

SUSE Manager 4.2

Large Deployments Guide

January 27 2023



Table of Contents

Large Deployments Guide Overview	1
1. Hardware Requirements	2
2. Using a Single Server to Manage Large Scale Deployments	4
2.1. Operation Recommendations	4
2.1.1. Salt Client Onboarding Rate	4
2.1.2. Salt Clients and the RNG	4
2.1.3. Clients Running with Unaccepted Salt Keys	5
2.1.4. Disabling the Salt Mine	5
2.1.5. Disable Unnecessary Taskomatic jobs	5
2.1.6. Swap and Monitoring	6
2.1.7. AES Key Rotation	6
3. Using Multiple Servers to Manage Large Scale Deployments	8
3.1. Hub Requirements	8
3.1.1. Peripheral Servers	8
3.2. Hub Installation	9
3.3. Using the Hub API	10
3.4. Hub XMLRPC API Namespaces	10
3.5. Hub XMLRPC API Authentication Modes	11
3.5.1. Authentication Examples	11
4. Managing Large Scale Deployments in a Retail Environment	17
5. Tuning Large Scale Deployments	18
5.1. The Tuning Process	18
5.2. Environmental Variables	20
5.3. Parameters	21
5.3.1. MaxClients	21
5.3.2. ServerLimit	22
5.3.3. maxThreads	22
5.3.4. connectionTimeout	23
5.3.5. keepAliveTimeout	23
5.3.6. Tomcat's -Xmx	24
5.3.7. java.message_queue_thread_pool_size	24
5.3.8. java.salt_batch_size	25
5.3.9. java.salt_event_thread_pool_size	26
5.3.10. java.salt_presence_ping_timeout	26

Large Deployments Guide Overview

Updated: 2023-01-27

SUSE Manager is designed by default to work on small and medium scale installations. For installations with more than 1000 clients per SUSE Manager Server, adequate hardware sizing and parameter tuning must be performed.

There is no hard maximum number of supported systems. Many factors can affect how many clients can reliably be used in a particular installation. Factors can include which features are used, and how the hardware and systems are configured.



Large installations require standard Salt clients. These instructions cannot be used in environments using traditional clients or Salt SSH minions.

There are two main ways to manage large scale deployments. You can manage them with a single SUSE Manager Server, or you can use multiple servers in a hub. Both methods are described in this book.

With large scale environments one should also consider the usage of SUSE Manager Proxies. Sizing and location of the Proxies will depend on the deployment topology. For more information, see [Installation](#) › [Install-proxy](#).

Additionally, if you are operating within a Retail environment, you can use SUSE Manager for Retail to manage large deployments of point-of-service terminals. There is an introduction to SUSE Manager for Retail in this book.

Tuning and monitoring large scale deployments can differ from smaller installations. This book contains guidance for both tuning and monitoring within larger installations.

Chapter 1. Hardware Requirements

Not all problems can be solved with better hardware, but choosing the right hardware is an absolute necessity for large scale deployments.

The minimum requirements for the SUSE Manager Server are:

- Eight or more recent x86-64 CPU cores.
- 32 GiB RAM. For installations with thousands of clients, use 64 GB or more.
- Fast I/O storage devices, such as locally attached SSDs. For PostgreSQL data directories, we recommend locally attached RAID-0 SSDs.

If the SUSE Manager Server is virtualized, enable the `elevator=noop` kernel command line option, for the best input/output performance. You can check the current status with `cat /sys/block/<DEVICE>/queue/scheduler`. This command will display a list of available schedulers with the currently active one in brackets. To change the scheduler before a reboot, use `echo noop > /sys/block/<DEVICE>/queue/scheduler`.

The minimum requirements for the SUSE Manager Proxy are:

- One SUSE Manager Proxy per 500-1000 clients, depending on available network bandwidth.
- Two or more recent x86-64 CPU cores.
- 16 GB RAM, and sufficient storage for caching.

Clients should never be directly attached to the SUSE Manager Server in production systems.

In large scale installations, the SUSE Manager Proxy is used primarily as a local cache for content between the server and clients. Using proxies in this way can substantially reduce download time for clients, and decrease Server egress bandwidth use.

The number of clients per proxy will affect the download time. Always take network structure and available bandwidth into account.

We recommend you estimate the download time of typical usage to determine how many clients to connect to each proxy. To do this, you will need to estimate the number of package upgrades required in every patch cycle. You can use this formula to calculate the download time:

$$\text{Size of updates} * \text{Number of clients} / \text{Theoretical download speed} / 60$$

For example, the total time needed to transfer 400 MB of upgrades through a physical link speed of 1 GB/s to 3000 clients:

$$400 \text{ MB} * 3000 / 119 \text{ MB/s} / 60 = 169 \text{ min}$$

Chapter 2. Using a Single Server to Manage Large Scale Deployments

This section discusses how to set up a single SUSE Manager Server to manage a large number of clients. It contains some recommendations for hardware and networking, and an overview of the tuning parameters that you need to consider in a large scale deployment.

2.1. Operation Recommendations

This section contains a range of recommendations for large scale deployments.



Always start small and scale up gradually. Monitor the server as you scale to identify problems early.

2.1.1. Salt Client Onboarding Rate

The rate at which SUSE Manager can onboard clients is limited and depends on hardware resources. Onboarding clients at a faster rate than SUSE Manager is configured for will build up a backlog of unprocessed keys. This slows down the process and can potentially exhaust resources. We recommend that you limit the acceptance key rate programmatically. A safe starting point would be to onboard a client every 15 seconds. You can do that with this command:

```
for k in $(salt-key -l un|grep -v Unaccepted); do salt-key -y -a $k; sleep 15; done
```

2.1.2. Salt Clients and the RNG

All communication to and from Salt clients is encrypted. During client onboarding, Salt uses asymmetric cryptography, which requires available entropy from the Random Number Generator (RNG) facility in the kernel. If sufficient entropy is not available from the RNG, it will significantly slow down communications. This is especially true in virtualized environments. Ensure enough entropy is present, or change the virtualization host options.

You can check the amount of available entropy with the `cat /proc/sys/kernel/random/entropy_avail`. It should never be below 100–200.

