MASARYK UNIVERSITY
FACULTY OF INFORMATICS



Synchronization of services communicating over HTTP

DIPLOMA THESIS

Barbora Vaššová

Brno, Spring 2017

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Barbora Vaššová

Advisor: Mgr. Marek Grác, Ph.D.

Acknowledgement

I would like to thank Mgr. Marek Hulán for his immense help, insight and patience that helped me greatly during my work on this thesis, as well as I would like to thank my advisor Mgr. Marek Grác, Ph.D.

Abstract

The aim of the diploma thesis is to provide an open source proxy mediator between Foreman and Smart Proxy that would enable offline communication using HTTP protocol. The mediator is built on top of Sinatra framework with use of REST and HTTP. Included in this work is a simple WebUI for the mediator.

Keywords

Foreman, Smart Proxy, Sinatra, Ruby, HTTP, REST

Contents

1 **Introduction** . . . 1
2 **Environment** . . . 3
2.1 *Foreman* . . . 3
2.2 *Smart Proxy* . . . 4
3 **Mediator proxy components and design** . . . 6
3.1 *REST* . . . 6
3.2 *YAML* . . . 7
3.3 *JSON* . . . 8
3.4 *Sinatra* . . . 10
3.4.1 Templates . . . 12
3.5 *Design and characteristics of the mediator proxy* . . . 13
3.5.1 Introduction . . . 13
3.5.2 Data format . . . 15
3.5.3 Specifics . . . 16
3.5.3.1 Foreman's side of the mediator . . . 16
3.5.3.2 Smart proxy's side of the mediator . . . 16
3.5.3.3 Timeouts . . . 17
3.5.3.4 Delivering the replies . . . 18
3.5.3.5 Buffering . . . 18
3.5.3.6 Logging . . . 19
3.5.3.7 SSL and security . . . 19
3.5.3.8 UUID . . . 20
3.5.3.9 Preload of tasks . . . 21
4 **Workflow and implementation** . . . 22
4.1 *Creating a task* . . . 22
4.2 *Exporting the tasks* . . . 23
4.3 *Importing the tasks to Smart Proxy* . . . 24
4.4 *Importing the responses to mediator proxy* . . . 26
4.5 *Classes* . . . 27
4.5.1 Task . . . 27
4.5.2 TaskQueue . . . 28
4.5.3 TaskResponse . . . 28
4.5.4 TaskResponseQueue . . . 28
4.6 *Modules* . . . 29
4.6.1 Puppet . . . 29

4.6.2 Puppet CA . . . 29
4.6.3 DHCP . . . 29
4.6.4 DNS . . . 30
4.6.5 TFTP . . . 30
5 **WebUI** . . . 31
5.1 *List features* . . . 31
5.2 *Delete all tasks and responses* . . . 32
5.3 *Delete task by UUID* . . . 32
5.4 *Display current list of tasks* . . . 32
5.5 *Save current list of tasks* . . . 32
5.6 *Preload tasks* . . . 33
5.7 *Upload responses* . . . 33
5.8 *List available responses* . . . 34
6 **Installation and initial setup** . . . 35
6.1 *Requirements* . . . 35
6.2 *Foreman* . . . 35
6.3 *Smart Proxy* . . . 37
6.4 *Mediator proxy* . . . 38
7 **Conclusion** . . . 40
7.1 *Future work* . . . 41
A **Appendix** . . . 43
B **Appendix** . . . 44

1 Introduction

Growth and expansion is the aim of every company which also, in case of most companies, often comes with the need of more computational resources. These resources may be even necessary to be located in different places, with different hardware or software requirements. Such requirements cause a common issue, where at some point the amount of resources is growing so much that it is becoming unmanageable (or at least very inefficient) by a human administrator in person. That is the point where system management software is necessary to effectively manage the resources - remotely, automatically, in bulk. One of the examples of such software is Foreman, an open source project that helps system administrators manage servers throughout their lifecycle, from provisioning and configuration to orchestration and monitoring [1].

In order to better manage servers in various locations, there is an option to add Smart Proxy, a delegated server for every location. Smart Proxy provides a mean for the communication between location-specific services (like DHCP) and Foreman. These Smart Proxies must be online in order to be able to successfully communicate with Foreman, which might be a challenge in case a particular proxy is located in a place with limited internet access. An example could be a military submarine – an internet connection for computers located on a submarine might be unreliable – often down for extended periods of time or simply impossible to be provided at all. Another, more common example could be a large chain of supermarkets, where some branches might be located in mountains with very limited internet connectivity.

This work aims to provide a proof-of-concept solution for this issue, by creating a mediator proxy and additional scripts that would serve as a way for Foreman and Smart Proxy to communicate, while not being connected.

It is important to note that while this work uses Foreman and Smart Proxy as the examples of communicating entities, the solution this work provides is based on general principles and with small adjustments can work with different services.

In the first phase of the work I was supposed to set up an environ-

ment consisting of Foreman and one Smart Proxy. These were the entities for which I was supposed to provide a mean of communication without direct connection. Subsequently, I set up a mediator proxy, a web server based on the Sinatra framework that I would use as a base in order to simulate the Smart Proxy's behavior against Foreman. This modified web server would be able to act as a Smart Proxy and accept Foreman's commands (in the form of HTTP requests called 'routes'), but unlike the real Smart Proxy, it would not perform any action defined by routes, but store the commands in the form of tasks and export it into a YAML file – via either a script or its own simple WebUI. This phase was heavily focused on creating logical aspects and design of the communication via the mediator proxy.

The second phase consisted of creating a Ruby script that would import the tasks from a YAML file provided. It would use the tasks against the real Smart Proxy, which would perform the actions the same way as if they were commands from the Foreman and reply accordingly. This script would save the replies in a file. In this part some more functionalities concerning task management were added in the WebUI.

The third phase of the work was focused on communicating the replies back to Foreman. The mediator proxy was modified to be able to import the file with replies. Therefore, if admin requested a task to be performed against the Smart Proxy upon Foreman's command, the mediator proxy would first look whether it already has an answer for this task and communicate it to Foreman. It was necessary to focus on the task logic, in order to provide correct answers at the correct time.

2 Environment

2.1 Foreman

Foreman is an open source project, used as a tool for automated system management. It is able to manage a complete lifecycle of both physical and virtual servers (also called *hosts*). This includes provisioning, configuration, updates and package management and audits. In order to increase functionality, it is also possible to use various plugins with Foreman.

In the context of system management, the term *provisioning* means an installation of a new operating system on a selected host, which in this case is done remotely. Foreman is capable of provisioning hosts with various Linux-based operating systems, including (but not exclusive to) Red Hat Enterprise Linux, Fedora, CentOS, Ubuntu, Debian or OpenSUSE, both on bare metal or via virtualization or cloud providers (also called "compute resources"), such as OpenStack, Libvirt or VMware. Some providers (as well as physical hosts) are able to support unattended installation using PXE (Preboot Execution Environment), while others are image-based [2].

PXE-based provisioning works with Network Interface Card (NIC) of the system by making it function like a boot device. An administrator defines parameters of a new host in Foreman's WebUI and Foreman requests an unused IP address from DHCP server associated with the subnet and the new host request is submitted. DHCP server reserves an entry for the new host. Followingly, DNS records are created as well as PXELinux menu on a TFTP server. The host machine sends a DHCP request, and DHCP server responds with standard information (IP, subnet mask, gateway, DNS, etc.). In addition, it provides information about the location of a TFTP server and boot image. The host requests the bootloader and menu from the TFTP server. The installer requests a provisioning template from Foreman and uses it to install the operating system.

Above described procedure is one of the possible use cases for the mediator proxy, since it supports DHCP, TFTP and also DNS modules.

If using image-based provisioning, the host does not follow the

standard PXE boot and operating system installation. Instead, the compute resource creates a copy of the chosen image for the new host to use [3].

The configuration and content management part of the host lifecycle consists of installing new packages, updating them and also their configuration, as well as configuration of the host itself (for example its users or network interfaces). It is possible to create repositories, upload packages to them, create content views and selectively install packages on a chosen host or group of hosts as well as provide updates.

For content management purposes, the Katello plugin is necessary to be installed.

As for the audit and report functionalities, Foreman is able to periodically retrieve reports from hosts managed by Puppet, which is another system management and configuration tool. These reports can be accessed from Foreman's WebUI dashboard and system administrators can use them to assess state of the hosts, to view what actions were ran during various Puppet runs and the log messages it generated. Similarly, Foreman is able to work not only with Puppet, but also with Salt, Chef or Ansible.

In addition, Foreman supports the OpenSCAP plugin, which provides auditing of the hosts, with focus on security. It enables Foreman to receive automated vulnerability assessment and security compliance audits from the hosts. These reports are helpful for system administrators to find vulnerabilites in hosts and also suggest a remediation plan to fix them.

It is worth to note that Foreman is one of the base components of Red Hat Satellite 6, a system management product by Red Hat which allows to deploy and manage Red Hat Enterprise Linux hosts.

