Call Redirectors

sipXregistry project supports writing plugins in C++ to redirect calls. Current implementation include directed call pickup a feature where you can pickup any ringing phone from another phone. Implementation also includes time of day routing.

> ⚠ **Incomplete**
>
> This page needs your help to finish. We need to expand on the API to write a plugin.

The sipXregistry redirect server maps addresses in a series of lookups, adding Contacts for each to a result set. In addition, some redirectors (processing steps) can edit Contacts found by earlier redirectors. When all the lookups are complete, it returns a redirect response to the request based on the contents of the result set (or a not found response if the result set is empty).

The Redirectors

The redirectors are (in their current order):

Registrations

```
$prefix/var/sipxdata/sipdb/registration.xml
```

If the target URI matches a registered address, then the Contact addresses sent in the registration are added. These are stored in the in-memory database, but are made persistent in the XML file.

Aliases

```
$prefix/var/sipxdata/sipdb/alias.xml
```

The file maps addresses based on the contacts configured in the XML file. Each AOR can translate to any number of contacts. This is used for direct mappings such as a user name for use in a SIP URL to an extension number (or the reverse). There is an example alias.xml file in this directory.

SUBSCRIBE Processing

If the request method is SUBSCRIBE, this step removes the q values from all the contacts, so the SUBSCRIBE always forks in parallel rather than in series. This ensures that the subscriber gets a prompt response from all targets for the requested extension. This is placed here in the order so that it does not apply to contacts added by later phases, but affects only contacts added by user agents directly or provisioned aliase; this has the effect that low q-value contacts like fallback to voicemail are not pursued. Since most SUBSCRIBEs are concerned only with local events, this reduces the number of messages forked to systems that don't do anything useful with them anyway.

Mapping Rules

```
$prefix/etc/sipxpbx/mappingrules.xml
```

Mapping rules provide the means to select addresses based on host and user name patterns and configured permissions, and then add contacts. Mapping rules may use the results of those matches and other configured data to add contacts to the result set.

Fallback Mapping Rules

```
$prefix/etc/sipxpbx/fallbackrules.xml
```

This phase is used *only* if there are no contacts in the result set when it is reached; if the result set is non-empty, this phase is skipped. This is normally used to provide mapping to non-local addresses (such as to a gateway or the proxy authoritative for another domain). The mapping rules file in this phase uses the same syntax and expressive ability as the first Mapping Rules phase.

Global Call Pick-Up Processing

```
$prefix/var/sipxdata/sipdb/credentials.xml
```

This redirector handles the mappings needed by the Global Call Pick-Up feature. It does two mappings: It maps the global call pick-up feature code to a proper invocation of the pick-up agent, and it maps the special "all extensions" user name ("*allcredentials") to all URIs in the credentials.xml file.

Other Configuration files

The following configuration files are also important, but they are also shared with other components, so they are documented in sipXcommserverLib/doc:

- $prefix/var/sipxdata/sipdb/credentials.xml
  Defines all user identities and the credentials used to authenticate them.

- $prefix/var/sipxdata/sipdb/permission.xml
  Defines permissions granted to each identity.

Suspending redirection processing

When a redirector processes a request, it returns a status code. The values are SUCCESS, ERROR (which means that redirection should terminate immediately with a 500 error response), and SUSPEND. If the redirector returns SUSPEND, it is indicating that its processing cannot be completed quickly.

Before returning SUSPEND, the redirector is responsible for arranging for asynchronous processing that will bring the redirector into a state where it can complete its processing quickly. Asynchronous processing then calls a function to indicate to the redirect server that the request should be reprocessed.

The redirect server will the re-execute the redirectors for the request. The redirectors are expected to succeed, after which the redirect server sends a response to the request. (A redirector is allowed to return SUSPEND several times for a single request, but this should be avoided for efficiency.)

A redirector has a method by which the redirect server may inform it that a request that it is working on has been canceled. Such cancellation may be due to a CANCEL from the requester, a transaction time-out (because some redirector remained suspended too long), reinitialization of the redirect server, or an error response from another redirector.

If a redirector asks for suspension, processing continues with the other redirectors in the sequence, but at the end of processing, the response that is being generated is discarded, and the entire sequence is re-executed once all the redirectors have indicated that they can succeed, generating a new, correct response. This is somewhat inefficient, but there is no simple scheme to make it more efficient that does not place tight constraints on the redirectors that we may want to loosen in the near future. The current target is that a suspended redirection should take no more than twice the work of a non-suspended redirection, and that a very small fraction of redirection requests will be suspended.

Each request that is suspended has a sequence number that is unique over long periods of time. Currently, the sequence number is a 32 bit unsigned integer that increments for every request. It is given to each redirector and is used to identify the request.

Redirectors may maintain their own data storage in one of two ways.

The first method is simple to code but is single-threaded. The redirect server maintains for every suspended request and every redirector a pointer to private storage for that redirector. The redirector may allocate an object and save the pointer to it. The redirect server guarantees to delete the object (thus executing its destructor) after processing of the request is done. Using service methods, the redirector can find the object for any particular sequence number, or examine the objests for all suspended requests. However, any access to these objects requires holding a global mutex.

The second method is more complex to code but can be parallelized. That method is for the redirector to maintain its own data structures, using the request serial numbers to coordinate its data items with the redirect server's list of suspended requests. The redirect server guarantees to call the redirector's "cancel request" method once its services are no longer needed for that request. The redirector's "cancel request" method can free storage allocated for the request.