2.1.3. Clients Running with Unaccepted Salt Keys

Idle clients which have not been onboarded, that is clients running with unaccepted Salt keys, consume more resources than idle clients that have been onboarded. Generally, this consumes about an extra 2.5 Kb/s of inbound network bandwidth per client. For example, 1000 idle clients will consume about 2.5 Mb/s extra. This consumption will reduce almost to zero when onboarding has been completed for all clients. Limit the number of non-onboarded clients for optimal performance.

2.1.4. Disabling the Salt Mine

In older versions, SUSE Manager used a tool called Salt mine to check client availability. The Salt mine would cause clients to contact the server every hour, which created significant load. With the introduction of a more efficient mechanism in SUSE Manager 3.2, the Salt mine is no longer required. Instead, the SUSE Manager Server uses Taskomatic to ping only the clients that appear to have been offline for twelve hours or more, with all clients being contacted at least once in every twenty four hour period by default. You can adjust this by changing the `web.system_checkin_threshold` parameter in `rhnc.conf`. The value is expressed in days, and the default value is 1.

Newly registered Salt clients will have the Salt mine disabled by default. If the Salt mine is running on your system, you can reduce load by disabling it. This is especially effective if you have a large number of clients.

Disable the Salt mine by running this command on the server:

```
salt '*' state.sls util.mgr_mine_config_clean_up
```

This will restart the clients and generate some Salt events to be processed by the server. If you have a large number of clients, handling these events could create excessive load. To avoid this, you can execute the command in batch mode with this command:

```
salt --batch-size 50 '*' state.sls util.mgr_mine_config_clean_up
```

You will need to wait for this command to finish executing. Do not end the process with `Ctrl + C`.

2.1.5. Disable Unnecessary Taskomatic jobs

To minimize wasted resources, you can disable non-essential or unused Taskomatic jobs.

You can see the list of Taskomatic jobs in the SUSE Manager Web UI, at [Admin › Task Schedules](#).

To disable a job, click the name of the job you want to disable, select **Disable Schedule**, and click **[Update Schedule]**.

To delete a job, click the name of the job you want to delete, and click **[Delete Schedule]**.

We recommend disabling these jobs:

- Daily comparison of configuration files: **compare-configs-default**
- Hourly synchronization of Cobbler files: **cobbler-sync-default**
- Daily gatherer and subscription matcher: **gatherer-matcher-default**

Do not attempt to disable any other jobs, as it could prevent SUSE Manager from functioning correctly.

2.1.6. Swap and Monitoring

It is especially important in large scale deployments that you keep your SUSE Manager Server constantly monitored and backed up.

Swap space use can have significant impacts on performance. If significant non-transient swap usage is detected, you can increase the available hardware RAM.

You can also consider tuning the Server to consume less memory. For more information on tuning, see [Salt › Large-scale-tuning](#).

2.1.7. AES Key Rotation

Communications from the Salt Master to clients is encrypted with a single AES key. The key is rotated when:

- The **salt-master** process is restarted, or
- Any minion key is deleted (for example, when a client is deleted from SUSE Manager)

After the AES key has been rotated, all clients must re-authenticate to the master. By default, this happens next time a client receives a message. If you have a large number of clients (several thousands), this can cause a high CPU load on the SUSE Manager Server. If the CPU load is

excessive, we recommend that you delete keys in batches, and in off-peak hours if possible, to avoid overloading the server.

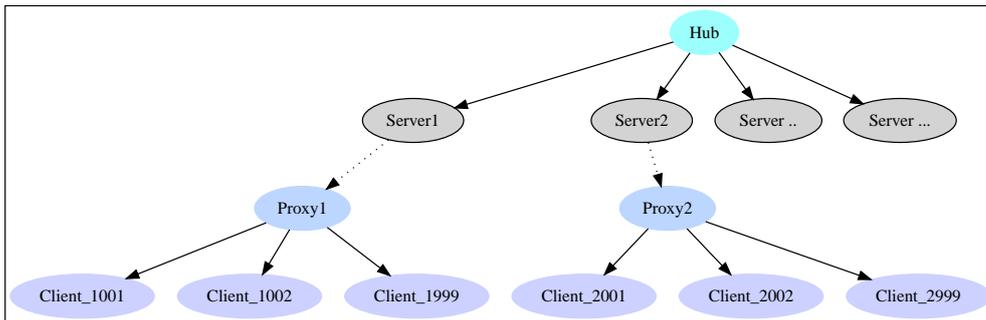
For more information, see:

- https://docs.saltstack.com/en/latest/topics/tutorials/intro_scale.html#too-many-minions-re-authing
- <https://docs.saltstack.com/en/getstarted/system/communication.html>

Chapter 3. Using Multiple Servers to Manage Large Scale Deployments

If you need to manage a large number of clients, in most cases you can do so with a single SUSE Manager Server, tuned appropriately. However, if you need to manage tens of thousands of clients, you might find it easier to use multiple SUSE Manager Servers, in a hub, to manage them.

SUSE Manager Hub helps you manage very large deployments. The typical Hub topology looks like this:



3.1. Hub Requirements

To set up a Hub installation, you require:

- One central SUSE Manager Server, which acts as the Hub Server.
- One or more additional SUSE Manager Servers, registered to the Hub as Salt clients. This document refers to these as peripheral servers.
- Any number of clients registered to the peripheral servers.
- Ensure the Hub Server and all peripheral servers are running SUSE Manager 4.1 or higher.



The Hub Server must not have clients registered to it. Clients should only be registered to the peripheral servers.

3.1.1. Peripheral Servers

Peripheral servers must be registered to the Hub Server as Salt clients. When you register the peripheral servers, assign them the appropriate **SUSE Manager Server** software channel as their base channel. Additionally, they must be registered to the Hub Server directly, do not use a proxy.

For more information about registering clients, see [Client-configuration](#) › [Registration-webui](#).

You need credentials to access the XMLRPC APIs on each server, including the Hub Server.

3.2. Hub Installation

Before you begin, you need to install the `hub-xmlrpc-api` package, and configure the Hub Server to use the API.

