

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

---

Corso di Laurea Triennale in Informatica

# **Analisi e sviluppo di un sistema di monitoraggio per sistemi cloud ed infrastrutture dinamiche**

Relatore:

Prof. Fabio Panzieri

Correlatore:

Dott. Andrea Ceccanti

Laureando:

Stefano Bovina

---

Sessione II - Anno Accademico 2016/2017

*...Alla mia famiglia e ad Elisa*



# Indice

<b>Introduzione</b>	<b>v</b>
<b>1 Un nuovo sistema di monitoraggio per il CNAF</b>	<b>1</b>
1.1 Il problema del monitoraggio . . . . .	1
1.2 Il progetto Bebob . . . . .	5
<b>2 Monitoraggio e Allarmistica</b>	<b>9</b>
2.1 Stato dell'arte . . . . .	9
2.1.1 Nagios . . . . .	10
2.1.2 Lemon . . . . .	14
2.2 Problemi dell'infrastruttura attuale . . . . .	16
2.3 Possibili Alternative . . . . .	18
2.3.1 Zabbix . . . . .	18
2.3.2 Collectd . . . . .	20
2.3.3 Time-series database . . . . .	21
2.3.4 Sensu . . . . .	23
2.4 La scelta . . . . .	24
<b>3 Implementazione</b>	<b>25</b>
3.1 Architettura . . . . .	25
3.2 Sensu . . . . .	27

---

3.2.1	Sensu Workflow . . . . .	29
3.2.2	Client . . . . .	30
3.2.3	Check . . . . .	32
3.2.4	Handler . . . . .	35
3.2.5	Filter . . . . .	37
3.3	Redis . . . . .	43
3.3.1	Configurazione . . . . .	43
3.3.2	Redis-HA . . . . .	45
3.4	RabbitMQ . . . . .	48
3.4.1	Configurazione . . . . .	51
3.5	HAProxy . . . . .	53
3.5.1	RabbitMQ . . . . .	54
3.5.2	Redis . . . . .	54
3.6	InfluxDB . . . . .	56
3.7	Dashboards . . . . .	60
3.7.1	Uchiwa . . . . .	61
3.7.2	Grafana . . . . .	63
3.8	Risultati raggiunti . . . . .	65
<b>4</b>	<b>Conclusioni e sviluppi futuri</b>	<b>67</b>
<b>A</b>	<b>Puppet</b>	<b>69</b>
A.0.1	Come funziona . . . . .	70
<b>B</b>	<b>Foreman</b>	<b>73</b>
	<b>Bibliografia</b>	<b>75</b>
	<b>Elenco delle figure</b>	<b>77</b>





# Introduzione

Questa tesi ha come obiettivo quello di descrivere il progetto e l'implementazione di un nuovo sistema centralizzato di monitoraggio per il CNAF, il centro di calcolo principale dell'INFN (Istituto Nazionale di Fisica Nucleare), all'interno del quale sono presenti più di 1500 sistemi da monitorare tra server e apparati di rete, oltre che ad una serie di servizi gestiti da diversi reparti indipendenti tra loro.

L'attività di monitoraggio è una attività fondamentale, in quanto permette di monitorare lo stato dei sistemi e tenerne traccia nel tempo, permettendo l'individuazione, la prevenzione e la risoluzione dei problemi in tempi brevi, garantendo una migliore operatività del centro.

L'aumento costante delle risorse, dei sistemi da monitorare e l'adozione di infrastrutture cloud, ha reso necessario la rivisitazione dell'attuale sistema di monitoraggio per renderlo più efficiente e sostenibile.

Per ottenere tale risultato, si è cercato di centralizzare il sistema di monitoraggio del centro, utilizzando tool di gestione automatizzata delle configurazioni e sistemi di classificazione dei server, così da razionalizzare le risorse e ridurre il tempo necessario alla sua manutenzione, con l'obiettivo di ottenere un setup altamente affidabile.

Il mio contributo principale, descritto in questa tesi, e svolto in stretta collaborazione con i dott. Misurelli e Michelotto, è consistito nella valutazione delle tecnologie da utilizzare, nella definizione dell'architettura, nel setup dei servizi centrali del nuovo sistema e nella migrazione del monitoraggio dei servizi gestiti del mio reparto (Sistema Informativo) al nuovo sistema. Oltre a queste attività, sono stato impegnato nella presentazione di un seminario per presentare il nuovo sistema ai colleghi e nel fornire supporto e assistenza durante la migrazione dei servizi degli altri reparti.



In questa tesi verranno approfonditi i concetti di monitoraggio e allarmistica, descrivendo la situazione attuale dei sistemi, le motivazioni che hanno portato ad analizzare nuove tecnologie per questa tipologia di attività, in relazione ai cambi di approccio nella gestione della infrastruttura e all'avanzamento delle tecnologie per la gestione di sistemi e servizi.

I requisiti principali per il progetto di aggiornamento del sistema di monitoraggio sono:

- Retrocompatibilità con i sistemi esistenti;
- Minimizzazione dello sforzo e del tempo richiesto per la migrazione tra i due sistemi;
- Centralizzazione del sistema di monitoraggio;
- Condivisione delle conoscenze necessarie nei vari reparti;
- Contenimento dei costi;
- Utilizzo di tool open source con eventuale supporto commerciale.

Nei prossimi capitoli verranno quindi descritte le varie fasi del lavoro di progettazione e implementazione del nuovo sistema di monitoraggio, concentrandosi su:

- Analisi dei punti di forza e debolezze del sistema attuale;
- Individuazione delle possibili alternative;
- Test e analisi dei punti di forza e debolezze dei sistemi individuati;
- Disegno della architettura;
- Implementazione della architettura e test di affidabilità;
- Migrazione alla nuova infrastruttura.

Nella parte finale del testo verrà poi descritta in maniera dettagliata l'architettura scelta, entrando nel merito di ogni singolo componente, motivando le scelte implementative e le varie configurazioni fatte.

# Capitolo 1

## Un nuovo sistema di monitoraggio per il CNAF

In questo capitolo verranno illustrati gli obiettivi dell'implementazione di un sistema di monitoraggio unico per il CNAF. Verranno inoltre illustrati i requisiti relativi alla implementazione, necessari per la sostituzione del sistema esistente.

### 1.1 Il problema del monitoraggio

Il *CNAF* [1] è il centro nazionale dell'INFN (Istituto Nazionale di Fisica Nucleare) per la ricerca e lo sviluppo nelle tecnologie informatiche e telematiche. In qualità di centro di calcolo principale dell'INFN, il CNAF si occupa della gestione e dello sviluppo dei principali servizi essenziali per le attività di ricerca dell'ente, oltre che a perseguire la ricerca e lo sviluppo nel campo ICT nel contesto degli obiettivi dell'INFN.

Fin dalla creazione del sistema di calcolo distribuito su scala geografica noto come "Grid", il CNAF si occupa della gestione e dello sviluppo del middleware e dell'infrastruttura Grid all'interno del consorzio internazionale WLCG (*Worldwide LHC Computing Grid* [23]).

Dal 2003, il CNAF ospita il Tier-1 (fig. 1.1) italiano per gli esperimenti di fisica delle Alte Energie del Large Hadron Collider di Ginevra, fornendo risorse, supporto e servizi necessari alle attività di storage, distribuzione, processamento e analisi dei dati. Inoltre, il CNAF rappresenta una importante computing facility per molti altri esperimenti, principalmente di astrofisica delle particelle e di fisica del neutrino, e uno dei principali centri di calcolo distribuito in Italia.



Figura 1.1: CNAF datacenter

Oltre alle attività indicate, il centro ha le seguenti mansioni:

- gestione dei servizi ICT nazionali dell'INFN (posta, DNS ecc);
- gestione del sistema informativo dell'INFN (gestione del cartellino, sistema contabile, gestione delle risorse umane, sistema stipendiale ecc);
- condurre attività di ricerca, sviluppo e scouting tecnologico a supporto delle attività di ricerca dell'INFN;
- contribuire alle attività di trasferimento tecnologico dell'INFN.

Le attività del centro sono strutturate in 7 gruppi per un totale di circa 50 persone:

- **Farming:** gestisce le risorse di calcolo;
- **Data Storage:** gestisce il sistema di archiviazione di massa;
- **Rete:** gestisce le connessioni LAN e WAN (fig. 1.2) del CNAF;
- **Infrastruttura:** gestisce le facility del centro;
- **SDDS (Software Development and Distributed Systems):** gestisce le attività di R&D nel campo ICT;
- **Sistema Informativo:** gestisce il sistema informativo dell'INFN;
- **Servizi Nazionali:** gestisce i servizi ICT nazionali dell'INFN.

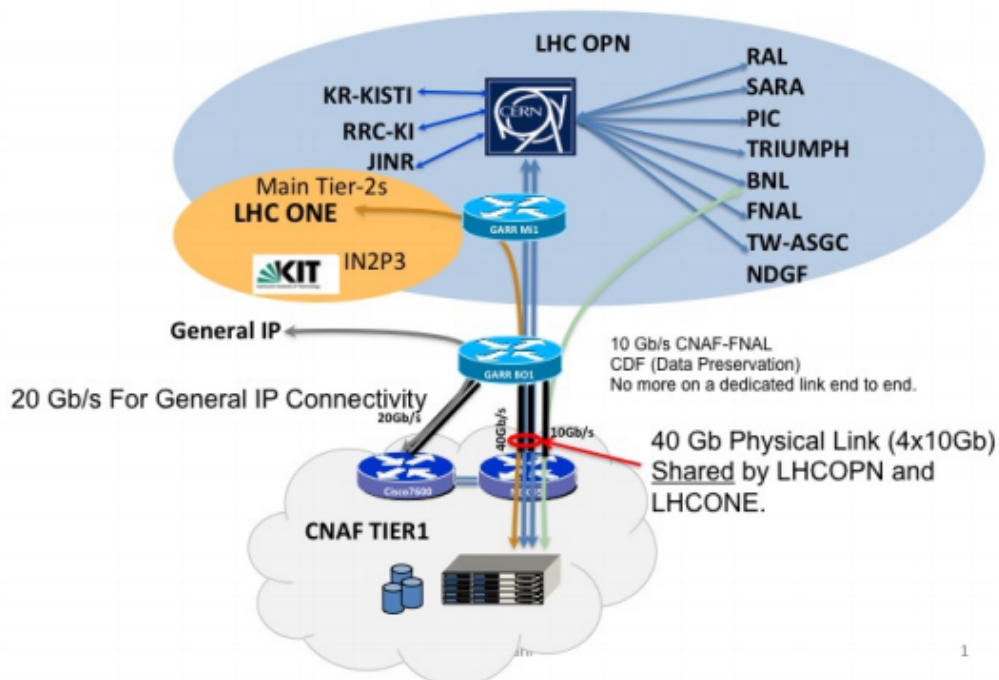


Figura 1.2: CNAF WAN

Accanto a questi gruppi, il gruppo di supporto esperimenti si occupa di aiutare i ricercatori e gli esperimenti a utilizzare le risorse (tabella 1.1) del data center in maniera efficiente.

L'attività di monitoraggio è fondamentale per la corretta operatività del centro in quanto permette di monitorare lo stato dei sistemi e il suo evolversi nel tempo, permettendo così l'individuazione e la risoluzione dei problemi in tempi brevi.

Il monitoraggio dei sistemi, in un centro di calcolo di grandi dimensioni come il CNAF, deve tenere in considerazione sia la grossa quantità di apparati da monitorare (più di 1500 tra server e apparati di rete) sia la necessità di avere approcci di gestione quanto più omogenei possibile per avere procedure condivise per la risoluzione dei problemi, così da poterli risolvere il più rapidamente possibile, cercando dove possibile di prevenirli.

<b>Risorsa</b>	<b>Quantità</b>
<b>Core</b>	22000
<b>Disk storage</b>	22PB
<b>Tape storage</b>	48PB
<b>Racks</b>	> 180 in un'area di 1000 metri quadri
<b>Potenza di calcolo</b>	250 kHS06
<b>Potenza elettrica</b>	5000 kVA

Tabella 1.1: CNAF in numeri

L'attuale approccio di monitoraggio usato nel centro, ha portato alla creazione di un sistema di monitoraggio eterogeneo, con diverse soluzioni adottate dai vari reparti.

Questa situazione ha portato ad avere sistemi di monitoraggio diversi tra loro, creando in alcune situazioni repliche della stessa infrastruttura, difficoltà nell'analisi e riutilizzo dei dati e debolezze nella infrastruttura stessa.

In seguito al costante aumento delle risorse e dei sistemi da monitorare, si è reso necessario rivedere l'attuale sistema di monitoraggio per renderlo

più efficiente e sostenibile, sia in termini di risorse usate che di tempo necessario per la sua manutenzione. Per risolvere questo problema è stato creato il gruppo "Bebop", dall'omonimo stile jazz caratterizzato da tempi molto veloci e da elaborazioni armoniche innovative, formato da persone provenienti da vari reparti del centro, per disegnare ed implementare il nuovo sistema di monitoraggio del CNAF.

## 1.2 Il progetto Bebop

La nascita del progetto per l'implementazione di un nuovo sistema di monitoraggio da utilizzare all'interno del centro, oltre che per i motivi precedentemente indicati, si è resa necessaria per dismettere sistemi legacy (o a fine vita) utilizzati per il monitoraggio, sfruttando tecnologie e tool più avanzati.

Data l'eterogeneità dei sistemi presenti e la necessità di centralizzare il sistema di monitoraggio, abbiamo ritenuto opportuno definire degli standard e discutere le scelte implementative, così da soddisfare le varie esigenze dei reparti e rendere meno complicato il passaggio al nuovo sistema.

Le criticità individuate negli strumenti attualmente utilizzati sono:

- Sono pensati per infrastrutture statiche e progettati, in alcuni casi, come sistemi monolitici con difficoltà nello scalare orizzontalmente<sup>1</sup>;
- Sulla base del punto precedente, difficilmente riescono a soddisfare a pieno le esigenze del centro, visto il grande numero di risorse da monitorare;
- Interazione con l'utente tramite API assente o limitata;
- User Interface (UI) basata su tecnologie legacy, poco intuitiva e con poche funzionalità;

---

<sup>1</sup>Scalare orizzontalmente implica che in caso di necessità, il sistema può crescere per soddisfare le richieste aggiungendo altre macchine.

- Complessità di accesso ai dati e difficoltà di integrazione con sistemi esterni di raccolta e analisi.

La centralizzazione del sistema di monitoraggio ha richiesto l'analisi delle esigenze dei vari reparti, cercando di rendere tale sistema il più affidabile possibile ed eseguendo il setup e la configurazione solo tramite tool di gestione automatizzata delle configurazioni come *Puppet* [6] (Vedi l'appendice A).

