

Proyecto Fin de Carrera Ingeniería de Telecomunicación

Network Service for Independent Software Vendors

Autor: Carlos Sánchez Montero

Tutor: Jorge Jesús Chávez Orzáez

**Dep. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla**

Sevilla, 2017



Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Network Service for Independent Software Vendors

Autor:

Carlos Sánchez Montero

Tutor:

Jorge Jesús Chávez Orzáez

Profesor Titular

Dep. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Contents

1	Introduction	1
1.1	Network Service: general objectives	2
1.1.1	Updates and Messages	2
1.1.2	Marketing Tool	3
1.1.3	Additional Services for End-users	3
1.2	Project Scope	3
1.3	Project Management: SCRUM	4
2	Environment and Technologies	6
2.1	Technologies	6
2.1.1	Ruby	6
2.1.2	Ruby on Rails	7
2.1.2.1	MVC Architecture	7
2.1.2.2	Components of Rails	8
2.1.2.3	File Structure	8
2.1.3	RSpec	8
2.1.4	Nginx	9
2.1.5	MySQL	10
2.1.6	Capistrano	10
2.1.7	God.rb	10
2.1.8	RESTful Design	11
2.2	Initial Data Structure	11
3	Properties Storage System	13
3.1	Receiving Metrics. Simple storage system	13
3.1.1	Short API description	13
3.1.2	Initial Properties Storage	15
3.1.3	API Modifications	17
3.2	Properties Storage with History	18
3.2.1	Time Slots	19
3.2.2	Empty Slots	20

3.2.3	Losing Resolution	20
3.2.4	Database Migration	21
3.2.5	Processing Algorithms	23
3.3	Improved Properties Storage	24
3.3.1	Installation Tiers	24
3.3.2	Merge Property Slots	24
3.3.3	API Improvements	26
3.3.3.1	Add Notifications and Monitoring Methods	26
3.3.3.2	Avoid Sending Duplicated Notifications	27
3.3.3.3	Add Background Processing	28
3.3.3.4	Switch the API and Workers to Merb	31
3.4	Performance Tuning	33
3.4.1	Properties stored in separate tables	33
3.4.2	Remove Empty Slots	36
3.4.3	Indexes Configuration	37
3.5	Storage Size Estimation	38
4	Filtering Application Users	42
4.1	Rails plugins structure	42
4.2	Filter Models Plugin	43
4.2.1	Objectives	44
4.2.2	Implementation	44
4.2.2.1	Model methods extension	45
4.2.2.2	Fields configuration	46
4.2.2.3	Filtering Rules Generation	48
4.2.2.4	Cached rules	53
4.2.2.5	WillPaginate Compatibility	53
4.2.2.6	Filterable Model Extensions	54
4.2.2.7	Filter Model Extensions	54
4.2.2.8	View helpers	55
4.3	Installation Filters	58
4.3.1	Special filters	59
4.3.2	Installation filters applied to notifications	62
5	Reports Generation	64
5.1	Reports generation plugin	64
5.1.1	Conventions	66
5.1.2	Datasets	66
5.1.2.1	Graph Generation	67
5.1.3	EventList	69
5.2	Network Service Reports system	69
5.2.1	General Structure	69
5.2.2	Filtering	71
5.2.2.1	Current statistics	72

5.2.2.2	Evolution over time	73
5.2.2.3	Notification reach	75
5.2.2.4	Numeric property value evolution	76
5.2.3	Background Processing	79
5.2.4	Iterative Loading and caching slots	79
5.2.4.1	Memcached Collisions	82
5.2.5	Cache Preloading	82
5.3	Views Generation	83
5.3.1	Chart generation	83
5.3.2	Number of Installations	84
5.3.3	Notification Reports	85
5.3.4	Property value evolution	87
Conclusion		88
Bibliography		90
A Network Service User Manual		91

List of Figures

1.1	Scrum Process	5
2.1	Initial Data Structure	12
3.1	Data Structure with Simple Properties	17
3.2	Property Time Slots Generation	19
3.3	Property Empty Slots Generation	20
3.4	Losing Resolution in Properties	21
3.5	Property Slots Merging	25
3.6	Agent behavior with installation tiers and monitoring API	27
3.7	Property Empty Slots with extended end at	37
4.1	FilterModels: rule generation through view helpers.	57
4.2	FilterModels: selecting filters through view helpers.	57
4.3	Installation Filters: Index page.	60
4.4	Installation Filters: Adding a new filter.	60
4.5	Installation Filters: Choosing notification destinations.	63
5.1	Example of chart generated with Gruff	68
5.2	Example of chart generated with Plotr	69
5.3	Example of chart generated with Simeline Timeline	70
5.4	Reports General Structure	71
5.5	Reports: number of users, main view.	85
5.6	Reports: number of users, detailed view.	86
5.7	Reports: notification reach, main report.	86
5.8	Reports: property value evolution.	87

List of Tables

2.1	Rails Directory Structure	9
3.1	Rails to MySQL type and storage size	39
4.1	Rails Plugin Files	43
4.2	Filtering Operations	49
4.3	ActiveRecord Associations	50
4.4	Filtering Operations to SQL Conditions	52
5.1	AllowedResolutions	66

Introduction

This document describes the design and implementation of some of the features to be included in the application *Network Service*. The main project has been developed in collaboration with BitRock S.L.¹.

Independent Software Vendors² need to know about how their applications are being used. What are the environments where they are running? How many users are really using the application? Which exact product versions? Are there any clients near their license limits? Who are the ones doing a more intensive usage? This kind of information is useful in order to make the correct decisions. For example, discontinuing the development on certain platforms if there are no people using them or identifying possible users interested in upgrading their licenses or buying additional services.

Getting this information may be specially problematic in open-source projects where people is be able to download the application code directly. Although some statistics about the number of downloads can be obtained, it will not provide the real usage information. For example, it is not possible to know if the user just tried the application once and removed it.

On the other hand, users need to be alerted when new application updates are available. Specially when there are critical bugs that affects the specific version they are running. ISVs should keep a channel of communication open with all their end-users. Some of them may even end up developing their own notification system to be integrated in their product.

The main objective of *Network Service* is to provide a solution for these common ISV needs. By using this service, they will be able to concentrate in their own product development instead of in secondary tools to monitor or contact their users.

As we have mentioned, in this project we will concentrate in a subset of the features:

¹ <http://bitrock.com>

² Independent Software Vendor (ISV): business term for companies specializing in making or selling software, designed for mass marketing or for niche markets.

- Properties storage system: store gathered metrics from the client machines.
- User filtering capabilities: include ways to define groups of users and to perform targeted operations on them.
- Reports: ability to get statistics about the number of users based on filters and properties as well as to get property value evolution over time.

We will cover the different aspects of the project in the following chapters:

1. First, in the current chapter, we will describe a general overview of Network Service, talking about its main objectives and functionalities. The project scope will be defined based on those general objectives and the project management methodologies will be described.
2. Once the main objectives and working methodologies has been defined, we will start describing the initial environment and technologies of Network Service.
3. Then, the development and evolution of the system to collect and store properties in the server. It will be done as a incremental approach, showing the different steps that have been taken during the process.
4. After that, we will talk about the implementation of a filtering plugin that will be helpful to perform different operations over the groups of users.
5. Finally, we will describe the development of the report generation system.

The Network Service documentation has been also updated in the process to describe all the new features. The latest version of the Network Service User Manual can be found in the Appendix A.

Network Service: general objectives

Network Service main objective is to include any tools that may useful for the software vendors, removing the need of building their own solution from scratch. This will include getting useful information about the application usage, providing a way to send notifications or new updates to the end-users, etc.

We can think of Network Service from different points of view:

Updates and Messages

Every application requires to maintain the communication with the clients. At least, bug fixes and application updates should be notified as fast as possible. Network Service provides a base system that can be easily integrated in the application to send that information as well as any notifications.

Using an external graphical tool or adding some elements in the application interface, the ISV is able to send notifications about their new products and versions directly to the end-users. Filters based on the metrics may be applied to them in order to decide the specific target.

The upgrade process may be controlled by sending the notification to specific groups of users or only to a percentage of them. That way, they may assure there are not important issues in the upgrade process or control the load in their servers.

Marketing Tool

Network Service can provide useful information about the clients: statistics about the number of users, their location, the architectures they are using, etc. This information could be used to analyze the impact of the promotion campaigns.

Application specific information can be also fetched. If the appropriate metrics are defined, this might help to determine the users that can be more interested in license upgrades or additional services.

Some simple tests about the licenses may be also performed. As long as a unique key that identifies the license is sent, they may be able to detect invalid licenses or inadequate usage.

It is also planned to integrate Network Service in with other client relation management tools like SugarCRM.

Additional Services for End-users

There are other needs that usually appear when an end-user works with an application. Network Service plans to make it easier to implement some of them:

- **Monitoring:** track system properties. Generation of alerts to prevent overload on end-user machine. Increased information to support team to deal with certain application issues.
- **Backups:** automated backups of the end-user data.

These services can be an additional income for the ISVs and the efforts to integrate them would be reduced by using a common solution.

Project Scope

During the development of this project, most of the objectives described in 1.1 have been implemented. However, only some of them are part of the current project scope.

We will concentrate in the functionalities related to the installation properties:

- Allow to receive and store the properties from installations in Network Service. It should be able to get the latest properties for a chosen installation as fast as possible and store the historical values. The storage should be restricted in resolution and size. We should minimize the impact of 'Basic' users (the ones without monitoring metrics).
- Add the ability to define installation groups based on the property values. It should be possible to filter all the statistics using those definitions. It should be also possible to perform operations on those specific user sets. For example, to send updates or notifications to one or more of them.
- Generate reports on server side about:
 - Number of Installations: grouped by product, by version or by specific property value. Being able to filter those statistics to a subset of installation groups
 - Notification Reach: describe the number of installations that have received a particular message or update, how much installations are pending to be notified. Furthermore, get that evolution by version, property value or filtered by installation groups
 - Information about the status of a specific installation.
 - Evolution over time for a chosen property with different resolutions and aggregation methods.

Project Management: SCRUM

The project management has been performed following the SCRUM methodologies. SCRUM is an agile software development method for project management. All the features that we would like to have implemented are enumerated and prioritized in a list called *Product Backlog*. That list will be maintained by a person called *Product Owner*.

The work is done in an incremental way: during a 15-30 days period (called *Sprint*) the team will work on a group of stories taken from the *Product Backlog* without external interruptions. Those stories are estimated and chosen by the whole team, in a meeting called *Sprint Meeting* that should be done before each *Sprint*.

In this meeting, the members should estimate the most important stories in the *Product Backlog* by giving them a number of points (those points will be related to the duration or complexity of the task). They have also to decide which will be the speed of the team for the next *Sprint* (taking into account the history of the team in past sprints). That speed will determine the number of points that the team will be able to burn and the stories will be chosen according to it and their estimation. The Team should choose the stories by their own taking into account the priorities

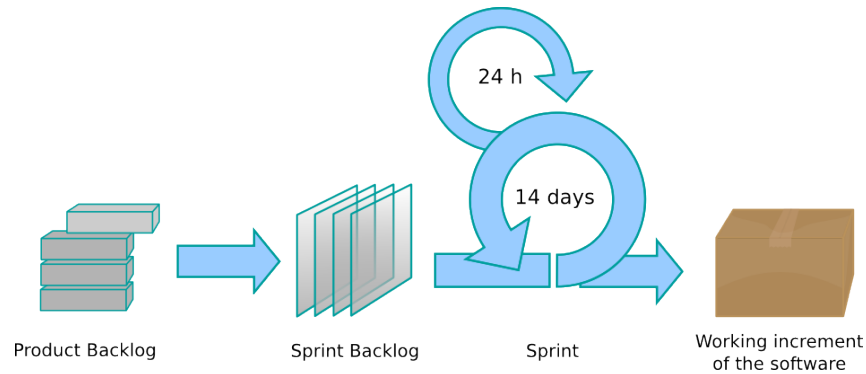


Figure 1.1 Scrum Process.

given by the *Product Owner*, the result is a *Sprint Backlog* with all the things that are supposed to be done on the next *Sprint*.

During the sprint there are short daily meetings (5-10 minutes) in which each one of the team members gives a short explanation about:

- What was done yesterday
- What will be done today
- Comment any problems that have appeared

Daily and Sprint Meetings should be coordinated by the SCRUM master, who is a member of the team elected by the rest of them.

After the sprint is finished there should be possible to do a demo with the latest implemented features. Product Owner should be present on it and should give any comments about the results obtained.

It is important to have a Retrospective Meeting after each Sprint, to analyze the things that have gone wrong and those that have been good. That way, we are able to improve the productivity by fixing the possible problems and adapting the SCRUM methodology to our own team.

All the meetings have a fixed length and should start at a specific time. It is also important to talk only about what the meeting is supposed to be about.

One of the most important benefits of this method is that the project is able to adapt its objectives to the needs of the market faster than other solutions. It is also important that the client gets involved in the evolution of the application. He can see how the product is growing and if it is what he really wants to get before it is finished. This avoids misunderstandings before it is too late.

Additional details about the involved process can be consulted at [10].

Environment and Technologies

We will start by studying the initial status of the application giving a short description of how it is organized and about the main technologies involved in Network Service at the beginning of the current project.

Technologies

Network Service is based on *Ruby* programming language and uses the framework *Ruby on Rails*. The application code tests are written as behavior specs, using *RSpec*.

On server side, *Nginx* is used as proxy balancer between a set of *Mongrel* processes. *MySQL* is the main database. All the processes are monitored by *god.rb*. Deployment is designed via *Capistrano*.

In the following sections we will give some short descriptions about them.

Ruby

Ruby is an object-oriented interpreted scripting language distributed under MIT license. It was developed by Yukihiro Matsumoto, being released in 1995. It has been designed to be clean and intuitive, with simple syntax and following a natural language.

When writing Ruby code, you should follow some simple conventions:

- Indentation size is two spaces and spaces are preferred to tabs.
- Naming conventions:
 - `CONSTANTS_USE_UPPER_CASE`, for example `Math::PI`
 - `ClassesAndModuleNamesUsePascalCase`, joining words in the name capitalizing the first letter of each word.
 - `method_names, local_variables, @instance_variables` and `@@class_variables` use lowercased and underscored names, joining words with underscores. For example: `some_variable`, not `someVariable` or `somevariable`.

- Keep acronyms in class names capitalized. `MyXMLClass`, not `MyXmlClass`. Variables should use all lower case
- Method definitions should include parentheses and no unnecessary spaces: `MyClass.my_method(my_arg)` not `my_method(my_arg)` or `my_method my_arg`
- Put parentheses around non-trivial parameter lists but without spaces after the method name: `method(params)`, not `method (params)`.
- Use curly braces for single-line blocks, use do-end for multi-line blocks.

More information can be found at [3], [4], [5] and, <http://ruby-lang.org/>.

Ruby on Rails

Rails, also known as Ruby on Rails or RoR, is an open source web development framework written in Ruby. It was created in 2003 by David Heinemeier Hansson and has since been extended by the Rails core team, with more than 1,400 contributors.

Rails simplifies the process of writing new applications by making assumptions about what developers need to do and how it should be done, encouraging people to use specific code organization. This kind of approach speeds up the development process and help developers to easily understand applications implemented by other teams.

Some of the principles used in Rails are:

- Favor Convention Over Configuration
- DRY (Don't Repeat Yourself): avoid writing same code over and over.
- REST: organize application around resources and standard HTTP verbs.
- MVC: follow model view controller architecture

MVC Architecture

Rails uses the MVC (Model, View, Controller) architecture. That is, we have three kinds of elements to organize the code:

- The **model** represents the information and the rules to manipulate it. It manages the interaction with database tables. In Rails, each model usually describes the data inside one table.
- **Views** construct the user interface, providing the tools to represent the application data in different formats (XML, HTML...).
- **Controllers** represents the connection between models and views. They are responsible for processing the incoming requests, using models to update or get the information and sending them to specific views to generate the response to them.

MVC provides isolation of business logic from the user interface and makes easier the maintenance, being helpful to know where to locate problems in the code.

Components of Rails

Rails has the following set of modules:

- **Railties:** it's the core of the framework, connecting the different components.
- **Action Controller:** manages the controllers, processing the requests coming to the application.
- **Action View:** manages the views, creating HTML and XML outputs by default, having built-in AJAX support and providing template rendering tools.
- **Active Record:** it provides models to the application, giving database independence and tools to configure the relations between models.
- **Action Mailer:** for building email services. It has the tools to construct, send emails as well as to receive and process them.
- **Active Resource:** manages the connection between business objects and RESTful web services.
- **Active Support:** extensive collection of utility classes and standard Ruby library extensions that are used in the Rails

File Structure

Rails defines a specific structure of files that should be used in any project. It can be seen in table 2.1.

Further details about Rails and common usage examples can be found at [1], [6] and, <http://rubyonrails.org/>.

RSpec

RSpec is a Behavior Driven Development framework for Ruby. It provides two frameworks for writing and executing examples about how the application should behave:

- a Story Framework for describing behavior at the application level
- a Spec Framework for describing behavior at the object level

More information about RSpec can be found at [13] and <http://rspec.info/>.

Table 2.1 Rails Directory Structure.

<code>/app/models</code>	The models subdirectory holds the classes that model and wrap the data stored in our application's database.
<code>/app/controllers</code>	contains the files with the controller classes. Those controllers will deal with the user requests.
<code>/app/views</code>	The views subdirectory holds the display templates to fill in with data from our application, convert to HTML, and return to the user's browser.
<code>/app/views/layouts</code>	Holds the template files for layouts to be used with views. This models the common header/footer method of wrapping views. In your views, define a layout using the <code>layout :default</code> and create a file named <code>default.rhtml</code> . Inside <code>default.rhtml</code> , call <code><% yield %></code> to render the view using this layout.
<code>/app/helpers</code>	The helpers subdirectory holds any helper classes used to assist the model, view, and controller classes. This helps to keep the model, view, and controller code small, focused, and uncluttered.
<code>/components</code>	This directory holds components tiny self-contained applications that bundle model, view, and controller.
<code>/config</code>	This directory contains the small amount of configuration code that your application will need, including your database configuration (in <code>database.yml</code>), your Rails environment structure (<code>environment.rb</code>), and routing of incoming web requests (<code>routes.rb</code>). You can also tailor the behavior of the three Rails environments for test, development, and deployment with files found in the <code>environments</code> directory.
<code>/db</code>	Usually, your Rails application will have model objects that access relational database tables. You can manage the relational database with scripts you create and place in this directory.
<code>/doc</code>	Ruby has a framework, called RubyDoc, that can automatically generate documentation for code you create. You can assist RubyDoc with comments in your code. This directory holds all the RubyDoc-generated Rails and application documentation.
<code>/lib</code>	You'll put libraries here, unless they explicitly belong elsewhere (such as vendor libraries).
<code>/log</code>	Error logs go here. Rails creates scripts that help you manage various error logs. You'll find separate logs for the server (<code>server.log</code>) and each Rails environment (<code>development.log</code> , <code>test.log</code> , and <code>production.log</code>).
<code>/public</code>	Like the public directory for a web server, this directory has web files that don't change, such as JavaScript files (<code>public/javascripts</code>), graphics (<code>public/images</code>), stylesheets (<code>public/stylesheets</code>), and HTML files (<code>public</code>).
<code>/script</code>	This directory holds scripts to launch and manage the various tools that you'll use with Rails. For example, there are scripts to generate code (<code>generate</code>) and launch the web server (<code>server</code>).
<code>/spec</code>	Rspec folder containing the specs for model, views controllers and libraries. Fixtures can be defined as well.
<code>/test</code>	The tests you write and those Rails creates for you all go here. You'll see a subdirectory for mocks (<code>mocks</code>), unit tests (<code>unit</code>), fixtures (<code>fixtures</code>), and functional tests (<code>functional</code>).
<code>/tmp</code>	Rails uses this directory to hold temporary files for intermediate processing.
<code>/vendor/plugins</code>	Libraries provided by third-party vendors.
<code>/vendor/rails</code>	If it exists, contains a frozen version of rails than will be used when the project runs.

Nginx

Nginx is a open-source, high-performance HTTP server and reverse proxy, as well as an IMAP/POP3 proxy server. It was written by Igor Sysoev in 2005 and it's known for its stability, rich feature set, simple configuration, and low resource consumption.

More information can be found at [11] and <http://nginx.org/>.

MySQL

The MySQL software delivers a very fast, multi-threaded, multi-user, and robust SQL (Structured Query Language) database server. MySQL Server is intended for mission-critical, heavy-load production systems as well as for embedding into mass-deployed software. MySQL is a registered trademark of Sun Microsystems, Inc.

More information can be found at [?], [2] and, <http://www.mysql.com/>.

Capistrano

It's a tool for automating tasks on one or more remote servers. It executes commands in parallel on all targeted machines, and provides a mechanism for rolling back changes across multiple machines.

- Great for automating tasks via SSH on remote servers, like software installation, application deployment, configuration management, ad hoc server monitoring, and more.
- Easy to customize. Its configuration files use the Ruby programming language syntax, but you don't need to know Ruby to do most things with Capistrano.
- Easy to extend. Capistrano is written in the Ruby programming language, and may be extended easily by writing additional Ruby modules.

More information can be found at [14] and, <http://capify.org/>.

God.rb

It's a simple and powerful monitoring framework written in Ruby. We use god.rb to control all the servers/processes in production server.

- it is controlling all servers states stop/start/restart.
- it is able to detect if the process went down and then start it once again
- it is restarting servers when their memory/cpu usage exceed previously defined limits

It will be running as a daemon continuously in production server to control all the process that should be working.

More information can be found at <http://god.rubyforge.org/>.

RESTful Design

The application has been designed following the REST principles:

- Application state and functionality are abstracted into resources
- Every resource is uniquely addressable using a universal syntax for use in hypermedia links
- All resources share a uniform interface for the transfer of state between client and resource, consisting of
 - A constrained set of well-defined operations
 - A constrained set of content types, optionally supporting code on demand
- All resources share a uniform interface for the transfer of state between client and resource, consisting of
 - A constrained set of well-defined operations
 - A constrained set of content types, optionally supporting code on demand

More information about REST approach can be seen at [.](#)

Initial Data Structure

The information is stored as different resources. We have *products* that represent the different applications that the ISV provides. Each of them has a set of *versions*. A version contains an unique identifier, called *product_guid*, that will be used to determine the version for each installation.

On the other hand, we have *installations* that always belongs to a specific version. Every installation has its own identifier, called *installation_guid*, that will be generated during the installation process in the end-user machine.

For messaging system the application defines *messages* and *updates* both of them have a related *notification* that contains the information that will be sent to the user. Each notification has a set of versions as destinations that can belong to different products and a time interval in which it will remain active. Updates have additional information about the location where the file can be found.

Both installation and version identifiers must be provided by the end-user installation each time it contacts with the server to check for new messages and updates. On each of those contacts an *event* will be generated with the information about the request. A new installation will be generated when necessary.

We have also a *status* resource that is useful to setup the visibility or activation for different elements: products, versions and, notifications. That way, we can disable any of those elements without losing its related information in the server.

To control the access to the frontend, the server has *accounts* and *roles* (added using the plugins: RESTfulAuthentication and ACL2System). Those elements allow us to have different views and permissions taking into account the roles of the network service user that is logged in.

Finally, we have *settings* that define general configuration values. They consist in a set of key-value pairs.

A diagram describing these relations can be seen in 2.1.

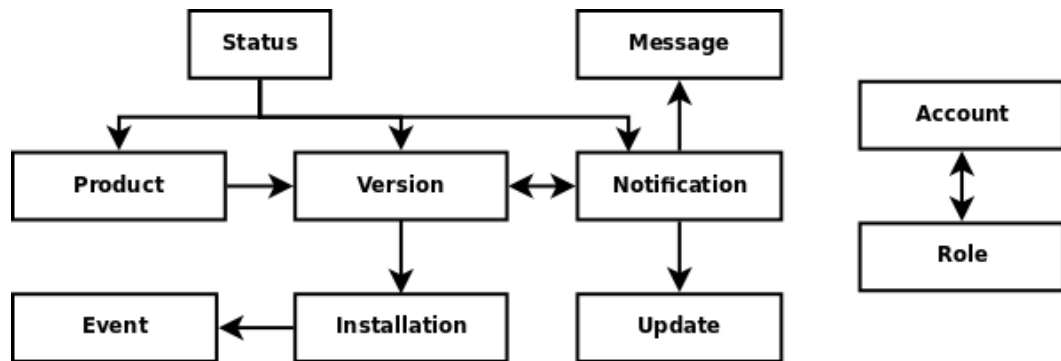


Figure 2.1 Initial Data Structure.

Properties Storage System

In this chapter, we will describe the implementation of the properties storage system in Network Service. It should include the reception, storage and any related processing of the metric values received from application installations.

We will describe the development by functionality increments, similar to the approach followed during the implementation. We will start with simple modifications to get static values. Then, we will add properties history. And finally, some different scalability and database improvements will be added.

Receiving Metrics. Simple storage system

We need a way to receive the property values. For doing so we will take advantage of an already implemented feature in Network Service: sending application updates and messages. In order to get these notifications, the application is using a simple API that will reply to HTTP requests done from the end-user machine.

We will need to modify the API so it is able to receive the values. On the other hand, the data structure should be altered to keep them in the server database. For now, we will only store the latest values as installation properties.

Short API description

Initially, the API is implemented as part of the Rails application. An specific controller, called *AgentController*, configures all the possible requests. The API is designed to use regular HTTP protocol, encoding the required parameters in URL format. It includes three GET methods.

- `messages`: returns the list of messages assigned to the application version related to the parameter `product_guid` received.
- `updates`: returns the updates.

- **news**: returns single notification (either a message or an update) that has been setup as default for the product.

The last method is useful for applications without ability to deal with more than one notification. This might make sense if we want to simplify the integration in the application GUI on client side.

All the queries to the server must contain the product and installation identifiers related to the application version and the specific installation. SSL communication is also forced and the certificate validated to assure the notifications come from the correct server.

An example query looks like:

```
GET https://server_name/api/messages? installation_guid =<guid_value>&
product_guid=<product_guid>
```

API responses are always encoded in XML format. Some example responses can be seen below.

- Response to GET messages:

```
<?xml version="1.0" encoding="UTF-8"?>
<messages>
  <message>
    <guid>7</guid>
    <title>Version 11.0 coming soon</title>
    <description>It will be available in a few days</description>
    <critical>false</critical>
    <created_at>Wed Jun 04 15:29:20 +0200 2011</created_at>
    <updated_at>Wed Aug 27 19:35:19 +0200 2011</updated_at>
    <start_date>2011-06-04</start_date>
    <finish_date>2011-06-12</finish_date>
    <webpage_url>http://acme.bitrock.com/news</webpage_url>
    <webpage_url_description>more info</webpage_url_description>
  </message>
  <message>...</message>
  ...
</messages>
```

- Response to GET updates

```
<?xml version="1.0" encoding="UTF-8"?>
<updates>
  <update>
    <update_url>
      http://www.acme.bitrock.com/acme_product_10.5.bin
    </update_url>
    <guid>8</guid>
    <title>Acme Product 10.5 Released</title>
    <description>New version is available</description>
    <critical>false</critical>
    <created_at>Thu Jul 31 23:10:25 +0200 2011</created_at>
    <updated_at>Wed Aug 27 19:45:16 +0200 2011</updated_at>
    <start_date>2011-05-14</start_date>
    <finish_date>2011-09-26</finish_date>
```

```

<webpage_url>http://www.acme.bitrock.com</webpage_url>
<webpage_url_description />
<update_type>bin</update_type>
<update_os>linux</update_os>
<update_size>100</update_size>
<update_md5>9ba9801841f5fa941420f0af90e20634</update_md5>
<update_cmd_switch/>
<update_instruction >
  Download the file and execute it as root
</update_instruction >
</update>
<update> ... </update>
...
</updates>

```

- Response to GET news:

```

<?xml version="1.0" encoding="UTF-8"?>
<answer>
  <status >1</status >
  <title >New Update available</title >
  <message>There is an important update of your application</message>
  <url> http://www.acme.bitrock.com/update/094</url>
  <url_description >Get the update here</url_description >
</answer>

```

- Response to GET news when there is no data:

```

<?xml version="1.0" encoding="UTF-8"?>
<answer>
  <status >0</status >
  <title />
  <message/>
  <url />
  <url_description />
</answer>

```

Initial Properties Storage

The data structure needs to be changed to store the properties that we are going to receive from client machines.

We will have different formats for properties. For instance, we can store strings, booleans, integers, floats, etc. To deal with them, `Property Type` resource is added. Basically, its functionality is to define the different property keys used and bind them to one of the possible formats. Those property types will belong to one or more products and their key should be unique through all of them - to allow cross-product analysis based on common properties).

Another resource is included to store the received values for each property: `Installation Property`. Each installation property will belong to a property type and to an installation. As we are using the same database table to contain properties with different property types we would need a variable column type which is not possible in MySQL. To work around this, we could store the properties serialized as

string or text and transform the values to their correct format before performing any operation.

However, this would impact the performance since any analysis would be much faster if we are able to use database aggregation methods. This can be done by having separate columns for each of the allowed formats. It will not mean a big increase in the size of the database since all the unused fields will be set to NULL. As long as InnoDB storage engine is used, this means an additional bit per unused format and table row[2]. Taking that into account, the fields are defined as <format>_value. And, in order to get or set the values it will be necessary to take into account the information of format contained in the associated PropertyType.

Finally, we will include a similar resource called Event Property with similar structure to installation properties. It will keep information related to the API event though. Event resources are generated on each API request containing the details about it. By adding the property values, we get some kind of log with the evolution in the property values. However, this has strong limitations since events table will become huge over time and we are not forcing any resolution or time restrictions.

A more detailed description can be seen in the following Rails migration:

```
class InitialPropertiesSetup < ActiveRecord::Migration
  create_table "property_types", :force => true do |t|
    t.string "key" #os, lang, ...
    t.string "value_type" #integer, string, ...
    t.timestamps
  end

  create_table "event_properties", :force => true do |t|
    t.integer "event_id"
    t.integer "property_type_id"
    t.boolean "boolean_value"
    t.integer "integer_value"
    t.float "float_value"
    t.string "string_value"
    t.text "text_value"
    t.datetime "datetime_value"
    t.binary "binary_value"
    t.timestamps
  end
  add_index "event_properties", ["event_id"],
    :name => "index_event_properties_on_event_id"
  add_index "event_properties", ["property_type_id"],
    :name => "index_event_properties_on_property_type_id"

  create_table "installation_properties", :force => true do |t|
    t.integer "installation_id"
    t.integer "property_type_id"
    t.boolean "boolean_value"
    t.integer "integer_value"
    t.float "float_value"
    t.string "string_value"
    t.text "text_value"
    t.datetime "datetime_value"
    t.binary "binary_value"
    t.timestamps
  end
  add_index "installation_properties", ["installation_id"],
    :name => "index_installation_properties_on_installation_id"
  add_index "installation_properties", ["property_type_id"],
```

```

:name => " index_installation_properties_on_property_type_id "
end

```

Once these modifications are applied, the data structure seen in 2.1 will be modified, taking the form shown in Figure 3.1.

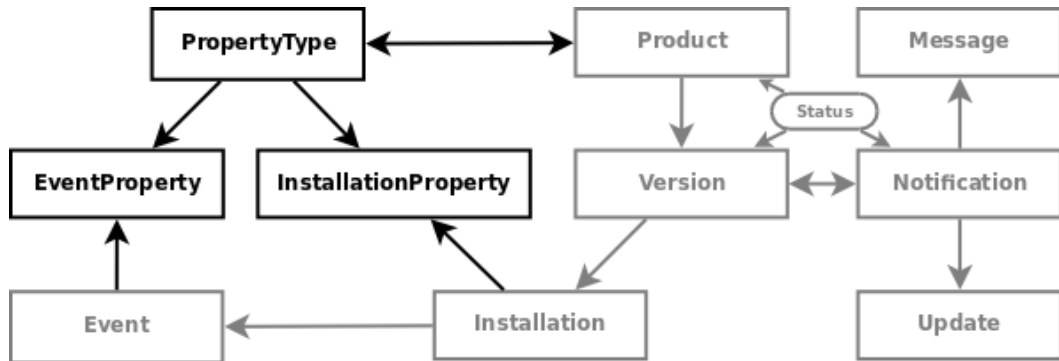


Figure 3.1 Data Structure with Simple Properties.

API Modifications

The information will be sent to the server during the client requests asking for new notifications. Following the HTTP approach, we include a new URL-encoded parameter: `properties`. It will be present in all the described methods in the section 3.1.1 (updates, messages and news). The properties will be sent as a set of key-value pairs where `key` will be a unique identifier for each property.

With this change, the requests will look like:

```

GET https://server_name/api/messages? installation_guid =<guid_value>&
product_guid=<product_guid>&properties[<key1>]=<value1>&
properties [<key2>]=<value2>&...

```

To store the incoming properties we should modify the `AgentController` in order to process this new parameter. However, it is important to minimize the impact in the response time in the API. That means that any processing related to properties should be done after sending the response to the user.

We modify the method which creates the API event resource so it generates all `EventProperty` objects. Rails maps the parameters received into a `Hash` simplifying the process to look for each pair key-value for the associated `PropertyType` based on the key, and insert the property value in the correct column based on it.

