

# services -- Writing PigeonDeliver 2.0 Services

**Carlo Contavalli**

**ccontavalli at masobit.net**

## **Revision History**

Revision 1.0.0 2004/08/22  
First Document Revision

This document describes how to write PigeonDeliver 2.0 Services. Its main purpose is to introduce programmers and users to the structure of PigeonDeliver Services. This document will be modified frequently, to reflect changes in the PigeonDeliver API. Each version of this document will refer to a particular family of PigeonDeliver releases that will be clearly declared at the beginning of the document. This version of the document refer to any known PigeonDeliver version.

## **1. Before starting**

This document was written as part of the documentation of the PigeonAir Project to provide help and support to users, system administrators or developers.

While every effort has been made to ensure that the information is accurate at the time of publication, this document may contain errors, omissions, incongruences or wrong technical details. No liability for damages is accepted by the Author, the publishers or any other organization or person providing the information, arising from any errors or omissions that may appear, however caused.

In case you find an error, you would like to propose better solutions than those discussed in this document or you would like to discuss an idea regarding this document or its content, we would be glad to hear from you and please feel free to contact us by writing to the <deliver-dev at ml.pigeonair.net> mailing list or by directly contacting one of the authors.

## **1.1. Intended Audience**

This document is meant to introduce developers to the structure of PigeonDeliver services and to provide all the information that may be needed to write a PigeonDeliver service.

This document assumes you are already familiar with the PigeonDeliver inner workings, installation and configuration, and no introductory details will be provided. The reader should thus have read all the introductory documents available on the internet site.

## **1.2. Copyright Notice**

This document was written by Carlo Contavalli <ccontavalli at masobit.net> and is thus Copyright (C) Carlo Contavalli 2003, 2004 and the PigeonAir Project.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts and no Back-Cover Texts.

Any example of program code available in this document should be considered protected by the terms of the GNU General Public License.

You should have received a copy of the GNU General Public License along with this document; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Trademarks are owned by their respective owners.

## **2. What is a PigeonDeliver Service?**

A PigeonDeliver Service is a module able to handle part of the delivery process of a mail.

To know more about how the PigeonDeliver handles emails delivery or how Services are used by PigeonDeliver, please refer to %TODO%.

A PigeonDeliver Service is a C written library/program conforming to the structure and rules described in the following sections of this document providing the described methods or callbacks.

PigeonDeliver Services are called and used by PigeonDeliver SAPI and PigeonDeliver IPC, whose role is to provide a layer of abstraction over a particular mail server or technology for mail delivery.

All PigeonDeliver SAPI and IPC must and do behave as described in the following few sections, where they will simply be referred as “callers” of the PigeonDeliver Service.

To know more about how to port PigeonDeliver to different mail servers or on how to write PigeonDeliver SAPI or IPC libraries, please refer to %TODO%.

### 3. Service provided methods

A PigeonDeliver Service must be written as closely as possible to the specification described on this document.

In particular the caller of the Service will expect it to provide some methods that will be used by the caller to perform the delivery of the email.

Some of the methods are mandatory, while others are not. A mandatory method *must* be provided by a Service which will be considered broken/incompatible/not compliant if not.

Here is the list of methods a Service may/should provide. Mandatory methods will be indicated in the description of the method itself

#### load

Called when the Service is first loaded. The caller must call this method at least and at most once per process. The caller must also guarantee that in a multithreaded environment, the caller will make use of the `load` method before spawning threads or performing the required locking.

It should be used by the Service to fetch configuration parameters from the configuration file, initialize static data structures or to prepare other Services to be used.

#### unload

Called to cleanup the resources allocated by each call to the `load` method. The `unload` method is thus called as many times as the `load` method.

It should be used to free up memory used to hold configuration variables, close database connections or free up global static structures initialized by the `load` method.

After the `unload` method is called, all the resources allocated by the Service must have been freed or returned back to the operating system.

Note also that the caller is allowed not to call the `unload` method if the `load` method returned a NULL descriptor to hold initialized data, since it is allowed to assume that no initialization was performed at all.

#### `init`

Called once per caller thread (if the thread spawns threads) or anyway to initialize one of possibly multiple parallel execution of this same Service.

It should be used to initialize thread or session specific variables, like database connections or memory used by the following methods.

#### `done`

Called once for each call to the `init` method, in order to free up resources allocated by the `init` method itself.

After each `done` call, all memory resources allocated by the corresponding `init` method call should have been freed.

Note also that the caller is allowed not to call the `done` method if the `init` method returned a NULL descriptor to hold initialized data, since it is allowed to assume that no initialization was performed at all.