Procedure: Installing and Configuring the Hub XMLRPC API

1. On the Hub Server, or on a host that has access to all peripheral servers' XMLRPC APIs, install the `hub-xmlrpc-api` package. The package is available in the SUSE Manager 4.2 repositories.
2. OPTIONAL: Set the Hub XMLRPC API service to start automatically at boot time, and start it immediately:

```
--- sudo systemctl enable hub-xmlrpc-api.service sudo systemctl start hub-xmlrpc-api.service ---
```
3. OPTIONAL: Check that these parameters in the `/etc/hub/hub.conf` configuration file are correct:
 - `HUB_API_URL`: URL to the Hub Server XMLRPC API endpoint. Use the default value if you are installing `hub-xmlrpc-api` on the Hub Server.
 - `HUB_CONNECT_TIMEOUT`: the maximum number of seconds to wait for a response when connecting to a Server. Use the default value in most cases.
 - `HUB_REQUEST_TIMEOUT`: the maximum number of seconds to wait for a response when calling a Server method. Use the default value in most cases.
 - `HUB_CONNECT_USING_SSL`: use HTTPS instead of HTTP for communicating with peripheral Servers. Recommended for a secure environment.
4. Restart services to pick up configuration changes.



To use HTTPS to connect to peripheral Servers, you must set the `HUB_CONNECT_USING_SSL` parameter to `true`, and ensure that the SSL certificates for all the peripheral Servers are installed on the machine where the `hub-xmlrpc-api` service runs. Do this by copying the `RHN-ORG-TRUSTED-SSL-CERT` certificate file from each peripheral Server's `http://<server-uri>/pub/` directory to `/etc/pki/trust/anchors/`, and run `update-ca-certificates`.

3.3. Using the Hub API

Make sure the `hub-xmlrpc-api` service is started:

```
systemctl start hub-xmlrpc-api
```

Once it is running, connect to the service at port 2830 using any XMLRPC-compliant client libraries.

For examples, see [Large-deployments](#) › [Hub-auth](#).

Logs are saved in `/var/log/hub/hub-xmlrpc-api.log`. Logs are rotated weekly, or when the log file size reaches the specified limit. By default, the log file size limit is 10 MB.

3.4. Hub XMLRPC API Namespaces

The Hub XMLRPC API operates in a similar way to the SUSE Manager API. For SUSE Manager API documentation, see <https://documentation.suse.com/suma>.

The Hub XMLRPC API exposes the same methods that are available from the server's XMLRPC API, with a few differences in parameter and return types. Additionally, the Hub XMLRPC API supports some Hub-specific end points which are not available in the SUSE Manager API.

The Hub XMLRPC API supports three different namespaces:

- The **hub** namespace is used to target the Hub XMLRPC API Server. It supports Hub-specific XMLRPC endpoints which are primarily related to authentication.
- The **unicast** namespace is used to target a single server registered in the hub. It redirects any call transparently to one specific server and returns any value as if the server's XMLRPC API endpoint was used directly.
- The **multicast** namespace is used to target multiple peripheral servers registered in the hub. It redirects any call transparently to all the specified servers and returns the results in the form of a **map**.
- If you do not specify a namespace, all calls are transparently redirected to the underlying SUSE Manager Server XMLRPC API of the Hub Server. This allows you to call all available methods on the SUSE Manager Server XMLRPC API.

Methods called without specifying any of the above namespaces will be forwarded to the normal

XMLRPC API of the hub. This is the API exposed on ports 80 and 443.

Some important considerations for hub namespaces:

- Individual server IDs can be obtained using `client.hub.listServerIds(hubSessionKey)`.
- The `unicast` namespace assumes all methods receive `hubSessionKey` and `serverID` as their first two parameters, then any other parameter as specified by the regular Server API.

```
client.unicast.[namespace].[methodName](hubSessionKey, serverId, param1, param2)
```

- The `hubSessionKey` can be obtained using different authentication methods. For more information, see [Large-deployments > Hub-auth](#).
- The `multicast` namespace assumes all methods receive `hubSessionKey`, a list of `ServerID` values, then lists of per-server parameters as specified by the regular server XMLRPC API. The return value is a `map`, with `Successful` and `Failed` entries for each server involved in the call.

```
client.multicast.[namespace].[methodname](hubSessionKey, [serverId1, serverId2], [param1_s1, param1_s2], [param2_s1, param2_s2])
```

3.5. Hub XMLRPC API Authentication Modes

The Hub XMLRPC API supports three different authentication modes:

- Manual mode (default): API credentials must be explicitly provided for each server.
- Relay mode: the credentials used to authenticate with the Hub are also used to authenticate to each server. You must provide a list of servers to connect to.
- Auto-connect mode: credentials are reused for each server, and any peripheral server you have access to is automatically connected.

3.5.1. Authentication Examples

This section provides examples of each authentication method.

Example: Manual Authentication

In manual mode, credentials have to be explicitly provided for each peripheral server before you can connect to it.

A typical workflow for manual authentication is:

1. Credentials for the Hub are passed to the **login** method, and a session key for the Hub is returned (**hubSessionKey**).
2. Using the session key from the previous step, SUSE Manager Server IDs are obtained for all the peripheral servers attached to the Hub via the **hub.listServerIds** method.
3. Credentials for each peripheral server are provided to the **attachToServers** method. This performs authentication against each server's XMLRPC API endpoint.
4. A **multicast** call is performed on a set of servers. This is defined by **serverIds**, which contains the IDs of the servers to target. In the background, **system.list_system** is called on each server's XMLRPC API
5. Hub aggregates the results and returns the response in the form of a **map**. The map has two entries:
 - **Successful**: list of responses for those peripheral servers where the call succeeded.
 - **Failed**: list of responses for those peripheral servers where the call failed.



If you want to call a method on just one SUSE Manager Server, then Hub API also provides a **unicast** namespace. In this case, the response will be a single value and not a map, in the same way as if you called that SUSE Manager server's API directly.

