Remote Execution Daemon (RED)
A Simple Service for Remote Execution and Remote Storage
Version 1.0

Georges-André Silber
CRI/ENSMP Technical Report E-267
April 25, 2005

Contents

1 Introduction 1

2 RED interface 2

2.1 Worlds 3

2.1.1 Method red_hello 3

2.1.2 Method red_world 4

2.1.3 Method red_properties 5

2.1.4 Method red_destroy 5

2.2 Files 6

2.2.1 Method red_list 6

2.2.2 Method red_directory 6

2.2.3 Method red_perm 7

2.2.4 Method red_remove 7

2.3 Jobs 8

2.3.1 Method red_sink 8

2.3.2 Method red_source 9

2.3.3 Method red_run 10

2.3.4 Method red_kill 11

2.3.5 Method red_jobs 11

3 Conclusion 11

1 Introduction

In this document, we describe the interface of RED (Remote Execution Daemon), a simple service for remote file storage and remote program execution. The implementation of this service can be a TCP/IP server that waits for connections on a given port. When the server receives a message containing a method call it returns immediately a message containing the method results. Many methods of a RED trigger actions that can run asynchronously on the host. The main methods that act this way are the methods red_run (section 2.3.3), red_sink (section 2.3.1), and red_source (section 2.3.2). The method red_jobs (section 2.3.5) can then be used afterward to get informations about the end of the action.
RED can be seen as very simple “grid” operating system model. The only way to interact with it is by its network service interface. To use this operating system, clients must connect to the service interface and request a workspace that we call a *world*. Worlds are identified by a key that must be used by a client to operate on a world. A world is mainly a private virtual file system to store programs and data, but it is also a container for running jobs. Operations on a world are performed via the service interface and can be grouped in three classes: 1) configuration and information about the world or the RED (section 2.1), 2) operations to transfer files to and from the world (section 2.2), and 3) creation and management of jobs (section 2.3).

When a RED is launched, it has only one world created by default. This world can be used by every client and can be seen as a public world. By default, when a new world is created on a RED by a client, it is almost empty: it only contains some directories, special files, and required dynamic libraries. The list of files present is dependent of the implementation. The method `red_list` (section 2.2.1) can be used to get the list of files that are present on a newly created world.

The emptiness of a world implies that prior to the run of a program into a world, the executable code and the files it requires must be transfered. The end of a world occured when the RED stops or when the user requests its destruction. All files contained in a world are lost when the world ends. By default, a RED does not provide any persistent storage of data. It loses all data when it is turned off and all keys created during the life of a RED are lost and cannot be reused on another RED. A RED has an underlying software and hardware architecture: it uses a particular kind of processor and a particular kind of kernel.

## 2 RED interface

We describe this interface with the hypothesis that this server runs over an existing Unix kernel (Linux, BSD, Mach, Solaris, etc.). Thus, all the notions used must be understood in the “natural” way, i.e. like the corresponding Unix notions.

File names appearing in the following methods can be absolute (beginning with ’/’) or relative, i.e. beginning with a file, a ’.’ (current directory) or a ’..’ (parent directory).

We use a C syntax to describe the methods, their parameters, and the message they returns. The system data types (like `time_t`) are from a Linux system. Correspondance with other systems should be easy.

A structure `fault` is contained in the message returned by all the methods of RED interface:

```c
struct red_fault {
    int red_errno;
    char *red_msg;
    char *red_extras;
} red_fault;
```

where `red_errno` contains the number of the error and `red_msg` a string describing the error. The string `red_extras` can contain a message that precise the error. The possible values of `red_errno` are described for each method. Two values are common to all methods and can be returned by all methods:

**RED\_ENONE** No error.

**RED\_EUNKNOWN** Unknown error. Usually, the field `red_msg` and `red_extras` gives a more precise description of the error. This error is a “fallover” for all errors that are not taken into account in this specification.
In the following, the use of ‘*’ in the type definition of a variable means that this variable is an array ended with the NULL element.