2.2 Smart Proxy

Smart Proxy is an autonomous web-based Foreman component that is placed on a host performing a specific function in the host commissioning phase, therefore it helps Foreman orchestrate the process of commissioning a new host. It receives requests from Foreman to perform operations that are required during the commissioning pro-

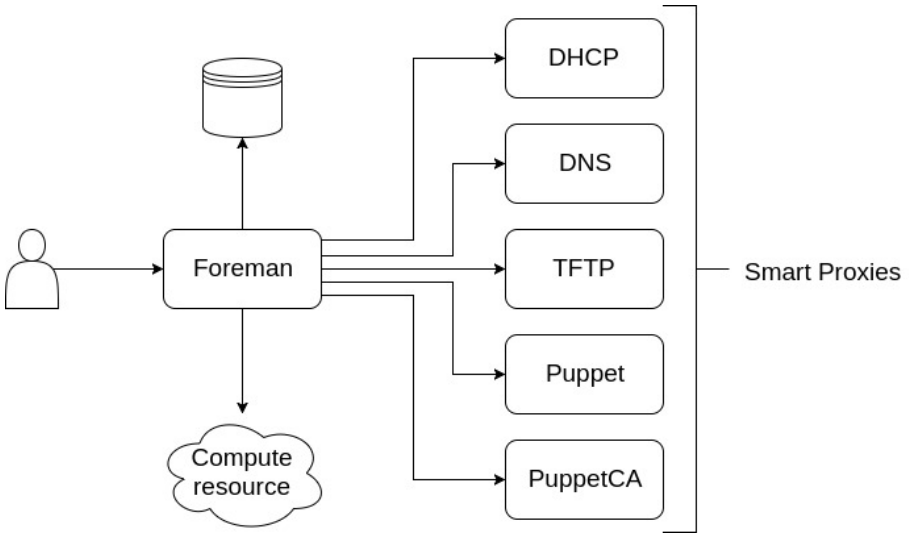


Figure 2.1: *Foreman architecture*

cess and executes them on its behalf [2]. The Smart Proxy manages remote services and is generally installed with all Foreman installations automatically.

Currently supported modules for Smart Proxy are:

1. DHCP
2. DNS
3. TFTP
4. Puppet
5. Puppet CA

If proxy has, for example, the TFTP module enabled, it can function as a TFTP server (which can be used in provisioning a new host). These modules can be installed on multiple separate Smart Proxies connected to Foreman. However, optionally, Smart Proxy can have multiple modules enabled, therefore one Smart Proxy instance can provide all necessities to provision a new host.

3 Mediator proxy components and design

3.1 REST

REST, also known as Representational State Transfer or RESTful Web Services is an architectural style for distributed hypermedia systems. It represents a way to provide interoperability between two entities (connectors) in a distributed environment. The primary connector types are client (web browser) and server.

The essential difference between the two is that a client initiates communication by making a request, while a server listens for connections and responds to requests in order to provide access to its services. Any component may include both client and server connectors.

The main principle is that the communication is stateless in nature, which means that each request from client to server must contain all of the information necessary for proper understanding of the request and also it must be independent of any requests that may have preceded it [4]. It cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.

Moreover, client has no knowledge about the server and the resources it hosts, while both client and server must agree beforehand on the media used for the communication (usually HTML).

This constraint induces the properties of visibility, reliability, and scalability.

Visibility is improved because the server does not have to obtain knowledge beyond a single request record in order to determine the full nature of the request. Reliability is improved since it makes the task of recovering from failures easier. Scalability is improved, since it is not necessary to store state between requests, the server component is able to quickly free resources. This also further simplifies implementation because the server does not have to manage resource usage across requests.

Like most architectural choices, the statelessness is a design tradeoff. Amongst its disadvantages is that it may decrease the performance of the network due to increase of the repetitive data sent in a series of requests (since that data cannot be left on the server in

a shared context). Moreover, keeping the application state on the client-side reduces the server's control over consistent application behavior, since the application becomes dependent on the correct implementation of its logic across multiple client versions.

As opposed to other forms of web services, such as WSDL or SOAP which are procedurally oriented, REST is data-oriented. It is commonly used for simple and unified access to resources – usually data, or state of the application. All resources have their own identificator and REST uses 4 basis methods of access, known under the acronym CRUD - create, retrieve, update and delete. These methods are implemented with corresponding HTTP methods – POST, GET, PUT and DELETE. In the context of this work, the abovementioned identifiers are actually parts of Sinatra routes.

Web services usually offer their REST-based API in order for other application to be able to use their resources. In a particular example, Smart Proxy offers API for its modules, where a particular route + method combination will provide some information or invoke an action on the proxy. For illustration,

```
GET localhost:8000/puppet/ca
```

where `localhost:8000` is the address and port of the Smart Proxy, will return (i.e. display in the browser) a list of all Puppet certificates stored in the Smart Proxy, as per its API, together with an HTTP response code (in this case it will be "200 OK", since it did not encounter any problems).

3.2 YAML

YAML Ain't Markup Language, also known as YAML is a data serialization language designed to be human-friendly and work well with modern programming languages for common everyday tasks [5]. Not only is YAML easily readable by humans, but it is also easy to implement and use by programmers. Data formatted in YAML are easily portable between various programming languages and it matches the native data structures of agile languages. Moreover, YAML is robust enough to embed another serialization formats (such as

JSON or XML) within a YAML file.

The syntax of YAML provides a way to easily map ordinary data types used in high-level programming languages, for example lists or hashes, as well as complex nested data structures. YAML is especially suitable for use in configuration files or document headers.

An example of a YAML file would be:

```
- - -
items:
- part_no: A4786
description: Apple
price: 1
quantity: 4

- part_no: E1628
description: Orange
price: 2
quantity: 1
```

Figure 3.1: *Example of the YAML format*

where `---` signs the beginning of the file, `items` is a name of a complex data structure displayed (for example an array of hashes) and every item is being represented as a hash.

In this work, YAML was used while exporting tasks into a file, as well as parsing them from it. It was chosen due to being easy to use and parse as well as its readable structure. Moreover, YAML is considered a standard choice for data serialization in Ruby projects which makes it easy to work with in Ruby.

3.3 JSON

JSON, or JavaScript Object Notation, is a lightweight data-interchange format that uses human-readable text in order to transmit data objects that consist of either arrays or pairs of attributes and their values. It is commonly used in client-server communication. While it is a language-independent format that can be easily processed by every

modern programming environment, it was derived from JavaScript, which also means if a client receives data from a server in JSON format, it can use it like any other JavaScript object. JSON API is designed to minimize both the number of requests and the amount of data transmitted between clients and servers [6].

Like YAML, JSON is easy to both generate and parse, not only due to its simplicity but also due to great support in many programming languages. However, since JSON's foremost design goal is simplicity and universality, it is done at the cost of reduced human readability. Its main advantage over YAML is that JSON is usually faster.

JSON can be also viewed as a natural subset of YAML, therefore, every JSON file is a valid YAML file, which makes it easy to migrate from JSON to YAML if/when the additional features are required.

An example of JSON syntax could be

```
{
    "colorsArray":[{
            "colorName":"red",
            "hexValue":"#f00"
        },
        {
            "colorName":"green",
            "hexValue":"#0f0"
        }
    ]
}
```

Figure 3.2: *Example of the JSON format*

It can be easily seen how this structure could be represented in YAML.

This work uses JSON-formatted strings when mediator proxy sends responses to Foreman, as well as during logging processing. It was chosen due to the fact that both Foreman and Smart Proxy already use JSON for these purposes (i.e. Smart Proxy answers to queries in JSON format, and Foreman accepts the answer in this for-

mat as well), therefore it made most sense to use JSON instead of complicating the issue using a different format that would have to be converted into JSON anyways.

3.4 Sinatra

Sinatra is a free and open source web application library and domain-specific language written in Ruby. It is dependent on the Rack web server interface and emphasizes a minimalist approach to development, offering only what is essential to handle HTTP requests and deliver responses to clients. It is not a typical Model-View-Controller framework, but ties a specific URL directly to relevant Ruby code and returns its output in response. However, it does enable the programmers to write clean, properly organized applications: separating views from application code, for instance [7].

Sinatra can be considered an alternative, more lightweight option to Ruby on Rails that also offers the programmer more fine-grained control over a web application. Its biggest advantages are speed, simplicity and efficiency and it is best used for smaller web applications.

Sinatra, just like most web frameworks, has a concept of routes in order to handle various HTTP request types – with the most common requests abiding by a RESTful architecture.

In Sinatra, a route is an HTTP method and a regular expression to match the requested URL. Each route is associated with a block of code that will be executed upon the visit of its URL. The routes are uniquely matchable, there must be no ambiguity. An example of a route looks like:

```
get '/hello' do
 "Hello World"
end
```

Figure 3.3: *Example of a Sinatra route*

which means that in case the user visits the corresponding address with `/hello` route and GET method (for example

`https://www.address.com/hello`), the webpage he obtains will display `Hello World`.

The four basic HTTP request methods are the following ones

- GET
- POST
- PUT
- DELETE

GET is used to obtain the data, DELETE to delete them and the difference between POST and PUT is that PUT is used in order to update the existing data while POST creates a new record.

Amongst many Sinatra's features belongs also an ability to work with URL parameters with simply accessing them by

```
put '/people/:id' do
   person = params[:id]
   #update the person with the particular id
end
```

It is even possible to use the Sinatra handler for sessions, just by adding this to the application:

```
enable :sessions
```

This could be used to, for example, display a session message in case of some particular action. All session information will be stored in a cookie in the user's web browser.

Another important functionality of Sinatra is the ability to let the user download or upload files which was essential for the implementation of exporting tasks and importing responses. Sinatra is able to either save the data into a file directly, or offer the data to the user in the form of a traditional web browser dialogue. Similarly, it is able to obtain a file – by directly accessing it on the hard drive when a path is supplied or via upload.

