SUSE Manager
Under the Hood

Silvio Moioli, Developer, moio@suse.com
Raúl Osuna Sánchez-Infante, Technical Support Engineer, rosuna@suse.com
Good morning, my name is Silvio Moioli and I have been working at SUSE Manager as a developer for the last 6 years. Currently I am coordinating a group of 5 developers in the area of performance and scalability.

I am co-presenting this session with Mr. Raúl Osuna Sánchez-Infante, Technical Support Engineer, as we’ve been collaborating a lot throughout the years on a multitude of issues faced by our customers.
The overall idea of this session is to help you in diagnosing and, in general, understanding your SUSE Manager system (particularly the SUSE Manager Server).

We want to give you some conceptual tools to handle those tasks better.

Let's take a hint from the medical world. What are the tools doctors have at their disposal when they have to diagnose a health issue?
Here is a very basic one: the anatomy atlas – knowledge about the structure of the system to diagnose.

Knowing what parts compose a system and what roles each part plays is fundamental to have a chance to understand it and ultimately cure it.

So this presentation wants to be a SUSE Manager Anatomy 101 course.
I will make extensive use of diagrams to model SUSE Manager, clients, minions and so on. This is my equivalent to Netter’s drawings – it’s the tool I have at my disposal to abstract the inner workings of SUSE Manager in a way that can be described in a few minutes. So I will pass a bit of time in explaining the symbols and caveats.

Then we will go through some very typical use cases: package installations (as an example of an Action) and registration - both from a traditional client and a minion point of view. Here my colleague Raúl will also tell you about typical pitfalls and how to avoid them, for each use case, so that the architectural discussion also stays well grounded in reality.

You are encouraged to interrupt us anytime, but in the end we still have a slot for questions.
We will be presenting a lot of diagrams today, in other words models of how SUSE Manager works internally.

The language used in these diagrams is called “interpreted Petri nets”, and it is a semi-formalism to describe software architecture “qualitatively”. Let's see how that looks like.
This symbol represents a user of some part of the system. Typically a human, in principle it could also be another piece of software via an API.
Those are software components that can interact.

Specifically, there is a communication channel (here represented with the circle) that allows component 1 to send a message to component 2. Component 2 is activated upon reception of a message.

This is a “simulation” of the diagram. The green dot, called a mark, represents a message originating in component 1, being delivered to component 2 and activating it.
In this slightly more complex example, a user activates component 1 which in turn activates component 2.
We model components that are time-triggered as “self-activating components”. Cron and Taskomatic are examples of this pattern.
In some cases, we do not really want to model activations explicitly, we just want to convey the concept that two components freely interact with one another, exchanging messages bidirectionally.

In these diagrams, that’s represented with the double arrow symbol.
All messages we have seen so far were volatile, temporary in nature. Often though we want to model persistent data stores such as filesystems or databases - we do so with the double line circle notation shown here.

The double arrow in this case means the component can both read and write in the persistent storage.

Note how marks “accumulate” in the storage symbol.
“Essentially, all models are wrong, but some are useful.”

—George Box, statistician

Before we get into the core of this presentation, please keep in mind this quote!

While we have taken all the efforts to ensure what we are explaining is correct at some level of abstraction, we are still necessarily simplifying and removing non-essential bits of information here.

Keep also in mind SUSE Manager is a continuously evolving product, and information presented today might be rendered obsolete with newer versions.

Furthermore, unless otherwise noted, all explanations that follow are valid for SUSE Manager 3.2.7 and 4.0 beta 3, and for the latest version of Uyuni.
Let's now take a look at a first concrete case: how does package installation work?

We have two parts: first the Action has to be scheduled in SUSE Manager, then it has to be performed.

Let's first concentrate on the scheduling part, assuming it is done via the Web UI.
This is how Actions get scheduled by users via browsers.

