Deadlock/Server

Release 0.1

June 08, 2016

Start here

The server performs the following functions:

- holds the authoritative version of the access rules - also generates the offline database for controllers
- collects access logs
- provides software updates and time synchronization for the other devices
- monitors system state (and reports it to the management UIs)
- provides an API for the outside world (and all our tools).

Except for the contents of the database, it is entirely stateless. This simplifies the code and makes replication and failover trivial.

The server is separated into 3 independent modules: `deadserver` (communicates with controllers), `deadapi` (API for the outside world), and `deadaux` (auxiliary jobs).

Contents:

CHAPTER 1

Concepts and common abstractions:

1.1 Introduction

TODO

1.2 `cfiles` – the “common files” abstraction

CHAPTER 2

Communication with controllers:

CHAPTER 3

HTTP API for the outside world:

Contents:

3.1 Introduction

`deadapi` provides the HTTP API used by the web management and monitoring interface, and the provided commandline interface. Thereby bridges the outside world and the database via a simple CRUD REST API.

It supports pushing events via a streamed long-running HTTP response, in accordance with the Server-sent events/Eventsource specification. Events are triggered via the LISTEN/NOTIFY pub/sub mechanism in Postgres, and the database in turn contains triggers that send a NOTIFY on certain table row changes. Therefore data changes can bubble all the way to clients, which can use the standard Eventsource API to subscribe to these.

It provides a quick way to stage firmware updates: a firmware image together with a list of controller IDs can be uploaded, and `deadapi` simply drops (or links) the file into subfolders dedicated to the given controllers (see Common files).

3.2 HTTP API

TODO generate this somehow

3.3 Server-sent events

CHAPTER 4

Auxiliary jobs supporting the other components:

CHAPTER 5

API documentation

5.1 common package

5.1.1 Subpackages

common.cfiles package

Submodules

common.cfiles.filetypes module

class common.cfiles.filetypes.**FileType**
Bases: enum.Enum

common.cfiles.filetypes.**filemeta**(*filename*)

common.cfiles.filetypes.**filename**(*ftype*, *version*)

common.cfiles.filetypes.**get_latest**(*fs*, *ftype*, *controller=None*)

common.cfiles.filetypes.**latest_filename**(*ftype*)

common.cfiles.filetypes.**set_latest**(*fs*, *ftype*, *version*, *controller=None*)

common.cfiles.fs module

Controller files: opens files meant to be transferred to controllers via the XFER message.

class common.cfiles.fs.**ControllerFiles**(*directory*)
Bases: object

open(*name*, *controller=None*, **args*, ***kwargs*)
Opens files for the given controller.

controller can be either a controller ID, or *None* for files common to all controllers.

See also *open_for* and *path_for*.

open_with_common(*name*, *controller=None*)
Opens a file: if one for this controller doesn't exist, falls back to a common one.

Returns the file object opened as read-only in binary mode and unbuffered.

path(*name*, *controller=None*)
Returns the real path to the file for this controller.

controller can be either a controller ID, or *None* for files common to all controllers.

path_with_common(*name*, *controller=None*)
Resolves a path: if this file for this controller doesn't exist, returns a common one.

Returns the real path to the file.

exception common.cfiles.fs.**NoSuchFile**(*name*, *controller*)
Bases: RuntimeError

common.cfiles.fs.**mkdirnx**(*path*)
mkdir if does not exist

Module contents

common.types package

Submodules

common.types.ipaddr module

class common.types.ipaddr.**IPaddr**(*addr*)
Bases: object

IPv4 or IPv6 address that can be stuffed into CBOR and YAML.

cbor_tag = 56834

classmethod from_cbor(*ip*)

classmethod from_yaml(*loader*, *node*)

to_cbor()

to_yaml(*dumper*)

yaml_tag = '!IP'

common.types.record module

class common.types.record.**Record**(**args*, ***kwargs*)
Bases: dict

cbor_tag = 56833

classmethod from_cbor(*obj*)

to_cbor()

common.types.serializable module

common.types.serializable.**cbor_decode**(*data*)

common.types.serializable.**cbor_encode**(*data*)

common.types.serializable.**cbor_friendly**(*obj*)
Coerce the given item into a more or less equivalent CBOR-serializable type.

common.types.serializable.**cbor_serializable**(*cls*)

common.types.serializable.**get_cbor_coder**()

common.types.serializable.**yaml_serializable**(*cls*)

common.types.utils module

common.types.utils.**dict_to_records_and_tags**(*d*)
Parses a dict into types.Record and cbor.Tag recursively.

