Call Resolver Design Specification

Revised: March 15, 2006

Introduction

This memo is the first step towards a design specification for the Call Resolver feature. It builds on documents from earlier projects. This draft also includes notes on project planning.

Business Requirements

- The Call Resolver analyzes the calls handled by a sipX PBX and distills a set of Call Detail Records (CDRs) describing the calls.
- The Call Resolver can be scheduled to run automatically on a daily basis.
- CDRs are intended for use by the customer to generate and validate billing.
- CDRs are sensitive data that must be handled securely.
- Each CDR corresponds to a single SIP call.
- Both successfully completed calls and calls that end with errors are covered.
- CDRs are made available in a relational database that can be accessed via standard means such as ODBC. The schema is stable and documented.
- 35 days worth of CDR data are retained. After 35 days, records are purged.

Non-requirements:

- Report generation is not provided.
- We do not (yet) link together multiple calls that comprise a larger transaction, such as:
  - Calls that are received by an ACD and subequently handed off to an agent
  - Transfers, whether blind or consultative
  - Conference calls
- CDR processing is not "*highly available.*" There are single points of failure and data may be lost in the event of system or network outages.

Functional Requirements

- The Call Resolver creates a CDR for each call handled by the ECS (a.k.a. PBX), that includes:
  - **Caller info** from SIP headers: AOR and contact URL.
  - **Callee info** from SIP headers: AOR and contact URL.
  - **Start time** of the call in GMT. This is when the SIP INVITE is received by the proxy.
  - **Connect time** of the call in GMT. This is when the proxy receives the caller's ACK in response to the callee's 200 OK.
  - **End time** of the call in GMT. This is when the call was terminated. Typically that happens when the SIP BYE is received by the proxy, but calls may also end in a error state.
  - **Termination code**: an enumeration value indicating how a call was terminated.
  - **Failure code**: if a call fails for some reason, then the SIP error response code, such as 4xx, 5xx, 6xx, is recorded in this data element.
  - **SIP dialog ID**: the combination of call ID, from tag, and to tag that uniquely identifies the call. Not directly relevant for billing, but is useful for debugging or for post-processing CDRs to link related calls together. For example, one might want to link

together the call legs involved in a consultative transfer.

- **Call direction**: inbound, outbound, or intranetwork. Inbound/outbound is with respect to a gateway:
  - An inbound call is a call initiated from a PSTN gateway.
  - An outbound call is a call initiated from a phone to a PSTN gateway.
  - An intranetwork call is a call that does not go through a PSTN gateway.

  Call direction is a customer-specific requirement that is implemented as a Call Resolver add-on.

- CDRs are created from **Call State Events (CSEs)** logged by the forking proxy and auth proxy. By default, 35 days of CSE data are also retained, but that is separately configurable if desired. Having the CSE data underpinning the CDRs is helpful if questions come up. It is always a good idea to keep at least an hour or so of CSE data around so that the Call Resolver can analyze calls that span reporting periods.
- **Configuration** of the Call Resolver and of CSE logging can be done through the config server so that the user is not forced to edit config files manually.
- **PostgreSQL** is the database product used for CDR storage, the same database product used by the config server. PostgreSQL version 7.4 or higher is required. Version 8.0 or higher is recommended.
- **Distributed proxies**. In the initial release, only non-HA configurations with a single forking proxy and auth proxy on a single machine are supported. However, the software design anticipates distributed proxies so that it will be straightforward to provide this in a future release.

Software Design

Call State Event (CSE) Logging

In support of the Call Resolver, CSEs are logged to a relational database rather than to XML. (XML event logging is still supported, but is deprecated.) The database provides robust, unified storage with a query capability. “Unified” means that CSEs from different proxies are stored together rather than being distributed across separate log files for each proxy. In the future this will support unified logging from distributed proxies; initially all proxies must run on the same machine.

Logging to a DB is done via ODBC. This new capability is implemented in the C++ code for the forking proxy and auth proxy.

See the separate CSE logging spec for details.

Call Resolver

The Call Resolver is a new subproject of sipXproxy. It is implemented in Ruby, using the ActiveRecord component of the Rails framework for object-relational mapping to the database. We chose Ruby because it is a high-productivity scripting language that is ideal for such tasks, and because ActiveRecord is a superior technology for database access.

Call Resolution

CDRs are stored in three tables:

- **dialogs**: holds the SIP dialog identifier consisting of the call_id, from_tag, and to_tag.
- **parties**: holds AORs and contact URLs for the caller and callee.

- **cdrs**: holds the CDR records with additional info such as the start_time, connect_time, and end_time.