Per ottenere tale risultato abbiamo sfruttato l'esperienza appresa da un progetto analogo del 2015, legato al sistema di gestione del provisioning di macchine (virtuali e fisiche) basato su *Foreman* [5] (Vedi l'appendice B) e *Puppet*.

Nel seguente elenco, i requisiti per il nuovo sistema di monitoraggio del centro:

- **Monitoraggio omogeneo:** ottenere un sistema di monitoraggio centralizzato, utilizzabile da tutti i reparti del CNAF, che sfrutti gli stessi strumenti di raccolta, memorizzazione e visualizzazione dei dati;
- **Facilità di setup:** fornire agli utenti finali procedure facilitate di installazione, apprendimento e migrazione dei sistemi;
- **Facilità di interazione:** ottenere un sistema con il quale sia facile interagire tramite API e dashboard;
- **Riusabilità dei dati:** ottenere un sistema in cui i dati raccolti siano facilmente accessibili e riutilizzabili da tutti i reparti, in quanto memorizzati nella stessa tipologia di database e reperibili con le medesime procedure o API;
- **Facilità di gestione dell'infrastruttura:** ottenere un sistema gestibile tramite il sistema di gestione automatizzata delle configurazioni già utilizzato al CNAF (*Puppet*);
- **Facilità di gestione dei client:** ottenere un sistema tramite il quale sia possibile indicare "cosa monitorare" anche tramite il sistema di gestione automatizzata delle configurazioni e di classificazione dei nodi già utilizzato al CNAF, basato su *Foreman* e *Puppet*, evitando onerosi metodi di discovery o modifiche continue lato server;

- **Multi-piattaforma:** il sistema dovrà supportare, almeno lato client, le seguenti piattaforme: Ubuntu/Debian, RHEL/CentOS e Microsoft Windows, oltre che prevedere la possibilità di interrogare apparati di rete;
- **Dashboard:** ottenere un sistema che ci permetta facilmente la creazione di dashboard, incrociando i dati raccolti, anche tra sorgenti diverse, per avere una panoramica completa dello stato dei sistemi;
- **Scalabilità e separazione:** ottenere un sistema che possa scalare ai numeri del CNAF e allo stesso tempo permetta di monitorare sistemi eterogenei, composti da server, apparati di rete, container, applicazioni e suddiviso tra più di cinque reparti tra loro indipendenti;
- **Retrocompatibilità:** il sistema dovrà essere pienamente retrocompatibile con quello attuale, in modo che l'unico sforzo per l'utente finale debba essere l'installazione e configurazione delle nuove componenti, mantenendo se necessario i sensori attuali;
- **Stesse funzionalità:** il sistema dovrà prevedere le stesse funzionalità dei sistemi attuali come risoluzione automatizzata di un problema in seguito ad un allarme ed invio di notifiche tramite email ed SMS;
- **Open source:** il sistema dovrà utilizzare solo tool open source con un eventuale supporto commerciale, sfruttando dove possibile le funzionalità e i plugin disponibili, proponendo modifiche al codice dei progetti se necessario;
- **Affidabilità:** il sistema dovrà essere altamente affidabile in quanto componente chiave per la corretta operatività del centro.

Sulla base di quanto detto, nel prossimo capitolo andremo ad introdurre i concetti di base relativi al monitoraggio, descrivendo lo stato attuale dei sistemi, evidenziandone i punti deboli e mostrando alcune possibili alternative.





# Capitolo 2

## Monitoraggio e Allarmistica

All'interno di questo capitolo andremo a spiegare cosa intendiamo per monitoraggio e allarmistica. Successivamente verrà introdotta l'architettura attuale ed il suo funzionamento, evidenziandone i punti deboli e mostrando alcune possibili alternative e le relative problematiche.

### 2.1 Stato dell'arte

All'interno di una infrastruttura IT è necessario disporre di sistemi che permettano di identificare e risolvere problemi prima che questi possano portare a conseguenze critiche o alla interruzione dei servizi. Questi sistemi devono permettere di monitorare lo stato di apparati di rete, server e servizi e tenerne traccia nel tempo, così da permettere l'individuazione, la prevenzione e la risoluzione dei problemi in tempi brevi, garantendo una migliore operatività del centro e dei servizi offerti.

Al crescere degli apparati e delle risorse da monitorare, questo tipo di attività è diventata sempre più cruciale per tenere sotto controllo la propria infrastruttura; allo stesso tempo, le persone necessarie a mantenere tali strumenti, il budget ed il tempo a disposizione rimangono pressoché invariati o diminuiscono.

Il monitoraggio consiste di due attività principali:

- **Monitoraggio:** quella attività che permette di tenere traccia nel tempo dello stato dei sistemi, permettendo di raccogliere dati, salvarli in qualche database e graficarli, permettendo così una analisi dettagliata dei sistemi, permettendo di intervenire preventivamente.
- **Allarmistica:** quella attività che permette di cogliere eventuali problemi su un determinato sistema ed eventualmente compiere attività di recovery e notifica tramite sistemi di messaggistica o email;

Lo scopo di entrambe le attività è l'individuazione, la prevenzione e la notifica di eventuali problemi sui sistemi.

Allo stato attuale il centro sfrutta due strumenti per tali scopi: *Nagios* [12] per l'allarmistica e *Lemon* [24] per il monitoraggio, che illustriamo nei paragrafi seguenti.

### 2.1.1 Nagios

Nagios è un sistema di monitoraggio open source molto popolare e utilizzato per monitorare server e risorse di rete che permette di eseguire azioni al verificarsi di un evento, quali la notifica o un qualunque comando custom definito dall'amministratore.

Nagios mette a disposizione le seguenti funzionalità:

- Monitoraggio di servizi di rete (SMTP, POP3, HTTP, NNTP, ICMP, SNMP, FTP, SSH);
- Monitoraggio delle risorse di sistema (cpu, disco ecc);
- Possibilità di estendere il sistema tramite plugin;
- Sistema di notifica tramite SMS, email ecc;
- Interfaccia web per visualizzare lo stato dei sistemi.

Il funzionamento di Nagios è basato su un modello client/server e necessita di un processo server che si occupa di schedulare l'esecuzione dei comandi sui vari client ed eventualmente eseguire azioni al verificarsi di un determinato evento.

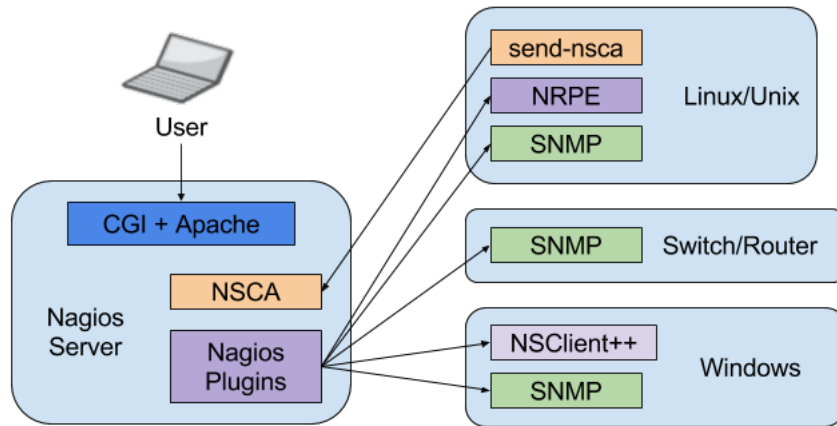


Figura 2.1: Architettura Nagios

Il principio di funzionamento di Nagios è il seguente:

1. Viene schedulata la richiesta di esecuzione di un comando su un client;
2. Il client esegue il comando richiesto;
3. Il comando ritorna un output al server con un relativo exit code.

Il server interpreta l'exit code del check eseguito dal client, determinando in quale stato si trova il client:

Exit code	Stato
0	OK
1	WARNING
2	CRITICAL
altro	UNKNOWN

Tabella 2.1: Tabella degli stati Nagios

Le modalità di esecuzione di un comando su un host sono normalmente di due tipi: NRPE (Nagios Remote Plugin Executor) o NSCA (Nagios Service Check Acceptor).

Nella modalità che utilizza NRPE, la schedulazione dei check viene delegata al server; il plugin NRPE (fig. 2.2) viene utilizzato per l'esecuzione dei comandi sugli host e la comunicazione dei risultati al server.

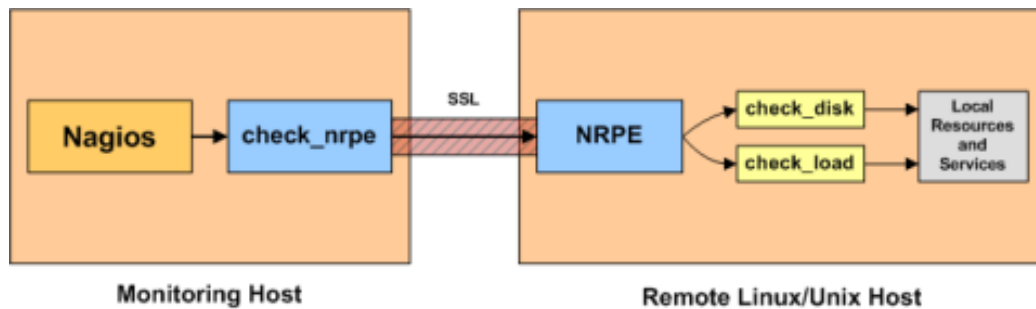


Figura 2.2: Schema di funzionamento del plugin NRPE

Nella modalità NSCA (fig. 2.3), è il client stesso a schedulare l'esecuzione del check, ad esempio per mezzo di un cron job, e successivamente provvede ad inviare i risultati al server.

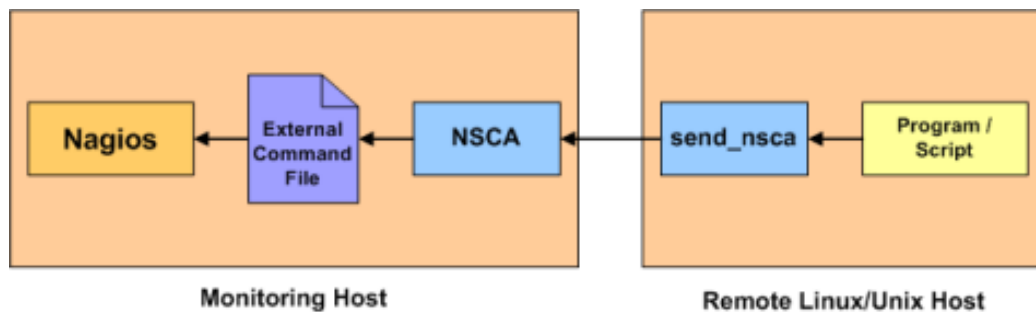


Figura 2.3: Schema di funzionamento del plugin NSCA

I due approcci non sono mutuamente esclusivi e al CNAF attualmente vengono utilizzati entrambi i plugin.

Sul server Nagios troviamo diverse componenti che permettono di configurare come monitorare gli host:

- **hosts**: definisce i server o gli apparati di rete da monitorare;
- **hostgroups**: definisce raggruppamenti logici di host;
- **timeperiod**: definisce archi temporali utilizzati nella esecuzione dei controlli;
- **commands**: definisce un comando da eseguire;
- **contacts**: definisce un contatto a cui inviare un determinato allarme;
- **contactgroups**: definisce gruppi di contatti;
- **service**: definisce quale comando deve essere eseguito su un determinato host o gruppi di host.

Oltre ai plugin citati, la community Nagios mette a disposizione una ampia gamma di plugin ed estensioni che permettono, oltre alla personalizzazione dell'interfaccia grafica, il supporto per diverse realtà di monitoraggio, andando così a coprire la maggior parte delle esigenze comuni.

Il sistema mette a disposizione una interfaccia grafica (fig. 2.4) mediante la quale è possibile accedere alle seguenti funzionalità:

- Visualizzazione dello stato generale dei sistemi;
- Visualizzazione dello stato di un host in maniera dettagliata;
- Visualizzazione dei problemi, divisi per criticità;
- Schedulare manualmente l'esecuzione di un check;
- Silenziare un host o un suo specifico check;
- Visualizzare report riguardanti l'allarmistica.