Users will interact with browsers, which communicate with our HTTP server via the https protocol. The Apache httpd server forwards most of those requests to Apache Tomcat, which is a container of Java Web applications. Note that some requests are served directly from the filesystem though - that’s the case for so-called “static resources” that do not change. Those requests that get served by Apache Tomcat are redirected to SUSE Manager-specific Servlets which actually implement saving the Action schedule in the database.

When troubleshooting relevant log files are: /var/log/apache2/error_log, /var/log/rhn/rhn_web_ui.log, /var/log/tomcat/*, /var/lib/pgsql/data/log/*. Alternatively the supportconfig tool will create an archive with all files in one place (note that all SUSE Manager specific logs are in a sub-archive called spacewalk-debug).

How about spacecmd, or in general Action scheduling via some script that interacts with SUSE Manager’s public API?
The diagram does not change significantly.

`spacecmd` is still communicating over https, only conveying XML messages instead of HTML, Javascript and CSS. `httpd` still forwards requests to `Tomcat`.


As this diagram represents the backbone of user interaction with SUSE Manager, we will use it again in many later diagrams. Let's introduce a more compact version to be used later.
Here the macro-component “UI” summarizes the components and interactions described so far.

In the animation, we see an extremely simplified model of an Action scheduling.

Now that the package installation Action is scheduled and stored in the database, how does it get executed?
This is what happens in the default case - the rhnsd daemon triggers the mgr_check command every 4 hours (randomized).

What mgr_check does is to query data on the server via a private XMLRPC API (not the publicly available one you can use in scripts, or consumed by spacecmd). This API is implemented with a Python service behind httpd on the server. This API will basically tell mgr_check about any Actions pending for the client at the current date and time, together with their details. Data about those Actions ultimately comes from the database, which was “written” by the user as per the previous slides.

In this case, our package installation Action is fetched on the client, and now mgr_check has to execute it. Execution complicates our diagram a bit.
In this slide two parts of the diagram have changed:

- On the left side, the Python API server now also serves packages. In the case of a SLE client, that means RPM files
- On the right side, `mgr_check` now calls `zypper` which in turn uses a plugin to download packages. The plugin reuses the XMLRPC channel and API as a way to access data on the SUSE Manager Server

Animation shows the overall flow - first `mgr_check` is triggered by `rhnsd`, then it receives Action information from the database through various layers as per the previous slides. Then it invokes `zypper`, which delegated package retrieval to the plugin, which again uses the XMLRPC “channel” to fetch the RPM (this time from the filesystem). Once the plugin is done RPM files are ready for `zypper` to actually install, and when it's done, `mgr_check` reports success back to the Server.

This diagram does not change substantially in case `yum` is used instead of `zypper` - other than of course the implementation details of its plugin are different.

When troubleshooting relevant log files are:

- on the client: `/var/log/up2date`, `/var/log/zypper.log`, `/var/log/zypp/*`

Also keep in mind general network requirements:

- network needs to be working in general, firewalls need to have appropriate open ports (check documentation)
- all systems must have a FQDN. Full-circle DNS resolution is also a requirement!
- NTP on all systems is also necessary, certificate checking depends on that

You might have a feeling now…
…that this sounds a lot like a famous 1993 videogame!

Indeed the mechanism may look complicated when seen for the first time, I at least hope this overview helps a bit.

Anyway, let’s make it one bit more complicated now!
Back to seriousness.

What if a 4-hour cycle time is not really acceptable? One idea could be increase the client check-in frequency - but this opens up new problems like having too many clients checking in at the wrong time - actually there are better solutions.

What is really wanted here is a different mechanism that “pushes” Actions from the Server to the client instead of rhnsd, which “pulls” them from the client side. We do have a solution for that, as you might have guessed, it's called OSAD.
What OSAD does is to make use of a chat protocol (XMPP) to “message” client systems. Once the client gets the message, it will immediately run `mgr_check` with the same consequences we have seen.