To make life simpler for users, we provide a single view, `view_cdrs`, that includes the most commonly used CDR data in a single flat list so that users don't have to worry about joining multiple tables. The user can simply do `select * from view_cdrs` and get a list of records that look like this:

Row	*id*	*caller_aor*	*callee_aor*	*start_time*	*connect_time*	*end_time*
1	1	sip:bob@biloxi.com	sip:jay@biloxi.com	1:00:00 PM	1:00:01 PM	1:05:00 PM

To fit on one line, the above presentation is simplified, showing just time values in date/time columns and omitting the `termination`, `failure_status` and `failure_reason` columns, which are part of the view. As noted in the code comments above, times are displayed in GMT. See `sipXproxy/etc/database/schema.sql` for details.

For each table, there is a matching class defined in the code: `Dialog`, `Party`, `Cdr`.

Define the following single-char codes for the CDR `termination` column:

- R: call requested – got a call_request event, but no other events.
- I: call in progress – got both call_request and call_setup events.
- C: successful call completion – call_request, call_setup, and call_end with no errors.
- F: call failed – an error occurred.

In the case of a failure, the `failure_status` column is set to the SIP error code, for example, 486 when the line is busy. The `failure_reason` column is set to error text. To save space, if the error text is the default for that SIP error code, then the column is left NULL. For example, the default error text for a 486 code is “Busy Here”. (We could easily set up the CDRs view to show the default error text rather than NULL, if desired.)

The Call Resolver is invoked with:

- start and end date/time values that specify the time window to analyze
- an override flag that indicates whether to overwrite CDRs that were completely analyzed in a previous run (default is false)

All CSEs in that window are processed as follows:

- Query the CSE DB for all events in the time window.
- Sort all events by `call_id`. Only CSEs are processed, observer events are ignored.
- For each `call_id`:
    - Examine the set of events with that call_id. If present, the from_url and to_url values must not be different, but some events in the call may not have one or both of these, and should be considered to be a match so long as the call_id value matches. If the from_url or to_url values differ, then “bail out”. “Bail out” means log a message at the appropriate level depending on the error condition, discard the events, and continue with the next call_id.

      Order the events within a call by the event_time value. Query the DB to pull in any additional events with the same call_id that occur outside the window. This allows us to handle calls that span time windows. Note the payoff from using a DB rather than

log files, which do not offer random access.

- Create an in-memory Cdr instance named `cdr`.
- Find the first call_request event (event_type = 'R'), the event with the earliest timestamp. If one is found, then:
  - Create a `Dialog` instance, `dialog`. Set `dialog.call_id` and `dialog.from_tag` from the event.
  - Create a `Party` instance, `caller`, for the caller:
    - Set `caller.aor` to the aor part of the `from_url` value from the call_request, stripping any tags from the `from_url`. (Convert aor to a canonical format using Url class?)
    - Set `caller.contact` to the `contact` value of the call_request.
  - Create a `Party` instance, `callee`, for the callee:
    - Set `callee.aor` to the aor part of the `to_url` value from the call_request, stripping any tags from the `to_url`.
  - Create a `Cdr` instance, `cdr`:
    - Set `cdr.start_time` to the timestamp from the call_request.
    - Set `cdr.termination` provisionally to “R” meaning “call requested”.
- If no call_request event is found, then bail out.
- Find the first call_setup event (event_type = 'S'). If one is found, then:
  - Set `dialog.to_tag` from the event.
  - Set `callee.contact` from the `contact` value of the event.
  - Set `cdr.connect_time` to the timestamp from the event.
  - Set `cdr.termination` provisionally to “I” meaning “call in progress”.
  - If `cdr.connect_time` is <= `cdr.start_time`, then bail out.
- If there are `call_failure` events, then find the last one and:
  - Set `cdr.end_time` to the timestamp from the `call_failure` event.
  - Set `cdr.termination` to “F” for failure.
  - Set `cdr.failure_status` and `cdr.failure_reason` from the `failure_status` and `failure_reason` values of the event. As noted earlier, if `failure_reason` in the event is the default error text, then leave `cdr.failure_reason` empty.
- Find the last `call_end` event whose call_id, from_tag, and to_tag exactly match those in the selected call_setup event. If one is found, then:
  - If no `call_setup` event was found then bail out.
  - Set `cdr.end_time` to the timestamp from the `call_end` event.
  - Set `cdr.termination` to “C” meaning “successful call completion”.

- If no matching call_end event is found, then keep going.
- If `cdr.end_time <= cdr.connect_time` then bail out.
- Look in the CDR DB for a CDR with matching dialog info (`call_id`, `from_tag`, `to_tag`).
  - If there is no preexisting CDR, then save the in-memory `dialog`, `caller`, `callee`, and `cdr` objects, with foreign key values tying them together. For the `dialog`, `caller`, and `callee`, look for preexisting objects in the CDR DB with matching values and don't create duplicates.
  - If there is a preexisting CDR, then overwrite it if it is incomplete (`termination` is R or I) or the `overwrite` flag is true. Otherwise discard the in-memory data without saving it.
- Continue with the next call_id.