#### `del_init`

Called once at the beginning of the delivery of a single mail.

Should be used to perform checks over the email (to test, for example, if the module is interested in delivering the email), or to prepare resources that will be used during the delivery itself.

#### `del_perform`

*mandatory* Called once per each recipient of a given mail.

Should be used to deliver the email itself. This method is often referred in other documents as the “delivery” method.

#### `del_done`

Called once after the completion of the delivery of a mail that was previously initialized calling the `del_init` method.

Should be used to free up resources allocated by the `del_init` method.

Please read the following sections for more detailed information about the described methods.

## 4. Expected execution paths

Although callers can make use of Services at their wishes as long as they respect the provided specifications, all the methods were thought to be used by callers following particular execution paths that will be described in the following few sections.

Note that:

- error handling has been completely omitted in all examples, since it is discussed in specific sections %TODO%.
- pseudo code is written using something that may look like perl or C, which I hope to be clear to the reader. Variables like `ld1`, `dd1`, `id1` are considered to have already been initialized, while variables like `&ld1` or `&dd1` are made available to the Service to initialize them.
- the reader should not worry about data structures like `load_data`, `init_data` or `del_data` whose type, kind and usage will be explained in the next few sections. Note that they are just placeholders used to make the example clearer. *There is no structure named `load_data`, `init_data` or `del_data` used by PigeonDeliver!!*
- a regression/compliance testing suite for modules is currently being written. All the execution paths described below will thus be tested.
- Services must behave correctly under any of the described execution paths.

### 4.1. Single thread

Using this execution path, the caller will call the `load` method once, exactly as the `init` method (and the corresponding `unload` and `done` methods). The caller will loop over the emails to deliver, calling on each loop the `del_init`, `del_perform` and `del_done` methods, or performing many deliveries at the same time.

Using (a rather ugly and incomprehensible) pseudo code, the first case of delivery path may look something like:

```
caller() {  
    load_data ld1;  
    init_data id1;
```

```
del_data dd;

load(&ld1);
init(ld1, &id1);

foreach email in (emails_to_deliver) {
    del_init(ld1, id1, &ddl);
    foreach recipient in (recipients_of_email(email)) {
        del_perform(ld1, id1, ddl, recipient);
    }
    del_done(ld1, id1, ddl);
}
}
```

If deliveries are performed parallely, the delivery path may look something like:

```
caller() {
    load_data ld1;
    init_data id1;
    del_data dd[];
    int i;

    load(&ld1);
    init(ld1, &id1);

    for(i=0; email=next_email(emails_to_deliver); i++)
        del_init(ld1, id1, &dd[i]);

    go_to_firstemail(emails_to_deliver);
    for(i=0; email=next_email(emails_to_deliver); i++) {
        foreach recipient in (recipients_of_email(email)) {
            del_perform(ld1, id1, dd[i], recipient);
        }
    }

    go_to_firstemail(emails_to_deliver);
    for(i=0; email=next_email(emails_to_deliver); i++)
        del_done(ld1, id1, dd[i]);

    done(ld1, id1);
    unload(ld1);
}
```

Note that the caller is allowed to also perform multiple deliveries at a time by using something like:

```
caller() {
    load_data ld1;
    init_data id1;
    init_data id2;
```

```
del_data dd1;
del_data dd2;
int i;

load(&ld1);
init(ld1, &id1);
init(ld1, &id2);

foreach email in (emails_to_deliver) {
    del_init(ld1, id1, &dd1);
    del_init(ld1, id2, &dd2);

    foreach recipient in (recipients_of_email(email)) {
        del_perform(ld1, id1, dd1, recipient);
        del_perform(ld1, id2, dd2, recipient);
    }
    del_done(ld1, id1, dd1);
    del_done(ld1, id1, dd2);
}

done(ld1, id1);
done(ld1, id2);
unload(ld1);
}
```

or a mix of the two.

## 4.2. Multiple threads

Using this execution path, the caller will call the load method exactly once, before spawning any thread, or by performing the correct locking.

Each thread will then call the `init` method to obtain a descriptor usable only in the given thread. In short, load descriptors are allowed to be shared among threads, while `init` descriptors are not.

