CGRates Documentation

Release 0.9.1 rc7

Radu Ioan Fericean/Dan Christian Bogos

December 02, 2015

Contents

Contents:

CHAPTER 1

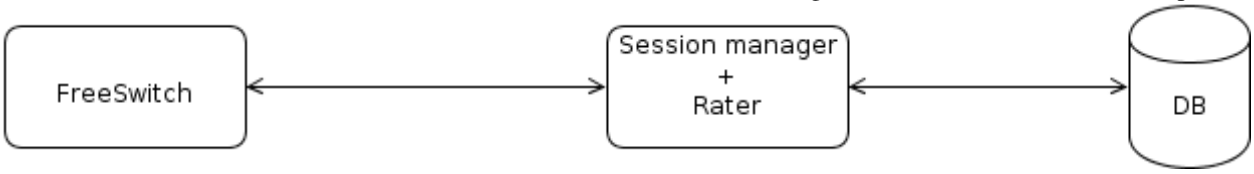
1.Introduction

CGRateS is a very fast and easy scalable rating engine targeted especially for ISPs and Telecom Operators.

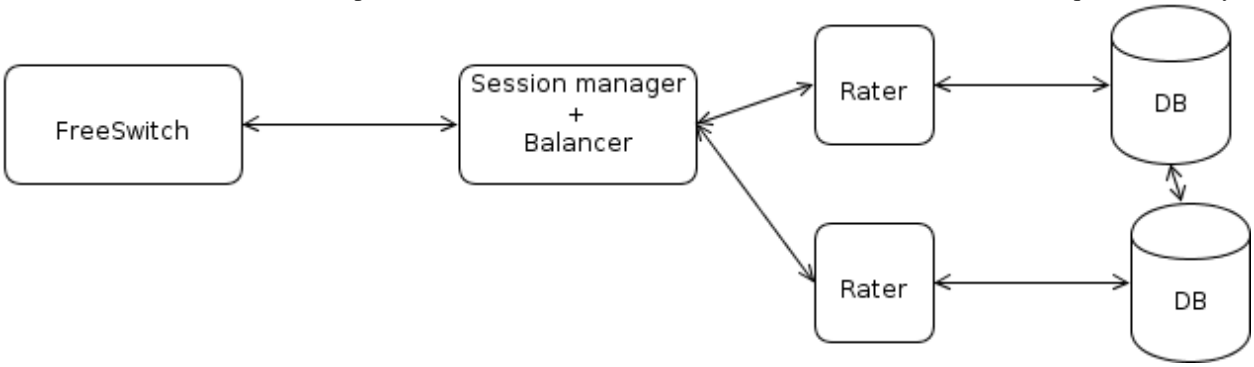
It is written in Go and accessible from any language via JSON RPC. The code is well documented (go doc compliant API docs) and heavily tested.

After testing various databases like Kyoto cabinet, Redis or Mongodb, the project focused on Redis as it delivers the best trade-off between speed, configuration and scalability. Despite that a connection to any database can be easily integrated by writing a simple adapter.

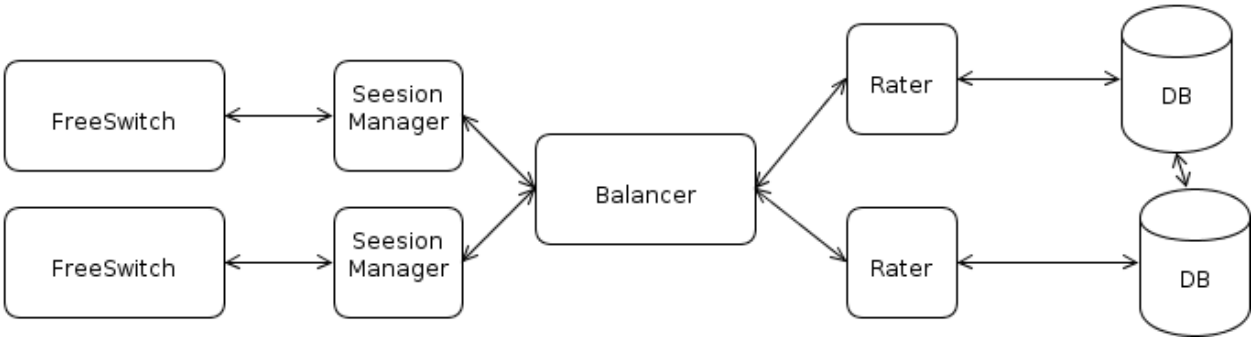
To better understand the CGRateS architecture, bellow are some of the configurations in which CGRateS can operate:



This scenario fits most of the simple installations. The Balancer can be left out and the Rater can be queried directly.



While the network grows more Raters can be thrown into the stack to offer more requests per seconds workload. This implies the usage of the Balancer to distribute the requests to the Raters running on the different machines.



Of course more SessionManagers can serve multiple Telecom Switches and all of them are connected to the same Balancer. We are planning to support multiple Balancers for huge networks if the need arises.

1.1 1.1. CGRateS Features

- **Reliable and Fast (very fast ;)). To get an idea about speed, we have benchmarked 13000+ req/sec on a rather modest ma**
  - Using most modern programming concepts like multiprocessor support, asynchronous code execution within microthreads.
  - Built-in data caching system per call duration.
  - In-Memory database with persistence over restarts.
  - Use of Balancer assures High-Availability of Raters as well as increase of processing performance where that is required.
  - Use of Linux enterprise ready tools to assure High-Availability of the Balancer where that is required (*Supervise* for Application level availability and *LinuxHA* for Host level availability).
- **Modular architecture**
  - Easy to enhance functionality by rewriting custom session managers or mediators.
  - Flexible API accessible via both Gob (Golang specific, increased performance) or JSON (platform independent, universally accesible).
- **Prepaid, Postpaid and Pseudo-Prepaid Controller.**
  - Mutiple Primary Balances per Account (eg: MONETARY, SMS, INTERNET_MINUTES, INTERNET_TRAFFIC).
  - Multiple Auxiliary Balances per Account (eg: Free Minutes per Destination, Volume Rates, Volume Discounts).
  - Concurrent sessions per account sharing the same balance with configurable debit interval (starting with 1 second).
  - Built-in Task-Scheduler supporting both one-time as well as recurrent actions (eg: TOPUP_MINUTES_PER_DESTINATION, DEBIT_MONETARY, RESET_BALANCE).
  - ActionTriggers (useful for commercial offerings like receive amounts of monetary units if a specified number of minutes was charged in a month).
- **Highly configurable Rating.**
  - Connect Fees.
  - Priced Units definition.

  - Rate increments.
  - Millisecond timestaps.
  - Four decimal currencies.
  - Multiple TypeOfRecord rating (eg: standard vs. premium calls, SMSes, Internet Traffic).
  - Rating subject concatenations for combined records (eg: location based rating for same user).
  - Recurrent rates definition (per year, month, day, dayOfWeek, time).
  - Rating Profiles activation times (eg: rates becoming active at specific time in future).
- Multi-Tenant for both Prepaid as well as Rating.
- Flexible Mediator able to run multiple mediation processes on the same CDR.
- Verbose action logging in persistent databases (eg: Postgres) to cope with country specific law requirements.
- Good documentation (that's me :).
- "Free as in Beer" with commercial support available on-demand.

CHAPTER 2

2. Architecture

The CGRateS suite consits of three software applications, described bellow.

2.1 2.1. cgr-engine

The most important and complex as functionality.

Customisable through the use of a configuration file, it will start on demand one or more services, outlined bellow.

```
rif@grace:~$ cgr-engine -help
Usage of cgr-engine:
  -cdrs
        Enforce starting of the cdrs daemon overwriting config
  -config_dir string
        Configuration directory path. (default "/etc/cgrates/")
  -cpuprofile string
        write cpu profile to file
  -pid string
        Write pid file
  -rater
        Enforce starting of the rater daemon overwriting config
  -scheduler
        Enforce starting of the scheduler daemon .overwriting config
  -singlecpu
        Run on single CPU core
  -version
        Prints the application version.
```

Example cgr-engine -config_dir=/etc/cgrates

2.1.1 2.1.1. Rater service

Responsible with the following tasks:

- Operates on balances.
- Computes prices for rating subjects.
- Monitors and executes triggers.

Accessed by components using it's functionality via RPC or directly within same running cgr-engine process.

2.1.2 2.1.2. Balancer service

Optional component, used as proxy/balancer to a pool of Rater workers.

The Raters will register their availability to the Balancer thus implementing dynamic HA functionality.

2.1.3 2.1.3. Scheduler service

Used to execute periodic/scheduled tasks.

2.1.4 2.1.4. SessionManager service

Responsible with call control on the Telecommunication Switch side. Operates in two different modes (per call or globally):

- Prepaid
- Monitors call start.
- Checks balance availability for the call.
- Enforces global timer for a call at call-start.
- Executes routing commands for the call where that is necessary (eg call un-park in case of FreeSWITCH).
- Periodically executes balance debits on call at the beginning of debit interval.
- Enforce call disconnection on insufficient balance.
- Refunds the balance taken in advance at the call stop.
- Postpaid
- Executes balance debit on call-stop.

All call actions are logged into CGRateS's LogDB.

2.1.5 2.1.5 Mediator service

Responsible to mediate the CDRs generated by Telecommunication Switch.

Has the ability to combine CDR fields into rating subject and run multiple mediation processes on the same record.

On Linux machines, able to work with inotify kernel subsystem in order to process the records close to real-time after the Switch has released them.

2.2 2.2. cgr-loader

Used for importing the rating information into the CGRateS database system.

```
rif@grace:~$ cgr-loader -help
Usage of cgr-loader:
  -cdrstats_address string
        CDRStats service to contact for data reloads, empty to disable automatic data reloads (defaul
  -datadb_host string
        The DataDb host to connect to. (default "127.0.0.1")
  -datadb_name string
```

```
      The name/number of the DataDb to connect to. (default "11")
-datadb_passwd string
      The DataDb user's password.
-datadb_port string
      The DataDb port to bind to. (default "6379")
-datadb_type string
      The type of the DataDb database <redis> (default "redis")
-datadb_user string
      The DataDb user to sign in as.
-dbdata_encoding string
      The encoding used to store object data in strings (default "msgpack")
-dry_run
      When true will not save loaded data to dataDb but just parse it for consistency and errors.
-flushdb
      Flush the database before importing
-from_stordb
      Load the tariff plan from storDb to dataDb
-history_server string
      The history server address:port, empty to disable automaticautomatic  history archiving (defa
-load_history_size int
      Limit the number of records in the load history (default 10)
-migrate_rc8
      Migrate Accounts, Actions and ActionTriggers to RC8 structures
-path string
      The path to folder containing the data files (default "./")
-rater_address string
      Rater service to contact for cache reloads, empty to disable automatic cache reloads (default
-runid string
      Uniquely identify an import/load, postpended to some automatic fields
-stats
      Generates statsistics about given data.
-stordb_host string
      The storDb host to connect to. (default "127.0.0.1")
-stordb_name string
      The name/number of the storDb to connect to. (default "cgrates")
-stordb_passwd string
      The storDb user's password. (default "CGRateS.org")
-stordb_port string
      The storDb port to bind to. (default "3306")
-stordb_type string
      The type of the storDb database <mysql> (default "mysql")
-stordb_user string
      The storDb user to sign in as. (default "cgrates")
-timezone string
      Timezone for timestamps where not specified <""|UTC|Local|$IANA_TZ_DB> (default "Local")
-to_stordb
      Import the tariff plan from files to storDb
-tpdb_host string
      The TariffPlan host to connect to. (default "127.0.0.1")
-tpdb_name string
      The name/number of the TariffPlan to connect to. (default "10")
-tpdb_passwd string
      The TariffPlan user's password.
-tpdb_port string
      The TariffPlan port to bind to. (default "6379")
-tpdb_type string
      The type of the TariffPlan database <redis> (default "redis")
-tpdb_user string
```

```
        The TariffPlan user to sign in as.
  -tpid string
        The tariff plan id from the database
  -users_address string
        Users service to contact for data reloads, empty to disable automatic data reloads (default "
  -validate
        When true will run various check on the loaded data to check for structural errors
  -verbose
        Enable detailed verbose logging output
  -version
        Prints the application version.
```

Example cgr-loader -flushdb

2.3 2.3. cgr-console

Command line tool used to interface with the Rater (or Balancer). Able to execute sub-commands

```
rif@grace:~$ cgr-console -help
Usage of cgr-console:
  -rpc_encoding string
        RPC encoding used <gob|json> (default "json")
  -server string
        server address host:port (default "127.0.0.1:2012")
  -verbose
        Show extra info about command execution.
  -version
        Prints the application version.

rif@grace:~$ cgr-console help_more
2013/04/13 17:23:51
Usage: cgr-console [cfg_opts...{-h}] <status|get_balance>
```

Example cgr-console status

CHAPTER 3

3.Installation

CGRateS can be installed via packages as well as Go automated source install. We recommend using source installs for advanced users familiar with Go programming and packages for users not willing to be involved in the code building process.

3.1 3.1. Using packages

3.1.1 3.1.2. Debian Jessie/Wheezy

This is for the moment the only packaged and the most recommended to use method to install CGRateS.

On the server you want to install CGRateS, simply execute the following commands:

```
cd /etc/apt/sources.list.d/
wget -O - http://apt.itsyscom.com/conf/cgrates.gpg.key|apt-key add -
wget http://apt.itsyscom.com/conf/cgrates.apt.list
apt-get update
apt-get install cgrates
```

Once the installation is completed, one should perform the post-install section in order to have the CGRateS properly set and ready to run. After post-install actions are performed, CGRateS will be configured in */etc/cgrates/cgrates.json* and enabled in */etc/default/cgrates*.

3.2 3.2. Using source

After the go environment is installed (at least go1.2) and configured issue the following commands:

```
go get github.com/cgrates/cgrates/...
```

This command will install the trunk version of CGRateS together with all the necessary dependencies.

For developing CGRateS and switching betwen lts versions we are using the new (experimental) vendor directory feature introduced in go 1.5. In a nutshell all the dependencies are installed and used from a folder named vendor placed in the root of the project.

To manage this vendor folder we use a tool named glide which will download specific versions of the external packages used by CGRateS. To configure the project with glide use the following commands:

```
export GO15VENDOREXPERIMENT=1 #this should be placed in the rc script of your shell
go get github.com/cgrates/cgrates
cd $GOPATH/src/github.com/cgrates/cgrates
glide up
```

The glide up command will install the external dependencies versions specified in the glide.yaml file in the vendor folder. There are different versions for each CGRateS branch, versions that are recorded in the yaml file when the GCRateS releases are made (using glide pin command).

Note that the vendor folder should not be registered with the VCS we are using. For more information and command options for use glide readme page.

3.3 3.3. Post-install

3.3.1 Database setup

For it's operation CGRateS uses more database types, depending on it's nature, install and configuration being further necessary.

At present we support the following databases:

- Redis

Used as DataDb, optimized for real-time information access. Once installed there should be no special requirements in terms of setup since no schema is necessary.

- MySQL

Used as StorDb, optimized for CDR archiving and offline Tariff Plan versioning. Once database is installed, CGRateS database needs to be set-up out of provided scripts (example for the paths set-up by debian package)

```
    cd /usr/share/cgrates/storage/mysql/
    ./setup_cgr_db.sh root CGRateS.org localhost
```

- PostgreSQL

Used as StorDb, optimized for CDR archiving and offline Tariff Plan versioning. Once database is installed, CGRateS database needs to be set-up out of provided scripts (example for the paths set-up by debian package)

```
    cd /usr/share/cgrates/storage/postgres/
    ./setup_cgr_db.sh
```

3.3.2 Git

The CGR-History component will use Git to archive tariff plan changes, hence it's installation is necessary before using CGR-History.

CHAPTER 4

4.Configuration

The behaviour of **CGRateS** can be externally influenced by following means:

- Engine configuration files, usually located at */etc/cgrates/*. There can be one or multiple file/folder hierarchies behind configuration folder with support for automatic includes. The folders/files will be imported in alphabetical order into final configuration object.
- Tariff Plans: set of files used to import various data used in CGRateS subsystems (eg: Rating, Accounting, LCR, DerivedCharging, etc).
- RPC APIs: set of JSON/GOB encoded APIs remotely available for various operational/administrative tasks.

4.1 cgr-engine configuration file

Organized into configuration sections. All configuration options come with defaults and we have tried our best to choose the best ones for a minimum of efforts necessary when running.