Service	Status	Last Check Time	Output
PMI_CHECKS	OK	10-11-2016 16:15:12 806d 5h 23m 27s	OK: pmtool version 1.8.14, no sensors, drive ok
MYSQL_CLIENT	OK	10-11-2016 16:15:11 530d 4h 52m 36s	OK: 'mysql'
MYSQL_PYTHON	OK	10-11-2016 16:15:11 530d 4h 52m 29s	OK: 'MySQL-python'
NOT_READY_ONLY_FS	OK	10-11-2016 16:15:00 606d 5h 18m 34s	OK: NOT readyonly file system
PING	OK	10-11-2016 16:15:12 806d 5h 28m 25s	PING OK - Packet loss = 0%, RTT = 0.22 ms
SENMAIL	OK	10-11-2016 16:15:02 806d 5h 18m 33s	OK: process senmail is running
SSH_AUTHENTICATION_TYPE	OK	10-11-2016 16:15:10 606d 5h 18m 28s	OK: option 'PasswordAuthentication no' in /etc/ssh/sshd_config file
SSH_FAILED_ACCESSSES	OK	10-11-2016 16:15:07 148d 10h 18m 28s	OK: 0 failure login attempts(warning at 10, critical at 20)
SSH_SERVICE	OK	10-11-2016 16:15:00 606d 5h 23m 28s	SSH OK - OpenSSH_5.3 (protocol 2.0)
SWAP	OK	10-11-2016 16:15:02 806d 5h 18m 28s	OK: 0% swap space used - 0 on 32287996 (warning at 30%, critical at 60%)
USERS_LOGGED	OK	10-11-2016 16:15:06 606d 5h 18m 31s	OK: logged: 0 users from
evanto-12 CROND_SERVICE	OK	10-11-2016 16:15:03 39d 1h 8m 4s	OK: process crond is running
DATE	OK	10-11-2016 16:15:10 39d 1h 8m 4s	OK: date is 'Tue Oct 11 16:15:10 CEST 2016' (offset is 0)
DISK_SPACE	OK	10-11-2016 16:15:06 39d 1h 8m 4s	OK: [used: 67% 32% 0%] (warning at 95%, critical at 99%)
HOST_TYPE	OK	10-11-2016 16:15:08 39d 1h 8m 4s	OK: Virtual Machine hosted on sysinfo-33.cnaif.infn.it
INFWEB_BACKGROUND_COLOR	OK	10-11-2016 16:15:03 13d 5h 26m 30s	OK: infweb background color is blue
INFO_HOST	OK	10-11-2016 16:15:03 39d 1h 8m 4s	OK: sysinfo-12.cnaif.infn.it - Presenze PPOD - Red Hat Enterprise Linux Server release 5.11 (Tikanga), Kernel: 2.6.18-406.0.0.1.el5, Hardware: x86_64, Processor: x86_64, Platform: x86_64, OS: CentOS Linux, MAC address: 00:15:35:16:01:20
MAIL_SMTP_POSTING	OK	10-11-2016 16:15:03 13d 5h 26m 29s	OK: infweb and screepservlet - postino.cnaif.infn.it/env-entry-value and postino.cnaif.infn.it/env-entry-value
MYSQL_CLIENT	OK	10-11-2016 16:15:12 39d 1h 47m 20s	OK: 'mysql'
MYSQL_PYTHON	OK	10-11-2016 16:15:12 39d 1h 8m 4s	OK: 'MySQL-python'
NOT_READY_ONLY_FS	OK	10-11-2016 16:15:08 39d 1h 8m 4s	OK: NOT readyonly file system
PING	OK	10-11-2016 16:15:13 39d 1h 8m 4s	PING OK - Packet loss = 0%, RTT = 0.32 ms
PROCESS_TOMCAT	OK	10-11-2016 16:15:03 13d 4h 19m 18s	OK: process java is running
SENMAIL	OK	10-11-2016 16:15:03 39d 1h 8m 4s	OK: process senmail is running
SSH_AUTHENTICATION_TYPE	OK	10-11-2016 16:15:01 39d 1h 8m 4s	OK: option 'PasswordAuthentication no' in /etc/ssh/sshd_config file
SSH_FAILED_ACCESSSES	OK	10-11-2016 16:15:10 39d 1h 8m 4s	OK: 0 failure login attempts(warning at 10, critical at 20)
SSH_SERVICE	OK	10-11-2016 16:15:05 8d 4h 47m 25s	SSH OK - OpenSSH_4.3 (protocol 2.0)
SWAP	OK	10-11-2016 16:15:03 39d 1h 8m 4s	OK: 0% swap space used - 0 on 6094840 (warning at 30%, critical at 60%)
SYNFOCERTS	OK	10-11-2016 16:15:11 39d 1h 8m 4s	OK: certificate will expire in 709 days on 2016-09-20 - certificate path is /root/sysinfo/certs/sysinfo-12.cnaif.infn.it.pem, warning days are 19
USERS_LOGGED	OK	10-11-2016 16:15:02 39d 1h 8m 4s	OK: logged: 0 users from 172.16.10.82
evanto-13 CROND_SERVICE	OK	10-11-2016 16:15:03 229d 21h 38m 25s	OK: process crond is running
DATE	OK	10-11-2016 16:15:09 11d 1h 1m 6s	OK: date is 'Tue Oct 11 16:15:09 CEST 2016' (offset is 0)
DISK_SPACE	OK	10-11-2016 16:15:00 229d 21h 28m 28s	OK: [used: 31% 0% 32%] (warning at 95%, critical at 99%)
HOST_TYPE	OK	10-11-2016 16:15:02 229d 21h 28m 28s	OK: Virtual Machine hosted on sysinfo-36.cnaif.infn.it
INFO_HOST	OK	10-11-2016 16:15:02 229d 21h 28m 28s	OK: sysinfo-13.cnaif.infn.it - Vmweb PPOD - CentOS release 6.7 (Final), Kernel: 2.6.32-0505.el6.x86_64, Processor: x86_64, Platform: x86_64, OS: CentOS Linux, MAC address: 00:15:35:16:01:20

Figura 2.4: Nagios UI

## 2.1.2 Lemon

Lemon (The LHC Era Monitoring) è un insieme di componenti (fig. 2.5) che permettono la raccolta, il salvataggio e la visualizzazione di dati di monitoraggio relativi a server, apparati di rete e infrastruttura (es: temperatura) sviluppato al CERN.

Il CNAF, vista la diretta collaborazione con il CERN, ha deciso a suo tempo di adottare lo stesso sistema come tool di monitoraggio per la raccolta di dati relativi alla propria infrastruttura.

Questo tool, oltre alla raccolta e visualizzazione di metriche prevede anche una componente di allarmistica non utilizzata al CNAF.

Lemon si basa su una architettura client/server, dove su ogni client è presente un agente che raccoglie i dati di monitoraggio per inviarli successivamente al server.

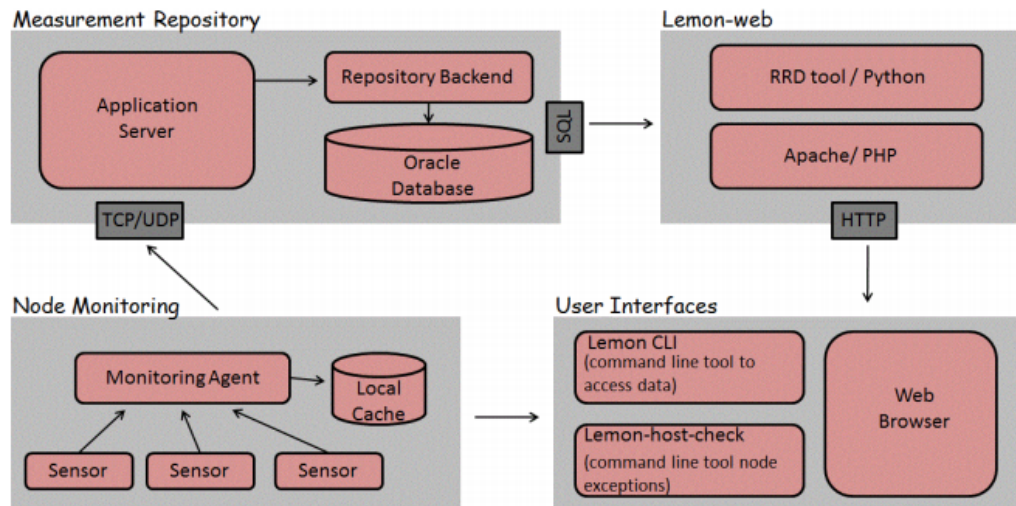


Figura 2.5: Architettura Lemon

Il funzionamento di Lemon è basato sulle seguenti componenti:

- **Agent:** contiene una serie di sensori, recupera e salva i dati di monitoraggio del sistema sul quale gira su una cache locale, per poi inviare i dati al server tramite il protocollo TCP o UDP;
- **Server:** processa i dati ricevuti dagli agent e li salva su un apposito backend, nel nostro caso un database Oracle;
- **Lemon-web:** applicazione web (fig. 2.6) che recupera il dati dal backend, li salva in un Round Robin Database (RRD, [15]) e mostra i grafici relativi ai dati di monitoraggio.



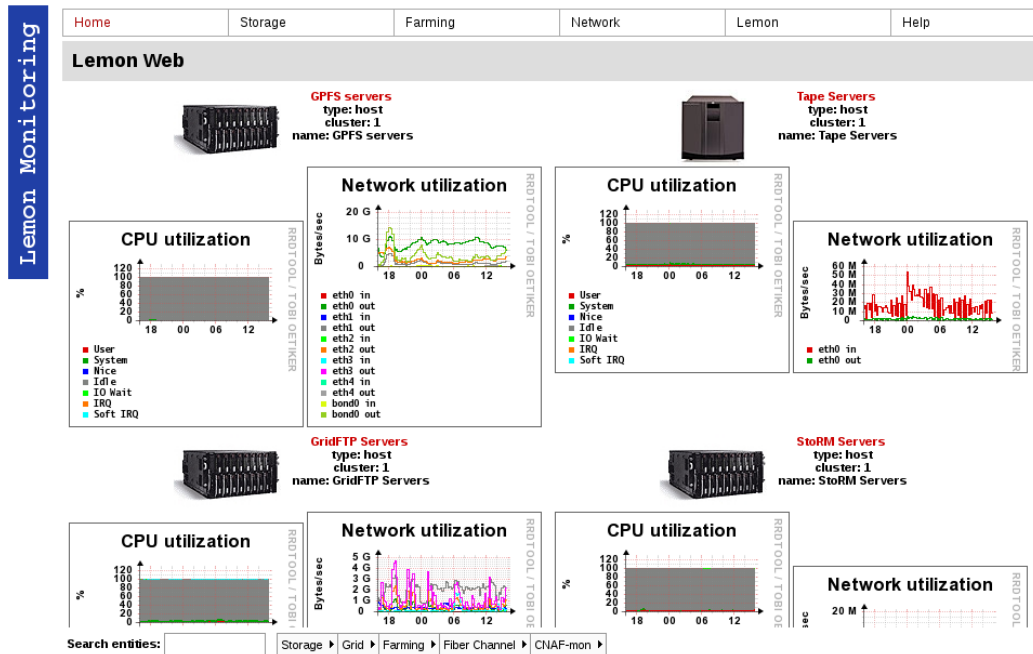


Figura 2.6: Lemon Web

## 2.2 Problemi dell'infrastruttura attuale

Andiamo ora a descrivere i problemi principali dei sistemi in uso e le loro limitazioni:

- **Interfaccia:** L'interfaccia di Nagios, sulla base dell'esperienza fatta, risulta essere vecchio stile basata su Iframe, poco intuitiva e priva di facilitazioni nell'esecuzione di alcuni compiti frequenti, quali il silenziamento dei check e degli host, riesecuzione in batch di check ed eliminazione degli host;
- **API:** Sebbene Nagios, tramite appositi plugin permetta un interfacciamento tramite API RESTful, l'accesso completo ai dati di monitoraggio e allarmistica rimane comunque complicato;
- **Scalabilità:** Sia Nagios che Lemon, nel setup attuale, non sono in grado di scalare ai numeri del centro, soprattutto in previsione della crescita delle risorse nei prossimi anni. Nagios è disegnato per sca-

lare verticalmente<sup>1</sup>, non riuscendo così a supportare pienamente le esigenze del centro. Lemon, nel setup attuale, è basato su un database relazionale che senza le dovute modifiche e i relativi upgrade hardware non è in grado di gestire la quantità di dati prodotto dal sistema di monitoraggio;

- **Persistenza:** Sia Nagios che Lemon non trovano nel canale di comunicazione client-server una qualche tipo di persistenza del dato. In caso di mancata comunicazione con il server, gli eventi vanno persi;
- **Gestione della configurazione:** l'aggiunta di un nuovo host e l'associazione di questo ad un determinato servizio implica una modifica manuale a livello di server e relativo restart. Questo approccio oltre a non essere sostenibile perché aumenta lo sforzo richiesto nel mantenere operativa l'infrastruttura, è abbastanza limitante in quanto non integrabile con i sistemi automatizzati di gestione delle configurazioni e non adatto ad infrastrutture dinamiche e virtualizzate;
- **End of Life (EOL):** Lemon ha raggiunto il suo EOL e non sarà più supportato sui sistemi Red Hat a partire dalla release 7, quindi è necessario sostituirlo il prima possibile.

---

<sup>1</sup>Scalare verticalmente implica che in caso di necessità, il sistema può crescere per soddisfare le richieste aggiungendo risorse alla macchina, come CPU, RAM, ecc.

## 2.3 Possibili Alternative

In seguito alle considerazioni fatte prima, si è deciso di trovare una alternativa per il monitoraggio del centro che possa sostituire almeno Lemon.

In questo paragrafo verranno mostrate alcune alternative ai sistemi attuali che sono state provate nella fase di valutazione delle alternative ai sistemi esistenti.

### 2.3.1 Zabbix

*Zabbix* [13] è una applicazione per il monitoraggio di server e risorse di rete simile a Nagios, del quale è il diretto concorrente.

A differenza di Nagios, Zabbix dispone delle seguenti funzionalità:

- Presenta una interfaccia web non solo read-only, ma una vera interfaccia di configurazione degli allarmi (fig. 2.7);
- Dispone di agent per le piattaforme più usate con all'interno la maggior parte dei controlli;
- Presenta una primitiva interfaccia di analisi dei log;
- Permette nativamente, senza ulteriori aggiunte, la raccolta di metriche e la creazione di grafici.

Sperimentando Zabbix per diverso tempo, abbiamo riscontrato i seguenti problemi:

- La gestione della configurazione, sia lato client che lato server, risulta difficile se non impossibile con tool come Puppet;
- Utilizza un database relazionale, che senza le opportune premure, presenta grossi problemi di performance al crescere degli host e delle metriche raccolte;
- Risulta difficile integrare i controlli esistenti all'interno del sistema senza ulteriore sforzo.

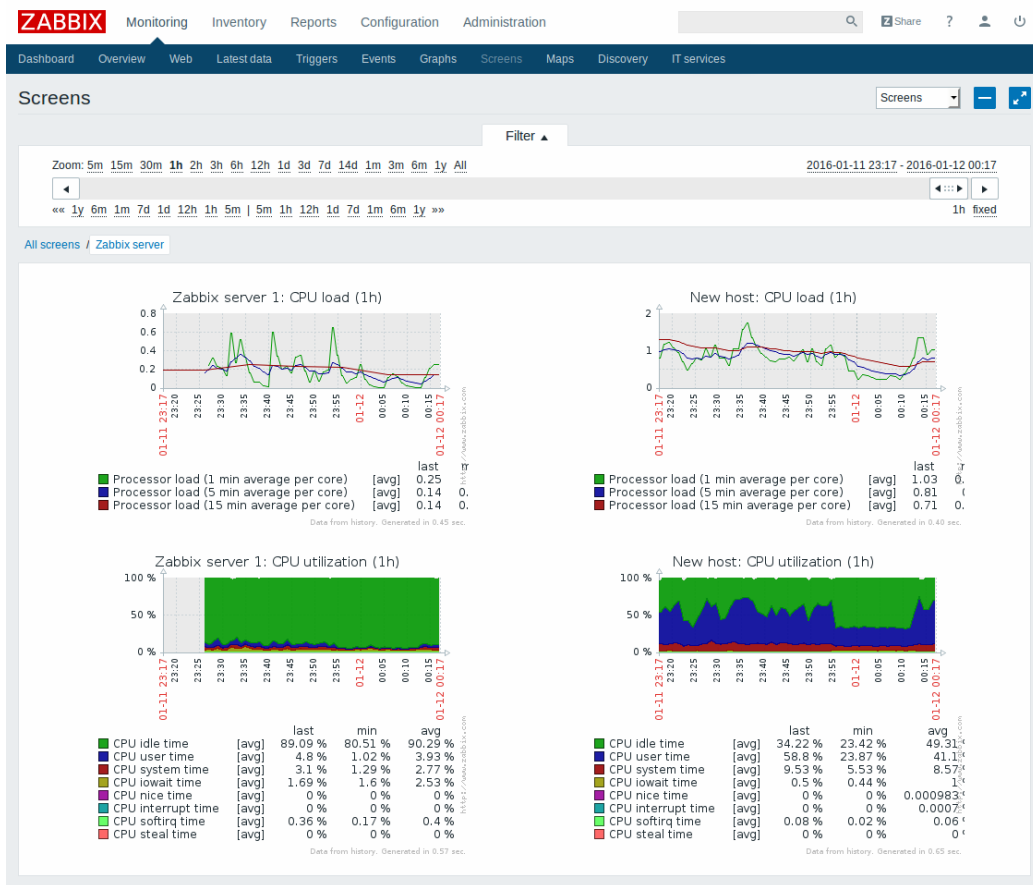


Figura 2.7: Zabbix UI

Abbiamo quindi deciso di scartare questo sistema in favore di altre soluzioni che forniscano maggiore flessibilità e retrocompatibilità con i sistemi esistenti, anche se comunque rimane un buon prodotto "all-in one".