Listing 1. Example Python Script for Manual Authentication:

```
#!/usr/bin/python
import xmlrpclib

HUB_XMLRPC_API_URL = "<HUB_XMLRPC_API_URL>"
HUB_USERNAME = "<USERNAME>"
HUB_PASSWORD = "<PASSWORD>"

client = xmlrpclib.Server(HUB_XMLRPC_API_URL, verbose=0)

hubSessionKey = client.hub.login(HUB_USERNAME, HUB_PASSWORD)

# Get the server IDs
serverIds = client.hub.listServerIds(hubSessionKey)

# For simplicity, this example assumes you are using the same username and password
# here, as on the hub server.
# However, in most cases, every server has its own individual credentials.
usernames = [HUB_USERNAME for s in serverIds]
passwords = [HUB_PASSWORD for s in serverIds]

# Each server uses the credentials set above, client.hub.attachToServers needs
# them passed as lists with as many elements as there are servers.
client.hub.attachToServers(hubSessionKey, serverIds, usernames, passwords)

# Perform the operation
systemsPerServer = client.multicast.system.list_systems(hubSessionKey, serverIds)
successfulResponses = systemsPerServer["Successful"]["Responses"]
failedResponses = systemsPerServer["Failed"]["Responses"]

for system in successfulResponses:
    print (system)

#logout
client.hub.logout(hubSessionKey)
```

Example: Relay Authentication

In relay authentication mode, the credentials used to sign in to the Hub API are also used to sign in into the APIs of the peripheral servers the user wants to work with. In this authentication mode, it is assumed that the same credentials are valid for every server, and that they correspond to a user with appropriate permissions.

After signing in, you must call the `attachToServers` method. This method defines the servers to target in all subsequent calls.

A typical workflow for relay authentication is:

1. Credentials for the Hub are passed to the **loginWithAuthRelayMode** method, and a session key for the Hub is returned (**hubSessionKey**).
2. Using the session key from the previous step, SUSE Manager Server IDs are obtained for all the peripheral servers attached to the Hub via the **hub.listServerIds** method
3. A call to **attachToServers** is made, and the same credentials used to sign in to the Hub are passed to each server. This performs authentication against each server's XMLRPC API endpoint.
4. A **multicast** call is performed on a set of servers. This is defined by **serverIds**, which contains the IDs of the servers to target. In the background, **system.list_system** is called on each server's XMLRPC API.
5. Hub aggregates the results and returns the response in the form of a **map**. The map has two entries:
 - **Successful**: list of responses for those peripheral servers where the call succeeded.
 - **Failed**: list of responses for those peripheral servers the call failed.

Listing 2. Example Python Script for Relay Authentication:

```
#!/usr/bin/python
import xmlrpclib

HUB_XMLRPC_API_URL = "<HUB_XMLRPC_API_URL>"
HUB_USERNAME = "<USERNAME>"
HUB_PASSWORD = "<PASSWORD>"

client = xmlrpclib.Server(HUB_XMLRPC_API_URL, verbose=0)

hubSessionKey = client.hub.loginWithAuthRelayMode(HUB_USERNAME, HUB_PASSWORD)

#Get the server IDs
serverIds = client.hub.listServerIds(hubSessionKey)

#authenticate those servers(same credentials will be used as of hub to authenticate)
client.hub.attachToServers(hubSessionKey, serverIds)

# perform the needed operation
systemsPerServer = client.multicast.system.list_systems(hubSessionKey, serverIds)
successfulResponses = systemsPerServer["Successful"]["Responses"]
failedResponses = systemsPerServer["Failed"]["Responses"]

for system in successfulResponses:
    print (system)

#logout
client.hub.logout(hubSessionKey)
```

Example: Auto-Connect Authentication

Auto-connect mode is similar to relay mode, it uses the Hub credentials to sign in in to all peripheral servers. However, there is no need to use the `attachToServers` method, as auto-connect mode connects to all available peripheral servers. This occurs at the same time as you sign in to the Hub.

A typical workflow for auto-connect authentication is:

1. Credentials for the Hub are passed to the `loginWithAutoconnectMode` method, and a session key for the Hub is returned (`hubSessionKey`).
2. A `multicast` call is performed on a set of servers. This is defined by `serverIds`, which contains the IDs of the servers to target. In the background, `system.list_system` is called on each server's XMLRPC API.

3. Hub aggregates the results and returns the response in the form of a **map**. The map has two entries:

- **Successful**: list of responses for those peripheral servers where the call succeeded.
- **Failed**: list of responses for those peripheral servers where the call failed.

Listing 3. Example Python Script for Auto-Connect Authentication:

```
#!/usr/bin/python
import xmlrpclib

HUB_XMLRPC_API_URL = "<HUB_XMLRPC_API_URL>"
HUB_USERNAME = "<USERNAME>"
HUB_PASSWORD = "<PASSWORD>"

client = xmlrpclib.Server(HUB_XMLRPC_API_URL, verbose=0)

loginResponse = client.hub.loginWithAutoconnectMode(HUB_USERNAME, HUB_PASSWORD)
hubSessionKey = loginResponse["SessionKey"]

#Get the server IDs
serverIds = client.hub.listServerIds(hubSessionKey)

# perform the needed operation
systemsPerServer = client.multicast.system.list_systems(hubSessionKey, serverIds)
successfulResponses = systemsPerServer["Successful"]["Responses"]
failedResponses = systemsPerServer["Failed"]["Responses"]

for system in successfulResponses:
    print (system)

#logout
client.hub.logout(hubSessionKey)
```

Chapter 4. Managing Large Scale Deployments in a Retail Environment

SUSE Manager for Retail 4.2 is an open source infrastructure management solution, optimized and tailored specifically for the retail industry. It uses the same technology as SUSE Manager, but is customized to address the needs of retail organizations.

SUSE Manager for Retail is designed for use in retail situations where customers can use point-of-service terminals to purchase or exchange goods, take part in promotions, or collect loyalty points. In addition to retail installations, it can also be used for novel purposes, such as maintaining student computers in an educational environment, or self-service kiosks in banks or hospitals.

SUSE Manager for Retail is intended for use in installations that include servers, workstations, point-of-service terminals, and other devices. It allows administrators to install, configure, and update the software on their servers, and manage the deployment and provisioning of point-of-service machines.

Point-of-Service (POS) terminals can come in many different formats, such as point-of-sale terminals, kiosks, digital scales, self-service systems, and reverse-vending systems. Every terminal, however, is provided by a vendor, who set basic information about the device in the firmware. SUSE Manager for Retail accesses this vendor information to determine how best to work with the terminal in use.

In most cases, different terminals will require a different operating system (OS) image to ensure they work correctly. For example, an information kiosk has a high-resolution touchscreen, where a cashier terminal might only have a very basic display. While both of these terminals require similar processing and network functionality, they will require different OS images. The OS images ensure that the different display mechanisms work correctly.

For more information about setting up and using SUSE Manager for Retail, see [Retail](#) > [Retail-overview](#).