Once event properties are modified, a similar task should be done over installation properties which are just a copy of the latest event properties in this first approach.

Properties Storage with History

We should improve the way we are storing the properties. Until now, we have been only storing the historical values of the properties as event properties. That way of storing properties has some inconveniences:

- Storage size has no limitations.
- Maximum resolution is not forced. All properties get saved every time a request is received.
- Events table size makes any report generation really slow.
- Events and information related to them should only contain temporary information.

Taking that into account we should find a way to

- Force the maximum resolution for each property.
- Decrease this resolution as the data is getting older. We are more interested in the most recent data.
- Simplify the process to get all properties at a given time.
- Treat all properties in a similar way to simplify the code.
- Use the less disk space as possible specially for users without monitoring functionality.

Some of these points sounds similar to the concepts used in Round Robin databases (*RRDtool*¹). However, using this tool might cause some problems:

- It is designed to store numerical data. In our case, we want to store any kind of information. In other words, we would need to find a way to store other formats on a different storage system increasing the complexity.
- Constant database size may be seen as an advantage. However, that would mean that any installation that contacts the server would cause a disk region to be reserved. If that installation has been just a test, or the user just wanted to try the application or it is upgraded to a newer version, we are losing a space which will never be used.
- Unsupported by *Active Record*². This means we will have limitations to filter installations over time based on *RRDtool* databases generated for each property. The main problem is that it will be required to write additional code to allow the interaction between *Active Record* models and *RRDtool*.

¹ <http://www.mrtg.org/rrdtool/>

² Ruby on Rails database connector, check section 2.1.2.2.

- Difficult migration to a different database: SQL is fully supported in most of the databases and it would be really easy to perform a migration to a new engine like PostgreSQL if our database is SQL compatible.
- Advanced scalability configurations are limited. For example, it would not be possible to use MySQL's master-slave configuration.

Because of these drawbacks, we finally decided to integrate the improvements in our MySQL models. We will apply some of RRDtool concepts though.

Time Slots

In order to store the time information we will use a structure based on slots. Every time a value is stored it will represent the value for a period of time or time slot. Those slots will have a specific minimum size configured in the property type. For example, if we have a property with resolution of one day and we have received the value "linux" at "12:32:12 12/02/2011", then the stored value between "00:00:00 12/02/2011" and "00:00:00 13/02/2011" for that property will be "linux".

With that slot structure we are forcing a specific minimum resolution and we avoid storing more entries than we really need for each specific property. Using slots adapts the storage to the data that will be shown to the user.

If more than one value is received in the same slot, aggregation methods are applied. In other words, the stored value will become the average, maximum, minimum, sum, first received, last received value in those cases. This aggregation will be configured as a property type parameter too. To perform the average aggregations we will need to store a counter of the number of values that has been received in a specific slot.

When we are talking about resolutions we are talking about a pair <number>-<time period>. For example, a possible resolution can be 1-day, that will mean that minimum slot size will be 1 day long. We restrict the allowed resolutions to a subset of combinations to simplify the conversions between them: 5 min, 10 min, 15 min, 30 min, 1 hour, 3 hours, 6 hours, 12 hours, 1 day, 1 month, 1 year. We will need the following fields to be added to property types: `resolution`, `resolution_units` and `aggregation_method`.

A small diagram of how the slots are generated can be seen at 3.2, where *V* represents the value and *C* the counter of the slot.



Figure 3.2 Property Time Slots Generation.

As we will see later, those slots will not have a constant size. That's why we should store somehow when it starts and how long it is, or the start and end timestamps. In our case, we have taken the second option storing two new fields for each installation property: 'start_at' and 'end_at'.

It will be also necessary to include a boolean, called `latest`, that will be used to index the latest properties received. This will help to speed up all the algorithms using the latest installation properties.

Since we don't need to access event properties once they get processed, we will replace event properties resource with a serialized field in events table. We keep this properties in the events as they come from the client because it opens the possibility of processing the events asynchronously.

Empty Slots

Imagine that we want to get the properties at a specific time. What happens if for an installation there's no slot at the time? We would need to look for the latest received value before that specific time, making the queries more complex.

In order to avoid that, we will add slots without received data that we will call empty slots. In those slots the value will take the latest value received before it has its own. The way to distinguish that those values are not real but extended from the previous one is that the counter takes the value 0 (no values have been merged to that slot).

An empty slot will be added after the latest slot generated with `end_at` value set to nil. That way we can consider that slot as an empty one without end. This avoids having to generate that empty slot if there's no data coming from the installation, otherwise we would need to check that a slot time has passed and no data has been received for each installation and property what would be a time consuming operation

In the figure 3.3 a small example of how are they added can be seen. *V* represents the value and *C* the counter of the slot.

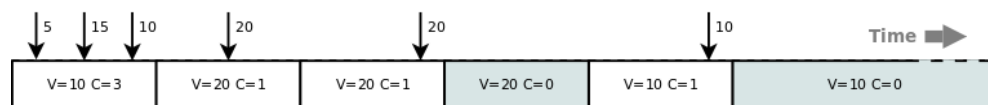


Figure 3.3 Property Empty Slots Generation.

Losing Resolution

The most interesting data for the user is the most recent. If we want to have a high resolution we will need a huge amount of data to be stored and the user will not need so much detail about the past. The best way to deal with that kind of scenario is to modify the resolution over time.

We define different regions that are common through all property values. Those regions will be taken into account how old a slot is. In other words, we will say that a property value (or slot) is inside a particular region if its `end_at` time is older than the start time of the region.

The maximum possible resolution is defined for each of those regions. The aggregation method configured in the related property type will be used to increase the size of the slot by merging it with adjacent ones.

In our case, we have configured it as a Hash which keys are the number of days that should pass to get into that region and the values are the maximum resolutions on them:

```
REGIONS = {2 => [1, :hour], 30 => [1, :day]}
```

If we configure a final region without resolution. The slots will be growing until that point, storing only one value in that final region.

To control how the regions are processed, we will mark the different slots with an integer that will represent the current region. That number will take the value of the days ago where the region is started.

A small example about the behavior can be seen in the figure 3.4. The numbers inside the slots represents (value,counter).

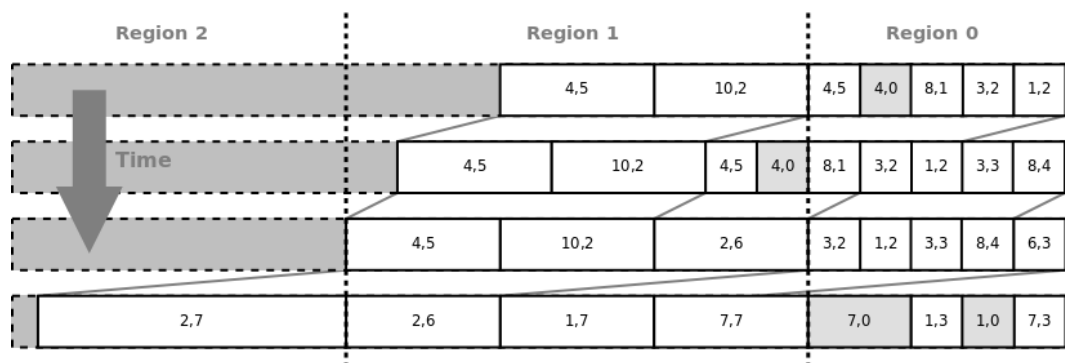


Figure 3.4 Losing Resolution in Properties.

Database Migration

The migration including the described changes can be seen below:

```
class AddHistoryToInstallationProperties < ActiveRecord::Migration
  def self.up
    # Modify the fields on the tables
    add_column :property_types, :resolution, :string
    add_column :property_types, :resolution_units, :integer
    add_column :property_types, :aggregation_method, :integer
```

```

add_column : installation_properties , : latest , : boolean ,
  : default => false
add_column : installation_properties , : start_at , : datetime
add_column : installation_properties , : end_at , : datetime
add_column : installation_properties , : counter , : integer ,
  : null => false , : default => 0
add_column : installation_properties , : resolution_region , : integer ,
  : null => false , : default => 0

add_index : installation_properties , [: latest ],
  : name => " index_properties_latest "
add_index : installation_properties , [: resolution_region ],
  : name => " index_properties_region "
add_index : installation_properties , [: start_at , : end_at ],
  : name => " index_properties_slots "
add_index : installation_properties , [: property_type_id ],
  : name => " index_properties_type "

# Reload schema information
InstallationProperty . reset_column_information
PropertyType . reset_column_information

# Serialize all event properties into properties field
puts " --- Serializing event properties ... "
offset = 0
while( evts = Event . find ( : all , : include => : event_properties ,
  : limit => 1000 , : offset => offset ))
  evts . each do |e|
    properties = {}
    e . event_properties . each do |ep|
      properties [" ep . property_type . key " ] =
        ep . send ( ep . property_type . value_type )
    end
    e . properties = properties
    e . save
  end
  offset += 1000
end
puts " --> Done "

# Generate historical values in installation properties
puts " --- Modifying properties to use time slots ... "
InstallationProperty . destroy_all
PropertyType . update_all ( [
  " resolution_units = 1 , resolution = ' day ' , aggregation_method = ? " ,
  PropertyType :: AVERAGE_VALUE
] )
Installation . process_events : no_time_limit => true
puts " --> Done "
end

def self . down
  remove_index : installation_properties , : name => " index_properties_latest "
  remove_index : installation_properties , : name => " index_properties_slots "
  remove_index : installation_properties , : name => " index_properties_type "
  remove_index : installation_properties , : name => " index_properties_region "

  remove_column : property_types , : resolution
  remove_column : property_types , : resolution_units
  remove_column : property_types , : aggregation_method

  remove_column : installation_properties , : latest
  remove_column : installation_properties , : start_at
  remove_column : installation_properties , : end_at
  remove_column : installation_properties , : counter
  remove_column : installation_properties , : resolution_region

```

```

InstallationProperty .reset_column_information
PropertyType .reset_column_information
end
end

```

Processing Algorithms

We need to also implement the algorithms to make the properties behave as described above. As we have mentioned before, the properties are processed by a worker in an asynchronous way. It will take the task to process the properties that are serialized, get the event and call its method `process_properties`.

First, the `country` and `host` will be added to the list of properties if the installation IP has changed. `Host` will be setup with the IP as value, being processed in a separate cron task in order to save time while processing the properties. The result of that method is that the serialized properties in the event are inserted in the installation properties structure, following the rules that have been described above.

For each of those properties the process is:

- Get the boundaries of the slot where we are going to insert the data, based on the property type resolution configuration.
- Get the latest slot for that property.
- If there isn't a latest slot, we generate a new slot with the value received and a blank slot that will start after it and has no end time.
- Else if the creation time of the event is inside that last slot, the value is merged, the counter increased and the final empty slot updated.
- Else if creation time is newer, create a new latest slot, add an empty slot between the latest and the new one if necessary, and updated the final empty slot.
- Else, (This case should not be reached if properties are processed chronologically):
 - Look for the slot in the creation time
 - If it exists merge the value and update the blank slot after it if any
 - Else generate the slot and add or modify the adjacent empty slots

That insertion process is able to generate new slots based on the incoming events. However, we have not yet implemented anything about losing resolution. That second processing will be done by as a daily cron task, executing the method `InstallationProperty.process_regions`.

For each region we get those properties that are outside their region, taking them in groups of a constant number of properties to avoid high resource requirements.

We will go through all of them which has passed from one region to the next one, checking if they are long enough for the new region (It's possible that a constant value has caused a longer slot), if not it will be merged with those properties that are inside the new slot limits. All the changes are performed in memory and then written to database together as a transaction for each group of properties taken from memory.

Improved Properties Storage

Once the basic behavior is in place, we are going to improve some aspects related to the disk usage and the API response times.

Installation Tiers

We include the concept of installation tiers in Network Service. They will allow us to distinguish between different functionality levels. Each installation tier has its own list of metrics to be sent. These levels should be configured on client machine, maybe modified based on the product license.

For example, the bigger installation tier will allow sending memory, cpu load and disk usage. The ISV may decide to only monitor them for specific clients that have purchased an extended license.

We decide to use three functionality levels:

- Basic** Default tier. Installations don't send any monitoring information, only the default properties. They only have access to the `news` method in the API, getting only short notifications. Small contact rate.
- Professional** They have some additional services, being able to get the full list of notifications and updates. They will send any additional application properties along with the default ones. They contact more often with the server.
- Monitoring** Includes monitoring metrics. They will do a bigger number of requests to the API in order to update the monitoring information.

Most of users will belong to Basic or Professional tier, but they have the lower priority from the ISV's perspective. For that reason, we should concentrate on minimize the impact of them in the system by providing a storage engine that would reduce the costs derived from them.

Merge Property Slots

As we have said previously, we should find ways to minimize the disk space used by those installations without monitoring information ('Basic' installations).

If we take a look at the metrics that we are storing in the database, we can see that most of them remain in specific values for long periods of time. In this context, having multiple slots of time to store the same value is not efficient. For that reason we are going to alter our storage algorithm to only store the changes as new slots.

The idea is to merge the slots that are together and share the same value. That way, instead of having a lot of adjacent slots storing the same value, we will have one slot that will be growing until there's a change in the received value.

An example of how the slots will be merged can be seen in the figure 3.5, V represents the value and C the counter of the slot.

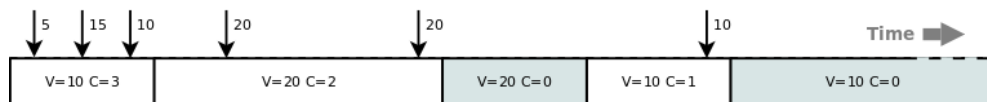


Figure 3.5 Property Slots Merging.

This will also reduce the required processing about regions, since they may be big enough already to pass the test of the maximum resolution when the region limit is reached. This avoids further unnecessary processing.

The algorithm described in 3.2.5 is modified as follows:

- Get the boundaries of the slot where we are going to insert the data, base on the property type resolution configuration.
- Get the latest slot for that property
- If there isn't a latest slot, we generate a new slot with the value received and a blank slot that will start after it and has no end time.
- Else if the creation time of the event is inside that last slot, the value is merged, the counter increased and the final empty slot updated
- Else if creation time is newer:
 - **If the latest slot has the same value as previous one, merge them**
 - Create a new latest slot.
 - Add an empty slot between the latest and the new one if necessary.
 - Update the final empty slot.
- Else, (This case should not be reached if properties are processed chronologically):
 - Look for the slot in the creation time
 - If it exists merge the value and update the blank slot after it if any

- Else generate the slot and add or modify the adjacent empty slots

Regions processing should be also changed: after the new slot is generated it will be checked if it can be merged with the previous one (if they have the same value).

API Improvements

Add Notifications and Monitoring Methods

One of the first things we can take into account to reduce the number of requests to the API is that Professional and Monitoring installations will ask for both messages and updates one after the other. That's why it's interesting to modify the API specification to add a common method that will return both. The method will be called *notifications* and the response will take the syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
< notifications >
<message>
  <guid>7</guid>
  < title >Version 11.0 coming soon</ title >
  < description >It will be available in a few days</ description >
  < critical >false </ critical >
  < created_at >Wed Jun 04 15:29:20 +0200 2011</created_at>
  < updated_at >Wed Aug 27 19:35:19 +0200 2011</updated_at>
  < start_date >2011-06-04</start_date>
  < finish_date >2011-06-12</finish_date>
  < webpage_url >http://acme.bitrock.com/news</webpage_url>
  < webpage_url_description >more info</ webpage_url_description >
</message>
<message> ... </message>
...
<update>
  < update_url >
    http://www.acme.bitrock.com/acme_product_10.5.bin
  </ update_url >
  <guid>8</guid>
  < title >Acme Product 10.5 Released</ title >
  < description >New version is available </ description >
  < critical >false </ critical >
  < created_at >Thu Jul 31 23:10:25 +0200 2011</created_at>
  < updated_at >Wed Aug 27 19:45:16 +0200 2011</updated_at>
  < start_date >2011-05-14</start_date>
  < finish_date >2011-09-26</finish_date>
  < webpage_url >http://www.acme.bitrock.com</webpage_url>
  < webpage_url_description />
  < update_type >bin</update_type>
  < update_os >linux</update_os>
  < update_size >100</update_size>
  < update_md5 >9ba9801841f5fa941420f0af90e20634</update_md5>
  < update_cmd_switch />
  < update_instruction >
    Download the file and execute it as root
  </ update_instruction >
</update>
<update> ... </update>
...
</ notifications >
```

The only change needed on client side is to modify the agent to send only one request directed to *notifications* with the same parameters that were used before.

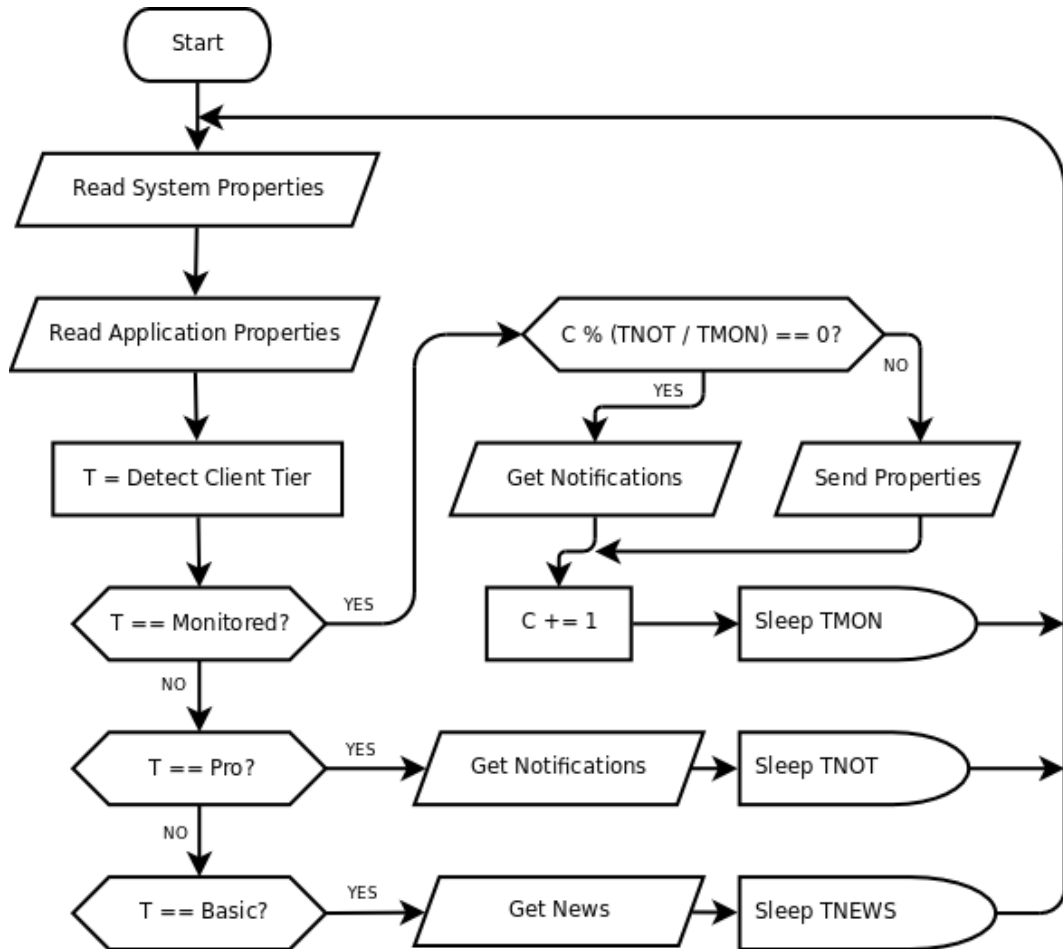


Figure 3.6 Agent behavior with installation tiers and monitoring API.

On the other hand, it is not really necessary to look for the notifications at the same ratings as the monitoring properties should be gathered. For that reason, we include a new method called *monitoring* that only gets the property values and returns the "200 OK" HTTP response. That reduces the load on the server side since notification filters are not taken into account decreasing the number of database operations required.

The figure 3.6 shows a simple schema about the installation agent behavior based on the installation tier and including the new API methods.

Avoid Sending Duplicated Notifications

We can also reduce the traffic in the API easily by not sending the notifications that the user already has. To do that, we should send through the API the list of identifiers of those messages and updates that are stored in the client machine. That way, we will be able to send only those that are new to the user.

For simplicity, we will send the list as another installation property, `stored_notification_ids` in the list of parameters:

```
GET https://server_name/api/messages? installation_guid =<guid_value>&
product_guid=<guid2>&properties[key1]=value1&properties[key2]=value2&
properties [ stored_notification_ids ][]=1&
properties [ stored_notification_ids ][]=2...
```

Obviously, it will not be required to store the received value on the server side. By just not defining an associated property type, we make the application ignore them. We will check that variable before sending any notification to the end-user, removing those that should not be resent.

Add Background Processing

In order to answer the client in the shorter time possible, we should modify the Agent controller methods to not work on anything that is not indispensable to generate the response. In other words, we should do any processing unrelated to the response (like updating the property history) asynchronously.

Each time a request is received we will create an event resource with all the information received. The identifier will be then appended to the list of events to be processed. Finally, a separate process will take it from the list and do any pending processing.

We will use `Beanstalkd`³ as the queuing system. It is a fast, distributed, in-memory workqueue service developed by *Philotic, Inc* to improve the response times in *Facebook*. Its interface is generic, but was originally designed for use in reducing the latency of page views in high-volume web applications by running most time-consuming tasks asynchronously. Its integration with Ruby can be done easily using `beanstalk-client` gem⁴.

It works as a daemon. Once it is started, it will be listening in a specific port waiting for requests for putting and getting jobs from different queues.

A simple example about how a message would be inserted in the queue through the previously mentioned gem looks like:

```
beanstalk = Beanstalk::Pool.new(['10.0.1.5:11300'])
...
beanstalk.put('hello')
```

Meanwhile, the consumer process will look like:

```
beanstalk = Beanstalk::Pool.new(['10.0.1.5:11300'])
loop do
  job = beanstalk.reserve
  puts job.body # prints "hello"
  job.delete
end
```

³ Learn more about Beanstalkd in <http://kr.github.io/beanstalkd/>

⁴ Learn more about the beanstalk-client gem in <https://github.com/kr/beanstalk-client-ruby>

We'll initialize the beanstalk connection in a variable called "Q". We include an initializer script to the rails application so the connection is configured when the processes are started:

```
require "beanstalk-client"
Q = Beanstalk::Pool.new(['localhost:11300'])
```

We will add a background job to process the incoming property values and update the historical data. Each time an API event gets created (that is each time that the installation contacts the API and sends new property values), we will enqueue a new element in the queue with the name of the operation "enqueue_process_properties" and the id of the event. We don't store the properties to be processed in the job itself to keep the queue size small.

```
class Event < ActiveRecord::Base
  ...
  def after_create
    Q.yput({:group => "events_process_properties", :id => self.id}, 65000, 1)
  end
  ...
end
```

The worker process will consist in an infinite loop that will keep waiting for jobs. When a new job is fetched, it will check the "group" stored in the job body and based on its value will execute the required code. If the job fails to be processed, we will "bury" the job. This moves it to a special queue that will allow us to retry the job once we analyze the problem. Otherwise the job will be deleted from the queue.

```
class KermitWorker
  class << self
    def do
      logger.info! "Worker started"
      loop do
        job = nil
        logger.info! "Waiting for a job ..."
        job = Q.reserve
        logger.info! "New job taken."
        job_hash = job.ybody

        case job_hash[:group]
        when 'events_process_properties'
          e = Event.find job_hash[:id]
          logger.info! "#{job_hash[:group]} Event ##{e.id} STARTED!"
          begin
            e.process_properties!
            job.delete
            logger.info! "#{job_hash[:group]} Event ##{e.id} FINISHED!"
          rescue Exception => ex
            job.bury # Move the job to buried state
            logger.info! "#{job_hash[:group]} Event ##{e.id} FAILED! #{ex.to_s}"
          end
        end

        # Other background tasks
      end
    end
  end
end
```

```
end
```

To start the background process it will only be needed to start a Rails runner that calls the method *KermitWorker.do*.

```
$ ruby script /runner -e production "KermitWorker.do"
```

In order to add future background tasks, we will only need to enqueue new jobs with a different "group" parameter and include the code to process the job in the "case" block.

As this new process behaves as a daemon, it will need to include some signals to stop and restart it without affecting any jobs that may be half processed. We can add simple handlers for TERM and HUP signals as it is shown below. Basically, it'll just store the signal that has been received and when a new loop starts (and the last taken jobs is fully processed) it will clean up the connections and exit the loop. If the signal was a HUP, we'll create a new process based on the current one.

```
class KermitWorker
  class << self
    def do
      @@pid = Process.pid
      logger.info! "Worker started (pid: #{@pid})..."

      job = nil
      signal_received = false
      Signal.trap("TERM") do # Stop worker gracefully
        signal_received = :term
        logger.info! "TERM signal received"
      end
      Signal.trap("HUP") do # Restart worker gracefully
        signal_received = :hup
        logger.info! "HUP signal received"
      end

      loop do
        if signal_received
          Q.close
          break
        end
        ...
      end

      logger.info! "Worker stopped"
      if signal_received == :hup
        # Start a new worker with same parameters as current one
        p1 = fork do
          sleep 2
          config = ::Config::CONFIG
          ruby = File::join(config['bindir'], config['ruby_install_name'] + config['EXEEXT'])
          exec("#{ruby} #{ $0 } #{MYARGV.join ' '}")
        end
        Process.detach(p1)
      end

      exit(0)
    end
  end
end
```

There is another problem that we should take into account: there is a delay between the resource creation (the Event) and the storage in the database. This can lead to problems if the background process tries to find the event before it is fully stored. As a protection against it, we'll send the job back to the queue with a small delay if we don't find the record:

```
class KermitWorker
  NOT_FOUND_DELAY = 5
  class << self
    def do
      loop do
        ...
        when 'events_process_properties'
          e = Event.find job_hash[:id] rescue (record_not_found(job, job_hash, "Event #{job_hash[:id]}
            NOT FOUND!"); redo)
          ...
        end
      end
    end
  end

  private
  def record_not_found job, job_hash, log_message="NOT FOUND!", removed_message=" REMOVED!",
    options={}
    delay = (options[:delay] || NOT_FOUND_DELAY).to_i
    if (job_hash[:id].to_i > 0 rescue false)
      logger.info! "#{job_hash[:group]} #{log_message}"
      job.release(job.pri, delay)
    else
      logger.info! "#{job_hash[:group]} #{log_message} #{removed_message}"
      job.delete
    end
  end
end
```

With these changes, the API controller will only need to create the event with the serialized properties in the database. The job will be generated and processed in the background by a separate process. This will reduce the load in the API controller and the response time.

Switch the API and Workers to Merb

The version of Rails that is used in our application doesn't provide a modular initialization, making it difficult to only load specific parts to save memory in background processes or those controllers that doesn't require all the code. We also use different plugins that are not really needed outside the controllers and views that build the Network Service webpage (like authentication and javascript related libraries).

For that reason, we decided to switch both the agent controller and the workers to a smaller framework called "Merb", since they are the most critical parts in the whole Network Service application and they don't require any of those features.

Merb is a MVC Ruby framework much faster and smaller than Ruby on Rails, being ORM-agnostic, JavaScript library agnostic, and template language agnostic. The main advantages in this case are its modularity and its smaller resource consumption

though. (Additional details about the Merb project and its benefits can be seen in <http://www.merbivore.com/>, [8], [7]).

We will include Merb structure in vendor/merb folder. Its structure is similar to Rails and it has support for ActiveRecord so we can reuse all models already implemented for Rails. To make the agent work with Merb we will link all models to the new structure and apply some simple changes to the Agent controller.

To make it easier a rake task has been implemented to generate Merb compatible files from the ones written for Rails which can be reused in other projects:

```
rake merb: update_files_from_rails_project
```

The code of the task is below. It just needs to do some simple substitutions in the Rails code.

```
def escape str
  str.gsub(/([<:*, " '\[\]\] ) /, '\\\1')
end

def substitute_in_files files = { }, options = { }
  files.each do | file , patterns |
    patterns_string = ""
    patterns.each do |k,v|
      unless options[: pattern_space ].nil?
        patterns_string << " -e '/#{k}_start /,#{ k}_end/ #{v == "" ? "d" : escape("c #{v}")}' "
      else
        patterns_string << " -e `s/#{escape k}V#{escape v}/g` "
      end
    end
  end

  Dir.glob( file ).each do |f|
    system "sed #{ patterns_string } -- #{f} > /tmp/temp_subst; cp -f /tmp/temp_subst #{f};"
  end
end

namespace :merb do
  desc "Update Merb Controller & View files - take it from RoR dir and substitute "
  task : update_files_from_rails_project do
    #agent controller
    system "cp -f #{RAILS_ROOT}/app/controllers/agent_controller.rb" +
      " #{RAILS_ROOT}/vendor/merb/app/controllers/agent.rb"

    substitute_in_files "#{RAILS_ROOT}/vendor/merb/app/controllers/agent.rb" =>
      { "ENV[\"RAILS_ENV\"]" => "Merb.environment",
        " ApplicationController" => "Agent",
        " ApplicationController " => "Application ",
        " before_filter " => "before",
        " after_filter " => "after",
        "render :nothing => true" => "render \"\", :layout => false",
        ":except =>" => ":exclude =>"
      }
    substitute_in_files ( { "#{RAILS_ROOT}/vendor/merb/app/controllers/agent.rb" =>
      { "remove_for_merb" => "" } }, : pattern_space => true)
    #agent views
    system "rm -rf #{RAILS_ROOT}/vendor/merb/app/views/agent"
    system "cp -rf #{RAILS_ROOT}/app/views/agent #{RAILS_ROOT}/vendor/merb/app/views/"
    substitute_in_files "#{RAILS_ROOT}/vendor/merb/app/views/agent/*" => {
      "render(" => "", ": partial => \"\" => "partial (:",
      "\"\", :object" => ", :with", "\"\", : collection " => ", :with"
    }
  end
end
```

```
end
```

Performance Tuning

By monitoring the behavior of the system running on production, we also found some ways to improve the performance which are described in this section.

Properties stored in separate tables

One of the first things we noticed was that the performance was worse when multiple properties were involved in the reports and filtering operations. The main reason is that we are storing all the property values inside a single table, requiring multiple joins of the same big table in the MySQL queries when more than one property is used.

For that reason, we modified the system to store each property type on a different table. This will provide us with:

- smaller indexes.
- increased speed on property insertions.
- properties with a big number of entries don't affect queries which they are not involved in.
- multiple columns for different formats ('boolean_value', 'integer_value'...) become unnecessary.
- multiple joins of a single big table are not needed.

As a drawback, we will need to generate ActiveRecord models dynamically since the tables will be generated for each PropertyType element and they are not static. They can be modified, created and destroyed at runtime. This is not a big issue thanks to Ruby flexibility. Using the methods below, we are able to define the new classes and include them as part of the environment at runtime.

```
Class :: new(parent class){}
Object :: cons_set(name, value)
```

For simplicity, we'll add a method `InstallationProperty[<key>]` that returns the model assigned to the specific property type key:

```
def [] property
  property_key =
  case property
  when PropertyType
```

```

    property.new_record?? raise ("PropertyType should be stored before") :
      property.key
  when Integer
    PropertyType.find(property, :select => "'key'").key
  else
    PropertyType.valid_key?(property) ? property.to_s :
      raise ("Invalid Property Key")
  end rescue raise ("Invalid PropertyType Id")
end
get_or_create_dynamic_model property_key
end

def get_or_create_dynamic_model property_key
  cname = "InstallationProperty_ #{property_key}".camelize
  version = 0
  begin
    model = Object::const_get(cname)
  rescue
    tname = "installation_properties_ #{property_key}".downcase
    model = Object::const_set cname,
      Class::new(InstallationProperty){ set_table_name(tname) }
    model.class_eval %Q{
      def self.key() "#{property_key}" end
      def self.property_type_version() #{version} end
    }
  end
  unless Installation.associated_properties.include?("#{property_key}")
    Installation.has_many "installation_properties_ #{property_key}".to_sym,
      :class_name => cname, :dependent => :destroy
    Installation.has_many "latest_installation_properties_ #{property_key}".to_sym,
      :class_name => cname, :conditions => {:latest => true}
    Installation.associated_properties =
      (Installation.associated_properties + [property_key.to_s.downcase]).sort
  end
  model
end
end

```

Model associations with `Installation` class are generated together with the new class and to speed up the process and avoid multiple association definitions we keep the list of keys which has been already initialized inside the `associated_properties` variable.

As we have said, property types will have an associated table that will be generated when the property is created according to the property type setup. On the other hand, each time the property is modified, the table should be updated accordingly if it is necessary. We will need to perform schema modifications on the fly. In order to do it we will use `ActiveRecord::Schema.define {}` which allows us to use migration methods. We will need to:

- Create new table `installation_properties_<key>` when a new property type is defined.
- Modify column type if it is changed in `value_type` field.

In both cases `InstallationProperty<key>` model should be generated or updated. To achieve it the following code will be called after `PropertyType` is saved.