Bellow is the default configuration file which comes hardcoded into cgr-engine, most of them being explained and exemplified there.

```
{

// Real-time Charging System for Telecom & ISP environments
// Copyright (C) ITsysCOM GmbH
//
// This file contains the default configuration hardcoded into CGRateS.
// This is what you get when you load CGRateS with an empty configuration file.

//"general": {
//    "http_skip_tls_verify": false,                  // if enabled Http Client will accept any TLS c
//    "rounding_decimals": 10,                               // system level precision for floats
//    "dbdata_encoding": "msgpack",                   // encoding used to store object data in string
//    "tpexport_dir": "/var/log/cgrates/tpe", // path towards export folder for offline Tariff Plans
//    "default_reqtype": "*rated",                    // default request type to consider when missin
//    "default_category": "call",                            // default Type of Record to consider w
//    "default_tenant": "cgrates.org",                // default Tenant to consider when missing fro
//    "default_subject": "cgrates",                   // default rating Subject to consider when miss
//    "connect_attempts": 3,                  // initial server connect attempts
//    "reconnects": -1,                       // number of retries in case of connection lost
//},


//"listen": {
```

```
//    "rpc_json": "127.0.0.1:2012",                    // RPC JSON listening address
//    "rpc_gob": "127.0.0.1:2013",                     // RPC GOB listening address
//    "http": "127.0.0.1:2080",                        // HTTP listening address
//},


//"tariffplan_db": {                                          // database used to store activ
//    "db_type": "redis",                                     // tariffplan_db type: <redis>
//    "db_host": "127.0.0.1",                          // tariffplan_db host address
//    "db_port": 6379,                                        // port to reach the tariffplan
//    "db_name": "10",                                        // tariffplan_db name to connec
//    "db_user": "",                                          // sername to use when connecti
//    "db_passwd": "",                                        // password to use when connect
//},


//"data_db": {                                                      // database used to sto
//    "db_type": "redis",                                     // data_db type: <redis>
//    "db_host": "127.0.0.1",                          // data_db host address
//    "db_port": 6379,                                        // data_db port to reach the da
//    "db_name": "11",                                        // data_db database name to con
//    "db_user": "",                                          // username to use when connect
//    "db_passwd": "",                                        // password to use when connect
//},


//"stor_db": {                                                      // database used to sto
//    "db_type": "mysql",                                     // stor database type to use: <
//    "db_host": "127.0.0.1",                          // the host to connect to
//    "db_port": 3306,                                        // the port to reach the stordb
//    "db_name": "cgrates",                            // stor database name
//    "db_user": "cgrates",                            // username to use when connecting to s
//    "db_passwd": "CGRateS.org",                      // password to use when connecting to stordb
//    "max_open_conns": 100,                                  // maximum database connections opened
//    "max_idle_conns": 10,                                   // maximum database connections idle
//},


//"balancer": {
//    "enabled": false,                                       // start Balancer service: <tru
//},


//"rater": {
//    "enabled": false,                                       // enable Rater service: <true
//    "balancer": "",                                         // register to balancer as work
//    "cdrstats": "",                                         // address where to reach the c
//    "historys": "",                                         // address where to reach the h
//    "pubsubs": "",                                          // address where to reach the p
//    "users": "",                                            // address where to reach the u
//},


//"scheduler": {
//    "enabled": false,                                       // start Scheduler service: <tr
//},
```

```
//"cdrs": {
//    "enabled": false,                                       // start the CDR Server service
//    "extra_fields": [],                                     // extra fields to store in CDR
//    "store_cdrs": true,                                     // store cdrs in storDb
//    "rater": "internal",                                // address where to reach the Rater for
//    "cdrstats": "",                                         // address where to reach the c
//    "reconnects": 5,                                        // number of reconnect attempts
//    "cdr_replication":[],                               // replicate the raw CDR to a number of
//},


//"cdrstats": {
//    "enabled": false,                                       // starts the cdrstats service:
//    "save_interval": "1m",                              // interval to save changed stats into
//},


//"cdre": {
//    "*default": {
//            "cdr_format": "csv",                                                    // exported CDR
//            "field_separator": ",",
//            "data_usage_multiply_factor": 1,                                // multiply data usage
//            "sms_usage_multiply_factor": 1,                                 // multiply data usage
//            "generic_usage_multiply_factor": 1,                                     // multiply dat
//            "cost_multiply_factor": 1,                                              // multiply cos
//            "cost_rounding_decimals": -1,                                   // rounding decimals fo
//            "cost_shift_digits": 0,                                                 // shift digits
//            "mask_destination_id": "MASKED_DESTINATIONS",   // destination id containing called add
//            "mask_length": 0,                                                               // leng
//            "export_dir": "/var/log/cgrates/cdre",                  // path where the exported CDRs
//            "header_fields": [],                                                    // template of
//            "content_fields": [                                                             // temp
//                    {"tag": "CgrId", "cdr_field_id": "cgrid", "type": "cdrfield", "value": "cgrid"
//                    {"tag":"RunId", "cdr_field_id": "mediation_runid", "type": "cdrfield", "value"
//                    {"tag":"Tor", "cdr_field_id": "tor", "type": "cdrfield", "value": "tor"},
//                    {"tag":"AccId", "cdr_field_id": "accid", "type": "cdrfield", "value": "accid"},
//                    {"tag":"ReqType", "cdr_field_id": "reqtype", "type": "cdrfield", "value": "reqt
//                    {"tag":"Direction", "cdr_field_id": "direction", "type": "cdrfield", "value": "
//                    {"tag":"Tenant", "cdr_field_id": "tenant", "type": "cdrfield", "value": "tenant
//                    {"tag":"Category", "cdr_field_id": "category", "type": "cdrfield", "value": "ca
//                    {"tag":"Account", "cdr_field_id": "account", "type": "cdrfield", "value": "acco
//                    {"tag":"Subject", "cdr_field_id": "subject", "type": "cdrfield", "value": "subj
//                    {"tag":"Destination", "cdr_field_id": "destination", "type": "cdrfield", "value
//                    {"tag":"SetupTime", "cdr_field_id": "setup_time", "type": "cdrfield", "value":
//                    {"tag":"AnswerTime", "cdr_field_id": "answer_time", "type": "cdrfield", "value"
//                    {"tag":"Usage", "cdr_field_id": "usage", "type": "cdrfield", "value": "usage"},
//                    {"tag":"Cost", "cdr_field_id": "cost", "type": "cdrfield", "value": "cost"},
//            ],
//            "trailer_fields": [],                                                   // template of
//    }
//},


//"cdrc": {
//    "*default": {
//            "enabled": false,                                                       // enable CDR c
//            "dry_run": false,                                                       // do not send
//            "cdrs": "internal",                                                     // address wher
```

```
//            "cdr_format": "csv",                                                  // CDR file format <csv
//            "field_separator": ",",                                               // separator used in ca
//            "run_delay": 0,                                                             // sleep interv
//            "max_open_files": 1024,                                               // maximum simultaneous
//            "data_usage_multiply_factor": 1024,                                // conversion factor for data u
//            "cdr_in_dir": "/var/log/cgrates/cdrc/in",       // absolute path towards the directory
//            "cdr_out_dir": "/var/log/cgrates/cdrc/out",     // absolute path towards the directory
//            "failed_calls_prefix": "missed_calls",          // used in case of flatstore CDRs to av
//            "cdr_source_id": "freeswitch_csv",                         // free form field, tag identi
//            "cdr_filter": "",                                                           // filter CDR r
//            "partial_record_cache": "10s",                             // duration to cache partial re
//            "header_fields": [],                                                        // template of the impo
//            "content_fields":[                                                          // import conte
//                    {"tag": "tor", "cdr_field_id": "tor", "type": "cdrfield", "value": "2", "mandat
//                    {"tag": "accid", "cdr_field_id": "accid", "type": "cdrfield", "value": "3", "ma
//                    {"tag": "reqtype", "cdr_field_id": "reqtype", "type": "cdrfield", "value": "4",
//                    {"tag": "direction", "cdr_field_id": "direction", "type": "cdrfield", "value":
//                    {"tag": "tenant", "cdr_field_id": "tenant", "type": "cdrfield", "value": "6", "
//                    {"tag": "category", "cdr_field_id": "category", "type": "cdrfield", "value": "
//                    {"tag": "account", "cdr_field_id": "account", "type": "cdrfield", "value": "8",
//                    {"tag": "subject", "cdr_field_id": "subject", "type": "cdrfield", "value": "9",
//                    {"tag": "destination", "cdr_field_id": "destination", "type": "cdrfield", "valu
//                    {"tag": "setup_time", "cdr_field_id": "setup_time", "type": "cdrfield", "value"
//                    {"tag": "answer_time", "cdr_field_id": "answer_time", "type": "cdrfield", "valu
//                    {"tag": "usage", "cdr_field_id": "usage", "type": "cdrfield", "value": "13", "r
//            ],
//            "trailer_fields": [],                                                       // template of
//    }
//},


//"sm_freeswitch": {
//    "enabled": false,                                 // starts SessionManager service: <true|false>
//    "rater": "internal",                        // address where to reach the Rater <""|internal|127.0.
//    "cdrs": "internal",                               // address where to reach CDR Server, empty to
//    "reconnects": 5,                                  // number of reconnect attempts to rater or cdr
//    "create_cdr": false,                        // create CDR out of events and sends them to CDRS comp
//    "cdr_extra_fields": [],                     // extra fields to store in CDRs when creating them
//    "debit_interval": "10s",                    // interval to perform debits on.
//    "min_call_duration": "0s",                  // only authorize calls with allowed duration higher th
//    "max_call_duration": "3h",                  // maximum call duration a prepaid call can last
//    "min_dur_low_balance": "5s",    // threshold which will trigger low balance warnings for prepai
//    "low_balance_ann_file": "",                 // file to be played when low balance is reached for pr
//    "empty_balance_context": "",    // if defined, prepaid calls will be transfered to this context
//    "empty_balance_ann_file": "",   // file to be played before disconnecting prepaid calls on empt
//    "subscribe_park": true,                     // subscribe via fsock to receive park events
//    "channel_sync_interval": "5m",  // sync channels with freeswitch regularly
//    "connections":[                                   // instantiate connections to multiple FreeSWI
//            {"server": "127.0.0.1:8021", "password": "ClueCon", "reconnects": 5}
//    ],
//},


//"sm_kamailio": {
//    "enabled": false,                                 // starts SessionManager service: <true|false>
//    "rater": "internal",                        // address where to reach the Rater <""|internal|127.0.
//    "cdrs": "internal",                               // address where to reach CDR Server, empty to
//    "reconnects": 5,                                  // number of reconnect attempts to rater or cdr
```

```
//    "create_cdr": false,                    // create CDR out of events and sends them to CDRS comp
//    "debit_interval": "10s",                // interval to perform debits on.
//    "min_call_duration": "0s",              // only authorize calls with allowed duration higher th
//    "max_call_duration": "3h",              // maximum call duration a prepaid call can last
//    "connections":[                                 // instantiate connections to multiple Kamailio
//            {"evapi_addr": "127.0.0.1:8448", "reconnects": 5}
//    ],
//},


//"sm_opensips": {
//    "enabled": false,                                       // starts SessionManager service: <true
//    "listen_udp": "127.0.0.1:2020",         // address where to listen for datagram events coming f
//    "rater": "internal",                            // address where to reach the Rater <""|interna
//    "cdrs": "internal",                                     // address where to reach CDR Server, e
//    "reconnects": 5,                                        // number of reconnects if connection f
//    "create_cdr": false,                            // create CDR out of events and sends them to C
//    "debit_interval": "10s",                        // interval to perform debits on.
//    "min_call_duration": "0s",                      // only authorize calls with allowed duration h
//    "max_call_duration": "3h",                      // maximum call duration a prepaid call can las
//    "events_subscribe_interval": "60s",     // automatic events subscription to OpenSIPS, 0 to disa
//    "mi_addr": "127.0.0.1:8020",            // address where to reach OpenSIPS MI to send session c
//},


//"historys": {
//    "enabled": false,                                                       // starts History servi
//    "history_dir": "/var/log/cgrates/history",      // location on disk where to store history file
//    "save_interval": "1s",                                          // interval to save changed cac
//},


//"pubsubs": {
//    "enabled": false,                                                       // starts PubSub servic
//},


//"users": {
//    "enabled": false,                                                       // starts User service
//    "indexes": [],                                  // user profile field indexes
//},


//"mailer": {
//    "server": "localhost",                                                          // the server t
//    "auth_user": "cgrates",                                                         // authenticate
//    "auth_passwd": "CGRateS.org",                                           // authenticate to emai
//    "from_address": "cgr-mailer@localhost.localdomain"      // from address used when sending email
//},


}
```

file ../data/conf/cgrates/cgrates.json

4.2 Tariff Plans

For importing the data into CGRateS database we are using cvs files. The import process can be started as many times it is desired with one ore more csv files and the existing values are overwritten. If the -flush option is used then the database is cleaned before importing.For more details see the cgr-loader tool from the tutorial chapter.

The rest of this section we will describe the content of every csv files.

4.2.1 4.2.1. Rates profile

The rates profile describes the prices to be applied for various calls to various destinations in various time frames. When a call is made the CGRateS system will locate the rates to be applied to the call using the rating profiles.

#Direction	Tenant	Category	Subject	ActivationTime	RatingPlanId	RatesFallbackSubject	CdrStatQueueIds
*out	cgrates.org	call	*any	2014-01-14T00:00:00Z	RP_RETAIL1		
*out	cgrates.org	call	1001	2014-01-14T00:00:00Z	RP_RETAIL2		
*out	cgrates.org	call	SPECIAL_1002	2014-01-14T00:00:00Z	RP_SPECIAL_1002		
*out	cgrates.org	cr_profile1	suppl1	2014-01-14T00:00:00Z	RP_RETAIL1		STATS_SUPPL1
*out	cgrates.org	cr_profile1	suppl2	2014-01-14T00:00:00Z	RP_RETAIL2		STATS_SUPPL2
*out	cgrates.org	cr_profile2	suppl1	2014-01-14T00:00:00Z	RP_RETAIL2		STATS_SUPPL1
*out	cgrates.org	cr_profile2	suppl2	2014-01-14T00:00:00Z	RP_RETAIL1		STATS_SUPPL2
*out	cgrates.org	cr_profile2	suppl3	2014-01-14T00:00:00Z	RP_SPECIAL_1002		

Direction: Can be *in or *out for the INBOUND and OUTBOUND calls.

Tenant: Used to distinguish between carriers if more than one share the same database in the CGRates system.

Category: Type of record specifies the kind of transmission this rate profile applies to.

Subject: The client/user for who this profile is detailing the rates.

ActivationTime: Multiple rates timings/prices can be created for one profile with different activation times. When a call is made the appropriate profile(s) will be used to rate the call. So future prices can be defined here and the activation time can be set as appropriate.

RatingPlanId: This specifies the profile to be used in case the call destination.

RatesFallbackSubject: This specifies another profile to be used in case the call destination will not be found in the current profile. The same tenant, tor and direction will be used.

CdrStatQueueIds: Stat Queue associated with this account

4.2.2 4.2.2. Rating Plans

This file makes links between a ratings and timings so each of them can be described once and various combinations are made possible.

#Id	DestinationRatesId	TimingTag	Weight
RP_RETAIL1	DR_FS_40CNT	PEAK	10
RP_RETAIL1	DR_FS_10CNT	OFFPEAK_MORNING	10
RP_RETAIL1	DR_FS_10CNT	OFFPEAK_EVENING	10
RP_RETAIL1	DR_FS_10CNT	OFFPEAK_WEEKEND	10
RP_RETAIL1	DR_1007_MAXCOST_DISC	ALWAYS	10
RP_RETAIL2	DR_1002_20CNT	PEAK	10
RP_RETAIL2	DR_1003_20CNT	PEAK	10
RP_RETAIL2	DR_FS_40CNT	PEAK	10
RP_RETAIL2	DR_1002_10CNT	OFFPEAK_MORNING	10
RP_RETAIL2	DR_1002_10CNT	OFFPEAK_EVENING	10
RP_RETAIL2	DR_1002_10CNT	OFFPEAK_WEEKEND	10
RP_RETAIL2	DR_1003_10CNT	OFFPEAK_MORNING	10
RP_RETAIL2	DR_1003_10CNT	OFFPEAK_EVENING	10
RP_RETAIL2	DR_1003_10CNT	OFFPEAK_WEEKEND	10
RP_RETAIL2	DR_FS_10CNT	OFFPEAK_MORNING	10
RP_RETAIL2	DR_FS_10CNT	OFFPEAK_EVENING	10
RP_RETAIL2	DR_FS_10CNT	OFFPEAK_WEEKEND	10
RP_RETAIL2	DR_1007_MAXCOST_FREE	ALWAYS	10
RP_SPECIAL_1002	DR_SPECIAL_1002	ALWAYS	10

Tag: A string by which this rates timing will be referenced in other places by.

DestinationRatesTag: The rating tag described in the rates file.

TimingTag: The timing tag described in the timing file

Weight: If multiple timings cab be applied to a call the one with the lower weight wins. An example here can be the Christmas day: we can have a special timing for this day but the regular day of the week timing can also be applied to this day. The weight will differentiate between the two timings.

4.2.3 4.2.3. Rates

Defines price groups for various destinations which will be associated to various timings.

#Id	ConnectFee	Rate	RateUnit	RateIncrement	GroupIntervalStart
RT_10CNT	0.2	0.1	60s	60s	0s
RT_10CNT	0	0.05	60s	1s	60s
RT_20CNT	0.4	0.2	60s	60s	0s
RT_20CNT	0	0.1	60s	1s	60s
RT_40CNT	0.8	0.4	60s	30s	0s
RT_40CNT	0	0.2	60s	10s	60s
RT_1CNT	0	0.01	60s	60s	0s
RT_1CNT_PER_SEC	0	0.01	1s	1s	0s

Tag: A string by which this rate will be referenced in other places by.

ConnectFee: The price to be charged once at the beginning of the call to the specified destination.

Rate: The price for the billing unit expressed in cents.

RateUnit: The billing unit expressed in seconds

RateIncrement: The time gap for the rate

GroupIntervalStart: When the rate starts

See also:

Rateincrement and GroupIntervalStart are when the calls has different rates in the timeframe. For example, the first 30 seconds of the calls has a rate of €0.1 and after that €0.2. The rate for this will the same TAG with two RateIncrements

4.2.4 4.2.4. Timings

Describes the time periods that have different rates attached to them.

#Tag	Years	Months	MonthDays	WeekDays	Time
ALWAYS	*any	*any	*any	*any	00:00:00
ASAP	*any	*any	*any	*any	*asap
PEAK	*any	*any	*any	1;2;3;4;5	08:00:00
OFFPEAK_MORNING	*any	*any	*any	1;2;3;4;5	00:00:00
OFFPEAK_EVENING	*any	*any	*any	1;2;3;4;5	19:00:00
OFFPEAK_WEEKEND	*any	*any	*any	6;7	00:00:00

Tag: A string by which this timing will be referenced in other places by.

Years: Integers or *any in case of always

Months: Integers from 1=January to 12=December separated by semicolons (;) specifying the months for this time period.

MonthDays: Integers from 1 to 31 separated by semicolons (;) specifying the month days for this time period.

WeekDays: Integers from 1=Monday to 7=Sunday separated by semicolons (;) specifying the week days for this time period.

Time: The start time for this time period. *now will be replaced with the time of the data importing.

4.2.5 4.2.5. Destinations

The destinations are binding together various prefixes / caller ids to define a logical destination group. A prefix can appear in multiple destination groups.

#Id	Prefix
DST_1002	1002
DST_1003	1003
DST_1007	1007
DST_FS	10

Tag: A string by which this destination will be referenced in other places by.

Prefix: The prefix or caller id to be added to the specified destination.

4.2.6 4.2.6. Account actions

Describes the actions to be applied to the clients/users accounts. There are two kinds of actions: timed and triggered. For the timed actions there is a scheduler application that reads them from the database and executes them at the appropriate timings. The triggered actions are executed when the specified balance counters reach certain thresholds.

The accounts hold the various balances and counters to activate the triggered actions for each the client.

Balance types are: MONETARY, SMS, INTERNET, INTERNET_TIME, MINUTES.

#Tenant	Account	ActionPlanId	ActionTriggersId	AllowNegative	Disabled
cgrates.org	1001	PACKAGE_1001	STANDARD_TRIGGERS		
cgrates.org	1002	PACKAGE_10	STANDARD_TRIGGERS		
cgrates.org	1003	PACKAGE_10	STANDARD_TRIGGERS		
cgrates.org	1004	PACKAGE_10	STANDARD_TRIGGERS		
cgrates.org	1007	USE_SHARED_A	STANDARD_TRIGGERS		

Tenant: Used to distinguish between carriers if more than one share the same database in the CGRates system.

Account: The identifier for the user's account.

Direction: Can be *in or *out for the INBOUND and OUTBOUND calls.

ActionPlanTag: Forwards to a timed action group that will be used on this account.

ActionTriggersTag: Forwards to a triggered action group that will be applied to this account.

4.2.7 4.2.7 Action triggers

For each account there are counters that record the activity on various balances. Action triggers allow when a counter reaches a threshold to activate a group of actions. After the execution the action trigger is marked as used and will no longer be evaluated until the triggers are reset. See actions for action trigger resetting.

#Tag[0]	UniqueId[1]	ThresholdType[2]	ThresholdValue[3]	Recurrent[4]	MinSleep[5]	BalanceTag[6]	BalanceType[7]	BalanceActions[8]	BalanceDestinationCategories[9]	BalanceTimings[10]	BalanceRatingSubject[11]	BalanceSharedGroup[12]	BalanceExpirationTime[13]	BalanceTimingIds[14]	BalanceWeight[15]	BalanceDestinationIds[16]	StatsMinimumItems[17]	Weight[18]	ActionsId[19]
STANDARD_TRIGGERS		*min_balance	2	false	0		*monetary	*out										10	LOG_WARNING
STANDARD_TRIGGERS		*max_event_counter	5	false	0		*monetary	*out		FS_USERS								10	LOG_WARNING
STANDARD_TRIGGERS		*max_balance	20	false	0		*monetary	*out										10	LOG_WARNING
STANDARD_TRIGGERS		*max_balance	100	false	0		*monetary	*out										10	DISABLE_AND_LOG
CDRST1_WARN		*min_asr	45	true	1m												3	10	LOG_WARNING
CDRST1_WARN		*min_acd	10	true	1m												5	10	LOG_WARNING
CDRST1_WARN		*max_acc	10	true	1m												5	10	LOG_WARNING
CDRST1001_WARN		*min_asr	65	true	1m												3	10	LOG_WARNING
CDRST1001_WARN		*min_acd	10	true	1m												5	10	LOG_WARNING
CDRST1001_WARN		*max_acc	5	true	1m												5	10	LOG_WARNING
CDRST3_WARN		*min_acd	60	false	1m												5	10	LOG_WARNING

Tag: A string by which this action trigger will be referenced in other places by.

UniqueID: Unique id for the trigger in multiple ActionTriggers

ThresholdType: The threshold type. Can have one of the following:

- ***min_counter**: Fire when counter is less than ThresholdValue
- ***max_counter**: Fire when counter is greater than ThresholdValue
- ***min_balance**: Fire when balance is less than ThresholdValue
- ***max_balance**: Fire when balances is greater than ThresholdValue
- ***min_asr**: Fire when ASR(Average success Ratio) is less than ThresholdValue
- ***max_asr**: Fire when ASR is greater than ThresholdValue
- ***min_acd**: Fire when ACD(Average call Duration) is less than ThresholdValue
- ***max_acd**: Fire when ACD is greater than ThresholdValue
- ***min_acc**: Fire when ACC(Average call cost) is less than ThresholdValue
- ***max_acc**: Fire when ACC is greater than ThresholdValue
- ***min_tcc**: Fire when TCC(Total call cost) is less than ThresholdValue
- ***max_tcc**: Fire when TCC is greater than ThresholdValue
- ***min_tcd**: fire when TCD(total call duration) is less than thresholdvalue
- ***max_tcd**: fire when TCD is greater than thresholdvalue
- ***min_pdd**: Fire when PDD(Post Dial Delay) is less than ThresholdValue
- ***max_pdd**: Fire when PDD is greater than ThresholdValue

ThresholdValue: The value of the balance counter that will trigger this action.

Recurrent(Boolean): In case of trigger we can fire recurrent while it's active, or only the first time.

MinSleep: When Threshold is triggered we can sleep for the time specified.

BalanceTag: Specifies the balance counter by which this action will be triggered. Can be:

- **MONETARY**
- **SMS**
- **INTERNET**
- **INTERNET_TIME**
- **MINUTES**

BalanceType: Specifies the balance type for this action:

- ***voice**: units of call minutes
- ***sms**: units of SMS
- ***data**: units of data
- ***monetary**: units of money

BalanceDirection: Can be ***in** or ***out** for the INBOUND and OUTBOUND calls.

BalanceCategory: Category of the call/trigger

BalanceDestinationTag: Destination of the call/trigger

BalanceRatingSubject:

BalanceSharedGroup: Shared Group of the call/trigger

BalanceExpiryTime:

BalanceTimingTags:

BalanceWeight:

StatsMinQueuedItems: Min of items that need to have a queue to reach this Trigger

ActionsTag: Forwards to an action group to be executed when the threshold is reached.

Weight: Specifies the order for these triggers to be evaluated. If there are multiple triggers are fired in the same time the ones with the lower weight will be executed first.

DestinationTag: This field is used only if the balanceTag is MINUTES. If the balance counter monitors call minutes this field indicates the destination of the calls for which the minutes are recorded.a

4.2.8 4.2.8. Action Plans

#Id	ActionsId	TimingId	Weight
PACKAGE_10	TOPUP_RST_10	*asap	10
PACKAGE_10_SHARED_A_5	TOPUP_RST_5	*asap	10
PACKAGE_10_SHARED_A_5	TOPUP_RST_SHARED_5	*asap	10
USE_SHARED_A	SHARED_A_0	*asap	10
PACKAGE_1001	TOPUP_RST_5	*asap	10
PACKAGE_1001	TOPUP_RST_SHARED_5	*asap	10
PACKAGE_1001	TOPUP_120_DST1003	*asap	10

Tag: A string by which this action timing will be referenced in other places by.

ActionsTag: Forwards to an action group to be executed when the timing is right.

TimingTag: A timing (one time or recurrent) at which the action group will be executed

Weight: Specifies the order for these timings to be evaluated. If there are multiple action timings set to be execute on the same time the ones with the lower weight will be executed first.

4.2.9 4.2.9. Actions

#Action-sId[0]	Action[1]	Extra-Parameters[2]	Balanceld[3]	Balance-Type[4]	Direc-tions[5]	Cate-gories[6]	Destination-Ids[7]	Rating-Subject[8]	SharedGroup[9]	Expiry-Time[10]	TimingIds[11]	Units[12]	Balance-Weight[13]	Bal-ance-Disabled[14]	Weight[15]
TOPUP_RST_10	*topup_reset			*mon-e-tary	*out		*any			*un-lim-ited		10	10	false	10
TOPUP_RST_5	*topup_reset			*mon-e-tary	*out		*any			*un-lim-ited		5	20	false	10
TOPUP_RST_5	*topup_reset			*voice	*out		DST_1002	SPE-CIAL_1002		*un-lim-ited		90	20	false	10
TOPUP_120_DST1003	*topup_reset			*voice	*out		DST_1003			*un-lim-ited		120	20	false	10
TOPUP_RST_SHARED_5	*topup			*mon-e-tary	*out		*any		SHARED_A	*un-lim-ited		5	10	false	10
SHARED_A_0	*topup_reset			*mon-e-tary	*out		*any		SHARED_A	*un-lim-ited		0	10	false	10
LOG_WARNING	*log													false	10
DIS-ABLE_AND_LOG	*log													false	10
DIS-ABLE_AND_LOG	*dis-able_account													false	10

Tag A string by which this action will be referenced in other places by.