### 2.3.2 Collectd

*Collectd* [22] è un demone che permette la raccolta e il salvataggio di dati di monitoraggio relativi ad un server o un apparato di rete, e quindi potenzialmente un valido sostituto di Lemon.

Oltre alla raccolta, prevede anche la possibilità di visualizzare i dati raccolti tramite una apposita interfaccia (fig. 2.8).

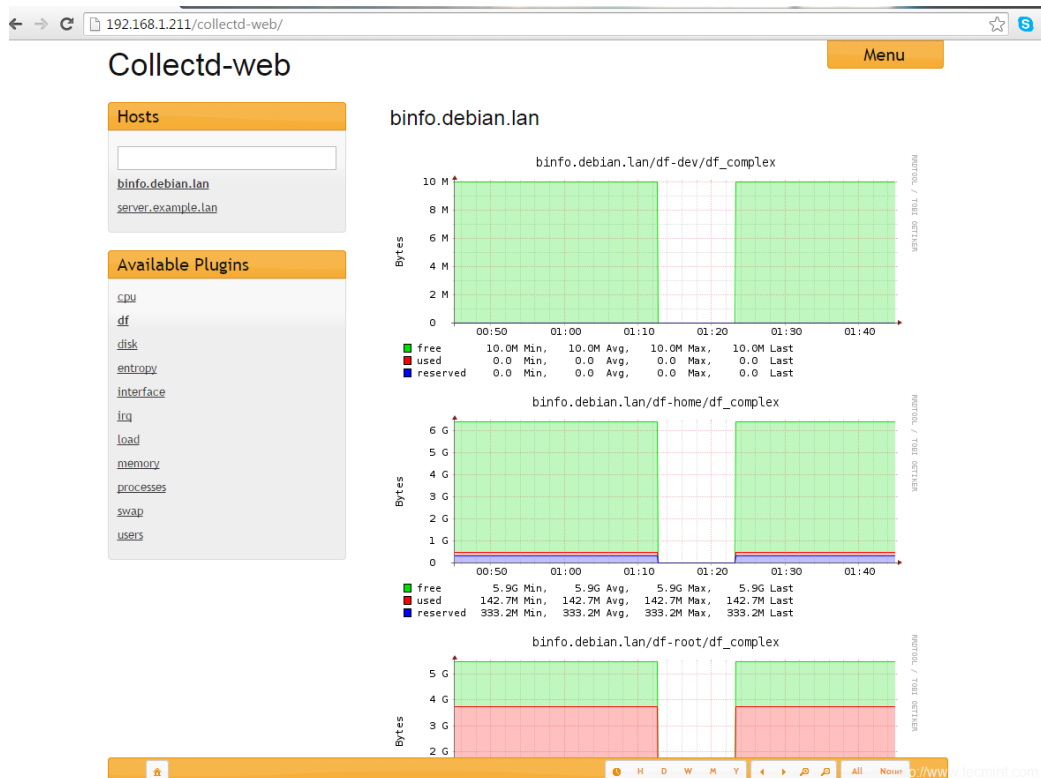


Figura 2.8: Collectd UI

La configurazione di Collectd risulta molto semplice e può essere tranquillamente gestita con i tool di gestione delle configurazioni ed è possibile utilizzare i sensori esistenti:

---

1	LoadPlugin	apache
2	LoadPlugin	cpu
3	LoadPlugin	df
4	LoadPlugin	interface
5	LoadPlugin	load
6	LoadPlugin	memory
7	LoadPlugin	processes
8	LoadPlugin	write_graphite

---

Collectd può essere usato in due modalità:

- Server e client: in questa modalità vi è un demone Collectd su ogni client e uno sul server in ascolto, sul quale vengono inviati i dati e memorizzati su un database di tipo RRD;
- Solo client: in questa modalità il demone sul client deve prevedere la scrittura su un database esterno di qualche tipo, per esempio un time-series database ( 2.3.3) come *Graphite* [16] o *InfluxDB* [17].

### 2.3.3 Time-series database

Un time-series database (TSDB) è un database altamente ottimizzato per gestire dati di tipo time-series, ossia dati organizzati per istante temporale. L'uso più comune dei database di questo tipo è la memorizzazione di dati relativi al monitoraggio delle risorse.

I vantaggi nell'utilizzare questa tipologia di database al posto dei più comuni database relazionali sono i seguenti:

- Alte performance in scrittura e lettura;
- Spazio occupato su disco dai dati molto ridotto;
- Schema free: non esiste uno schema di database, ma è il dato inserito ad autodefinirlo.

- Rollup del dato<sup>2</sup>;

Tra i prodotti open source in grado di gestire questo tipo di dato troviamo InfluxDB, Graphite, *OpenTSDB* [18] ed *Elasticsearch* [20].

Proprietà	InfluxDB	Graphite
Linguaggio	Golang	Python
Byte per punto	2.2	12
Performance in scrittura	350k metriche/sec	50k metriche/sec
Protocolli di input	http, tcp, udp	tcp
Rollup del dato	SI	SI
Supporto commerciale	SI	NO

Tabella 2.2: InfluxDB vs Graphite [21]

Proprietà	OpenTSDB	Elasticsearch
Linguaggio	Java	Java
Byte per punto	12	22
Performance in scrittura	20k metriche/sec	40k metriche/sec
Protocolli di input	http, tcp, udp	http
Rollup del dato	NO	NO
Supporto commerciale	SI	NO

Tabella 2.3: OpenTSDB vs Elasticsearch [21]

<sup>2</sup>Rollup del dato, detto anche downsampling, indica quella operazione che permette di ridurre la granularità delle informazioni con il passare del tempo. Questa procedura (la cui implementazione varia da database a database), permette di mantenere un livello di dettaglio molto alto per un periodo di tempo relativamente breve, per poi diminuire con il passare del tempo, così da ridurre lo spazio occupato su disco e i dati estratti tramite query su archi temporali molto ampi.

Abbiamo scartato openTSDB ed Elasticsearch in quanto per la raccolta di metriche richiedono una architettura troppo complessa per operare (nel caso di openTSDB, si richiede *Hadoop* [2]) e sia le performance, che lo spazio occupato su disco sono peggiori rispetto ad InfluxDB e Graphite, vedi tabella 2.2 e 2.3

### 2.3.4 Sensu

*Sensu* [11] è un framework per il monitoraggio di server e risorse di rete simile a Nagios e Zabbix.

Sensu nativamente può contare sulle seguenti funzionalità:

- Una interfaccia API RESTful;
- Ogni suo componente può scalare orizzontalmente;
- Totalmente retrocompatibile con i probe Nagios e Lemon;
- Open source e con supporto commerciale;
- Permette di monitorare server, container, servizi, applicazioni e dispositivi di rete;
- Pienamente configurabile tramite i sistemi di gestione automatizzata delle configurazioni;
- Non necessita di una configurazione lato server per l'aggiunta di host o servizi, permettendo così la definizione dinamica degli host da monitorare;
- Interfaccia web basata su tecnologie moderne come AngularJS [4] e Go [14].



## 2.4 La scelta

In seguito alla analisi sui possibili sistemi di monitoraggio presenti, la scelta è ricaduta su InfluxDB come database per le metriche e Sensu come tool per gestire la raccolta di metriche e sistema di allarmistica.

Vista la facilità di utilizzo e di migrazione fornita da Sensu, si è deciso fin da subito di migrare sia il sistema di monitoraggio basato su Lemon, sia il sistema di gestione degli allarmi basato su Nagios.

La scelta del database per le metriche è ricaduta su InfluxDB in quanto, oltre a garantire una grande elasticità della struttura dati, permette di sostenere un grande numero di operazioni in scrittura limitando anche l'utilizzo di spazio disco usato, caratteristiche necessarie per monitorare un ampio numero di sistemi limitando i costi.

I motivi che ci hanno fatto scegliere Sensu e che ci hanno permesso di ottenere risultati concreti in breve tempo sono stati i seguenti:

- Piena retrocompatibilità con i probe Nagios e Lemon;
- Sistema totalmente gestibile tramite Puppet;
- Modularità e possibilità di estendere il sistema facilmente;
- Possibilità di scalare orizzontalmente ogni suo componente.

Nel prossimo capitolo andremo ad illustrare la scelta fatta sulla base di quanto detto e illustreremo nel dettaglio l'architettura del sistema di monitoraggio.

# Capitolo 3

## Implementazione

In questo capitolo andremo a descrivere nel dettaglio l'implementazione della infrastruttura di monitoraggio, spiegando il ruolo di ogni suo componente e le relative scelte tecniche.

### 3.1 Architettura

Il setup finale della architettura del sistema di monitoraggio (fig. 3.1) consiste in una centralizzazione delle componenti necessarie al suo funzionamento quali *RabbitMQ* [10], *Redis* [9], *InfluxDB* e *HAProxy* [3], che verranno illustrate nei paragrafi seguenti.

La centralizzazione delle componenti indicate è stata necessaria per poter implementare un setup altamente affidabile e ridurre i costi di gestione e manutenzione della infrastruttura, fornendo all'utilizzatore finale una serie di endpoint relativi a *RabbitMQ* e *InfluxDB* ai quali collegarsi, rendendo totalmente trasparente l'infrastruttura sottostante.

Le componenti non gestite a livello centrale, che rimangono in gestione ai singoli reparti, sono i *Sensu* server e quelle relative alla gestione delle dashboard: *Grafana* [8] e *Uchiwa* [7].

Queste componenti sono quelle relative alla personalizzazione del monitoraggio della propria infrastruttura, ossia la creazione di nuovi check, modifiche al sistema di notifica e creazione di nuove dashboard.

Per garantire la continuità del servizio, essendo componenti chiave del monitoraggio del centro, ognuna di esse oltre ad essere in un setup altamente affidabile, viene tenuta sotto attento monitoraggio e backup.

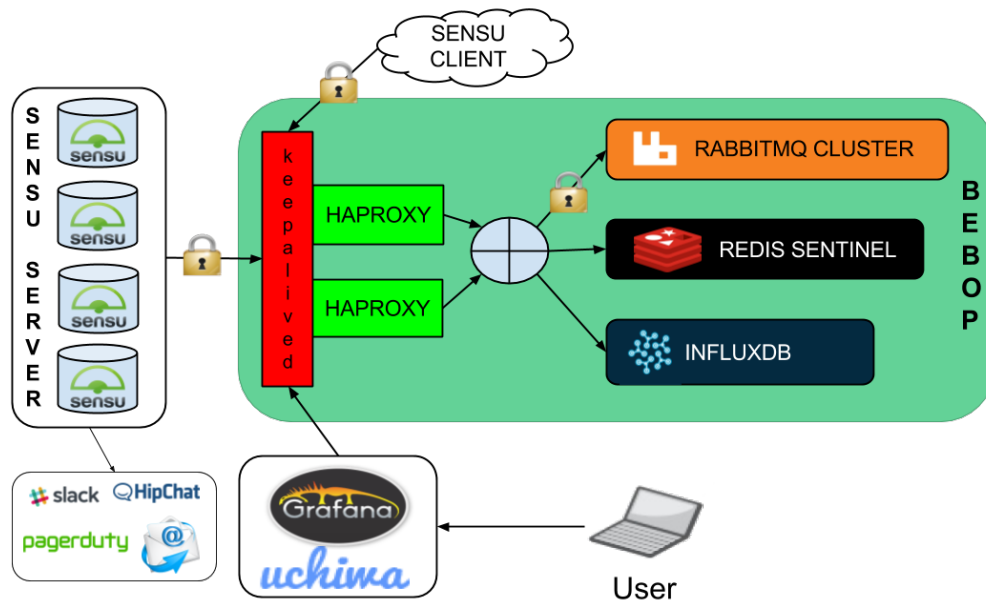


Figura 3.1: Infrastruttura di monitoraggio

Le componenti software necessarie al funzionamento della infrastruttura di monitoraggio sono le seguenti:

- **Sensu server:** il server che gestisce la schedulazione dei check sui client e gestisce gli eventi ad essi collegati;
- **RabbitMQ:** broker di messaggistica utilizzato per la comunicazione asincrona tra Sensu client e Sensu server (vedi paragrafo 3.4);
- **Redis:** database chiave valore usato dai Sensu server per la persistenza dei dati relativi allo stato dei client e dei check;
- **Sensu client:** il client sul quale vengono eseguiti i check;
- **HAProxy:** TCP/HTTP load balancer utilizzato per gestire le connessioni verso i servizi;
- **InfluxDB:** time-series database utilizzato per salvare le metriche di monitoraggio.

## 3.2 Sensu

Sensu è un framework di monitoraggio di server e risorse di rete, nato come alternativa a Nagios e disegnato per scalare orizzontalmente.

Il framework definisce le seguenti entità:

- **Check:** componente generico utilizzato per dichiarare comandi relativi ad allarmi o per reperire metriche sui client;
- **Handler:** gestore di eventi relativi ad allarmi o metriche, come l'invio di notifiche;
- **Filter:** definisce delle condizioni utilizzate per limitare l'esecuzione degli handler;
- **Mutator:** componente utilizzato per modificare l'output di un check (non utilizzato nel nostro caso d'uso).

Tra i servizi presenti sul Sensu server troviamo:

- **Sensu-server**: il demone che si occupa della gestione dei check, degli handler e di tutte le funzionalità gestite a livello server;
- **Sensu-api**: il demone che espone una interfaccia API RESTful di interazione con il server e i dati salvati in Redis.

La configurazione di Sensu, così come i messaggi di comunicazione tra client e server, è basata totalmente sul formato json. Essendo basata su json è possibile specificare, nella configurazione di ogni suo componente, attributi custom che non fanno parte delle specifiche del framework, che sarà poi possibile utilizzare o visualizzare sulla dashboard.

A seguire verranno mostrate le peculiarità dei singoli componenti indicati.