Chapter 5. Tuning Large Scale Deployments

SUSE Manager is designed by default to work on small and medium scale installations. For installations with more than 1000 clients per SUSE Manager Server, adequate hardware sizing and parameter tuning must be performed.



The instructions in this section can have severe and catastrophic performance impacts when improperly used. In some cases, they can cause SUSE Manager to completely cease functioning. Always test changes before implementing them in a production environment. During implementation, take care when changing parameters. Monitor performance before and after each change, and revert any steps that do not produce the expected result.



We strongly recommend that you contact SUSE Consulting for assistance with tuning.

SUSE will not provide support for catastrophic failure when these advanced parameters are modified without consultation.



Tuning is not required on installations of fewer than 1000 clients. Do not perform these instructions on small or medium scale installations.

5.1. The Tuning Process

Any SUSE Manager installation is subject to a number of design and infrastructure constraints that, for the purposes of tuning, we call environmental variables. Environmental variables can include the total number of clients, the number of different operating systems under management, and the number of software channels.

Environmental variables influence, either directly or indirectly, the value of most configuration parameters. During the tuning process, the configuration parameters are manipulated to improve system performance.

Before you begin tuning, you will need to estimate the best setting for each environment variable, and adjust the configuration parameters to suit.

To help you with the estimation process, we have provided you with a dependency graph. Locate the environmental variables on the dependency graph to determine how they will influence other

- Solid connecting lines indicate that changing a configuration parameter requires checking another one to prevent issues

After the initial tuning has been completed, you will need to consider tuning again in these cases:

- If your tuning inputs change significantly
- If special conditions arise that require a certain parameter to be changed. For example, if specific warnings appear in a log file.
- If performance is not satisfactory

To re-tune your installation, you will need to use the dependency graph again. Start from the node where significant change has happened.

5.2. Environmental Variables

This section contains information about environmental variables (inputs to the tuning process).

Network Bandwidth

A measure of the typically available egress bandwidth from the SUSE Manager Server host to the clients or SUSE Manager Proxy hosts. This should take into account network hardware and topology as well as possible capacity limits on switches, routers, and other network equipment between the server and clients.

Channel count

The number of expected channels to manage. Includes any vendor-provided, third-party, and cloned or staged channels.

Client count

The total number of actual or expected clients. It is important to tune any parameters in advance of a client count increase, whenever possible.

OS mix

The number of distinct operating system versions that managed clients have installed. This is ordered by family (SUSE Linux Enterprise, openSUSE, Red Hat Enterprise Linux, or Ubuntu based). Storage and computing requirements are different in each case.

User count

The expected maximum amount of concurrent users interacting with the Web UI plus the

number of programs simultaneously using the XMLRPC API. Includes `spacecmd`, `spacewalk-clone-by-date`, and similar.

5.3. Parameters

This section contains information about the available parameters.

5.3.1. MaxClients

Description	The maximum number of HTTP requests served simultaneously by Apache httpd. Proxies, Web UI, and XMLRPC API clients each consume one. Requests exceeding the parameter will be queued and might result in timeouts.
Tune when	<code>User count</code> and proxy count increase significantly and this line appears in <code>/var/log/apache2/error_log: [...]</code> <code>[mpm_prefork:error] [pid ...] AH00161: server reached MaxRequestWorkers setting, consider raising the MaxRequestWorkers setting.</code>
Value default	150
Value recommendation	150–500
Location	<code>/etc/apache2/server-tuning.conf</code> , in the <code>prefork.c</code> section
Example	<code>MaxClients = 200</code>
After changing	Immediately change <code>ServerLimit</code> and check <code>maxThreads</code> for possible adjustment.
Notes	This parameter was renamed to <code>MaxRequestWorkers</code> , both names are valid.
More information	https://httpd.apache.org/docs/2.4/en/mod/mpm_common.html#maxrequestworkers

5.3.2. ServerLimit

Description	The number of Apache httpd processes serving HTTP requests simultaneously. The number must equal MaxClients .
Tune when	MaxClients changes
Value default	150
Value recommendation	The same value as MaxClients
Location	<code>/etc/apache2/server-tuning.conf</code> , in the <code>prefork.c</code> section
Example	<code>ServerLimit = 200</code>
More information	https://httpd.apache.org/docs/2.4/en/mod/mpm_common.html#serverlimit

5.3.3. maxThreads

Description	The number of Tomcat threads dedicated to serving HTTP requests
Tune when	MaxClients changes. <code>maxThreads</code> must always be equal or greater than MaxClients
Value default	150
Value recommendation	The same value as MaxClients
Location	<code>/etc/tomcat/server.xml</code>
Example	<code><Connector port="8009" protocol="AJP/1.3" redirectPort="8443" URIEncoding="UTF-8" address="127.0.0.1" maxThreads="200" connectionTimeout="20000"/></code>
More information	https://tomcat.apache.org/tomcat-9.0-doc/config/http.html

5.3.4. connectionTimeout

Description	The number of milliseconds before a non-responding AJP connection is forcibly closed.
Tune when	Client count increases significantly and AH00992 , AH00877 , and AH01030 errors appear in Apache error logs during a load peak.
Value default	900000
Value recommendation	20000-3600000
Location	<code>/etc/tomcat/server.xml</code>
Example	<code><Connector port="8009" protocol="AJP/1.3" redirectPort="8443" URIEncoding="UTF-8" address="127.0.0.1" maxThreads="200" connectionTimeout="1000000" keepAliveTimeout="300000"/></code>
More information	https://tomcat.apache.org/tomcat-9.0-doc/config/http.html

5.3.5. keepAliveTimeout

Description	The number of milliseconds without data exchange from the JVM before a non-responding AJP connection is forcibly closed.
Tune when	Client count increases significantly and AH00992 , AH00877 , and AH01030 errors appear in Apache error logs during a load peak.
Value default	300000
Value recommendation	20000-600000
Location	<code>/etc/tomcat/server.xml</code>

Example	<code><Connector port="8009" protocol="AJP/1.3" redirectPort="8443" URIEncoding="UTF-8" address="127.0.0.1" maxThreads="200" connectionTimeout="1000000" keepAliveTimeout="400000"/></code>
More information	https://tomcat.apache.org/tomcat-9.0-doc/config/http.html