Action The action type. Can have one of the following:

- ***allow_negative**: Allow to the account to have negative balance
- ***call_url**: Send a http request to the following url
- ***call_url_async**: Send a http request to the following url Asynchronous
- ***cdrlog**: Log the current action in the storeDB
- ***debit**: Debit account balance.
- ***deny_negative**: Deny to the account to have negative balance
- ***disable_account**: Disable account in the platform
- ***enable_account**: Enable account in the platform
- ***log**: Logs the other action values (for debugging purposes).
- ***mail_async**: Send a email to the direction

- ***reset_account**: Sets all counters to 0
- ***reset_counter**: Sets the counter for the BalanceTag to 0
- ***reset_counters**: Sets *all* the counters for the BalanceTag to 0
- ***reset_triggers**: reset all the triggers for this account
- ***set_recurrent**: (pending)
- ***topup**: Add account balance. If the specific balance is not defined, define it (example: minutes per destination).
- ***topup_reset**: Add account balance. If previous balance found of the same type, reset it before adding.
- ***unset_recurrent**: (pending)
- ***unlimited**: (pending)

ExtraParameters: In Extra Parameter field you can define a argument for the action. In case of call_url Action, extraParameter will be the url action. In case of mail_async the email that you want to receive.

BalanceTag: The balance on which the action will operate

Units The units which will be operated on the balance BalanceTag.

BalanceType:

Specifies the balance type for this action:

- ***voice**: units of call minutes
- ***sms**: units of SMS
- ***data**: units of data
- ***monetary**: units of money

BalanceDirection: Can be ***in** or ***out** for the INBOUND and OUTBOUND calls.

DestinationTag: This field is used only if the balanceTag is MINUTES. Specifies the destination of the minutes to be operated.

RatingSubject: The ratingSubject of the Actions

SharedGroup: In case of the account uses any shared group for the balances.

ExpiryTime:

TimingTags: Timming tag when the action can be executed. Default ALL.

Units: Number of units for decrease the balance. Only use if BalanceType is voice.

BalanceWeight:

Weight: If there are multiple actions in a group, they will be executed in the order of their weight (smaller first).

4.2.10 4.2.10. Derived Chargers

For each call we can bill more than one time, for that we need to use the following options:

#Direction[0]	Tenant[1]	Category[2]	Account[3]	Subject[4]	Destination[5]	RunID[6]	ReqTypeField[7]	DirectionField[8]	TenantField[9]	CategoryField[10]	AccountField[11]	SubjectField[12]	DestinationField[13]	SetupTimeField[14]	PddField[15]	AnswerTimeField[16]	UsageField[17]	SupplierField[18]	DisconnectCauseField[19]	RatedField[20]	CostField[21]	
*out	cgrates.org	call	1001	1001		derived_run1	^*rated	*default	*default	*default	*default	^1002	*default	*default	*default	*default	*default	*default	*default	*default	*default	*default

In derived charges we have 2 different kind of options, filters, and actions:

Filters: With the following fields we filter the calls that need to run a extra billing parameter.

- Direction
- Tenant
- Category
- Account
- Subject

Actions: In case of the filter options match, platform creates extra runid with the fields that we want to modify.

- RunId
- RunFilter
- ReqTypeField
- DirectionField
- TenantField
- CategoryField
- AccountField
- SubjectField
- DestinationField
- SetupTimeField
- AnswerTimeField
- UsageField

In the example, all the calls with direction=out, tenant=cgrates.org, category="call" and account and subject equal 1001. Will be created a new cdr in the table *rated_cdrs* with the runID derived_run1, and the subject 1002.

This feature it's useful in the case that you want to rated the calls 2 times, for example rated for different tenants or resellers.

4.2.11 4.2.10. CDR Stats

CDR Stats enabled some realtime statistics in your platform for multiple purposes, you can read more, see *CDR Stats Server*

#Id[0]	QueueLength[1]	TimeWindow[2]	SaveInterval[3]	Metric[4]	SetupInterval[5]	TOR[6]	CdrHost[7]	CdrSource[7]	ReqType[8]	Direction[9]	Tenant[10]	Category[11]	Account[12]	Subject[13]	Destination[14]	Pdd[15]	Usage[16]	Supplier[17]	DisconnectCause[18]	RunId[19]	RatingPlanId[20]	RatingSubject[21]	ActivationInterval[22]	Weight[23]	Triggers[24]
CDRST1	10	10s	ASR								cgrates.org								*default						CDRST1_WARN
CDRST1			ACD																						
CDRST1			ACC																						
CDRST1			TCD																						
CDRST1			TCC																						
CDRST_1001	10	10s	ASR								cgrates.org			1001					*default						CDRST1001_WARN
CDRST_1001			ACD																						
CDRST_1001			ACC																						
CDRST_1002	10	10s	ASR								cgrates.org			1002					*default						CDRST1001_WARN
CDRST_1002			ACD																						
CDRST_1002			ACC																						
CDRST_1003			ASR								cgrates.org				1003				*default						CDRST3_WARN
CDRST_1003			ACD																						
STATS_SUPPL1			ACD															suppl1							
STATS_SUPPL1			ASR															suppl1							
STATS_SUPPL1			ACC															suppl1							
STATS_SUPPL1			TCD															suppl1							
STATS_SUPPL1			TCC															suppl1							
STATS_SUPPL2			ACD															suppl2							
STATS_SUPPL2			ASR															suppl2							
STATS_SUPPL2			ACC															suppl2							
STATS_SUPPL2			TCD															suppl2							
STATS_SUPPL2			TCC															suppl2							

ID: Tag name for the Queue id

QueueLength: Maximum number of calls in this queue

TimeWindow: Window frame to store the calls

Save Interval: Each interval queue stats will save in the stordb

Metric: Type of metric see cdrstats-metric

SetupInterval:

TOR:

CdrHost

CdrSource:

ReqType: Filter by reqtype

Tenant: Used to distinguish between carriers if more than one share the same database in the CGRates system.

Category: Type of record specifies the kind of transmission this rate profile applies to.

Account: The identifier for the user's account.

Subject: The client/user for who this profile is detailing the rates.

DestinationPrefix: Filter only by destinations prefix. Can be multiple separated with ;

PDDInterval:

UsageInterval:

Supplier:

DisconnectCause:

MediationRunids:

RatedAccount: Filter by rated account

RatedSubject: Filter by rated subject

CostInterval Filter by cost

ActionTriggers: ActionTriggers associated with this queue

5. Administration

The general steps to get CGRateS operational are:

1. Create CSV files containing the initial data for CGRateS.
2. Load the data in the databases using the Loader application.
3. Start the a Balancer or a Rater. If Balancer is used, start one or more Raters serving that Balancer.
4. Start the SessionManager talking to your VoIP Switch or directly make API calls to the Balancer/Rater.
5. Make API calls to the Balancer/Rater or just let the SessionManager do the work.

6. Advanced Topics

6.1 API Calls

The general API usage of the CGRateS involves creating a CallDescriptor structure sending it to the balancer via JSON/GOB RPC and getting a response from the balancer in form of a CallCost structure or a numeric value for requested information.

6.1.1 CallDescriptor structure

- Direction, TOR, Tenant, Subject, Account, DestinationPrefix string
- TimeStart, TimeEnd Time
- Amount float64

Direction The direction of the call (inbound or outbound)

TOR Type Of Record, used to differentiate between various type of records

Tenant Customer Identification used for multi tenant databases

Subject Subject for this query

Account Used when different from subject

Destination Destination call id to be matched

TimeStart, TimeEnd The start end end of the call in question

Amount The amount requested in various API calls (e.g. DebitSMS amount)

The **Subject** field is used usually used to identify both the client in the detailed cost list and the user in the balances database. When there is some additional info added to the subject for the call price list then the **Account** attribute is used to specify the balance for the client. For example: the subject can be rif:from:ha or rif:form:mu and for both we would use the rif account.

6.1.2 CallCost structure

- TOR int
- CstmId, Subject, DestinationPrefix string
- Cost, ConnectFee float64
- Timespans []*TimeSpan

TOR Type Of Record, used to differentiate between various type of records (for query identification and confirmation)

CstmId Customer Identification used for multi tenant databases (for query identification and confirmation)

Subject Subject for this query (for query identification and confirmation)

DestinationPrefix Destination prefix to be matched (for query identification and confirmation)

Cost The requested cost

ConnectFee The requested connection cost

Timespans The timespans in witch the initial TimeStart-TimeEnd was split in for cost determination with all pricing and cost information attached.

As stated before the balancer (or the rater directly) can be accesed via json rpc.

The smallest python snippet to acces the CGRateS balancer is this:

```
cd = {"Tor":0,
        "CstmId": "vdf",
        "Subject": "rif",
        "DestinationPrefix": "0256",
        "TimeStart": "2012-02-02T17:30:00Z",
        "TimeEnd": "2012-02-02T18:30:00Z"}

s = socket.create_connection(("127.0.0.1", 2001))
s.sendall(json.dumps({"id": 1, "method": "Responder.Get", "params": [cd]}))
print s.recv(4096)
```

This also gives you a pretty good idea of how JSON-RPC works. You can find details in the specification. A call to a JSON-RPC server simply sends a block of data through a socket. The data is formatted as a JSON structure, and a call consists of an id (so you can sort out the results when they come back), the name of the method to execute on the server, and params, an array of parameters which can itself consist of complex JSON objects. The dumps() call converts the Python structure into JSON.

In the stress folder you can find a better example of python client using a class that reduces the actual call code to:

```
rpc =JSONClient(("127.0.0.1", 2001))
result = rpc.call("Responder.Get", cd)
print result
```

6.1.3 Call API

GetCost Creates a CallCost structure with the cost information calculated for the received CallDescriptor.

Debit Interface method used to add/substract an amount of cents or bonus seconds (as returned by GetCost method) from user's money balance.

MaxDebit Interface method used to add/substract an amount of cents or bonus seconds (as returned by GetCost method) from user's money balance. This methods combines the Debit and GetMaxSessionTime and will debit the max available time as returned by the GetMaxSessionTime method. The amount filed has to be filled in call descriptor.

DebitBalance Interface method used to add/substract an amount of cents from user's money budget. The amount filed has to be filled in call descriptor.

DebitSMS Interface method used to add/substract an amount of units from user's SMS budget. The amount filed has to be filled in call descriptor.

DebitSeconds Interface method used to add/substract an amount of seconds from user's minutes budget. The amount filed has to be filled in call descriptor.

GetMaxSessionTime Returns the approximate max allowed session for user budget. It will try the max amount received in the call descriptor and will decrease it by 10% for nine times. So if the user has little credit it will still allow 10% of the initial amount. If the user has no credit then it will return 0.

AddRecievedCallSeconds Adds the specified amount of seconds to the received call seconds. When the threshold specified in the user's tariff plan is reached then the received call budget is reseted and the bonus specified in the tariff plan is applied. The amount filed has to be filled in call descriptor.

FlushCache Cleans all internal cached (Destinations, RatingProfiles)

6.1.4 Tariff plan importer APIs

These operate on a tpid and are used to import the tariff plan content into storDb.

TariffPlan

ApierV1.GetTPIds

Queries tarrif plan identities gathered from all tables.

Request:

Data:

```
type AttrGetTPIds struct {
}
```

***JSON sample*:**

```
{
 "id": 9,
 "method": "ApierV1.GetTPIds",
 "params": []
}
```

Reply:

Data:

```
[]string
```

***JSON sample*:**

```
{
 "error": null,
 "id": 9,
 "result": [
     "SAMPLE_TP",
     "SAMPLE_TP_2"
 ]
}
```

Errors:

`SERVER_ERROR` - Server error occurred.

`NOT_FOUND` - No tariff plans defined.

Timings

ApierV1.SetTPTiming

Creates a new timing within a tariff plan.

Request:

Data:

```
type ApierTPTiming struct {
      TPid      string // Tariff plan id
      TimingId  string // Timing id
      Years     string // semicolon separated list of years this timing is valid on, \*any suppo
      Months    string // semicolon separated list of months this timing is valid on, \*any supp
      MonthDays string // semicolon separated list of month's days this timing is valid on, \*an
      WeekDays  string // semicolon separated list of week day names this timing is valid on \*a
      Time      string // String representing the time this timing starts on
}
```

Mandatory parameters: `[]string{"TPid", "TimingId", "Years", "Months", "MonthDays", "WeekDays", "Time"}`

JSON sample:

```
{
  "id": 3,
  "method": "ApierV1.SetTPTiming",
  "params": [
      {
          "MonthDays": "*any",
          "Months": "*any",
          "TPid": "TEST_SQL",
          "Time": "00:00:00",
          "TimingId": "ALWAYS",
          "WeekDays": "*any",
          "Years": "*any"
      }
  ]
}
```

Reply:

Data:

```
string
```

Possible answers: `OK` - Success.

JSON sample:

```
{
  "error": null,
  "id": 3,
  "result": "OK"
}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

ApierV1.GetTPTiming

Queries specific Timing on tariff plan.

Request:

Data:

```
type AttrGetTPTiming struct {
     TPid     string // Tariff plan id
     TimingId string // Timing id
}
```

Mandatory parameters: `[]string{"TPid", "TimingId"}`

JSON sample:

```
{
  "id": 5,
  "method": "ApierV1.GetTPTiming",
  "params": [
      {
          "TPid": "TEST_SQL",
          "TimingId": "ALWAYS"
      }
  ]
}
```

Reply:

Data:

```
type ApierTPTiming struct {
      TPid      string // Tariff plan id
      TimingId  string // Timing id
      Years     string // semicolon separated list of years this timing is valid on, \*any suppo
      Months    string // semicolon separated list of months this timing is valid on, \*any supp
      MonthDays string // semicolon separated list of month's days this timing is valid on, \*an
      WeekDays  string // semicolon separated list of week day names this timing is valid on \*a
      Time      string // String representing the time this timing starts on
}
```

JSON sample:

```
{
  "error": null,
  "id": 5,
  "result": {
      "MonthDays": "*any",
      "Months": "*any",
      "TPid": "TEST_SQL",
      "Time": "00:00:00",
      "TimingId": "ALWAYS2",
      "WeekDays": "*any",
      "Years": "*any"
  }
}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`NOT_FOUND` - Requested timing id not found.

ApierV1.GetTPTimingIds

Queries timing identities on tariff plan.

Request:

Data:

```
type AttrGetTPTimingIds struct {
      TPid string // Tariff plan id
}
```

Mandatory parameters: `[]string{"TPid"}`

JSON sample:

```
{
  "id": 4,
  "method": "ApierV1.GetTPTimingIds",
  "params": [
      {
          "TPid": "CGR_API_TESTS"
      }
  ]
}
```

Reply:

Data:

```
[]string
```

JSON sample:

```
{
  "error": null,
  "id": 4,
  "result": [
      "ASAP"
  ]
}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`NOT_FOUND` - Requested tariff plan not found.

Destinations

ApierV1.SetTPDestination

Creates a new destination within a tariff plan id.

Request:

Data:

```
type ApierTPDestination struct {
     TPid          string   // Tariff plan id
     DestinationId string   // Destination id
     Prefixes      []string // Prefixes attached to this destination
}
```

Required parameters: `[]string{"TPid", "DestinationId", "Prefixes"}`

***JSON sample*:**

```
{
 "id": 6,
 "method": "ApierV1.SetTPDestination",
 "params": [
     {
         "DestinationId": "FS_USERS",
         "Prefixes": [
             "10"
         ],
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
string
```

Possible answers: `OK` - Success.

***JSON sample*:**

```
{
 "error": null,
 "id": 6,
 "result": "OK"
}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`DUPLICATE` - The specified combination of TPid/DestinationId already exists in StorDb.

ApierV1.GetTPDestination

Queries a specific destination.

Request:

Data:

```
type AttrGetTPDestination struct {
     TPid          string // Tariff plan id
     DestinationId string // Destination id
}
```

Required parameters: `[]string{"TPid", "DestinationId"}`

***JSON sample*:**

```
{
 "id": 7,
 "method": "ApierV1.GetTPDestination",
 "params": [
     {
         "DestinationId": "FS_USERS",
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
type ApierTPDestination struct {
     TPid          string   // Tariff plan id
     DestinationId string   // Destination id
     Prefixes      []string // Prefixes attached to this destination
}
```

***JSON sample*:**

```
{
 "error": null,
 "id": 7,
 "result": {
     "DestinationId": "FS_USERS",
     "Prefixes": [
         "10"
     ],
     "TPid": "CGR_API_TESTS"
 }
}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`NOT_FOUND` - Requested destination id not found.

ApierV1.GetTPDestinationIds

Queries destination identities on specific tariff plan.

Request:

Data:

```
type AttrGetTPDestinationIds struct {
     TPid string // Tariff plan id
}
```

Required parameters: `[]string{"TPid"}`

***JSON sample*:**

```
{
 "id": 8,
 "method": "ApierV1.GetTPDestinationIds",
 "params": [
     {
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
[]string
```

***JSON sample*:**

```
{
 "error": null,
 "id": 8,
 "result": [
     "FS_USERS"
 ]

}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`NOT_FOUND` - Requested tariff plan not found.

Rates

ApierV1.SetTPRate

Creates a new rate within a tariff plan.

Request:

Data:

```
type TPRate struct {
     TPid      string     // Tariff plan id
     RateId    string     // Rate id
     RateSlots []RateSlot // One or more RateSlots
}

type RateSlot struct {
     ConnectFee         float64 // ConnectFee applied once the call is answered
     Rate               float64 // Rate applied
     RateUnit           string  //  Number of billing units this rate applies to
     RateIncrement      string  // This rate will apply in increments of duration
     GroupIntervalStart string  // Group position
     RoundingMethod     string  // Use this method to round the cost
```

```
        RoundingDecimals    int     // Round the cost number of decimals
    }
```

Mandatory parameters: `[]string{"TPid", "RateId", "ConnectFee", "RateSlots"}`

***JSON sample*:**

```
{
 "id": 2,
 "method": "ApierV1.SetTPRate",
 "params": [
     {
         "RateId": "1CENTPERSEC",
         "RateSlots": [
             {
                 "ConnectFee": 0,
                 "GroupIntervalStart": "0",
                 "Rate": 0.01,
                 "RateIncrement": "1s",
                 "RateUnit": "1s",
                 "RoundingDecimals": 4,
                 "RoundingMethod": "*middle",
                 "Weight": 10
             }
         ],
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
string
```

Possible answers: `OK` - Success.

***JSON sample*:**

```
{
 "error": null,
 "id": 0,
 "result": "OK"
}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`DUPLICATE` - The specified combination of TPid/RateId already exists in StorDb.

ApierV1.GetTPRate

Queries specific rate on tariff plan.

Request:

Data:

```
type AttrGetTPRate struct {
     TPid   string // Tariff plan id
     RateId string // Rate id
}
```

Mandatory parameters: `[]string{"TPid", "RateId"}`

***JSON sample*:**

```
{
 "id": 3,
 "method": "ApierV1.GetTPRate",
 "params": [
     {
         "RateId": "1CENTPERSEC",
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
type TPRate struct {
     TPid      string     // Tariff plan id
     RateId    string     // Rate id
     RateSlots []RateSlot // One or more RateSlots
}

type RateSlot struct {
     ConnectFee         float64 // ConnectFee applied once the call is answered
     Rate               float64 // Rate applied
     RateUnit           string  //  Number of billing units this rate applies to
     RateIncrement      string  // This rate will apply in increments of duration
     GroupIntervalStart string  // Group position
     RoundingMethod     string  // Use this method to round the cost
     RoundingDecimals   int     // Round the cost number of decimals
}
```

***JSON sample*:**

```
{
 "error": null,
 "id": 3,
 "result": {
     "RateId": "1CENTPERSEC",
     "RateSlots": [
         {
             "ConnectFee": 0,
             "GroupIntervalStart": "0",
             "Rate": 0.01,
             "RateIncrement": "1s",
             "RateUnit": "1s",
             "RoundingDecimals": 4,
             "RoundingMethod": "*middle"
         }
     ],
     "TPid": "CGR_API_TESTS"
```

```
        }
    }
```

Errors:

MANDATORY_IE_MISSING - Mandatory parameter missing from request.

SERVER_ERROR - Server error occurred.

NOT_FOUND - Requested rate id not found.

ApierV1.GetTPRateIds

Queries rate identities on tariff plan.

Request:

Data:

```
type AttrGetTPRateIds struct {
    TPid string // Tariff plan id
}
```

Mandatory parameters: `[]string{"TPid"}`

***JSON sample*:**

```
{
 "id": 4,
 "method": "ApierV1.GetTPRateIds",
 "params": [
     {
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
[]string
```

***JSON sample*:**

```
{
 "error": null,
 "id": 4,
 "result": [
     "1CENTPERSEC"
 ]
}
```

Errors:

MANDATORY_IE_MISSING - Mandatory parameter missing from request.

SERVER_ERROR - Server error occurred.

NOT_FOUND - Requested tariff plan not found.

DestinationRates

ApierV1.SetTPDestinationRate

Creates a new DestinationRate profile within a tariff plan.

Request:

Data:

```
type TPDestinationRate struct {
     TPid              string // Tariff plan id
     DestinationRateId string // DestinationRate profile id
     DestinationRates     []DestinationRate // Set of destinationid-rateid bindings
}

type DestinationRate struct {
     DestinationId string // The destination identity
     RateId          string // The rate identity
}
```

Mandatory parameters: `[]string{"TPid", "DestinationRateId", "DestinationRates"}`

***JSON sample*:**

```
{
 "id": 7,
 "method": "ApierV1.SetTPDestinationRate",
 "params": [
     {
          "DestinationRateId": "DR_1CENTPERSEC",
          "DestinationRates": [
              {
                   "DestinationId": "FS_USERS",
                   "RateId": "1CENTPERSEC"
              }
          ],
          "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
string
```

Possible answers: `OK` - Success.

***JSON sample*:**

```
{
 "error": null,
 "id": 7,
 "result": "OK"
}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`DUPLICATE` - The specified combination of TPid/DestinationRateId already exists in StorDb.

ApierV1.GetTPDestinationRate

Queries specific DestinationRate profile on tariff plan.

Request:

Data:

```
type AttrGetTPDestinationRate struct {
     TPid   string // Tariff plan id
     DestinationRateId string // Rate id
}
```

Mandatory parameters: `[]string{"TPid", "DestinationRateId"}`

***JSON sample*:**

```
{
 "id": 8,
 "method": "ApierV1.GetTPDestinationRate",
 "params": [
     {
         "DestinationRateId": "DR_1CENTPERSEC",
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
type TPDestinationRate struct {
     TPid              string // Tariff plan id
     DestinationRateId string // DestinationRate profile id
     DestinationRates     []DestinationRate // Set of destinationid-rateid bindings
}

type DestinationRate struct {
     DestinationId string // The destination identity
     RateId          string // The rate identity
}
```

***JSON sample*:**

```
{
 "error": null,
 "id": 8,
 "result": {
     "DestinationRateId": "DR_1CENTPERSEC",
     "DestinationRates": [
         {
             "DestinationId": "FS_USERS",
             "RateId": "1CENTPERSEC"
         }
```

```
            ],
            "TPid": "CGR_API_TESTS"
        }
```

}

Errors:

MANDATORY_IE_MISSING - Mandatory parameter missing from request.