```

def update_or_create_property_table
  type = self.value_type.to_sym

```

```

tname      = self.associated_property_table_name
default_stdout = $stdout
$stdout = dev_null_class
res = true
begin
  ActiveRecord::Schema.define do
    if table_exists?(tname)
      change_column tname, :value, type if self.changes["value_type"]
    else
      create_table tname, :force => true do |t|
        t.integer :installation_id, :null => false
        t.boolean :latest, :null => false, :default => false
        t.datetime :start_at, :null => false
        t.datetime :end_at, :null => false
        t.datetime :extended_end_at
        t.integer :resource_index, :default => 0, :null => false
        case type
          when :boolean then t.boolean :value, :default => false, :null => false
          when :integer then t.integer :value
          when :float then t.float :value
          when :string then t.string :value
          when :text then t.text :value
          when :datetime then t.datetime :value
          when :binary then t.binary :value
        end
        t.integer :counter, :default => 1, :null => false
        t.integer :resolution_region, :default => 0, :null => false
      end
      add_index tname, [:installation_id, :start_at, :end_at], :name => "historical_property_index"
      add_index tname, [:latest, :installation_id], :name => "latest_property_index"
    end
  end
  InstallationProperty[self.key] # Generates specific model and installation relations
rescue Exception => err
  res = nil
end
$stdout = default_stdout
res
end

```

One important thing that we should take into account is that these models will be loaded as separate objects in each of the mongrels and workers. For that reason, the modification in one of the property types must generate a global update in all of them. To be able to do that, we use Memcache which is shared between them. Each time the new installation property model is defined using `InstallationProperty[<key>]` a version value is set up according to the value stored in Memcache. This value is checked each time the class is loaded and a reload is done if necessary.

That version value will be increased each time any of the fields that affects the model definition is changed in the associated property type. In order to avoid collisions updating the version, we use Memcache "INCR" method which is atomic. In other words, if multiple mongrels perform a modification to a property type simultaneously, version will be increased once per change. Taking this into account, we add the code below to the `PropertyType after_save` callback

```

if (self.changes["key"] || self.changes["values_type"]) rescue true)
  M.incr(" InstallationProperty_ #{self.key}".camelize + ":: property_type_version ")
end

```

As we have mentioned, each time the installation property model is used we will check if the version at definition time is the same which is stored in Memcache removing old definitions and defined relations with Installation model. To achieve this, we include the following lines in InstallationProperty model:

```

def get_or_create_dynamic_model property_key
  cname = " InstallationProperty_ #{property_key}".camelize
+  version = (M.get("#{cname}::property_type_version") || 0 rescue 0)
  begin
    model = Object::const_get(cname)
+  unless model.property_type_version == version
+    remove_dynamic_model
+    model = get_or_create_dynamic_model property_key
+  end
  rescue
    ...
  end

+ def remove_dynamic_model property_key
+  cname = " InstallationProperty_ #{property_key}".camelize
+  Object::instance_eval { remove_const cname } rescue nil
+  if Installation . associated_properties . include?(property_key.to_s.downcase)
+    Installation . associated_properties . delete (property_key.to_s.downcase)
+    if e = Dependencies.loaded.find { |e| e =~ /modelsV installation$ /}
+      Dependencies.loaded.delete e
+      Dependencies.remove_constant " Installation "
+      load "app/models/ installation .rb"
+      initialize_dynamic_models
+    end
+  end
+ end

```

Remove Empty Slots

Another problem is that each time an operation is performed we have to check if there is an empty slot after it. This increases the number of queries and the time spent during the events processing.

The empty slots are basically telling us for how long the installation has not sent a new property value. We may remove this intermediate slots if we store the same information in the previous slot. In other words, if we store how long is the empty space, blank slots become unnecessary.

In order to do so, we add an additional time to the installation property tables `extended_end_at` which will take the same values as `end_at` in the equivalent blank slot: the next slot `start_at` or null if there are none. This new timestamp column will extend until new data is received from the installation. If there are no holes in the data, it will just keep the same value as the slot end time.

The figure 3.7 shows a small example of how is the storage structure modified. V represents the value, C the counter, E the end time and EE the extended end time of each slot.

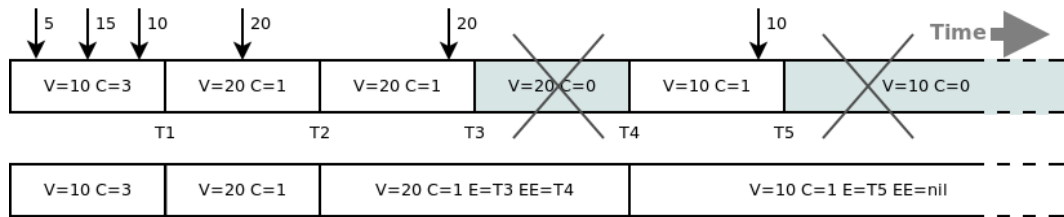


Figure 3.7 Property Empty Slots with extended end at.

Indexes Configuration

Our initial MySQL indexes configuration was mostly based on single columns. This is not the best approach since most of the queries performed over the installation property tables are using multiple columns in restrictions and ordering. Although MySQL can take multiple indexes into account in a single query, we should study the case of adding indexes based on multiple columns.

Taking a look at the different queries we do, we can identify two main types of query

- Latest Properties: `installation_id, latest`
- Properties History: `installation_id, start_at` and, `end_at` or `extended_end_at`

Column `property_type_id` was also used in both queries but it was removed with the modification described in section 3.4.1 which will help to simplify the index.

We use a B-Tree format. It is configured as an ordered list of columns, being used always starting from left to right. In other words, the index will be taken into account in those cases where the first columns are used and optimized following that order. The best approach will be the one with higher cardinality columns first (since a higher number of rows will be discarded sooner). All fields used in the index should be together, that is if we have an index with rules in keys `col1, col2, col3` and a query uses `col1, col3` only `col1` will use the index. Another limitation to take into account, is that if any of the columns is used in a range restriction (for example `col1 <= 1`), the following columns in the index will not be used. In our case we usually use ranges with time columns so they should be the last ones in the index.

Following that rule we will generate the following indexes for each installation property table:

```
add_index tname, [: installation_id , : start_at , :end_at], :name => " historical_property_index "
add_index tname, [: latest , : installation_id ], :name => " latest_property_index "
```

The old indexes in `latest` and `installation_id` are no longer needed since MySQL will use its first column in a similar way if there's only one of those in the query to be optimized.

These configuration changes has been performed following the hints described in [2].

Storage Size Estimation

In this section, we will try to give a simple mechanism to estimate the property storage size that an specific configuration can cause. This will allow us to setup production system accordingly for each specific client.

We can divide the analysis in two parts, first we will need to be able to calculate the maximum number of rows for a given number of installations and property types, then we will need to calculate the size of each row to get the total size.

To get the number of rows, we will consider that the property is changing its value constantly (to get the worst case scenario). In this case, the slots are only merged when properties came into a different region. We also consider that installations have been active during all the time the properties are kept in the server (without 'holes' in the data) and that they are sending all possible properties available in their tier.

We define the following parameters:

- Suffixes for tiers: monitoring M , professional P , basic B .
- Suffixes for property type: boolean: b , integer: i , float: f , string: s , text: t , datetime: d , binary: r .
- I_x : Number of installations with tier x .
- L_x : Time in days where next region starts, last value represents the time where property values history is discarded and $L_0 = 0$.
- NR : Number of regions + 1
- R_x : Resolution in minutes applied to that region.
- PR_x : Resolution of property x
- N : Number of properties
- P_{xy} : 1 if installations in tier y has property x , else 0

Table 3.1 Rails to MySQL type and storage size.

RoR Type	Mysql Type	Storage Size (bytes)	
boolean	TINYINT(1)	1	
integer	INT	4	
float	FLOAT	4	
string	VARCHAR(255)	$C \cdot (L + 1)$	$L < 255$
text	TEXT	$L + 2$	$L < 2^{16}$
binary	BLOB	$L + 2$	$L < 2^{16}$
datetime	DATETIME	8	

The maximum number of rows that can be stored for a property x can be obtained as follows:

$$ROWS_x = \sum_{i=B,P,M} I_i \cdot P_{xi} \cdot \left[\sum_{j=1}^{NR} \frac{(L_j - L_{j-1}) \cdot 24 \cdot 60}{MAX(R_j, PR_x)} \right] \quad (3.1)$$

Calling S_x to the maximum size of a property with type x and T_i to the type of the property i , we can get the maximum size as:

$$SIZE = \sum_{i=1}^N S_{T_x} \cdot ROWS_x \quad (3.2)$$

Where we will need to investigate the values of S_x for each type (boolean, integer ...). In order to do that, we will need to use the information contained in MySQL documentation about storage requirements, which would depend on the storage engine used (InnoDB with compact row format in our case), the mappings performed by ActiveRecord to store them, as well as the maximum value size that is allowed to be stored in string, text and binary fields and text encoding. A list with that information can be seen on table 3.1, where C represents the number of bytes used for each character and L the string/text length. In our case, we are using utf-8 and characters may require up to 3 bytes.

Using the table we can identify the types at MySQL level:

- 1 TINYINT(1) (latest), without null value
- 4 INT (installation_id, resource_index, counter, resolution_region), all of them without null value.
- 3 DATETIME (start_at, end_at, extended_end_at), 2 of them without null
- One column which changes according to property type between all types described in table 3.1.

In our case, we are using InnoDB with COMPACT row format⁵:

- Each index record contains a five-byte header that may be preceded by a variable-length header. The header is used to link together consecutive records, and also in row-level locking.
- The variable-length part of the record header contains a bit vector for indicating NULL columns. If the number of columns in the index that can be NULL is N , the bit vector occupies $(N+7)/8$ bytes. Columns that are NULL do not occupy space other than the bit in this vector. The variable-length part of the header also contains the lengths of variable-length columns. Each length takes one or two bytes, depending on the maximum length of the column. If all columns in the index are NOT NULL and have a fixed length, the record header has no variable-length part.
- For each non-NULL variable-length field, the record header contains the length of the column in one or two bytes. Two bytes will only be needed if part of the column is stored externally in overflow pages or the maximum length exceeds 255 bytes and the actual length exceeds 127 bytes. For an externally stored column, the two-byte length indicates the length of the internally stored part plus the 20-byte pointer to the externally stored part. The internal part is 768 bytes, so the length is $768+20$. The 20-byte pointer stores the true length of the column.
- The record header is followed by the data contents of the non-NULL columns.
- Records in the clustered index contain fields for all user-defined columns. In addition, there is a six-byte transaction ID field and a seven-byte roll pointer field.
- If no primary key was defined for a table, each clustered index record also contains a six-byte row ID field.
- Each secondary index record also contains all the primary key fields defined for the clustered index key that are not in the secondary index. If any of these primary key fields are variable length, the record header for each secondary index will have a variable-length part to record their lengths, even if the secondary index is defined on fixed-length columns.
- Internally, InnoDB stores fixed-length, fixed-width character columns such as CHAR(10) in a fixed-length format. Before MySQL 5.0.3, InnoDB truncates trailing spaces from VARCHAR columns.

⁵ <http://dev.mysql.com/doc/refman/5.0/en/innodb-physical-record.html>

Taking that into account, the size of a row of type x will be

$$S_x = 5 + ((NNF) + 7/8) + FP \cdot NF + \sum_i (\text{Fields type } i) * (\text{Size type } i) \quad (3.3)$$

where NNF is the number of fields that can be *null*, NF is number of fields and FP is the field pointer (1 or 2 bytes depending on the row size).

After doing appropriate substitutions, we get the sizes for each type:

$$S_b = 6 + 1 * 9 + (1 * 1 + 4 * 4 + 3 * 8 + 1 * 1) = 53 + 1 = 54 \text{ bytes} \quad (3.4)$$

$$S_i = 6 + 1 * 9 + (1 * 1 + 4 * 4 + 3 * 8 + 1 * 4) = 53 + 8 = 57 \text{ bytes} \quad (3.5)$$

$$S_f = 6 + 1 * 9 + (1 * 1 + 4 * 4 + 3 * 8 + 1 * 4) = 53 + 8 = 57 \text{ bytes} \quad (3.6)$$

$$S_d = 6 + 1 * 9 + (1 * 1 + 4 * 4 + 3 * 8 + 1 * 4) = 53 + 8 = 57 \text{ bytes} \quad (3.7)$$

$$S_s = 6 + 2 * 9 + (1 * 1 + 4 * 4 + 3 * 8 + (L + 1) * 3) = 53 + 8 = 65 + 3L \text{ bytes} \quad (3.8)$$

$$S_t = 6 + 2 * 9 + (1 * 1 + 4 * 4 + 3 * 8 + (L + 2) * 3) = 53 + 8 = 68 + 3L \text{ bytes} \quad (3.9)$$

$$S_r = 6 + 2 * 9 + (1 * 1 + 4 * 4 + 3 * 8 + L + 2) = 53 + 8 = 64 + L \text{ bytes} \quad (3.10)$$

Using this values in the equation 3.2 we can get the maximum size possible. Obviously, this is not the size that the database will have since merging operations will reduce the number of rows for those properties which keeps certain constant value. However, it gives us a general idea of what is the biggest we could get in case all properties are changing constantly.

Filtering Application Users

In this chapter we will add filtering capabilities to the application. This will allow to define groups of end-users based on the property values received from the installations. Furthermore, the ISVs will be able to perform operations in these groups once they are defined and to generate statistics restricted to a subset of installation groups (as we will see in chapter 5). For instance, the user may want to send a notification to installations located in a specific country or, get information about the impact that a campaign is having in the number of installations in that country.

The implementation will be divided in two sections. First, a Rails plugin will be included to extend `ActiveRecord` (the default database connector in Rails) with additional filtering capabilities. By implementing it as a plugin, it will be possible to reuse it in the future in other projects. Then, we will add the specific installations filtering functionality in `Network Service` based on the new plugin.

Rails plugins structure

Rails application functionalities can be extended using pieces of code called plugins. These extensions can be easily added, upgraded or removed from the application through a simple script. The script takes care of updating or removing the files inside `APP_PATH/vendor/plugins` and executes specific installation or uninstallation actions contained in them.

To start creating a plugin we can use the default plugin generator:

```
$ script /generate plugin plugin_name
  create vendor/plugins/plugin_name/lib
  create vendor/plugins/plugin_name/tasks
  create vendor/plugins/plugin_name/test
  create vendor/plugins/plugin_name/README
  create vendor/plugins/plugin_name/MIT-LICENSE
  create vendor/plugins/plugin_name/Rakefile
  create vendor/plugins/plugin_name/init.rb
  create vendor/plugins/plugin_name/install.rb
  create vendor/plugins/plugin_name/uninstall.rb
  create vendor/plugins/plugin_name/lib/plugin_name.rb
```

```
create vendor/plugins/plugin_name/tasks/plugin_name_tasks.rake
create vendor/plugins/plugin_name/test/plugin_name_test.rb
```

As we can see, it provides the basic structure to start the implementation. Every time the application is started all the plugins are initialized loading their init file. All class definitions and libraries that we want to be available in our application should be loaded from it. More details can be found in the table 4.1.

Table 4.1 Rails Plugin Files.

<code>init.rb</code>	Executed every time the Rails application is started.
<code>install.rb</code>	It's run once when the plugin is first installed.
<code>uninstall.rb</code>	It's run once when it is uninstalled
<code>lib/</code>	Contains the code of the plugin. Libraries and classes in this folder will be accessible from the rails application. For some specific changes <code>init.rb</code> file can be needed.
<code>tasks/</code>	It contains <code>.rake</code> files that will be added to rake tasks.
<code>test/</code>	Specific tests for the plugin code.
<code>generators/</code>	You can use it to add new code generators to the application.
<code>rdoc/</code>	Contains the documentation.

Filter Models Plugin

In general, any application will need to get information stored in the database based on some rules. `ActiveRecord` provides tools to interact with the database and is the default interface used by Rails applications. `ActiveRecord` defines a class for every table in the database called `model`, which includes the methods for finding, updating and destroying rows as well as to describe the associations between the tables.

For example, we can look for some entries following some rules:

```
Person.find(:all, :conditions =>
  ["age < ? and genre IN(?) and times_seen BETWEEN(?)", 30, ["male"], [3,5]])
```

or we can search them taking into account other related models:

```
Person.find(:all, :include => [:job], :conditions => ["jobs.name LIKE ?", "%a%"])
```

When we are implementing a web application, we will usually show the user different search fields inside a form and, after the user has sent the information, we will need to generate similar restrictions as the ones shown above. This is not a problem if there are only some fields with simple rules like "field1 equals something", "field2 contains something", "has child element with field equals something"... and the relations between models doesn't change.

However, if we want to add the ability to use different rules (<, >, BETWEEN...) over the same field, we would need to include additional code to generate the needed find parameters and we will find ourselves repeating the same code structure for different records and search restrictions. Furthermore, a change in the location of a specific field or the modification of the relations between models, i.e. adding an intermediate table, would force us to modify those pieces of code.

To avoid that and speed up the process of adding any additional information with advanced filtering rules (without the need of rewriting the code to generate queries and form fields) we will implement a new plugin called FilterModels.

Objectives

The main goals for FilterModels plugin are the following:

- Allow to configure and store filters in the database
- Define helpers to generate forms based on defined fields
- Fetch the records that pass one or more filters.
- Count the number of records that passes one or more filters.
- Check if one model instance passes the filter.
- Check if a set of field values pass the filter (not taking into account any database stored data) .
- Provide methods to generate specific queries based on defined fields instead of database column names.

Implementation

The plugin will create two new concepts on top of the regular database models:

- Filterable Model: it is the model in which the filter rules will be applied.
- Filter: set of rules that may be applied on a filterable model. It can also be a model.

Taking this into account, the code is divided in four main files:

- filter: class to be used over any filterable model.
- filter_methods: modifications to make a stored model behave as a filter
- model_methods: model related to record that will be filtered changes
- view_helpers: tools to generate application views

In this case, we don't need to perform any changes in the application for installing or uninstalling the plugin. The file structure will be the following:

```

vendor/plugins/ filter_models /
|-- README
|-- Rakefile
|-- init .rb
|-- lib
    |-- filter_models
    |   |-- filter .rb
    |   |-- filter_methods .rb
    |   |-- model_methods.rb
    |   |-- view_helpers.rb
    |-- filter_models .rb

```

Model methods extension

In order to make the models behave as filter or filterable, we will need to extend the default model class with specific methods. These changes shouldn't affect any other models that are not involved in any filtering. We can achieve this using the Ruby calls 'include' and 'extend', that can add functions on the fly to the class definition or the instances of a specific class.

Based on this, we add two new methods to the parent class for all models (`ActiveRecord::Base`). When one of them is called from a model definition, they will load the required code to make it behave as a Filter (`filter_model`) or as a Filterable (`filterable_model`).

For example, to add an additional method `filterable_model` to Base, we add a module with the structure below:

```

module FilterModels
  module ModelMethods
    def self .included(base)
      base.extend ClassMethods
    end

    module ClassMethods
      def filterable_model options={}
        # Any initialization code
        include FilterModels :: ModelMethods::InstanceMethods
        extend FilterModels :: ModelMethods::SingletonMethods
      end
    end

    module InstanceMethods
      # List of methods that will be available for any instance of
      # the model defined as filterable
    end

    module SingletonMethods
      # Methods that will be available though model class defined
      # as filterable
    end
  end
end

```

It is also necessary to add the following line to the plugin init script to make the new method available in the application.

```
ActiveRecord::Base.send(:include, FilterModels::ModelMethods)
```

At initialization time, the `include` method will be called for `ActiveRecord::Base` class. This will execute `ModelMethods.included` in its context, defining the method `filterable_model` as part of `ActiveRecord::Base` class. If this new method is called from any model definition, it will include the functionality into it adding any instance and class methods with the 'include' and 'extend' calls.

In order to make a specific model filterable, we just need to call the new method:

```
def ModelClassName << ActiveRecord::Base
  filterable_model
  # Rest of the code
end
```

In other words, we have added a trigger to all models that allow us to load the functionality when it is needed.

Fields configuration

The fields available to be used in the filter rules should be defined. This is useful to include additional information apart from the one stored in the database column and to restrict the fields to a subset of them. It also behaves as a layer between the database structure and the logic stored in the rules. In other words, It will make possible to have alterations in the database without the need of updating the rules in any filter, only the field definition would need to be changed.

The fields will be configured at model definition or dynamically overwriting the method `'filter_fields'`. In both cases, they should be written as a Hash where each pair represents the unique identifier that will be used in the different rule definitions and form inputs and the specific settings for that field:

```
filter_fields = { field_id => settings1, field2_id => settings2 ... }
```

Settings are formatted in pairs `setting-value` inside a Hash. The main options are:

- `:field` [needed] It should be the name of the column in the database table described by the current model or, a route to the field taking into account the relations between models (defined using `ActiveRecord` association methods). In that case, it should be inserted as:

```
{assoc_name1 => {assoc_name2 => ...{assoc_name_n => :column_name}...}}
```

- `:type` value format ('string', 'integer', 'boolean'...). This is useful to be able to check rules over hashes outside the database.
- `:name` naming that is shown to the user while through the view helpers provided in the plugin.
- `:hidden` it avoids showing this field while using helpers.
- `:options` possible values that the field could take. It can be inserted as an array of pairs [value,name] or as a string containing ruby code that would generate that array dynamically.
- `:conditions` additional conditions that will be added to the particular joins in the route to the field defined in 'field' option. These restrictions are inserted as pairs association_name, restriction inside a Hash.
- `:join_name` alternative naming of joined table in the SQL query to allow multiple joins over the same table. They should be written as pairs association name, join name in a Hash object.

A simple configuration example can be seen below:

```

def Writer << ActiveRecord::Base
  has_many :posts
  belongs_to :group

  filterable_model : filter_fields => {
    :name => { :field => :name, :name => "User Name", :type => :string },
    :group => { :field => :group_id, :type => :integer,
      :options => "Group.find(:all).map{|g| [g.id, g.name]}" },
    :post_title => { :field => { :posts => :title }, :type => :string },
    :readers_name => { :field => { :posts => { :readers => :name } } },
    :type => :string },
  }
end

def Group << ActiveRecord::Base
  has_many :writers
end

def Post << ActiveRecord::Base
  belongs_to :writer
  has_and_belongs_to_many :readers
end

def Reader << ActiveRecord::Base
  has_and_belongs_to_many :posts
end

```

In this example we would be able to look for writers based on their name, group, post title and also names of the post readers. The identifiers for the fields (the ones to be used in the rules) are `name`, `:group`, `:post_title` and `:post_readers`. Group values are restricted to those ids stored in database.

The settings `:conditions` and `:join_name` are useful in those cases where we need to filter using more than one rule over an association. MySQL lacks the ability to easily apply rules to column values in more than one joined row. For instance, in our particular case we needed to apply rules based on the properties stored in the `'installation_properties'` table.

In other words, if we do a join between `'installations'` and `'installation_properties'` we can't look for those installations with both property A equal something and property B equal something else, but only for those with A equal sth OR B equal sth. The following query will always return 0, since it's checked row by row:

```
SELECT COUNT(distinct installations.id) FROM installations as i
LEFT JOIN installation_properties as ip ON ip.installation_id = i.id
WHERE (ip.property_type_id = 1 AND ip.string_value = "a") AND
(ip.property_type_id = 2 AND ip.string_value = "b")
```

As we will see later, we can define multiple fields in this case modifying conditions and the `join_name`. The cost of this will be the multiple joins over the same table (one for each field involved). At MySQL level, the query takes the following format:

```
SELECT COUNT(distinct installations.id) FROM installations as i
LEFT JOIN installation_properties as ip1
ON ip1.installation_id = i.id AND property_type_id = 1
LEFT JOIN installation_properties as ip2
ON ip2.installation_id = i.id AND property_type_id = 2
WHERE ip1.string_value = "a" AND ip2.string_value = "b"
```

Note that these field settings are no longer needed in Network Service after the properties storage was splitted into multiple tables as we described in section 3.4.1.

Filtering Rules Generation

Once we have defined the fields, we should be able to describe rules over them. We will define one rule per field, as it is enough for our needs and simplifies the process. Taking that into account, the chosen format is:

```
{ :field_name => [<method>, <value1>, <value2> ...], :field_name2 => ... }
```

.

For each field involved, we specify the comparison method to be used (`:eql`, `neq` ...) and one or more values. The complete list of methods that could be used is described on table 4.2.

In order to check those rules over stored records we must know which joins are necessary and which conditions should be applied.

Getting the necessary joins

Table 4.2 Filtering Operations.

<code>:null</code>	Not value assigned.
<code>:nnull</code>	Any not null value has been assigned.
<code>:eq1</code>	Equals at least to one of the values.
<code>:neq</code>	Not equals to any of the values.
<code>:lt</code>	Less than.
<code>:gt</code>	Greater than.
<code>:let</code>	Less or equal than.
<code>:get</code>	Greater or equal than.
<code>:btw</code>	Between 2 values.
<code>:nbtw</code>	Not between 2 values.
<code>:like</code>	Like any of the values inserted (wildcard allowed '*').
<code>:regexp</code>	Pass a regular expression.
<code>:let_ago</code>	Until some time ago (values are: number of units and time unit:days, hours...).
<code>:get_ago</code>	Since some time ago.

`ActiveRecord` allows to define relations between models. This relations are stored as class variables in the model class that includes information about the conditions involved, types of relation, other model classes, etc. We can access this using the method `reflect_on_association(:assoc_name)`, which returns an `AssociationReflection` with the details. The available association types inside `ActiveRecord` are shown in table 4.3.

Thanks to the data stored in the models, the list of association names used in the field definitions are enough to fetch all the details required about the relation between the models involved in the rules.

For each field involved in the process (which can be obtained by checking the keys in the rule definition), we go through the steps in the field configuration getting the needed information to join the table or tables related to that association. This includes the name of the table to be joined and the needed `ON` clause, that will depend on the type of association. This can be obtained using the method `'macro'` in the reflection object, which returns the perspective of the relation from the current model. This allows us to know how the table should be included in the queries:

- If it returns `:belongs_to`, the foreign key is stored in the table that has been inserted in the previous step, being necessary the following SQL:

```
LEFT JOIN #{new_model.quoted_table_name} ON
  #{previous_model.quoted_table_name}.#{ reflection .primary_key} =
  #{ reflection .klass .quoted_table_name}.id
```

- If it returns `:has_one` or `:has_many`, foreign key is in the one that is being inserted. In that case we will need to do:

```
LEFT JOIN #{new_model.quoted_table_name} ON
  #{previous_model.quoted_table_name}.id =
```

Table 4.3 ActiveRecord Associations.

Type	Configuration
one to one	<p>:belongs_to in model which table contains the foreign key and :has_one in the other model.</p> <pre> class Employee < ActiveRecord::Base has_one :office end class Office < ActiveRecord::Base belongs_to :employee # foreign key - employee_id end </pre>
one to many	<p>:belongs_to in the associated model and :has_many in the base.</p> <pre> class Manager < ActiveRecord::Base has_many :employees end class Employee < ActiveRecord::Base belongs_to :manager # foreign key - manager_id end </pre>
many to many	<p>:has_and_belongs_to_many in both models, foreign keys are stored in join table. It's used when there's no defined model for the intermediate table.</p> <pre> class Programmer < ActiveRecord::Base has_and_belongs_to_many :projects end class Project < ActiveRecord::Base has_and_belongs_to_many :programmers end </pre>

```
#{ reflection . klass .quoted_table_name }.#{ reflection . primary_key }
```

- If it returns :has_and_belongs_to_many, we should insert two tables, including an intermediate join table that contains both foreign keys. The name of that additional table is taken as the one specified in the options of :has_and_belongs_to_many that can be taken from reflection.options[:table_name] or the concatenation of the other two table names in alphabetical order. In this case we will have two joins:

```

LEFT JOIN '#{intermediate_table}' ON
  '#{previous_model.quoted_table_name}.id =
  '#{intermediate_table}'.#{reflection.primary_key_name}
LEFT JOIN '#{new_model.quoted_table_name}' ON
  '#{reflection.klass.quoted_table_name}.id =
  '#{intermediate_table}'.#{reflection.association_foreign_key}

```

In all of them, previous model is modified in each step and reflection represents the result of evaluating the method `reflect_on_association` in that model. The call to `reflection.klass` returns the model that represents the table that is being joined in current iteration, and the one that should be used as previous model in the next step.

In other words, if we have a field in a filterable model defined as:

```
: field => { :assoc_1 => { :assoc_2 => { ... => { :assoc_N => :column_name } ... } }
```

Where each association refers to a model called `Model<i>` with $i=1..N$ and we are generating the joins to evaluate a rule based on that field: in the first iteration, the previous model is set to `FilterableModel` and reflection will take the value returned by

```
reflection = FilterableModel.reflect_on_association (:assoc_1).
```

After that, previous model is setup as the value returned by `reflection.class` (that will be `Model<i>`) and next reflection takes the value obtained from

```
reflection = Model<i>.reflect_on_association (:assoc_<i+1>)
```

This process is repeated N times, until `:column_name` is reached.

Each of those necessary joins in SQL format, that are generated for every field and node until reaching the needed value, are stored in an Array in order to be able to remove duplicates easily. This avoids the possibility of joining the same table twice for the same field.

Getting restrictions

Apart from the needed joins, it's necessary to obtain the different restrictions according to the configured rules. We should transform the operations shown in table 4.2 into a valid SQL code that could be inserted as part of the `WHERE` clause. The field description (table name and column) is obtained in a way similar to joins generation.

We will consider that the information stored in the rules as field values are probably provided by the user. This means it could be affected by SQL injection attacks, specially if we just join those values into a string without any further processing. For example, we could try to drop a table or run any other sql commands:

```
name = "a"; DROP TABLE users; SELECT 'a'
User.find (: first , : conditions => "name = '#{name}'")
```

Fortunately, `ActiveRecord` provides the required protection if we use the insertion methods based on question mark substitutions. The method consists in giving the conditions parameter an array where the first element is a string which contains the restrictions with the symbol `'?'` instead of values. Then, for each `'?'`, we should add the corresponding value to the array. Those values will be inserted in the correct format and any security issue related to SQL injection is avoided.

For example if we have the incoming values from the user stored in 'name' and 'email' variables, we can find an user safely as is shown below.