A dict with a single string key marked like <THIS> will be interpreted as a Tag, everything else as a Record.

common.types.utils.**prettyprint**(*rec*)

common.types.utils.**prettyread**(*buf*)

common.types.utils.**records_and_tags_to_dict**(*r*)

Module contents

common.utils package

Submodules

common.utils.conversions module

common.utils.conversions.**bytes2mac**(*buf*)

common.utils.conversions.**mac2bytes**(*s*)

common.utils.db module

Helpers for working with the DB.

All of these expect to be passed a records.Database wrapping a psycopg2 connection.

class common.utils.db.**Notify**(*channel*, *payload*)
Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *channel*, *payload*)
Create new instance of Notify(channel, payload)

__repr__()
Return a nicely formatted representation string

channel
Alias for field number 0

payload
Alias for field number 1

common.utils.db.**listen_for_notify**(*db*, *channels*, *callback*, *debounce_timeout=1*)
Listen for Postgres's NOTIFY, debouncing/deduplicating notifications.

debounce_timeout is in seconds.

common.utils.logging module

class common.utils.logging.**Formatter**(*fmt=None*, *datefmt=None*, *style='%'*)
Bases: logging.Formatter

Deadlock-friendly formatter for logging. It likes to shorten things.

Changes from logging.Formatter:

- strips first name component from logger name, as it is redundant in both prod and devel config

format(*record*)

Module contents

5.1.2 Submodules

5.1.3 common.rules module

Evaluates access rules.

Depends on the *in_expr* table being alive and well – see *sql/01-materialize-rules.sql*.

common.rules.**ask**(*db*, *accesspoint*, *when*, *identity*)
Evaluate the rules and return True iff access is allowed.

Parameters: db: something with a *records.Database*-like interface accesspoint: ID of the accesspoint, as specified in the DB time: *datetime.datetime* identity: ID of the identity, as specified in the DB

You probably want to use this with *common.utils.db.transaction*.

5.1.4 common.tag_names module

common.tag_names.**name**(*tag*)

common.tag_names.**value**(*tag*)

5.1.5 common.tags module

5.1.6 Module contents

5.2 config package

5.2.1 Submodules

5.2.2 config.defaults module

5.2.3 config.local module

5.2.4 Module contents

5.3 controller module

5.4 deadapi package

5.4.1 Subpackages

deadapi.resources package

Submodules

deadapi.resources.accesslog module

class deadapi.resources.accesslog.**AccessLog**(*db*)
Bases: *deadapi.utils.Resource*

GET(**params*)

default_params = {'limit': 200}

deadapi.resources.accesspoint module

class deadapi.resources.accesspoint.**AccessPoint**(*db*)
Bases: *deadapi.utils.Resource*

deadapi.resources.controller module

class deadapi.resources.controller.**Controller**(*db*)
Bases: *deadapi.utils.Resource*

GET()
List all controllers.

POST(**params*)
Add a controller, generating a key.

deadapi.resources.identity_expr module

class deadapi.resources.identity_expr.**IdentityExpr**(*db*)
Bases: *deadapi.utils.Resource*

GET(*id*)

deadapi.resources.ruleset module

class deadapi.resources.ruleset.**Ruleset**(*db*)
Bases: *deadapi.utils.Resource*

GET(*id=None*)

deadapi.resources.status module

class deadapi.resources.status.**Status**(*db*)
Bases: *deadapi.utils.Resource*

GET()

Module contents

5.4.2 Submodules

5.4.3 deadapi.api module

The HTTP API to the Deadlock server. Defines API endpoints / HTTP resource mountpoints.

class deadapi.api.**Root**(*db*)
Bases: object

exposed = True

5.4.4 deadapi.events module

Translates Postgres NOTIFY to HTTP Eventsource.

class deadapi.events.**EventSource**(*db*, *channels_map*)
Bases: object

Translates Postgres NOTIFY to HTTP Eventsource.