SERVER_ERROR - Server error occurred.

NOT_FOUND - Requested DestinationRate id not found.

ApierV1.GetTPDestinationRateIds

Queries DestinationRate identities on specific tariff plan.

Request:

Data:

```
type AttrTPDestinationRateIds struct {
    TPid string // Tariff plan id
}
```

Mandatory parameters: `[]string{"TPid"}`

***JSON sample*:**

```
{
 "id": 9,
 "method": "ApierV1.GetTPDestinationRateIds",
 "params": [
     {
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
[]string
```

***JSON sample*:**

```
{
 "error": null,
 "id": 9,
 "result": [
     "DR_1CENTPERSEC"
 ]
}
```

Errors:

MANDATORY_IE_MISSING - Mandatory parameter missing from request.

SERVER_ERROR - Server error occurred.

NOT_FOUND - Requested tariff plan not found.

DestinationRateTimings

ApierV1.SetTPDestRateTiming

Creates a new DestinationRateTiming profile within a tariff plan.

Request:

Data:

```
type TPDestRateTiming struct {
     TPid             string           // Tariff plan id
     DestRateTimingId string           // DestinationRate profile id
     DestRateTimings  []DestRateTiming // Set of destinationid-rateid bindings
}

type DestRateTiming struct {
     DestRatesId string  // The DestinationRate identity
     TimingId    string  // The timing identity
     Weight      float64 // Binding priority taken into consideration when more DestinationR
}
```

Mandatory parameters: `[]string{"TPid", "DestRateTimingId", "DestRateTimings"}`

***JSON sample*:**

```
{
 "id": 10,
 "method": "ApierV1.SetTPDestRateTiming",
 "params": [
     {
          "DestRateTimingId": "DRT_1CENTPERSEC",
          "DestRateTimings": [
              {
                   "DestRatesId": "DR_1CENTPERSEC",
                   "TimingId": "ALWAYS",
                   "Weight": 10
              }
          ],
          "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
string
```

Possible answers: `OK` - Success.

***JSON sample*:**

```
{
 "error": null,
 "id": 10,
 "result": "OK"
}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`DUPLICATE` - The specified combination of TPid/DestRateTimingId already exists in StorDb.

ApierV1.GetTPDestRateTiming

Queries specific DestRateTiming profile on tariff plan.

Request:

Data:

```
type AttrGetTPDestRateTiming struct {
     TPid             string // Tariff plan id
     DestRateTimingId string // Rate id
}
```

Mandatory parameters: `[]string{"TPid", "DestRateTimingId"}`

***JSON sample*:**

```
{
 "id": 11,
 "method": "ApierV1.GetTPDestRateTiming",
 "params": [
     {
         "DestRateTimingId": "DRT_1CENTPERSEC",
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
type TPDestRateTiming struct {
     TPid             string           // Tariff plan id
     DestRateTimingId string           // DestinationRate profile id
     DestRateTimings  []DestRateTiming // Set of destinationid-rateid bindings
}

type DestRateTiming struct {
     DestRatesId string  // The DestinationRate identity
     TimingId    string  // The timing identity
     Weight      float64 // Binding priority taken into consideration when more DestinationR
}
```

***JSON sample*:**

```
{
 "error": null,
 "id": 11,
 "result": {
     "DestRateTimingId": "DRT_1CENTPERSEC",
     "DestRateTimings": [
         {
```

```
                    "DestRatesId": "DR_1CENTPERSEC",
                    "TimingId": "ALWAYS",
                    "Weight": 10
                }
            ],
            "TPid": "CGR_API_TESTS"
        }
    }
```

Errors:

MANDATORY_IE_MISSING - Mandatory parameter missing from request.

SERVER_ERROR - Server error occurred.

NOT_FOUND - Requested DestRateTiming profile not found.

ApierV1.GetTPDestRateTimingIds

Queries DestRateTiming identities on specific tariff plan.

Request:

Data:

```
type AttrTPDestRateTimingIds struct {
    TPid string // Tariff plan id
}
```

Mandatory parameters: `[]string{"TPid"}`

***JSON sample*:**

```
{
 "id": 12,
 "method": "ApierV1.GetTPDestRateTimingIds",
 "params": [
     {
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
[]string
```

***JSON sample*:**

```
{
 "error": null,
 "id": 12,
 "result": [
     "DRT_1CENTPERSEC"
 ]
}
```

Errors:

MANDATORY_IE_MISSING - Mandatory parameter missing from request.

SERVER_ERROR - Server error occurred.

NOT_FOUND - Requested tariff plan not found.

RatingProfiles

ApierV1.SetTPRatingProfile

Creates a new RatingProfile within a tariff plan.

Request:

Data:

```
type TPRatingProfile struct {
     TPid                 string             // Tariff plan id
     RatingProfileId        string             // RatingProfile id
     Tenant               string             // Tenant's Id
     TOR                  string             // TypeOfRecord
     Direction            string             // Traffic direction, *out is the only one supp
     Subject              string             // Rating subject, usually the same as account
     RatesFallbackSubject string             // Fallback on this subject if rates not found
     RatingActivations    []RatingActivation // Activate rate profiles at specific time
}

type RatingActivation struct {
     ActivationTime   int64  // Time when this profile will become active, defined as unix e
     DestRateTimingId string // Id of DestRateTiming profile
}
```

Mandatory parameters: `[]string{"TPid", "RatingProfileId", "Tenant", "TOR", "Direction", "Subject", "RatingActivations"}`

***JSON sample*:**

```
{
 "id": 14,
 "method": "ApierV1.SetTPRatingProfile",
 "params": [
     {
         "Direction": "*out",
         "RatesFallbackSubject": "",
         "RatingActivations": [
             {
                 "ActivationTime": "2012-01-01T00:00:00Z",
                 "DestRateTimingId": "DRT_1CENTPERSEC"
             }
         ],
         "RatingProfileId": "RP_ANY",
         "Subject": "*any",
         "TOR": "call",
         "TPid": "CGR_API_TESTS",
         "Tenant": "cgrates.org"
     }
 ]
}
```

Reply:

Data:

```
string
```

Possible answers: `OK` - Success.

***JSON sample*:**

```
{
 "error": null,
 "id": 14,
 "result": "OK"
}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`DUPLICATE` - The specified combination of TPid/RatingProfileId already exists in StorDb.

ApierV1.GetTPRatingProfile

Queries specific RatingProfile on tariff plan.

Request:

Data:

```
type AttrGetTPRatingProfile struct {
     TPid             string // Tariff plan id
     RatingProfileId    string // RatingProfile id
}
```

Mandatory parameters: `[]string{"TPid", "RatingProfileId"}`

***JSON sample*:**

```
{
 "id": 15,
 "method": "ApierV1.GetTPRatingProfile",
 "params": [
     {
         "RatingProfileId": "RP_ANY",
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
type TPRatingProfile struct {
     TPid                 string             // Tariff plan id
     RatingProfileId      string             // RatingProfile id
     Tenant               string             // Tenant's Id
     TOR                  string             // TypeOfRecord
     Direction            string             // Traffic direction, *out is the only one supp
     Subject              string             // Rating subject, usually the same as account
     RatesFallbackSubject string             // Fallback on this subject if rates not found
```

```
        RatingActivations    []RatingActivation // Activate rate profiles at specific time
}

type RatingActivation struct {
        ActivationTime   int64  // Time when this profile will become active, defined as unix e
        DestRateTimingId string // Id of DestRateTiming profile
}
```

***JSON sample*:**

```
{
 "error": null,
 "id": 15,
 "result": {
     "Direction": "*out",
     "RatesFallbackSubject": "",
     "RatingActivations": [
         {
             "ActivationTime": "2012-01-01T00:00:00Z",
             "DestRateTimingId": "DRT_1CENTPERSEC"
         }
     ],
     "RatingProfileId": "RP_ANY",
     "Subject": "*any",
     "TOR": "call",
     "TPid": "CGR_API_TESTS",
     "Tenant": "cgrates.org"
 }
}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`NOT_FOUND` - Requested RatingProfile profile not found.

ApierV1.GetTPRatingProfileIds

Queries specific RatingProfile on tariff plan. Attribute parameters used as extra filters.

Request:

Data:

```
type AttrTPRatingProfileIds struct {
        TPid      string // Tariff plan id
        Tenant    string // Tenant's Id
        TOR       string // TypeOfRecord
        Direction string // Traffic direction
        Subject   string // Rating subject, usually the same as account
}
```

Mandatory parameters: `[]string{"TPid"}`

***JSON sample*:**

```
{
 "id": 16,
```

```
 "method": "ApierV1.GetTPRatingProfileIds",
 "params": [
     {
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
[]string
```

***JSON sample*:**

```
{
 "error": null,
 "id": 16,
 "result": [
     "RP_ANY"
 ]
}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`NOT_FOUND` - There is no data to be returned based on filters set.

Actions

ApierV1.SetTPActions

Creates a new Actions profile within a tariff plan.

Request:

Data:

```
type TPActions struct {
     TPid      string   // Tariff plan id
     ActionsId string   // Actions id
     Actions   []Action // Set of actions this Actions profile will perform
}

type Action struct {
     Identifier      string  // Identifier mapped in the code
     BalanceType     string  // Type of balance the action will operate on
     Direction       string  // Balance direction
     Units           float64 // Number of units to add/deduct
     ExpiryTime      string  // Time when the units will expire
     DestinationId   string  // Destination profile id
     RatingSubject   string  // Reference a rate subject defined in RatingProfiles
     BalanceWeight   float64 // Balance weight
     ExtraParameters string
     Weight          float64 // Action's weight
}
```

Mandatory parameters: `[]string{"TPid", "ActionsId", "Actions", "Identifier", "Weight"}`

***JSON sample*:**

```
{
 "id": 39,
 "method": "ApierV1.SetTPActions",
 "params": [
     {
         "Actions": [
             {
                 "BalanceType": "*monetary",
                 "BalanceWeight": 0,
                 "DestinationId": "*any",
                 "Direction": "*out",
                 "ExpiryTime": "0",
                 "Identifier": "*topup_reset",
                 "RatingSubject": "",
                 "Units": 10,
                 "Weight": 10
             }
         ],
         "ActionsId": "TOPUP_10",
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
string
```

Possible answers: `OK` - Success.

***JSON sample*:**

```
{
 "error": null,
 "id": 39,
 "result": "OK"
}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`DUPLICATE` - The specified combination of TPid/ActionsId already present in StorDb.

ApierV1.GetTPActions

Queries specific Actions profile on tariff plan.

Request:

Data:

```
type AttrGetTPActions struct {
     TPid      string // Tariff plan id
     ActionsId string // Actions id
}
```

Mandatory parameters: `[]string{"TPid", "ActionsId"}`

***JSON sample*:**

```
{
 "id": 40,
 "method": "ApierV1.GetTPActions",
 "params": [
     {
         "ActionsId": "TOPUP_10",
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
type TPActions struct {
     TPid      string   // Tariff plan id
     ActionsId string   // Actions id
     Actions   []Action // Set of actions this Actions profile will perform
}

type Action struct {
     Identifier      string  // Identifier mapped in the code
     BalanceType     string  // Type of balance the action will operate on
     Direction       string  // Balance direction
     Units           float64 // Number of units to add/deduct
     ExpiryTime      string  // Time when the units will expire
     DestinationId   string  // Destination profile id
     RatingSubject   string  // Reference a rate subject defined in RatingProfiles
     BalanceWeight   float64 // Balance weight
     ExtraParameters string
     Weight          float64 // Action's weight
}
```

***JSON sample*:**

```
{
 "error": null,
 "id": 40,
 "result": {
     "Actions": [
         {
             "BalanceType": "*monetary",
             "BalanceWeight": 0,
             "DestinationId": "*any",
             "Direction": "*out",
             "ExpiryTime": "0",
             "ExtraParameters": "",
             "Identifier": "*topup_reset",
             "RatingSubject": "",
             "Units": 10,
```

```
                    "Weight": 10
                }
            ],
            "ActionsId": "TOPUP_10",
            "TPid": "CGR_API_TESTS"
        }
       }
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`NOT_FOUND` - Requested Actions profile not found.

ApierV1.GetTPActionIds

Queries Actions identities on specific tariff plan.

Request:

Data:

```
type AttrGetTPActionIds struct {
     TPid string // Tariff plan id
}
```

Mandatory parameters: `[]string{"TPid"}`

***JSON sample*:**

```
{
 "id": 41,
 "method": "ApierV1.GetTPActionIds",
 "params": [
     {
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
[]string
```

***JSON sample*:**

```
{
 "error": null,
 "id": 41,
 "result": [
     "TOPUP_10"
 ]
}
```

Errors:

MANDATORY_IE_MISSING - Mandatory parameter missing from request.

SERVER_ERROR - Server error occurred.

NOT_FOUND - There are no Actions profiles defined on the selected TPid.

ActionTimings

ApierV1.SetTPActionTimings

Creates a new ActionTimings profile within a tariff plan.

Request:

Data:

```
type ApiTPActionTimings struct {
     TPid            string         // Tariff plan id
     ActionTimingsId string         // ActionTimings id
     ActionTimings   []ApiActionTiming // Set of ActionTiming bindings this profile will gro
}

type ApiActionTiming struct {
     ActionsId string  // Actions id
     TimingId  string  // Timing profile id
     Weight    float64 // Binding's weight
}
```

Mandatory parameters: `[]string{"TPid", "ActionTimingsId", "ActionTimings", "ActionsId", "TimingId", "Weight"}`

***JSON sample*:**

```
{
 "id": 42,
 "method": "ApierV1.SetTPActionTimings",
 "params": [
     {
         "ActionTimings": [
             {
                 "ActionsId": "TOPUP_10",
                 "TimingId": "ASAP",
                 "Weight": 10
             }
         ],
         "ActionTimingsId": "AT_FS10",
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
string
```

Possible answers: OK - Success.

***JSON sample*:**

```
{
 "error": null,
 "id": 42,
 "result": "OK"
}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`DUPLICATE` - The specified combination of TPid/ActionTimingsId already present in StorDb.

ApierV1.GetTPActionTimings

Queries specific ActionTimings profile on tariff plan.

Request:

Data:

```
type AttrGetTPActionTimings struct {
     TPid      string // Tariff plan id
     ActionTimingsId string // ActionTimings id
}
```

Mandatory parameters: `[]string{"TPid", "ActionTimingsId"}`

***JSON sample*:**

```
{
 "id": 43,
 "method": "ApierV1.GetTPActionTimings",
 "params": [
     {
         "ActionTimingsId": "AT_FS10",
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
type ApiTPActionTimings struct {
     TPid            string         // Tariff plan id
     ActionTimingsId string         // ActionTimings id
     ActionTimings   []ApiActionTiming // Set of ActionTiming bindings this profile will gro
}

type ApiActionTiming struct {
     ActionsId string  // Actions id
     TimingId  string  // Timing profile id
     Weight    float64 // Binding's weight
}
```

***JSON sample*:**

```
{
 "error": null,
 "id": 43,
 "result": {
     "ActionTimings": [
         {
             "ActionsId": "TOPUP_10",
             "TimingId": "ASAP",
             "Weight": 10
         }
     ],
     "ActionTimingsId": "AT_FS10",
     "TPid": "CGR_API_TESTS"
 }
}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`NOT_FOUND` - Requested ActionTimings profile not found.

ApierV1.GetTPActionTimingIds

Queries ActionTimings identities on specific tariff plan.

Request:

Data:

```
type AttrGetTPActionTimingIds struct {
     TPid string // Tariff plan id
}
```

Mandatory parameters: `[]string{"TPid"}`

***JSON sample*:**

```
{
 "id": 44,
 "method": "ApierV1.GetTPActionTimingIds",
 "params": [
     {
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
[]string
```

***JSON sample*:**

```
{
 "error": null,
 "id": 44,
```

```
        "result": [
            "AT_FS10"
        ]
    }
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`NOT_FOUND` - There are no ActionTimings profiles defined on the selected TPid.

ActionTriggers

ApierV1.SetTPActionTriggers

Creates a new ActionTriggers profile within a tariff plan.

Request:

Data:

```
type ApiTPActionTriggers struct {
    TPid             string             // Tariff plan id
    ActionTriggersId string             // Profile id
    ActionTriggers   []ApiActionTrigger // Set of triggers grouped in this profile

}

type ApiActionTrigger struct {
    BalanceType      string  // Id of the balance this trigger monitors
    Direction      string  // Traffic direction
    ThresholdType  string  // This threshold type
    ThresholdValue float64 // Threshold
    DestinationId  string  // Id of the destination profile
    ActionsId      string  // Actions which will execute on threshold reached
    Weight         float64 // weight
}
```

Mandatory parameters: `[]string{"TPid", "ActionTriggersId","BalanceType", "Direction", "ThresholdType", "ThresholdValue", "ActionsId", "Weight"}`

***JSON sample*:**

```
{
    "id": 45,
    "method": "ApierV1.SetTPActionTriggers",
    "params": [
        {
            "ActionTriggers": [
                {
                    "ActionsId": "LOG_BALANCE",
                    "BalanceType": "*monetary",
                    "DestinationId": "",
                    "Direction": "*out",
                    "ThresholdType": "*min_balance",
                    "ThresholdValue": 2,
```

```
                        "Weight": 10
                    },
                    {
                        "ActionsId": "LOG_BALANCE",
                        "BalanceType": "*monetary",
                        "DestinationId": "",
                        "Direction": "*out",
                        "ThresholdType": "*max_balance",
                        "ThresholdValue": 20,
                        "Weight": 10
                    },
                    {
                        "ActionsId": "LOG_BALANCE",
                        "BalanceType": "*monetary",
                        "DestinationId": "FS_USERS",
                        "Direction": "*out",
                        "ThresholdType": "*max_counter",
                        "ThresholdValue": 15,
                        "Weight": 10
                    }
                ],
                "ActionTriggersId": "STANDARD_TRIGGERS",
                "TPid": "CGR_API_TESTS"
            }
        ]
       }
```

Reply:

Data:

```
    string
```

Possible answers: `OK` - Success.

***JSON sample*:**

```
    {
     "error": null,
     "id": 45,
     "result": "OK"
    }
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`DUPLICATE` - The specified combination of TPid/ActionTriggersId already present in StorDb.

ApierV1.GetTPActionTriggers

Queries specific ActionTriggers profile on tariff plan.

Request:

Data:

```
type AttrGetTPActionTriggers struct {
     TPid             string // Tariff plan id
     ActionTriggersId string // ActionTrigger id
}
```

Mandatory parameters: `[]string{"TPid", "ActionTriggersId"}`

***JSON sample*:**

```
{
 "id": 46,
 "method": "ApierV1.GetTPActionTriggers",
 "params": [
     {
         "ActionTriggersId": "STANDARD_TRIGGERS",
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
type ApiTPActionTriggers struct {
     TPid             string             // Tariff plan id
     ActionTriggersId string             // Profile id
     ActionTriggers   []ApiActionTrigger // Set of triggers grouped in this profile

}

type ApiActionTrigger struct {
     BalanceType      string  // Id of the balance this trigger monitors
     Direction      string  // Traffic direction
     ThresholdType  string  // This threshold type
     ThresholdValue float64 // Threshold
     DestinationId  string  // Id of the destination profile
     ActionsId      string  // Actions which will execute on threshold reached
     Weight         float64 // weight
}
```

***JSON sample*:**

```
{
 "error": null,
 "id": 46,
 "result": {
     "ActionTriggers": [
         {
             "ActionsId": "LOG_BALANCE",
             "BalanceType": "*monetary",
             "DestinationId": "",
             "Direction": "*out",
             "ThresholdType": "*min_balance",
             "ThresholdValue": 2,
             "Weight": 10
         },
         {
             "ActionsId": "LOG_BALANCE",
             "BalanceType": "*monetary",
```

```
                    "DestinationId": "",
                    "Direction": "*out",
                    "ThresholdType": "*max_balance",
                    "ThresholdValue": 20,
                    "Weight": 10
                },
                {
                    "ActionsId": "LOG_BALANCE",
                    "BalanceType": "*monetary",
                    "DestinationId": "FS_USERS",
                    "Direction": "*out",
                    "ThresholdType": "*max_counter",
                    "ThresholdValue": 15,
                    "Weight": 10
                }
            ],
            "ActionTriggersId": "STANDARD_TRIGGERS",
            "TPid": "CGR_API_TESTS"
        }
    }
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`NOT_FOUND` - Requested ActionTriggersId profile not found.

ApierV1.GetTPActionTriggerIds

Queries ActionTriggers identities on specific tariff plan.

Request:

Data:

```
type AttrGetTPActionTriggerIds struct {
    TPid string // Tariff plan id
}
```

Mandatory parameters: `[]string{"TPid"}`

***JSON sample*:**

```
{
 "id": 47,
 "method": "ApierV1.GetTPActionTriggerIds",
 "params": [
     {
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
[]string
```

***JSON sample*:**

```
{
 "error": null,
 "id": 47,
 "result": [
     "STANDARD_TRIGGERS"
 ]
}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`NOT_FOUND` - There are no ActionTriggers profiles defined on the selected TPid.

AccountActions

ApierV1.SetTPAccountActions

Creates a new AccountActions profile within a tariff plan.

Request:

Data:

```
type ApiTPAccountActions struct {
     TPid             string // Tariff plan id
     AccountActionsId string // AccountActions id
     Tenant           string // Tenant's Id
     Account          string // Account name
     Direction        string // Traffic direction
     ActionTimingsId  string // Id of ActionTimings profile to use
     ActionTriggersId string // Id of ActionTriggers profile to use
}
```

Mandatory parameters: `[]string{"TPid", "AccountActionsId","Tenant","Account","Direction","Act`

***JSON sample*:**

```
{
 "id": 48,
 "method": "ApierV1.SetTPAccountActions",
 "params": [
     {
         "Account": "1005",
         "AccountActionsId": "AA_1005",
         "ActionTimingsId": "AT_FS10",
         "ActionTriggersId": "STANDARD_TRIGGERS",
         "Direction": "*out",
         "TPid": "CGR_API_TESTS",
         "Tenant": "cgrates.org"
     }
 ]
}
```

Reply:

Data:

```
string
```

Possible answers: `OK` - Success.