The animation shows how this works.

Note that we typically leave `rhnsd` also enabled to have a backup way to access the system in case `osad` stops working for whatever reason. Since the cycle time is 4 hours, not so many check-ins will be generated.

When troubleshooting relevant log files are:

- **on the Server:** `/var/log/apache2/error_log`, `/var/lib/pgsql/data/log/*`, `/var/log/rhn/rhn_server_xmlrpc.log`, `/var/log/rhn/osa_dispatcher.log`, `/var/log/messages` (jabberd)
- **on the client:** `/var/log/up2date`, `/var/log/zypper.log`, `/var/log/zypp/*`, `/var/log/osad`

Please also note most of the mechanisms presented so far are valid for all Actions, not just package installations - obviously `mgr_check` will behave differently, but the triggering mechanisms, the communication channels and the software components will be the same.
In fact, most of the Server-side machinery introduced so far operates in a similarly when registration of a new system is performed.

What happens on the client side, though, is pretty different, so let's go through that process in detail.
A few steps need to be performed in advance on a Server before registering a client and here is a list.

- repository synchronization of target OS: SUSE Manager can basically only manage OSs it knows about, so the first step is to sync them
- bootstrap repository is a small repo that contains minimal client-side software to complete the registration (to solve the chicken-and-egg problem: software can only be installed with SUSE Manager, but that can only happen after registration, but registration itself requires software)
- an Activation Key defines minimal configuration to apply immediately after the client is registered. It is here, for example, that one can add osad to the packages to be installed at registration time, thereby activating the OSAD contact method described before
- a bootstrap script is typically created on one Activation Key and implements registration. This script needs to be generated once steps above are done (via the mgr-bootstrap tool)

Note: SUSE Manager updates include bug fixes and improvements to the whole product, including client software. This means that over time, bootstrap repositories will become outdated and need to be refreshed to avoid issues.

Same goes for bootstrap scripts, which depend on bootstrap repos as well.
This is what happens during registration, client and server side.

Please note that certificates and GPG keys are needed in order to use the bootstrap repo, that in turn is required to install client tool packages which contain `rhnsd`, `osad` and other software including the `rhnreg_ks` program that performs the registration against the Server through the same Python API used by `mgr_check`.

Please also note the inventory means basically packages, hardware and virtualization details.

Finally, note that configuring bootstrap repo includes disabling of existing repos.
We are now going through the same use cases if the system is registered through Salt.

First is package installation. We will for now assume that `salt-minion` is already installed and operating correctly on the target system.
Here is how that looks like.

Action scheduling is exactly the same, indeed one can very well use the exact same Web UI without really realizing the difference between a minion and a traditional client.

Execution is triggered by those components and, architecturally, resembles OSAD in the sense that Salt also operates in “push” mode - you can see similarities with the diagram before.

Differences start to get bigger when we add package downloading to the picture.
Note the added part - access to packages is now via plain https. In order to secure accesses, we use a cryptographic token which is stored in the database.

Tomcat aids httpd in checking this access token by reading relevant data from the database and comparing it with its content, thereby deciding whether the request for a certain package is legitimate or not.

Assuming access is granted, the RPM file is transferred via http and ultimately installed by the package manager.

What happens once the package installation is finished is more complex, and requires a new diagram.
Here is the return path: salt-minion communicates the result to salt-master, which will generate a Salt event which is intercepted by our SUSE Manager-specific Salt engine, mgr_events.

Salt engines are daemons that are started when salt-master is started and stopped when salt-master is stopped, and can communicate with it.

What mgr_engine will do is to write the event to Postgres and notify Tomcat that a new event is ready to be processed. Tomcat then dispatches events across a configurable number of worker threads that will react to the event - in this case, updating the database with the new installed package.

Note that this mechanism is new as to SUSE Manager 3.2.6.