GET()

exposed = True

class deadapi.events.**Events**(*db*, *channels_map*)
Bases: object

Fans out notifies into all queues gotten from *self.get_queue*.

Translates event names according to the given channels map.

DEBOUNCE_TIMEOUT = 2

forward_notify(*notify*)

get_queue()

listen()

5.4.5 deadapi.server module

5.4.6 deadapi.utils module

class deadapi.utils.**Resource**(*db*)
Bases: object

An exposed RESTful resource that likes JSON and needs a DB connection.

exposed = True

deadapi.utils.**header**(*key*, *value*)

deadapi.utils.**json_handler**(**args*, ***kwargs*)
Streaming JSON encoder for CherryPy. It can handle a few extra types.

deadapi.utils.**m**(*orig*, **over*)
Return a new dict merged from the given ones. Last wins.

5.4.7 Module contents

The HTTP API for Deadlock management, rules configuration and status monitoring.

5.5 deadaux package

5.5.1 Submodules

5.5.2 deadaux.echotest module

5.5.3 deadaux.offlinedb module

5.5.4 deadaux.utils module

5.5.5 Module contents

5.6 deadcli module

5.7 deadserver package

5.7.1 Subpackages

deadserver.handlers package

Submodules

deadserver.handlers.alog module

Handler for ALOG requests: receive access logs

deadserver.handlers.alog.**handle**(*controller*, *data*, *ctx*)

Handles the ALOG message.

Messages are idempotent, which in this case means that every particular (controller, time, card_id, allowed) combination will be recorded at most once, even if such a packet is received multiple times.

deadserver.handlers.ask module

Handler for ASK requests: should I allow access now?

deadserver.handlers.ask.**handle**(*controller*, *data*, *ctx*)

deadserver.handlers.defs module

Provides functions for defining request handlers, such as the *handles(msg_type)* decorator.

deadserver.handlers.defs.**get_handler_for**(*msg_type*)

deadserver.handlers.defs.**handles**(*msg_type*)

deadserver.handlers.echotest module

Handler for ECHOTEST requests: echo for testing purposes

`deadserver.handlers.echotest.`**`handle_echotest`**(*controller*, *data*, *ctx*)

deadserver.handlers.ping module

Handler for PING requests: keepalive, DB and FW version info

`deadserver.handlers.ping.`**`get_latest_or_0`**(*cf*, *ftype*, *controller*)

`deadserver.handlers.ping.`**`handle`**(*controller*, *req*, *ctx*)

deadserver.handlers.xfer module

Handler for XFER requests: transfer a file chunk

`deadserver.handlers.xfer.`**`handle`**(*controller*, *req*, *ctx*)

Module contents

Collects request handlers for the various message types.

How to write a request handler:

Your module must define *function(controller_id, request_data) -> status, response_data*. This must be registered as a handler for a request type using the *@handles(deadserver.protocol.MsgType)* decorator. See below for note on importing.

Hello world example:

‘‘‘python from deadserver.protocol import MsgType, ResponseStatus

@handles(deadserver.protocol.MsgType.HELLO) # actually, this doesn’t exist, but if it did... def handle_hello(controller_id, data):

return ResponseStatus.OK, b’Hello ‘ + controller_id + b’! You sent: ‘ + data

‘‘‘

See *./open.py* for a real-world example.

In order to be executed (and therefore registered), your handler module must be imported somewhere. This file is a good place for that, as it is imported by *deadserver.api*. Unless you have a reason to do this differently, add your handlers below.

5.7.2 Submodules

5.7.3 deadserver.messages module

The controller server protocol message handler.

Reads messages and passes them to the appropriate handlers defined in *handlers/*.py*.

class deadserver.messages.**Context**(*config*, *db*)
Bases: object

class deadserver.messages.**MessageHandler**(*ctx*)
Bases: object

handle(*in_buf*)

pass_to_handlers(*controller*, *msg_type*, *indata*)

5.7.4 deadserver.server module

The UDP server that handles controller messages.

Wraps *messages.MessageHandler* by hanging it onto a socket.

deadserver.server.**serve**(*config*)

5.7.5 Module contents

The Deadlock server – communicates with controllers.

5.8 integration_test module

5.9 server

5.9.1 runapi module

5.9.2 runaux module

5.9.3 runsrv module

c

d

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

V

Y