***JSON sample*:**

```
{
 "error": null,
 "id": 48,
 "result": "OK"
}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`DUPLICATE` - The specified combination of TPid/AccountActionsId already present in StorDb.

ApierV1.GetTPAccountActions

Queries specific AccountActions profile on tariff plan.

Request:

Data:

```
type AttrGetTPAccountActions struct {
     TPid             string // Tariff plan id
     AccountActionsId string // AccountActions id
}
```

Mandatory parameters: `[]string{"TPid", "AccountActionsId"}`

***JSON sample*:**

```
{
 "id": 49,
 "method": "ApierV1.GetTPAccountActions",
 "params": [
     {
         "AccountActionsId": "AA_1005",
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
type ApiTPAccountActions struct {
     TPid             string // Tariff plan id
     AccountActionsId string // AccountActions id
     Tenant           string // Tenant's Id
     Account          string // Account name
     Direction        string // Traffic direction
```

```
        ActionTimingsId  string // Id of ActionTimings profile to use
        ActionTriggersId string // Id of ActionTriggers profile to use
    }
```

***JSON sample*:**

```
    {
     "error": null,
     "id": 49,
     "result": {
         "Account": "1005",
         "AccountActionsId": "AA_1005",
         "ActionTimingsId": "AT_FS10",
         "ActionTriggersId": "STANDARD_TRIGGERS",
         "Direction": "*out",
         "TPid": "CGR_API_TESTS",
         "Tenant": "cgrates.org"
     }
    }
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`NOT_FOUND` - Requested AccountActions profile not found.

ApierV1.GetTPAccountActionIds

Queries AccountActions identities on specific tariff plan.

Request:

Data:

```
    type AttrGetTPAccountActionIds struct {
        TPid string // Tariff plan id
    }
```

Mandatory parameters: `[]string{"TPid"}`

***JSON sample*:**

```
    {
     "id": 50,
     "method": "ApierV1.GetTPAccountActionIds",
     "params": [
         {
             "TPid": "CGR_API_TESTS"
         }
     ]
    }
```

Reply:

Data:

```
    []string
```

***JSON sample*:**

```
        {
         "error": null,
         "id": 50,
         "result": [
             "AA_1005"
         ]
        }
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

`NOT_FOUND` - There are no AccountAction profiles defined on the selected TPid.

6.1.5 Management API

These operate on live data.

Gets the destinations for the specified tag.

```
type AttrDestination struct {
        Id       string
        Prefixes []string
}
```

Only the Id field must be set for get operation.

Example GetDestination(attr *AttrDestination, reply *AttrDestination)

Gets a specific balance of a user acoount.

```
type AttrGetBalance struct {
             Tenant    string
             Account   string
             BalanceId string
             Direction string
     }
```

The Tenant is the network tenant of the account.

The Account is the id of the account for which the balance is desired.

The BalanceId can have one of the following string values: MONETARY, SMS, INTERNET, INTERNET_TIME, MINUTES.

Direction can be the strings IN or OUT (default OUT).

Return value is the balance value as float64.

Example GetBalance(attr *AttrGetBalance, reply *float64)

Adds an amount to a specific balance of a user account.

```
type AttrAddBalance struct {
            Tenant    string
            Account   string
            BalanceId string
            Direction string
            Value     float64
    }
```

The Tenant is the network tenant of the account.

The Account is the id of the account for which the balance is set.

The BalanceId can have one of the following string values: MONETARY, SMS, INTERNET, INTERNET_TIME, MINUTES.

Direction can be the strings IN or OUT (default OUT).

Value is the amount to be added to the specified balance.

Example AddBalance(attr *AttrAddBalance, reply *float64)

Executes specified action on a user account.

```
type AttrExecuteAction struct {
            Direction string
            Tenant    string
            Account   string
            ActionsId string
    }
```

Example ExecuteAction(attr *AttrExecuteAction, reply *float64)

```
type AttrAddActionTrigger struct {
        Tenant         string
        Account        string
        Direction      string
        BalanceId      string
        ThresholdValue float64
        DestinationId  string
        Weight         float64
        ActionsId      string
}
```

Example AddTriggeredAction(attr *AttrAddActionTrigger, reply *float64)

```
type AttrSetAccount struct {
        Tenant          string
        Direction       string
        Account         string
        Type            string // <*prepaid|*postpaid>
        ActionTimingsId string
}
```

Example AddAccount(attr *AttrAddAccount, reply *string)

RatingProfiles

ApierV1.SetRatingProfile

Process dependencies and load a specific rating profile from storDb into dataDb.

Request:

Data:

```
        type AttrSetRatingProfile struct {
             TPid          string
             RatingProfileId string
        }
```

Mandatory parameters: `[]string{"TPid", "RatingProfileId"}`

***JSON sample*:**

```
{
 "id": 37,
 "method": "ApierV1.SetRatingProfile",
 "params": [
     {
         "RatingProfileId": "RP_ANY",
         "TPid": "CGR_API_TESTS"
     }
 ]
}
```

Reply:

Data:

```
string
```

Possible answers: `OK` - Success.

***JSON sample*:**

```
{
 "error": null,
 "id": 37,
 "result": "OK"
}
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

Accounts

ApierV1.SetAccountActions

Process dependencies and load a specific AccountActions profile from storDb into dataDb.

Request:

Data:

```
type AttrSetAccountActions struct {
     TPid            string
     AccountActionsId string
}
```

Mandatory parameters: `[]string{"TPid", "AccountActionsId"}`

***JSON sample*:**

```
{
 "id": 0,
 "method": "ApierV1.SetAccountActions",
 "params": [
     {
```

```
                "AccountActionsId": "AA_1005",
                "TPid": "CGR_API_TESTS"
            }
        ]
       }
```

Reply:

Data:

```
    string
```

Possible answers: `OK` - Success.

***JSON sample*:**

```
    {
     "error": null,
     "id": 0,
     "result": "OK"
    }
```

Errors:

`MANDATORY_IE_MISSING` - Mandatory parameter missing from request.

`SERVER_ERROR` - Server error occurred.

6.1.6 Administration APIs

CDR APIs

Set of CDR related APIs.

ApierV1.ExportCsvCdrs

Used to request a new CDR export file. In can include specific interval for CDRs *answer_time*. Any of the two interval limits can
NOTE: Since CGRateS does not keep anywhere a history of exports, it becomes the responsibility of the system administrator to make sure that his exports are not doubled.

Request:

Data:

```
type AttrExpCsvCdrs struct {
      TimeStart    string // If provided, will represent the starting of the CDRs interval (>=)
      TimeEnd      string // If provided, will represent the end of the CDRs interval (<)
 }
```

Mandatory parameters: none

***JSON sample*:**

```
    {
     "id": 3,
     "method": "ApierV1.ExportCsvCdrs",
     "params": [
         {
             "TimeEnd": "1383823746"
```

```
            }
         ]
        }
```

Reply:

Data:

```
        type ExportedCsvCdrs struct {
             ExportedFilePath          string // Full path to the newly generated export file
             NumberOfCdrs              int    // Number of CDRs in the export file
        }
```

***JSON sample*:**

```
        {
         "error": null,
         "id": 3,
         "result": {
             "ExportedFilePath": "/var/log/cgrates/cdr/out/cgr/csv/cdrs_1384104724.csv",
             "NumberOfCdrs": 2
         }
        }
```

Errors:

`SERVER_ERROR` - Server error occurred.

Cache APIs

Set of cache related APIs.

ApierV1.ReloadCache

Used to enforce a cache reload. It can be fine tuned to reload individual destinations and rating plans. In order to reload all destinations and/or rating plans, one can use empty list or null values instead.

Request:

Data:

```
    type ApiReloadCache struct {
          DestinationIds      []string
          RatingPlanIds       []string
      RatingProfileIds    []string
      ActionIds           []string
     }
```

Mandatory parameters: none

***JSON sample*:**

```
        {
         "id": 1,
         "method": "ApierV1.ReloadCache",
         "params": [
             {
                 "DestinationIds": [
                     "GERMANY",
```

```
                    "GERMANY_MOBILE",
                    "FS_USERS"
                ],
                "RatingPlanIds": [
                    "RETAIL1"
                ]
            }
        ]
        }
```

Reply:

Data:

```
    string
```

Possible answers:

- *OK*

***JSON sample*:**

```
    {
     "error": null,
     "id": 1,
     "result": "OK"
    }
```

Errors:

`SERVER_ERROR` - Server error occurred.

Scheduler APIs

Set of scheduler related APIs.

ApierV1.ReloadScheduler

When called CGRateS will reorder/reschedule tasks based on data available in dataDb. This command is necessary after each data load, in some cases being automated in the administration tools (eg: inside *cgr-loader*)

Request:

Data:

```
    string
```

Mandatory parameters: none

***JSON sample*:**

```
    {
     "id": 0,
     "method": "ApierV1.ReloadScheduler",
     "params": [
         ""
     ]
    }
```

Reply:

Data:

```
string
```

Possible answers: **OK**

***JSON sample*:**

```
{
 "error": null,
 "id": 0,
 "result": "OK"
}
```

Errors:

`SERVER_ERROR` - Server error occurred.

6.2 CDR Server

An important component of every rating system is represented by the CDR Server. CGRateS includes an out of the box CDR Server component, controlable in the configuration file and supporting multiple interfaces for CDR feeds. This component makes the CDRs real-time accessible (influenced by the time of receiving them) to CGRateS subsystems.

Following interfaces are supported:

6.2.1 CDR-CGR

Available as handler within http server.

To feed CDRs in via this interface, one must use url of the form: <http://$ip_configured:$port_configured/cgr>.

The CDR fields are received via http form (although for simplicity we support inserting them within query parameters as well) and are expected to be urlencoded in order to transport special characters reliably. All fields are expected by CGRateS as string, particular conversions being done on processing each CDR. The fields received are split into two different categories based on CGRateS interest in them:

Primary fields: the fields which CGRateS needs for it's own operations and are stored into cdrs_primary table of storDb.

- tor: type of record, meta-field, should map to one of the TORs hardcoded inside the server <*voice|*data|*sms>
- accid: represents the unique accounting id given by the telecom switch generating the CDR
- cdrhost: represents the IP address of the host generating the CDR (automatically populated by the server)
- cdrsource: formally identifies the source of the CDR (free form field)
- reqtype: matching the supported request types by the **CGRateS**, accepted values are hardcoded in the server <prepaid|postpaid|pseudoprepaid|rated>.
- direction: matching the supported direction identifiers of the CGRateS <*out>
- tenant: tenant whom this record belongs
- category: free-form filter for this record, matching the category defined in rating profiles.
- account: account id (accounting subsystem) the record should be attached to
- subject: rating subject (rating subsystem) this record should be attached to

- destination: destination to be charged
- setup_time: set-up time of the event. Supported formats: datetime RFC3339 compatible, SQL datetime (eg: MySQL), unix timestamp.
- answer_time: answer time of the event. Supported formats: datetime RFC3339 compatible, SQL datetime (eg: MySQL), unix timestamp.
- usage: event usage information (eg: in case of tor=*voice this will represent the total duration of a call)

Extra fields: any field coming in via the http request and not a member of primary fields list. These fields are stored as json encoded into *cdrs_extra* table of storDb.

Example of sample CDR generated simply using curl:

```
curl --data "curl --data "tor=*voice&accid=iiaasbfdsaf&cdrhost=192.168.1.1&cdrsource=curl_cdr&reqtype
```

6.2.2 CDR-FS_JSON

Available as handler within http server, it implements the mechanism to store CDRs received from FreeSWITCH mod_json_cdr.

This interface is available at url: <http://$ip_configured:$port_configured/freeswitch_json>.

This handler has a different implementation logic than the previous CDR-CGR, filtering fields received in the CDR from FreeSWITCH based on predefined configuration. The mechanism of extracting CDR information out of JSON encoded CDR received from FreeSWITCH is the following:

- When receiving the CDR from FreeSWITCH, CGRateS will extract the content of ''variables" object.
- **Content of the ''variables" will be filtered out and the following information will be stored into an internal CDR object:**
  - **Fields used by CGRateS in primary mediation, known as primary fields. These are:**
    - uuid: internally generated uuid by FreeSWITCH for the call
    - sip_local_network_addr: IP address of the FreeSWITCH box generating the CDR
    - sip_call_id: call id out of SIP protocol
    - cgr_reqtype: request type as understood by the CGRateS
    - cgr_category: call category (optional)
    - cgr_tenant: tenant this call belongs to (optional)
    - cgr_account: account id in CGRateS (optional)
    - cgr_subject: rating subject in CGRateS (optional)
    - cgr_destination: destination being rated (optional)
    - user_name: username as seen by FreeSWITCH (considered if cgr_subject or cgr_account not present)
    - dialed_extension: destination number considered if cgr_destination is missing
  - Fields stored at request in cdr_extra and definable in configuration file under *extra_fields*.
- Once the content will be filtered, the real CDR object will be processed, stored into storDb under *cdrs_primary* and *cdrs_extra* tables and, if configured, it will be passed further for mediation.

6.2.3 CDR-RPC

Available as RPC handler on top of CGR APIs exposed (in-process as well as GOB-RPC and JSON-RPC). This interface is used for example by CGR-SM component capturing the CDRs over event interface (eg: OpenSIPS or FreeSWITCH-ZeroConfig scenario)

The RPC function signature looks like this:

```
CDRSV1.ProcessCdr(cdr *utils.StoredCdr, reply *string) error
```

The simplified StoredCdr object is represented by following:

```
type StoredCdr struct {
  CgrId          string
  OrderId        int64             // Stor order id used as export order id
  TOR            string            // type of record, meta-field, should map to one of the TORs hardc
  AccId          string            // represents the unique accounting id given by the telecom switch
  CdrHost        string            // represents the IP address of the host generating the CDR (autor
  CdrSource      string            // formally identifies the source of the CDR (free form field)
  ReqType        string            // matching the supported request types by the **CGRateS**, accept
  Direction      string            // matching the supported direction identifiers of the CGRateS <*c
  Tenant         string            // tenant whom this record belongs
  Category       string            // free-form filter for this record, matching the category defined
  Account        string            // account id (accounting subsystem) the record should be attached
  Subject        string            // rating subject (rating subsystem) this record should be attache
  Destination    string            // destination to be charged
  SetupTime      time.Time         // set-up time of the event. Supported formats: datetime RFC3339 c
  AnswerTime     time.Time         // answer time of the event. Supported formats: datetime RFC3339 c
  Usage          time.Duration     // event usage information (eg: in case of tor=*voice this will re
  ExtraFields    map[string]string // Extra fields to be stored in CDR
```

}

6.3 CDR Client (cdrc)

It's role is to gather offline CDRs and post them to CDR Server(CDRS) component.

Part of the *cgr-engine*, can be started on a remote server as standalone component.

Controlled within *cdrc* section of the configuration file.

Has two modes of operation:

- Automated: CDR file processing is triggered on file creation/move.
- Periodic: CDR file processing will be triggered at configured time interval (delay/sleep between processes) and it will be performed on all files present in the folder (IN) at run time.

Principles behind functionality:

- Monitor/process a CDR folder (IN) as outlined above.
- For every file processed, extract the information based on configuration and post it via configured mechanism to CDRS.
- The fields extracted out of each CDR row are the same ones depicted in the CDRS documentation (following primary and extra fields concept).

- Once the file processing completes, move it in it's original format in another folder (OUT) in order to avoid re-processing. Here it's worth mentioning the auto-detection of duplicated CDRs at server side based on accid and host fields.
- Advanced configuration like forking a number of simultaneous client instances monitoring different folders possible through the use of *.xml* configuration.

6.3.1 Import Templates

To specifiy custom imports (for various sources) one can specify *Import Templates*. These are definable within both *.cfg* as well as *.xml* advanced configuration files. For increased flexibility the Import Template can be defined using CGR-RSR fields capturing both ReGexp as well as static rules. The static values will be way faster in processing but limited in functionality.

CGR-RSR Regexp Rule

Format:

```
~field_id:s/regexp_search_and_capture_rule/output_teplate/
```

Example of usage:

```
Input CDR field:
       {
       "account": "First-Account123"
       }
Capture Rule:
       ~account:s/^*+(Account123)$/$1-processed/
Result after processing:
       {
       "account": "Account123-processed"
       }
```

CGR-RSR Static Rule

Format:

```
^field_id:static_value
```

Example of usage:

Input CDR field:

{ "account": "First-Account123" }

Capture Rule: ^account:MasterAccount

Result after processing: { "account": "MasterAccount" }

CDR Formats supported:

6.3.2 CDR .CSV

Most widely used format by Telecom Switches.

Light to read and generic to process. CDRC should be able to process in this way any .csv CDR, independent of the Telecom Switch generating them. Incompatibilities here can come out of answer time and duration formats which

can vary between CDR writer implementations. As answer time we support a number of formats already - rfc3339, SQL/MySQL, unix timestamp. As duration we support nanoseconds granularity in our code. Time unit can be specified (eg: ms, s, m, h), or if missing, will default to nanoseconds.

In case of *.csv* files the Import Template will contain indexes for the possition where primary fields are located (0 representing the first field) and fieldname/position format for extra fields which need not only to be extracted by row index but also to be named since .csv format does not save field names/labels. CDRC uses the following convention for extra fields in the configuration: *<label_extrafield_1>:<index_extrafield_1>[...,<label_extrafield_n>:<index_extrafield_n>]...*.

6.4 CDR Exporter

Component to retrieve rated CDRs from internal CDRs database.

Although nowadays it is custom to read a storage/database with tools, we do not recommend doing it so due to possibility that reads can slow down complete rating system. For this purpose we have created exporter plugins which are meant to work in tight relationship with CGRateS internal components in order to best optimize performance and avoid system locks.

6.4.1 Export Templates

For advanced needs CGRateS Export Templates are configurable via *.cfg*, *.xml* as well as directly within RPC call requesting the export to be performed. Inside each Export Template one can either specify simple CDR field ids or use CGR-RSR fields capturing both Regexp as well as static rules.

CGR-RSR Regexp Rule

Format:

```
~field_id:s/regexp_search_and_capture_rule/output_teplate/
```

Example of usage:

```
Input CDR field:
  {
  "account": "First-Account123"
  }
Capture Rule:
  ~account:s/^*+(Account123)$/$1-processed/
Result after processing:
  {
  "account": "Account123-processed"
  }
```

CGR-RSR Static Rule

Format:

```
^field_id:static_value
```

Example of usage:

Input CDR field:

{ “account”: “First-Account123” }

Capture Rule: ^account:MasterAccount

Result after processing: { “account”: “MasterAccount” }

Export interfaces implemented:

6.4.2 CGR-CSV

Simplest way to export CDRs in a format internally defined (with parts like *CDRExtraFields* configurable in main configuration file).

Principles behind exports:

- Exports are to be manually requested (although automated is planned for the future through the used of built-in scheduled actions) via exposed JSON-RPC api. Example of api call from python call provided as sample script:

```
rpc.call("ApierV1.ExportCsvCdrs",{"TimeStart":"1383823746","TimeEnd":"1383833746"} )
```

- On each export call there will be a .csv format file generated using configured separator. Location of the export folder is definable inside *cgrates.cfg*.
- File name of the export will appear like: *cdrs_$(timestamp).csv* where $(timestamp) will be replaced by unix timestamp of the server running the export process or requested via API call.
- Each exported file will have as content all the CDRs inside time interval defined in the API call. Both TimeStart and TimeEnd are optional, hence being able to obtain a full export of the available CDRs with one API call.
- To be noted here that CGRateS does not keep anywhere a history of exports, hence it is the responsibility of the system administrator to make sure that his exports are not doubled.
- If not otherwise defined, each line within the exported file will follow an internally predefined template:

cgrid,mediation_runid,tor,accid,reqtype,direction,tenant,category,account,subject,destination,setup_time,answer_time,usage,co

```
$(cgrid),$(mediation_runid),$(tor),$(accid),$(reqtype),$(direction),$(direction),$(tenant),$(cat
```

The significance of the fields exported:

- tor: type of record, meta-field, should map to one of the TORs hardcoded inside the server <*voice|*data|*sms>
- accid: represents the unique accounting id given by the telecom switch generating the CDR
- cdrhost: represents the IP address of the host generating the CDR (automatically populated by the server)
- cdrsource: formally identifies the source of the CDR (free form field)
- reqtype: matching the supported request types by the **CGRateS**, accepted values are hardcoded in the server <prepaid|postpaid|pseudoprepaid|rated>.
- direction: matching the supported direction identifiers of the CGRateS <*out>
- tenant: tenant whom this record belongs
- category: free-form filter for this record, matching the category defined in rating profiles.
- account: account id (accounting subsystem) the record should be attached to
- subject: rating subject (rating subsystem) this record should be attached to
- destination: destination to be charged

- setup_time: set-up time of the event. Supported formats: datetime RFC3339 compatible, SQL datetime (eg: MySQL), unix timestamp.
- answer_time: answer time of the event. Supported formats: datetime RFC3339 compatible, SQL datetime (eg: MySQL), unix timestamp.
- usage: event usage information (eg: in case of tor=*voice this will represent the total duration of a call)
- **extra_cdr_fields:**
  - selected list of cdr_extra fields via *cgrates.cfg* configuration or
  - alphabetical order of the cdr extra fields stored in cdr_extra table

Sample CDR export file content which was made available at path: */var/log/cgrates/cdr/out/cgr/csv/cdrs_1384104724.csv*

```
dbafe9c8614c785a65aabd116dd3959c3c56f7f6,default,*voice,dsafdsaf,rated,*out,cgrates.org,call,1001,10(
```

6.4.3 CGR-FWV

Fixed width form of export CDR. Advanced template configuration available via *.xml* configuration file.

6.4.4 Hybrid CSV-FWV

For advanced needs **CGRateS** supports exporting the CDRs as combination between *.csv* and *.fwv* formats.

6.5 CDR Stats Server

Collects CDRs from various sources (eg: CGR-CDRS, CGR-Mediator, CGR-SM, third-party CDR source via RPC) and builds real-time stats based on them. Each StatsQueue has attached *ActionTriggers* with monitoring and actions capabilities.