### 3.2.1 Sensus Workflow

Il workflow di esecuzione di un check è riassumibile nei seguenti step:

1. La richiesta di check viene pubblicata dal server su RabbitMQ;
2. Il client esegue i check in base alle proprie subscriptions (vedi paragrafo 3.2.3);
3. Il risultato viene pubblicato su RabbitMQ;
4. Il server processa il risultato dei check e ne salva lo stato su Redis;
5. Il server, in base al risultato dei check, applica uno o più handler;
6. Agli handler vengono applicati eventuali filtri e trasformazioni (mutators);
7. L'evento associato all'handler viene eseguito, se non è stato filtrato (es: per ridurre il numero di notifiche).



Figura 3.2: Sensus workflow

Il caso di un check standalone è un sottoinsieme del workflow appena descritto.

### 3.2.2 Client

Ogni Sensu client deve possedere i seguenti elementi:

- Gli script da eseguire indicati nella definizione del check;
- Configurazione del client;
- Configurazione del trasporto (RabbitMQ).

Configurazione del client:

---

```
1 {
2   "client": {
3     "address": "192.168.95.19",
4     "keepalive": {
5       "handlers": [
6         "slack",
7         "email"
8       ],
9       "refresh": 30,
10      "thresholds": {
11        "critical": 180,
12        "warning": 180
13      }
14    },
15    "name": "test.example.com",
16    "socket": {
17      "port": 3030,
18      "bind": "127.0.0.1"
19    },
20    "safe_mode": false,
21    "subscriptions": [
22      "all",
23      "linux",
24      "linux-cnaf"
25    ],
26    "environment": "dev"
27  }
28 }
```

---

In questo caso "environment" è un attributo custom definito dall'utente per indicare se la macchina in questione è di produzione oppure no, che sarà poi possibile utilizzare in altri contesti.

Nella configurazione del client, sono di particolare importanza gli attributi `keepalive` e `subscriptions`:

- **Keepalive:** il campo nel quale viene indicata la configurazione relativa al controllo di "vita" del client, ossia ogni quanto controllare se il client è "vivo", con i relativi valori di soglia e gli handler collegati (vedi paragrafo 3.2.4)
- **Subscriptions:** il campo che indica le sottoscrizioni del client (che spiegheremo più avanti nel dettaglio); possono essere viste come etichette associate al client.

Il client necessita inoltre della connessione al sistema di trasporto RabbitMQ, in quanto client e server non comunicano direttamente, ma solo tramite scambio di messaggi via RabbitMQ.

```
1 {  
2   "rabbitmq": {  
3     "port": 5671,  
4     "password": "superpassword",  
5     "reconnect_on_error": true,  
6     "host": "mytransport",  
7     "prefetch": 1,  
8     "ssl": {  
9       "cert_chain_file": "/etc/sensu/ssl/cert.pem",  
10      "private_key_file": "/etc/sensu/ssl/key.pem"  
11    },  
12    "user": "prova",  
13    "vhost": "/prova"  
14  }  
15 }
```

L'autenticazione a RabbitMQ viene fatta tramite username e password su un specifico virtual host.



### 3.2.3 Check

Un check è un comando eseguito da un Sensu client che monitora lo stato di un servizio (check standard) o agisce da sensore per una qualche risorsa (metrica) e che comunica al server il risultato della sua esecuzione.

Le specifiche di un check per Sensu sono le stesse di Nagios, ossia:

- Il risultato viene dato su STDOUT o STDERR, dove per i check standard è un messaggio relativo allo stato del controllo

---

```
1 /opt/sensu/embedded/bin/check-load.rb
2 CheckLoad OK: Load average: 0.84, 0.83, 0.75
```

---

mentre per una metrica è l'output della misura fatta

---

```
1 /opt/sensu/embedded/bin/metrics-load.rb
2 test.example.com.load_avg.one 0.86 1476279079
3 test.example.com.load_avg.five 0.84 1476279079
4 test.example.com.load_avg.fifteen 0.75 1476279079
```

---

- Come per Nagios, gli exit code determinano in quale dei seguenti stati si trova il client:

Exit code	Stato
0	OK
1	WARNING
2	CRITICAL
altro	UNKNOWN

Tabella 3.1: Tabella degli stati Sensu

La configurazione di un check standard, come controllo dello stato di un servizio, è così definita:

---

```
1 {
2   "checks": {
3     "check-load": {
4       "refresh": 59,
5       "handlers": [
6         "slack",
```

```
7     "email"
8   ],
9   "low_flap_threshold": 20,
10  "subscribers": [
11    "linux"
12  ],
13  "occurrences": 3,
14  "interval": 60,
15  "timeout": 30,
16  "standalone": false,
17  "high_flap_threshold": 60,
18  "command": "check-load.rb -w 10,7,5 -c 20,15,10 -p"
19 }
20 }
21 }
```

---

La configurazione dei check utilizzati come sensori è così definita:

```
1 {
2   "checks": {
3     "metrics-load": {
4       "command": "metrics-load.rb",
5       "type": "metric",
6       "interval": 300,
7       "standalone": false,
8       "handlers": [
9         "metrics"
10      ],
11      "subscribers": [
12        "linux"
13      ]
14    }
15  }
16 }
```

---

A livello di configurazione, la differenza tra le due definizioni (check generico e metrica) è l'attributo "type" che, se settato a metric, indica che il check in questione deve essere trattato come metrica.

A seconda che il check sia di tipo check o metrica, cambia l'algoritmo tramite cui il server interpreta gli exit code:

- **Check:** il server interpreta un exit code diverso da 0 come un problema, generando un evento che verrà preso in carico dagli handler per una eventuale azione, non facendo niente altrimenti;

- **Metrica:** il server interpreta un exit code diverso da 0 come un problema, non facendo niente. L'evento viene generato solo con exit code uguale a 0 (lo script utilizzato come sensore non è andato in errore)

La schedulazione dei check da parte del server viene fatta utilizzando il sistema di trasporto RabbitMQ tramite il campo subscribers nella definizione del check.

Il campo subscribers applica una label ad un check. I client che sottoscrivono tale label, tramite il campo subscriptions nella configurazione del client, eseguiranno tale check.

In Sensu, come in Nagios, abbiamo la possibilità di indicare chi, tra client e server, debba schedulare l'esecuzione dei check. In questo caso non abbiamo due plugin differenti ma è il campo standalone ad indicare la modalità usata:

- Se il campo standalone è impostato a false, l'esecuzione del check verrà schedulata dal server utilizzando il sistema di trasporto (come descritto prima);
- In caso contrario è il client stesso a schedulare l'esecuzione del check, utilizzando il sistema di trasporto solo per comunicare i risultati al server.

Di fondamentale importanza troviamo anche i campi:

- **handlers:** definisce gli handler che eventualmente dovranno gestire l'evento generato;
- **interval:** indica la frequenza di schedulazione del check.

In aggiunta alla definizione di un check generico mostrata in precedenza, possiamo trovare la seguente definizione, in cui il check viene definito di tipo aggregato (impostando il campo aggregate a true) e disabilitando l'esecuzione degli handler su di esso (impostando il campo handle a false).

---

```
1 {
2   "checks": {
3     "check-ntp": {
4       "refresh": 59,
5       "handlers": [
6         "slack",
7         "email"
8       ],
9       "low_flap_threshold": 20,
10      "subscribers": [
11        "linux"
12      ],
13      "occurrences": 3,
14      "interval": 60,
15      "timeout": 30,
16      "standalone": false,
17      "aggregate": true,
18      "handle": false,
19      "high_flap_threshold": 60,
20      "command": "check-ntp.rb"
21    }
22  }
23 }
```

---

In questo caso il controllo e la eventuale notifica, viene demandato ad un check apposito (normalmente eseguito lato server) che controlla lo stato del check interrogando le API SENSU (es: invio di una notifica se il numero di host in errore sul check è maggiore dell'85%). Questa tipologia di check è utile nel caso in cui si voglia monitorare lo stato complessivo di un check su tutti i sistemi, in cui lo stato di una singola entità risulta poco significativo e potrebbe non essere un problema reale.

### 3.2.4 Handler

Un handler è una azione eseguita dal SENSU server in seguito alla generazione di un evento, come l'invio di una notifica o la richiesta di scrittura di una metrica in un time-series database.

Il framework mette a disposizione quattro tipologie di handler (dove la tipologia pipe è normalmente la più usata):

- **Pipe:** utilizzato per inviare l'output del check ad un altro comando attraverso lo STDIN;

- **TCP/UDP:** utilizzato per inviare l'output del check ad un socket TCP o UDP;
- **Transport:** permette di pubblicare l'output del check sul Sensu transport (RabbitMQ) indicando una apposita coda;
- **Set:** permette di raggruppare più handler sotto un unico identificativo.

Nel nostro caso d'uso le tipologie usate sono la pipe e la set:

```

1 {
2   "handlers": {
3     "slack": {
4       "handlers": [
5         "slack_create",
6         "slack_resolve",
7         "slack_create_prod",
8         "slack_resolve_prod"
9       ],
10      "type": "set",
11      "severities": [
12        "ok",
13        "warning",
14        "critical",
15        "unknown"
16      ],
17      "filters": [
18
19      ],
20      "command": null
21    }
22  }
23 }
24
25 {
26   "handlers": {
27     "slack_create": {
28       "type": "pipe",
29       "severities": [
30         "ok",
31         "warning",
32         "critical",
33         "unknown"
34       ],
35       "filters": [
36         "recurrences_slack",
37         "slack_create",
38         "not_production"

```

```
39     ],
40     "command": "handler-slack-multichannel.rb -j slack_multi",
41   }
42 }
43 }
```

---

Nell'esempio è possibile notare i due tipi di handler, set e pipe.

L'handler slack di tipo set raggruppa altri quattro handler di tipo pipe (l'handler email è analogo): `slack_create`, `slack_resolve`, `slack_create_prod`, `slack_resolve_prod`.

Questi handler, se non limitati in qualche modo, invieranno un totale di quattro notifiche (una per ogni handler) ad ogni evento generato, in quanto eseguiti in parallelo.

Per limitare il flusso di notifiche viene utilizzata la funzionalità di filtro.

Ad ogni handler è quindi associata una lista di filtri, indicati nel campo `filters`. Alla creazione di un evento, vengono richiamati tutti gli handler dichiarati, che prima di essere eseguiti valutano tutti i filtri in AND tra loro.

### 3.2.5 Filter

La funzione di filtro è quella di ispezionare l'output associato ad un evento e determinare se questo debba essere effettivamente gestito dall'handler oppure no utilizzando delle espressioni logiche.

Come detto precedentemente, la funzione principale dei filtri è limitare l'esecuzione degli handler e nel nostro caso permettere la gestione delle notifiche.

Nel nostro esempio, sono presenti due livelli di notifica:

- **Slack:** utilizzato come primo livello di notifica e come raccogliitore generico di tutte le notifiche;
- **Email:** utilizzato come secondo livello di notifica in caso di problema persistente.

Come ulteriore discriminante per le notifiche abbiamo la separazione tra ambienti di produzione e non produzione, necessaria per differenziare

le notifiche e la loro frequenza.

Il sistema separa gli eventi in due tipologie:

- **Create:** indica la creazione di un nuovo evento: avviene quando l'exit code di un check passa da 0 ad un valore diverso da 0 per la prima volta;
- **Resolve:** indica la risoluzione di un evento precedentemente creato (exit code passa da un valore diverso da 0 a 0).

In base a quanto detto, abbiamo bisogno di un filtro per identificare ognuna di queste condizioni, ottenendo così i seguenti filtri:

- **not\_production**

---

```
1  {
2    "filters": {
3      "not_production": {
4        "attributes": {
5          "client": {
6            "environment": "prod"
7          }
8        },
9        "negate": true
10     }
11   }
12 }
```

---

- **production**

---

```
1  {
2    "filters": {
3      "production": {
4        "attributes": {
5          "client": {
6            "environment": "prod"
7          }
8        },
9        "negate": false
10     }
11   }
12 }
```

---

- **recurrences\_slack**

---

```
1  {
2  "filters": {
3    "recurrences_slack": {
4      "negate": false,
5      "attributes": {
6        "occurrences": "eval: value == 10 || value % 1440
7          == 0"
8      }
9    }
10 }
```

---

- `recurrences_slack_prod`

---

```
1  {
2  "filters": {
3    "recurrences_slack_prod": {
4      "negate": false,
5      "attributes": {
6        "occurrences": "eval: value == 3 || value % 1440
7          == 0"
8      }
9    }
10 }
```

---

- `recurrences_slack_resolve`

---

```
1  {
2  "filters": {
3    "recurrences_slack_resolve": {
4      "negate": false,
5      "attributes": {
6        "occurrences": "eval: value >= 10"
7      }
8    }
9  }
10 }
```

---

- `recurrences_slack_resolve_prod`

---

```
1  {
2  "filters": {
3    "recurrences_slack_resolve_prod": {
4      "negate": false,
5      "attributes": {
6        "occurrences": "eval: value >= 3"
7      }
8    }
9  }
10 }
```



```
8     }
9     }
10    }
```

---

- **recurrences\_email**

---

```
1     {
2     "filters": {
3       "recurrences_email": {
4         "negate": false,
5         "attributes": {
6           "occurrences": "eval: value == 1440 || value %
                          1440 == 0"
7         }
8       }
9     }
10    }
```

---

- **recurrences\_email\_prod**

---

```
1     {
2     "filters": {
3       "recurrences_email_prod": {
4         "negate": false,
5         "attributes": {
6           "occurrences": "eval: value == 60 || value % 1440
                          == 0"
7         }
8       }
9     }
10    }
```

---

- **recurrences\_email\_resolve**

---

```
1     {
2     "filters": {
3       "recurrences_email_resolve": {
4         "negate": false,
5         "attributes": {
6           "occurrences": "eval: value >= 1440"
7         }
8       }
9     }
10    }
```

---

- **recurrences\_email\_resolve\_prod**

---

```
1  {
2  "filters": {
3    "recurrences_email_resolve_prod": {
4      "negate": false,
5      "attributes": {
6        "occurrences": "eval: value >= 60"
7      }
8    }
9  }
10 }
```

---

I filtri creati, una volta associati all'handler, ci permettono di gestire una escalation delle notifiche, differenziando la frequenza a seconda dell'ambiente.

Per esempio, in un ambiente di produzione:

- Dopo 3 errori verrà inviata una notifica su Slack;
- Ogni 1440 errori verrà inviato un reminder su Slack;
- Dopo 60 errori verrà inviata una notifica via Email;
- ogni 1440 errori verrà inviato un reminder su Email.