2.1 Worlds

2.1.1 Method red_hello

The structure red_key contains a key that uniquely identifies a RED on a given network. A key contains the address of a RED, the number of its service port, and the time when the RED has been started. We do the hypothesis that two RED cannot be created at the same time on a host. The combination of those three elements uniquely identifies a RED on a given network.

```
struct red_key {
    struct timeval red_start; /* time when RED started */
    struct red_addr red_addr; /* address of the RED */
};
```

```
typedef enum { RED_AF_INET, RED_AF_INET6 } red_addrtype;
struct red_addr {
    char *red_hostname; /* name of RED host */
    char *red_address; /* IP address of RED host */
    int red_length; /* length of address */
    red_addrtype red_addrtype; /* RED_AF_INET or RED_AF_INET6 */
    u_int16_t red_port; /* service port number */
};
```

The method red_hello can be used to detect RED servers on a network, and select some of them based on their processors and kernel types. The main purpose of this method is to get a red_key to identify a RED and to use more elaborate methods. This method is the only one that can be called without a red_key.

```
struct red_hello_response {
    struct red_fault fault; /* error status */
    char red[4]; /* contains 'R','E','D',0 */
    struct red_key key; /* RED key */
    struct red_world_key dworld; /* Default world */
    char *machine; /* result of 'uname -m' */
    char *kernel_name; /* result of 'uname -s' */
    char *kernel_release; /* result of 'uname -r' */
    char *kernel_version; /* result of 'uname -v' */
    char **properties; /* list of available properties */
};
```

```
struct red_hello_response
red_hello ();
```

The field properties contains a list of the available properties for this RED running on this particular host. This is an array of chains of characters ended with the null chain (NULL). A chain describing a property must have the following structure:

PROPERTYNAME-property description
i.e., a property name without spaces and characters '-', and a property description. For instance, 

CPUFREQUENCY-Frequency of the CPU in GHz

is a valid property description.

2.1.2 Method red_world

The structure red_world_key uniquely identifies a world on a RED and is called a world key. We do the hypothesis that two worlds cannot be created at the same time with the same name on a RED. A combination of a red key and a world key uniquely identifies a world over all running RED.

struct red_world_key {
    struct timeval world_creation; /* time of world’s creation */
    char *world_name; /* name given to the world by the RED */
};

To call methods of a RED, a world must be provided. The method red_world is used to create a world in a RED and get a key to enter this world. When a RED is launched, all worlds are invalid except the default world. The name of this world is implementation dependent.

A world key opens the access to a private directory that is the root '/' of the file system. All jobs and files created in a world cannot be seen by other worlds. A world can only be accessed with the world key.

struct red_world_response {
    struct red_fault fault; /* error status */
    struct red_world key; /* world key */
};

struct red_world_response
red_world (struct red_key rkey /* RED key */);

The field fault.red_errno can take the following values:

RED_EBADKEY The key used to access this RED is not valid.

RED_ENOMOREWORLD A RED can create a limited number of worlds. This error is returned when a RED has created all possible worlds.

Note to implementers. A world is a private directory in the main file system of the system running a RED, and a specific Unix user owner of this world/filesystem. The more obvious way to implement a world is with a RED running under super user identity, and that goes to a world with 'chroot(2)', 'chdir(2)', and 'setuid(2)' calls. Note that only methods that actually create jobs has to actually go to a world. Those methods are red_run, red_sink, and red_source.
2.1.3 Method red_properties

The structure `red_property` is a container for a property asked to or received from a RED.

```c
struct red_property {
    char *property_tag; /* Tag identifying the property */
    char *property_content; /* Content of the property */
};
```

When a property is asked, the field `property_content` is empty. The same structure is returned with the field `property_content` filled if the property is available on the RED.

The method `red.properties` returns information about a RED. The list of available properties is returned by the `red.world` method.

```c
struct red_properties_response {
    struct red_fault fault; /* error status */
    struct red_property *properties; /* values of the properties */
};
```

```c
struct red_properties_response
red_properties (struct red_key rkey, /* RED key */
               struct red_world_key wkey, /* World key */
               struct red_property *properties
               /* Asked properties */);
```

The field `fault.red_errno` can take the following values:

- `RED_EBADKEY` The key used to access this RED is not valid.
- `RED_EBADWORLD` The world does not exist on this RED.
- `RED_EBADPROPERTY` A property asked is not available on this RED. This property is returned in `fault.red.extras`.

**Note to implementers.** The list of properties available for a RED is dependent of the implementation.

2.1.4 Method red_destroy

This method destroys a world and inhibits its key. All running jobs of this world are destroyed and so is the file system of the world. Note that this operation is not recoverable.

```c
struct red_destroy_response {
    struct red_fault fault; /* error status */
};
```

```c
struct red_destroy_response
red_destroy (struct red_key rkey, /* RED key */
             struct red_world_key wkey /* World key */);
```

The field `fault.red_errno` can take the following values:

- `RED_EBADKEY` The key used to access this RED is not valid.
- `RED_EBADWORLD` The world does not exist on this RED.
Note to implementers. This operation does not need to be done under a world identity. It mainly consists to completely delete a file system and to destroy all the processes running under a world’s identity.