Principles of functionality:

- Standalone component (can be started individually on remote hardware, isolated form other CGRateS compoenents).
- Performance oriented. Should be able to process tens of thousands of CDRs per second.
- Cache driven technology. But SaveInterval can be set to store this information on redis.
- Stats are build within *StatsQueues* a CDR Stats Server being able to support unlimited number of StatsQueues. Each CDR will be passed to all of StatsQueues available and will be processed by individual StatsQueue based on configuration.
- Stats will be build inside Metrics (eg: ASR, ACD, ACC, TCC) and attached to specific StatsQueue.
- Each StatsQueue will have attached one *ActionTriggers* profile which will monitor Metrics values and react on thresholds reached (unlimited number of thresholds and reactions configurable).
- CDRs are processed by StatsQueues if they pass CDR field filters.
- CDRs are auto-removed from StatsQueues in a *fifo* manner if the QueueLength is reached or if they do not longer fit within TimeWindow defined.

6.5.1 Configuration

Individual StatsQueue configurations are loaded inside TariffPlan defitions, one configuration object is internally represented as:

```
type CdrStats struct {
    Id                string          // Config id, unique per config instance
    QueueLength       int             // Number of items in the stats buffer
    TimeWindow        time.Duration   // Will only keep the CDRs who's call setup time is not older t
    SaveInterval      time.Duration   // Interval to store the info into database
    Metrics           []string        // ASR, ACD, ACC, TCC, TCD, PDD
    SetupInterval     []time.Time     // CDRFieldFilter on SetupInterval, 2 or less items (>= start i
    TOR               []string        // CDRFieldFilter on TORs
    CdrHost           []string        // CDRFieldFilter on CdrHosts
    CdrSource         []string        // CDRFieldFilter on CdrSources
    ReqType           []string        // CDRFieldFilter on ReqTypes
    Direction         []string        // CDRFieldFilter on Directions
    Tenant            []string        // CDRFieldFilter on Tenants
    Category          []string        // CDRFieldFilter on Categories
    Account           []string        // CDRFieldFilter on Accounts
    Subject           []string        // CDRFieldFilter on Subjects
    DestinationPrefix []string        // CDRFieldFilter on DestinationPrefixes
    UsageInterval     []time.Duration // CDRFieldFilter on UsageInterval, 2 or less items (>= Usage,
    PddInterval       []time.Duration // CDRFieldFilter on PddInterval, 2 or less items (>= Pdd, <Pdd
    Supplier          []string        // CDRFieldFilter on Suppliers
    DisconnectCause   []string        // Filter on DisconnectCause
    MediationRunIds   []string        // CDRFieldFilter on MediationRunIds
    RatedAccount      []string        // CDRFieldFilter on RatedAccounts
    RatedSubject      []string        // CDRFieldFilter on RatedSubjects
    CostInterval      []float64       // CDRFieldFilter on CostInterval, 2 or less items, (>=Cost, <C
    Triggers          ActionTriggerPriotityList
}
```

6.5.2 Metrics Types

- ACC *(Average Call Cost)*: Queue with the average call cost
- ACD *(Average Call Duration)*: Queue with the average call duration
- ASR *(Answer-Seizure Ratio)*: Queue with the answer ratio
- PDD *(Post Dial Delay)*: Queue with the average Post Dial Delay in seconds
- TCC *(Total Call Cost)*: Queue with the Total cost for the time frame.
- TCD *(Total Call Duration)*: Queue with the total call duration for the

time frame

6.5.3 ExternalQueries

The Metrics calculated are available to be real-time queried via RPC methods.

To facilitate interaction there are four commands built in the provided *cgr-console* tool:

- *cdrstats_queueids*: returns the queue ids processing CDR Stats.
- *cdrstats_metrics*: returns metrics calculated within specific CDRStatsQueue.

- *cdrstats_reload*: reloads the CdrStats configurations out of DataDb.
- *cdrstats_reset*: resets calculated metrics for one specific or all StatsQueues.

6.5.4 Example use

When you work with balance maybe you want to keep a eye in your users, so you can add a new queue for the last 5 hours to check that your customer it's not hacked, this is an example:

CDR stats:

```
"result":{
    "CdrStats": [
      {
        "Accounts": "my_account",
        "ActionTriggers": "FRAUD_CHECK",
        "Categories": "",
        "CdrHosts": "",
        "CdrSources": "",
        "CostInterval": "",
        "DestinationPrefixes": "",
        "Directions": "",
        "DisconnectCauses": "",
        "MediationRunIds": "",
        "Metrics": "TCC",
        "PddInterval": "",
        "QueueLength": "0",
        "RatedAccounts": "",
        "RatedSubjects": "",
        "ReqTypes": "",
        "SaveInterval": "15s",
        "SetupInterval": "",
        "Subjects": "",
        "Suppliers": "",
        "TORs": "",
        "Tenants": "foehn",
        "TimeWindow": "5h",
        "UsageInterval": ""
      }
    ],
    "CdrStatsId": "FRAUD_ACCOUNT",
    "TPid": "test"
  }
```

Action Trigger:

```
"result": {
    "ActionTriggers": [
      {
        "ActionsId": "LOG_WARNING",
        "BalanceCategory": "",
        "BalanceDestinationIds": "",
        "BalanceDirection": "",
        "BalanceExpirationDate": "",
        "BalanceId": "",
        "BalanceRatingSubject": "",
        "BalanceSharedGroup": "",
        "BalanceTimingTags": "",
        "BalanceType": "",
```

```
        "BalanceWeight": 0,
        "Id": "",
        "MinQueuedItems": 0,
        "MinSleep": "3h",
        "Recurrent": true,
        "ThresholdType": "\\*max_tcc",
        "ThresholdValue": 150,
        "Weight": 10
      }
    ],
    "ActionTriggersId": "FRAUD_CHECK",
    "TPid": "test"
  }
```

Using *cgr-console* you can check the status of the queue anytime:

```
cgr-console 'cdrstats_metrics StatsQueueId="FRAUD_ACCOUNT"'
```

6.6 LCR System

In voice telecommunications, least-cost routing (LCR) is the process of selecting the path of outbound communications traffic based on cost. Within a telecoms carrier, an LCR team might periodically (monthly, weekly or even daily) choose between routes from several or even hundreds of carriers for destinations across the world. This function might also be automated by a device or software program known as a "Least Cost Router." *[WIKI2015]*

6.6.1 Data structures

The LCR rule parameters are: Direction, Tenant, Category, Account, Subject, DestinationId, RPCategory, Strategy, StrategyParameters, ActivationTime, Weight.

The first five are used to match the rule for a specific call descriptor. They can have a value or marked as *any.

The DestinationId can be used to filter the LCR rules entries or to make the rule more specific.

RPCategory is used to indicate the rating profile category.

Strategy indicates supplier selection algorithm and StrategyParams will be specific to each strategy. Strategy can be one of the following:

***static (filter)** Will use the suppliers provided as params. StrategyParams: suppier1;supplier2;etc

***lowest_cost (sorting)** Matching suppliers will be sorted by ascending cost. StrategyParams: None

***highest_cost (sorting)** Matching suppliers will be sorted by descending cost. StrategyParams: None

***qos_with_threshold (filter)** The system will reject the suppliers that have out of bounds average success ratio or average call duration. StrategyParams: min_asr;max_asr;min_acd;max_acd;min_tcd;max_tcd;min_acc;max_acc;min_tcc;max_tcc

***qos (sorting)** The system will sort by metrics in the order of appearance. StrategyParams: metric1;metric2;etc

***load_distribution (sorting/filter)** The system will sort the suppliers in order to achieve the specified load distribution. - if all have less than ratio return random order - if some have a cdr count not divisible by ratio return them first and all ordered by cdr times, oldest first - if all have a multiple of ratio return in the order of cdr times, oldest first StrategyParams: supplier1:ratio;supplier2:ratio;*default:ratio

ActivationTime is the date/time when the LCR entry starts to be active.

Weight is used to sort the rules with the same activation time.

Example

```
*in, cgrates.org,call,*any,*any,EU_LANDLINE,LCR_STANDARD,*static,ivo;dan;rif,2012-01-01T00:00:00Z,10
```

6.6.2 Code implementation

The general process of getting LCRs is this.

The LCR rules for a specific call descriptor are searched using direction, tenant, category, account and subject of the call descriptor matched as strictly as possible with LCR rules.

Because a rule can have several entries they will be sorted by activation time.

Next the system will find out the most recent LCR entry that applies to this call considering entries activation times.

The LCR entry is processed according to it's strategy. For static strategy the cost is calculated for each supplier found in the parameters and the suppliers are listed as they are found.

For the QOS strategies the suppliers are searched using call descriptor parameters (direction, tenant, category, account, subject), than the cdrstats module is queried for the QOS values and the suppliers are filtered or sorted according to the StrategyParameters field. The suppliers that have the QOS parameters in the stats queues but did not get the chance to process any calls are favored in the QOS sorting algorithm. If a certain QOS metric is missing from the supplier queues than the metric is ignored and the sorting or filtering is done using the next metrics that are considered.

For the lowest/highest cost strategies the matched suppliers are sorted ascending/descending on cost.

```
{
 "Entry": {
      "DestinationId": "*any",
      "RPCategory": "LCR_STANDARD",
      "Strategy": "*lowest_cost",
      "StrategyParams": "",
      "Weight": 20
  },
  "SupplierCosts": [{"Supplier":"rif", Cost:"2.0"},{"Supplier":"dan", Cost:"1.0"}]
 }
```

6.7 DerivedCharging

DerivedCharging is the process of forking original request into a number (configured) of emulated ones, derived from the original parameters. This mechanism used in combination with multi-tenancy supported by default by **CGRateS** can give out complex charging scenarios, needed for example in case of whitelabel-ing.

DerivedCharging occurs in two separate places:

- SessionManager: necessary to handle each derived (emulated) session in it's individuall loop (eg: individual resellers will have their own charging policies implemented, some paying per minute, others per second and so on) and keep them in sync (eg: one reseller is left out of money, original call should be disconnected and all emulated sessions should end their debit loops).
- Mediator: necessary to fork the CDRs into a number of derived ones influenced by the derived charging configuration and rate them individually.

6.7.1 Configuration

DerivedCharging is configured in two places:

- Platform level configured within *cgrates.cfg* file.
- Account level configured as part of TarrifPlans defition or interactively via RPC methods.

One DerivedCharger object will be configured by an internal object like:

```
type DerivedCharger struct {
       RunId               string      // Unique runId in the chain
       RunFilters          string      // Only run the charger if all the filters match
       ReqTypeField        string      // Field containing request type info, number in case of csv s
       DirectionField      string      // Field containing direction info
       TenantField         string      // Field containing tenant info
       CategoryField       string      // Field containing tor info
       AccountField        string      // Field containing account information
       SubjectField        string      // Field containing subject information
       DestinationField    string      // Field containing destination information
       SetupTimeField      string      // Field containing setup time information
       AnswerTimeField     string      // Field containing answer time information
       UsageField          string      // Field containing usage information
}
```

CGRateS is able to attach an unlimited number of DerivedChargers to a single request, based on configuration.

6.8 Rating history

Enhances CGRateS with ability to archive rates modifications.

Large scaling posibility using server-agents approach. In a distributed environment, there will be a single server (which can be backed up using technologies such as Linux-HA) and more agents sending the modifications to be archived.

6.8.1 History-Server

Part of the *cgr-engine*.

Controlled within *history_server* section of the configuration file.

Stores rating archive in a .git folder, hence making the changes available for analysis via any git browser tool (eg: gitg in linux).

Functionality:

- On startup reads the rating archive out of .git folder and caches the data.
- When receiving rating information from the agents it will recompile the cache.
- Based on configured save interval it will dump the rating cache (if changed) into the .git archive.
- Archives the following rating data:
- Destinations inside *destinations.json* file.
- Rating plans inside *rating_plans.json* file.
- Rating profiles inside *rating_profiles.json* file.

6.8.2 History-Agent

Integrated in the rating loader components.

Part of *cgr-engine* and *cgr-loader*.

Enabled via *history_agent* configuration section within *cgr-engine* and *history_server* command line parameter in case of *cgr-loader*.

Sends the complete rating data loaded into ratingDb to *history_server* for archiving.

6.9 Rating logic

Let's start with the most important function: finding the cost of a certain call.

The call information comes to CGRateS having the following vital information like subject, destination, start time and end time. The engine will look up the database for the rates applicable to the received subject and destination.

```
type CallDescriptor struct {
       Direction
       TOR
       Tenant, Subject, Account, Destination
       TimeStart, TimeEnd
       LoopIndex       // indicates the position of this segment in a cost request loop
       CallDuration    // the call duration so far (partial or final)
       FallbackSubject // the subject to check for destination if not found on primary subject
       RatingPlans
}
```

When the session manager receives a call start event it will first check if the call is prepaid or postpaid. If the call is postpaid than the cost will be determined only once at the end of the call but if the call is prepaid there will be a debit operation every X seconds (X is configurable).

In prepaid case the rating engine will have to set rates for multiple parts of the call so the *LoopIndex* in the above structure will help the engine add the connect fee only to the first part. The *CallDuration* attribute is used to set the right rate in case the rates database has different costs for the different parts of a call e.g. first minute is more expensive (we can also define the minimum rate unit).

The **FallbackSubject** is used in case the initial call subject is not found in the rating profiles list (more on this later in this chapter).

What are the activation periods?

> At one given time there is a set of prices that applay to different time intervals when a call can be made. In CGRateS one can define multiple such sets that will become active in various point of time called activation time. The activation period is a structure describing different prices for a call on different intervals of time. This structure has an activation time, which specifies the active prices for a period of time by one ore more (usually more than one) rate intervals.

```
type RateInterval struct {
       Years
       Months
       MonthDays
       WeekDays
       StartTime, EndTime
       Weight, ConnectFee
       Prices
       RoundingMethod
       RoundingDecimals
}

type Price struct {
```

```
        GroupIntervalStart
        Value
        RateIncrement
        RateUnit
}
```

An **RateInterval** specifies the Month, the MonthDay, the WeekDays, the StartTime and the EndTime when the RateInterval's price profile is in effect.

> **Example** The RateInterval {"Month": [1], "WeekDays":[1,2,3,4,5], "StartTime":"18:00:00"} specifies the *Price* for the first month of each year from Monday to Friday starting 18:00. Most structure elements are optional and they can be combined in any way it makes sense. If an element is omitted it means it is zero or any.

The *ConnectFee* specifies the connection price for the call if this interval is the first one of the call.

The *Weight* will establish which interval will set the price for a call segment if more then one applies to it.

> **Example** Let's assume there is an interval defining price for the weekdays and another interval that defines a special holiday rates. As that holiday is also one of the regular weekdays than both intervals are applicable to a call made on that day so the interval with the smaller Weight will give the price for the call in question. If both intervals have the same Weight than the interval with the smaller price wins. It is, however, a good practice to set the Weight for the defined intervals.

The *RoundingMethod* and the *RoundingDecimals* will adjust the price using the specified function and number of decimals (more on this in the rates definition chapter).

The **Price** structure defines the start (*GroupIntervalStart*) of a section of a call with a specified rate *Value* per *RateUnit* diving and rounding the section in *RateIncrement* subsections.

So when there is a need to define new sets of prices just define new RatingPlans with the activation time set to the moment when it becomes active.

Let's get back to the engine. When a GetCost or Debit call comes to the engine it will try to match the best rating profile for the given *Direction*, *Tenant*, *TOR* and *Subject* using the longest *Subject* prefix method or using the *FallbackSubject* if not found. The rating profile contains the activation periods that might apply to the call in question.

At this point in rating process the engine will start splitting the call into various time spans using the following criterias:

1. Minute Balances: first it will handle the call information to the originator user acount to be split by available minute balances. If the user has free or special price minutes for the call destination they will be consumed by the call.
2. Activation periods: if there were not enough special price minutes available than the engine will check if the call spans over multiple activation periods (the call starts in initial rates period and continues in another).
3. RateIntervals: for each activation period that apply to the call the engine will select the best rate intervals that apply.

```
type TimeSpan struct {
        TimeStart, TimeEnd
        Cost
        RatingPlan
        RateInterval
        MinuteInfo
        CallDuration  // the call duration so far till TimeEnd
}
```

The result of this splitting will be a list of *TimeSpan* structures each having attached the MinuteInfo or the RateInterval that gave the price for it. The *CallDuration* attribute will select the right *Price* from the *RateInterval Prices* list. The

final cost for the call will be the sum of the prices of these times spans plus the *ConnectionFee* from the first time span of the call.

6.9.1 User balances

The user account contains a map of various balances like money, sms, internet traffic, internet time, etc. Each of these lists contains one or more Balance structure that have a wheight and a possible expiration date.

```
type UserBalance struct {
       Type           // prepaid-postpaid
       BalanceMap
       UnitCounters
       ActionTriggers
}

type Balance struct {
       Value
       ExpirationDate
       Weight
}
```

CGRateS treats special priced or free minutes different from the rest of balances. They will be called free minutes further on but they can have a special price.

The free minutes must be handled a little differently because usually they are grouped by specific destinations (e.g. national minutes, ore minutes in the same network). So they are grouped in balances and when a call is made the engine checks all applicable balances to consume minutes according to that call.

When a call cost needs to be debited these minute balances will be queried for call destination first. If the user has special minutes for the specific destination those minutes will be consumed according to call duration.

A standard debit operation consist of selecting a certaing balance type and taking all balances from that list in the weight order to be debited till the total amount is consumed.

CGRateS provide api for adding/substracting user's money credit. The prepaid and postpaid are uniformly treated except that the prepaid is checked to be always greater than zero and the postpaid can go bellow zero.

Both prepaid and postpaid can have a limited number of free SMS and Internet traffic per month and this budget is replenished at regular intervals based on the user tariff plan or as the user buys more free SMSs (for example).

Another special feature allows user to get a better price as the call volume increases each month. This can be added on one ore more thresholds so the more he/she talks the cheaper the calls.

Finally bonuses can be rewarded to users who received a certain volume of calls.

7. Tutorials

7.1 FreeSWITCH Integration Tutorials

In these tutorials we exemplify few cases of integration between FreeSWITCH and **CGRateS**. We start with common steps, installation and postinstall processes then we dive into particular configurations.

7.1.1 Software installation

As operating system we have chosen Debian Jessie, since all the software components we use provide packaging for it.

FreeSWITCH

More information regarding installing FreeSWITCH on Debian can be found on it's official installation wiki.

To get FreeSWITCH installed and configured, we have choosen the simplest method, out of *vanilla* packages plus one individual module we need: *mod-json-cdr*.

We got FreeSWITCH installed via following commands:

```
wget -O - http://files.freeswitch.org/repo/deb/freeswitch-1.6/key.gpg |apt-key add -
echo "deb http://files.freeswitch.org/repo/deb/freeswitch-1.6/ jessie main" > /etc/apt/sources.list.d
apt-get update
apt-get install freeswitch-meta-vanilla freeswitch-mod-json-cdr libyuv-dev
```

Once installed we proceed with loading the configuration out of specific tutorial cases bellow.

7.1.2 CGRateS Installation

As operating system we have choosen Debian Wheezy, since all the software components we use provide packaging for it.

Prerequisites

Some components of **CGRateS** (whether enabled or not is up to the administrator) depend on external software like:

- Git used by **CGRateS** History Server as archiver.
- Redis to serve as Rating and Accounting DB for **CGRateS**.

- MySQL to serve as StorDB for **CGRateS**.

We will install them in one shot using the command bellow.

```
apt-get install git redis-server mysql-server
```

Note: For simplicity sake we have used as MySQL root password when asked: *CGRateS.org*.

Installation

Installation steps are provided within **CGRateS** install documentation.

To get **CGRateS** installed execute the following commands over ssh console:

```
cd /etc/apt/sources.list.d/
wget -O - http://apt.itsyscom.com/conf/cgrates.gpg.key | apt-key add -
wget http://apt.itsyscom.com/conf/cgrates.apt.list
apt-get update
apt-get install cgrates
```

As described in post-install section, we will need to set up the MySQL database (using *CGRateS.org* as our root password):

```
cd /usr/share/cgrates/storage/mysql/
./setup_cgr_db.sh root CGRateS.org localhost
```

At this point we have **CGRateS** installed but not yet configured. To facilitate the understanding and speed up the process, **CGRateS** comes already with the configurations used in these tutorials, available in the */usr/share/cgrates/tutorials* folder, so we will load them custom on each tutorial case.

7.1.3 SIP UA - Jitsi

On our ubuntu desktop host, we have installed Jitsi to be used as SIP UA, out of stable provided packages on Jitsi download and had Jitsi configured with 4 accounts: 1001/1234, 1002/1234, 1003/1234 and 1004/1234.

7.1.4 FreeSWITCH generating *http-json* CDRs

Scenario

- FreeSWITCH with *vanilla* configuration adding *mod_json_cdr* for CDR generation.
- Modified following users (with configs in *etc/freeswitch/directory/default*): 1001-prepaid, 1002-postpaid, 1003-pseudoprepaid, 1004-rated, 1006-prepaid, 1007-rated.
- Have added inside default dialplan CGR own extensions just before routing towards users (*etc/freeswitch/dialplan/default.xml*).
- FreeSWITCH configured to generate default *http-json* CDRs.
- **CGRateS** with following components:
- CGR-SM started as prepaid controller, with debits taking place at 5s intervals.
- CGR-CDRS component receiving raw CDRs from FreeSWITCH, storing them and attaching costs inside CGR StorDB.
- CGR-CDRE exporting processed CDRs from CGR StorDB (export path: */tmp*).

- CGR-History component keeping the archive of the rates modifications (path browsable with git client at */tmp/cgr_history*).

Starting FreeSWITCH with custom configuration

```
/usr/share/cgrates/tutorials/fs_evsock/freeswitch/etc/init.d/freeswitch start
```

To verify that FreeSWITCH is running we run the console command:

```
fs_cli -x status
```

Starting CGRateS with custom configuration

```
/usr/share/cgrates/tutorials/fs_evsock/cgrates/etc/init.d/cgrates start
```

Check that cgrates is running

```
cgr-console status
```

CDR processing

At the end of each call FreeSWITCH will issue a http post with the CDR. This will reach inside **CGRateS** through the *CDRS* component (close to real-time). Once in-there it will be instantly rated and it is ready to be exported:

```
cgr-console 'cdrs_export CdrFormat="csv" ExportDir="/tmp"'
```

CGRateS Usage

Since it is common to most of the tutorials, the example for **CGRateS** usage is provided in a separate page here

7.1.5 CGRateS Usage

Loading CGRateS Tariff Plans

Before proceeding to this step, you should have **CGRateS** installed and started with custom configuration, depending on the tutorial you have followed.