Invoking the Call Resolver

The Call Resolver is invoked as follows:

```
sipxcallresolver.sh --start start_time --end end_time
```

where *start_time* and *end_time* define the time window to be analyzed. Example:

```
sipxcallresolver.sh --start “2007-01-31T03:00:00” --end “2007-02-01T03:00:00”
```

covers the time window from 3 AM on January 31, 2007 until 3 AM on February 1, 2007. A variety of time input formats are supported: RFC 2822 (email), RFC 2616 (HTTP), and ISO 8601 (the subset used by XML Schema). Times are interpreted relative to the local timezone if a timezone is not specified.

Typically the Call Resolver is scheduled to run automatically by setting configuration parameters, as described in the next section, rather than by being invoked manually.

Scheduling the Call Resolver

The Call Resolver has a config file, callresolver.in. Set

SIP_CALLRESOLVER_RUN_DAILY : ENABLE (default is DISABLE)

SIP_CALLRESOLVER_TIME_TO_RUN : 1:11 AM (default is 3:00 AM)

for example, to run the resolver every day at 1:11 AM. If you wish to run the resolver more frequently, then disable the daily run and set up your own cron job outside of sipX, to run the resolver at any desired interval.

In order to supply the resolver with CSE data, CSE logging to the database must be enabled for both the forking proxy and the auth proxy. See the CSE logging spec.

In general, users can ignore config files and set these parameters through sipXconfig.

Logging

Like other sipX executables, the Call Resolver supports logging, in this case to `var/log/sipxpbx/sipcallresolver.log`. Following the standard pattern, these configuration parameters control logging:

- SIP_CALLRESOLVER_LOG_CONSOLE
- SIP_CALLRESOLVER_LOG_DIR
- SIP_CALLRESOLVER_LOG_LEVEL

Call Direction

Algorithm

With a SIP call, the caller and the callee each have an “address of record” (AOR) such as "Bob <sip:bob@biloxi.com>". They each also have a “contact”, where the AOR has been mapped to a network address, for example "<sip:bob@192.0.2.4>". We'll use the term “contact host” for the host part of the contact address, e.g., “192.0.2.4”. Here's how to figure out call direction:

- When the contact host for the original INVITE resolves to a PSTN gateway, the call is incoming.
- When the contact host for the 200 OK response resolves to a PSTN gateway, the call is outgoing.
- An intranetwork call is a call for which neither the caller contact host nor the callee contact host matches a PSTN gateway address.

In looking for a match, we need to keep in mind that a contact host may be a DNS name like “gateway.biloxi.com” or an IP address like “192.0.2.5”. Both cases must be checked. We will accomplish that by converting all DNS names to IP addresses, to provide a standard format for address comparison. Note that reverse lookup from IP address to DNS name is possible in principle, but in practice the necessary DNS records are often not set up properly.

Schema

The file `sipXproxy/etc/database/schema.sql` defines the base schema. To support call direction, we add:

- a `call_directions` table that adds call direction to the `cdrs` table
- a view, `view_cdrs_cleveland`, that adds call direction to the `view_cdrs` view

The word “cleveland” is a code name for the customer. These add-ons to the schema are defined in `sipXproxy/etc/database/cleveland.sql`. See that file for details.

Requirements

The Call Resolver must know all the gateway addresses. It gets them by querying the configuration server database. We require that:

- All active gateways are registered with the configuration server, rather than being configured manually.
- All gateways are PSTN gateways for the purposes of figuring out call direction.
- The Call Resolver is installed on the same physical server as the configuration server.

These requirements apply to the first Call Resolver release and could be loosened in future releases.

Note that this simple design can run into problems in a scenario where there are branch offices, each with their own PBX and gateways, with a central Call Manager that routes between them.

The Call Manager acts as a gateway, so interoffice calls through the Call Manager will look like incoming or outgoing calls to the calling and called offices, respectively, even though they don't go to the PSTN. If we make the software a little smarter so that it knows that the Call Manager is not a PSTN gateway, then there are still problems if the Call Manager is used to implement least cost routing. In that case, an outgoing call originating in one office could be routed by the Call Manager to a gateway in another branch office, to save money. But the caller's PBX won't know about that other gateway, so it won't know that the call should be considered outgoing. For the initial release, we'll assume that interoffice calls are much less common than intraoffice calls, and therefore that this limitation is acceptable.

Unit testing

Achieve good (80-90%) code coverage with unit tests.

Notes

- **Database independence**. Avoid lock-in to PostgreSQL or any other database product. We can still take advantage of database-specific features as long as they don't compromise portability, e.g. PostgreSQL's TEXT data type for unbounded variable-length text values.
- **Distributed proxies**. The CSE DB to which a proxy writes CSEs is always colocated (same machine) with the proxy, to reduce load on the runtime system and to ensure that CSEs can be recorded without any dependency on network connections. Thus if proxies are distributed, there are multiple CSE DBs. But there is never more than one CDR DB.
- **Security**. Access to either the CSE or CDR DB is authenticated. Data is encrypted on the wire using a protocol such as SSL or TLS. In the initial release, all data is local, so there is no “on the wire” to worry about.