2.2 Files

2.2.1 Method red_list

The method red_list returns informations files. If filename is not a directory, it returns informations about this file. If filename is a directory, it returns informations about the directory (the first element of the field files in the returned structure red_list_response) and a non recursive list of informations about the files it contains.

typedef enum { RED_FDIR, RED_FFILE, RED_FSLINK, RED_FDEV } red_file_type;

struct red_file {
    char *filename; /* Filename */
    red_file_type type; /* Type of the file */
    struct red_mode mode; /* Permissions */
    off_t size; /* Size of the file */
};

struct red_list_response {
    struct red_fault fault;
    struct red_file *files;
};

struct red_list_response
red_list (struct red_key rkey, /* RED key */
        struct red_world_key wkey, /* World key */
        const char *dir /* Directory */);

The field fault.red_errno can take the following values:

RED_EBADKEY The key used to access this RED is not valid.

RED_EBADWORLD The world does not exist on this RED.

RED_ENOENT A component of the path filename does not exist, or the path is an empty string.

2.2.2 Method red_directory

This method creates a directory.

struct red_directory_response {
    struct red_fault fault;
};

struct red_directory_response
red_directory (struct red_key rkey, /* RED key */
               struct red_world_key wkey, /* World key */
               const char *dir /* Directory */);
The field fault.red_errno can take the following values:

**RED_EBADKEY** The key used to access this RED is not valid.

**RED_EBADWORLD** The world does not exist on this RED.

**RED_ENOENT** A component of the path `dir` does not exist, or the path is an empty string.

### 2.2.3 Method `red_perm`

This method changes the permissions of a file.

```c
struct red_mode {
    char read; /* 0=don’t set, *=set */
    char write; /* 0=don’t set, *=set */
    char execute; /* 0=don’t set, *=set */
};

struct red_perm_response {
    struct red_fault fault;
};

struct red_perm_response red_perm (struct red_key rkey, /* RED key */
                                   struct red_world_key wkey, /* World key */
                                   const char *filename, /* Filename */
                                   struct red_mode mode /* Permissions */);
```

The field fault.red_errno can take the following values:

**RED_EBADKEY** The key used to access this RED is not valid.

**RED_EBADWORLD** The world does not exist on this RED.

**RED_ENOENT** A component of the path `filename` does not exist, or the path is an empty string.

### 2.2.4 Method `red_remove`

This method removes a file or an empty directory.

```c
struct red_remove_response {
    struct red_fault fault;
};

struct red_remove_response red_remove (struct red_key rkey, /* RED key */
                                        struct red_world_key wkey, /* World key */
                                        const char *filename /* Filename */);
```

The field fault.red_errno can take the following values:

**RED_EBADKEY** The key used to access this RED is not valid.
RED_EBADWORLD The world does not exist on this RED.

RED_ENOENT A component of the path filename does not exist, or the path is an empty string..

RED_ENOTEMPTY filename is a directory and is not empty.

2.3 Jobs

The following methods create jobs. All those methods two common parameters that are props and wait. The first one is used to tell RED the properties that are to be returned from the job. The second one decides whether the method has to respond immediately or after the end of the launched job.

The structure red_job_info contains informations about a job.

```c
struct red_job_info {
    int job_number; /* Number of the job */
    char *job_name; /* Job name */
    struct red_job_property *job_props; /* Job properties */
};
```

The structure red_job_property is a container for properties asked or received from a job.

```c
struct red_job_property {
    char *job_property_tag; /* Tag identifying the property */
    char *job_property_content; /* Content of the property */
};
```

When a property is asked, the field property_content is empty. The same structure is returned with the field property_content filled.

**Note to implementers.** The list of properties available for jobs on a RED is up to the implementer. It could be interesting to define a minimal set of properties existing on all platforms, like START_TIME, STOP_TIME, MEM_USED, etc.

2.3.1 Method red_sink

This method prepares a file “sink” on a RED. Informally, it prepares a server that waits for a TCP connection on a port and when this connection arrives, copies all received data into a file. The job really starts when a connection is made on the incoming port.

The parameter blocksize determines the number of bytes that are read from the socket before they are written in the file. A value 0 means that the red_sink method uses a default value (implementation dependent).

```c
struct red_sink {
    char *filename; /* New or existing local file */
    struct red_addr incoming; /* Connection to wait for data */
    int append; /* 0=create/erase, 1=append */
};
```

```c
struct red_sink_response {
    struct red_fault fault;
};
```
struct red_job_info job; /* Informations about the job */
;

struct red_sink_response
red_sink (struct red_key rkey, /* RED key */
    struct red_world_key wkey, /* World key */
    struct red_sink sink, /* Sink */
    struct red_job_property *props, /* Properties */
    unsigned int wait, /* Wait for job's end ? */
    unsigned int blocksize /* Buffer size */);

The field fault.red_errno can take the following values:

**RED EBADKEY** The key used to access this RED is not valid.

**RED EBADWORLD** The world does not exist on this RED.

**RED EBADSINK** The sink is invalid (bad filename, bad receiving port).

2.3.2 Method red_source

This method creates a file “source” on a RED. Informally, it creates a TCP client that establishes a TCP connection on a specific host/port and transfers in this connection the data of a file present on the RED.