When troubleshooting relevant log files are:

- **on the minion:** /var/log/zypper.log, /var/log/zypp/*, /var/log/salt/minion
How does registration (also called onboarding) work in the case of minions? Many aspects share similarities high-level, although naturally we do have important low-level differences.
Prerequisites are similar and I will not repeat them here. Please note that, like in the traditional client case, having up-to-date bootstrap repos and bootstrap scripts is important. An up-to-date salt-minion version is essential for SUSE Manager to perform optimally.

In the case of minions, there are three ways to perform registration with varying degrees of customizability. We are going to cover them all in the next few slides.
Minion

- Bootstrap script:
  - configures certificates and GPG keys
  - configures the bootstrap repo
  - installs salt-minion → serve bootstrap repo (Apache httpd)
  - sets minion_id, master
  - starts salt-minion → show key for acceptance (salt-master)

Server

First method: via bootstrap script.

Once again we want to install some software and configure it in order to complete registration - in this case there are no real "client tools" but the Salt minion. That means we have to have a repo where to get it from, which implies a bootstrap repo is needed, as well as certificates and GPG keys to make sure packages are delivered securely.

Once salt-minion is installed there are a few configuration changes to be made, the bare minimum being the master and minion_id variables. At that point salt-minion can be started and will contact the salt-master running in the SUSE Manager Server.

Remaining steps (after key acceptance) will be explained later as they are common for all methods.

What changes if we perform the registration from the UI? High-level, remarkably little.
Second method: UI

Note that the `bootstrap` state implements basically the same changes that the bootstrap script does, only in Salt States instead of a shell script. Moreover the triggering is different, as it is UI-initiated, and uses Salt SSH under the hood. One final difference is key acceptance, which is automated in this case.
Third method: manually

In this case the installation of the right version of `salt-minion` and its configuration is totally up to the user.

Server does nothing but wait for the minion start event, and when that happens it will prompt for its key to be accepted.
After key acceptance

- Minion is ready from Salt’s point of view
- Remaining steps:
  - Inventory: two Actions are scheduled (software and hardware)
  - Application of Activation Key and other configuration: several states
  - Errata cache refresh

In all cases (but the UI) key will need to be accepted. Usually it is recommended to do so manually for security reasons, but that can also be automated so that keys are automatically accepted upon receiving.

Once the key is accepted from a “pure Salt” point of view the minion is ready and fully functional.

SUSE Manager will at this point make use of Salt to perform some more tasks, mainly:
- configure the minion (according to its Activation Key: repos, package installation, etc.)
- update the database with inventory data
This is a feature completely implemented on the server side, so it does not matter whether a system is a minion or a traditional client.

What this feature is about is checking systems’ status against a certain CVE number. Possible outcomes are:

- not affected (there are no installed packages that are vulnerable to the requested number)
- patched (system is not vulnerable any more - an update fixing the security bug had already been applied)
- affected, patch available (system is vulnerable, has an update ready in one of its assigned channels)
- affected, patch not available (system is vulnerable, has an update ready in one channel which is not assigned)
Information that the user gets out of the CVE Audit feature, regardless if it’s via the UI or API, always comes out of the database, which knows about all systems, packages, channels and their assignments.

This information comes mainly from three sources - traditional client reports, minion events and repository synchronization (spacewalk-repo-sync, mgr-sync, Taskomatic jobs, etc.).

Crucially, all this information has to be pre-elaborated in order for the CVE audit to complete within reasonable runtime. This precomputation happens in Taskomatic’s cve-server-audit, nightly by default. Thus, data changes within the last 24 hours might be out-of-date.

When troubleshooting relevant log file is Taskomatic’s /var/log/rhn/rhn_taskomatic_daemon.log.
Back-up question: what are the requirements for http proxies on the Server?
Answer: proxy needs to be set both on the OS level (via the PROXY environment variable) and as a SUSE Manager setting (available in the UI: Admin → Setup Wizard)
Thanks for your attention!