All of the aforementioned features were used in this work.

3.4.1 Templates

Sinatra is built upon an incredibly powerful templating engine, Tilt. Tilt is designed to be a "thin interface" for frameworks that want to support multiple template engines [7].

In order to use templates, only `erb` (Embedded Ruby) is necessary (and it is already included in Ruby). Additionally, the view files are used to allow splitting the markup for a view into a separate file. The default directory for view files is located in `views/` in the application's root directory.

```
get '/greeting' do
   text = "Hello"
   erb :greeting, :locals => { :message => text }
end
```

The source code example above will load the `greeting.erb` file from the `views/` directory. Erb files are typically normal HTML files that allow the programmer to include Ruby code inside the specific tags. This Ruby code will be parsed before being sent to the browser in a similar way how it is possible to include PHP or ASP code in a webpage.

In this particular `greeting.erb` file, the example content could look like

```
<html>
   <head>..</head>
   <body>
     <p><%= @text %></p>
   </body>
</html>
```

Which means, that upon visiting `/greeting` route, it will load the `greeting.erb` view (which will display `Hello`). The text in special signifiers `<%= @text %>` is actually code that will be interpreted as Ruby code, together with using a variable obtained from the code of the route. Instead of the message, or any other HTML code, it can be yielded to an actual Ruby code and there is also a possibility to pass

variables or objects to the view where they could be processed and displayed. This was heavily used in this work while implementing the WebUI component.

3.5 Design and characteristics of the mediator proxy

3.5.1 Introduction

The generalized usecase for the mediator proxy is to provide an offline connection between two entities that originally need to be online in order to communicate to each other.

In this work, mediator proxy provides a mean for an offline communication between Foreman and its Smart Proxy. In some environments, an internet connection might be unreliable, too slow, restricted or completely nonexistent. Mediator proxy is able to present itself as a "real" Smart Proxy towards Foreman and also to communicate like one. It will report all of the available functionalities of the real proxy to the Foreman. Followingly, it is able to communicate with a real Smart Proxy on behalf of Foreman. Therefore, it serves as a messenger between these two entities.

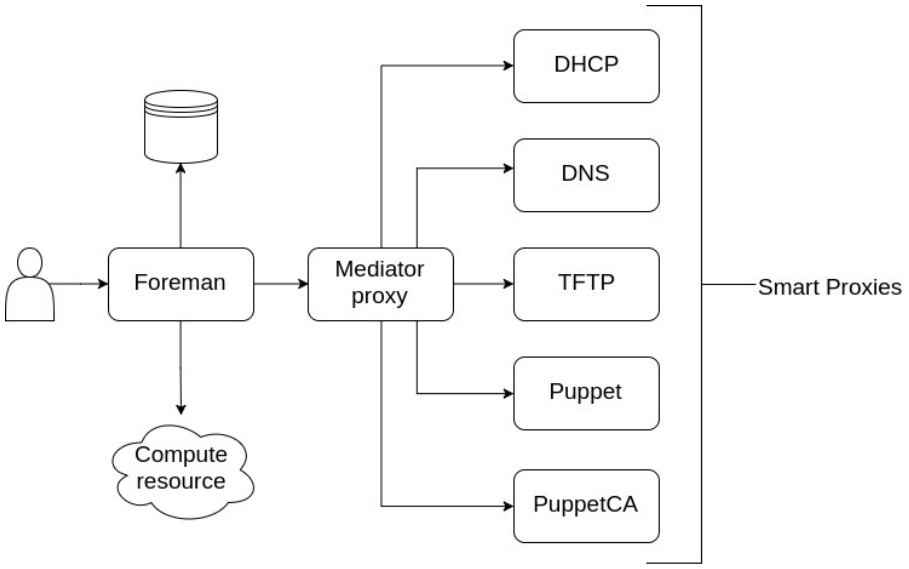


Figure 3.4: *Foreman architecture with mediator proxy*

Since the communication with mediator proxy is presumed to be offline-based, it was decided that the mediator proxy would have 2 separate parts – one facing the Foreman while the second would be facing the actual Smart Proxy. Both parts would be located on their respective nodes. Their mutual communication is based on an exchange of files with information data.

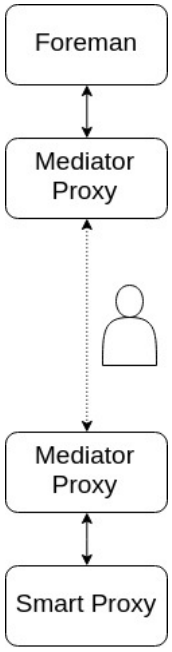


Figure 3.5: *Scheme with mediator proxy*

Due to the communication being offline, this exchange was designed with a particular situation in mind - a human administrator would export the request data from one node (in this case, Foreman itself) and save it into a file, which they would subsequently store on a suitable storage device. Then, they would physically deliver the data to a location of the second node (in this case Smart Proxy) and import them from the storage device. After the imported data would be processed, they would export the response into a file, save it into the aforementioned storage device and return back to the original location where they would import the responses back to the first node. For both nodes, the communication looks like a regular online exchange.

Real Smart Proxy may (and usually does) contain more than one module, but it is also possible to have the presence of the modules divided between multiple proxies, (e.g. one would provide DNS server functionality, another would be a TFTP server, etc). The solution presented in this work is ready to be adjusted to be able to work with this scenario – one mediator proxy would serve as a representative for a group of smart proxies and the tasks will be exported into separate files and answered on their respective proxies. The difference would be in sorting the incoming Foreman tasks into separate buffers (and then, subsequently, files), each containing tasks intended for one of the proxies, depending on the module necessary for the task. Each Smart Proxy would then have its own second half of mediator proxy, responsible for obtaining the answers.

It is also worth to note that during the registration of the mediator proxy, Foreman requests to provide a list of supported modules. In order to be able to satisfy the widest scope of requests (for example when used with multiple smart proxies, as described above), the mediator proxy has a hardcoded answer that it supports all of the possible modules. This leaves it up to system administrator's discretion which tasks to request when intending to export and import them to a particular Smart Proxy.

3.5.2 Data format

The data extracted from Foreman are stored in the form of tasks. Any request from Foreman to a Smart Proxy is transformed to a task and saved in a queue of tasks, together with its parameters and a unique identificator. Upon the task export, the queue of tasks is saved as a YAML file, which makes it easy to parse in subsequent phases of the communication. On the side of the Smart Proxy, the queue of tasks is being imported and transformed back into the same request Foreman would send, were the communication online.

Responses from the actual Smart Proxy for the requests are collected and saved as a YAML file. When imported back into the Foreman's side of Smart Proxy, they are being stored in a queue of responses. Since both tasks and responses use unique identificator, the mediator proxy is able to match them together easily. The replies are handed over to Foreman in a JSON format, as it would be were it a

regular online communication.

3.5.3 Specifics

3.5.3.1 Foreman's side of the mediator

The part of mediator proxy located at the Foreman's side is the more complicated one. It consists of a web server and also a Ruby script. In order to correctly obtain requests from Foreman, the web server part of the mediator proxy has to be running all the time. The server part takes care of recording the requests in the form of tasks, providing initial reply to Foreman (more details in section *3.5.3.4.*), exporting, importing the responses and presenting the responses back to Foreman. All of this can be administered using its own WebUI. More information about the functionalities offered by mediator proxy's WebUI are provided in chapter *5*. The script that is a part of this side provides an optional command line option for exporting tasks and importing responses.

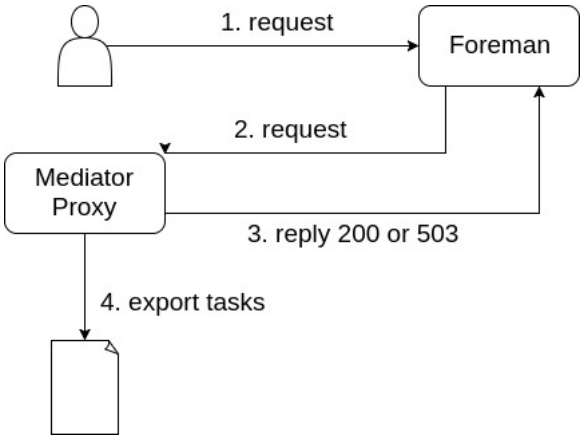


Figure 3.6: *Foreman's side of mediator proxy*

3.5.3.2 Smart proxy's side of the mediator

This side is the more simple one, since its main functionality is just a simple one-time communication exchange with the actual Smart Proxy (it does not need to be running all the time, it is only invoked

when necessary) and it also does not keep any data inside. It consists of a Ruby script, whose purpose is to parse tasks from the YAML file, create HTTP requests from them and send it towards the Smart Proxy. Subsequently, it receives a reply from the Smart Proxy and stores it into a YAML file.

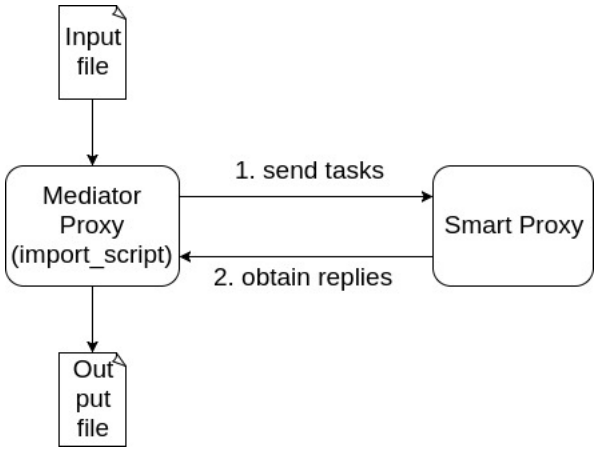


Figure 3.7: *Smart Proxy's side of mediator proxy*

An additional feature is the ability to list the tasks saved in the obtained YAML file in a more readable and compressed output.