5.3.6. Tomcat's `-Xmx`

Description	The maximum amount of memory Tomcat can use
Tune when	<code>java.message_queue_thread_pool_size</code> is increased or <code>OutOfMemoryException</code> errors appear in <code>/var/log/rhn/rhn_web_ui.log</code>
Value default	1 GiB
Value recommendation	4-8 GiB
Location	<code>/etc/sysconfig/tomcat</code>
Example	<code>JAVA_OPTS="... -Xmx8G ..."</code>
After changing	Check memory usage
More information	https://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html

5.3.7. `java.message_queue_thread_pool_size`

Description	The maximum number of threads in Tomcat dedicated to asynchronous operations
Tune when	Client count increases significantly
Value default	5
Value recommendation	50 - 150

Location	<code>/etc/rhn/rhn.conf</code>
Example	<code>java.message_queue_thread_pool_size = 50</code>
After changing	Check <code>hibernate.c3p0.max_size</code> , as each thread consumes a PostgreSQL connection, starvation might happen if the allocated connection pool is insufficient. Check <code>thread_pool</code> , as each thread might perform Salt API calls, starvation might happen if the allocated Salt thread pool is insufficient. Check Tomcat's <code>-Xmx</code> , as each thread consumes memory, <code>OutOfMemoryException</code> might be raised if insufficient.
Notes	Incoming Salt events are handled in separate thread pool, see <code>java.salt_event_thread_pool_size</code>
More information	<code>man rhn.conf</code>

5.3.8. `java.salt_batch_size`

Description	The maximum number of minions concurrently executing a scheduled action.
Tune when	<code>Client count</code> reaches several thousands and actions are not executed quickly enough.
Value default	200
Value recommendation	200-500
Location	<code>/etc/rhn/rhn.conf</code>
Example	<code>java.salt_batch_size = 300</code>
After changing	Check <code>memory usage</code> . Monitor memory usage closely before and after the change.
More information	Salt › Salt-rate-limiting

5.3.9. java.salt_event_thread_pool_size

Description	The maximum number of threads in Tomcat dedicated to handling of incoming Salt events.
Tune when	The number of queued Salt events grows. Typically, this can happen during onboarding of large number of minions with higher value of java.salt_presence_ping_timeout . The number of events can be queried by <code>echo "select count(*) from susesaltevent;" spacewalk-sql --select-mode-direct -</code>
Value default	8
Value recommendation	20-100
Location	<code>/etc/rhn/rhn.conf</code>
Example	<code>java.salt_event_thread_pool_size = 50</code>
After changing	Check the length of Salt event queue. Check hibernate.c3p0.max_size , as each thread consumes a PostgreSQL connection, starvation might happen if the allocated connection pool is insufficient. Check thread_pool , as each thread might perform Salt API calls, starvation might happen if the allocated Salt thread pool is insufficient. Check Tomcat's -Xmx , as each thread consumes memory, OutOfMemoryException might be raised if insufficient.
More information	<code>man rhn.conf</code>

5.3.10. java.salt_presence_ping_timeout

Description	Before any action is executed on a client, a presence ping is executed to make sure the client is reachable. This parameter sets the amount of time before a second command (find_job) is sent to the client to verify its presence. Having many clients typically means some will respond faster than others, so this timeout could be raised to accommodate for the slower ones.
Tune when	Client count increases significantly, or some clients are responding correctly but too slowly, and SUSE Manager excludes them from calls. This line appears in <code>/var/log/rhn/rhn_web_ui.log: "Got no result for <COMMAND> on minion <MINION_ID> (minion did not respond in time)"</code>
Value default	4 seconds
Value recommendation	4-400 seconds
Location	<code>/etc/rhn/rhn.conf</code>
Example	<code>java.salt_presence_ping_timeout = 40</code>
After changing	Large <code>java.salt_presence_ping_timeout</code> value can reduce overall throughput. This can be compensated by increasing <code>java.salt_event_thread_pool_size</code>
More information	Salt › Salt-timeouts === <code>java.salt_presence_ping_gather_job_timeout</code> [cols="1,1"]

| Description | Before any action is executed on a client, a presence ping is executed to make sure the client is reachable. After **`java.salt_presence_ping_timeout`** seconds have elapsed without a response, a second command (**find_job**) is sent to the client for a final check. This parameter

sets the number of seconds after the second command after which the client is definitely considered offline. Having many clients typically means some will respond faster than others, so this timeout could be raised to accommodate for the slower ones. | Tune when | [Client count](#) increases significantly, or some clients are responding correctly but too slowly, and SUSE Manager excludes them from calls. This line appears in `/var/log/rhn/rhn_web_ui.log`: "Got no result for <COMMAND> on minion <MINION_ID> (minion did not respond in time)" | Value default | 1 second | Value recommendation | 1-100 seconds | Location | `/etc/rhn/rhn.conf` | Example | `java.salt_presence_ping_gather_job_timeout = 10` | More information | [Salt](#) › [Salt-timeouts](#)

```
=== java.taskomatic_channel_repodata_workers
```

```
[cols="1,1"]
```

| Description | Whenever content is changed in a software channel, its metadata needs to be recomputed before clients can use it. Channel-altering operations include the addition of a patch, the removal of a package or a repository synchronization run. This parameter specifies the maximum number of Taskomatic threads that SUSE Manager will use to recompute the channel metadata. Channel metadata computation is both CPU-bound and memory-heavy, so raising this parameter and operating on many channels simultaneously could cause Taskomatic to consume significant resources, but channels will be available to clients sooner. | Tune when | [Channel count](#) increases significantly (more than 50), or more concurrent operations on channels are expected. | Value default | 2 | Value recommendation | 2-10 | Location | `/etc/rhn/rhn.conf` | Example | `java.taskomatic_channel_repodata_workers = 4` | After changing | Check **`taskomatic.java.maxmemory`** for adjustment, as every new thread will consume memory | More information | `man rhn.conf`

```
=== taskomatic.java.maxmemory
```

```
[cols="1,1"]
```

| Description | The maximum amount of memory Taskomatic can use. Generation of metadata, especially for some OSs, can be memory-intensive, so this parameter might need raising depending on the managed [OS mix](#). | Tune when | **`java.taskomatic_channel_repodata_workers`** increases, OSs are added to SUSE Manager (particularly Red Hat Enterprise Linux or Ubuntu), or `OutOfMemoryException` errors appear in `/var/log/rhn/rhn_taskomatic_daemon.log`. | Value default | 4096 MiB | Value recommendation | 4096-16384 MiB | Location | `/etc/rhn/rhn.conf` | Example | `taskomatic.java.maxmemory = 8192` | After changing | Check [memory usage](#). | More information | `man rhn.conf`

```
=== org.quartz.threadPool.threadCount
```

```
[cols="1,1"]
```

