->
          <min_children>5</min_children>

          <!-- maximun number of children to fork -->
          <max_children>25</max_children>

          <!-- minimun number of spare forked children -->
          <min_spare_children>2</min_spare_children>

          <!-- max number of spare forked children -->
          <max_spare_children>5</max_spare_children>

        </unix_config>

        <!-- Any additional setting for a particular application go in the app_settings n
        <app_settings>

          <!-- sqlite database file -->
          <dbfile>/openils/var/persist.db</dbfile>

        </app_settings>
      </opensrf.persist>

      <opensrf.math>
        <keepalive>3</keepalive>
        <stateless>1</stateless>
        <language>c</language>
        <implementation>osrf_math.so</implementation>
        <max_requests>7</max_requests>
        <unix_config>
          <unix_sock>opensrf.math_unix.sock</unix_sock>
          <unix_pid>opensrf.math_unix.pid</unix_pid>
          <max_requests>1000</max_requests>
          <unix_log>opensrf.math_unix.log</unix_log>
        <min_children>5</min_children>
        <max_children>15</max_children>
        <min_spare_children>2</min_spare_children>
        <max_spare_children>5</max_spare_children>
      </unix_config>
    </opensrf.math>

    <opensrf.dbmath>
      <keepalive>3</keepalive>
      <stateless>1</stateless>
      <language>c</language>
      <implementation>osrf_dbmath.so</implementation>
      <max_requests>99</max_requests>
      <unix_config>
        <max_requests>1000</max_requests>
        <unix_log>opensrf.dbmath_unix.log</unix_log>
        <unix_sock>opensrf.dbmath_unix.sock</unix_sock>
        <unix_pid>opensrf.dbmath_unix.pid</unix_pid>
        <min_children>5</min_children>
        <max_children>15</max_children>
        <min_spare_children>2</min_spare_children>
        <max_spare_children>5</max_spare_children>
      </unix_config>
    </opensrf.dbmath>

    <opensrf.simple-text>
      <keepalive>3</keepalive>
      <stateless>1</stateless>
      <language>perl</language>
      <implementation>OpenSRF::Application::Demo::SimpleText</implementation>
      <unix_config>
        <max_requests>100</max_requests>
        <unix_log>opensrf.simple-text_unix.log</unix_log>
        <unix_sock>opensrf.simple-text_unix.sock</unix_sock>
        <unix_pid>opensrf.simple-text_unix.pid</unix_pid>
        <min_children>5</min_children>
        <max_children>15</max_children>
        <min_spare_children>2</min_spare_children>
        <max_spare_children>5</max_spare_children>
      </unix_config>
    </opensrf.simple-text>

    <opensrf.settings>
      <keepalive>1</keepalive>
      <stateless>1</stateless>
      <language>perl</language>
      <implementation>OpenSRF::Application::Settings</implementation>
      <max_requests>17</max_requests>
      <unix_config>
        <unix_sock>opensrf.settings_unix.sock</unix_sock>
        <unix_pid>opensrf.settings_unix.pid</unix_pid>
        <max_requests>1000</max_requests>
        <unix_log>opensrf.settings_unix.log</unix_log>
        <min_children>5</min_children>
        <max_children>15</max_children>
        <min_spare_children>3</min_spare_children>
        <max_spare_children>5</max_spare_children>
      </unix_config>
    </opensrf.settings>
    </apps>
  </default>

  <hosts>

    <localhost>
      <!-- ^-=-
        Should match the fully qualified domain name of the host.

        On Linux, the output of the following command is authoritative:
        $ perl -MNet::Domain -e 'print Net::Domain::hostfqdn() . "\n";'

        To use 'localhost' instead, run osrf_ctl.sh with the -l flag
      -->
      <!-- List all of the apps this server will be running -->
      <activeapps>
        <appname>opensrf.persist</appname>
        <appname>opensrf.settings</appname>
        <appname>opensrf.math</appname>
        <appname>opensrf.dbmath</appname>
        <appname>opensrf.simple-text</appname>
      </activeapps>

      <apps>

<!-- Example of an app-specific setting override -->
        <opensrf.persist>
          <app_settings>
            <dbfile>/openils/var/persist-override.db</dbfile>
          </app_settings>
        </opensrf.persist>

      </apps>

    </localhost>

  </hosts>

</opensrf>
```

7.3. Python client

Following is a Python client that makes the same OpenSRF calls as the Perl client:

```
#!/usr/bin/env python
"""OpenSRF client example in Python"""
import osrf.system
import osrf.ses

def osrf_substring(session, text, sub):
    """substring: Accepts a string and a number as input, returns a string"""
    request = session.request('opensrf.simple-text.substring', text, sub)

    # Retrieve the response from the method
    # The timeout parameter is optional
    response = request.recv(timeout=2)

    request.cleanup()
    # The results are accessible via content()
    return response.content()

def osrf_split(session, text, delim):
    """split: Accepts two strings as input, returns an array of strings"""
    request = session.request('opensrf.simple-text.split', text, delim)
    response = request.recv()
    request.cleanup()
    return response.content()

def osrf_statistics(session, strings):
    """statistics: Accepts an array of strings as input, returns a hash"""
    request = session.request('opensrf.simple-text.statistics', strings)
    response = request.recv()
    request.cleanup()
    return response.content()


if __name__ == "__main__":
    file = '/openils/conf/opensrf_core.xml'

    # Pull connection settings from <config><opensrf> section of opensrf_core.xml
    osrf.system.System.connect(config_file=file, config_context='config.opensrf')

    # Set up a connection to the opensrf.settings service
    session = osrf.ses.ClientSession('opensrf.simple-text')

    result = osrf_substring(session, "foobar", 3)
    print(result)
    print

    result = osrf_split(session, "This is a test", " ")
    print("Received %d elements: [" % len(result)),
    print(', '.join(result)), ']'

    many_strings = (
        "First I think I'll have breakfast",
        "Then I think that lunch would be nice",
        "And then seventy desserts to finish off the day"
    )
    result = osrf_statistics(session, many_strings)
    print("Length: %d" % result["length"])
    print("Word count: %d" % result["word_count"])

    # Cleanup connection resources
    session.cleanup()
```

Note

Python’s `dnspython` module refuses to read `/etc/resolv.conf`, so to access hostnames that are not served up via DNS, such as the extremely common case of `localhost`, you may need to install a package like `dnsmasq` to act as a local DNS server for those hostnames.

8. License