3.5.3.3 Timeouts

The first design challenge was the issue with timeouts. This solution, as opposed to online communication, is unable to implement any timeout capabilities. Due to its very nature, it is not possible to ensure when the task would be exported, imported to the Smart Proxy or when the responses will be delivered. Therefore the tasks do not expire, however their order is being preserved and they are processed in the order of creation.

If the administrator wishes to obtain fresher replies to the tasks, they can simply go on with the whole procedure again and import the newer responses; these will always replace the older ones.

3.5.3.4 Delivering the replies

One of the biggest issues was delivering the replies to Foreman. When Foreman sends a request towards the Smart Proxy, the mediator proxy replies to him with only an HTTP code - 202 Accepted or 503 Service Unavailable, depending on the type of the task. Whichever it is, it is a finite reply and for Foreman, the matter is closed.

While the original Smart Proxy replies positively with 200 OK, in this case, 202 Accepted is a better-fitting response code. It signs that the request has been accepted for processing but the processing has not been completed [8]. Therefore, it replies with outcome-independent confirmation, which is exactly describing the whole scenario, since this reply is used for data-modifying tasks, i.e. the PUT/-POST/DELETE methods, where the mediator proxy will request the action towards the actual proxy, but cannot guarantee the result.

503 Service Unavailable is the reserved reply for non-data-modifying tasks (i.e. the GET method) – for example, when Foreman requests a list of some entities, like certificates – he awaits a reply with content, not only a positive or negative confirmation, but mediator proxy is not able to provide it at that time. 503 signifies that server is temporarily unavailable, which fits the scenario very well, since the content will be eventually offered to Foreman.

In order to deliver the actual reply, after the replies are imported back to the mediator proxy, Foreman must make exactly the same request once again. This can be provided by a human system administrator and currently it is the only way to ensure the reply delivery.

In the future, this might be avoided by implementing custom headers that would prompt Foreman into automatically repeating the same request some predefined time later, however, this would require changes in Foreman itself.

3.5.3.5 Buffering

The initial design stored the created tasks by appending them to a file upon their creation so that the tasks would be already considered "exported" and ready to be copied to a storage device without any further explicit action. This was changed in favor of the buffers, which is a more optimized solution.

There are 2 buffers at the Foreman's side of the mediator proxy – one of them stores tasks and the other stores imported responses. If the system administrator wishes to export the tasks, all tasks in the task buffer are flushed into a YAML file.

It is worth to note that due to the implementation of the buffers, both of them are cleared whenever the mediator proxy is restarted. Therefore there is no persistent storage solution present at the moment, which also means that in case the mediator proxy is restarted before the responses are delivered (and consumed by Foreman), responses delivered after the restart will never be matched to their respective tasks (due to the matching being done via UUIDs of the tasks) and therefore the whole process will have to be repeated.

However, it is possible to implement a storage in the future, most likely in the form of a database on the Foreman's side of the mediator proxy. This would ensure additional stability and persistence for mediator proxy tasks.

3.5.3.6 Logging

The mediator proxy creates a log record upon the creation of every task. Its logs are saved into a file and are accessible from Foreman. What was omitted from logging was time, since the time of task creation is not important in this case. Logging level is not important either, since the mediator proxy does not actually perform any of the tasks it obtains from the Foreman, therefore it does not produce error messages.

However, the order of the creation of tasks is preserved. It should be noted that logging serves only for basic informational purpose.

3.5.3.7 SSL and security

This work omits the use of SSL since it was not necessary in order to provide a proof of concept and would only unnecessarily complicate the implementation.

It order to support SSL functionality, the mediator proxy would have to have its own certificate and both parts of the mediator proxy (i.e. including scripts) would have to be able to work with it.

Similarly, SSL certificates could be used in order to sign the ex-

ported tasks/imported responses to ensure their authenticity and integrity, especially with regards to the offline transportation part.

3.5.3.8 UUID

In order to identify the tasks, it was decided to use UUID hexadecimal numbers, generated during the creation of the task.

Normal ordinal numbering would not suffice - the mediator proxy does not use any database or any other storage solution, after every restart, its buffers are cleared and the numbering therefore would therefore start from zero. This would cause confusion with regards to previous tasks.

Therefore, the chosen solution is the use of `SecureRandom` function from the standard Ruby library. While it is not in fact guaranteed that the UUIDs provided by this function would be unique, the chance of collision is small enough to consider this solution satisfactory.

The main purpose of UUID is to match a task (stored in the task buffer) to its response (stored in the response buffer), which is usually done while searching for a response to a corresponding task in order to offer it to the Foreman or while importing responses into the buffer - unmatched responses will not be stored.

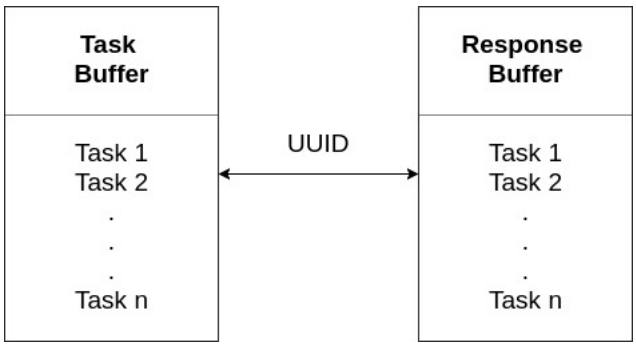


Figure 3.8: *UUID matching for buffers*

3.5.3.9 Preload of tasks

If the administrator wishes to save some time, there is a preload functionality that creates a batch of predefined tasks (for example, obtaining a list of certificates) and pushes them into the task buffer, the same way it would have been done were Foreman to request them.

Tasks predefined in this feature are the ones whose purpose is just to obtain information, not create or modify it (they also do not require any parameters).

This feature saves time for when Foreman asks these requests, as they can be already answered and prepared for use. It can be especially useful during the phase when an administrator invokes the request in Foreman for the second part (see section *3.5.3.4*) – if the request is part of the preload batch (and was preloaded), its answer can be immediately offered to Foreman.

One of the possible improvements for this part would be including an expiration time for these preloaded tasks.

4 Workflow and implementation

This chapter describes details of the implementational part of this work as well as the supposed workflow. For illustrational purposes, an example workflow with screenshots is provided in *Appendix B*.

4.1 Creating a task

When a system administrator clicks in Foreman's WebUI on a particular Smart Proxy-related action (for example even upon clicking on the particular proxy in the smart proxies listing), Foreman issues an HTTP request (URL + method) towards it and awaits a reply. Therefore it is necessary to register a mediator proxy instead of the actual proxy to Foreman. The actual proxy does not need to be registered, since it is not online and any request towards it would be unsuccessful.

When the mediator proxy obtains the request, it matches it unambiguously onto one of the routes it has implemented. Subsequently, the function `reply_or_create_task` is invoked. This is an essential function for the task creation. It searches the `responses` buffer, looking for a response to a task with the same route, method and parameters. For this it uses the helper function `find_by_query_method_parameters`. If such response exists, it saves it into a variable, deletes the response from the `responses` buffer, as well as its corresponding task in the task buffer. Subsequently, it replies with the saved response to Foreman.

In case there is no response available for the request, it will create a new task for it. For this purpose, it is necessary to obtain details of the task, using `request.env` method available in Sinatra. This method provides access to parameters of the environment. In particular, the function needs to know the route (with `request.env['PATH_INFO']`), as well as the HTTP method (`request.env['REQUEST_METHOD']`) and parameters (`request.env['QUERY_STRING']`). Using these details, it creates a new task, and pushes it into the `buffer`.

Originally, the environment variables were passed manually to the function, but since Sinatra enables obtaining them directly via

the `request.env` method, it was deemed unnecessary and, currently, the `reply_or_create_task` function has only one argument, the log message for the particular action.

There is also another way to create a task which is preloading a batch of tasks from WebUI. This option in mediator proxy's WebUI iterates over a predefined (hardcoded) array of task parameters and for each one it creates a task object and pushes it into the task buffer.

After creating the task, the mediator proxy still has to reply to Foreman. It was decided that for non-modifying tasks (i.e. GET method) it will reply `503 Service Unavailable` and for others, modifying methods it replies `202 Accepted`. The idea behind this solution is that, after obtaining the information, Foreman usually wants to display it - however, in reality, the mediator proxy did not send any information to display. Therefore it makes sense to reply with an error message, so that Foreman keeps its old information for the time being. As for the modifying tasks, they need to be replied with a positive affirmation, due to Foreman's behavior where in the case of a succession of multiple modifying tasks (for example, during the provisioning a new host, where it is necessary to create various DNS, DHCP and TFTP records), in case one of the tasks fails (therefore gets an error response), the whole chain of operations is dismissed. Therefore the mediator proxy has to let Foreman think that every modifying operation was finished successfully. This Foreman's behavior however creates an issue that while provisioning, Foreman does not requests only modifying but also non-modifying actions, and dismisses the whole process when these are replied negatively (which they are by design). This can be avoided in future by implementing special routes for these actions to be preloaded in advance.