Here is how the delivery path may look like without using locks:

```
thread(thread_data th1, load_data ld) {
    init_data id;
    del_data dd;

    init(ld, &id);

    while(still_want_to_process_emails) {
        wait_for_emails(&emails_to_deliver);

        foreach email in (emails_to_deliver) {
            del_init(ld, id, &dd);
```

```
        foreach recipient in (recipients_of_email(email))
            del_perform(ld, id, dd, recipient);
        del_done(ld, id, dd);
    }
}

done(ld, id);
}

caller() {
    load_data ld1;
    thread_data th1;
    thread_data th2;
    thread_data th3;

    load(&ld1);

    spawn_thread(&th1, thread, ld1);
    spawn_thread(&th2, thread, ld1);
    spawn_thread(&th3, thread, ld1);

    wait_threads(th1, th2, th3);
    unload(ld1);
}
```

where the caller spawns three threads to handle requests.

Using locks, it may look something like:

```
load_data ld;
thread_lock lock;

thread(thread_data th1) {
    init_data id;
    del_data dd;

    if(!load_data_initialized(ld)) {
        lock(&lock);
        if(!load_data_initialized(ld))
            load(&ld);
        unlock(&lock);
    }

    /* Here, we are pretty sure ld has
     * been initialized */
    init(ld, &id);

    while(still_want_to_process_emails) {
        wait_for_emails(&emails_to_deliver);

        foreach email in (emails_to_deliver) {
```

```

        del_init(ld, id, &dd);
        foreach recipient in (recipients_of_email(email))
            del_perform(ld, id, dd, recipient);
        del_done(ld, id, dd);
    }
}

done(ld, id);
}

caller() {
    thread_data th1;
    thread_data th2;
    thread_data th3;

    spawn_thread(&th1, thread, ld1);
    spawn_thread(&th2, thread, ld1);
    spawn_thread(&th3, thread, ld1);

    wait_threads(th1, th2, th3);

    if(load_data_initialized(ld1))
        unload(ld1);
}

```

Note that the caller should ensure, whenever possible, that the `load` method is called exactly once, since it may complete CPU intensive tasks.

Combinations with threads and the execution paths shown in the previous section are also allowed.

## 5. Error handling

Most methods are allowed to return an error. Callers will handle errors in a method/caller specific way, but always obeying some rules. To make sure everything will work correctly, the Service methods must behave in a well defined way:

- if the `load`, `init`, `del_init` methods encounter an error, they must return an error and free up any resource they may have allocated, like if they were never called, and clean up global variables they may have initialized. In case of the `del_init` method, the email must be left untouched.

The `del_init` method must thus be very careful in handling the stream, paying attention not to leave it in an undefined state (it should either complete to use it or not use it at all).

It is probably best practice to leave stream usage to the delivery method (`del_perform`).

- the `unload` and `done` method should not return any error. They must, in any case, free up all the resources allocated by the corresponding counterpart. They are however allowed to output warnings or return error strings, which may or may not be used by the caller. The impossibility to free up resources, will probably cause the caller to exit.
- the `del_done` method can return an error, in which case the delivery of the whole mail will be considered *failed*. The `del_done` method must in any case free up all the resources allocated by the `del_init` method.
- the `del_perform` method may return any error status, and the caller will behave as necessary to respond to the status. Mainly, the `del_perform` can return a value indicating an error in the delivery process (for example, need to bounce the message) or an internal error. In any case, all the resources allocated by the method must be freed, in order to avoid memory/resource leaks.

## 6. Starting to write a PigeonDeliver Service

Before starting, keep in mind that many of the “mandatory” indications contained in this section are actually “suggestions”. However, if you decide not to obey some or any of the suggested rules, and doing things the way you like, you are on your own if you ever have problems, and you make life harder for those who want to help or work on your code.

If you find there is something which is hard to do following the specifications of the described model, please don't look for ways to work around/hack the imposed limits. If the model has limits, then probably the model needs to be changed, and I'm always glad to expand it/make it easier to use. Just take your time to discuss it on one of the mailing lists. Maybe there's a simpler way to go you didn't think about...

First of all, you need to create a directory where to put your own Service and a `.c` file with the name of your own module.

For example, if you want to write a module called “mailToMars”, you need to create the directory `mailToMars` with the command `mkdir mailToMars`, `cd` into it with something like `cd mailToMars` and finally create a file `mailToMars.c`.

Open `mailToMars.c` and then start it by typing:

```
#include <dscm0/dscm.h>
#include <dscm0/dscm-service.h>
```

in order to include the correct header files.