| Description | The number of Taskomatic worker threads. Increasing this value allows Taskomatic to serve more clients in parallel. | Tune when | [Client count](#) increases significantly | Value default | 20 | Value recommendation | 20-200 | Location | `/etc/rhn/rhn.conf` | Example | `org.quartz.threadPool.threadCount = 100` | After changing | Check `hibernate.c3p0.max_size` and `thread_pool` for adjustment | More information | <http://www.quartz-scheduler.org/documentation/2.4.0-SNAPSHOT/configuration.html>

```
=== org.quartz.scheduler.idleWaitTime
```

```
[cols="1,1"]
```

| Description | Cycle time for Taskomatic. Decreasing this value lowers the latency of Taskomatic. | Tune when | [Client count](#) is in the thousands. | Value default | 5000 ms | Value recommendation | 1000-5000 ms | Location | `/etc/rhn/rhn.conf` | Example | `org.quartz.scheduler.idleWaitTime = 1000` | More information | <http://www.quartz-scheduler.org/documentation/2.4.0-SNAPSHOT/configuration.html>

```
=== MinionActionExecutor.parallel_threads
```

```
[cols="1,1"]
```

| Description | Number of Taskomatic threads dedicated to sending commands to Salt clients as a result of actions being executed. | Tune when | [Client count](#) is in the thousands. | Value default | 1 | Value recommendation | 1-10 | Location | `/etc/rhn/rhn.conf` | Example | `taskomatic.com.redhat.rhn.taskomatic.task.MinionActionExecutor.parallel_threads = 10`

```
=== SSHMinionActionExecutor.parallel_threads
```

```
[cols="1,1"]
```

| Description | Number of Taskomatic threads dedicated to sending commands to Salt SSH clients as a result of actions being executed. | Tune when | [Client count](#) is in the hundreds. | Value default | 20 | Value recommendation | 20-100 | Location | `/etc/rhn/rhn.conf` | Example | `taskomatic.com.redhat.rhn.taskomatic.task.SSHMinionActionExecutor.parallel_threads = 40`

```
=== hibernate.c3p0.max_size
```

```
[cols="1,1"]
```

| Description | Maximum number of PostgreSQL connections simultaneously available to both Tomcat and Taskomatic. If any of those components requires more concurrent connections, their requests will be queued. | Tune when | **`java.message_queue_thread_pool_size`** or **`maxThreads`** increase significantly, or when **`org.quartz.threadPool.threadCount`** has changed significantly. Each thread consumes one connection in Taskomatic and Tomcat, having more threads than connections might result in starving. | Value default | 20 | Value recommendation | 100 to 200, higher than the maximum of **`java.message_queue_thread_pool_size`** + **`maxThreads`** and **`org.quartz.threadPool.threadCount`** | Location | `/etc/rhn/rhn.conf` | Example | `hibernate.c3p0.max_size = 100` | After changing | Check **`max_connections`** for adjustment. | More information | <https://www.mchange.com/projects/c3p0/#maxPoolSize>

```
=== rhn-search.java.maxmemory
```

```
[cols="1,1"]
```

| Description | The maximum amount of memory that the `rhn-search` service can use. | Tune when | `Client count` increases significantly, and `OutOfMemoryException` errors appear in `journalctl -u rhn-search`. | Value default | 512 MiB | Value recommendation | 512-4096 MiB | Location | `/etc/rhn/rhn.conf` | Example | `rhn-search.java.maxmemory = 4096` | After changing | Check `memory usage`.

```
=== shared_buffers
```

```
[cols="1,1"]
```

| Description | The amount of memory reserved for PostgreSQL shared buffers, which contain caches of database tables and index data. | Tune when | RAM changes | Value default | 25% of total RAM | Value recommendation | 25-40% of total RAM | Location | `/var/lib/pgsql/data/postgresql.conf` | Example | `shared_buffers = 8192MB` | After changing | Check `memory usage`. | More information | <https://www.postgresql.org/docs/10/runtime-config-resource.html#GUC-SHARED-BUFFERS>

```
=== max_connections
```

```
[cols="1,1"]
```

| Description | Maximum number of PostgreSQL connections available to applications. More connections allow for more concurrent threads/workers in various components (in particular Tomcat and Taskomatic), which generally improves performance. However, each connection consumes resources, in particular **work_mem** megabytes per sort operation per connection. | Tune when | **hibernate.c3p0.max_size** changes significantly, as that parameter determines the maximum number of connections available to Tomcat and Taskomatic | Value default | 400 | Value recommendation | $2 * \text{hibernate.c3p0.max_size} + 50$, if less than 1000 | Location | `/var/lib/pgsql/data/postgresql.conf` | Example | `max_connections = 250` | After changing | Check [memory usage](#). Monitor memory usage closely before and after the change. | More information | <https://www.postgresql.org/docs/10/runtime-config-connection.html#GUC-MAX-CONNECTIONS>

```
=== work_mem
```

```
[cols="1,1"]
```

| Description | The amount of memory allocated by PostgreSQL every time a connection needs to do a sort or hash operation. Every connection (as specified by **max_connections**) might make use of an amount of memory equal to a multiple of **work_mem**. | Tune when | Database operations are slow because of excessive temporary file disk I/O. To test if that is happening, add `log_temp_files = 5120` to `/var/lib/pgsql/data/postgresql.conf`, restart PostgreSQL, and monitor the PostgreSQL log files. If you see lines containing **LOG: temporary file**: try raising this parameter's value to help reduce disk I/O and speed up database operations. | Value recommendation | 2-20 MB | Location | `/var/lib/pgsql/data/postgresql.conf` | Example | `work_mem = 10MB` | After changing | check if the SUSE Manager Server might need additional RAM. | More information | <https://www.postgresql.org/docs/10/runtime-config-resource.html#GUC-WORK-MEM>

```
=== effective_cache_size
```

```
[cols="1,1"]
```

| Description | Estimation of the total memory available to PostgreSQL for caching. It is the explicitly reserved memory (**shared_buffers**) plus any memory used by the kernel as cache/buffer. | Tune when | Hardware RAM or memory usage increase significantly | Value