4.2 Exporting the tasks

When the system administrator wishes to finish the tasks, they need to extract them into a file in order to be able to deliver them to Smart Proxy. This can be done in two ways – either via mediator proxy's WebUI or via a script. The WebUI method is based on a simple click and download of the whole list of tasks and its details are described in the WebUI section of this work.

The script, also called `manage_script`, is a Ruby script that does the same, but from the command line. Due to the integration of Ruby's `OptionParser`, it also offers a help option, invoked with `-h` argument or when used with no arguments. This help will list all of the options this script offers together with their description.

The `manage_script` offers 2 main functionalities – export of the task list (`download`) and import of the response list (`upload`), the latter of which will be described in the section concerning the delivery of the responses.

As for the script arguments, the administrator needs to provide the address of the mediator proxy, together with the port number and also the file name they wish the task list to be saved as. In case the address is not the actual address of the mediator proxy, the script is able to detect it, inform the administrator about this error and abort.

In order to export the tasks from the mediator proxy into a YAML file, the script sends an HTTP GET request (via RestClient) to a `/file` route which is a part of the mediator proxy's WebUI and serves to offer YAML file of tasks. This script saves the reply from the request (i.e. the YAML structure) into a variable which subsequently writes into a new file, the name of which would be provided by the administrator in the input arguments. In case such file already exists, it will be overwritten. Finally, it notifies the administrator that the tasks were successfully exported.

4.3 Importing the tasks to Smart Proxy

After the file with tasks is stored and successfully transported onto the location of the actual Smart Proxy, an administrator has to use another script, called `import_script`. It is responsible for sending the tasks to the Smart Proxy and saving the answers.

Similarly to the `manage_script`, the `import_script` provides a help option, working in the same fashion. The script requires the address and port of the Smart Proxy, the path of the task file and the filename of the output file with responses.

Firstly, the script invokes a special method where it loads the input file and then it parses the YAML content into an array of task objects. In case the file does not exist, it aborts the operation.

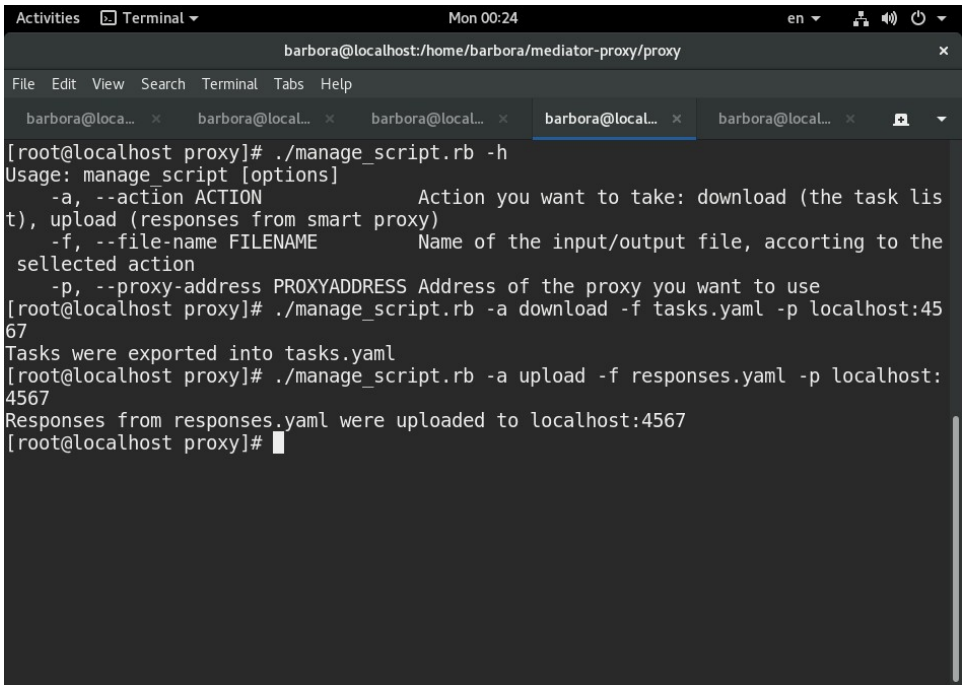


Figure 4.1: *Manage script*

Subsequently, the script iterates through the array and for each task it creates an HTTP request via RestClient with corresponding combination of route, method and parameter, obtained from the task. This is done through a function contained in the Task class whose return value is the response from the proxy. Moreover, it informs the administrator about every task sent.

The script saves the response in a variable and pushes it into a buffer of responses. After it is done iterating through the tasks, it flushes the response buffer into a YAML-formatted file, with the name as was specified in the output file argument.

In case the proxy address is invalid or the Smart Proxy does not provide specific functionality (e.g. the script sends a task concerning the TFTP module, but the proxy does not have TFTP enabled), it throws an exception, informs the administrator about the problem and aborts.

Finally, it informs the administrator that the tasks were successfully saved in the file.

Additionally, this script provides a listing function, where it can list the tasks from an imported task list file in a nicely formatted manner, which is helpful in case the administrator wants to review it before sending.

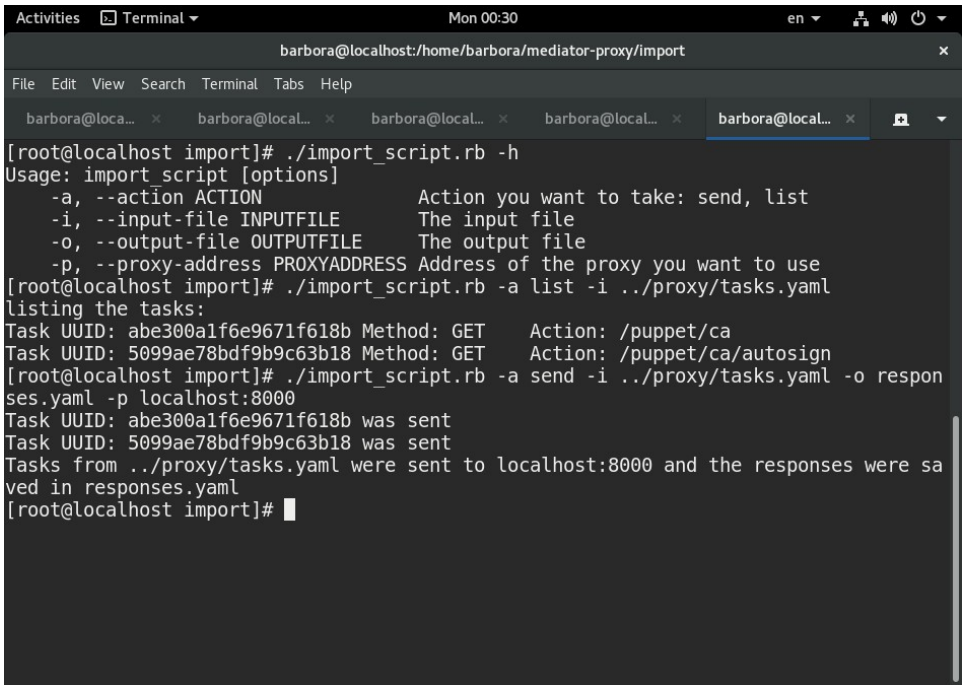


Figure 4.2: *Import script*

4.4 Importing the responses to mediator proxy

The last part of the process again offers two ways for the administrator – they can either upload the YAML file with the responses via a form in the WebUI, or use the aforementioned `manage_script` to perform the same operation from the command line.

The `manage_script` opens the input file and creates a RestClient request against `/responses` route on the mediator proxy, which is actually part of its WebUI (and it is also used in the WebUI based solution of this operation) and its purpose is to receive a YAML-formatted file, parse it into responses and store them.

As was previously mentioned, in order to consume a response, an administrator must invoke the request again from the Foreman's WebUI. It is worth to note that, in some cases (especially with the tasks that are supposed to modify data), Foreman may, even after if obtains the actual response, automatically (i.e. not triggered by an administrator) request some more information from the Smart Proxy. For example, if the administrator wishes to create an autosign record, after the whole procedure, when it comes to the point that he is creating it again (to get the actual response), right after the creation, Foreman additionally requests a list of all autosign records. Therefore, in order to have complete information, the whole procedure must be repeated again. It is an obstruction that cannot be avoided due to Foreman's own behavior.

4.5 Classes

This section describes the classes used in this work as well as the special methods included in them.

It is worth to note that all these classes have actually two versions, due to the mediator proxy consisting of two separate parts – one is used on the side of the mediator proxy that communicates with Foreman, the other is on the Smart Proxy's side. These two versions are essentially the same, except for some minor differences that will be explained.

4.5.1 Task

Class Task defines the basic task object. It was created in order to better represent tasks in this work and to make the manipulation with them easier.

Task objects consist of the following attributes: `action` (the route Foreman used), `method` (HTTP request method), `parameters` (parameters used with the action), `date` (date of the task creation) and `uuid` (unique identifier of the task).

The constructor only takes 3 parameters - `action`, `method` and `parameters`. The rest of the arguments are filled with values automatically – `date` is set to the actual date and time of the creation of

the task and as for the `uuid`, it is being set to a hexadecimal identifier by the `SecureRandom` function from standard Ruby library.

The Task class on the side of the Smart Proxy has an additional method `send_to_proxy` that takes the proxy address as a parameter and creates a RestClient request that is sent against the proxy. It saves the response and returns it.