For our tutorial we load again prepared data out of shared folder, containing following rules:

- Create the necessary timings (always, asap, peak, offpeak).
- Configure 3 destinations (1002, 1003 and 10 used as catch all rule).
- As rating we configure the following:
- Rate id: *RT_10CNT* with connect fee of 20cents, 10cents per minute for the first 60s in 60s increments followed by 5cents per minute in 1s increments.
- Rate id: *RT_20CNT* with connect fee of 40cents, 20cents per minute for the first 60s in 60s increments, followed by 10 cents per minute charged in 1s increments.
- Rate id: *RT_40CNT* with connect fee of 80cents, 40cents per minute for the first 60s in 60s increments, follwed by 20cents per minute charged in 10s increments.
- Rate id: *RT_1CNT* having no connect fee and a rate of 1 cent per minute, chargeable in 1 minute increments.

- Will charge by default *RT_40CNT* for all 10xx destinations during peak times (Monday-Friday 08:00-19:00) and *RT_10CNT* during offpeatimes (rest).
- Account 1001 will receive a special *deal* for 1002 and 1003 destinations during peak times with *RT_20CNT*, otherwise having default rating.
- Accounting part will have following configured:
- Create 5 accounts: 1001, 1002, 1003, 1004, 1007.
- Create 1 account alias (1006 - alias of account 1002).
- Create 1 rating profile alias (1006 - alias of rating profile 1001).
- 1002, 1003, 1004 will receive 10units of **monetary* balance.
- 1001 will receive 5 units of general **monetary*, 5 units of shared balance in the shared group "SHARED_A" and 90 seconds of calling destination 1002 with special rates *RT_1CNT*.
- 1007 will receive 0 units of shared balance in the shared group "SHARED_A".
- Define the shared balance "SHARED_A" with debit policy **highest*.
- For each balance created, attach 4 triggers to control the balance: log on balance<2, log on balance>20, log on 5 mins talked towards 10xx destination, disable the account and log if a balance is higher than 100 units.
- *DerivedCharging* will execute one extra mediation run when the sessions will have as account and rating subject 1001 resulting in a cloned session with most of parameters identical to original except RequestType which will be set on *rated* instead of original *prepaid* one. The extra run will be identified by *derived_run1* in CDRs.
- Will configure 4 extra CdrStatQueues:
  - *CDRST1* with 10 CDRs in the Queue and unlimited time window, calculating *ASR*, *ACD* and *ACC* for CDRs with Tenant matching *cgrates.org* and MediationRunId matching *default*. On this StatsQueue we will attach an ActionTrigger profile identified by *CDRST1_WARN*
  - *CDRST_1001* with 10 CDRs in the Queue and 10 minutes time window calculating *ASR*, *ACD* and *ACC* for CDRs with Tenant matching *cgrates.org*, RatingSubject matching *1001* and MediationRunId matching *default*. On this StatsQueue we will attach an ActionTrigger profile identified by *CDRST1001_WARN*
  - *CDRST_1002* with 10 CDRs in the Queue and 10 minutes time window calculating *ASR*, *ACD* and *ACC* for CDRs with Tenant matching *cgrates.org*, RatingSubject matching *1002* and MediationRunId matching *default*. On this StatsQueue we will attach an ActionTrigger profile identified by *CDRST1001_WARN*
  - *CDRST_1003* with 10 CDRs in the Queue and 10 minutes time window calculating *ASR*, and *ACD* for CDRs with Tenant matching *cgrates.org*, Destination matching *1003* and MediationRunId matching *default*. On this StatsQueue we will attach an ActionTrigger profile identified by *CDRST3_WARN*
  - The ActionTrigger *CDRST1_WARN* will monitor following StatsQueue Metric values:
  - ASR drop under 45 and a minimum of 3 CDRs in the StatsQueue will call Action profile *LOG_WARNING* which will log the StatsQueue to syslog. The Action will be recurrent with a sleep time of 1 minute.
  - ACD drop under 10 and a minimum of 5 CDRs in the StatsQueue will cause the same log to syslog. The Action will be recurrent with a sleep time of 1 minute.
  - ACC increase over 10 and a minimum of 5 CDRs in the StatsQueue will cause the StatsQueue to be again logged to syslog. The Action will be recurrent with a sleep time of 1 minute.
  - The ActionTrigger *CDRST1001_WARN* will monitor following StatsQueue Metric values:

- ASR drop under 65 and a minimum of 3 CDRs in the StatsQueue will call Action profile *LOG_WARNING* which will log the StatsQueue to syslog. The Action will be recurrent with a sleep time of 1 minute.
- ACD drop under 10 and a minimum of 5 CDRs in the StatsQueue will cause the same log to syslog. The Action will be recurrent with a sleep time of 1 minute.
- ACC increase over 5 and a minimum of 5 CDRs in the StatsQueue will cause the StatsQueue to be again logged to syslog. The Action will be recurrent with a sleep time of 1 minute.
- The ActionTrigger *CDRST3_WARN* will monitor ACD Metric and react at a minimum ACD of 60 with 5 CDRs in the StatsQueue by writing again to syslog. This ActionTrigger will be fired one time then cleared by the scheduler.

```
cgr-loader -verbose -path=/usr/share/cgrates/tariffplans/tutorial
```

To verify that all actions successfully performed, we use following *cgr-console* commands:

- Make sure all our balances were topped-up:

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1001"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1002"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1003"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1004"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1005"]'
```

- Query call costs so we can see our calls will have expected costs (final cost will result as sum of *ConnectFee* and *Cost* fields):

```
cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1002" TimeSta
cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1002" TimeSta
cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1003" TimeSta
cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1003" TimeSta
cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1004" TimeSta
cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1004" TimeSta
```

- Make sure *CDRStats Queues* were created:

```
cgr-console cdrstats_queueids
cgr-console 'cdrstats_metrics StatsQueueId="*default"'
```

Test calls

1001 -> 1002

Since the user 1001 is marked as *prepaid* inside the telecom switch, calling between 1001 and 1002 should generate pre-auth and prepaid debits which can be checked with *get_account* command integrated within *cgr-console* tool. Charging will be done based on time of day as described in the tariff plan definition above.

Note: An important particularity to note here is the ability of **CGRateS** SessionManager to refund units booked in advance (eg: if debit occurs every 10s and rate increments are set to 1s, the SessionManager will be smart enough to refund pre-booked credits for calls stoped in the middle of debit interval).

Check that 1001 balance is properly deducted, during the call, and moreover considering that general balance has priority over the shared one debits for this call should take place at first out of general balance.

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1001"]'
```

1002 -> 1001

The user 1002 is marked as *postpaid* inside the telecom switch hence his calls will be debited at the end of the call instead of during a call and his balance will be able to go on negative without influencing his new calls (no pre-auth).

To check that we had debits we use again console command, this time not during the call but at the end of it:

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1002"]'
```

1003 -> 1001

The user 1003 is marked as *pseudoprepaid* inside the telecom switch hence his calls will be considered same as prepaid (no call setups possible on negative balance due to pre-auth mechanism) but not handled automatically by session manager. His call costs will be calculated directly out of CDRs and balance updated by the time when mediation process occurs. This is sometimes a good compromise of prepaid running without influencing performance (there are no recurrent call debits during a call).

To check that there are no debits during or by the end of the call, but when the CDR reaches the CDRS component(which is close to real-time in case of *http-json* CDRs):

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1003"]'
```

1004 -> 1001

The user 1004 is marked as *rated* inside the telecom switch hence his calls not interact in any way with accounting subsystem. The only action perfomed by **CGRateS** related to his calls wil be rating/mediation of his CDRs.

1006 -> 1002

Since the user 1006 is marked as *prepaid* inside the telecom switch, calling between 1006 and 1002 should generate pre-auth and prepaid debits which can be checked with *get_account* command integrated within *cgr-console* tool. One thing to note here is that 1006 is not defined as an account inside CGR Accounting Subsystem but as an alias of another account, hence *get_account* ran on 1006 will return "not found" and the debits can be monitored on the real account which is 1001.

Check that 1001 balance is properly debitted, during the call, and moreover considering that general balance has priority over the shared one debits for this call should take place at first out of general balance.

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1006"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1001"]'
```

1007 -> 1002

Since the user 1007 is marked as *prepaid* inside the telecom switch, calling between 1007 and 1002 should generate pre-auth and prepaid debits which can be checked with *get_account* command integrated within *cgr-console* tool. Since 1007 has no units left into his accounts but he has one balance marked as shared, debits for this call should take place in accounts which are a part of the same shared balance as the one of *1007/SHARED_A*, which in our scenario corresponds to the one of the account 1001.

Check that call can proceed even if 1007 has no units left into his own balances, and that the costs attached to the call towards 1002 are debited from the balance marked as shared within account 1001.

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1007"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1001"]'
```

CDR Exporting

Once the CDRs are mediated, they are available to be exported. One can use available RPC APIs for that or directly call exports from console:

```
cgr-console 'cdrs_export CdrFormat="csv" ExportDir="/tmp"'
```

Fraud detection

Since we have configured some action triggers (more than 20 units of balance topped-up or less than 2 and more than 5 units spent on *FS_USERS* we should be notified over syslog when things like unexpected events happen (eg: fraud with more than 20 units topped-up). Most important is the monitor for 100 units topped-up which will also trigger an account disable together with killing it's calls if prepaid debits are used.

To verify this mechanism simply add some random units into one account's balance:

```
cgr-console 'balance_set Tenant="cgrates.org" Account="1003" Direction="*out" Value=23'
tail -f /var/log/syslog -n 20

cgr-console 'balance_set Tenant="cgrates.org" Account="1001" Direction="*out" Value=101'
tail -f /var/log/syslog -n 20
```

On the CDRs side we will be able to integrate CdrStats monitors as part of our Fraud Detection system (eg: the increase of average cost for 1001 and 1002 accounts will signal us abnormalities, hence we will be notified via syslog).

7.2 Kamailio Integration Tutorials

In these tutorials we exemplify few cases of integration between Kamailio and CGRateS. We start with common steps, installation and postinstall processes then we dive into particular configurations, depending on the case we run.

7.2.1 Software installation

As operating system we have choosen Debian Jessie, since all the software components we use provide packaging for it.

Kamailio

We got Kamailio installed via following commands:

```
apt-key adv --recv-keys --keyserver keyserver.ubuntu.com 0xfb40d3e6508ea4c8
echo "deb http://deb.kamailio.org/kamailio43 jessie main" > /etc/apt/sources.list.d/kamailio.list
apt-get update
apt-get install kamailio kamailio-extra-modules kamailio-json-modules
```

Once installed we proceed with loading the configuration out of specific tutorial cases bellow.

7.2.2 CGRateS Installation

As operating system we have choosen Debian Wheezy, since all the software components we use provide packaging for it.

Prerequisites

Some components of **CGRateS** (whether enabled or not is up to the administrator) depend on external software like:

- Git used by **CGRateS** History Server as archiver.
- Redis to serve as Rating and Accounting DB for **CGRateS**.
- MySQL to serve as StorDB for **CGRateS**.

We will install them in one shot using the command bellow.

```
apt-get install git redis-server mysql-server
```

Note: For simplicity sake we have used as MySQL root password when asked: *CGRateS.org*.

Installation

Installation steps are provided within **CGRateS** install documentation.

To get **CGRateS** installed execute the following commands over ssh console:

```
cd /etc/apt/sources.list.d/
wget -O - http://apt.itsyscom.com/conf/cgrates.gpg.key | apt-key add -
wget http://apt.itsyscom.com/conf/cgrates.apt.list
apt-get update
apt-get install cgrates
```

As described in post-install section, we will need to set up the MySQL database (using *CGRateS.org* as our root password):

```
cd /usr/share/cgrates/storage/mysql/
./setup_cgr_db.sh root CGRateS.org localhost
```

At this point we have **CGRateS** installed but not yet configured. To facilitate the understanding and speed up the process, **CGRateS** comes already with the configurations used in these tutorials, available in the */usr/share/cgrates/tutorials* folder, so we will load them custom on each tutorial case.

7.2.3 SIP UA - Jitsi

On our ubuntu desktop host, we have installed Jitsi to be used as SIP UA, out of stable provided packages on Jitsi download and had Jitsi configured with 4 accounts: 1001/1234, 1002/1234, 1003/1234 and 1004/1234.

7.2.4 Kamailio interaction via *evapi* module

Scenario

- Kamailio default configuration modified for **CGRateS** interaction. For script maintainability and simplicity we have separated CGRateS specific routes in *kamailio-cgrates.cfg* file which is included in main *kamailio.cfg* via include directive.

- Considering the following users (with configs hardcoded in the *kamailio.cfg* configuration script and loaded in htable): 1001-prepaid, 1002-postpaid, 1003-pseudoprepaid, 1004-rated, 1005-rated, 1006-prepaid, 1007-prepaid.
- **CGRateS** with following components:
- CGR-SM started as translator between Kamailio and CGR-Rater for both authorization events as well as accounting ones.
- CGR-CDRS component processing raw CDRs from CGR-SM component and storing them inside CGR StorDB.
- CGR-CDRE exporting rated CDRs from CGR StorDB (export path: */tmp*).
- CGR-History component keeping the archive of the rates modifications (path browsable with git client at */tmp/cgr_history*).

Starting Kamailio with custom configuration

```
/usr/share/cgrates/tutorials/kamevapi/kamailio/etc/init.d/kamailio start
```

To verify that Kamailio is running we run the console command:

```
kamctl moni
```

Starting CGRateS with custom configuration

```
/usr/share/cgrates/tutorials/kamevapi/cgrates/etc/init.d/cgrates start
```

Make sure that cgrates is running

```
cgr-console status
```

CDR processing

At the end of each call Kamailio will generate an CDR event via *evapi* and this will be directed towards the port configured inside *cgrates.json*. This event will reach inside **CGRateS** through the *SM* component (close to real-time). Once in-there it will be instantly rated and be ready for export.

CGRateS Usage

Since it is common to most of the tutorials, the example for **CGRateS** usage is provided in a separate page here

7.2.5 CGRateS Usage

Loading CGRateS Tariff Plans

Before proceeding to this step, you should have **CGRateS** installed and started with custom configuration, depending on the tutorial you have followed.

For our tutorial we load again prepared data out of shared folder, containing following rules:

- Create the necessary timings (always, asap, peak, offpeak).
- Configure 3 destinations (1002, 1003 and 10 used as catch all rule).

- As rating we configure the following:
- Rate id: *RT_10CNT* with connect fee of 20cents, 10cents per minute for the first 60s in 60s increments followed by 5cents per minute in 1s increments.
- Rate id: *RT_20CNT* with connect fee of 40cents, 20cents per minute for the first 60s in 60s increments, followed by 10 cents per minute charged in 1s increments.
- Rate id: *RT_40CNT* with connect fee of 80cents, 40cents per minute for the first 60s in 60s increments, follwed by 20cents per minute charged in 10s increments.
- Rate id: *RT_1CNT* having no connect fee and a rate of 1 cent per minute, chargeable in 1 minute increments.
- Will charge by default *RT_40CNT* for all 10xx destinations during peak times (Monday-Friday 08:00-19:00) and *RT_10CNT* during offpeatimes (rest).
- Account 1001 will receive a special *deal* for 1002 and 1003 destinations during peak times with *RT_20CNT*, otherwise having default rating.
- Accounting part will have following configured:
- Create 5 accounts: 1001, 1002, 1003, 1004, 1007.
- Create 1 account alias (1006 - alias of account 1002).
- Create 1 rating profile alias (1006 - alias of rating profile 1001).
- 1002, 1003, 1004 will receive 10units of **monetary* balance.
- 1001 will receive 5 units of general **monetary*, 5 units of shared balance in the shared group "SHARED_A" and 90 seconds of calling destination 1002 with special rates *RT_1CNT*.
- 1007 will receive 0 units of shared balance in the shared group "SHARED_A".
- Define the shared balance "SHARED_A" with debit policy **highest*.
- For each balance created, attach 4 triggers to control the balance: log on balance<2, log on balance>20, log on 5 mins talked towards 10xx destination, disable the account and log if a balance is higher than 100 units.
- *DerivedCharging* will execute one extra mediation run when the sessions will have as account and rating subject 1001 resulting in a cloned session with most of parameters identical to original except RequestType which will be set on *rated* instead of original *prepaid* one. The extra run will be identified by *derived_run1* in CDRs.
- Will configure 4 extra CdrStatQueues:
  - *CDRST1* with 10 CDRs in the Queue and unlimited time window, calculating *ASR*, *ACD* and *ACC* for CDRs with Tenant matching *cgrates.org* and MediationRunId matching *default*. On this StatsQueue we will attach an ActionTrigger profile identified by *CDRST1_WARN*
  - *CDRST_1001* with 10 CDRs in the Queue and 10 minutes time window calculating *ASR*, *ACD* and *ACC* for CDRs with Tenant matching *cgrates.org*, RatingSubject matching *1001* and MediationRunId matching *default*. On this StatsQueue we will attach an ActionTrigger profile identified by *CDRST1001_WARN*
  - *CDRST_1002* with 10 CDRs in the Queue and 10 minutes time window calculating *ASR*, *ACD* and *ACC* for CDRs with Tenant matching *cgrates.org*, RatingSubject matching *1002* and MediationRunId matching *default*. On this StatsQueue we will attach an ActionTrigger profile identified by *CDRST1001_WARN*
  - *CDRST_1003* with 10 CDRs in the Queue and 10 minutes time window calculating *ASR*, and *ACD* for CDRs with Tenant matching *cgrates.org*, Destination matching *1003* and MediationRunId matching *default*. On this StatsQueue we will attach an ActionTrigger profile identified by *CDRST3_WARN*
  - The ActionTrigger *CDRST1_WARN* will monitor following StatsQueue Metric values:

- ASR drop under 45 and a minimum of 3 CDRs in the StatsQueue will call Action profile *LOG_WARNING* which will log the StatsQueue to syslog. The Action will be recurrent with a sleep time of 1 minute.
- ACD drop under 10 and a minimum of 5 CDRs in the StatsQueue will cause the same log to syslog. The Action will be recurrent with a sleep time of 1 minute.
- ACC increase over 10 and a minimum of 5 CDRs in the StatsQueue will cause the StatsQueue to be again logged to syslog. The Action will be recurrent with a sleep time of 1 minute.
- The ActionTrigger *CDRST1001_WARN* will monitor following StatsQueue Metric values:
- ASR drop under 65 and a minimum of 3 CDRs in the StatsQueue will call Action profile *LOG_WARNING* which will log the StatsQueue to syslog. The Action will be recurrent with a sleep time of 1 minute.
- ACD drop under 10 and a minimum of 5 CDRs in the StatsQueue will cause the same log to syslog. The Action will be recurrent with a sleep time of 1 minute.
- ACC increase over 5 and a minimum of 5 CDRs in the StatsQueue will cause the StatsQueue to be again logged to syslog. The Action will be recurrent with a sleep time of 1 minute.
- The ActionTrigger *CDRST3_WARN* will monitor ACD Metric and react at a minimum ACD of 60 with 5 CDRs in the StatsQueue by writing again to syslog. This ActionTrigger will be fired one time then cleared by the scheduler.

```
cgr-loader -verbose -path=/usr/share/cgrates/tariffplans/tutorial
```

To verify that all actions successfully performed, we use following *cgr-console* commands:

- Make sure all our balances were topped-up:

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1001"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1002"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1003"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1004"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1005"]'
```

- Query call costs so we can see our calls will have expected costs (final cost will result as sum of *ConnectFee* and *Cost* fields):

```
cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1002" TimeSta
cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1002" TimeSta
cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1003" TimeSta
cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1003" TimeSta
cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1004" TimeSta
cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1004" TimeSta
```

- Make sure *CDRStats Queues* were created:

```
cgr-console cdrstats_queueids
cgr-console 'cdrstats_metrics StatsQueueId="*default"'
```

Test calls

1001 -> 1002

Since the user 1001 is marked as *prepaid* inside the telecom switch, calling between 1001 and 1002 should generate pre-auth and prepaid debits which can be checked with *get_account* command integrated within *cgr-console* tool. Charging will be done based on time of day as described in the tariff plan definition above.

Note: An important particularity to note here is the ability of **CGRateS** SessionManager to refund units booked in advance (eg: if debit occurs every 10s and rate increments are set to 1s, the SessionManager will be smart enough to refund pre-booked credits for calls stoped in the middle of debit interval).

Check that 1001 balance is properly deducted, during the call, and moreover considering that general balance has priority over the shared one debits for this call should take place at first out of general balance.

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1001"]'
```

1002 -> 1001

The user 1002 is marked as *postpaid* inside the telecom switch hence his calls will be debited at the end of the call instead of during a call and his balance will be able to go on negative without influencing his new calls (no pre-auth).

To check that we had debits we use again console command, this time not during the call but at the end of it:

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1002"]'
```

1003 -> 1001

The user 1003 is marked as *pseudoprepaid* inside the telecom switch hence his calls will be considered same as prepaid (no call setups possible on negative balance due to pre-auth mechanism) but not handled automatically by session manager. His call costs will be calculated directly out of CDRs and balance updated by the time when mediation process occurs. This is sometimes a good compromise of prepaid running without influencing performance (there are no recurrent call debits during a call).

To check that there are no debits during or by the end of the call, but when the CDR reaches the CDRS component(which is close to real-time in case of *http-json* CDRs):

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1003"]'
```

1004 -> 1001

The user 1004 is marked as *rated* inside the telecom switch hence his calls not interact in any way with accounting subsystem. The only action perfomed by **CGRateS** related to his calls wil be rating/mediation of his CDRs.

1006 -> 1002

Since the user 1006 is marked as *prepaid* inside the telecom switch, calling between 1006 and 1002 should generate pre-auth and prepaid debits which can be checked with *get_account* command integrated within *cgr-console* tool. One thing to note here is that 1006 is not defined as an account inside CGR Accounting Subsystem but as an alias of another account, hence *get_account* ran on 1006 will return "not found" and the debits can be monitored on the real account which is 1001.

Check that 1001 balance is properly debitted, during the call, and moreover considering that general balance has priority over the shared one debits for this call should take place at first out of general balance.