La gestione del routing delle notifiche, è possibile sfruttando i campi custom nel json, esempio:

---

```
1  {
2    "checks": {
3      "check-load": {
4        "refresh": 59,
5        "handlers": [
6          "slack",
7          "email"
8        ],
9        "low_flap_threshold": 20,
10       "subscribers": [
11         "linux"
12       ],
13       "occurrences": 3,
14       "interval": 60,
15       "timeout": 30,
16       "standalone": false,
17       "high_flap_threshold": 60,
```

```
18     "command": "check-load.rb -w 10,7,5 -c 20,15,10 -p",
19     "slack": {
20       "channels": [
21         "#ops"
22       ]
23     },
24   }
25 }
26 }
```

---

In questo esempio, il campo `slack` verrà utilizzato dall'handler per indicare in quale canale della chat, dovrà essere inviata la notifica.

## 3.3 Redis

Redis è un in-memory database di tipo chiave valore, che prevede la possibilità di mantenere la persistenza del dato su disco.

Nella architettura di Senu, Redis viene usato per il salvataggio, da parte dei Senu server, dei dati relativi ai client e dei check

Questo componente, essendo un componente chiave da cui dipende il funzionamento dell'intero sistema, necessita di alcuni accorgimenti legati alla sicurezza ed all'alta affidabilità.

### 3.3.1 Configurazione

Rispetto alla configurazione standard, vanno tenute in considerazione le seguenti tematiche:

- La configurazione base non prevede nessun tipo di autenticazione al sistema;
- Le connessioni al database non prevedono nessun tipo di encryption;
- La configurazione di default prevede l'utilizzo di Redis come cache, senza persistenza del dato a lungo termine;
- Configurazione della memoria da mettere a disposizione del database.

Il problema dell'autenticazione è risolvibile con la seguente configurazione

---

```
1 requirepass superstrongpassword34334587762f??!!
```

---

Con questa configurazione l'accesso al database sarà permesso solo tramite password.

Il problema relativo all'encryption delle connessioni non è stato tenuto in considerazione in quanto pensato per essere contattato solo all'interno della stessa LAN, solo ed esclusivamente dai processi sensu-server o sensu-api e mai dai client.

Il problema legato alla persistenza del dato prevedere due possibili soluzioni a seconda della implementazione scelta:

- RDB
  - **Pro:**
    - \* Backup puntuale su un singolo file;
    - \* Ottimo per backup e disaster recovery essendo un file singolo e compatto;
    - \* In caso di ripristino, questo risulta facile e veloce;
    - \* Non impatta sulle performance di Redis.
  - **Contro:** Possibile perdita di dati se avviene un crash in un determinato istante temporale tra un salvataggio e un altro.
- AOF
  - **Pro:** Garantisce di non perdere dati;
  - **Contro:**
    - \* File di maggiori dimensioni;
    - \* Più complesso il backup;
    - \* Più complesso il ripristino dei dati;
    - \* Impatta sulle performance del sistema.

Sulla base delle considerazioni fatte e basandoci sulle politiche di alta affidabilità che andremo a spiegare meglio nel prossimo paragrafo, abbiamo deciso di abilitare il metodo RDB, con le seguenti impostazioni:

- Abilitazione dello snapshot periodico del dato su disco

---

```
1 save 900 1
2 save 300 10
3 save 60 10000
```

---

La keyword save indica rispettivamente che:

- Viene fatto uno snapshot ogni 900 secondi se almeno 1 chiave cambia;

- Viene fatto uno snapshot ogni 300 secondi se almeno 10 chiavi cambiano;
- Viene fatto uno snapshot ogni 60 secondi se almeno 10000 chiavi cambiano.

- Abilitazione della modalità RDB.

---

```
1 appendonly no
```

---

La gestione della memoria per default prevede la rimozione delle chiavi secondo un algoritmo LRU (Least Recently Used). Questo algoritmo è normalmente usato quando Redis viene utilizzato come cache, mentre nel nostro caso il dato deve essere mantenuto nel tempo.

Per mantenere il dato, la memoria va configurata come segue:

---

```
1 maxmemory 512000kb
2 maxmemory-policy noeviction
```

---

Questa configurazione abilita la policy noeviction, che in caso di saturazione della memoria, ritorna un errore sulle operazioni di scrittura, ma mantiene il dato in memoria. Inoltre, viene impostato un limite massimo di memoria utilizzabile da Redis.

### 3.3.2 Redis-HA

Essendo Redis uno dei componenti chiave del sistema abbiamo bisogno di una soluzione che ci permetta, in caso di problemi su una delle istanze Redis, di non interrompere il servizio e che non preveda interventi umani. La soluzione ufficiale di Redis si chiama Redis Sentinel. Redis Sentinel sfrutta un meccanismo già presente in Redis che consiste nella replica del dato, in una architettura master-slave, con l'aggiunta di un sistema di failover ed elezione di un nuovo master.

Per ottenere un setup master-slave, abbiamo bisogno di configurare almeno un'altra istanza Redis, con i seguenti parametri:

---

```
1 slaveof <masterip> <masterport>
2 masterauth <master-password>
3 slave-read-only yes
```

---

Una volta configurato lo slave, abbiamo bisogno di uno strumento che in caso di fallimento del master, permetta di eleggere come nuovo master una delle istanze slave: Redis Sentinel.

Per un setup che permetta di tollerare un guasto nel caso pessimo, necessitiamo di almeno tre macchine ognuna con il processo Redis sentinel (il numero tre è richiesto per evitare il problema dello split-brain). In caso di problemi sul master, il problema viene individuato dai processi Redis Sentinel. Quando viene raggiunto un quorum in merito alla decisione se il master è accessibile o meno, questi provvedono a riconfigurare Redis e ad eleggere un nuovo master.

Lo schema risultante, è il seguente:

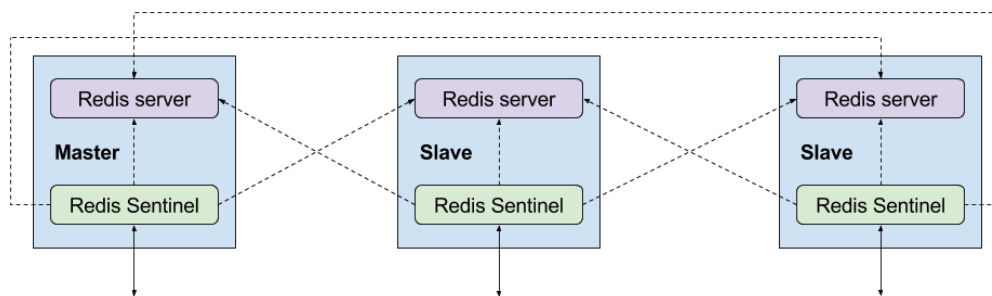


Figura 3.3: Redis Sentinel

Per collegarci ad una istanza e capire se questa è master o meno e monitorare la stato della replica possiamo utilizzare il seguente comando:

```
1 redis-cli -h localhost -p 6381 -a superstrongpassword34334587762
  f??!!! info replication
2 # Replication
3 role:master
4 connected_slaves:2
5 slave 0: ip=192.168.96.11, port=6381, state=online, offset
  =170468795277, lag=0
6 slave 1: ip=192.168.96.12, port=6381, state=online, offset
  =170468790515, lag=0
7 master_repl_offset:170468802516
8 repl_backlog_active:1
9 repl_backlog_size:1048576
10 repl_backlog_first_byte_offset:170467753941
11 repl_backlog_histlen:1048576
```

Per permettere ad un client di scrivere sul database, abbiamo bisogno che esso possa capire quale istanza è la master, onde evitare tentativi di scrittura su una istanza read-only.

Per raggiungere tale obiettivo, possiamo seguire due approcci:

- Dichiarare a livello applicativo tutte le istanze Redis Sentinel, ed interrogarle in modo da poter conoscere quale istanza è in quel momento master;
- Utilizzare HAProxy per identificare l'istanza master.

L'opzione scelta è la seconda, in modo da svincolare l'utente finale da ulteriori configurazioni e gestire l'infrastruttura in maniera più agile, mascherandone le componenti sottostanti. L'implementazione verrà descritta nel paragrafo 3.5



## 3.4 RabbitMQ

RabbitMQ è un broker di messaggistica che implementa il protocollo AMPQ (Advanced Message Queuing Protocol), utilizzato nella architettura di Senu per la comunicazione tra client e server, ed in particolare per permettere al server di schedare l'esecuzione dei check sui client e permettere ai client di comunicare i risultati al server.

In RabbitMQ possiamo identificare due entità distinte:

- **Producer:** colui che produce e pubblica il messaggio;
- **Consumer:** colui che consuma i messaggi pubblicati.

Il metodo di scambio di messaggi più comune utilizza delle semplici code, dove ogni messaggio è consegnato ad un solo consumer in stile FIFO (First In First Out).

Nella architettura Senu descritta nel paragrafo 3.2 sono stati introdotti i concetti di subscribers e subscriptions. Queste parole chiave, utilizzate per eseguire un singolo check su più server, fanno riferimento al paradigma publish/subscribe sfruttato per richiedere l'invio di un singolo messaggio a più consumer.

Questo modello introduce il concetto di exchange: una exchange (fig. 3.4) è un oggetto che accetta i messaggi da un producer e li distribuisce su più code, dalle quali i consumers leggeranno i messaggi.

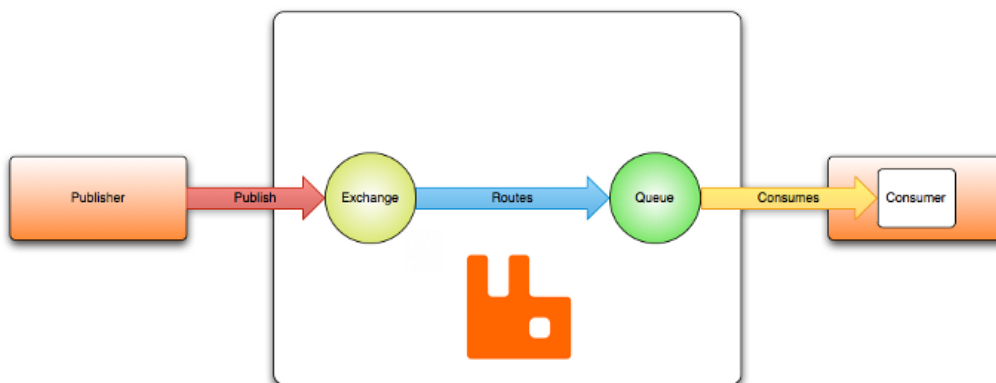


Figura 3.4: RabbitMQ exchange

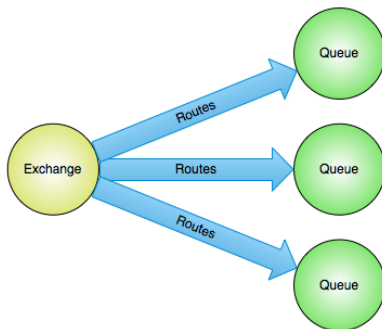


Figura 3.5: Fanout exchange

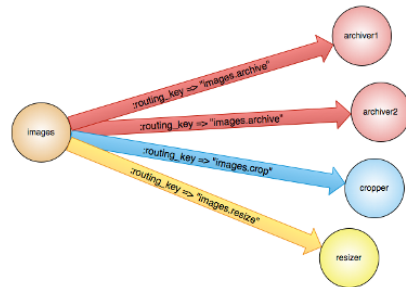


Figura 3.6: Direct exchange

Le exchange utilizzate da Sensu sono di due tipi:

- **Fanout** (fig. 3.5): permette il broadcast di tutti i messaggi in ingresso a tutte le code conosciute dalla exchange;
- **Direct** (fig. 3.6): permette la consegna dei messaggi alle code basandosi sulla routing key associata ai messaggi, ed è normalmente usata per comunicazioni di tipo unicast.

In Sensu le exchange di tipo fanout corrispondono alle subscriptions, in cui ogni exchange è una subscription e le relative code associate sono quelle dei client che ne richiedono il check (fig. 3.7).

Le exchange di tipo direct sono utilizzate solo per gli eventi di keepalive e pubblicazione dei risultati.

## Exchange: httpd

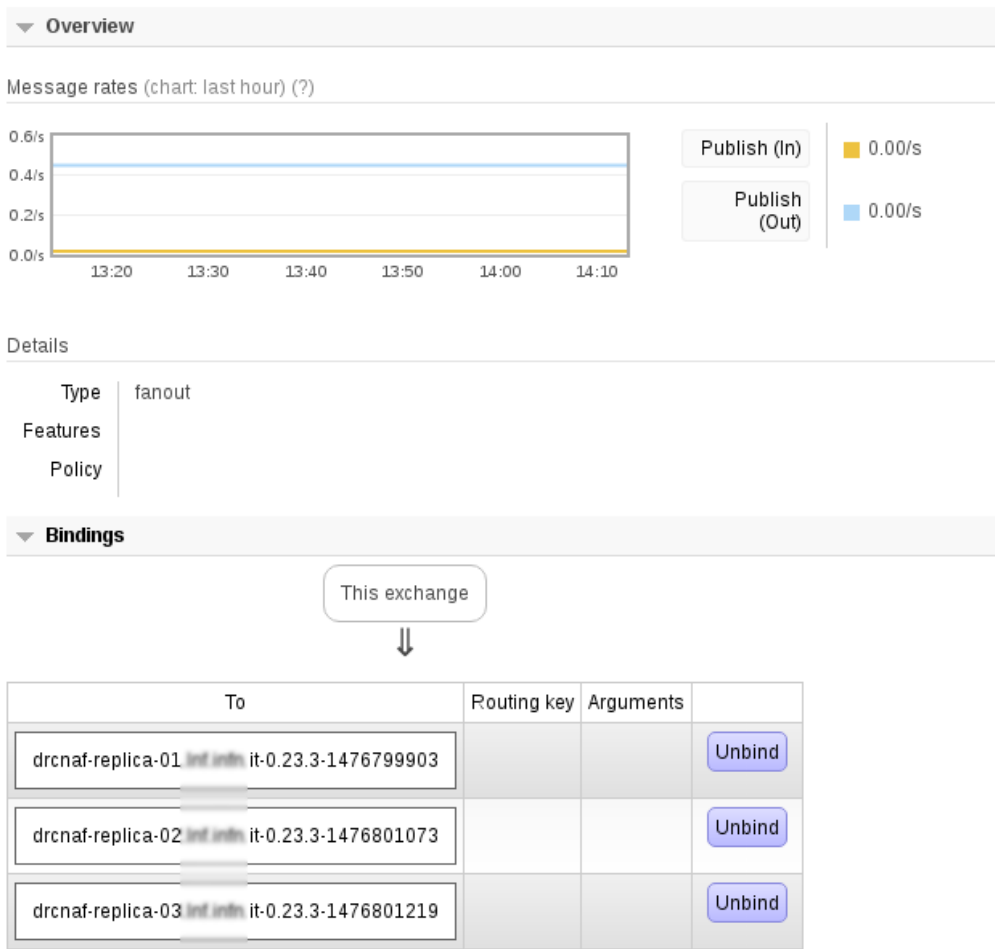


Figura 3.7: Sensu exchange: tre code relative a tre host diversi che stanno sottoscrivendo httpd.