4.5.2 TaskQueue

Class TaskQueue was created in order to provide a custom buffer solution for Task objects. It is an Array wrapped in a custom class that can enqueue tasks, delete them by UUID provided in an argument, or clear itself completely.

Both versions of the class are the same.

4.5.3 TaskResponse

Class TaskResponse creates the objects representing responses obtained from Smart Proxy.

Amongst its attributes belongs `response`, which is the body of the response itself. It was not necessary to store any headers or any other information, therefore only response body is used. Additional attributes are `query` that defines the route of the request, `method`, i.e. the HTTP method used, `uuid` that identifies the task that particular response belongs to and also the `parameters` of the task belonging to the response.

Both versions of this class are the same for the both parts of the mediator proxy.

4.5.4 TaskResponseQueue

The purpose of this class is to provide similar buffer than TaskQueue, but for TaskResponse objects.

Similarly, as it is an Array that provides basic `enqueue` and `clear` methods as well as some additional functions. It can delete a response based on the provided `uuid`, while also it can find a response based on the provided `query`, `method` and `parameters` in the

`find_by_query_and_method` which is especially useful during the task creation process.

In both parts of the mediator proxy, this class is the same.

4.6 Modules

Smart Proxy offers various functionalities, provided in the form of modules. Every proxy may contain multiple modules, or even all of them. Currently, Smart Proxy supports Puppet, PuppetCA, DHCP, DNS and TFTP – all of which are necessary in order to provision a new host.

4.6.1 Puppet

Puppet is a system management and configuration tool. It stores information about the host systems in so called *manifests* and uses them in order to create environments. Followingly, it can apply a defined management action on all computers in the selected environment.

With this module, Foreman can request proxy to trigger a Puppet run, return the list of Puppet environments as well as all of the classes in a specified environment.

4.6.2 Puppet CA

Puppet CA (Certification Authority) provides a way for Smart Proxy to manage SSL certificates for Puppet.

This module can be used to list of all Puppet certificates [9], as well as remove (revoke) them or sign pending Puppet certificate requests. Moreover, it also works with autosign entries – it is able to list them, add a certificate name as a new autosign entry as well as delete a requested entry.

4.6.3 DHCP

DHCP, or Dynamic Host Configuration Protocol, is a standardized network protocol. DHCP server can dynamically distribute network configuration parameters – for example an IP address.

A Smart Proxy containing this module can provide a list of subnets as well as list of records in a particular subnet. Moreover, it can provide an unused IP address in a particular subnet, create a new reservation or delete a previously filed one.

4.6.4 DNS

DNS is a hierarchical decentered naming system for computers on a particular network. It serves to assign domain names to the IP addresses.

Smart Proxy with DNS capability can be used to create a new DNS record or to remove a particular one (depending on the provided value).

4.6.5 TFTP

Trivial File Transfer Protocol, or TFTP, allows a server to obtain or put a file onto a remote host.

This functionality can be used to create a pxelinux (as well as pxeconfig) configuration for a host with provided MAC address, as well as delete it or provide a default configuration file. It can also fetch the TFTP server name to be used within a DHCP record.

5 WebUI

The purpose of mediator proxy's WebUI is to provide a simple and easy-to-use graphical interface for the mediator proxy. While the Ruby scripts (`manage_script` and `import_script`) provide the way for the basic purpose of the mediator proxy, WebUI offers additional useful functionalities.

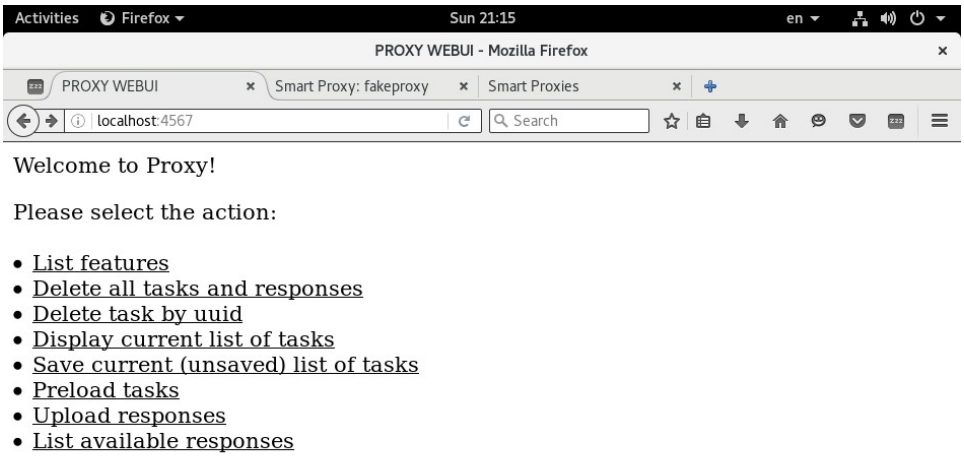


Figure 5.1: *WebUI of mediator proxy*

5.1 List features

This option provides a listing of modules supported by mediator proxy, in JSON format.

5.2 Delete all tasks and responses

Clicking on this option will trigger the *clear* operation on both buffers - the one with saved tasks and also the one with responses for the tasks.

Subsequently, it will display a session message, informing the administrator that the tasks were deleted.

5.3 Delete task by UUID

If the administrator wishes to delete a particular task, they can use their UUID (obtainable from the list of tasks output) and use a small form under this option. It is tied to a method that will search for a task with the corresponding UUID in the task buffer and delete it.

If this task already has a response in the response buffer, it will be deleted, too.

This option has the potential to be expanded into a list of tasks with checkboxes where it could be possible to delete multiple tasks at once.

5.4 Display current list of tasks

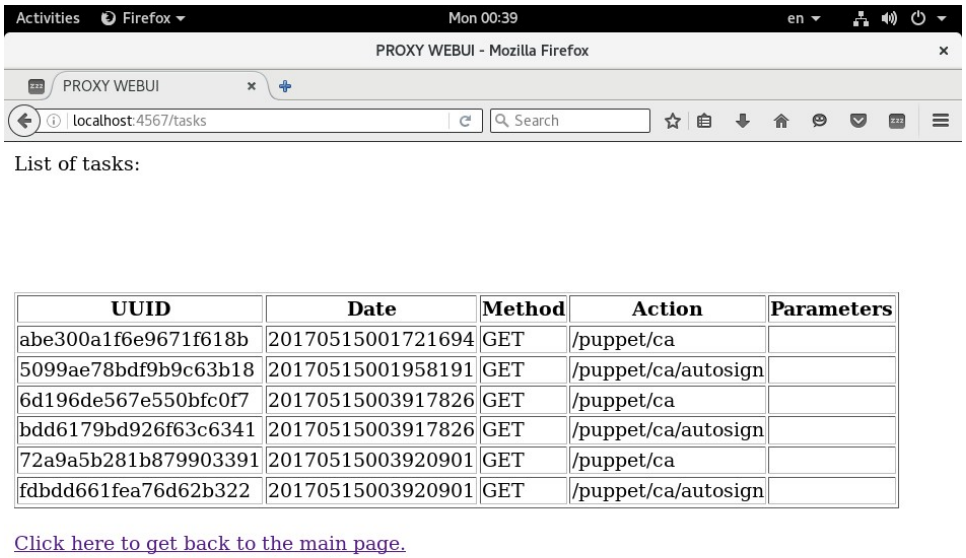
This option serves an informational purpose where an administrator can view the current state of the task buffer.

Originally, it displayed the contents of the buffer in YAML format, which was easily readable; however, it was later improved and now it generates an HTML table, where every row represents a task and every column represents a task attribute.

5.5 Save current list of tasks

This option serves to provide a YAML file containing exported tasks. It is a corresponding WebUI way to the `download` option in the `manage_script`.

Clicking on this option will offer the content of the task buffer in the form of a YAML-formatted file via classic web browser dialogue.



UUID	Date	Method	Action	Parameters
abe300a1f6e9671f618b	20170515001721694	GET	/puppet/ca	
5099ae78bdf9b9c63b18	20170515001958191	GET	/puppet/ca/autosign	
6d196de567e550bfc0f7	20170515003917826	GET	/puppet/ca	
bdd6179bd926f63c6341	20170515003917826	GET	/puppet/ca/autosign	
72a9a5b281b879903391	20170515003920901	GET	/puppet/ca	
fdbdd661fea76d62b322	20170515003920901	GET	/puppet/ca/autosign	

Figure 5.2: *List of tasks in WebUI*

In the future, this functionality can be improved by adding an option to export only a selected task or a group of tasks, which would be useful in case the mediator proxy communicates on behalf of multiple smart proxies, with different modules. This way it would be possible to export only tasks belonging to a particular module.

5.6 Preload tasks

Clicking on this option will preload a predefined batch of tasks into the task buffer, as previously described in section *3.5.3.10*.

5.7 Upload responses

This is a corresponding WebUI-based way to the `upload` option in the `manage_script`. It redirects the administrator to a special page where they can (via a classic web browser form) select a YAML file

with responses and its content will be uploaded, parsed and saved as response objects into the response buffer.

5.8 List available responses

A parallel function to the one that lists the tasks. It will generate and display an HTML table from the response buffer, where the rows represent the responses and columns represent the actual response attributes.