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1006"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1001"]'
```

1007 -> 1002

Since the user 1007 is marked as *prepaid* inside the telecom switch, calling between 1007 and 1002 should generate pre-auth and prepaid debits which can be checked with *get_account* command integrated within *cgr-console* tool. Since 1007 has no units left into his accounts but he has one balance marked as shared, debits for this call should take place in accounts which are a part of the same shared balance as the one of *1007/SHARED_A*, which in our scenario corresponds to the one of the account 1001.

Check that call can proceed even if 1007 has no units left into his own balances, and that the costs attached to the call towards 1002 are debited from the balance marked as shared within account 1001.

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1007"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1001"]'
```

CDR Exporting

Once the CDRs are mediated, they are available to be exported. One can use available RPC APIs for that or directly call exports from console:

```
cgr-console 'cdrs_export CdrFormat="csv" ExportDir="/tmp"'
```

Fraud detection

Since we have configured some action triggers (more than 20 units of balance topped-up or less than 2 and more than 5 units spent on *FS_USERS* we should be notified over syslog when things like unexpected events happen (eg: fraud with more than 20 units topped-up). Most important is the monitor for 100 units topped-up which will also trigger an account disable together with killing it's calls if prepaid debits are used.

To verify this mechanism simply add some random units into one account's balance:

```
cgr-console 'balance_set Tenant="cgrates.org" Account="1003" Direction="*out" Value=23'
tail -f /var/log/syslog -n 20

cgr-console 'balance_set Tenant="cgrates.org" Account="1001" Direction="*out" Value=101'
tail -f /var/log/syslog -n 20
```

On the CDRs side we will be able to integrate CdrStats monitors as part of our Fraud Detection system (eg: the increase of average cost for 1001 and 1002 accounts will signal us abnormalities, hence we will be notified via syslog).

7.3 OpenSIPS Integration Tutorials

In these tutorials we exemplify few cases of integration between OpenSIPS and CGRateS. We start with common steps, installation and postinstall processes then we dive into particular configurations, depending on the case we run.

7.3.1 Software installation

As operating system we have choosen Debian Jessie, since all the software components we use provide packaging for it.

OpenSIPS

We got OpenSIPS installed via following commands:

```
wget -O - http://apt.opensips.org/key.asc | apt-key add -
echo "deb http://apt.opensips.org/debian/stable-2.1/jessie opensips-2.1-jessie main" > /etc/apt/sourc
apt-get update
apt-get install opensips opensips-json-module opensips-restclient-module
```

Once installed we proceed with loading the configuration out of specific tutorial cases bellow.

7.3.2 CGRateS Installation

As operating system we have choosen Debian Wheezy, since all the software components we use provide packaging for it.

Prerequisites

Some components of **CGRateS** (whether enabled or not is up to the administrator) depend on external software like:

- Git used by **CGRateS** History Server as archiver.
- Redis to serve as Rating and Accounting DB for **CGRateS**.
- MySQL to serve as StorDB for **CGRateS**.

We will install them in one shot using the command bellow.

```
apt-get install git redis-server mysql-server
```

Note: For simplicity sake we have used as MySQL root password when asked: *CGRateS.org*.

Installation

Installation steps are provided within **CGRateS** install documentation.

To get **CGRateS** installed execute the following commands over ssh console:

```
cd /etc/apt/sources.list.d/
wget -O - http://apt.itsyscom.com/conf/cgrates.gpg.key | apt-key add -
wget http://apt.itsyscom.com/conf/cgrates.apt.list
apt-get update
apt-get install cgrates
```

As described in post-install section, we will need to set up the MySQL database (using *CGRateS.org* as our root password):

```
cd /usr/share/cgrates/storage/mysql/
./setup_cgr_db.sh root CGRateS.org localhost
```

At this point we have **CGRateS** installed but not yet configured. To facilitate the understanding and speed up the process, **CGRateS** comes already with the configurations used in these tutorials, available in the */usr/share/cgrates/tutorials* folder, so we will load them custom on each tutorial case.

7.3.3 SIP UA - Jitsi

On our ubuntu desktop host, we have installed Jitsi to be used as SIP UA, out of stable provided packages on Jitsi download and had Jitsi configured with 4 accounts: 1001/1234, 1002/1234, 1003/1234 and 1004/1234.

7.3.4 OpenSIPS interaction via *event_datagram*

Scenario

- OpenSIPS out of *residential* configuration generated.
- Considering the following users (with configs hardcoded in the *opensips.cfg* configuration script): 1002-postpaid, 1003-pseudoprepaid, 1004-rated, 1007-rated.
- For simplicity we configure no authentication (WARNING: Not for production usage).
- **CGRateS** with following components:
- CGR-SM started as translator between OpenSIPS and **cgr-rater** for both authorization events (pseudoprepaid) as well as CDR ones.
- CGR-CDRS component processing raw CDRs from CGR-SM component and storing them inside CGR StorDB.
- CGR-CDRE exporting rated CDRs from CGR StorDB (export path: */tmp*).
- CGR-History component keeping the archive of the rates modifications (path browsable with git client at */tmp/cgr_history*).

Starting OpenSIPS with custom configuration

```
/usr/share/cgrates/tutorials/osips_async/opensips/etc/init.d/opensips start
```

To verify that OpenSIPS is running we run the console command:

```
opensipsctl moni
```

Starting CGRateS with custom configuration

```
/usr/share/cgrates/tutorials/osips_async/cgrates/etc/init.d/cgrates start
```

Make sure that cgrates is running

```
cgr-console status
```

CDR processing

At the end of each call OpenSIPS will generate an CDR event and due to automatic handler registration built in **CGRateS-SM** component, this will be directed towards the port configured inside *cgrates.json*. This event will reach inside **CGRateS** through the *SM* component (close to real-time). Once in-there it will be instantly rated and be ready for export.

CGRateS Usage

Since it is common to most of the tutorials, the example for **CGRateS** usage is provided in a separate page here

7.3.5 CGRateS Usage

Loading CGRateS Tariff Plans

Before proceeding to this step, you should have **CGRateS** installed and started with custom configuration, depending on the tutorial you have followed.

For our tutorial we load again prepared data out of shared folder, containing following rules:

- Create the necessary timings (always, asap, peak, offpeak).
- Configure 3 destinations (1002, 1003 and 10 used as catch all rule).
- As rating we configure the following:
- Rate id: *RT_10CNT* with connect fee of 20cents, 10cents per minute for the first 60s in 60s increments followed by 5cents per minute in 1s increments.
- Rate id: *RT_20CNT* with connect fee of 40cents, 20cents per minute for the first 60s in 60s increments, followed by 10 cents per minute charged in 1s increments.
- Rate id: *RT_40CNT* with connect fee of 80cents, 40cents per minute for the first 60s in 60s increments, follwed by 20cents per minute charged in 10s increments.
- Rate id: *RT_1CNT* having no connect fee and a rate of 1 cent per minute, chargeable in 1 minute increments.
- Will charge by default *RT_40CNT* for all 10xx destinations during peak times (Monday-Friday 08:00-19:00) and *RT_10CNT* during offpeatimes (rest).
- Account 1001 will receive a special *deal* for 1002 and 1003 destinations during peak times with *RT_20CNT*, otherwise having default rating.
- Accounting part will have following configured:
- Create 5 accounts: 1001, 1002, 1003, 1004, 1007.
- Create 1 account alias (1006 - alias of account 1002).
- Create 1 rating profile alias (1006 - alias of rating profile 1001).
- 1002, 1003, 1004 will receive 10units of **monetary* balance.
- 1001 will receive 5 units of general **monetary*, 5 units of shared balance in the shared group "SHARED_A" and 90 seconds of calling destination 1002 with special rates *RT_1CNT*.
- 1007 will receive 0 units of shared balance in the shared group "SHARED_A".
- Define the shared balance "SHARED_A" with debit policy **highest*.
- For each balance created, attach 4 triggers to control the balance: log on balance<2, log on balance>20, log on 5 mins talked towards 10xx destination, disable the account and log if a balance is higher than 100 units.
- *DerivedCharging* will execute one extra mediation run when the sessions will have as account and rating subject 1001 resulting in a cloned session with most of parameters identical to original except RequestType which will be set on *rated* instead of original *prepaid* one. The extra run will be identified by *derived_run1* in CDRs.
- Will configure 4 extra CdrStatQueues:
    - *CDRST1* with 10 CDRs in the Queue and unlimited time window, calculating *ASR*, *ACD* and *ACC* for CDRs with Tenant matching *cgrates.org* and MediationRunId matching *default*. On this StatsQueue we will attach an ActionTrigger profile identified by *CDRST1_WARN*

- *CDRST_1001* with 10 CDRs in the Queue and 10 minutes time window calculating *ASR*, *ACD* and *ACC* for CDRs with Tenant matching *cgrates.org*, RatingSubject matching *1001* and MediationRunId matching *default*. On this StatsQueue we will attach an ActionTrigger profile identified by *CDRST1001_WARN*
- *CDRST_1002* with 10 CDRs in the Queue and 10 minutes time window calculating *ASR*, *ACD* and *ACC* for CDRs with Tenant matching *cgrates.org*, RatingSubject matching *1002* and MediationRunId matching *default*. On this StatsQueue we will attach an ActionTrigger profile identified by *CDRST1001_WARN*
- *CDRST_1003* with 10 CDRs in the Queue and 10 minutes time window calculating *ASR*, and *ACD* for CDRs with Tenant matching *cgrates.org*, Destination matching *1003* and MediationRunId matching *default*. On this StatsQueue we will attach an ActionTrigger profile identified by *CDRST3_WARN*
- The ActionTrigger *CDRST1_WARN* will monitor following StatsQueue Metric values:
- ASR drop under 45 and a minimum of 3 CDRs in the StatsQueue will call Action profile *LOG_WARNING* which will log the StatsQueue to syslog. The Action will be recurrent with a sleep time of 1 minute.
- ACD drop under 10 and a minimum of 5 CDRs in the StatsQueue will cause the same log to syslog. The Action will be recurrent with a sleep time of 1 minute.
- ACC increase over 10 and a minimum of 5 CDRs in the StatsQueue will cause the StatsQueue to be again logged to syslog. The Action will be recurrent with a sleep time of 1 minute.
- The ActionTrigger *CDRST1001_WARN* will monitor following StatsQueue Metric values:
- ASR drop under 65 and a minimum of 3 CDRs in the StatsQueue will call Action profile *LOG_WARNING* which will log the StatsQueue to syslog. The Action will be recurrent with a sleep time of 1 minute.
- ACD drop under 10 and a minimum of 5 CDRs in the StatsQueue will cause the same log to syslog. The Action will be recurrent with a sleep time of 1 minute.
- ACC increase over 5 and a minimum of 5 CDRs in the StatsQueue will cause the StatsQueue to be again logged to syslog. The Action will be recurrent with a sleep time of 1 minute.
- The ActionTrigger *CDRST3_WARN* will monitor ACD Metric and react at a minimum ACD of 60 with 5 CDRs in the StatsQueue by writing again to syslog. This ActionTrigger will be fired one time then cleared by the scheduler.

```
cgr-loader -verbose -path=/usr/share/cgrates/tariffplans/tutorial
```

To verify that all actions successfully performed, we use following *cgr-console* commands:

- Make sure all our balances were topped-up:

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1001"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1002"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1003"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1004"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1005"]'
```

- Query call costs so we can see our calls will have expected costs (final cost will result as sum of *ConnectFee* and *Cost* fields):

```
cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1002" TimeSta
cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1002" TimeSta
cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1003" TimeSta
cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1003" TimeSta
```

```
    cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1004" TimeSta
    cgr-console 'cost Category="call" Tenant="cgrates.org" Subject="1001" Destination="1004" TimeSta
```

- Make sure *CDRStats Queues* were created:

```
    cgr-console cdrstats_queueids
    cgr-console 'cdrstats_metrics StatsQueueId="*default"'
```

Test calls

1001 -> 1002

Since the user 1001 is marked as *prepaid* inside the telecom switch, calling between 1001 and 1002 should generate pre-auth and prepaid debits which can be checked with *get_account* command integrated within *cgr-console* tool. Charging will be done based on time of day as described in the tariff plan definition above.

Note: An important particularity to note here is the ability of **CGRateS** SessionManager to refund units booked in advance (eg: if debit occurs every 10s and rate increments are set to 1s, the SessionManager will be smart enough to refund pre-booked credits for calls stoped in the middle of debit interval).

Check that 1001 balance is properly deducted, during the call, and moreover considering that general balance has priority over the shared one debits for this call should take place at first out of general balance.

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1001"]'
```

1002 -> 1001

The user 1002 is marked as *postpaid* inside the telecom switch hence his calls will be debited at the end of the call instead of during a call and his balance will be able to go on negative without influencing his new calls (no pre-auth).

To check that we had debits we use again console command, this time not during the call but at the end of it:

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1002"]'
```

1003 -> 1001

The user 1003 is marked as *pseudoprepaid* inside the telecom switch hence his calls will be considered same as prepaid (no call setups possible on negative balance due to pre-auth mechanism) but not handled automatically by session manager. His call costs will be calculated directly out of CDRs and balance updated by the time when mediation process occurs. This is sometimes a good compromise of prepaid running without influencing performance (there are no recurrent call debits during a call).

To check that there are no debits during or by the end of the call, but when the CDR reaches the CDRS component(which is close to real-time in case of *http-json* CDRs):

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1003"]'
```

1004 -> 1001

The user 1004 is marked as *rated* inside the telecom switch hence his calls not interact in any way with accounting subsystem. The only action perfomed by **CGRateS** related to his calls wil be rating/mediation of his CDRs.

1006 -> 1002

Since the user 1006 is marked as *prepaid* inside the telecom switch, calling between 1006 and 1002 should generate pre-auth and prepaid debits which can be checked with *get_account* command integrated within *cgr-console* tool. One thing to note here is that 1006 is not defined as an account inside CGR Accounting Subsystem but as an alias of another account, hence *get_account* ran on 1006 will return "not found" and the debits can be monitored on the real account which is 1001.

Check that 1001 balance is properly debitted, during the call, and moreover considering that general balance has priority over the shared one debits for this call should take place at first out of general balance.

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1006"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1001"]'
```

1007 -> 1002

Since the user 1007 is marked as *prepaid* inside the telecom switch, calling between 1007 and 1002 should generate pre-auth and prepaid debits which can be checked with *get_account* command integrated within *cgr-console* tool. Since 1007 has no units left into his accounts but he has one balance marked as shared, debits for this call should take place in accounts which are a part of the same shared balance as the one of *1007/SHARED_A*, which in our scenario corresponds to the one of the account 1001.

Check that call can proceed even if 1007 has no units left into his own balances, and that the costs attached to the call towards 1002 are debited from the balance marked as shared within account 1001.

```
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1007"]'
cgr-console 'accounts Tenant="cgrates.org" AccountIds=["1001"]'
```

CDR Exporting

Once the CDRs are mediated, they are available to be exported. One can use available RPC APIs for that or directly call exports from console:

```
cgr-console 'cdrs_export CdrFormat="csv" ExportDir="/tmp"'
```

Fraud detection

Since we have configured some action triggers (more than 20 units of balance topped-up or less than 2 and more than 5 units spent on *FS_USERS* we should be notified over syslog when things like unexpected events happen (eg: fraud with more than 20 units topped-up). Most important is the monitor for 100 units topped-up which will also trigger an account disable together with killing it's calls if prepaid debits are used.

To verify this mechanism simply add some random units into one account's balance:

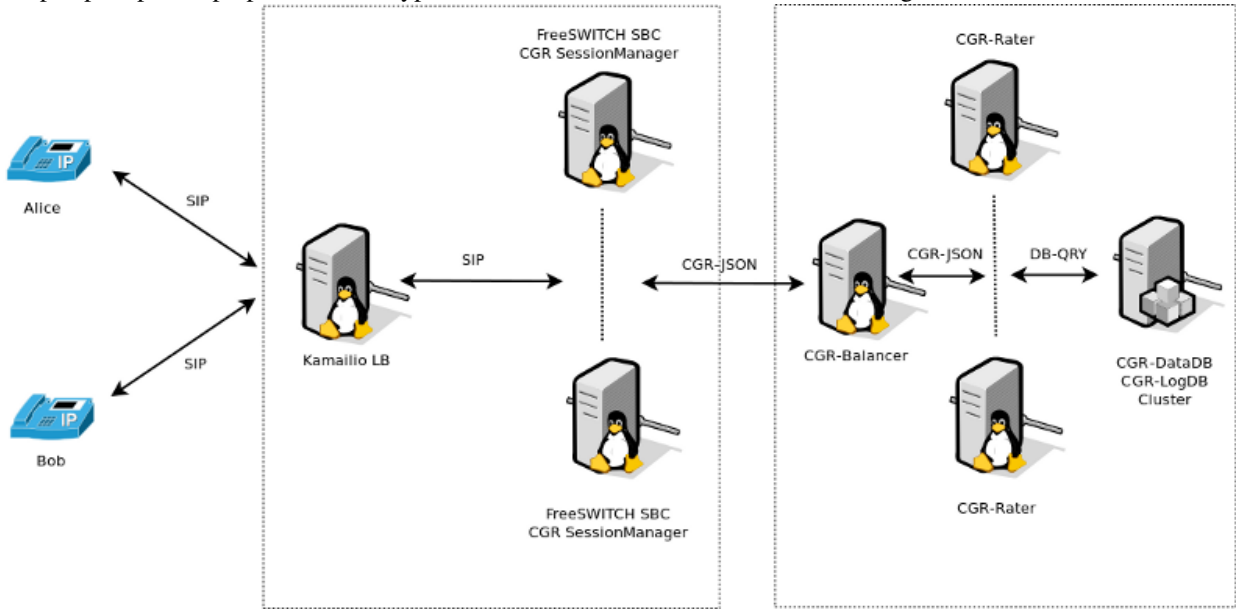
```
cgr-console 'balance_set Tenant="cgrates.org" Account="1003" Direction="*out" Value=23'
tail -f /var/log/syslog -n 20

cgr-console 'balance_set Tenant="cgrates.org" Account="1001" Direction="*out" Value=101'
tail -f /var/log/syslog -n 20
```

On the CDRs side we will be able to integrate CdrStats monitors as part of our Fraud Detection system (eg: the increase of average cost for 1001 and 1002 accounts will signal us abnormalities, hence we will be notified via syslog).

8. Miscellaneous

8.1 8.1. FreeSWITCH integration

FreeSWITCH used as Telecom Switch is fully supported by all of the 4 rating modes: prepaid/postpaid/pseudoprepaid/rated. A typical use case would be like the one in the diagram bellow:



The process of rating is decoupled into two different components:

8.1.1 8.1.1. SessionManager

- Attached to FreeSWITCH via the socket library, enhancing CGRateS with real-time call monitoring and call control functions.
- **In Prepaid mode implements the following behaviour:**
  - **On *CHANNEL_PARK* event received from FreeSWITCH:**
    - Authorize the call by calling *GetMaxSessionTime* on the Rater.
    - **Sets the channel variable *cgr_notify* via *uuid_setvar* to one of the following values:**

        - MISSING_PARAMETER: if one of the required channel variables is missing and CGRateS cannot make rating.
        - SYSTEM_ERROR: if rating could not be performed due to a system error.
        - INSUFFICIENT_FUNDS: if MaximSessionTime is 0.
        - AUTH_OK: Call is authorized to proceed.
      * Un-Park the call via *uuid_transfer* to original dialed number. The FreeSWITCH administrator is expected to make use of *cgr_notify* variable value to either allow the call going further or reject it (eg: towards an IVR or returning authorization fail message to call originator).
    - **On *CHANNEL_ANSWER* event received:**
      * Index the call into CGRateS's cache.
      * Starts debit loop by calling at configured interval *MaxDebit* on the Rater.
      * **If any of the debits fail:**
        - Set *cgr_notify* channel variable to either SYSTEM_ERROR in case of errors or INSUFFICIENT_FUNDS of there would be not enough balance for the next debit to proceed.
        - Send *hangup* command with cause *MANAGER_REQUEST*.
    - **On *CHANNEL_HANGUP_COMPLETE* event received:**
      * Refund the reserved balance back to the user's account (works for both monetary and minutes debited).
      * Save call costs into CGRateS LogDB.
- In Postpaid mode:
  - **On *CHANNEL_ANSWER* event received:**
    * Index the call into CGRateS's cache.
  - **On *CHANNEL_HANGUP_COMPLETE* event received:**
    * Call *Debit* RPC method on the Rater.
    * Save call costs into CGRateS LogDB.
- **On CGRateS Shutdown execute, for security reasons, hangup commands on calls which can be CGR related:**
  - *hupall MANAGER_REQUEST cgr_reqtype prepaid*
  - *hupall MANAGER_REQUEST cgr_reqtype postpaid*

8.1.2 8.1.2. Mediator

Attaches costs to FreeSWITCH native written .csv files. Since writing channel variables during hangup is asynchronous and can be missed by the CDR recorder mechanism of FreeSWITCH, we decided to keep this as separate process after the call is completed and do not write the costs via channel variables.

8.1.2.1. Modes of operation

The Mediator process for FreeSWITCH works in two different modes:

- **Costs from LogDB (activated by setting -1 as *subject_idx* in the *cgrates.cfg*:**

  - Queries LogDB for a previous saved price by SessionManager.
  - This behavior is typical for prepaid/postpaid calls which were previously processed by SessionManager and important in the sense that we write in CDRs exactly what was billed real-time from user's account.

- **Costs queried from Rater:**
  - This mode is specific for multiple process mediation and does not necessary reflect the price which was deducted from the user's account during real-time rating.
  - Another application for this mode is pseudoprepaid when there is no SessionManager monitoring and charging calls in real-time (debit done directly from CDRs).
  - This mode is triggered from configuration file by setting proper indexes (or leave them defaults if cgrates rating template is using whitin FreeSWITCH cdr_csv configuration file.

A typical usage into our implementations is a combination between the two modes of operation (by setting at a minimum -1 as subject_idx to run from LogDB and succesive mediation processes with different indexes).

8.1.2.2. Implementation logic

- The Mediator process is configured and started in the *cgrates.cfg* file and is alive as long as the *cgr-engine* application is on.
- To avoid concurrency issues, the Mediator does not process active maintained CDR csv files by FreeSWITCH but picks them up as soon as FreeSWITCH has done with them by rotating. The information about rotation comes in real-time on the Linux OS through the use of inotify.
- Based on configured indexes in the configuration file, the Mediator will start multiple processes for the same CDR.
- For each mediation process configured the Mediator will apped the original CDR with costs calculated. In case of errors of some kind, the value *-1* will be prepended.
- When mediation is completed on a file, the file will be moved to configured *cdr_out_dir* path.

[WIKI2015] http://en.wikipedia.org/wiki/Least-cost_routing