### 3.4.1 Configurazione

Questo componente, essendo l'unico mezzo di comunicazione tra Sensu client e Sensu server, necessita di alcuni accorgimenti legati alla sicurezza ed all'alta affidabilità:

- **Separazione dei contesti:** per ogni utilizzatore del sistema di monitoraggio, abbiamo creato un virtual host ad-hoc

---

```
1 sudo rabbitmqctl add_vhost /sensu
2 sudo rabbitmqctl add_user sensu <strongpassword>
3 sudo rabbitmqctl set_permissions -p /sensu sensu ".*" ".*"
  ".*"
4 sudo rabbitmqctl list_permissions -p /sensu
```

---

- **SSL:** abbiamo abilitato l'SSL per qualunque comunicazione tra Sensu client e Sensu server in quanto RabbitMQ è l'endpoint principale della infrastruttura

---

```
1 {ssl_listeners, [5671]},
2 {ssl_options, [
3     {cacertfile, "/etc/rabbitmq/ssl/cacert.pem"},
4     {certfile, "/etc/rabbitmq/ssl/cert.pem"},
5     {keyfile, "/etc/rabbitmq/ssl/key.pem"},
6     {verify, verify_peer},
7     {fail_if_no_peer_cert, true},
8     {versions, ['tlsv1.1', 'tlsv1.2']}
9 ]},
```

---

- **Guest account:** abbiamo rimosso il guest account
- **Cluster:** abbiamo ritenuto necessario implementare, sia per motivi di carico, che per motivi di alta affidabilità, un setup di tipo cluster, ottenendo un cluster a tre nodi

---

```
1 [{nodes, [{disc, ['rabbit@broker01', 'rabbit@broker02',
2               'rabbit@broker03']}]},
3 {running_nodes, ['rabbit@broker02', 'rabbit@broker03',
4                 'rabbit@broker01']}],
```

---

- **Mirroring:** una volta abilitata l'opzione cluster è stato necessario creare delle high-availability policy in modo da prevedere un mirroring delle code utilizzate da Sensu, così che un eventuale crash non preveda la perdita di dati:

```

1 sudo rabbitmqctl set_policy ha-sensu "^(results$|
   keepalives$)" '{"ha-mode":"all", "ha-sync-mode":"
   automatic"}' -p /sensu
2 sudo rabbitmqctl list_policies -p /sensu

```

- **Admin Plugin:** abbiamo abilitato il plugin di amministrazione (fig. 3.8) per monitorare facilmente lo stato del cluster e di tutte le sue componenti.

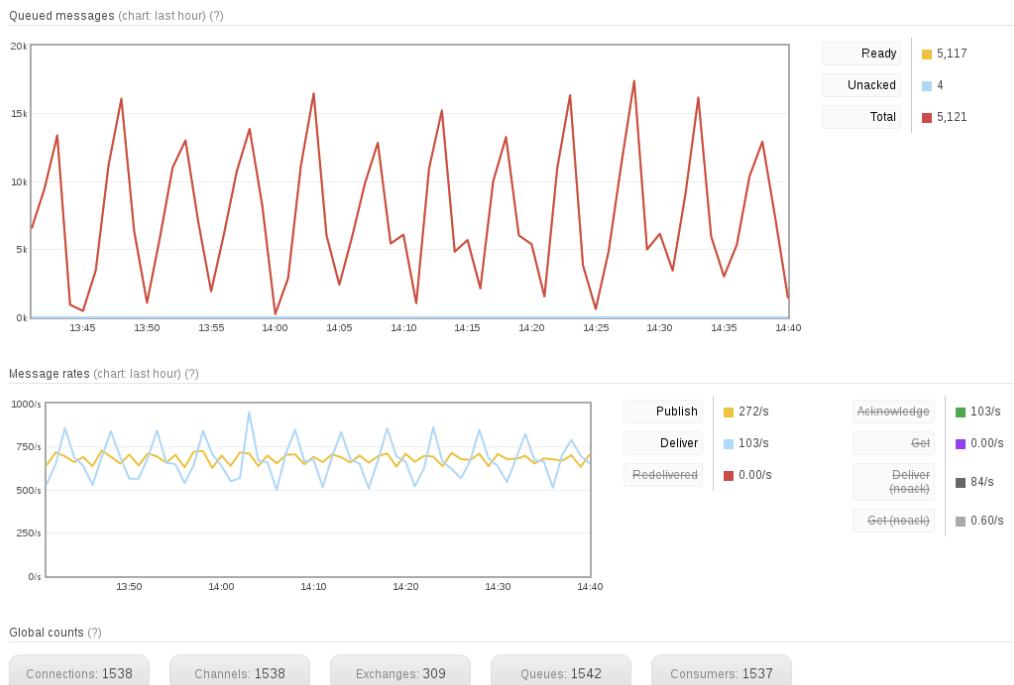


Figura 3.8: RabbitMQ admin plugin

- **Partition handling:** RabbitMQ cluster è un sistema non pensato per lavorare tra WAN diverse, per il quale esistono altre soluzioni (Federation/Shovel); anche lavorando all'interno della stessa LAN abbiamo il problema della network partition. RabbitMQ fornisce alcune policy per la gestione di questi eventi; nel nostro caso abbiamo scelto `pause_minority`. Nella modalità `pause_minority` RabbitMQ metterà automaticamente in pausa i nodi che saranno in minoranza (ossia meno o metà del totale dei nodi). Questo assicura che in caso di network partition, almeno un nodo continui a rispondere.

## 3.5 HAProxy

HAProxy è un TCP/HTTP load balancer open source che permette di gestire lo smistamento delle connessioni tra più server.

Nella nostra architettura è previsto l'utilizzo di due HAProxy assieme alla utility *Keepalived* [19].

Keepalived è una utility che permette di gestire i VIP (virtual IP); tramite una serie di controlli e attraverso il protocollo VRRP<sup>1</sup>, è possibile gestire un failover dei VIP, assicurando che siano presenti sempre su uno e uno solo degli HAProxy, garantendo che in caso di crash di uno dei due, i servizi rimangano raggiungibili.

Ogni servizio relativo alla architettura di monitoraggio, ossia Redis, RabbitMQ ed InfluxDB, viene esposto attraverso un IP dedicato (VIP) e raggiungibile solo ed esclusivamente attraverso HAProxy (fig. 3.9)

Grazie a questo approccio è possibile monitorare i servizi e gestire il carico sui server di backend (ossia quelli che realmente espongono il servizio), oltre a fornire una grande elasticità nella gestione della infrastruttura.

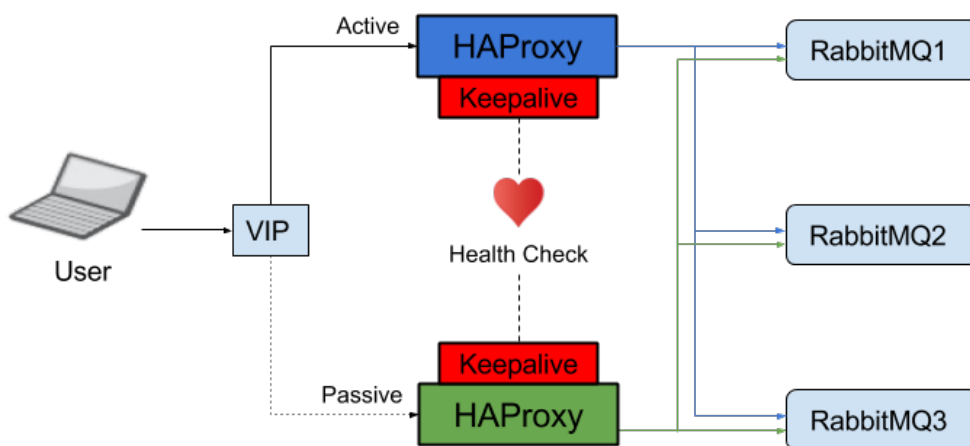


Figura 3.9: Esempio HAProxy RabbitMQ

<sup>1</sup>VRRP (Virtual Router Redundancy Protocol) è un protocollo per le reti di calcolatori definito nella RFC 2338.

### 3.5.1 RabbitMQ

Per il bilanciamento delle connessioni a RabbitMQ, essendo le code in mirroring, possiamo controllare esclusivamente la porta utilizzata per la connessione a RabbitMQ per lo scambio di messaggi, ossia 5671, abilitando anche un controllo dell'SSL tramite l'opzione `ssl-hello-chk`.

Il bilanciamento del carico in questo caso può essere fatto tramite l'algoritmo round robin in quanto è indifferente quale dei backend viene contattato (fig. 3.10)

```

1 listen broker00
2 bind 192.168.97.15:5671
3 balance roundrobin
4 fullconn 8000
5 option tcplog
6 option tcpka
7 option ssl-hello-chk
8 timeout client 3h
9 timeout server 3h
10 server broker01.example.com 192.168.97.16:5671 check
11 server broker02.example.com 192.168.97.17:5671 check
12 server broker03.example.com 192.168.97.18:5671 check

```

broker00		Queue			Session rate			Sessions				Bytes		Denied		Errors		Warnings		Status			
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Req	Redis		
Frontend	1	64	-	1302	1579	8000	1083	035					557 261 589 937	284 798 012 966	0	0	0						OPEN
broker01.example.com	0	0	-	32	459	622	-	361 925		361 925	2s		238 632 957 535	127 096 093 426	0	0	0	0	0	0	0	0	24420h UP
broker02.example.com	0	0	-	3	378	382	-	359 187		359 187	4s		94 663 864 040	74 848 823 505	0	0	0	367	0	2	16417h UP		
broker03.example.com	0	0	-	1 32	465	606	-	361 925		361 925	0s		223 964 768 362	82 853 096 035	0	0	0	0	0	0	0	0	24420h UP
Backend	0	0	1	64	1302	1579	8000	1083 035		1 083 037	0s		557 261 589 937	284 798 012 966	0	0	0	0	367	0	2	24420h UP	

Figura 3.10: HAProxy RabbitMQ

### 3.5.2 Redis

Per la gestione delle connessioni verso Redis, abbiamo bisogno di un controllo ad-hoc per verificare quale delle istanze di backend (ossia quelle che espongono il servizio) sia effettivamente la master, altrimenti un client potrebbe richiedere la scrittura di dati su una istanza read-only e quindi andare in errore.

Per questo tipo di controllo utilizziamo la opzione `tcp-check` che permette di inviare a Redis i comandi per identificare se è master oppure no.

```

1 listen redis_instance00
2   bind 192.168.95.18:6381
3   option tcplog
4   option tcp-check
5   tcp-check send AUTH\ superstrongpassword34334587762f??!!!\ r\n
6   tcp-check expect string +OK
7   tcp-check send PING\r\n
8   tcp-check expect string +PONG
9   tcp-check send info\ replication\r\n
10  tcp-check expect string role:master
11  tcp-check send QUIT\r\n
12  tcp-check expect string +OK
13  timeout client 15s
14  timeout connect 4s
15  timeout server 15s
16  server redis01.example.com 192.168.96.10:6381 check inter 1s
17  server redis02.example.com 192.168.97.11:6381 check inter 1s
18  server redis03.example.com 192.168.98.12:6381 check inter 1s

```

A differenza di quanto mostrato per RabbitMQ, nel caso di Redis avremo due istanze indicate in rosso (fig. 3.11) in quanto read-only (quindi DOWN per HAProxy) e una sola verde (l'unica master)

redis_cnbeebop00		Queue			Session rate			Sessions				Bytes		Denied		Errors		Warnings		Status			
		Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	
Frontend					0	0	-	0	0		8 000	0		0	0	0	0	0	0	0	0	0	OPEN
	command01 redis01	0	0	-	0	0	-	0	0	-	0	0	?	0	0	0	0	0	0	0	0	0	24d20h UP
	command02 redis02	0	0	-	0	0	-	0	0	-	0	0	?	0	0	0	0	0	0	0	0	0	24d20h DOWN
	command03 redis03	0	0	-	0	0	-	0	0	-	0	0	?	0	0	0	0	0	0	0	0	0	24d20h DOWN
Backend		0	0		0	0		0	0		800	0	0	?	0	0	0	0	0	0	0	0	24d20h UP

Figura 3.11: HAProxy Redis



## 3.6 InfluxDB

InfluxDB è un time-series database, ossia un database ottimizzato per gestire dati di tipo time-series.

Un esempio di dato di tipo time-series è il seguente:

---

```
1 load_avg.one 0.86 1476279079
2 load_avg.five 0.84 1476279079
3 load_avg.fifteen 0.75 1476279079
```

---

Nel nostro caso d'uso, i dati vengono inviati ad InfluxDB in due modi distinti:

- **Tramite Sensu:** il dato viene inviato ad InfluxDB attraverso una POST http, sfruttando un handler apposito presente in Sensu e in batch di grosse dimensioni (1000 righe). Questo metodo viene usato per tutte le metriche raccolte dal sistema di monitoraggio stesso, come cpu, disco ecc;
- **Manuale:** utilizzando uno dei client disponibili o tramite una POST http. In questo caso i dati sono inviati in seguito allo scatenarsi di un qualche evento (es: crontab).

Per la scrittura dei dati tramite POST, possiamo utilizzare il seguente comando:

---

```
1 curl -i -XPOST 'http://localhost:8086/write?db=mydb' --data-binary @cpu_data.txt
```

---

Il contenuto del file sono righe, separate dal carattere newline, nel seguente formato:

---

```
1 <measurement>[,<tag_key>=<tag_value>[,<tag_key>=<tag_value>]] <field_key>=<field_value>[,<field_key>=<field_value>] [<timestamp>]
```

---

Esempio di una singola riga (punto):

---

```
1 cpu_load ,datacenter=CNAF,host=myhost.example.com one=0.86 ,five=0.84 ,fifteen=0.75 1476279079
```

---

Il formato descritto sopra, introduce alcune parole chiave, quali measurement e tag:

- **Tag:** una coppia chiave-valore utilizzata per memorizzare metadati. Questo tipo di informazioni sono indicizzate e mantenute in memoria, per rendere efficiente la ricerca sui metadati;
- **Measurement:** struttura dati che descrive il dato salvato nei relativi campi (nell'esempio `cpu_load`);
- **Serie:** la chiave univoca formata da measurement e tag; nell'esempio:

---

```
1 "cpu_load , datacenter=CNAF, host=myhost.example.com"
```

---

Per visualizzare le informazioni inserite, possiamo usare linguaggio simile a SQL, chiamato InfluxQL:

---

```
1 SELECT "one" FROM "cpu_load" WHERE "datacenter" = 'CNAF' limit 5
```

---

Questo linguaggio mette anche a disposizione un ampio set di funzioni (aggregazione, selezione, trasformazione e predizione) come per esempio la media:

---

```
1 SELECT MEAN("one") FROM "cpu_load"
```

---

I dati vengono memorizzati all'interno di database, nel nostro caso separati per utilizzatore del sistema:

---

```
1 create database mydb
2 create user myuser with password 'superpassword'
3 grant all on mydb to myuser
4
5 > show grants for myuser
6 database  privilege
7 mydb     ALL PRIVILEGES
8
9 > show users
10 user  admin
11 myuser  false
```

---

La gestione dei dati memorizzanti viene fatta tramite l'utilizzo dei seguenti strumenti:

- **Retention policy (RP):** una regola che indica quanto tempo il dato memorizzato deve essere mantenuto prima di essere cancellato;
- **Continuous query (CQ):** query che viene eseguita periodicamente in automatico su un database per operazioni di rollout del dato.