The parameter blocksize determines the number of bytes that are read from the file before they are written to the socket. A value 0 means that the red_source method uses a default value (implementation dependent).

The host defined in the source parameter of the method must be available when this method is called, otherwise, a RED EBADSOURCE error is returned.

struct red_source {
    char *filename; /* existing local file */
    struct red_addr outgoing; /* Destination of file content */
};

struct red_source_response {
    struct red_fault fault;
    struct red_job_info job; /* Informations about launched job */
};

struct red_source_response
red_source (struct red_key rkey, /* RED key */
    struct red_world_key wkey, /* World key */
    struct red_source source, /* Source */
    struct red_job_property *props, /* Properties */
    unsigned int wait, /* Wait for job's end ? */
    unsigned int blocksize /* Buffer size */);

The field fault.red_errno can take the following values:

**RED EBADKEY** The key used to access this RED is not valid.
The world does not exist on this RED.

The source is invalid (bad filename, bad distant port).

### 2.3.3 Method `red_run`

This method prepares a program execution on a RED. Informally, it prepares a server that waits for a TCP connection on a port and when this connection arrives, it runs the given program, connecting the `stdin`, `stdout` and `stderr` streams of the running program to the specified hosts/ports or local files.

```c
struct red_command {
    char *command; /* Command name */
    char **argv; /* NULL terminated list of arguments*/
};

struct red_run {
    struct red_addr net_stdio; /* Port for incoming connection */
    char *file_stdio; /* Name of local file */
    /* (override net_stdio) */

    struct red_addr net_stdout; /* Port for outgoing connection */
    char *file_stdout; /* Name of local file */
    /* (override net_stdout) */
    int append_stdout; /* 0=create/erase, 1=append */

    struct red_addr net_stderr; /* Port for outgoing connection */
    char *file_stderr; /* Name of local file */
    /* (override net_stdout) */
    int append_stderr; /* 0=create/erase, 1=append */
};

struct red_run_response {
    struct red_fault fault;
    struct red_job_info job; /* Informations about launched job */
};

red_run (struct red_key rkey, /* RED key */
    struct red_world_key wkey, /* World key */
    struct red_command cmd, /* Command */
    struct red_run run, /* run parameters */
    struct red_job_property *props, /* Properties */
    unsigned int wait /* Wait for job's end ? */);
```

The field `fault.red_errno` can take the following values:

**RED_BADKEY** The key used to access this RED is not valid.

**RED_BADWORLD** The world does not exist on this RED.
The command is invalid.

The run parameters are invalid.

2.3.4 Method red_kill

This method sends a signal to a specified job. Valid signal numbers are the POSIX reliable signals (in Linux systems, they are called “standard signals”).

```c
struct red_kill_response {
    struct red_fault fault;
};
```

```c
struct red_kill_response
red_kill (struct red_key rkey, /* RED key */
    struct red_world_key wkey, /* World key */
    int job_number, /* Job number */
    int signal /* Signal to send */);
```

If `job_number` is positive, then signal `signal` is sent to `job_number`. If `job_number` is null or negative, `signal` is sent to every job of the world. The field `fault.red_errno` can take the following values:

- RED_EBADKEY The key used to access this RED is not valid.
- RED_EBADWORLD The world does not exist on this RED.
- RED_EBAD_SIGNAL The signal is invalid.
- RED_EBADJOB The job is invalid.

2.3.5 Method red_jobs

This method returns the list of running or zombie jobs. Zombie jobs are the finished jobs for whom no `red_job_info` structures have been reclaimed. Once the method `red_jobs` has returned a `red_job_info` structure for those jobs, they are not zombie anymore and are removed from the finished jobs table.

```c
struct red_jobs_response {
    struct red_fault fault;
    struct red_job_info *jobs;
};
```

```c
struct red_jobs_response
red_jobs (struct red_key rkey, /* RED key */
    struct red_world_key wkey /* World key */);
```

3 Conclusion

In this document, we describe the interface of RED, a simple service for remote file storage and remote program execution. This service can be easily implemented on top of an existing Linux kernel, with an XML-RPC over HTTP messaging system.