```
User.find (: first , : conditions => ["name = ? AND email = ?", name, email])
```

Taking that into account, the restrictions applied will be generated as it's shown in table 4.4. To get the complete restriction, the strings shown as the first array element in the table will be concatenated using AND, the rest of the elements will be added on the same order as those concatenations. Once all the restrictions are joined, the result will be a single array.

Table 4.4 Filtering Operations to SQL Conditions.

:null		["table.field IS NULL"]
:nnull		["table.field IS NOT NULL"]
:eq1	1 value	["table.field = ?", values[0]]
	>1 value	["table.field IN (?)", values]
:neq	1 value	["table.field != ?", values[0]]
	>1 value	["table.field NOT IN (?)", values]
:lt		["table.field < ?", values[0]]
:gt		["table.field > ?", values[0]]
:let		["table.field <= ?", values[0]]
:get		["table.field >= ?", values[0]]
:btw		["table.field BETWEEN ? AND ?", values[0], values[1]]
:nbtw		["table.field NOT BETWEEN ? AND ?", values[0..1]]
:like		["table.field LIKE ?", values[0].gsub('*', '%')]
:regexp		["table.field REGEXP(?)", values[0]]
:let_ago		["table.field <= NOW() - INTERVAL #{values[0].to_i} #{values[1].upcase}"]
:get_ago		["table.field >= NOW() - INTERVAL #{values[0].to_i} #{values[1].upcase}"]

When we are working with multiple filters, the relation between them should be also configured in order to be able to join the restrictions in the correct way. Calling restrictions<i> to the restrictions related to the filter<i> that is used in the query, we have 3 ways to join them:

- and - all filters should be passed. The rules are joined by AND operator:

```
WHERE (<restrictions1>) AND (<restrictions2>) ... AND (<restrictionsN>)
```

- or - at least one of them should be passed. OR operator is used:

```
WHERE (<restrictions1>) AND (<restrictions2>) ... AND (<restrictionsN>)
```

- none - no filter is satisfied. We use NOT over OR joined restrictions:

```
WHERE NOT ( ( <restrictions1> ) OR ... OR ( < restrictions N> ) )
```

Furthermore, we include the ability to force the first specified filter to be passed, which is particularly useful for those situations where we are filtering through a HTML form some of the fields. That way, the form will be transformed into a filter and added to the list of filters. After applying the option `:force_first_filter`, the WHERE clause will take the form:

```
WHERE ( < restrictions 1> ) AND ( < restrictions 2 – N joined by and,or,none methods > )
```

Cached rules

In those cases where the filter is stored in the database, we can speed up the process of gathering the information by caching joins and query restrictions (once they have been generated taking into account the reflections between models).

To be able to apply multiple filters, we should store the joins as a serialized array, where each of the elements represents one table join. That way, it's possible to avoid multiple joins on the same table by removing duplicated entries between those arrays. On the other hand, restrictions should be stored keeping the array form which was used in previous sections (with '?' symbols) to prevent SQL injections.

Both elements are stored into the database at update time. They are constant once the filter rule has been specified and while filter fields definitions are not modified. If the object is updated the cache will be regenerated.

WillPaginate Compatibility

`WillPaginate`¹ is one of the most common plugins used in Rails applications. It provides the tools to paginate a list of database records in a more reliable way than the mechanisms that Rails provides by default. These lists usually comes from a search and, in our case from a rule definition.

We should take this into account and provide the information necessary to use its helpers in order to get the correct representation for the paginated content. This can be achieved easily by modifying the methods used to fetch the records.

We just need to use the method `paginate` instead of the default search method (`find`) provided by `ActiveRecord`. The parameters are similar, adding just some additional options related to pagination: the total number of entries, which is current page or the number of elements per page.

A trigger option is included to switch between `find` and `paginate` methods in our plugins to allow both paginated and full list results.

¹ Learn more about willpaginate https://github.com/mislav/will_paginate/wiki

Filterable Model Extensions

Models that are defined as filterable are extended with the following class and instance methods:

```
# Class methods
module SingletonMethods
  # Similar to ActiveRecord::find, adding the rule of being inside a
  # filter
  def find_with_filter find_mode, filter, options = {}
    ...
  # Similar to ActiveRecord::find, adding the rule of being inside
  # one, all or none of the filters
  def find_with_filters find_mode, filters = [], logic=FilterModels::RULE_BOOLEAN[:and], options = {}
    ...
  # Similar to find_with_filter adding willpaginate support
  def paginate_with_filters filters = [], logic=FilterModels::BOOL[:and], options = {}
    ...
  # Similar to find_with_filters adding willpaginate support
  def paginate_with_filter filter, options = {}
    ...
  # Count number of records in filter
  def count_in_filter filter, options = {}
    ...
  # Count number of records in one, all or none of filters
  def count_in_filters filters = [], logic=FilterModels::BOOL[:and], options={ }
end

# This module contains instance methods
module InstanceMethods
  # Determines if current element passes a filter
  def pass_filter?( filter, field_values = nil)
    ...
  # Determines if current element is inside one, all or none of the filters
  def pass_filters?( filters = [], logic=FilterModels::BOOL[:and], field_values = nil)
    ...
  # Returns the value stored in one of the fields
  def read_field_value( field )
end
```

Filter Model Extensions

Models used as filter are extended with the following class and instance methods:

```
# This module contains singleton methods for filters
module SingletonMethods
  # Returns the list of used fields in a list of filters
  def used_fields filters
    ...
  # Returns the list of joins to be used in a list of filters
  def sql_joins filters, options = {}
    ...
  # Returns the array of conditions to be applied to filter out those
  # elements in any, all or none of the filters
  def sql_conditions filters, logic=FilterModels::RULE_BOOLEAN[:and], options={ }
    ...
  # Returns the array of conditions and joins to be applied to filter
  # out those elements in any, all or none of the filters
  def sql_joins_and_conditions filters, logic, options = {}
    ...
  # Returns the route to specific field in SQL format
  def sql_field_value filter_class, field_definition
    ...
end
```

```

# Remove cache assigned to all filters
def remove_all_sql_cached
  ...
# Checks if a list of pairs field-value passes any, all or none of
# the filters
def check_field_values field_values={}, filters=[], logic=FilterModels::BOOL[:and]
end

# This module contains instance methods for filters
module InstanceMethods
# Adds an additional restriction
def add( field , rule , values )
  ...
# Removes all restrictions over a field
def del( field )
  ...
# Returns the fields used in current filter instance
def used_fields
  ...
# Returns the list of joins to be used
def sql_joins options = {}
  ...
# Returns the array of conditions to be applied
def sql_conditions
  ...
# Returns the list of joins and conditions
def sql_joins_and_conditions filters , logic
  ...
# Used to configure the new rules from view helpers
def rule_as_hash= rules
  ...
# Returns the list of properties in correct format to be used in
# view helpers
def rule_as_hash
  ...
# Generate filter cache
def generate_sql_cached
  ...
# Transforms the value to the correct format according to the format
# configuration in field associated with key
def transform_value key, value
  ...
# Transforms a list of value to the correct format according to the
# format configurations in fields associated with keys
def transform_values field_values={}
  ...
end

```

View helpers

The Filter Models plugin includes some view helpers which make it easier to create forms based on rules. This includes the ability to select filters already stored and to create a new set of rules on the fly. It will be possible to include this selectors in any operations which we want to target to subsets of installations.

The process to include the new functionalities to the default view generation in Rails will be similar to the one followed in previous sections to extend ActiveRecord. In this case, we just need to include some class methods to the class `ActionView::Base`. However, we should take into account that Merb does not have such a class. We must protect that modification to not be made in case `ActionView` is not defined. The code in `init.rb` file can be seen below:


```

if defined? ActionView
  require 'filter_models/view_helpers'
  ActionView::Base.class_eval { include FilterModels::ViewHelpers }
end

```

This snippet of code will make any public functions in the module `FilterModels::ViewHelpers` to be available in any of the html templates in the Rails application.

Input Rules

In order to be able to create and store filters, we will need a way to define a set of rules. A new method is included in the views to insert a widget already customized with the filter fields configuration:

```

module FilterModels
  module ViewHelpers
    def input_rules(object, params = {})
      ...
    end
  end
end

```

Where `object` should be a filter element and `params` allows some customizations as selecting the fields to be included through the options `:hide_fields` and `fields` or, modifying the name of the form parameters returned in the responses.

When this method is inserted in any view a selector with the list of available fields (the ones defined as filter fields) will be included in the view. When one of the fields is chosen through the interface, a new text input appears below with a set of rules (`==`, `!=`, `<` ...) based on the type of field. In case of selecting a rule that allows multiple values (like `==`), two links will be shown to "add" and "remove" text inputs. To remove an already inserted rule it is just necessary to uncheck the checkbox shown next to the rule.

An example of how the widget looks like can be seen in figure 4.1.

The widget behaviour is written in javascript using the `prototype.js` library which comes with Rails by default.

Select Filters

Another tool that will be needed in the views generation is a way to select which already defined filters should be applied to specific actions. We will include a widget to select the filters inside any forms used for sending information related to the operations to be targeted to specific subsets of installations. The method will be added as it is shown below:

```

module FilterModels
  module ViewHelpers
    # Shows a form to choose between filters and the logic to be used
    # (any, all, none of them should be fulfilled)
    def select_filters name, filters, selected_ids, relation_name, selected_relation, options = {}
      ...
    end
  end
end

```

Add restriction :
 Installation First Contact >=(ago) 32 days
 Country ==
 Product Name !=
 Mem Total > 1048576
 Installation GUID regexp (3232|444)\$

Figure 4.1 FilterModels: rule generation through view helpers..

end
end

Adding it to any view will allow us select two things. First, we can include a list of filters to be taken into account. It will be sent to the server in the parameter specified as name and will take one or more of the filters passed to the method. By default, those included in the list of selected ids will be chosen. The second, is the logic to be used between the filters (choosing if all of them, any of them or none of them should be passed), that will be stored in the relation_name variable, By default, it will be set to selected_relation.

The options field allows some additional changes in the behavior. For example, it is possible to avoid a blank selection using :noblank=>true or, alter the class prefix used in internal javascript to allow multiple selectors per page using :class option.

Grafically, when this method is inserted in any view, it provides two selectors with the list of filters and available joins between them. When a filter is chosen, it is added to a list on the left which includes a checkbox to remove any of the elements included. An example of how it looks like can be seen in the figure 4.2.

Mac Users
 Linux Users
 Registered Users

Add filter:
 Join Method: all filters (AND)

Figure 4.2 FilterModels: selecting filters through view helpers..

Installation Filters

We will now apply the plugin to the Network Service application. As we mentioned in previous sections, our goal is to filter the installations and perform targeted operations to the groups defined by those filters.

In our case, the `Installation` class will behave as a filterable model. In order to store the filters, a new model is included, `InstallationFilter`, which will be configured as a filter model. The new table just needs to have a field to store the generated rule plus other describe fields like 'name' or 'description'. Detailed database setup can be seen below:

```
create_table "installation_filters", :force => true do |t|
  t.string "name", :default => "", :null => false
  t.text "rule", :default => "", :null => false
  t.integer "special"
  t.text "description"
  t.timestamps
end
```

As can be seen in the migration, there is also a column called 'special'. This will be used as an identifier in order to define specific internal filters. More details about it will be explained later.

The model will take the form shown below, where it is defined as a filter model for `Installation` elements and, some basic validations have been included.

```
class InstallationFilter < ActiveRecord::Base
  validates_presence_of :rule # Avoid empty rule
  validates_presence_of :name # Avoid empty name
  serialize :rule

  filter_model :model => "Installation"
end
```

In order to make the installations filterable. We will need to include the field definitions that will be available through the filters. In our case, the configuration will be modified depending on the database contents, since property types will define which tables should be used to get property values from. This makes it necessary to overwrite the method `filter_fields` and perform some database queries. Taking all this into account, the `Installation` model structure is modified as it is shown below.

```
class Installation < ActiveRecord::Base
  ...
  filterable_model

  def self.filter_fields option={}
    if (fields = M.get('installation_filter_fields').blank? || option[:reload])
      fields = {
        :i_guid => {:name => "Installation GUID", :field => :guid},
        :i_active => {:name => "Installation Active", :field => :active},
        :i_registered => {:name => "Installation Registered", :field => :registered},
      }
    end
  end
end
```

```

: i_testing => {:name => "Testing Installation ", :field => :testing },
: i_latest_contact => {:name => "Installation Last Contact", :field => :latest_contact },
: i_created_at => {:name => "Installation First Contact", :field => :created_at },
:version_name => {:name => "Version Name", :type => :string , :field => {:version => :name}},
: version => {:name => "Version", :type => :integer , :field => :version_id , :options => "
  Version.find (: all , :include => :product, :order => 'versions.name').map{|p| [ p.id, '#{p.
  product.key_name} #\{p.name}\'}" },
:product => {:name => "Product", :type => :integer , :field => {:version => :product_id } , :
  options => "Product.find (: all , :order => :name).map{|p| [ p.id, p.name]}" },
}
PropertyType.find (: all ).each do |p|
  fields [{"prop_#{p.key}".to_sym] = {
    :name => "#{(p.name.blank? ? p.key. titleize : p.name.gsub(/%.* /, '')) }", :type => p.value_type.
    to_sym,
    :field => {" latest_installation_properties_ #{p.key}".downcase => :value }
  }
end
M.set(' installation_filter_fields ', fields ) rescue nil
end
fields
end
...
end

```

Where we cache the field definitions in Memcache to reduce the number of database queries.

From the application code perspective, this configuration allows to define rules based on the properties using the fields `prop_<key>`. For example, we can count the number of installations with a certain number of users and in specific languages:

```

f = InstallationFilter .new :rule => {:prop_lang => [:eql, "es", "en", "ja"], :prop_users => [:gt, 10]}
Installation .count_in_filter (f)

```

However, the user will not need to use this approach, just the widgets described in previous sections.

A new section is included in the interface to manage the installation filters. The main page lists the defined filters, showing the number of active installations in each of them. It can be seen in the Figure 4.3. When the user wants to edit or create a new filter, a view based on the helpers included in FilterModels plugin is shown. This page can be seen in the Figure 4.4.

Special filters

The installations that are tracked in Network Service, may be in a special status that are important for generating statistics and performing other operations. Three special status values are included:

- **active:** if it has contacted the server in the last x days. By default all statistics are based on them.
- **testing:** identifies the installations made with test purposes, they are ignored by default in all the stats.

Acme, Inc. Network Service

My Account Log out Help

Start Page Products Updates Messages Reports Installations Clients Alarms Administration

Installations Overview
Manage Filters
Mass Tagging
Tags
Tag Cloud

The Installations section allows you to get information about different installations and manage installation filters.

User Defined Filters Special Filters

Name	Description	Active Installations
10+ Users	Installations with more than 10 users	353
Linux Users		873
Spanish Linux Users with -50 users	Installations from Spain which has Linux as operating system and less than 50 users	13
Windows Registered Users		0

NEW

Figure 4.3 Installation Filters: Index page..

Acme, Inc. Network Service

My Account Log out Help

Start Page Products Updates Messages Reports Installations Clients Alarms Administration

Installations Overview
Manage Filters
Mass Tagging
Tags
Tag Cloud

The Installations section allows you to get information about different installations and manage installation filters.

Manage Filters

Adding New Installation Filter

Name: Interesting Installations

Description: Those are the most interesting installations because ...

Rules

Add restriction: [dropdown]

Operating System: [dropdown] Solaris

license_key_data: [dropdown] regex s32-(23|43|sda).*-key

Number Of Users: [dropdown] >= 50

Language: [dropdown] Greek, English, Esperanto, Spanish

CREATE BACK

Figure 4.4 Installation Filters: Adding a new filter..

- **registered:** relevant installations from the ISV perspective.

We store them as flags, which should be updated according to specific rules. Since these rules are easily implementable with the installation filters, we include filters with specific identifiers for each of them. The user will have limited access to them (not being able to delete them for example).

Based on those filters, a cron task will update all installation flags. This task basically assigns to each flag the value of evaluating the filter rules. In other words, it should do an update similar to:

```
UPDATE `installations` <JOINS> SET `installations`.`<FLAG>` = (<FILTER.sql_conditions>)
```

To speed up the process it will be run with `Merb`. It is important to note also that only active installations should be taken into account, since the properties haven't changed since the latest installation contact. If any of the special filter definitions is modified, a new job will be generated which will reprocess every installation in the database (including inactive).

Taking that into account, the following elements are added to `InstallationFilter`.

```
class InstallationFilter < ActiveRecord::Base
  ...
  after_save :process_triggers_in_background, :if => "changes.keys.include?('rule') && changes['rule'][0]!=
    changes['rule'][1]"
  def process_triggers options={}
    if self.special && (field_to_update = SPECIALS[self.special][: installation_flag ])
      cond = self.sql_conditions
      Installation.connection.execute %Q{
        UPDATE LOW_PRIORITY 'installations' #{self.sql_joins}
        SET 'installations'.'#{field_to_update}' = (#{ActiveRecord::Base.instance_eval { sanitize_conditions
          cond }})
        #{'WHERE 'installations'.active = TRUE' unless SPECIALS[self.special][: process_inactive ] ||
          options[: process_inactive ]}; }
    end
  end

  def process_triggers_in_background
    Q.yput {:group => "background_update", :class => "InstallationFilter", :id => self.id, :method => :
      process_triggers, :params => [[: process_inactive => true]], 65000, 1
  end
end
...
```

We also include a new background job to the worker that we created in section 3.3.3.3: "background_update". That allows us to send specific method calls to be processed in the background like the old filter triggers processing mentioned above.

```
class KermitWorker
  class << self
    def do
      ...
      loop do
        ...
        case job_hash[:group]
          ...
          when 'background_update'
            logger.info! "[#{job_hash[:group]}] STARTED!"
            begin
              c = job_hash[:class].constantize
              if job_hash[:id]
                element = c.find(job_hash[:id])
                result = element.send(job_hash[:method], *(job_hash[:params] || []))
              else
                result = c.send(job_hash[:method], *(job_hash[:params] || []))
              end
            end
            logger.info! "[#{job_hash[:group]}] FINISHED!"
            job.delete
          rescue Exception => e
            logger.info! "[#{job_hash[:group]}] FAILED!"
            notify_exception e, job_hash
            job.bury
          end
        end
      end
    end
  end
  ...
end
```

```

end
end
end
end

```

Installation filters applied to notifications

Once we have defined the groups of installations we should be able to use them in different places. For example, we are going to explain the process of making notifications configurable with groups.

In order to do it, we should create `has_and_belongs_to_many` relations between the installation filters and the notification records which will need a connection table. It will be also needed to store the logic between the selected filters inside the notification element.

The changes in the database are shown below:

```

create_table "notifications_installation_filters",
:primary_key => "notification_id , installation_filter_id", :force => true do |t|
  t.integer "notification_id", :null => false
  t.integer "installation_filter_id", :null => false
end
add_column :notifications, :integer, "installation_filters_relation"

```

The models should be updated adding the corresponding ActiveRecord relations:

```

class Notification < ActiveRecord::Base
  has_and_belongs_to_many :installation_filters, :join_table => :notifications_installation_filters
  ...
end

class InstallationFilter < ActiveRecord::Base
  has_and_belongs_to_many :notifications, :join_table => :notifications_installation_filters
  ...
end

```

Each time the list of notifications is sent to the end-user, we will need to check if the notification is allowed to be sent to that installation. In other words, we need to check if the filters are passed. The problem is that the properties are stored in a background process after the notifications have already being sent.

However, we can use the method `check_field_values` over the yet unprocessed properties (protecting those generated internally from being received from the client). Taking this into account, we can define a method in `Notification` as follows:

```

def allowed_installation ? installation, new_properties={}
  if new_properties.blank?
    installation . pass_filters ? self . installation_filters , self . installation_filters_relation
  else
    InstallationFilter . check_field_values installation . latest_field_values (new_properties)
  end
end

```

where the new properties are the ones received through the API already filtered and merged to the already existing values. This method is added to every API method that sends notifications.

Finally, we should just use the view helper to choose installation filters in the forms to edit and create messages and updates. The user will be able to define the target to be notified as a combination between the installation versions and the installation filters previously defined in the database. This can be seen in Figure 4.5.

Acme, Inc. Network Service

My Account Log out Help

Start Page Products Updates Messages Reports Installations Clients Alarms Administration

Messages Overview
New Message

The Message section allows you to create, edit and delete messages. You can also view reports for specific messages.

Adding New Message

Select Destination

Product Versions

- Acme Product
- 10.2
- 10.5
- 11.0-beta

Filter Installations

- Linux Users

Add filter:

Join Method: all filters (AND)

Tip You can manage the filters in the administration section.

SAVE BACK

Figure 4.5 Installation Filters: Choosing notification destinations..

Reports Generation

We already have the ability to store the required data in the server, tools to manage installation groups and to perform operations and searches based on them.

However, our application still lacks an easy way to access the information. The next step will be to add the ability to get detailed information about the installations and their usage. This includes the following:

- Number of Installations: grouped by product, by version or by specific property value. With the ability to filter those statistics by previously defined groups.
- Notification Reach: number of installations that have received a particular message or update. How many are pending to be notified? How is it reaching different versions, installation groups or installations with specific property values?
- Evolution over time of a chosen property for a specific installation. For example, "Free/Used RAM Memory"
- Gantt diagrams with information about release dates and time periods where each notification is active.

We will divide the implementation in three steps. First, we will develop a plugin which will have the common tools to store and manage the information that is going to be presented as charts or Gantt diagrams. Then, we will implement the specific way to store and generate the reports in Network Service context. Finally, we will improve some aspects of the system.

Reports generation plugin

Although some parts of the reports implementation are really specific to the application, we can identify some things that may be useful in other projects. As a first step, we will develop a simple plugin which will be helpful to provide the necessary tools

to store, modify and represent resolution based data as well as events with certain duration over time. In other words, we will include the tools to generate time based reports in different formats.

The main objectives to be fulfilled are:

- Define conventions to work with resolutions and lengths.
- Provide a common way to store numeric data over time (`Datasets`) to be used in charts or events with durations (`EventList`) to be used in Gantt diagrams.
- Transform defined data into new lengths and resolutions.
- Merge new information over an already initialized set.
- Filter, reorder, accumulate and other simple modifications to datasets.
- Charts generation, simplifying integration process with new graphic libraries.
- Export information to different formats: CSV, XLS ...

It is important to note that gathering data is not included as part of this plugin since it is application specific, being implemented as part of the Network Service code as we will describe in the next sections.

The file structure that will be followed can be seen below:

```

vendor/plugins / reports_generation
|-- Rakefile
|-- init .rb
'-- lib
    |-- datasets .rb
    |-- event_list .rb
    |-- reports_generation .rb
    |-- default .rb
    |-- graph
    |   |-- default .rb
    |   |-- generator_gruff .rb
    |   |-- generator_plotr .rb
    |   |-- graph.rb
    |   '--- prototype_generator .rb
    |-- file_tools .rb
    '--- time_tools.rb

```

The main elements to consider are `Datasets` and `EventList`, which are the classes designed to store the information with time information and events representable on a Gantt diagram. Files inside the `graph` folder include the tools to generate the charts. Each graph library will have a generator as we will see below. `FileTools` and `TimeTools` are defined as modules with a list of helpers used by main classes.

Table 5.1 AllowedResolutions.

Number	Resolution Unit
1	years
1, 3, 6	months
1	days
1, 3, 6, 12	hours
1, 5, 15, 30	minutes

Conventions

We need to define a set of valid resolutions and lengths that will be used through all the application. We represent both of them as a pair of integers, one of them represents the time unit used (days, hours, minutes ...) and the other the number of units. The reason for storing time units as integers, apart from reducing the storage needs, is that we could do comparisons easily with numerical operators like '<', '>' which will be faster than using strings.

To simplify the processing and transformations between different resolutions, we limit the possible values to a subset of those which have the property of being a divider of bigger resolutions. In other words, it's possible to work with 3 hours because it is easily transformed into 1 day resolution by grouping 8 data points. The complete list of valid resolutions can be seen in Table 5.1.

Report lengths will have similar restrictions since they are formed by a set of points with certain resolution.

On the other hand, in order to be able to modify the resolution in the generated reports, it will be necessary to have a set of aggregation methods to obtain the new values based on higher resolution ones. We will include the following aggregation methods: average, maximum, minimum, sum, last value, first value.

Datasets

`Datasets` object is the main element able to store and manage the different sets of data which may be presented on a chart. It provides flexibility to store simple information as well as to perform complex operations over data.

Data is stored following a structure based on slots. Resolution and length of the represented data may be set, so the slots and boundaries are known before inserting any information. Furthermore, resolutions may be defined for each section, allowing a behavior similar to the one described in previous chapters to loose resolution as data gets older.

The information can be inserted at once or in an incremental way. Different statuses are defined for each slot or data point (*empty*: null was inserted, *full*: something not null inserted, *loaded*: something was inserted, *not loaded*: nothing inserted yet).

They are updated based on how the information gets inserted, being possible to easily detect which slots of time are empty and which are not loaded. This will allow us to generate and cache reports in an incremental way as we will see later.

To be able to support average aggregation methods, a counter is associated with every slot. It will be incremented with each insertion or can be set directly to specific values when the data is added.

Datasets also provide some transformations:

- Alter resolution.
- Modify time boundaries or include new periods of time.
- Modify slot statuses.
- Merge sets together.
- Get n most relevant sets accumulating the rest as "others".
- Generate a "total" dataset which combines the rest.

Another important aspect is that the object has the ability to be imported and exported as a Hash element which simplifies the storage in any database or in memory.

Graph Generation

In general we may want to use different graphic libraries minimizing the changes in our application code in case we want to move from one to another. In order to simplify it, we define graph generators which act as an interface between the Datasets object and the way the library receives the information. For instance, the library may require to write the data in JSON or just put it inside an XML separated by whitespaces although the data to be represented is be the same.

Common configuration options such as "title", "width", "height", "type of chart" should be mapped inside each generator to the specific library parameter. For example, we define a type called "line" which may have different names on different libraries and can be configured in different ways.

All generators should inherit from the prototype class, which declares the methods that will be used to get the chart and defines a set of methods useful during any library chart generation.

The methods to be overwritten in each generator are:

```
# Returns the code which should be inserted where chart should appear
def generate_image
  ""
end

# Returns any script code that should be loaded
def generate_script_code
  nil
end
```

```

end

# May be used to prevent the chart to be generated multiple times
def chart_cached
  false
end

```

The reason for having separate method for the script code is to be able to move it to the end of the page load, which is useful to speed up rendering the page.

A couple of generators are defined as part of the plugin:

- **Gruff** Written by Geoffrey Grosenbach. It is based on RMagick gem and ImageMagick library. It generates images dynamically based on the datasets provided. As a drawback the time to generate a new chart is big compared to other solutions and increases the load in the server. An advantage is that it provides an easy way to cache the charts storing them as images, being only required to check if the file already exists before creating it. It also allows the user to see the charts without having to install flash players or any additional software. An example of chart generated with Gruff can be seen in figure 5.1.

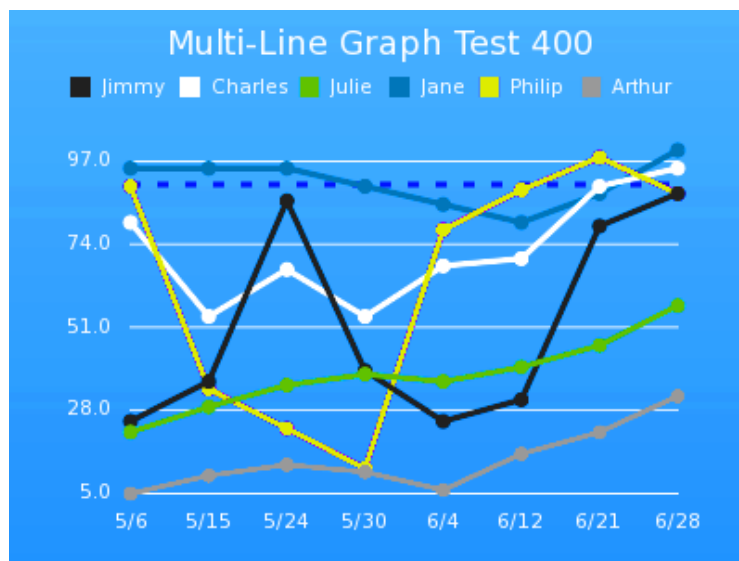


Figure 5.1 Example of chart generated with Gruff.

- **Plotr** Written by Bas Wenneker. It is based in Javascript and Prototype library. In this case the processing of the data to generate the chart is done on client side. It doesn't require to have flash player installed in client side, but only javascript which is usually included and enabled by default in modern browsers. An example of chart generated with Plotr can be seen in figure 5.2.

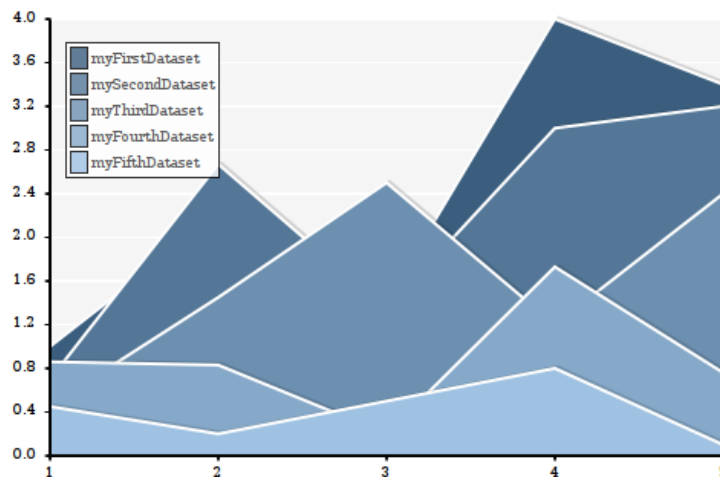


Figure 5.2 Example of chart generated with Plotr.

EventList

Not all the information to be shown on a report is based on having values for a set of points. We may also be able to define "events", which are elements which happens in a specific moment or which are active during a period of time.

In this case, the information to be stored is just a list of elements with title, description and time boundaries, being possible to use only start time for those without duration.

In order to represent the information contained in an EventList object we add support to show it through "Simile Timeline" library. Written by David François Huynh and maintained as part of Simile Project. It is based in Javascript and generates an interactive line of time containing elements with or without duration. Those elements may contain a description which is shown as a popup when clicking on them. An example can be seen in figure 5.3

Network Service Reports system

On this section we are going to implement the reports data acquisition. As well as the general structure which will be used to deal with them. Both `FilterModels` and `ReportsGeneration` plugins will be used.

General Structure

How can we represent a report as data objects? In general, a report will consist in a group of different elements (charts, tables) which will contain information about a particular object (product, version, installation, notification). Each piece of

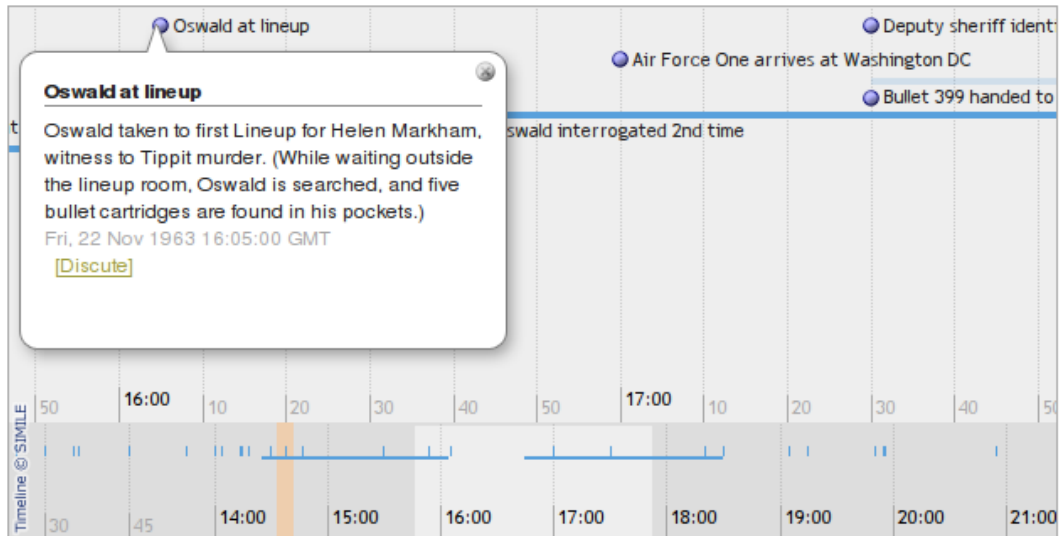


Figure 5.3 Example of chart generated with Simile Timeline.

information shown will be stored as a `ReportElement` and the group of elements will form a `Report`.

While getting the data from the database, the information has to be filtered and grouped in the way we are interested (by product, by version, by property value, by property key). In order to achieve this, we use the class `ReportDataFilter` which will have the methods to get the information based on `InstallationFilter`.

However, we will have at least two different sources to obtain the results: the database and the cache. Each element will have one or more sources depending on the object which the report is based on. We add a new element called `ReportDataSource`, which will be specific for a particular object, and chooses between cache and the database to return the information to report elements.

The structure followed to represent a report can be seen in the diagram 5.4.

This means we will include the following elements:

- `Report`: is the global description, it has some general properties like the title and the format to be used. It contains a set of report elements.
- `ReportElement`: presents a piece of data that is shown in the report, for example the total number of users shown as a pie chart. It has information about the time limits or if it represent total or latest values received. It also has the information about the specific format to be used. A `ReportElement` will be associated to one or more `ReportDataSource`
- `ReportDataSource`: it represents a source of information, without any time or format information which will be received from `ReportElement` requests. It associates an object and a specific report data filter. It will take data from cache or from database by using report filter methods.

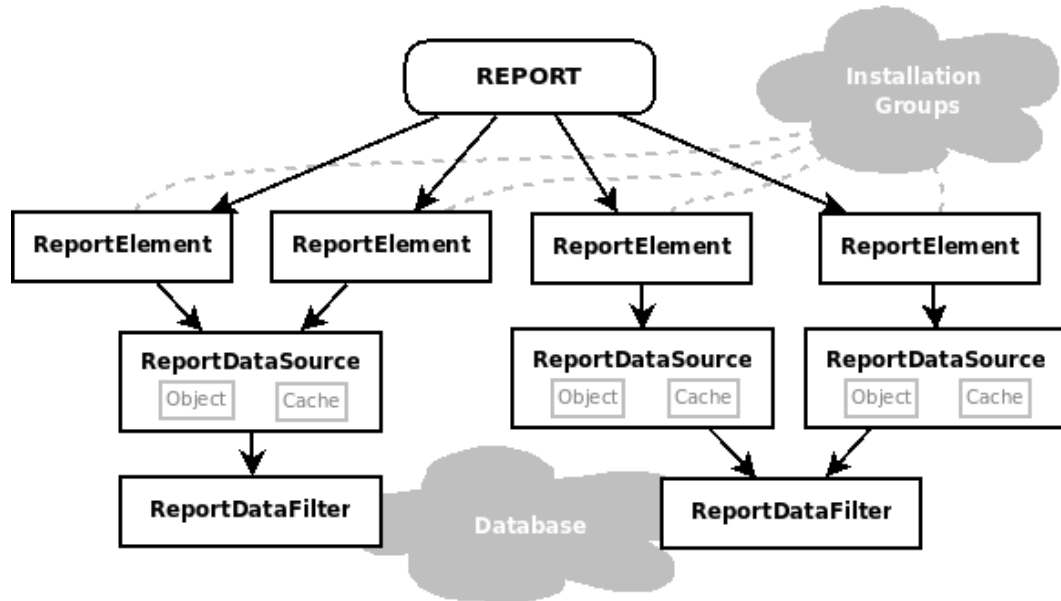


Figure 5.4 Reports General Structure.

- `ReportDataFilter`: it represents the restrictions and groupings necessary to get the information from database. It can be used by one or more `ReportDataSource`.

The only model which is assigned to a specific object is the source of data. That's why `ReportDataSource` elements should be generated dynamically. That way, we can define reports that could be applied to different installations or notifications without having to define or configure them for each element. Also, new generated elements will have all the reports already defined for them.

To be able to behave in that way, `Report` will be customized with a specific object at rendering time. That is, we will define reports for kinds of objects: installations, messages, updates, products. Then, we will be able to apply those reports for any object inside that group of elements. In this case, `ReportElement` should contain a reference to the specific `ReportDataFilter` that should be applied to get the information, being generated the source element as a join between that filter and the customized object.

Filtering

We should be able to get the information we are interested in from the database. Basically, we want to get:

- Number of installations by version and property value
- Number of installations reached by a particular notification by version and property value

- Evolution of specific property values for a given installation

Adding also the ability to use already defined installation groups or filters. That is, the user should be able to show the stats for those inside one, none or all the filters selected.

As we have said before, this filtering will be located in model `ReportDataFilter`. We will define a method called `get_datasets` which will return the stats with and without time information, including counters in case they are necessary. For instance, they may be useful to perform aggregations when showing the evolution of a numeric property over the time.

Indeed, most of the work has already been done in `Filter Models` plugin. Each property is already defined as a `field` which may be used in the options passed to `find_with_filter` method (`select_fields`, `include_field` and `group_fields`).

Using filter definitions instead of directly performing the joins manually is a powerful way to make the reports easily adaptable to database modifications. In case new fields are added, we just need to include a new field definition, being possible to group or show any reports based on it after that without any other internal change.

Current statistics

The first step will consist in getting the latest stats. That is, the numbers at current time, without showing any historical evolution. All the tools we need to get this is `FilterModels` plugin and the options we have mentioned before.

For instance, to get the number of installations by operating system we may do something as follows:

```
stats = Installation . find_with_filters :all, filters, filter_rule,
  :group_fields => {:prop_os=> {}},
  :select_fields => {
    :prop_os => {:as => "dataset_name"},
    :installation => {:method => "COUNT", :distinct => true, :as => "number"}
  }
result = {}
stats .each { |e| results [e['dataset_name']] = e['number'] }
```

where `filters` and `filter rule` contains the list of filters and the rule to join them (one,none,all).

However, additional restrictions should be made. First, some groups will be banned by default:

- "Inactive" installations: that has not contacted the server after a period of time.
- "Testing" installations: which are done by the software provider with testing purposes.

More details about both groups may be seen in section 4.3.1. Although they are not added by default we add a pair of checkboxes so it is possible to obtain reports including them on a similar way as we did while we were filtering installations.

Also, it is necessary to restrict the counts to those installations which are related to the object which is the report target (a specific product, a notification..).

To take all this into account we generate a temporary filter which will be added to the query containing the additional rules. Using the option `force_first_filter` so all installations have to pass that filter not taking into account the rule applied to join the rest of filters.

For example, to get the numbers for a specific product we may do:

```
f = InstallationFilter .new
f.add :i_active , :eql, true unless include_inactive
f.add :i_testing , :neq, true unless include_testing
f.add :product, :eql, Product.find(product_id)
filters = [f] + filters
stats = Installation . find_with_filters :all, filters , filter_rule ,
      : force_first_filter => true, ...
```

where `product_id` contains the identifier of the product the report will be talking about and, `include_active`, `include_testing` allows the user to include those groups in case it is necessary.

Following a similar approach, we are able to generate the rules for each possible object:

```
case object
when Product then filter .add :product, :eql, object.id
when Installation then filter .add :installation , :eql, object.id
when Client then filter .add :client , :eql, object.id
when Notification then filter .add :notified_with , :eql, object.id
end
```

Evolution over time

However, we do have a problem still. How could we get historical values? All the information we are getting until now are just the latest values. In order to obtain the evolution over time we are going to alter the conditions generated by the FilterModels plugin replacing the joins associated with those fields over time.

The complexity of doing so appears because the properties are stored using variable slots as we studied in chapter 3. When we are showing the chart, the space between the points is fixed so we need to translate the database contents to chart space. The solution consists in generating a vector of times at SQL level which is used to join the properties accordingly over the time:

```
SELECT times.start_time AS time, ... LEFT JOIN (
  SELECT '2014-10-23 00:00:00' as start_time,
         '2014-10-24 00:00:00' as end_time UNION ALL
```

```

SELECT '2014-10-24 00:00:00' as start_time,
       '2014-10-25 00:00:00' as end_time UNION ALL
SELECT '2014-10-25 00:00:00' as start_time,
       '2014-10-26 00:00:00' as end_time UNION ALL
...
) as times ...

```

Which provides total control over the slots of time that will be using to show the information. We use pairs of times to make it easier to perform the operations and to join property values. The start time of each slot is taken as the time to be returned to represent the location over the time for each value. It's important to note that generating this vector doesn't have appreciable impact in MySQL response times.

Once we have the times defined, we have to join appropriately the properties. They will vary depending on the type of report we are doing. In order to keep using InstallationFilters and current field definitions, we just have to replace the default join condition "latest = true" in the generated joins:

```
... JOIN table_name ON .... AND table_name.latest = true ...
```

with the ranges of times where the property value should be taken from:

- Without taking into account blank spaces:

```
start_at < end_time AND (extended_end_at IS NULL OR extended_end_at >= end_time)
```

- Taking blank spaces into account:

```
start_at < end_time AND end_at > start_time
```

- Property values at installation creation time:

```
start_at <= installations . created_at AND (extended_end_at IS NULL OR extended_end_at > installations .
created_at )
```

- Property values the first time a notification was send:

```
start_at <= notification_reaches . first_sent_at AND (extended_end_at IS NULL OR extended_end_at >
notification_reaches . first_sent_at )
```

We would need to do the modifications for each property involved in the query. We may simplify it by using regular expressions. For instance, for the second case we may do as follows to alter join string.

```
join ,gsub !(/ s+(\w+). latest \s*=\s*true/, " #{$1}. start_at < #{$1}.end_time AND #{$1}.end_at > #{$1}.
start_time" )
```

When we are talking about generating historical number of installations we may think of two different approaches. Showing the total values over the time or the new installations over time:

- Total values: installation should be created before the slot ends; properties are joined without taking into account blank spaces.
- New installations: 'installation' should be created inside the slot; properties are joined at installation creation time.

Another aspect to take into account is that the activity of the installation varies over the time. In other words, we can't use the 'active' flag to detect which are the 'active' installations over time. On the other hand, we will assume 'testing' flag doesn't change over the time, being possible to use it as we did for non historical reports. selected.

In order to track the history of both "active" and "registered" statuses, two new boolean properties are added. They will store the values of those flags chronologically. Instead of using the field `i_active` (based on the installations column) we use a new one defined as the rest of properties, `prop_active` (see section 4.3). The new rules are just:

```
filter .add :prop_active , :eq1, true unless include_active
filter .add :i_testing , :neq, true unless include_testing
```

Once we have the results from the database query, we may get a Hash representing the values obtained:

```
results = {}
temp.each do |v|
  results [v[:dataset_name]] ||= {}
  results [v[:dataset_name]][v[:time].to_time] = v[:number].to_i
end
```

After this transformation, we have data compatible with `ReportsGeneration::Datasets`.

Notification reach

When we talk about a notification reach report, we are referring to get information about how a message or update is being sent to the end-users. Each time a notification is sent to a new installation a `NotificationReach` entry is generated. A counter is incremented in case it is sent more times to the same installation, which may be useful to track the number of downloads.

The approach to get this information is similar to the user count, joining the properties with the first time a notification was sent (as we show in section 5.2.2.2). Also, a specific filter field is used to restrict the results for a specific notification:

```
: notified_with => { :name => "Received Notification ",
  :type => : integer , : field => { : notification_reaches => : notification_id }
```

A similar concept to "new" and "total" installations is applied to notifications reach. In this case, it represents showing the total number of installations notified or the number of new notified installations over time. The only difference from database perspective is adding or not the second condition in:

```
WHERE ... notification_reaches . first_sent_at < end_time
AND notification_reaches . first_sent_at >= start_time ...
```

where `start_time` and `end_time` come from the static time vector.

As we are using installation filters all filters and groupings can be done easily as in previous section.

Numeric property value evolution

Another special case to consider consists on getting the history of a numeric property over time. Static values are in this case just the latest ones received, so no special queries are required. However, evolution over time should be studied.

Instead of counting different installation ids, we should get the values inside the slots defined in the time vector applying appropriate aggregation methods. To simplify the processing, the values shown will belong to a single installation.

This time, we should include additional information which has not been taken into account. On one hand, we should be able to distinguish different resources. For instance, if we are receiving the disk usage property for different disks (C:, D:), values are stored with different resource indexes.

A new field is added for each property so we are able to filter them properly:

```
fields [ "prop_#{p.key}_resource".to_sym ] = {
  :type => : integer , : field => { :hidden => true,
    " latest_installation_properties_ #{p.key}".downcase =>
    :resource_index }
}
```

Each time a report is generated, this resource index may be specified, being 0 taken as default. All affected properties will be filtered with that index:

```
filter .add "prop_#{p.key}_resource".to_sym, :eq1, resource_index
```

On the other hand, counters should be also included in the response. This provides the ability to perform aggregations over the results without doing additional requests to the database. Those counters should be scaled based on the length of the new slots, that is, if we are representing a big stored slot with several small ones, counters should be distributed accordingly. In other words, if counter has value 10 and a

length of 10 days, and we are representing it with 1 day resolution, the counters for each day should take one as value. The reason why we can suppose that uniform distribution came from the fact that the agent is using a constant rate to contact the server. Mathematically, for each slot, limited by `start_time` and `end_time`, we should sum all counters partially inside:

$$C = \sum_i counter_i \cdot \frac{MIN(end_time, end_i) - MAX(start_time, start_i)}{end_i - start_i}$$

In order to translate this into a MySQL query, we need to simulate minimum and maximum between columns. Also, it will be required to transform time objects to numbers so we are able to divide and multiply.

```
SELECT CAST(
SUM({table}.counter * (
TIME_TO_SEC(TIMEDIFF(
IF({table}.end_at < end_time, {table}.end_at, end_time),
IF({table}.start_at < start_time, start_time, {table}.start_at)
)) /
TIME_TO_SEC(TIMEDIFF({table}.start_at, {table}.end_at))
)
AS UNSIGNED
) AS {prop_key}_counter
```

Where `{table}` and `{prop_key}` should be filled for each involved property with the table name containing the values and the property key. `CAST` method has been added to obtain an unsigned integer as result.

Multiple properties may be gathered together. Since we are performing the analysis over a single installation and installation filters are not required, we are able to get all of them with a single database query. In order to achieve this, we select the fields (value and counter) with names identified by property: `<key>_value` and `<key>_counter`.

It's important to notice that properties should be joined taking blank spaces into account as we specified in section 5.2.2.2. This join selects for each slot all the properties that are partially inside it. Blank spaces will contain a `nil` value as no values are joined.

Once we are able to get the counters we should perform the aggregations taking into account the aggregation settings stored in `PropertyType` objects:

- **Maximum / minimum value**

In this cases scaled counter is not necessary to obtain the values, we can directly use MySQL aggregation methods `MAX` and `MIN`:

```
SELECT MAX({table}.value) AS {prop_key}_value, ...
SELECT MIN({table}.value) AS {prop_key}_value, ...
```

- **First / last value**

To get the last and first value inside a slot we use that applying MAX method to a list with NULL and NOT NULL elements will return non NULL as result. We just force to NULL all the elements but the first or the last inside the slot:

```
// First
MAX(IF({table}.'start_at' <= start_time , {table}.'value' , NULL))
// Last
MAX(IF({table}.'end_at' >= end_time, {table}.'value' , NULL))
```

- **Average**

In this case, we generate the average using the scaled counter, which is defined above but removing CAST and SUM methods.

```
SUM({table}.'value' * {scaled_counter}) / SUM({scaled_counter})
```

- **Sum**

To keep sums coherent, we should take scaled counter into account. We distribute the area uniformly. That way, if we get the report with a big resolution and then transform it into one with bigger slots the numbers are correct.

```
SUM({table}.'value' * {scaled_counter})
```

For each property we should include two elements in SELECT statement, one for the counters and another for the aggregated value. In both of them {table} and {prop_key} should be modified.

After the query is executed, we will get a Hash containing the results for each property in separated keys. We may get the results with a simple loop:

```
query_results .each do |v|
  time = v[:time] && v[:time].to_time(: utc) . localtime
  properties .each do |prop|
    results [prop.id] ||= {}
    counters [prop.id] ||= {}
    if v["#{prop.key}_value" . to_sym]
      results [prop.id][time] = v["#{prop.key}_value" . to_sym]
      counters [prop.id][time] = v["#{prop.key}_counter" . to_sym]
    end
  end
end
```

Both results and counters may be directly used to initialize a Datasets object after this change.

Background Processing

Some reports may have complex rules which translates in long loading times. If we generate them directly while loading the view the page may time out. In order to avoid this, we will generate the reports in a background process.

When a report is loaded through the browser, a new job is generated with information about the time limits. After that, using AJAX, the server will be polled, being shown to the user as soon as the data gets loaded.

To not affect to API processing we will use Beanstalk "tube" feature: we may insert the jobs in different queues called tubes. Then, we could have specific workers listening to one or several of them using `Q.watch(tube)` configuration method.

We add a new tube called "reports" with multiple workers. This tube will be use to process slow operations which does not have tight time restrictions: generating reports, exporting installation lists to csv or xml, etc. This way, background processing related with the API will not be affected since different workers will take care of them.

This method is not perfect though. If the report is taking a long time the user may think than the server has failed and he may try refreshing the page several times. The result is that the queue may be full with the same long processing report cloned jobs. Being possible to affect the processing of any other unrelated reports or doing exports. For that reason, we have to improve it by avoiding repeated jobs and not processing any already loaded sections.

Iterative Loading and caching slots

As we have said before, complex reports may take some time to be processed. Furthermore, we don't have any way to avoid generating the same reports multiple times. This may be resolved by inserting a cache system. This new element should take into account how the reports are shown and how data is being changed.

In our case, the information is being inserted chronologically. After some time, data become constant, being only modified in those cases where we are losing resolution of property history. Usually, when an user shows a chart old points will get the same result being only necessary to load certain sections. This specially true if we are preloading the reports, having to load the most recent data only.

How could we track those sections to be loaded? As we described in section 5.1.2, we included the ability to track time slot statuses in `ReportsGeneration::Datasets`. Our cache will use this feature, following this process while loading the information at source level:

- `Datasets` element will be stored in Memcache, being imported and exported in Hash format.

- Each time the source of data is used it will try to import the existent Datasets, else a new one will be generated with appropriate resolution.
- Resolution is updated to force loosing resolution in case.
- All missing time slots which are going to be represented are added in NO_LOADED state.
- Cut anything outside of report scope.
- Get list of NO_LOADED slots.
- Generate a new job with that list.
- Return Datasets object with the slots already loaded.

As we can see, `ReportDataSource` will return a `Datasets` object to `ReportElement` with already loaded information, generating any jobs necessary to load pending data. Then, `ReportElement` is able to use any generators to represent that information.

On client side, the webpage should be able to detect the processing status. If any report element is not fully loaded, a javascript timer is started. Each time it ends, an AJAX request will be sent to the server. It will answer sending the code to update the `ReportElement` in case new data is available. When the element is fully loaded, the code received should stop the timer. Using this approach the user is able to see loading status.

To identify which is the exact request an unique key is generated. That key will be sent to the server as parameter with all the AJAX requests. It will also used as key in Memcache to store the state of loading process. Below can be seen how the state value is modified by both client and workers:

- On each client request:
 - Initial request & pending slots → STATE = 1
 - STATE > LIMIT → STATE = 1 & regenerate jobs
 - STATE == NEW_DATA → STATE = 1
 - STATE == FINISHED → delete STATE
 - else → STATE += 1
- Each time worker loads a chunk of data:
 - STATE has expired → remove job
 - Pending slots to load → STATE = NEW_DATA
 - else → STATE = FINISHED

When a new request is performed, state is set to 1, being incremented each time the client polls the server and the worker has not loaded new data. If it reaches certain limit we suppose that the worker has had some kind of problem and regenerate the jobs restarting the state to 1. Each time a worker processes a chunk of information it will set the status to a know code called `NEW_DATA` if it is not finished yet or, to `FINISHED` if that's the case.

When the client polls the server and it is at `NEW_DATA` state the chart is updated and state is set to 1 again. However, if it is at `FINISHED` state the chart will be loaded, the poll stopped and state removed.

As we have said, pending slots are processed by workers. That is, a new job with the list of time slots pending to be processed is generated. In order to prevent processing multiple times the same slots, we introduce a shared array of pending slots which will be stored in *Memcache*.

Each worker will behave as follows:

- Get the job from the queue.
- Look for the affected source.
- Take a number of slots from the job which are present in shared list.
- Update shared list.
- Get their data and update source element:
 - Source cache gets loaded adding new time section
 - Do the database query
 - Update real user access time
 - Export the source to cache
- Update loading state.
- Generate a new job with the rest of the slots, removing those which are no longer in shared list.

This iterative loading allows us to provide feedback to the user about the progress. It also avoids long queries keeping workers busy: if there are multiple elements to be loaded all of them are progressing simultaneously. Additionally, it provides the ability to stop loading them in case they are no longer needed. For instance, user may stop it, change the time boundaries or close the page, not being longer necessary to show the result.

Even though we are using a cache to avoid reprocessing already shown sections, it is important to assure that we reload the values which are not definitive. In our case the

information is inserted chronologically, which means that the most recent slots are the ones that may vary.

We should set a reasonable expiration time for those slots so they are reloaded in the near future. In order to do so, we store the time when it was loaded. If current time is greater than a margin those final slots will be removed and reprocessed, storing the new time as the new loading time. Expiration time should be set according to the resolution being at least $1.5 \cdot SlotTime$.

Memcached Collisions

We use Memcached for loading state and shared slots variables because it provides shared storage between the different processes: Rails mongrels and Merb workers. Using this shared space may cause collision problems which should be taken into account. SET and DELETE are not atomic operations.

However, we may use ADD which is atomic to define locks which will prevent multiple processes collisions while modifying other entries with SET, DELETE. The idea consists in storing a new key: "lock : {key}" which existence means that a process has taken control over the entry {key} in Memcache.

If two processes run ADD simultaneously only one will receive "STORED\r\n" message. We may add a couple of methods to MemCache class as follows:

```
def get_lock key, options={}
  opt = { :release_time => 10, :wait_time => 0.25, :wait_limit => 5 }.merge(options)
  waiting = 0
  while add("lock::#{key}", true, opt[:release_time]) != "STORED\r\n"
    sleep(opt[:wait_time])
    waiting += opt[:wait_time]
    return nil if waiting > opt[:wait_limit]
  end
  true
end