Now, you need to define the main structure that will be used by the PigeonDeliver interface to call your module by declaring the name of your Service and the methods to be called, using something like:

```
DSCM_SERVICE(dscm_mailToMars) {
    DSCM_NAME(dscm_mailToMars),
    /* load */      NULL,
    /* unload */    NULL,
    /* del_init */  NULL,
    /* del_perform */ NULL,
    /* del_done */  NULL,
    /* init */      NULL,
    /* done */      NULL
};
```

Note that it is a good idea:

- to write mailToMars exactly the same way twice as shown above.
- to always define a `del_perform` method, otherwise your module will be discarded as invalid (`del_perform` is mandatory).
- to leave the comments where they are, in order to make life easier for those reading the code for the first time.
- keep all pointers NULL until you write the fully compliant function able to handle the method.
- to always leave this structure at the end of the file. Rationale: this structure contains many pointer to functions. Those functions should be declared static, as explained in the next few sections, kept private and their declaration never put in any `.h` files (yes, it works anyway, but you better keep things clean anyway). If you always keep this structure *at the end* of your file, you won't run into troubles with your compiler. Additionally, as being a common habit to leave it at the end of the file, you make life easier for other programmers.

You can now start writing your own methods to handle requests. In the following sections, the prototypes of the functions and their expected behavior is described. To know what they should do or why they should be defined, please read the section Service provided methods.

## 7. About Methods

If you need a description on how methods are called and how methods are supposed to handle errors, please read the first few sections of this same document.

In this section we will go into details about writing each of the methods a Service should/may provide.

Please keep in mind that only few functions of the PigeonDeliver API will be discussed in the following sections. For a full list of functions, please take a look at %TODO%.

Each of the following sections information about how to declare the given method, the parameters that the caller will take care to pass and the return value it will be allowed to return.

Functions have all been declared as if they were part of the mailToMars Service. Please replace mailToMars with the name of your own module.

Error statuses, or the type `dscm_mstatus_srv_e`, will be discussed further in their own section %TODO%.

For a complete example of a module, please look into the sources directory, `dscm/services`. A good example may be the mailForward or mailAntivirus module.

## 7.1. load method

```
static dscm_mstatus_srv_e dscm_mailToMars_load(dscm_state_t *state, void
**ld);
```

- `state` contains the state parameter used by many of the PigeonDeliver API functions.
- `ld` allows the function to return a custom descriptor, the `load_data` described in the examples in the previous sections. This descriptor is then passed over to all the other Service methods.

In case an error verifies, the function must return an error and cleanup and reset all the allocated resources, like if it was never called. It is acceptable for the caller to try to call the `load` method again in case of failure.

The `load` method can assume it will *never be* called with a NULL `state` or `ld` parameter, and should generate an assertion failed error in case it is.

The `load` method should always make use of the `ld` parameter. If it is left NULL, the caller can assume: that the load method was never called and the `ld` parameter still needs initialization or that there is no need to call other Service methods. In the rare case you don't need it, please set it to a non-zero value anyway.

## 7.2. unload method

```
static dscm_mstatus_srv_e dscm_mailToMars_unload(void **ld);
```

- `ld` a pointer to the data initialized by the `load` method (in the examples, referred as `load_data`).

Although allowed, it is unacceptable for this function to fail and return an error. In case it does, however, the caller will probably die, since it couldn't free up important resources.

The `unload` method can assume it will *never be* called with a NULL `ld` parameter, and should generate an assertion failed error in case it is.

### 7.3. init method

```
static dscm_mstatus_srv_e dscm_mailToMars_init(void *ld, void **id,  
dscm_secret_t *secret);
```

- `ld` a pointer to the data initialized by the `load` method (in the examples, referred as `load_data`).
- `id` allows the function to return a custom descriptor, the `init_data` described in the examples in the previous sections. This descriptor is then passed over to all the other Service methods.
- `secret` an authentication descriptor used by many PigeonDeliver API to perform authentication. This parameter may disappear in future releases of PigeonDeliver.

In case an error verifies, the function must return an error and cleanup and reset all the allocated resources, like if it was never called. It is acceptable for the caller to try to call the `init` method again in case of failure.

The `init` method can assume it will *never be* called with a NULL `ld`, `id` or `secret` parameter, and should generate an assertion failed error in case it is.

### 7.4. done method

```
static dscm_mstatus_srv_e dscm_mailToMars_done(void *ld, void *id);
```

- `ld` a pointer to the data initialized by the `load` method (in the examples, referred as `load_data`).
- `id` a pointer to the data initialized by the `init` method (in the examples, referred as `init_data`).

Although allowed, it is unacceptable for this function to fail and return an error. In case it does, however, the caller will probably die, since it couldn't free up important resources.

The `done` method can assume it will *never be* called with a NULL `ld` or `id` parameter, and should generate an assertion failed error in case it is.