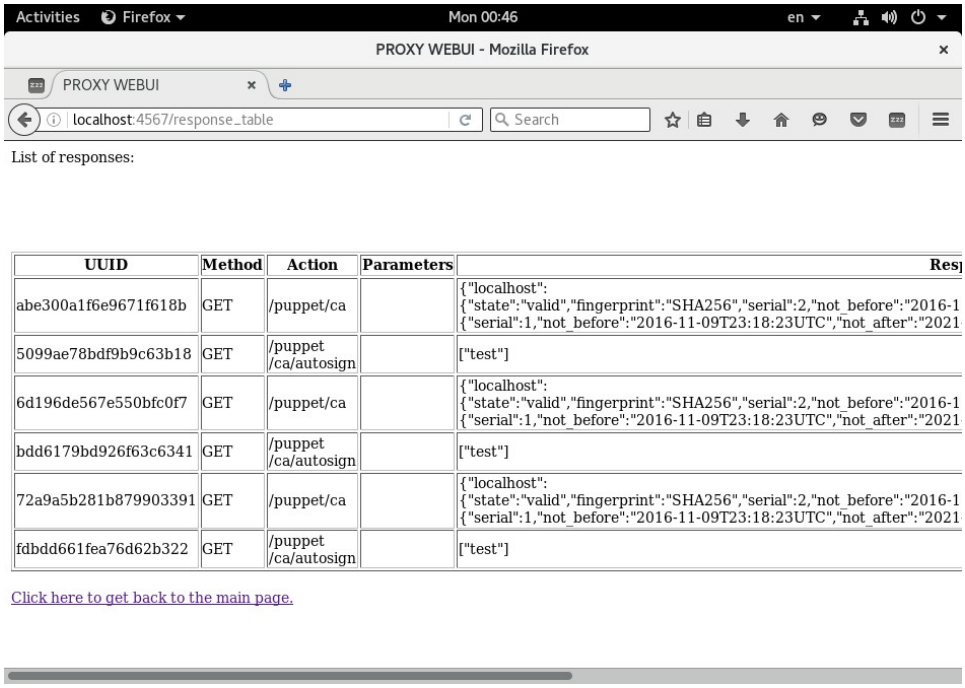


Figure 5.3: *List of responses in WebUI*

6 Installation and initial setup

6.1 Requirements

In order to be able to try the mediator proxy functions, it is necessary to set up an instance of Foreman, as well as a Smart Proxy. Foreman is available for a plethora of Linux-based operating systems, however in this chapter we will focus on the Red Hat-based distributions – Red Hat Enterprise Linux 7, Cent OS 7 and Fedora 24. It is necessary for the server to be of x86_64 architecture, have at least 2GB of memory (4GB optimal) and 1GB of disk space (2GB optimal).

The following instructions will use `yum` command as an example, but it is worth to note that this command is native to RHEL or CentOS systems only. When using Fedora, it is necessary to use `dnf` instead of `yum` command.

6.2 Foreman

It is not necessary to use a developer setup of Foreman in order to use the mediator proxy. Package installation is a sufficient and easy way in this case.

On CentOS, as a prerequisite, it is necessary to be subscribed to the EPEL repository which can be done with

```
$ yum -y install http://dl.fedoraproject.org/pub/epel/epel
-release-latest-7.noarch.rpm
```

```
$ yum -y install https://yum.theforeman.org/releases/
1.14/el7/x86_64/foreman-release.rpm
```

and also have Puppet installed

```
$ https://yum.puppetlabs.com/puppetlabs-release-pc1-el-7.
noarch.rpm
```

On Fedora, it is only necessary to enable the Foreman repository

with

```
$ dnf -y install https://yum.theforeman.org/releases/1.14/
f24/x86_64/foreman-release.rpm
```

As for the Red Hat Enterprise Linux System 7, first, it is necessary to enable RHEL Optional repository:

```
$ yum-config-manager --enable rhel-7-server-optional-rpms
```

then install Puppet

```
$ yum -y install https://yum.puppetlabs.com/puppetlabs-release
-pc1-el-7.noarch.rpm
```

and enable both EPEL and Foreman repositories

```
$ yum -y install https://dl.fedoraproject.org/pub/epel/
epel-release-latest-7.noarch.rpm
```

```
$ yum -y install https://yum.theforeman.org/releases/1.14
/el7/x86_64/foreman-release.rpm
```

When the repositories are enabled, we can proceed with downloading the installer

```
$ yum -y install foreman-installer
```

Now we are able to run the installer with

```
$ foreman-installer
```

In contrast with the installation of Smart Proxy, it is not necessary to make any SSL-related changes in Foreman configuration files in prior to the installation. Since the communication between Foreman and Smart Proxy in the scope of this work is one-sided (Foreman sends requests and Smart Proxy replies), Foreman does not require

any authentication from Smart Proxy.

After a successful completion, the installer displays an output with the address of the Foreman (accessible via any internet browser) and admin credentials.

After the installation, Foreman should be already running and we can log into its WebUI.

In case it would be necessary to restart Foreman, use the following command:

```
$ service foreman restart
```

6.3 Smart Proxy

The Smart Proxy shall be installed on a separate server, however, for trial purposes, it is completely in running order to use it on the same server as Foreman.

The `foreman-proxy` package necessary to run Smart Proxy is installed together with the Foreman itself. For our purposes, we do not need SSL, therefore we have to edit */etc/foreman-proxy/settings.yml* where we comment out all SSL/HTTPS related parameters like `ssl_certificate`, `ssl_ca_file`, `ssl_sertificate`, `ssl_private_key`, `foreman_ssl_ca`, `foreman_ssl_cert`, `foreman_ssl_key`, allow `http_port` line and then run

```
$ service foreman-proxy start
```

Smart Proxy modules can be enabled via configuration files in */etc/foreman-proxy/settings.d* directory. Each module has its respective YAML configuration file with the option

```
# Can be true, false, or http/https to enable just one of
 the protocols
 :enabled: false
```

where it is necessary to rewrite `false` to `http`.

In case of problems with connection to Foreman, it is worth to try to turn off SELinux with

```
$ setenforce 0
```

or optionally to turn off the firewall, adjust rules in `iptables` or enable Foreman as a trusted host in the configuration file.

6.4 Mediator proxy

Prerequisite for running mediator proxy is to have installed Ruby in version (at least) 2.2.2 and also corresponding `ruby-devel` package

```
$ yum -y install ruby-devel
```

In case of using a clean installation of the operating system, it is necessary to also install `gcc-c++` package with

```
$ yum -y install gcc-c++
```

Followingly, update all gem packages currently present on your system with

```
$ gem update
```

Then change your working directory to the one where you wish to install the mediator proxy and copy the source code directory from a medium attached to this diploma thesis.

Alternatively, it is also possible to install `git`

```
$ yum install git
```

and clone the source code from its git repository

```
$ git clone https://github.com/barbora1992/mediator-proxy/
```

Enter the source code directory and install the gems that are necessary for mediator proxy to work with

```
$ bundle install
```

Following that, it should be possible to run the mediator proxy with

```
$ bundle exec ruby main.rb
```

WebUI of a mediator proxy will be accessible from a standard web browser, with address localhost:4567 or 127.0.0.1:4567.

In order to register the mediator proxy in Foreman, it is necessary to open Foreman WebUI, visit *Infrastructure -> Smart Proxies*, and add a new Smart Proxy with corresponding address and port.

7 Conclusion

The aim of this work was to create a solution that would serve as a mean for offline communication between the Foreman and it Smart Proxy. This communication was supposed to be
time-independent and based on exporting and importing information and delivering them in form of the YAML-formatted files.

This thesis consists of multiple parts. The first part describes the work environment, in particular the Foreman – a tool for system administrators used for provisioning the hosts, as well as their configuration and lifecycle management. Another important component of the environment is a Smart Proxy – a server that, depending on its configuration, can function as a DNS/DHCP/TFTP/Puppet/PuppetCA server, or even all at once.

The second part introduces the mediator proxy, which would serve as the aforementioned solution, together with its characteristics and design details. It describes the components or principles used during the creation of this proxy, such as the REST architecture or Sinatra Web Framework, upon which was the application built. Similarly, it provides characteristics of other solutions used - namely YAML and JSON, both of which are languages used for data serialization.

Following this, it describes the actual implementation of the mediator proxy – it gives an insight into the actual succession of the actions, internal classes necessary to represent the data used in the communication, as well as it describes the actual modules implemented in the mediator proxy. The important parts of the implementation are also two helper scripts and last, but not least, a WebUI, provided for the system administrator to ease their work.

Finally, the last part of this thesis provides the instructions necessary for the installation and configuration of the Foreman environment that includes the mediator proxy. There are separate sections for the installation of Foreman, Smart Proxy and also the mediator proxy.

In this work, it was proven that such communication as a concept is possible and there are no major design obstructions in either Foreman or Smart Proxy that would disallow it.

7.1 Future work

The concept presented in this work allows multiple various improvements and additions in the future.

One of the possible additions for the mediator proxy could be the ability to represent multiple Smart Proxies, by being able to sort the incoming Foreman requests into multiple buffers or files, with regard to the targeted Smart Proxies and their supported modules.

Another improvement would be the implementation of persistent storage. The solution to this could be a simple database that would preserve the recorded tasks even in case of restart of the mediator proxy.

As for the WebUI, there are plenty of functionalities that can be added – an option to export only a selected task or a group of tasks, possibly with regards the module they belong to, as well as a similar delete option. Moreover, new preloadable routes can be added, which would be especially useful for provisioning purposes.

Bibliography

[1] Foreman. *What is Foreman?*, May 2017. https://theforeman.org/introduction.html.