def remove_lock key
  delete("lock::#{key}")
end
```

Each time a shared key is going to be modified, a call to `get_lock` should be made. The lock may have an expiration time avoiding orphan lock keys if the process got killed for some reason. Also, we may configure how many times should retry getting the lock and which interval to use. As soon as it is possible `remove_lock` should be called to free the key.

Cache Preloading

The number of users in Network Service frontend is reduced. For that reason, it is important to have a way to speed up the process that is not based on "the first request generates the cache and the following ones are faster". We are going to simulate that first user by a cron task which refresh the last data in the reports. That way, when a user comes to the server, he can see information faster.

Since the user may be using any combination of filters, we can't cache all possible reports. Instead of doing so, we restrict the operation to the sources which are being used. Each time a real user gets a report, we update a time stamp in the related source. The information is preloaded unless that time stamp is older than a certain limit. In that case, all associated cache is removed.

During the preloading, old cached points get removed unless the report has been recently used. Basically, we limit the total number of points stored in cache. This is useful to prevent it from growing indefinitely.

In order to have full access to all the cached information, a new element is defined in Memcache space. It will store the list of keys that contain report information. Following this approach, we are able to use the vector to easily detect all the sources that may be preloaded. It also provides the ability to remove all cached information related to reports easily, removing the keys contained in it. It is not necessary to store that vector in a persistent database since losing the information stored in Memcache would mean to lose all the described cache as well.

Views Generation

In this section we are going to describe how the reports are included as part of Network Service interface. As we described when we started talking about the reports, we will have three kinds of report:







- Number of Installations: how much people is using the application.
- Notification Reach: how much people received a notification.
- Property type evolution: how is a property changing.

Chart generation

We decide to use Plotr graphic library as the default chart generation. Between the advantages we may list the following ones:

- Since it is based on Javascript, it will reduce the load in the server - no image generation is needed at server side.
- No additional software required. Javascript is fully supported by default in most of the browsers.
- Better user interaction.

Each time a chart is rendered a set of controls will be added to allow the user vary the information shown:

-  let you move to the previous or next values in time.
-  allows you to choose the time interval to be shown.
-  changes the chart type (line, area, bars, etc.).
-   switches between the total number or accumulated values view to changes or new entries view.
-  increases or decreases resolution

When we are talking about showing accumulated or new values, we are referring to alter the view between absolute values or the increments over time. In other words, if we are representing the number of users we may get the total number of users or how many new users do we have for each day

While the chart is being processed, a loading message will appear instead of the controls. Next to that text, a stop button allows to cancel the operation and switch to the control list so the user may vary the conditions.



Number of Installations

A really important information from ISVs perspective is the number of users they have and how it is changing over time. We include a new main section called "Reports" which objective is to provide that information.

Initially, only a report with the number of installations by product version. To simplify the process of adding new reports, we implement a default template that allows the user to define new reports by only choosing the property to use to group the users by. In other words, the number of users will be presented by the different values contained in that property. For instance, we may get the statistics by country or by operating system.

This template includes a report with two report elements. The first one shows a pie chart with the latest values. The second element represents the evolution in the number of users in the last 30 days. In order to get more exact information, a table with the all current values is included next to the pie chart.

When entering the reports section a list with all the defined reports is shown. Report elements are shown using a smaller size. This view provides a general overview about what is going on. You can see how it looks like at figure 5.5.

Each report has also its own page accessible using a menu which appear in main view. The main difference is that report elements are shown in greater detail. An example can be seen on figure 5.6.

Furthermore, all defined reports can be applied to any specific group of users defined as installation filters or, as a combination of them (using OR, AND and

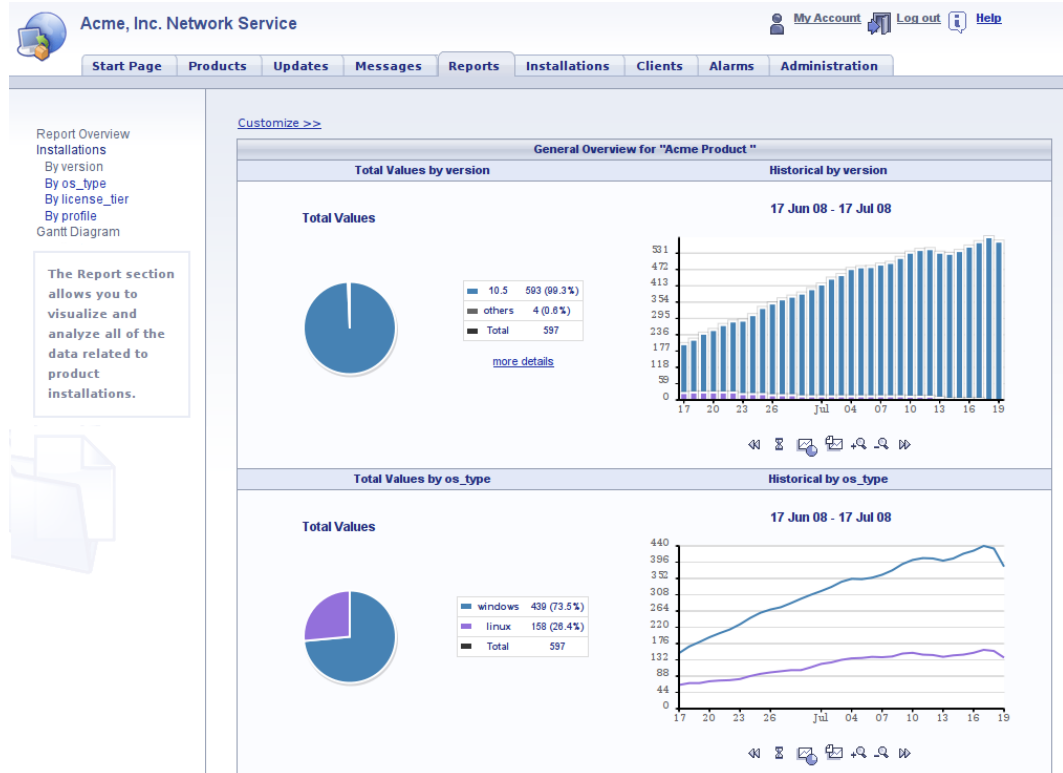


Figure 5.5 Reports: number of users, main view..

NAND rules). In order to do so, we apply the same form helpers that we used to search installations described in section 4.2.2.8. For example, we can get the number of installations by operating system on a specific country.

By default both *testing* and *active* installations are not taken into account. However, we also include the possibility to count them in the statistics by selecting their associated checkboxes in customization form.

Notification Reports

A similar approach to main reports section is followed. We also define a template which includes both latest and evolution over time elements. Installation filters can be also applied and active and testing installations may be taken into account.

In this case, the main view is just the total number of installations that have received the notification over time and a small table with information about the status. As you can see in figure 5.7, it includes the number of is installations that have been and are pending to be notified. Those numbers take into account the versions and filters that were configured during notification definition.

Other specific reports will have a similar view to the one used for number of installations.

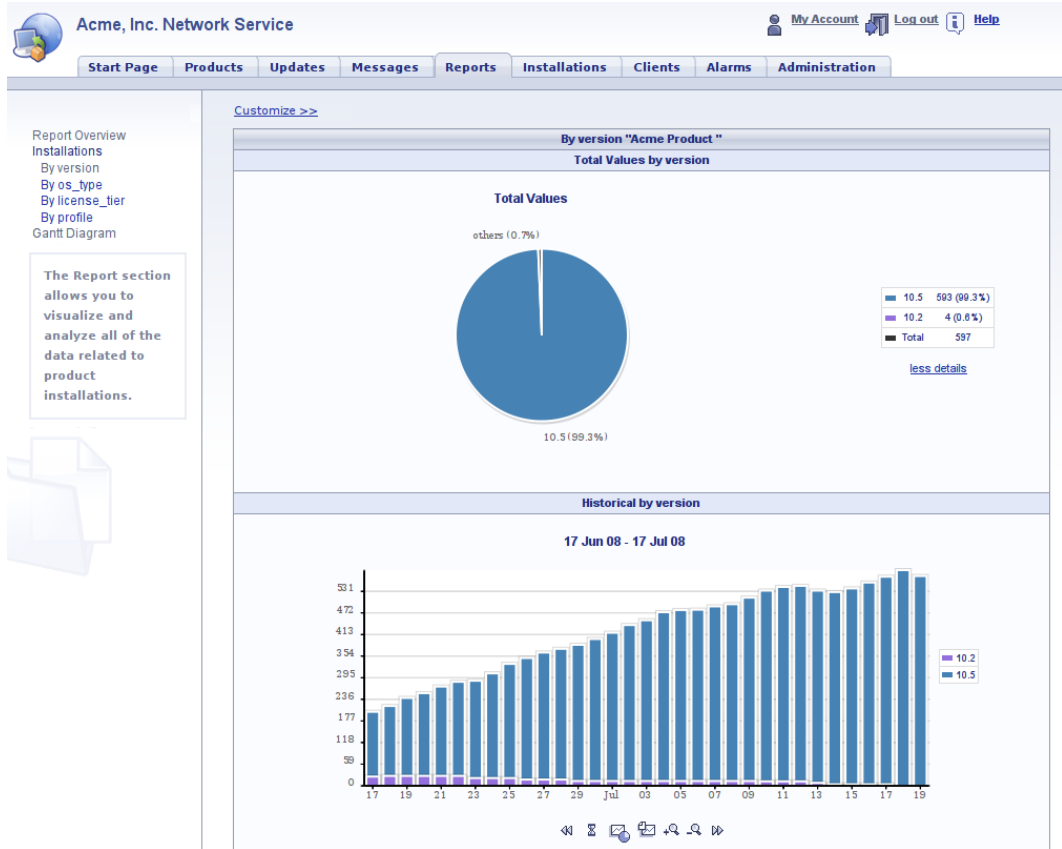


Figure 5.6 Reports: number of users, detailed view..

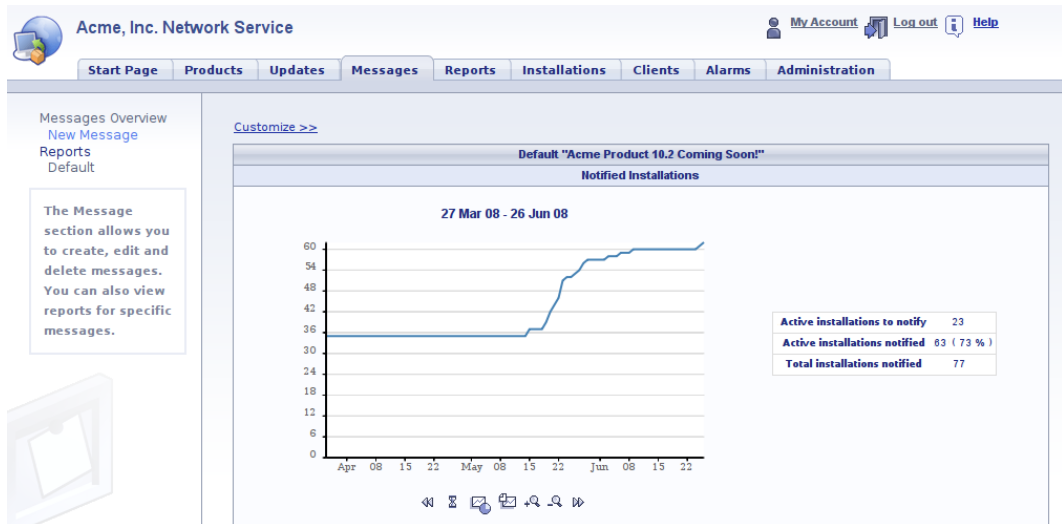


Figure 5.7 Reports: notification reach, main report..

Property value evolution

Another important aspect is to be able not only to check current status for a specific installation but also check the evolution over the time. This is particularly useful to track the end-user machine behaviour (memory, cpu, disk usage).

In this case, we show a report element which shows the evolution over time of a group of properties. We also include a table with the latest received values and latest contact time.

We show groups of properties instead of separated charts for each one because it is more useful to represent the properties related to the specific resources together. That way, we can have a chart showing memory stats together as we can see in figure 5.8. Similar charts may be added for disk and cpu usage or other properties.

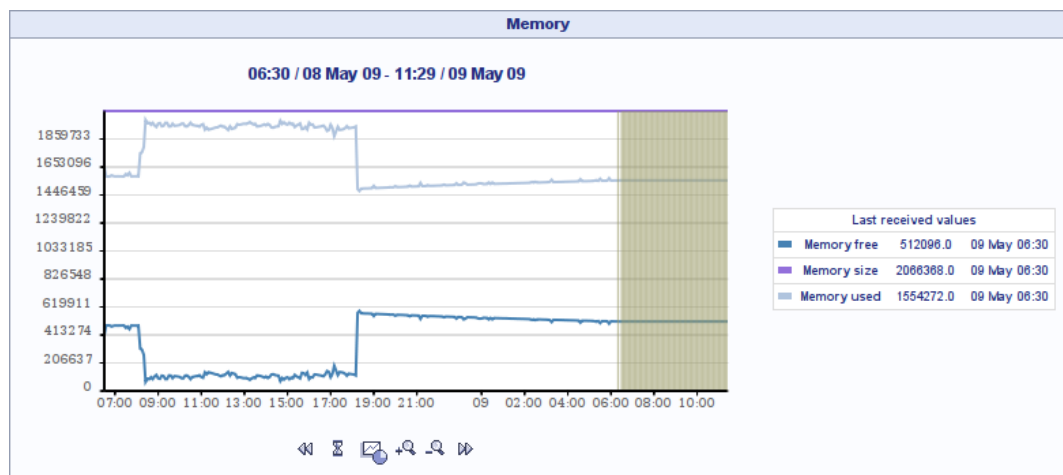


Figure 5.8 Reports: property value evolution..

In order to clearly show empty regions, we have extended Plotr library to make it able to paint grey regions. We use that ability to mark periods of time without information, where installation has not contacted the server, to warn the user about the fact that the value has been guessed according to adjacent points.

While using different resolutions the aggregations used will be the ones configured in property types.

Conclusion

In general, both the present project and the global Network Service solution, have been implemented and put into production successfully. Achieving all the goals that we described in the initial chapters of this document. Which we could resume as:

- Gathering, storage and analysis of real end-user based on custom properties.
- Property-based and controlled percentage messages and updates notifications.
- Reusable data filtering and graphics and reports generation.

The project has also been really interesting. For example, evolving the storage architecture, as it was required to apply advanced indexing structures and to improve the queries and the background data processing, trying to reach an acceptable performance without moving away of the SQL technologies. As well as discovering Ruby, Rails and the Merb frameworks, which have demonstrated how powerful they are for implementing fast evolving projects without a big team of developers behind the implementation (In comparison of other strict languages as Java).

Although some of the newer technologies available nowadays, will certainly help evolve the system into a more robust and scalable solution (i.e. applying new NoSQL and big data technologies), the system has been deployed and maintained in production with the described architecture successfully, processing hundreds of millions of property events without the need of any mayor redesign.

On the other hand, the decision of adding the complexity of including both Rails and Merb frameworks (with the automated scripts we added to avoid supporting two different code bases), proved to be correct, as we were able to take advantage of both the Rails Web extensions and the powerful modularity of Merb for the background processing. As it also demonstrates the fact that both frameworks themselves decided to join their code bases in the more recent Rails versions to put these advantages together.

About the project organization, the SCRUM methodology proved to be helpful in order to move fast and help focusing in the implementation of all the features. However, it required some time to adapt the rules to our specific needs, evolving the initial method philosophy to the reality of the team in each moment. SCRUM

helped this evolution thanks to the applied retrospectives and continuous incremental analysis of itself.

In other words, this project has been really useful internally and externally to the company involved in it, BitRock S.L, and its clients through a productized application. As well as a good starting point to get real world knowledge about SQL databases in production environments and helping introducing myself in the Ruby language and frameworks.

I would like to thank again BitRock S.L. for the opportunity of being involved in this project and allowing me to grow as a professional.

Bibliography

- [1] Sam Ruby, Dave Thomas, David Heinemeier Hansson, *Agile Web Development with Rails*. The Pragmatic Bookshelf, 3rd edition.
- [2] Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy D. Zawodny, Arjen Lentz and Derek J. Balling, *High Performance MySQL*. O'Reilly, 2nd Edition.
- [3] Dave Thomas, Andrew Hunt, *Programming Ruby: The Pragmatic Programmers' Guide*. Addison-Wesley Professional.
- [4] David Flanagan, Yukihiro Matsumoto, Why The Lucky Stiff, *The Ruby Programming Language*. O'Reilly.
- [5] Russ Olsen, *Design Patterns in Ruby*. Addison-Wesley Professional Ruby Series.
- [6] Chad Fowler, *Rails Recipes (Pragmatic Programmers)*. Pragmatic Bookshelf.
- [7] Foy Savas, *The Merb Way*. Addison-Wesley Professional Ruby Series.
- [8] Aaron Farnham, Brian W. Smith, Ben Burket, *Merb: What You Need To Know*. Apress.
- [9] Matt Pelletier, Zed Shaw, *Mongrel: Serving, Deploying, and Extending Your Ruby Applications*. Pearson Education.
- [10] Ken Schwaber, *Agile Project Management with Scrum*. Microsoft Press.
- [11] Rahul Sharma, *Nginx High Performance*. Packt Publishing.
- [12] Leonard Richardson, Sam Ruby, *RESTful Web Services*. O'Reilly.
- [13] David Chelimsky, Dan North, Aslak Hellesoy, Dave Astels, Bryan Helmkamp, Zach Dennis, Jacquelyn Carter, *The RSpec Book*. Pragmatic Bookshelf.
- [14] Jamis Buck, Aaron Huslage, *Beyond Rails with Capistrano: Managing Production Systems with Ruby, Python, Perl and More*. Addison Wesley.

Network Service User Manual

Contents

1	General Definitions	1
1.1	General Resources	1
1.2	Active, Registered and Testing Installations	1
1.3	Installation Filter / Installation Group	1
1.4	Notifications, Messages and Updates	1
1.5	Default Message/Update	1
2	How to create a ...	2
2.1	New Product	2
2.2	New Product Version	2
2.3	New Message	3
2.4	New Update	3
2.5	New Default Message or Update	4
2.6	New Alarm	4
2.7	New Client	5
2.8	New Installation Filter	5
2.9	Export a File With Installation Data	6
3	How to view data on ...	7
3.1	Number Of Installations	7
3.2	Installation Status and Monitoring Properties	7
3.3	Delivery of Specific Update or Message Notification	7
4	How to configure ...	8
4.1	My Account Properties	8
4.2	Installation Filters/Groups	8
4.3	Alarm Definitions	8
4.4	Testing or Registered Installations	8
5	Section Descriptions	9
5.1	General View	9
5.2	General Controls	9
5.3	Start Page	10
5.4	Products	10
5.4.1	Main view	10
5.4.2	Show Product And Manage Versions	11
5.4.3	New Product	11
5.4.4	Edit Product	12
5.5	Messages	12
5.5.1	Main view	13
5.5.2	Show Message	13
5.5.3	New Message	13
5.5.4	Edit Message	15
5.5.5	Message Reports	16
5.6	Updates	17
5.6.1	Main view	17
5.6.2	Show Update	18
5.6.3	New Update	18
5.6.4	Edit Update	21
5.6.5	Update Reports	22
5.7	Reports	23
5.7.1	Main View	23
5.7.2	Detailed Report	24
5.7.3	Gantt	25

5.8	Installations	26
5.8.1	Export Installations to CSV or XLS	27
5.8.2	Installation Dashboard	27
5.8.3	Installation Monitoring	29
5.8.4	Installation Alarms And Events	30
5.8.5	Manage Filters	30
5.8.5.1	Main View	31
5.8.5.2	Special Filters	31
5.8.5.3	New / Edit Installation Filter	31
5.8.6	Mass Tagging	32
5.8.7	Tag List	33
5.8.8	Tag Cloud	33
5.9	Clients	33
5.9.1	Main view	33
5.9.2	Show Client Details	34
5.9.3	New Client	34
5.9.4	Import Clients List	35
5.9.5	Edit Client	35
5.10	Alarms	35
5.10.1	Show Alarm	36
5.10.2	Alarms Definitions	36
5.10.2.1	Main View	36
5.10.2.2	Show Alarm Definition	36
5.10.2.3	New / Edit Alarm Definition	37
5.11	Administration	37
5.11.1	My account	38

Chapter 1

General Definitions

1.1 General Resources

Basic resources are *Products*, *Versions* and *Installations*.

Product identifies the application.

Version identifies a specific application release.

Installation represents an end user which has successfully performed the application setup.

1.2 Active, Registered and Testing Installations

They define general groups of installations which are used throughout the application. Their definitions can be seen in *Installations* → *Manage Filters* → *Special Filters*.

Active installations: Installations which has contacted the server in the last 30 days. Exact definition can be seen in "Active Installations" special filter. By default, all listings and reports take only this installations into account.

Registered installations: They are the ones which passes the rule defined by the user in the special filter "Registered Installations". It could be used to distinguish between free users and those which owns a license.

Testing installations: This group is used to filter out those installations which are generated during application tests. They will not be shown in the lists or reports. It's defined as a special filter called "Test Installations". For example, it could filter out those installations coming from certain IPs.

1.3 Installation Filter / Installation Group

We call installation groups to those groups of installations which passes a serie of rules, the rules are stored inside an object which we call installation filter and it can take into account any received property or installation status field.

Groups are useful to perform operations to specific subsets of users (like sending messages or updates) or to get reports based on them.

To manage the defined filters you should go to *Installations* → *Manage Filters*

1.4 Notifications, Messages and Updates

We call *Notifications* to both messages and updates that are sent to the user.

Messages are used to give information about the product.

Updates are used to notify the user about new releases or upgrades.

1.5 Default Message/Update

This is only used in very specific case in which the application only accepts one simple notification. In that case, *Default Message/Update* is the one that will be sent to end users. Only one Message or Update per product can be defined as default.

Chapter 2

How to create a ...

2.1 New Product

1. Go to the *Products* section
2. Select *New Product*
3. Fill in all required fields, which are denoted with an asterisk (*)
 - You may use the *Archived* status instead of deleting a product - all product data will be kept on the server
 - Choose *Default message* - it will be sent to your *basic tier* users.
4. Click on *Create* to finish

NOTE



This feature is currently only available to Network Service administrators.

2.2 New Product Version

1. Select the product in the *Products* section or from the *Start Page*
2. Select *Product Versions* and click on the *New* button
3. Fill in all required fields, which are denoted with an asterisk (*)
 - *Product GUID* is the unique ID used to identify your product version. You should use the same value in your application configuration to let your users properly connect to Network Service Server.
4. Click on *Create* to finish.

WARNING



Please do not change the Product GUID once the Product Version is in use and the first end user installations are already created.

2.3 New Message

1. Select the *Messages* tab
2. Click on *New Message* in the left menu
3. Fill in all notification fields:
 - a. Enter *Title* and *Webpage*
 - b. Click *Next* to continue creating the *Update*
 - c. Choose which versions of the product should receive that message
 - d. Select the installation groups that should receive the message. (You can manage those groups called *Installation Filters* in *Installations* → *Manage Filters*)
 - e. Click *Next* to continue creating the *Update*
 - f. Select the *Status*. An *Archived* message will be saved on the system but is no longer active. The *Draft* status enables you to enter messages to be sent at a future date into the system. Messages are sent to the specified end users when they are set to *Active*.
 - g. Select the appropriate option in the *Critical* field. *Critical* messages will be denoted with an exclamation point (!) when displayed to the end user.
 - h. Select the time period during which the message should be active and sent to your end users by clicking on the calendar icons and selecting the appropriate start and finish dates.
4. Click on *Create* to finish

2.4 New Update

1. Select the *Updates* section
2. Click on *New Update* in the left menu
3. Fill in all notification fields:
 - a. Enter *Title* and *Webpage*
 - b. Click *Next* to continue creating the *Update*
 - c. Select which versions of your product should receive the message
 - d. Select the installation groups that should receive the message. (You can manage those groups, called *Installation Filters*, in *Installations* → *Manage Filters*)
 - e. Click *Next* to continue creating the *Update*
 - f. Select the *Status*. An *Archived* update will be saved on the system but is no longer active. The *Draft* status enables you to enter updates to be sent at a future date into the system. Updates are sent to the specified end users when they are set to *Active*.
 - g. Setup the *critical* field. *Critical* updates will be denoted with an exclamation point (!) when displayed to the end user.
 - h. Select the time period during which the message should be active and sent to your end users by clicking on the calendar icons and selecting the appropriate start and finish dates.
 - i. Click *Next* to continue creating the *Update*
4. Fill in all *Update* fields: the URL to the file, the type of file, MD5, instructions to perform the update and other information
5. Click on *Create* to finish

2.5 New Default Message or Update

1. Create a new update or message as described above
2. Select the product in *Products* section or on the *Start Page*
3. Click on the *edit* button
4. Use the *Default Message* selector to choose the message / update you want to send.
5. Click the *Save* button to finish

NOTE



The default message / update notification will be sent to your *basic tier* users. It will be the same notification for all versions inside a product. If you want the default message to only be delivered to certain types of users, you can specify the user group by applying one or more filters in the destinations on the update or message destination page. To learn how to create a new filter, see the "How to Create New Installation Filter" section.

2.6 New Alarm

1. Go to the *Alarms* section
2. Choose *Alarm Definitions* in left menu
3. Click on *New*
4. Fill in all required fields, which are denoted with an asterisk (*)
 - The alarm notification will be sent via email if you check the option *Do you want to notify?*. It will be sent to the address that has been setup in the *Main Settings* as *Contact Mail Address* - only administrators may change this email address.
 - The *Priority* drop down menu allows you to select a number from 1 to 10, with 1 being the highest priority and 10 being the lowest.
 - Alarm definition is based on specific group of installations which is identified by the *Installation Filter* you need to choose. Use *Manage Filters* in the *Installations* section to modify *Installation Filters*.
 - You can add different *Alarm Rules*. The rule may be based on numeric or string comparisons or you can compare event names.
5. Click on *Create* to finish

For example, let's say we want to raise an alarm for all those installations of our application with the number of configured users greater than 10. We want to take into account only Windows installations. First we need to create installation filter to cover only Windows machines as it was explained in section "How to Create Installation Filter". Create the alarm definition and add this filter. Then choose numeric comparison and set the *number_of_users* property to be greater than 10. Make the alarm definition *active*. Right now, new alarm will be generated for each event triggered by the Windows installation with *number_of_users* property greater than 10.

If we want to generate alarms also for all Windows installations with our special property status including string "activated". This property may be longer and have other values as well that is why we will use regular expressions. Create new alarm definition and add previously generated installation filter. Choose string comparison for property *status* and use the following regular expression: `".*activated.*"`. Right now, new alarm will be generated for all Windows installations with *status* property including "activated" string.

The last example covers the case when we want to generate an alarm for all Windows installations with Commercial support when they are checking for updates. First create proper installation filter with operating system property equals "Windows" and your special property *commercial_support* equals 1. Create alarm definition, use previously generated installation filter and choose events name comparison with "get_updates" value.

2.7 New Client

You can import a list of clients using a CSV file or add clients individually.

Importing clients from a CSV file:

1. Prepare a CSV file with the list of clients you want to import in the following format: *client_name*, *binding_property_value*, *email*, *description*
2. Go to the *Clients* section
3. Click on *Import Clients* in left menu
4. Select the *Binding Property* - it is used to bind clients and installations. For example, it may be *license_id*, which should be available as an *Installation Property* for all your installations.
5. Select the CSV file.
6. Click the *OK* button

Creating a new client:

1. Go to the *Clients* section
2. Choose *New Client* in left menu
3. Fill in all client data
4. Select *Binding Property*
5. Add any binding rules (will be checked with the values of the binding property).
6. Click on *Create* to finish

2.8 New Installation Filter

1. Go to the *Installations* section
2. Choose *Manage Filters* in left menu
3. Click on *New* button
4. Insert a name for the filter
5. Using *add filter* selector, add the properties that will be used in rules
6. For each of them choose one or more property values
7. Click on *Create* to finish

2.9 Export a File With Installation Data

1. Go to the *Installations* section
2. If you only want to export data for certain types of installations, click on *Advanced Search*. Then, specify which filters, tags, product versions or other variables that you want to filter the installations list by and click *Apply*.
3. Click on *.CSV* or *.XSL* to start exporting.
4. A new download link will appear on left menu in showing that the file is in a loading state.
5. When the checkbox appears, the file is ready to download. Click to download the file.

If desired installations are stored inside a filter, they can be also obtained through the links *CSV* and *XSL* in *Installations* → *Manage Filters*.

Chapter 3

How to view data on ...

3.1 Number Of Installations

Some basic information can be taken from the front page. We can get the total number, and the number of active and registered installations for each product as well as a small chart with the number of active installations over the time for each product version.

For more detailed information, we recommend that you use the *Reports* section. You will get total values and historical views of the number of installations by current product versions and different property values there. The main page in the *Reports* section shows a general view of all reports. Using left menu, you can select a specific report to view a larger version of the chart.

Installation Filters can be applied to the reports to view collected data on specific groups of users.

IMPORTANT




Inactive and testing installations are not included in the reports by default.

3.2 Installation Status and Monitoring Properties

If you are looking for a specific client, type the client's name into the search box in the *Installations* section to locate the client. Otherwise, you can use the Advanced Search functionality to search for users that meet certain criteria. Once you have identified the installation you want to view, click on the *Show* button to view the installation state and its properties, such as the operating system, version installed and the time of the last contact with that installation. Depending on the functionality present in your Network Service installation, you may also be able to view data such as disk space, swap, and other system properties. To view a chart of how the installation's environment has changed over time, click on *Monitoring Data* in the left menu.