## 7.5. del\_init method

```
static dscm_mstatus_srv_e dscm_mailToMars_del_init(void *ld, void *id, void
**dd, dscm_request_t *req, dscm_stream_t *stream);
```

- *ld* a pointer to the data initialized by the `load` method (in the examples, referred as `load_data`).
- *id* a pointer to the data initialized by the `init` method (in the examples, referred as `init_data`).
- *dd* allows the function to return a custom descriptor, the `del_data` described in the examples in the previous sections. This descriptor is then passed over to all the other Service methods.
- *req* a structure representing the delivery request. Used by other PigeonDeliver API to gather information about the current delivery. Although the API is not complete yet, this parameter should never be used directly by the Service.
- *stream* a structure representing the control stream used by other PigeonDeliver API. This parameter will disappear in future version of PigeonDeliver.

In case an error verifies, the function must return an error and cleanup and reset all the allocated resources, like if it was never called. It is acceptable for the caller to try to call the `del_init` method again in case of failure.

The `del_init` method can assume it will *never be* called with a NULL *ld*, *id*, *dd*, *req* or *stream* parameter, and should generate an assertion failed error in case it is.

## 7.6. del\_done method

```
static dscm_mstatus_srv_e dscm_mailToMars_del_done(void *ld, void *id, void
**dd);
```

- *ld* a pointer to the data initialized by the `load` method (in the examples, referred as `load_data`).
- *id* a pointer to the data initialized by the `init` method (in the examples, referred as `init_data`).
- *dd* a pointer to the data initialized by the `del_init` method (in the examples, referred as `del_data`).

In case this function returns an error, the caller may assume that all the resources were freed correctly but that the email could not be delivered.

The `del_done` method can assume it will *never be* called with a NULL *ld*, *id* or *dd* parameter, and should generate an assertion failed error in case it is.

## 7.7. del\_perform method

```
static dscm_mstatus_srv_e dscm_mailToMars_del_done(void *ld, void *id, void
*dd, dscm_recipient_t *rcpt, dscm_attr_t *db_attrs, int *mail_status,
dscm_inject_t **inject);
```

- *ld* a pointer to the data initialized by the `load` method (in the examples, referred as `load_data`).
- *id* a pointer to the data initialized by the `init` method (in the examples, referred as `init_data`).
- *dd* a pointer to the data initialized by the `del_init` method (in the examples, referred as `del_data`).
- *rcpt* a pointer to a structure representing the current recipient of a mail.
- *db\_attrs* a pointer to a structure representing the configuration of the recipient the caller would like to be applied to the delivery.
- *mail\_status* while the return value of the function represents if the function itself succeeded or failed, this status indicates what sort the mail should follow once the function returns.
- *inject* a pointer to a structure used by other PigeonDeliver API in order to allow a Service to change a mail being delivered (inject a mail in the queue).

The `del_perform` method can assume it will *never be* called with a NULL *ld*, *id*, *dd*, *rcpt*, *mail\_status* or *inject* parameter, and should generate an assertion failed error in case it is. Note that *db\_attrs* is thus allowed to be NULL.

## 8. Error statuses

Ok, any of the above methods can return a status. Statuses are defined in `dscm0/dscm-service.h`:

```
typedef enum dscm_mstatus_srv_e {
    DSCM_STANDARD_STATUS(Srv),
    dscmSrvOk,
    dscmSrvError,
    dscmSrvConfig,

    dscmSrvBounce,
    dscmSrvDefer,
    dscmSrvSkip
} dscm_mstatus_srv_e;
```

Ok, *dscmSrvOk* should always returned, unless something bad happens. `init`, `done` and `del_init` are allowed to return any of the values *dscmSrvOk*, *dscmSrvError* (to indicate a generic error), *dscmSrvConfig* (to indicate a configuration validation error). Additionally, the `del_init` can return

*dscmSrvSkip* to indicate that *del\_done* and *del\_perform* should not be called for this mail. *del\_perform* can additionally return *dscmSrvDefer*, to defer a message (not supported yet, will be in future releases) or *dscmSrvBounce* to bounce it. *del\_done* is allowed to return *dscmSrvOk* or *dscmSrvError*, in which case the delivery is considered failed, while *done* and *unload* should always return *dscmSrvOk*, while *dscmSrvError* and any other value may cause the caller to die.

## 9. PigeonDeliver API for Services

Although Services may make use of any public PigeonDeliver function, in this section we will try to discuss generic issues arising while writing PigeonDeliver Services and how they can be solved and of which functions to make use of.

For a full reference of all API functions, please refer to %TODO%.

### 9.1. Accessing user configurations

### 9.2. Outputting errors or warnings

### 9.3. Reading the email being delivered

### 9.4. Changing the email being delivered

### 9.5. Forwarding a mail

### 9.6. Generating a mail

## **9.7. Modifying the delivery process**