[2] Foreman. *The Foreman Manual*, May 2017. https://theforeman.org/manuals/1.14/index.html.

[3] Red Hat, Inc. *Red Hat Satellite 6.2 Provisioning Guide*, May 2017. https://access.redhat.com/documentation/en-us/red_hat_satellite/6.2/html-single/provisioning_guide.

[4] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000. https://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm.

[5] I. döt Net O. Ben-Kiki, C. Evans. *YAML Ain't Markup Language (YAML™) Version 1.2*, October 2009. http://www.yaml.org/spec/1.2/spec.html.

[6] D. Gebhardt T. Kellen E. Resnick S. Klabnik, Y. Katz. *JSON Specification (v1.0)*, May 2017. http://jsonapi.org/format/.

[7] *Sinatra The Book*, May 2017. http://sinatra-org-book.herokuapp.com/.

[8] Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. W3C, June 1999. https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

[9] Foreman. *Smart Proxy API*, May 2017. http://projects.theforeman.org/projects/smart-proxy/wiki/API.

A Appendix

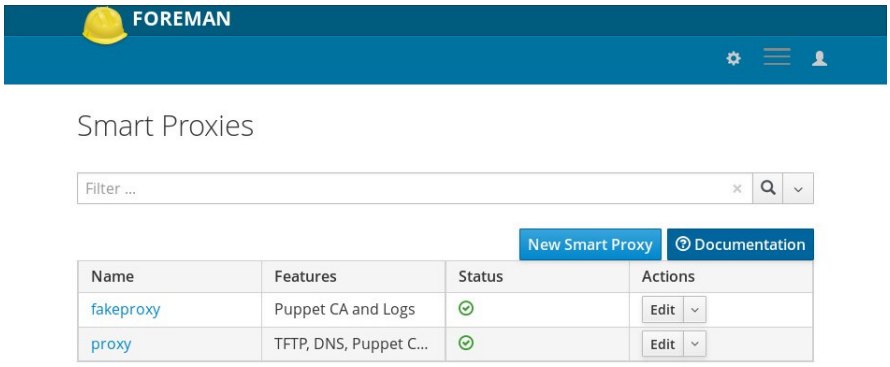
Part of this work is an electronic archive consisting of:

1. Text part of the work in PDF format
2. Source code of the application

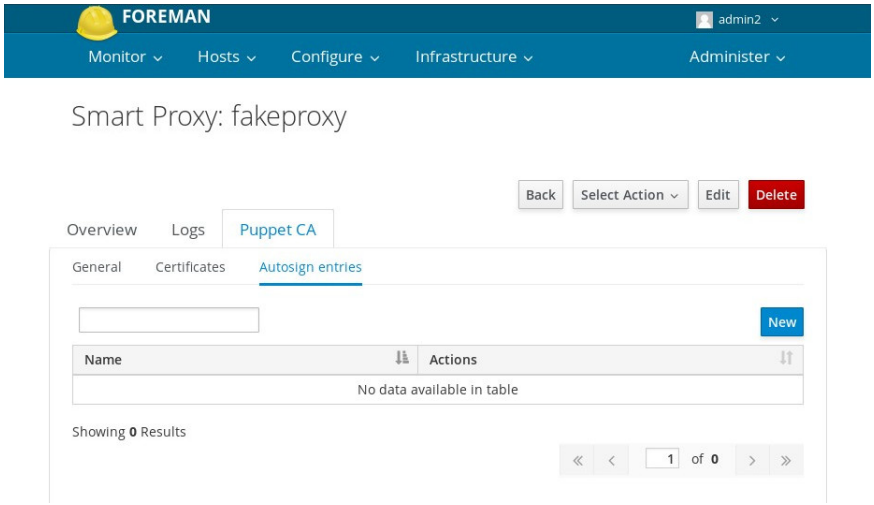
B Appendix

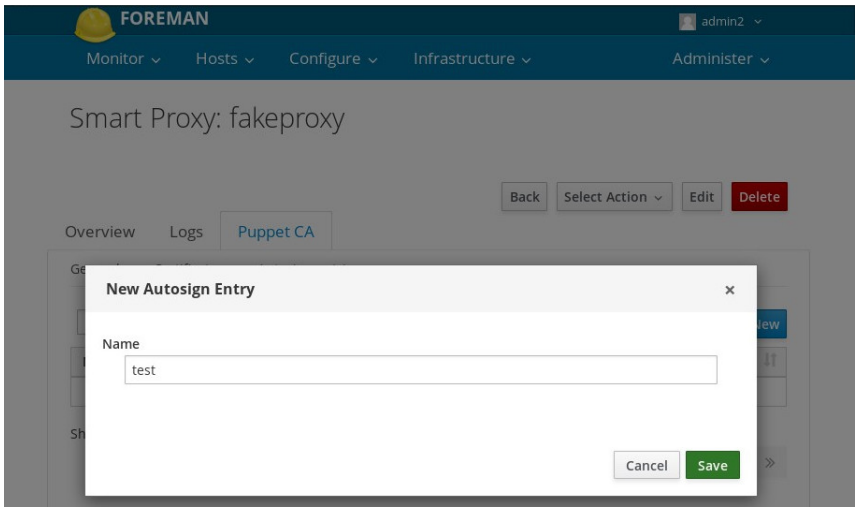
Example of workflow of the application:

We have registered a mediator proxy called *fakeproxy*. It will work with an actual Smart Proxy called *proxy* that we can also see registered in the Foreman. For the purposes of the illustration we will assume that *proxy* is at that time not online and we will communicate with it only via mediator proxy.

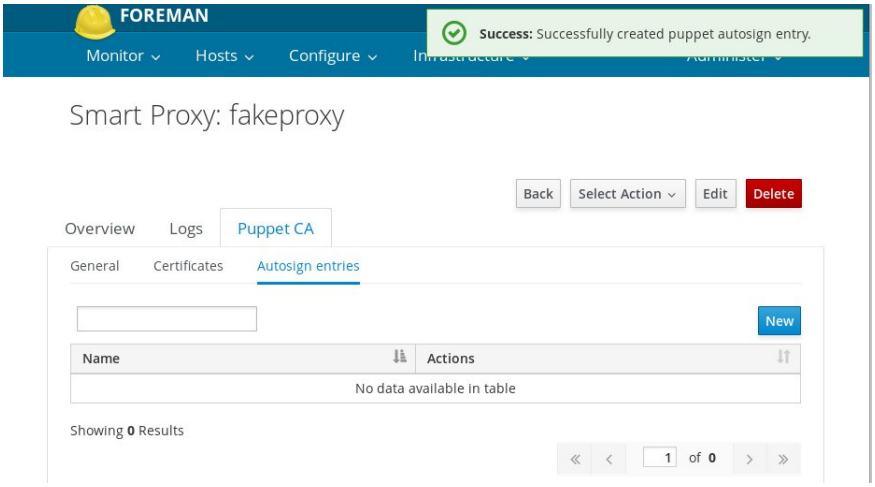


We can see *fakeproxy* does not have any autosign record, so we try to create one.





When we click on Save, we can see positive confirmation message (as expected), but there is no record created. If we refresh the page, it will show an error since Foreman will ask for lists of autosigns we do not have yet.



In WebUI we can have a look at the list of tasks. We can see there is a POST task recorded.

Now we will export the tasks with mediator proxy with `manage_script`, use `import_script` to send the tasks to the actual proxy and record the responses and again use `manage_script` to load

List of tasks:

UUID	Date	Method	Action	Parameters
716bb69768783685bc3c	20170515015543238	POST	/puppet/ca/autosign/test	
2a66ae9280f7b6dffc2c	20170515015546020	GET	/puppet/ca/autosign	
d88a480c117938d529dc	20170515015546128	GET	/puppet/ca/autosign	

Click here to get back to the main page.

the responses back into the mediator proxy.

```
[root@localhost proxy]# ./manage_script.rb -a download -f tasks.yaml -p localhost:45
67
Tasks were exported into tasks.yaml
[root@localhost proxy]#
```

```
[root@localhost import]# ./import_script.rb -a send -i ../proxy/tasks.yaml -o respon
ses.yaml -p localhost:8000
Task UUID: 716bb69768783685bc3c was sent
Task UUID: 2a66ae9280f7b6dffc2c was sent
Task UUID: d88a480c117938d529dc was sent
Tasks from ../proxy/tasks.yaml were sent to localhost:8000 and the responses were sa
ved in responses.yaml
[root@localhost import]#
```

```
[root@localhost proxy]# ./manage_script.rb -a download -f tasks.yaml -p localhost:45
67
Tasks were exported into tasks.yaml
[root@localhost proxy]# ruby manage_script.rb -a upload -f /home/barbora/mediator-pr
oxy/import/responses.yaml -p localhost:4567
Responses from /home/barbora/mediator-proxy/import/responses.yaml were uploaded to l
ocalhost:4567
[root@localhost proxy]#
```

We can see the responses in the WebUI now.

We have to file the record again and since Foreman asks for additional information right after clicking on Save, we have to repeat the export-import process again. When the tasks are imported, we can click on Refresh and we will see the new record.

List of responses:

UUID	Method	Action	Parameters
716bb69768783685bc3c	POST	/puppet/ca/autosign/test	
2a66ae9280f7b6dffc2c	GET	/puppet/ca/autosign	
d88a480c117938d529dc	GET	/puppet/ca/autosign	

Click here to get back to the main page.