3.3 Delivery of Specific Update or Message Notification

To view delivery information for a specific update or message, click on the *Report* button  next to the update or message listing on the *Start Page*, *Messages* or *Updates* sections.

The default report shows the number of active installations reached over time and the number of: *Unique Installations Notified*, *Delivered Notifications*, *Installations to Notify* and the *Percentage of Installations Notified*. It is possible to run additional reports based on property values.

Chapter 4

How to configure ...

4.1 My Account Properties

1. Go to *Administration*
2. Click on *Edit*
3. Change the parameters of your account
4. Save the changes

4.2 Installation Filters/Groups

1. Go to *Installations*
2. Choose *Manage Filters* in left menu

4.3 Alarm Definitions

1. Go to *Alarms*
2. Choose *Alarm Definitions* in left menu
3. The list of current alarm definitions will be shown

4.4 Testing or Registered Installations

1. Go to *Installations*
2. Choose *Manage Filters* → *Special Filters*
3. Click on *Edit* or *Activate* button in the *Testing Installations* or *Registered Installations* filter
4. Modify the rule as desired

Chapter 5

Section Descriptions

5.1 General View

The Network Service portal has common design for all sections. You can navigate through different sections using top main menu. Clicking one of the main menu links, you will get customized left menu including operations links specific for current section. There are three links in the top right corner: *Main account*, which gives you access to your *Settings*, a *Log out* link, which lets you safely quit the Network Service Portal, and *Help*, which links to this User Guide.

If you have more than one *Product* configured, you will see the *Current Product* selector on the top right side just below the main menu. Select one of your products to filter all messages, reports and other data by, or the *All* entry, which will display data for all products..

5.2 General Controls

Here is a list of buttons used throughout the Network Service and their function:

Controls used in any elements list














-  allows you to edit an item.
-  goes to detailed view of the element.
-  removes the element.
-  creates or activates an element.
-  goes to the report page for the element.
-  shows a calendar to allow you to select a date

Chart Controls

-  let you move to the previous or next values in time.
-  allows you to choose the time interval to be shown.
-  changes the chart type.
-  switches to total number / accumulated values view.
-  switches to changes / new entries view.
-  increases / decreases resolution

Others

-  extend an element to show more details, transform a single option selector to a multi-option selector

5.3 Start Page

The *Start Page* gives you general information about your products, notifications and updates sent and number of installations:

- *Current version* and *Release date*
- Number of installations: total, active and registered
- Recent activity: latest updates and messages related with the product
- A small chart that shows the number of active installations over time by version



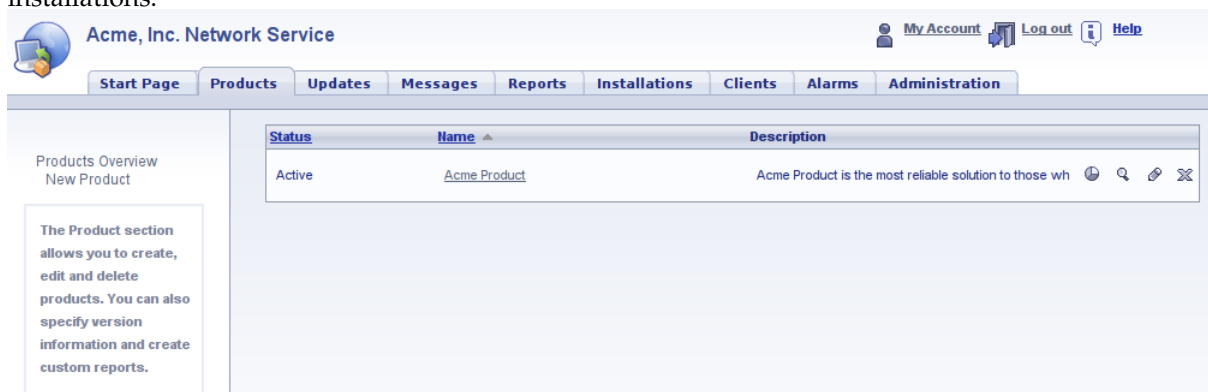
5.4 Products

In this section you can :

- *Manage Products*: Add new products, edit them, and remove them.
- *Manage Versions*: In subsection *Show Product/Product Versions*.
- *Setup Default Notification*: In subsection *Edit Product*.

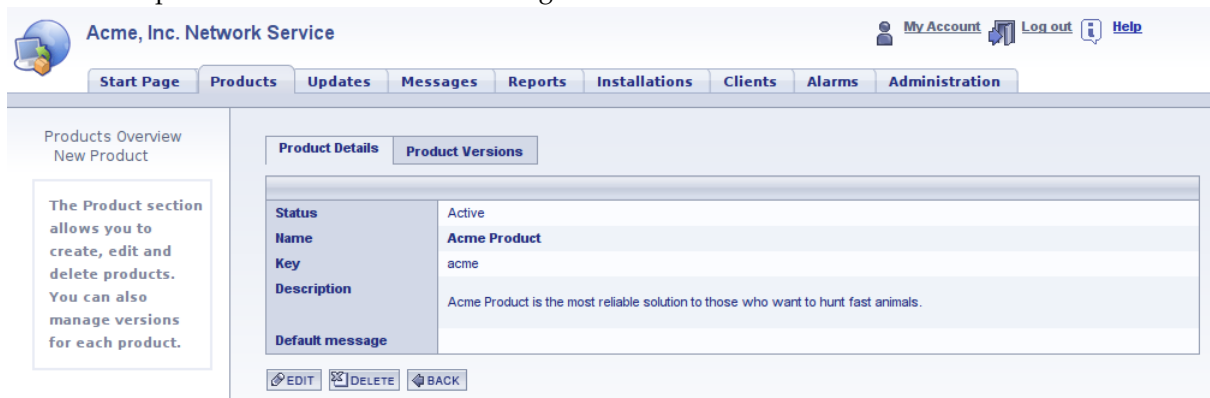
5.4.1 Main view

The main view shows the list of products. Click on the icons on the right side of each product to view additional product details, edit their properties, remove the product or view a report on the number of installations.



5.4.2 Show Product And Manage Versions

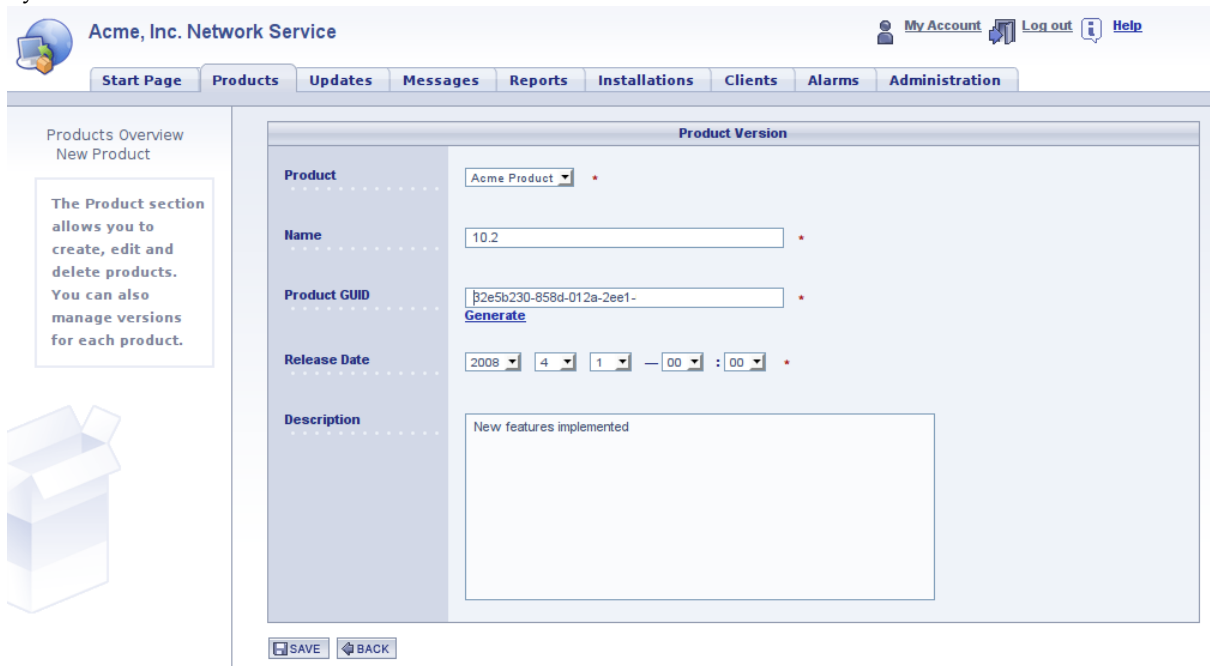
Click on the product name or *Show* button to get to the screen below.



This screen shows a more detailed view of the product and lets you edit its versions using the *Product Versions* tab. Inside that subsection, you can view a list of the different product versions and you can add, edit and remove versions.



All versions have a GUID that is automatically generated by the application. That identifier is used by the installations to contact Network Service.



5.4.3 New Product

Access this screen by clicking on the New Product link on left menu.

Acme, Inc. Network Service

My Account Log out Help

Start Page Products Updates Messages Reports Installations Clients Alarms Administration

Products Overview
New Product

The Product section allows you to create, edit and delete products. You can also manage versions for each product.

Status Draft *

Name New Incredible Product *

Tip The name of the product must be unique.

Key nip *

Default message None

Tip Default message is sent to the end users using simple API. It is very usefull when the client on the end user side is able to show only one notification at the time: update or message.

Description This is a new incredible product

CREATE BACK

At a minimum, you must setup a name and a unique key to generate the new product.

The *default message* will be sent to all users upon installation of your software and/or activation of the Network Service. The same message will be shown for all versions of the product. If you want the *default message* to only be delivered to certain types of users, you can specify the user group by applying one or more filters in the destinations on the update or message destination page.

The status must be set to *Active* in order for messages to be sent to new product installations. The "archive" function can be used to de-activate a message while preventing the loss of data.

5.4.4 Edit Product

From the *Product* page, click *Edit* button to make changes to a product. Then, you can modify the *Default Message*, the status and other product-related properties.

5.5 Messages

This section allows you to manage the Messages that will be delivered to end users.

5.5.1 Main view

The screenshot shows the main view of the Acme, Inc. Network Service. The top navigation bar includes links for My Account, Log out, and Help. Below the navigation bar are tabs for Start Page, Products, Updates, Messages (selected), Reports, Installations, Clients, Alarms, and Administration. The main content area is divided into two sections. On the left, there is a sidebar with 'Messages Overview' and 'New Message' links, and a text box explaining that the Message section allows users to create, edit, and delete messages, and view reports for specific messages. The main area on the right features a table of messages and a timeline chart below it.

Status	Title	Start Date	Finish Date	Default?	
Active	Acme Product 10.2 Coming Soon!	16 Mar	1 Apr	make it default	
Active	New beta release available	15 Apr	26 Jun	make it default	
Active	Version 11.0 coming soon	4 Jun	31 Jul	default	

Below the table is a timeline chart showing the messages plotted against time. The chart has a horizontal axis for years (2007, 2008, 2009, 2010) and a vertical axis for months (br, May, Jun). Three messages are plotted: 'Acme Product 10.2 Coming Soon!' (Mar 16, Apr 1), 'New beta release available' (Apr 15, Jun 26), and 'Version 11.0 coming soon' (Jun 4, Jul 31).

Clicking on the icons to the right of each message to view message details, edit its properties, remove the message or show related reports.

At the bottom, a timeline chart displays recent messages. To view past/future messages, use the mouse to scroll to the left and right.

5.5.2 Show Message

To view details of a message, click on its title or the *Show* button.

The screenshot shows the 'Show Message' view of the Acme, Inc. Network Service. The top navigation bar and tabs are the same as in the main view. The main content area is divided into two sections. On the left, there is a sidebar with 'Messages Overview', 'New Message', 'Reports', and 'Default' links, and a text box explaining that the Message section allows users to create, edit, and delete messages, and view reports for specific messages. The main area on the right features a 'Notification Details' section with a table of message properties and a row of action buttons at the bottom.

Notification Details	
Title	Version 11.0 coming soon
Status	Active
Language	
Critical	False
Start Date	2008-06-04
Finish date	2008-07-31
Webpage Url	http://acme.bitrock.com/news
Description	It will be available in a few days
Created	2008-06-04 03:29 PM
Modified	2008-08-01 01:03 AM

At the bottom of the details section are four buttons: REPORT, EDIT, DELETE, and BACK.

5.5.3 New Message

To create a new message, click on the *New Message* link on left menu.

Follow these steps to generate a new message:

- Insert title, message contents and a webpage link to additional details

Acme, Inc. Network Service

My Account Log out Help

Start Page Products Updates Messages Reports Installations Clients Alarms Administration

Messages Overview
New Message

The Message section allows you to create, edit and delete messages. You can also view reports for specific messages.

Adding New Message

General Notification Information

Title: message

Description: this is the text of the new message

Webpage Uri: http://www.acme.bftrock.com

Webpage Uri Description: more info

BACK NEXT

- Choose which product versions and installation groups should receive this message
 - Installations groups are defined in *Installations* → *Manage Filters*

Acme, Inc. Network Service

My Account Log out Help

Start Page Products Updates Messages Reports Installations Clients Alarms Administration

Messages Overview
New Message

The Message section allows you to create, edit and delete messages. You can also view reports for specific messages.

Adding New Message

Select Destination

Product Versions

- Acme Product
 - 10.2
 - 10.5
 - 11.0-beta

Filter Installations

- Linux Users

Add filter:

Join Method: all filters (AND)

Tip You can manage the filters in the administration section.

SAVE BACK

- Insert status, duration and critical properties
 - *Critical* will mark the message in client application as important if it is supported

The screenshot shows the 'Acme, Inc. Network Service' web interface. At the top, there is a navigation bar with links for 'My Account', 'Log out', and 'Help'. Below this is a secondary navigation bar with tabs for 'Start Page', 'Products', 'Updates', 'Messages', 'Reports', 'Installations', 'Clients', 'Alarms', and 'Administration'. The 'Messages' tab is selected. On the left side, there is a sidebar with 'Messages Overview' and 'New Message' links. A text box explains: 'The Message section allows you to create, edit and delete messages. You can also view reports for specific messages.' The main content area is titled 'Adding New Message' and contains a form with the following fields: 'Status' (dropdown menu set to 'Active'), 'Critical' (checkbox set to 'False'), 'Start Date' (text input with '2008-08-14' and a calendar icon), and 'Finish Date' (text input with '2008-08-29' and a calendar icon). A tip below the dates reads 'Tip Date format: YYYY-MM-DD'. At the bottom of the form are 'PREVIOUS' and 'CREATE' buttons.

- After clicking on *Create* the message will be created

NOTE



Notification will be sent only if the status is set to active, the current time is between the specified notification dates, the installation has one of the designated versions and meets the criteria set by the installation filters.

5.5.4 Edit Message


Click *Edit* to get to the edit message screen.

It has two sections:

- Notification: In the notification section, you specify the message title, contents, duration, status and other information.

Messages Overview
[New Message](#)
[Reports](#)
[Default](#)

The Message section allows you to create, edit and delete messages. You can also view reports for specific messages.



Notification
Destination

Title

.....

Status

.....

Critical

.....

Start Date

.....

Finish Date

.....

Webpage Url

.....

Webpage Url Description

.....

Description

.....

Version 11.0 coming soon

Active

False

2008-06-04

2008-07-31

Tip Date format: YYYY-MM-DD

http://acme.bitrock.com/news

more info


It will be available in a few days

SAVE
BACK

- Destination: The destination section allows you to specify which product versions and installation groups should receive the message.

Messages Overview
[New Message](#)
[Reports](#)
[Default](#)

The Message section allows you to create, edit and delete messages. You can also view reports for specific messages.



Notification
Destination

Product Versions

.....

Filter Installations

.....

Acme Product

10.2

10.5

11.0-beta

Linux Users

Add filter:

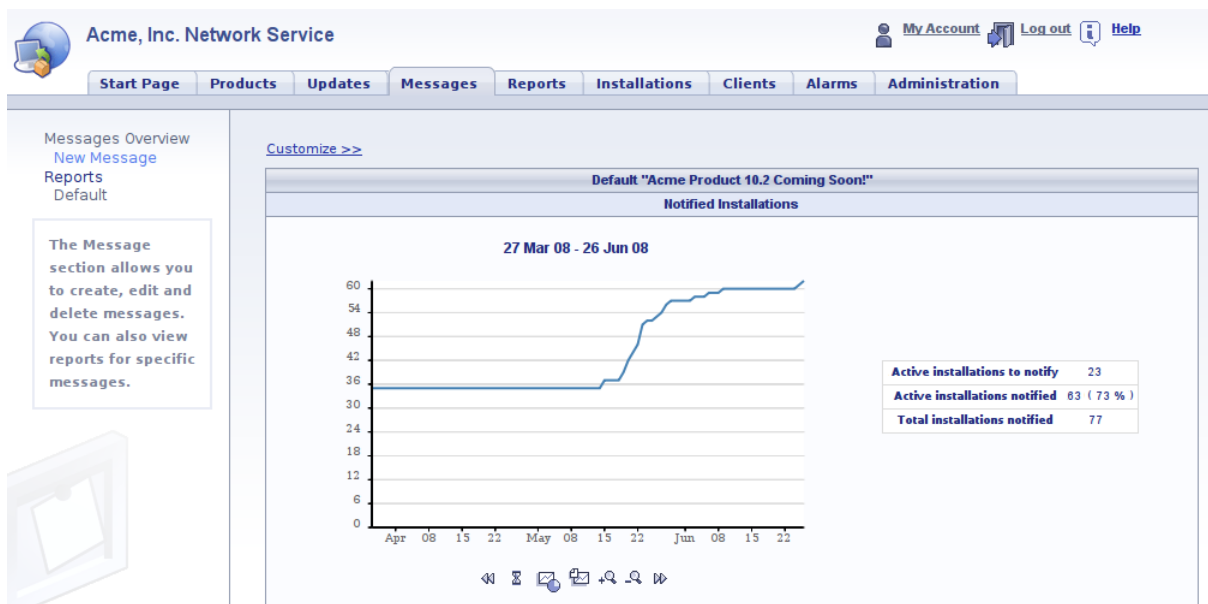
Join Method:

Tip You can manage the filters in the administration section.

SAVE
BACK

5.5.5 Message Reports

Reports on the uptake of messages can be viewed by clicking on the *Report* button or using the links in left menu while viewing a message.



Message reports show you the uptake on a specific message. By default, the chart shows the number of installations that have received the message over time. The table displays the number of active installations that have received the message, the percentage of active installations that number represents, the number of current active installations that should receive the message and the total installation count (including inactive).

The reports can be customized based on installation filters, product versions or can take inactive installations into account. To customize the message reports, click on the "Customize >>" link. If additional filters based on property values have been defined, they will be available as links on left menu.

In this subsection you can get information about how is the message being delivered. By default you'll get the evolution over the time about the number of active installations that has received the message and a table which represents: number of active installations notified, percentage of active installations, current active installations that could receive the message and total count (including inactive).

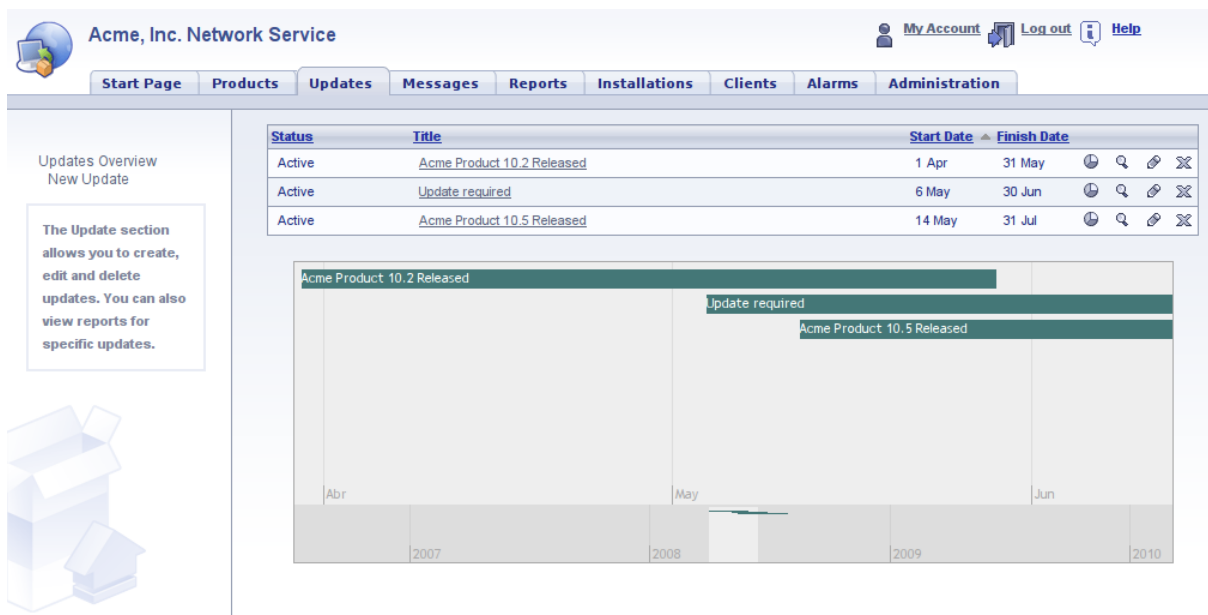
It is possible to obtain reports based on installation filters or take into account inactive installations on the historical chart through "Customize >>" link in a similar way as it is done in Reports section. If additional filters based on property values have been defined, they will be available as links on left menu.

5.6 Updates

The Updates section allows you to define and manage the Updates that will be delivered to the final users.

5.6.1 Main view

The main page shows a list of current product updates.



Click on the icons to the right of each update to view update details, edit its properties, delete the update or show related reports.

At the bottom, a timeline chart displays recent messages. To view past/future messages, use the mouse to scroll to the left and right.

5.6.2 Show Update

To view update details, click on the update’s title or on the *Show* button.

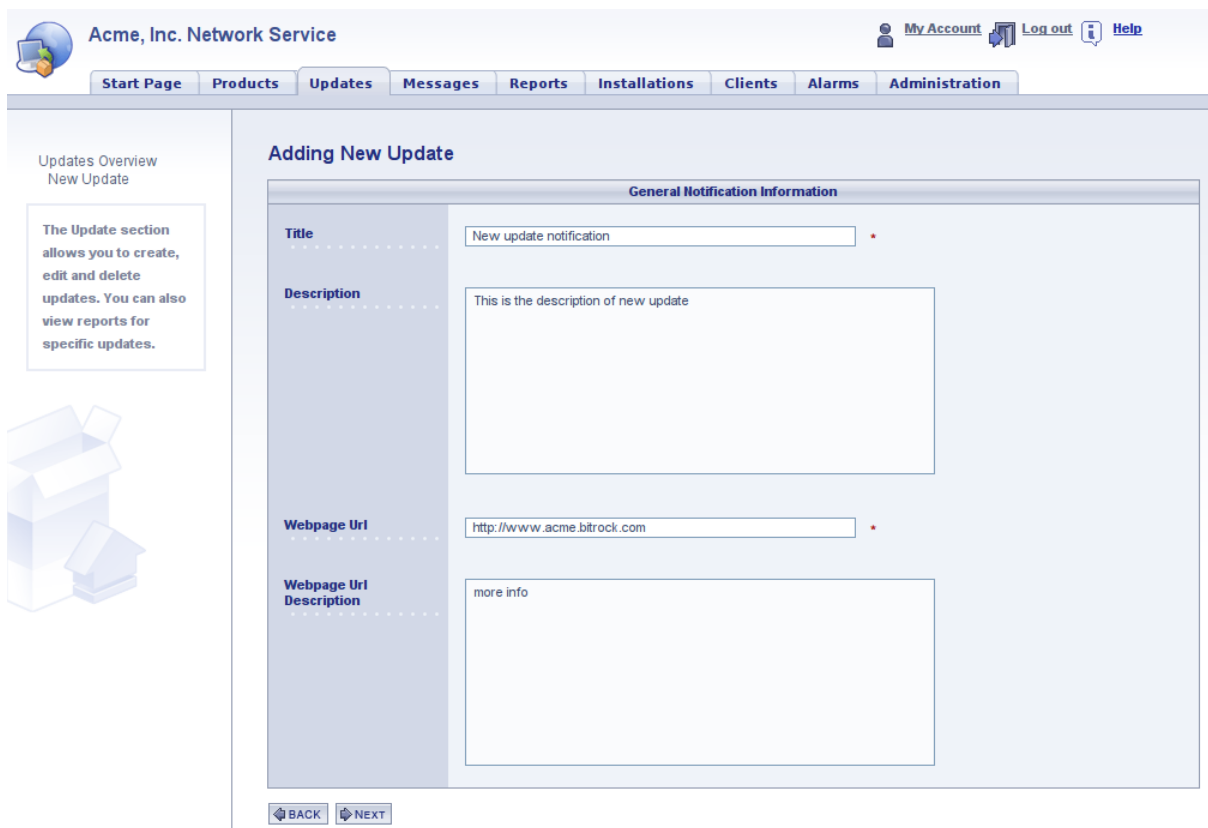


5.6.3 New Update

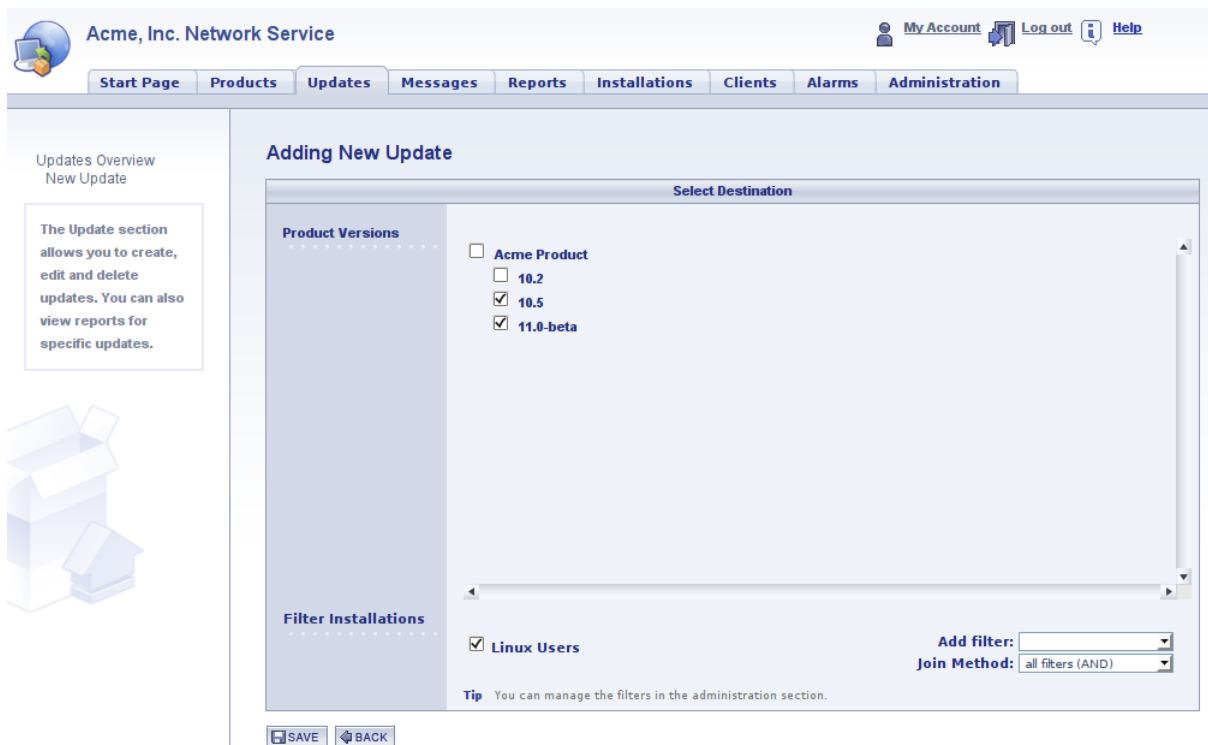
To create a new update, click on the "New Update" link on left menu.

Follow these steps to generate a new update:

- Insert notification details: provide the update title, content and a webpage link to additional information.



- Choose which product versions and installation groups should receive the update
 - Installations groups are defined in *Installations* → *Manage Filters*



- Insert the update status, duration and specify whether or not the update should be displayed as "important"
 - *Important* will mark the update in client application as important if it is supported

Acme, Inc. Network Service

My Account Log out Help

Start Page Products Updates Messages Reports Installations Clients Alarms Administration

Updates Overview
New Update

The Update section allows you to create, edit and delete updates. You can also view reports for specific updates.

Adding New Update

Status: Active

Critical: False

Start Date: 2008-08-20

Finish Date: 2008-08-30

Tip: Date format: YYYY-MM-DD

PREVIOUS NEXT

- Enter the update information: the URL to the file, the type of file, MD5, instructions to perform the update and other information

Acme, Inc. Network Service

My Account Log out Help

Start Page Products Updates Messages Reports Installations Clients Alarms Administration

Updates Overview
New Update

The Update section allows you to create, edit and delete updates. You can also view reports for specific updates.

Adding New Update

Update Information

Type: bin

Url: http://www.acme.bitrock.com/acme_product_12.bin

Size: 1023

MD5: 26fd982725013b0b4981e5b9873b33f5

Operating System: Solaris

Command Line Switches:

Instruction:

PREVIOUS CREATE

- After clicking on *create*, the update will be generated

NOTE



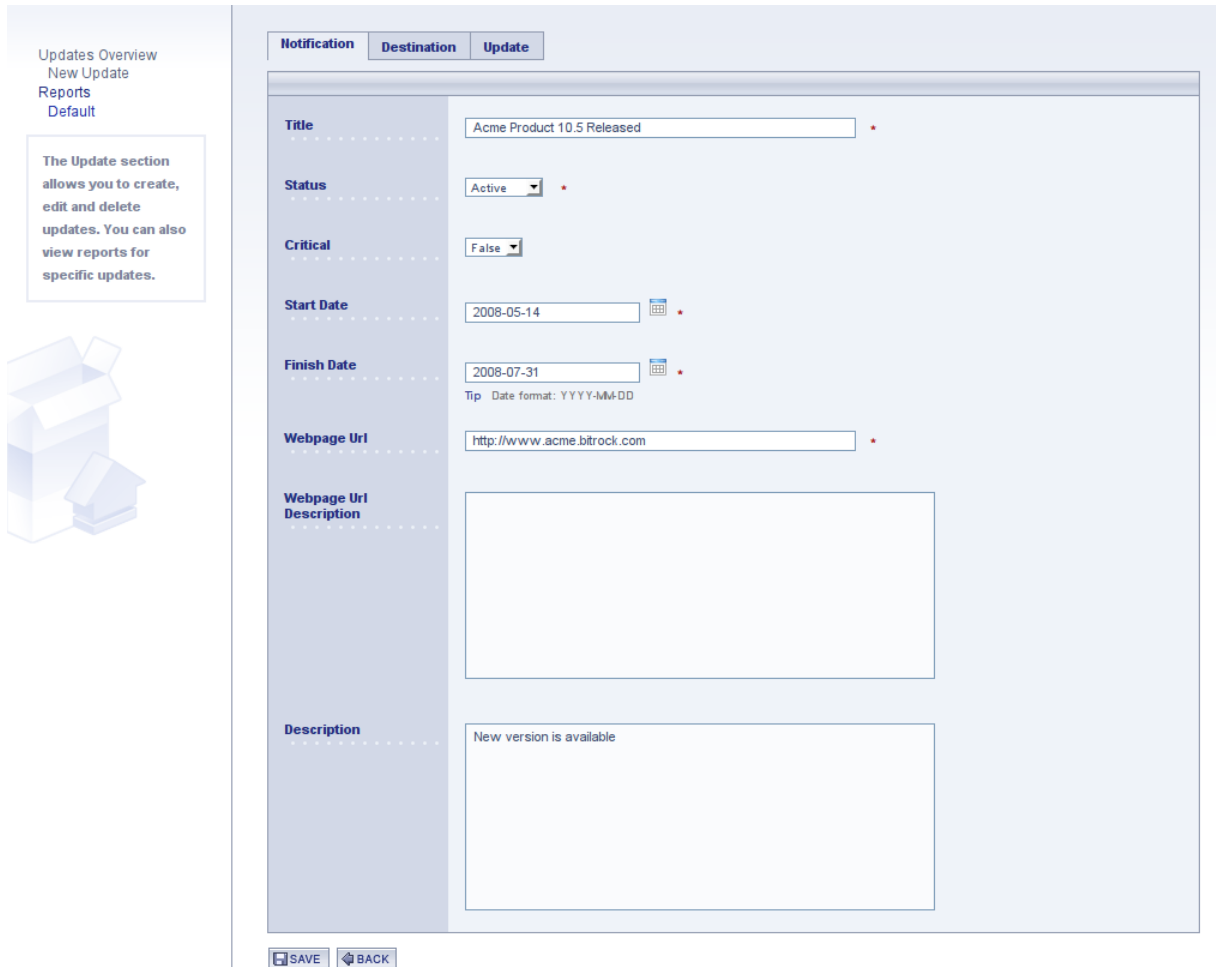
Notification will be sent only if the status is set to active, the current time is between date limits, the installation has one of the specified versions and the installation filter criteria are passed.

5.6.4 Edit Update

Click on Edit to edit an update.

It has three sections:

- Notification: The notification tab allows you to specify properties including the update. title, contents, duration, status and other information

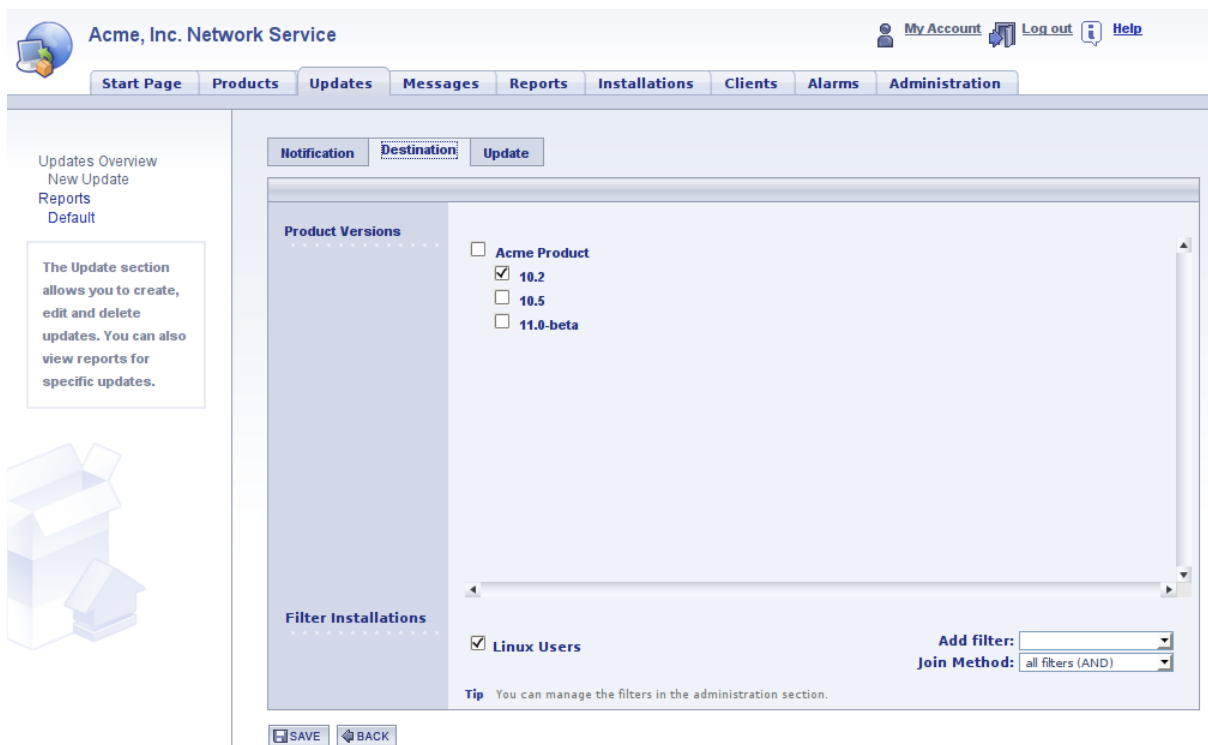


The screenshot displays the 'Edit Update' web interface. On the left, there is a navigation menu with links for 'Updates Overview', 'New Update', 'Reports', and 'Default'. Below the menu is a box containing the text: 'The Update section allows you to create, edit and delete updates. You can also view reports for specific updates.' and an icon of an open cardboard box. The main content area has three tabs: 'Notification' (selected), 'Destination', and 'Update'. The 'Notification' tab contains the following fields:

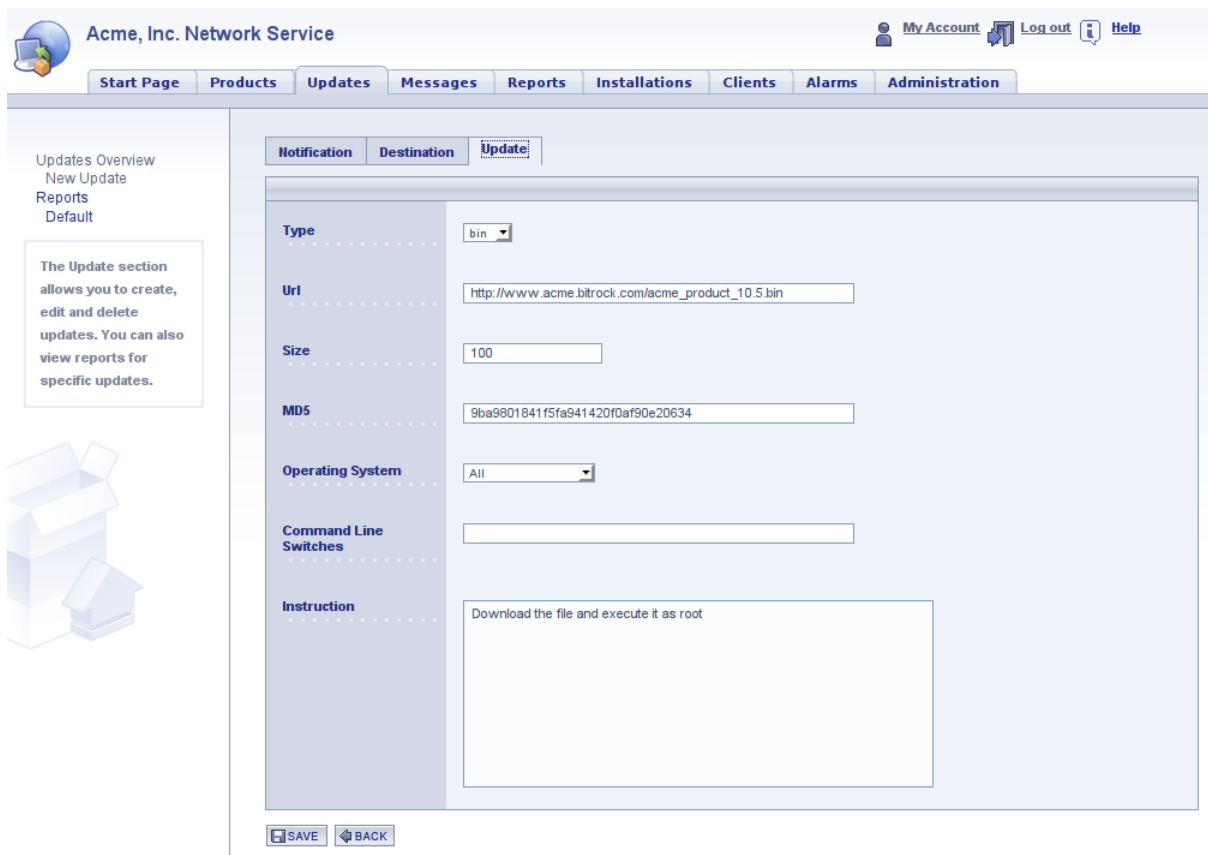
- Title:** Acme Product 10.5 Released
- Status:** Active (dropdown menu)
- Critical:** False (dropdown menu)
- Start Date:** 2008-05-14 (calendar icon)
- Finish Date:** 2008-07-31 (calendar icon)
- Webpage Url:** http://www.acme.bitrock.com
- Webpage Url Description:** (empty text area)
- Description:** New version is available

At the bottom of the form are two buttons: 'SAVE' and 'BACK'.

- Destination: The destination tab allows you to specify which installations receive the update.



- Update: The update tab allows you to specify which product versions should receive the update, as well as update information such as the URL to the file, the file type, the MD5 and other information.



5.6.5 Update Reports

To access update reports, click on the *report* button or use the links on left menu while viewing the update.



This section provides information on update delivery. By default, the chart shows the number of installations that have received the update over time. The table displays the number of active installations that have received the update, the percentage of active installations that number represents, the number of current active installations that should receive the message, and total installation count (including inactive).

The reports can be customized based on installation filters, product versions or can take inactive installations into account. To customize the message reports, click on the "Customize >>" link. If additional filters based on property values have been defined, they will be available as links on left menu.

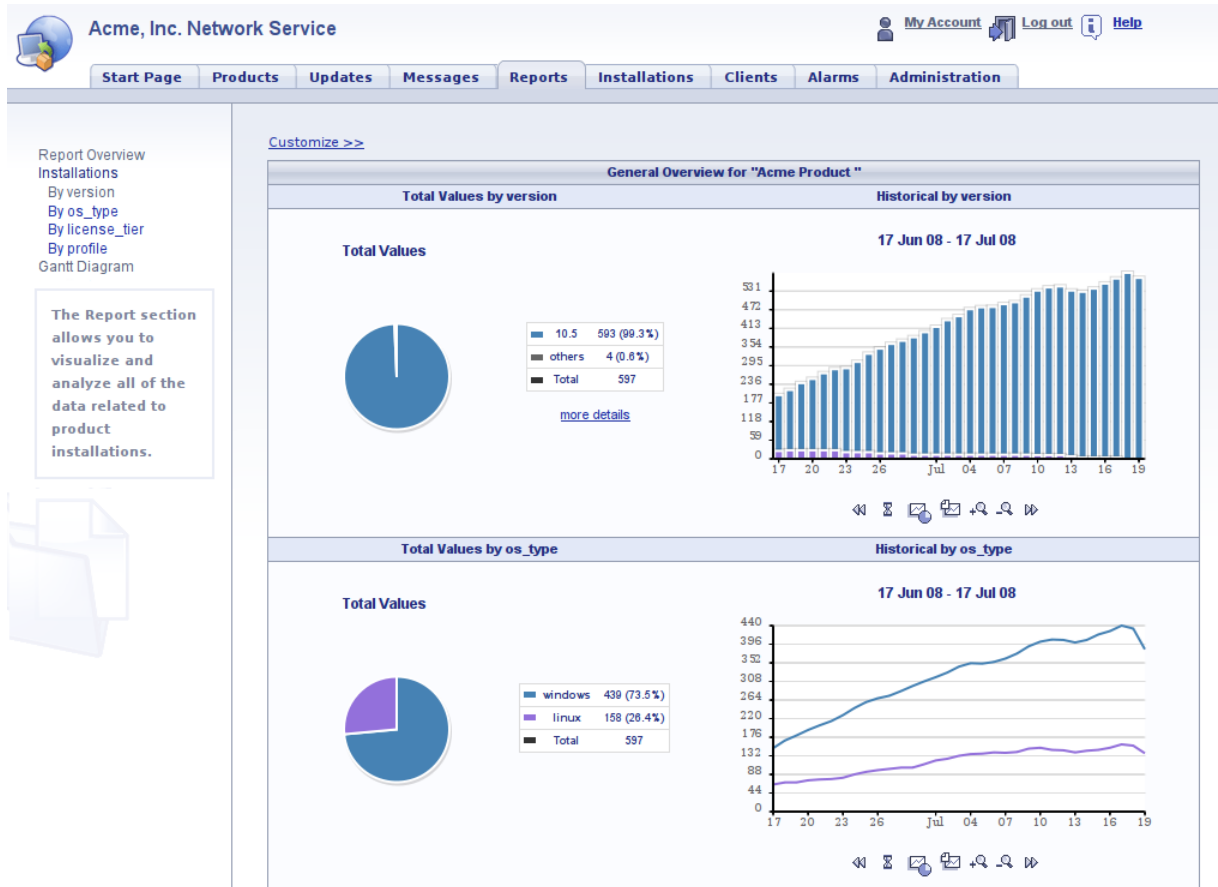
5.7 Reports

The reports section allows you to view detailed information about the end users of your product.

5.7.1 Main View

The main view shows the total number of installations by version and other collected property values over time.

The main page consists of a small pie chart, a table with total values and a chart that shows how the property has changed over time.



You can also use the installation filters to restrict the reports to a specific group. To do so, click on "Customize >>", choose the required filters and select whether each installation must meet all or just one of the specified filters.

The 'Customize >>' interface allows users to filter installations. It includes sections for 'Include Installations' and 'Filter installations'.

Include Installations:

- Inactive
- Tagged as test

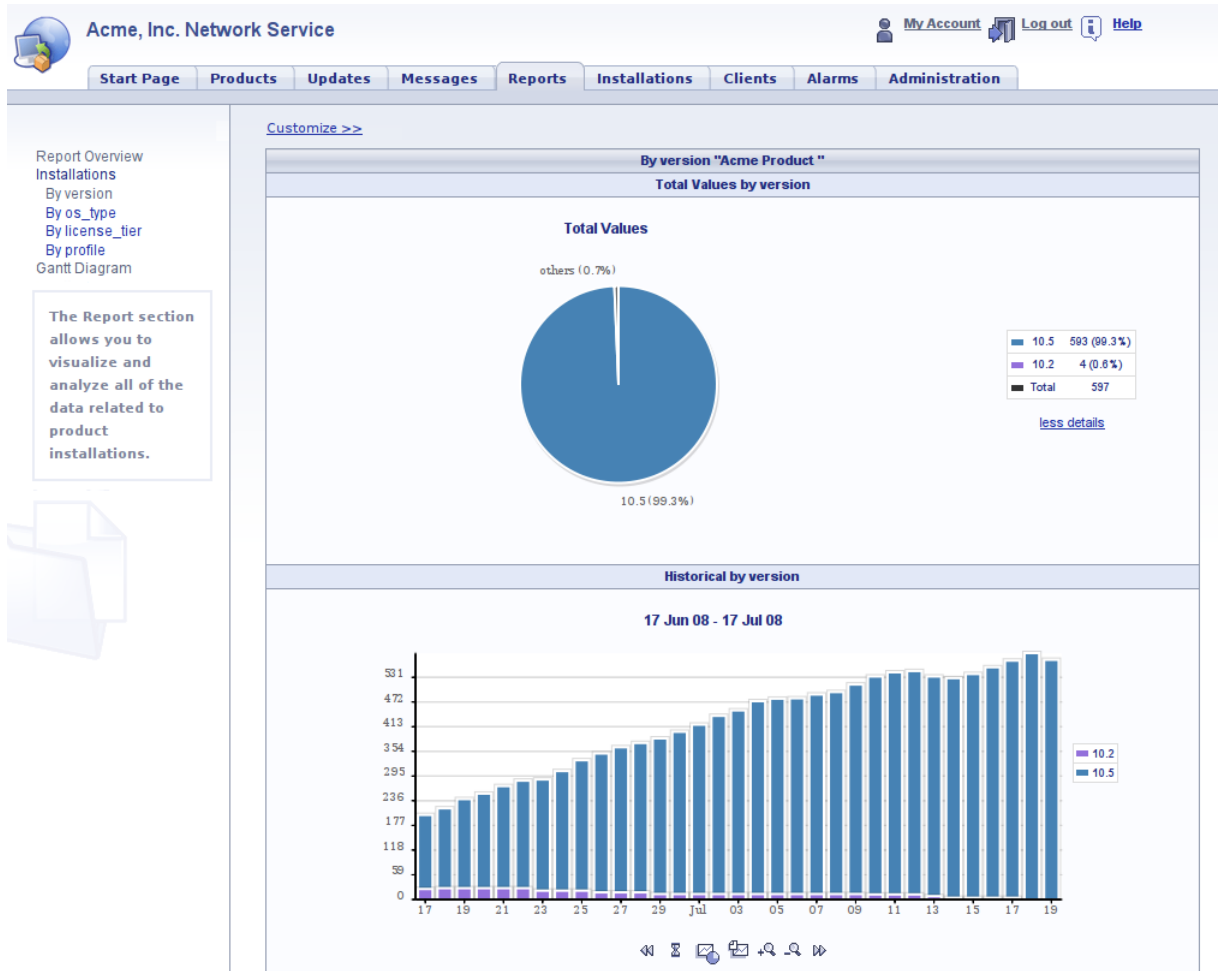
Filter installations:

- Linux Users
- Windows Active Users

Additional options include an 'Add filter:' dropdown and a 'Join Method:' dropdown set to 'one of the filters (OR)'. A tip states: "By default those installations are not taken into account in the calculations" and "You can manage the filters in Installations section."

5.7.2 Detailed Report

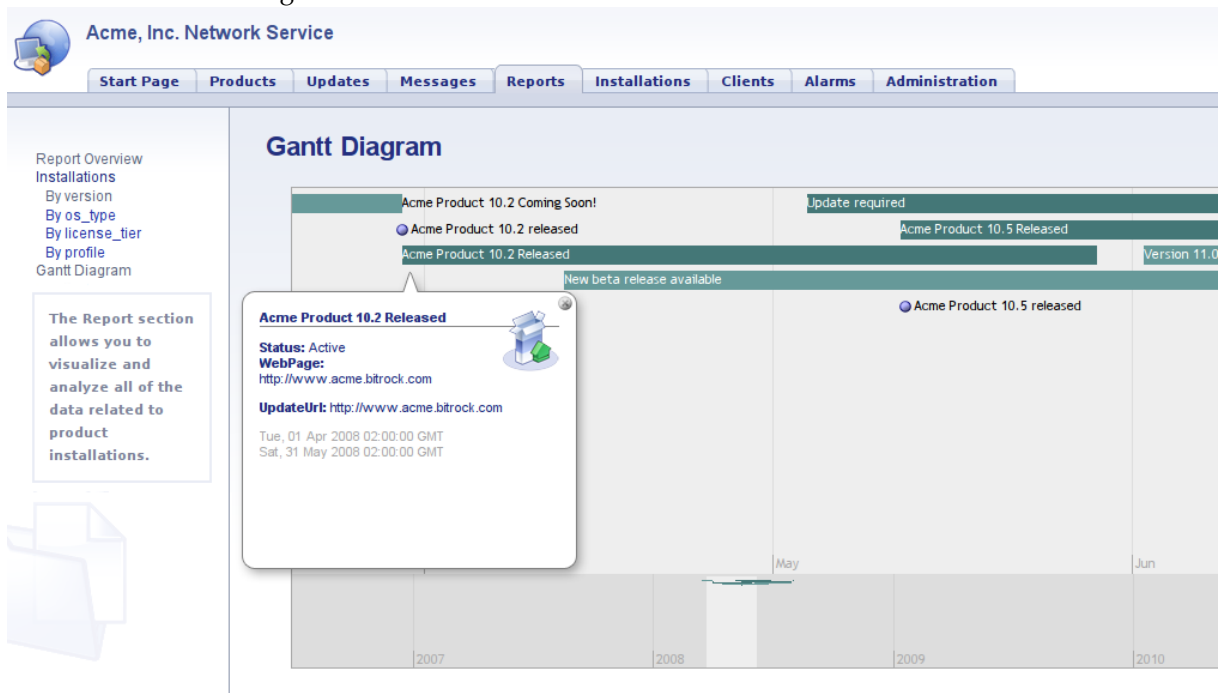
To view detailed reports, use the links on the menu.



The same modifications, as in *Main* view, could be made through the "Customize >>" link.

5.7.3 Gantt

Click on the "Gantt Diagram" link on the left menu.

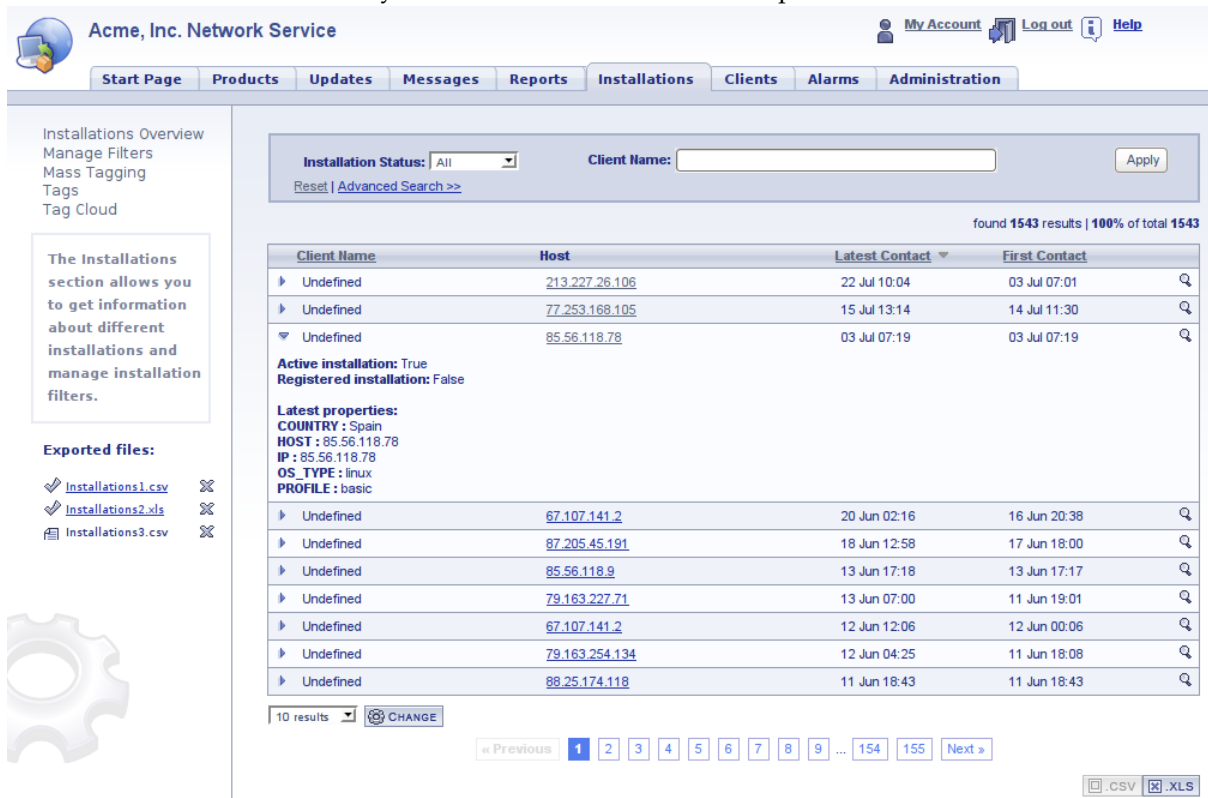


The Gantt chart shows information about messages, updates and the publish dates of product ver-

sions over the time. Use your mouse to scroll to the left and right to view past and future information.

5.8 Installations

The installations section allows you to view information about a specific installation.



Acme, Inc. Network Service

My Account Log out Help

Start Page Products Updates Messages Reports **Installations** Clients Alarms Administration

Installations Overview
Manage Filters
Mass Tagging
Tags
Tag Cloud

The Installations section allows you to get information about different installations and manage installation filters.

Exported files:
 Installations1.csv ✕
 Installations2.xls ✕
 Installations3.csv ✕

Installation Status: All Client Name: Apply
[Reset](#) | [Advanced Search >>](#)

found 1543 results | 100% of total 1543

Client Name	Host	Latest Contact	First Contact
Undefined	213.227.26.106	22 Jul 10:04	03 Jul 07:01
Undefined	77.253.168.105	15 Jul 13:14	14 Jul 11:30
Undefined	85.56.118.78	03 Jul 07:19	03 Jul 07:19
Active installation: True Registered installation: False Latest properties: COUNTRY : Spain HOST : 85.56.118.78 IP : 85.56.118.78 OS_TYPE : linux PROFILE : basic			
Undefined	67.107.141.2	20 Jun 02:16	16 Jun 20:38
Undefined	87.205.45.191	18 Jun 12:58	17 Jun 18:00
Undefined	85.56.118.9	13 Jun 17:18	13 Jun 17:17
Undefined	79.163.227.71	13 Jun 07:00	11 Jun 19:01
Undefined	67.107.141.2	12 Jun 12:06	12 Jun 00:06
Undefined	79.163.254.134	12 Jun 04:25	11 Jun 18:08
Undefined	88.25.174.118	11 Jun 18:43	11 Jun 18:43

10 results CHANGE

« Previous 1 2 3 4 5 6 7 8 9 ... 154 155 Next »

.CSV .XLS

The main view contains a full list of installations. You can search for specific installations using the filters at the top of the page:

- Status: Allows you to specify whether you want to view registered or unregistered installations.
- Include Installations: Check this box if you want to view inactive or testing installations (by default inactive and testing installations are not included in the list).
- Client Name and Installation GUID: wildcards * can be used in searches.
- Tags: select installations by selecting one or more of the tags.
- Product Version
- Installation Tier: it could be basic, pro and monitoring
- Installation Filters: usage of filters defined in *Installations* → *Manage Filters*

Acme, Inc. Network Service

My Account | Log out | Help

Start Page | Products | Updates | Messages | Reports | **Installations** | Clients | Alarms | Administration

Installations Overview
Manage Filters
Mass Tagging
Tags
Tag Cloud

The Installations section allows you to get information about different installations and manage installation filters.

Installation Status: Registered Client Name: Apply

Reset | Advanced Search >>

Include Installations Inactive Testing
Tip: By default inactive and testing installations are not taken into account in the calculations

Installation Guide:

Tags:

Product Version: All

Installation Type / Tier: All

Apply Installation Filters Linux Users 10+ Users Add filter: Join Method: all filters (AND)

Tip: You can modify filters in manage filters section.

found 2440 results | 25% of total 9388

To view more detailed information regarding the specific installation, click on the host link or show button. Basic details can be viewed by using the triangle button next to each entry in the list.

5.8.1 Export Installations to CSV or XLS

The list of installations can be exported in *CSV* or *XLS* formats. In order to export the data, click either the *CSV* or *XLS* button below the list. After you click on the appropriate button, a new entry will be shown on left menu, inside the "Exported Files" list. The file will first appear in a loading state represented by a small, moving document icon. Once the file is generated, the icon will change to a check mark. Click on the link that appears to download the file. The Exported Files list is shown in throughout the whole application, so you can perform other operations while exporting data.

5.8.2 Installation Dashboard

The installation dashboard is displayed whenever you are viewing the details on a specific installation.

Acme, Inc. Network Service

My Account | Log out | Help

Start Page | Products | Updates | Messages | Reports | **Installations** | Clients | Alarms | Administration

Installations Overview
Manage Filters
Mass Tagging
Tags
Tag Cloud
Current Installation
Dashboard
Monitoring Data

The Installations section allows you to get information about different installations and manage installation filters.

Details | Events | Alarms | Latest Properties

Unknown Client ID: 2221c126-5aba-4690-7aaf-██████████

Acme Product 10.5 87.205

2 Users 2919 New Events

0 Documents 19 New Alarms

more

MONITORING

Memory: Free (3.9%) / Used (96.1%)

Swap: Free (3.7%) / Used (96.3%)

Disk /media/sda11: Free (1.8%) / Used (98.2%)

Disk /: Free (12.6%) / Used (12.6%)

Disk /media/sda9: Free (5.9%) / Used (5.9%)

Disk /boot: Free (27.1%) / Used (72.9%)

This view provides the following information:

- Client's name if it has been binded.

- Installation Tier:  Monitoring  Professional  Basic

- Operating System:   

- Product and Version
- Host and Country
- Any set of configured property values with small sparkline charts
- Number of unacknowledged Alarms and Events related with that installation

The information on disk space, memory and swap usage will be shown only for installation with the *monitoring* profile. In that case, you will also see a set of pie charts.

More system information can be seen by clicking *more* link.



Acme Product 10.5
87.205. 

[less](#)