recommendation | Start with 75% of total RAM. For finer settings, use **shared_buffers** + free memory + buffer/cache memory. Free and buffer/cache can be determined via the **free -m** command (**free** and **buff/cache** in the output respectively) | Location | `/var/lib/pgsql/data/postgresql.conf` | Example | `effective_cache_size = 24GB` | After changing | Check [memory usage](#) | Notes | This is an estimation for the query planner, not an allocation. | More information | <https://www.postgresql.org/docs/10/runtime-config-query.html#GUC-EFFECTIVE-CACHE-SIZE>

```
=== thread_pool
```

```
[cols="1,1"]
```

| Description | The number of worker threads serving Salt API HTTP requests. A higher number can improve parallelism of SUSE Manager Server-initiated Salt operations, but will consume more memory. | Tune when | **java.message_queue_thread_pool_size** or **org.quartz.threadPool.threadCount** are changed. Starvation can occur when there are more Tomcat or Taskomatic threads making simultaneous Salt API calls than there are Salt API worker threads. | Value default | 100 | Value recommendation | 100-500, but should be higher than the sum of **java.message_queue_thread_pool_size** and **org.quartz.threadPool.threadCount** | Location | `/etc/salt/master.d/susemanager.conf`, in the `rest_cherry` section. | Example | `thread_pool: 100` | After changing | Check **worker_threads** for adjustment. | More information | https://docs.saltstack.com/en/latest/ref/netapi/all/salt.netapi.rest_cherry.html#performance-tuning

```
=== worker_threads
```

```
[cols="1,1"]
```

| Description | The number of **salt-master** worker threads that process commands and replies from minions and the Salt API. Increasing this value, assuming sufficient resources are available, allows Salt to process more data in parallel from minions without timing out, but will consume significantly more RAM (typically about 70 MiB per thread). | Tune when | **Client count** increases significantly, **thread_pool** increases significantly, or **SaltReqTimeoutError** or **Message timed out** errors appear in `/var/log/salt/master`. | Value default | 8 | Value recommendation | 8-200 | Location | `/etc/salt/master.d/tuning.conf` | Example | `worker_threads: 50` | After changing | Check [memory usage](#). Monitor memory usage closely before and after the change. | More information | <https://docs.saltstack.com/en/latest/ref/configuration/master.html#worker-threads>

```
=== pub_hwm
```

```
[cols="1,1"]
```

| Description | The maximum number of outstanding messages sent by **salt-master**. If more than this number of messages need to be sent concurrently, communication with clients slows down, potentially resulting in timeout errors during load peaks. | Tune when | **Client count** increases significantly and **Salt request timed out. The master is not responding.** errors appear when pinging minions during a load peak. | Value default | 1000 | Value recommendation | 10000-100000 | Location | `/etc/salt/master.d/tuning.conf` | Example | `pub_hwm: 10000` | More information | <https://docs.saltstack.com/en/latest/ref/configuration/master.html#pub-hwm>, <https://zeromq.org/socket-api/#high-water-mark>

```
=== zmq_backlog
```

```
[cols="1,1"]
```

| Description | The maximum number of allowed client connections that have started but not concluded the opening process. If more than this number of clients connects in a very short time frame, connections are dropped and clients experience a delay re-connecting. | Tune when | **Client count** increases significantly and very many clients reconnect in a short time frame, TCP connections to the **salt-master** process get dropped by the kernel. | Value default | 1000 | Value recommendation | 1000-5000 | Location | `/etc/salt/master.d/tuning.conf` | Example | `zmq_backlog: 2000` | More information | <https://docs.saltstack.com/en/latest/ref/configuration/master.html#zmq-backlog>, <http://api.zeromq.org/3-0:zmq-getsockopt> (**ZMQ_BACKLOG**)

```
=== swappiness
```

```
[cols="1,1"]
```

| Description | How aggressively the kernel moves unused data from memory to the swap partition. Setting a lower parameter typically reduces swap usage and results in better performance, especially when RAM memory is abundant. | Tune when | RAM increases, or swap is used when RAM memory is sufficient. | Value default | 60 | Value recommendation | 1-60. For 128 GB of RAM, 10 is expected to give good results. | Location | `/etc/sysctl.conf` | Example | `vm.swappiness = 20` | More information | <https://documentation.suse.com/sles/15-SP3/html/>

[SLES-all/cha-tuning-memory.html#cha-tuning-memory-vm](https://www.suse.com/sles-all/cha-tuning-memory.html#cha-tuning-memory-vm)

<p>=== Memory Usage</p> <p>Adjusting some of the parameters listed in this section can result in a higher amount of RAM being used by various components. It is important that the amount of hardware RAM is adequate after any significant change.</p> <p>To determine how RAM is being used, you will need to check each process that consumes it.</p> <p>Operating system:: Stop all SUSE Manager services and inspect the output of <code>free -h</code>.</p> <p>Java-based components:: This includes Taskomatic, Tomcat, and <code>rhn-search</code>. These services support a configurable memory cap.</p> <p>The SUSE Manager Server:: Depends on many factors and can only be estimated. Measure PostgreSQL reserved memory by checking <code>shared_buffers</code>, permanently. You can also multiply <code>work_mem</code> and <code>max_connections</code>, and multiply by three for a worst case estimate of per-query RAM. You will also need to check the operating system buffers and caches, which are used by PostgreSQL to host copies of database data. These often automatically occupy any available RAM.</p> <p>It is important that the SUSE Manager Server has sufficient RAM to accommodate all of these processes, especially OS buffers and caches, to have reasonable PostgreSQL performance. We recommend you keep several gigabytes available at all times, and add more as the database size on disk increases.</p> <p>Whenever the expected amount of memory available for OS buffers and caches changes,</p>	<pre>tee pidstat-memory.log ----</pre> <p>This command will save a copy of displayed data in the <code>pidstat-memory.log</code> file for later analysis.</p> <pre>:leveloffset!</pre> <pre>:leveloffset: +1</pre> <p>= GNU Free Documentation License</p> <p>Copyright © 2000, 2001, 2002 Free Software Foundation, Inc. 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA. Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.</p> <pre>[float] == 0. PREAMBLE</pre> <p>The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.</p> <p>This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.</p>
--	--