```

Active: True
Registered: False
CPU count : 1
Country : Poland
Cpu model : Intel(R) Pentium(R) M processor 1.70
Disk / size : 19141396 KB
Disk /boot size : 52659 KB
Disk /home size : 14877060 KB
Disk /home/media size : 14877060 KB
Disk /media/sda10 size : 505604 KB
Disk /media/sda11 size : 256666 KB
Disk /media/sda9 size : 505604 KB
Disk /opt size : 24797380 KB
Host : 87.205. 
IP : 87.205. 
Memory size : 1555004 KB
OS Kernel : 2.6.24-16-generic #1 smp thu apr 10
Operating System : linux
System glibc : 2.7
Last contact: 2008-07-14 03:03 PM
  
```

In the *Latest Properties* tab you can view a complete list of all information received from the installation during last contact.

The screenshot shows the 'Acme, Inc. Network Service' web interface. The top navigation bar includes 'Start Page', 'Products', 'Updates', 'Messages', 'Reports', 'Installations', 'Clients', 'Alarms', and 'Administration'. The 'Installations' tab is active. On the left, there is a sidebar with 'Installations Overview' and a list of filters. The main content area shows 'Latest Installation Properties' with the following details:

- CPU count: 1
- Country: Poland
- Cpu mhz: 1700
- Cpu model: Intel(R) Pentium(R) M processor 1.70GHz
- Disk / free: 2420496 KB
- Disk / size: 19141396 KB
- Disk / used: 16720900 KB
- Disk / boot free: 14274 KB
- Disk / boot size: 52659 KB
- Disk / boot used: 38385 KB
- Disk / home free: 268624 KB
- Disk / home size: 14877060 KB
- Disk / home used: 14608436 KB
- Disk / home/media free: 181308 KB
- Disk / home/media size: 14877060 KB
- Disk / home/media used: 14695752 KB
- Disk / media/sda10 free: 226313 KB
- Disk / media/sda10 size: 505604 KB
- Disk / media/sda10 used: 279291 KB
- Disk / media/sda11 free: 4608 KB
- Disk / media/sda11 size: 256666 KB
- Disk / media/sda11 used: 252058 KB
- Disk / media/sda9 free: 29836 KB
- Disk / media/sda9 size: 505604 KB
- Disk / media/sda9 used: 475768 KB
- Disk / opt free: 451880 KB
- Disk / opt size: 24797380 KB
- Disk / opt used: 24345500 KB

5.8.3 Installation Monitoring

From the Installation Dashboard, you can get additional information on monitored metrics if that is enabled in your Network Service installation.

To view them, get it you should click *Monitoring* on left menu or on any of the configured properties shown as sparklines in the *Dashboard*.

The screenshot shows the 'Acme, Inc. Network Service' web interface with the 'Installation Report' for a specific installation. The report is titled 'Installation Report "1116c126-5aba-4690-7aaf-..."'. It displays two charts: 'Memory' and 'Swap'. The 'Memory' chart shows usage from 06:30 / 08 May 08 to 11:29 / 09 May 08. The 'Swap' chart shows usage from 06:00 / 08 May 08 to 06:00 / 09 May 08. Both charts show historical values over time, with grey areas indicating periods without received data. The 'Last received values' table for Memory is as follows:

Last received values		
Memory free	512086.0	09 May 08:30
Memory size	2088388.0	09 May 08:30
Memory used	1554272.0	09 May 08:30

The 'Last received values' table for Swap is as follows:

Last received values		
Swap free	1427188.0	09 May 06:30
Swap total	1461904.0	09 May 06:30
Swap used	34716.0	09 May 06:20

The monitoring screen shows a list of the latest values reported by the agent, as well as charts with the historical values over time. Periods of time without received data are illustrated by grey areas.

5.8.4 Installation Alarms And Events

You can view a list of the latest alarms and events related to a specific installation by clicking on the *Alarms* or *Events* tabs in the *Installation Dashboard*.

Acme, Inc. Network Service

Start Page Products Updates Messages Reports Installations Clients Alarms Administration

Installations Overview
Manage Filters
Mass Tagging
Tags
Tag Cloud
Current Installation
Dashboard
Monitoring Data

The Installations section allows you to get information about different installations and manage installation filters.

Details Events Alarms Latest Properties

Name	IP	Date
<input type="checkbox"/> READ_NOTIFICATIONS	87.205	14 Jul 15:03
<input type="checkbox"/> READ_MESSAGE	87.205	14 Jul 15:03
<input type="checkbox"/> READ_MESSAGE	79.163	09 Jun 16:14
<input type="checkbox"/> READ_MESSAGE	79.163	09 Jun 16:14
<input type="checkbox"/> READ_NOTIFICATIONS	79.163	09 Jun 16:14
<input checked="" type="checkbox"/> READ_NOTIFICATIONS	79.163	09 Jun 16:10
<input checked="" type="checkbox"/> READ_MESSAGE	79.163	09 Jun 16:10
<input checked="" type="checkbox"/> READ_MESSAGE	79.163	09 Jun 16:10
<input checked="" type="checkbox"/> READ_MESSAGE	79.163	06 Jun 09:42
<input checked="" type="checkbox"/> READ_MESSAGE	79.163	06 Jun 09:42

ACKNOWLEDGE ACKNOWLEDGE ALL

10 results

« Previous 1 2 3 4 5 6 7 8 9 ... 289 290 Next »

Acme, Inc. Network Service

Start Page Products Updates Messages Reports Installations Clients Alarms Administration

Installations Overview
Manage Filters
Mass Tagging
Tags
Tag Cloud
Current Installation
Dashboard
Monitoring Data

The Installations section allows you to get information about different installations and manage installation filters.

Details Events Alarms Latest Properties

Name	Counter	Description	Updated	Created
<input type="checkbox"/> MEMORY_FREE > 1000	9	MEMORY_FREE > 1000	14 Jul 15:03	29 May 12:36
<input type="checkbox"/> NUMBER_OF_DOCUMENTS = 0	181		14 Jul 15:03	28 May 16:03
<input type="checkbox"/> NUMBER_OF_DOCUMENTS = 0	45		28 May 16:03	28 May 15:21
<input type="checkbox"/> NUMBER_OF_DOCUMENTS = 0	87		28 May 15:21	28 May 15:16
<input type="checkbox"/> NUMBER_OF_DOCUMENTS = 0	357		28 May 15:15	28 May 14:46
<input type="checkbox"/> NUMBER_OF_DOCUMENTS = 0	25		28 May 14:46	28 May 14:36
<input type="checkbox"/> NUMBER_OF_DOCUMENTS = 0	176		28 May 14:36	28 May 14:30
<input type="checkbox"/> NUMBER_OF_DOCUMENTS = 0	57		28 May 14:30	28 May 14:28
<input type="checkbox"/> NUMBER_OF_DOCUMENTS = 0	87		28 May 14:27	28 May 14:07
<input type="checkbox"/> NUMBER_OF_DOCUMENTS = 0	30		28 May 14:06	28 May 14:04

ACKNOWLEDGE ACKNOWLEDGE ALL

10 results

« Previous 1 2 Next »

To acknowledge an alarm or the event, check the corresponding checkbox and click *Acknowledge*. After alarms have been acknowledged, they will not be taken into account by the counters on the dashboard.

5.8.5 Manage Filters

Click on *Manage Filters* in the left menu in the Installations section to manage your filters.

In this section, you can manage the installation filters, which define different groups of installations based on property values. Filters can be used throughout the application to define which specific user groups should receive messages and updates and to filter the report results.

5.8.5.1 Main View

The main view shows a list of current user defined installation filters.

The screenshot shows the 'Acme, Inc. Network Service' main view. The navigation menu includes 'Start Page', 'Products', 'Updates', 'Messages', 'Reports', 'Installations', 'Clients', 'Alarms', and 'Administration'. The 'Installations' section is active, showing 'User Defined Filters' and 'Special Filters' tabs. The 'User Defined Filters' tab is selected, displaying a table of filters:

Name	Description	Active Installations
10+ Users	Installations with more than 10 users	353
Linux Users		873
Spanish Linux Users with -50 users	Installations from Spain which has Linux as operating system and less than 50 users	13
Windows Registered Users		0

Each row in the table includes icons for viewing details, editing, deleting, and exporting to CSV or XLS. A 'NEW' button is located below the table. On the left, a sidebar contains links for 'Installations Overview', 'Manage Filters', 'Mass Tagging', 'Tags', and 'Tag Cloud', along with a descriptive text box.

Using the icons to the right of each entry, you can view additional details, edit their properties, or to delete a filter. For each entry, the count of active installations which pass the filter is shown. Clicking on it will display the complete list of those installations. .

It is also possible to export a filtered list of active installations directly into a .CSV or .XLS file using the links in the active installations column. After clicking one of the links, a new entry is added to "Exported Files" section on left menu in a loading state. Once the file has been generated, a link to the file will appear in that section and for you to download.

5.8.5.2 Special Filters

Click on the "Special Filters" tag in the main view to arrive at the special filters tab.

The screenshot shows the 'Acme, Inc. Network Service' special filters tab. The navigation menu is the same as in the previous screenshot. The 'Special Filters' tab is selected, displaying a table of predefined filters:

Name	Description	Active Installations
Filtered Test Installations	Installations which pass the defined rule will be tagged with "test" and removed from all statistics and calculations by default	
Registered Installations	Installations which pass the defined rule will be marked as registered	331

Each row includes icons for activating, editing, deleting, and exporting to CSV or XLS. A 'NEW' button is located below the table. The sidebar on the left is identical to the previous screenshot.

On the special filters tab, you will see a list of predefined filters with programmed behaviors. If you want to activate any of them, click on its *activate* icon and insert the desired rule in a similar way to any other filter. After clicking on *Create* the filter will be active and you you'll get information about the number of active installations affected by it and the links to export them similar to the ones from the user defined filters section.

5.8.5.3 New / Edit Installation Filter

To create an installation filter, click on *New*.

To create a new filter, complete the form by defining the rules that you want to apply for the filter. Select one or more properties and one or more possible values for each of property for the rule logic.

To add another property, use the "add filter" selector. A new element will appear below with options for that particular property.

The installation group will include all installations that meet the criteria that you have specified.

NOTE



Be careful when modifying filters because they affect not only reports, but also which installations receives certain messages and updates.

5.8.6 Mass Tagging

The "Mass Tagging" feature allows you to apply multiple tags to a group of installations. To use the feature, click on the *Mass Tagging* link in left menu inside the *Installations*.

Then, specify the list of tags to be assigned by typing a comma-separated list into the "Tags" field. Then, click on browse to select the CSV file you want to import. The first column in the CSV file must include the GUIDs of the installations that you want to tag.

5.8.7 Tag List

Click on the "Tag List" link in left menu inside the Installations section to view a list of tags. From here, you can rename or remove tags. You can also access the list of active installations for a certain tag export them directly into a CSV or XLS file, similar to the *Installation Filters* section.

The screenshot shows the 'Tags' section of the Acme, Inc. Network Service. The table below lists the tags and their active installations:

Name	Description	Active Installations
super clients		8
problematic		8
premium support		240
OSBC2009		6
interested in X service		366

Each row in the table includes icons for search, CSV export, XLS export, and delete. A 'NEW' button is located below the table.

5.8.8 Tag Cloud

By clicking on *Tag Cloud*, you can view a cloud of all tags. The larger the text for a tag, the more active installations the tag applies to. By clicking on the tag, you can view a list of active installations that the tag has been applied to..

The screenshot shows the 'Tag Cloud' section of the Acme, Inc. Network Service. The tags are displayed as follows:

super clients(8) OSBC2009(6) problematic(8) interested in X service(366) premium support(240)

5.9 Clients

The Clients allows you to manage your clients and which installations are bound to them. You can also view information regarding the number of installations and machines assigned to a specific client.

5.9.1 Main view

The main view shows a list of current clients and the number of installations (total and active) that are assigned to each client.

The screenshot shows the 'Clients' section of the Acme, Inc. Network Service. The table below lists the clients and their active installations:

Name	Email	Active Installations	Installations
Example Client		2	2
Product Client	client@acme.mail	3	6
Product Client 2		0	3

Each row in the table includes icons for search, edit, and delete. A 'CHANGE' button is located below the table.

The icons to the right of each entry allow you to view additional information, edit the client properties, or remove the client.

5.9.2 Show Client Details

To view details on a specific client, click on the client name or on *show*.

The screenshot shows the 'Client Details' page for a 'Product Client'. The client's email is 'client@acme.mail'. It has 6 installations, 3 active installations, and 2 machines. Below this, there are two sections for 'Installations by Machine'. The first machine (5ae9f183-de0e-94bb-d586) has two hosts: 'access. [redacted].com' with latest contact on 25 Jun 10:28 and first contact on 24 Jun 12:07, and another 'access. [redacted].com' with latest contact on 11 Jul 07:24 and first contact on 25 Jun 13:29. The second machine (59378386-c557-f74f-5c89) has one host: '41_240 [redacted]' with latest contact on 17 Jul 14:00 and first contact on 16 Jul 13:09. At the bottom, there are '10 results' and a 'CHANGE' button.

On the *Client Detail* screen, you can view the number of installations and machines detected, as well as how the installations distributed across those machines. To view details on a particular installation, click on the host name..

5.9.3 New Client

To create a new client, use the "New Client" link on left menu. It allows you to create a new client and assign it to one or more installations.

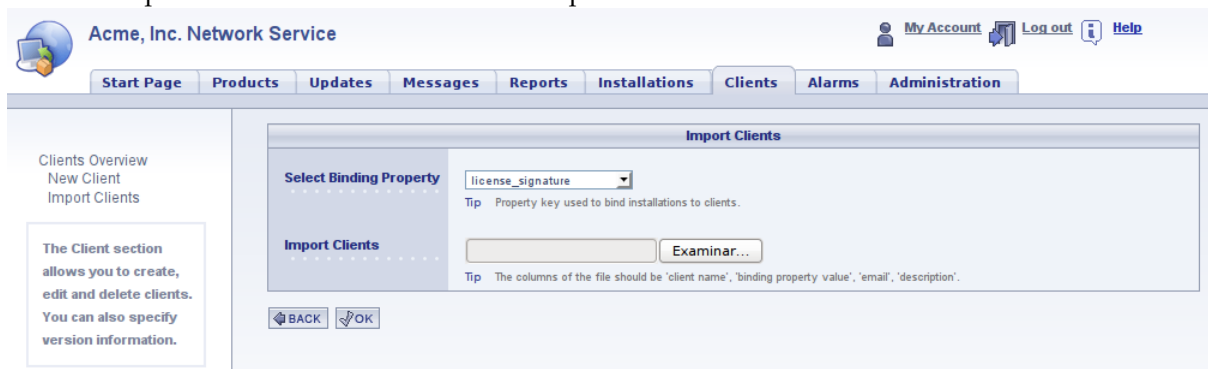
The screenshot shows the 'New Client' form. The 'Name' field contains 'New Client'. The 'Email' field contains 'new@client.com'. The 'Description' field contains 'This is a new client'. The 'Select Binding Property' dropdown is set to 'license_license_id'. Below this, there is a tip: 'Tip Property key used to bind installations to clients.' The 'Binding Rules' section contains three rules: '038410923', '439419230', and '431432431', each with a 'remove' link. At the bottom, there are 'CREATE' and 'BACK' buttons.

Select the binding property and add a list of rules to setup the installations of the new client. Use the "add binding" or "remove" controls to add or delete those rules. If any of the rules are met, the

installation will be assigned to that client.

5.9.4 Import Clients List

Use the "Import Clients" link on left menu to import a client list.



To import the clients, select the binding property and provide a CSV file formatted as follows: *client name, binding property value, email, description*.

5.9.5 Edit Client

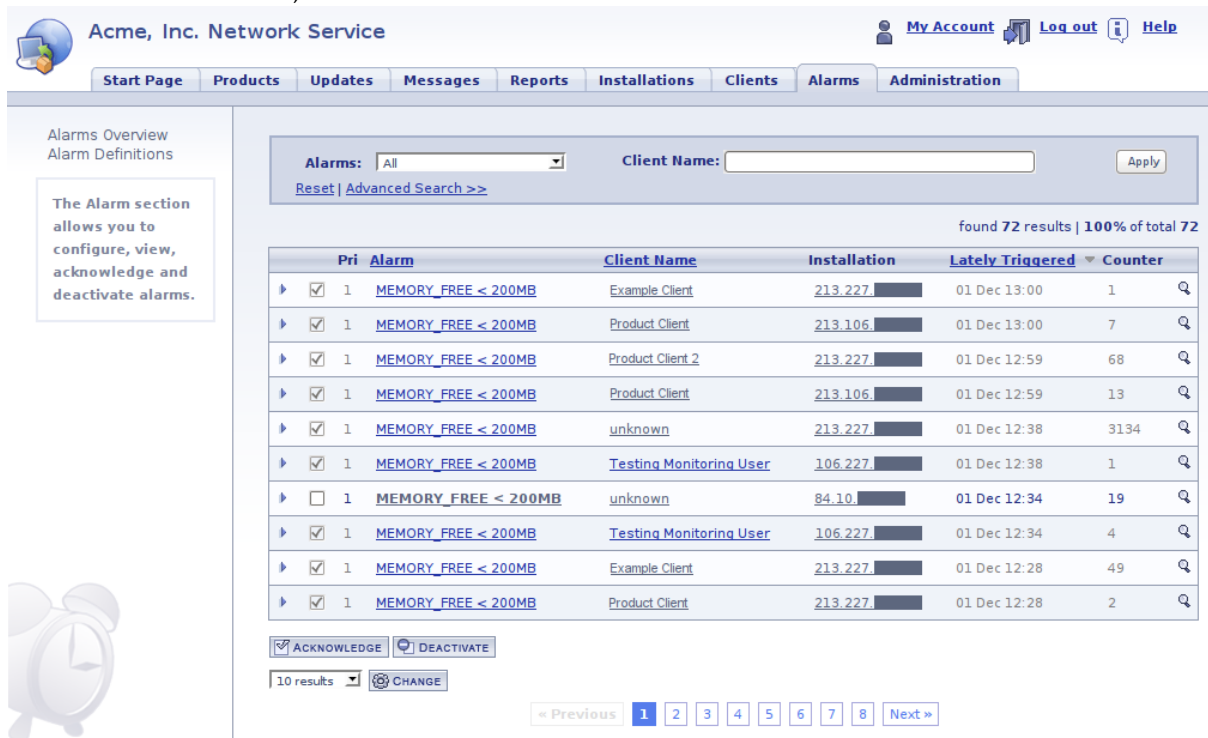
Click edit to make changes to a client.

The form is similar to the one used to add new clients. It allows you to edit client properties and which conditions must be met to attach an installation to the owned by current client.

5.10 Alarms

The *Alarms* section shows a completelist of alarms for all installations. Alarms can be filtered by priority, product version, client name and installation groups. From this page, you can view, acknowledge and deactivate alarms.

To access this section, use the Alarms tab in main menu.



5.10.1 Show Alarm

By clicking on an alarm, you can view the following details about a specific alarm:

- Alarm name
- Counter of number of checks done while alarm has been maintained as active
- Start and End times of the alarm condition
- Installation and Client information

The screenshot shows the 'Acme, Inc. Network Service' web interface. The top navigation bar includes 'Start Page', 'Products', 'Updates', 'Messages', 'Reports', 'Installations', 'Clients', 'Alarms', and 'Administration'. The 'Alarms' section is active. On the left, there is a sidebar with 'Alarms Overview' and 'Alarm Definitions'. A text box explains: 'The Alarm section allows you to configure, view, acknowledge and deactivate alarms.' The main content area displays the details for an alarm with the following information:

- Priority:** 1
- Counter:** 19
- Alarm Type:** MEMORY_FREE < 200MB
- First Time Triggered:** 2008-07-14 11:36 AM
- Last Time Triggered:** 2008-12-01 12:34 PM
- Description:** MEMORY_FREE < 200MB

Below this, there are sections for 'Installation - monitoring' (Host: 84.10. [redacted]) and 'Client' (Email, Description). A 'BACK' button is at the bottom left.

NOTE

By default, inactive and testing installations are not included in the list.

5.10.2 Alarms Definitions

Use the "Alarm Definitions" link in the left menu in the *Alarms* section to manage defined Alarms..

5.10.2.1 Main View

The main view shows a list of currently defined alarms and the controls to show, edit or remove them.

The screenshot shows the 'Acme, Inc. Network Service' web interface. The top navigation bar is the same as in the previous screenshot. The 'Alarms' section is active. The sidebar shows 'Alarms Overview' and 'Alarm Definitions'. A text box explains: 'The Alarm section allows you to configure, view, acknowledge and deactivate alarms.' The main content area displays the 'Alarm Definitions' table:

Name	Priority	Installation Group	Rules	Notify			
Low disk space	1	Registered Windows users	Compare Numbers	false			
MEMORY_FREE > 1000	0	All Installations	Compare Numbers	false			
NUMBER_OF_DOCUMENTS = 0	1	Linux Users	Compare Numbers	true			
NUMBER_OF_USERS >= 3	0	Linux Users	Compare Numbers	false			

A 'NEW' button is located below the table.

5.10.2.2 Show Alarm Definition

Clicking on "Show Alarm Definition", you can view details on a specific alarm definition.

Acme, Inc. Network Service

My Account Log out Help

Start Page Products Updates Messages Reports Installations Clients Alarms Administration

Alarms Overview
Alarm Definitions

The Alarm section allows you to configure, view, acknowledge and deactivate alarms.

Alarm Definitions

Details	
Name	Low disk space
Priority	1
Notification	will NOT be sent
Installation Group	Registered Windows users
Rules	Type: Compare Numbers property fs_free <= value 12000
Description	

EDIT DELETE BACK

5.10.2.3 New / Edit Alarm Definition

Click on *Edit* to edit an alarm definition. To enter in a new alarm, click *New* at the bottom of the list. In both cases the same form is used.

Acme, Inc. Network Service

My Account Log out Help

Start Page Products Updates Messages Reports Installations Clients Alarms Administration

Alarms Overview
Alarm Definitions

The Alarm section allows you to configure, view, acknowledge and deactivate alarms.

Alarm Definitions

Product Alarm_Definition

Name: 100 users reached

Notification: Notification title: High Number of Users detected
 Do you want to notify?

Priority: 1

Installation Group: Linux Users

Rule: Compare Numbers
number_of_users >= 100

Description:

CREATE BACK

The alarm definition will be applied to a specific group of installations and the alarm will be created automatically when the related rule is passed.

A new alarm will be created for a specific installation when the following conditions are satisfied: the alarm definition is active, the installation belongs to the configured installation group and the rule is passed. A rule is defined as a numeric or string comparison with any of the property values, or a specific event name. The active status is configured using checkbox "Do you want to activate it?".

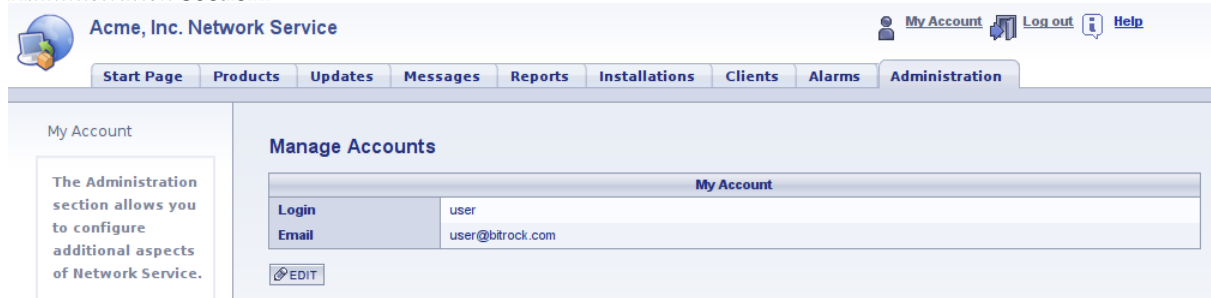
When alarms are triggered, an email notification can be sent. To trigger email notifications, check the checkbox next to "Do you want to notify?" and insert the notification name, which will be used in the mail subject.

5.11 Administration

The Administration section allows you to configure other aspects of the application.

5.11.1 My account

Access the "My account" by clicking on *My Account* at the top of the page or in the left menu inside the *Administration* section.



The administration allows you to see and modify the information about your Network Service user account.