Alla creazione del database, se indicato nell'apposito file di configurazione, viene creata una retention policy di default, ma nessuna continuous query.

---

```

1 > show retention policies on mydb
2 name      duration  shardGroupDuration  replicaN  default
3 autogen   0         168h0m0s            1         true
4
5 > show continuous queries
6 name: mydb
7 _____
8 name  query

```

---

La retention policy indicata come default (default a true) indica la retention policy sulla quale verranno scritti i dati se non diversamente indicato.

InfluxDB mette a disposizione un database sul quale vengono memorizzate in automatico statistiche relative al sistema, per permettere di dimensionare il sistema che ospita il database (secondo le linee guida fornite dalla documentazione) e ricavare informazioni sulla mole di dati memorizzati nei database.

Il grande numero di sistemi presenti al CNAF, implica una grossa quantità di dati raccolti, che nel nostra cosa sono i seguenti:

- Cardinalità totale delle serie: 932296;
- Totale punti scritti al secondo: 2312;
- Circa 5GB al giorno di dati.

Per mantenere grosse quantità di dati per lungo tempo, senza incorrere in problemi di performance o di spazio occupato, è consigliabile ridurre la granularità del dato raccolto tramite retention policy e continuous query:

- Creazione retention policy per ospitare il dato dopo ridotto di granularità

---

```

1 CREATE RETENTION POLICY one_week ON mydb DURATION 7d
  REPLICATION 1 DEFAULT
2 CREATE RETENTION POLICY one_month ON mydb DURATION 4w
  REPLICATION 1
3 CREATE RETENTION POLICY six_months ON mydb DURATION 24w
  REPLICATION 1
4 CREATE RETENTION POLICY thirteen_months ON mydb DURATION
  52w REPLICATION 1

```

---

- Creazione delle continuous query che prelevano il dato, ne riducono la granularità e lo salvano nella rispettiva retention policy:

---

```

1 CREATE CONTINUOUS QUERY populate_one_month ON mydb BEGIN
  SELECT mean("value") AS "value", mean("duration") AS "
  duration" INTO "one_month"::MEASUREMENT FROM ./ GROUP
  BY time(15m),* fill(none) END
2 CREATE CONTINUOUS QUERY populate_six_months ON mydb BEGIN
  SELECT mean("value") AS "value", mean("duration") AS "
  duration" INTO "six_months"::MEASUREMENT FROM ./
  GROUP BY time(30m),* fill(none) END
3 CREATE CONTINUOUS QUERY populate_thirteen_months ON mydb
  BEGIN SELECT mean("value") AS "value", mean("duration
  ") AS "duration" INTO "thirteen_months"::MEASUREMENT
  FROM ./ GROUP BY time(1h),* fill(none) END

```

---

La procedura utilizzata per ridurre la granularità delle informazioni permette di mantenere un livello di dettaglio molto alto per un periodo di tempo relativamente breve, per poi diminuire con il passare del tempo, così da ridurre lo spazio occupato su disco e i dati estratti tramite query su archi temporali molto ampi.

Il risultato ottenuto è il seguente:

- Il dato viene memorizzato nella retention policy di default `one-week`, nel quale viene mantenuto alla granularità originale per sette giorni;
- Tramite `continuous query`, il dato viene aggregato e scritto nelle rispettive retention policy:
  - **`one_month`**: il dato viene mantenuto per quattro settimane con una granularità di quindici minuti;
  - **`six_months`**: il dato viene mantenuto per sei mesi con una granularità di trenta minuti;
  - **`thirteen_months`**: il dato viene mantenuto per tredici mesi con una granularità di un'ora.

### 3.7 Dashboards

Per presentare i dati raccolti in maniera chiara all'utente finale, vengono utilizzate delle dashboard che permettono di accedere al dato salvato in InfluxDB e Redis, fornendo all'utente uno strumento per consumare il dato raccolto in maniera semplice ed intuitiva.

Per raggiungere tale obiettivo utilizziamo due strumenti distinti:

- Uchiwa: interfaccia sui dati raccolti da Sensu relativi ai dati di Allarmistica;
- Grafana: sistema per la creazione di dashboard e grafici a partire dai dati relativi alle metriche salvati in InfluxDB.

### 3.7.1 Uchiwa

Uchiwa è la interfaccia grafica (fig. 3.12), scritta in AngularJS e con un backend scritto in Go, che fornisce una panoramica sui sistemi sulla base di quanto raccolto da Sensu.

Uchiwa ottiene lo stato dei sistemi contattando uno o più endpoint delle API fornite da Sensu, permettendo così di reperire i dati salvati su Redis.

Il ruolo principale del sistema è quello di fornire nel modo più intuitivo possibile lo stato dei sistemi, mostrando i client che presentano qualche tipo di problema.

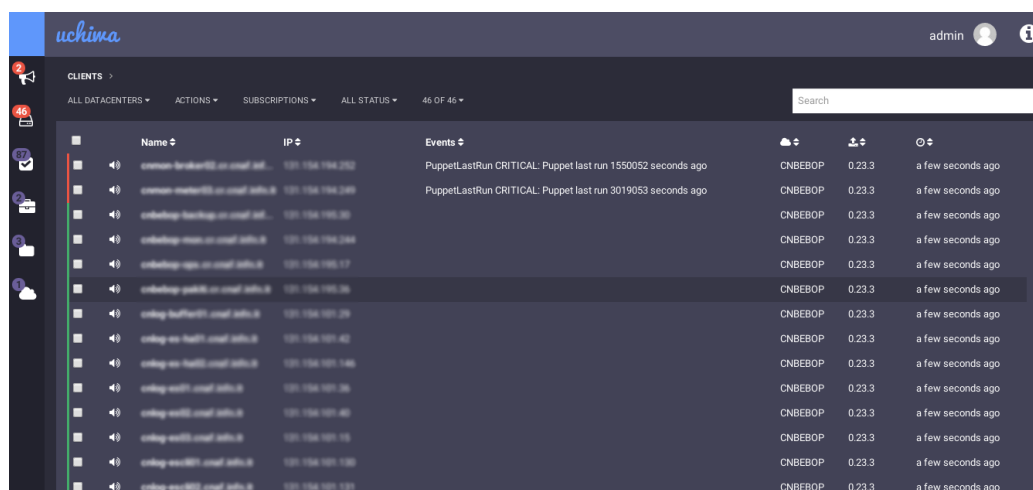


Figura 3.12: Uchiwa UI

Il sistema inoltre permette di:

- Ricercare i client tramite ricerca testuale;
- Ricercare i client tramite le subscription;
- Silenziare un client o un check di un client;
- Rischiodare l'esecuzione di un check su uno o più client;
- Rimozione di uno o più client;
- Visualizzare lo stato dei check aggregati (fig. 3.13);
- Visualizzare in dettaglio lo stato di un client e/o di un check (fig. 3.14).



### 3.7.2 Grafana

Grafana è un dashboard composer system, tramite il quale è possibile generare dashboard partendo dai dati presenti su un database, di solito di tipo time-series.

Grafana permette di generare grafici interrogando diversi tipi di data-source quali InfluxDB, Graphite, openTSDB, Elasticsearch, anche all'interno dello stesso grafico.

Il sistema mette a disposizione un generatore di query interattivo (fig. 3.15), in modo da rendere facile la generazione di grafici per l'utente finale.

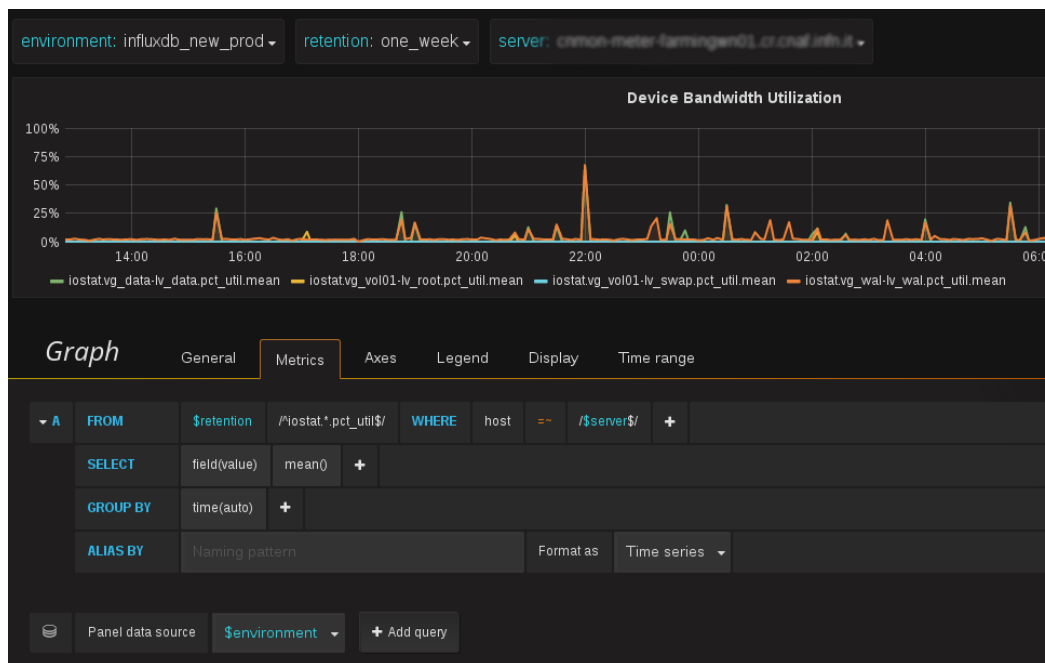


Figura 3.15: Grafana query composer



Grafana permette inoltre la creazione di variabili a livello di dashboard (fig. 3.16). Queste variabili sono utilizzabili all'interno delle query, in modo da rendere parametrica la generazione dei grafici.

Inoltre è possibile selezionare più valori in uno stesso dropdown menù, facendo in modo che una stessa riga venga moltiplicata per ogni elemento selezionato, così da poter confrontare più elementi nello stesso momento.

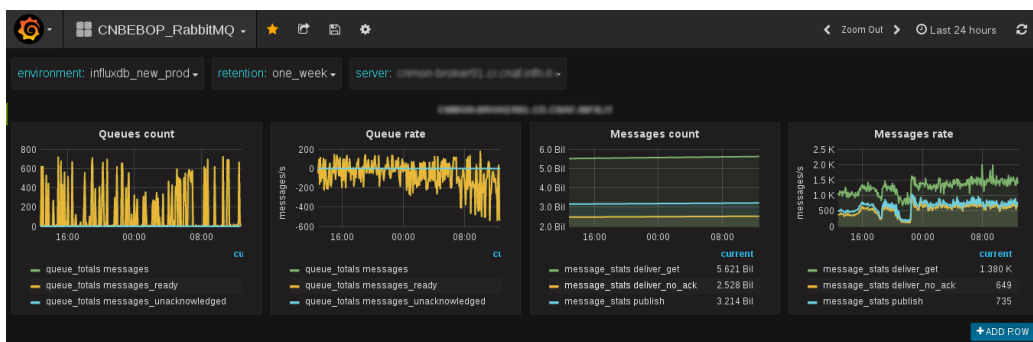


Figura 3.16: Grafana template variable

Il risultato ottenuto è una dashboard parametrica con diverse tipologie di grafici quali singlestat panel, table e grafici (fig. 3.17).

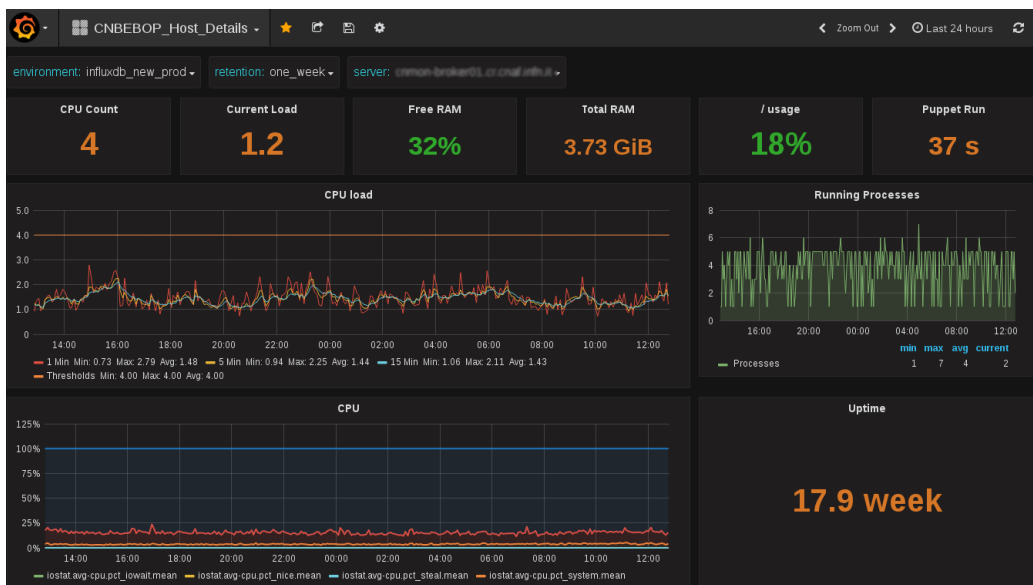


Figura 3.17: Grafana dashboard

## 3.8 Risultati raggiunti

Tramite il setup descritto, è stato possibile ottenere un infrastruttura di monitoraggio centralizzata, al momento in produzione per quattro reparti del CNAF.

Ogni reparto (in completa autonomia), a seconda delle proprie esigenze, può decidere di collegare un numero arbitrario di Sensu server, per ottenere un setup ridondante o per necessità di smaltimento dei messaggi in RabbitMQ, in seguito ad un alto numero di client connessi.

Per facilitare e ridurre al minimo lo sforzo necessario alla migrazione dal precedente sistema, sono stati predisposti dei moduli Puppet di interfaccia (parametrizzati) da fornire ad ogni reparto, in modo da rendere il setup di tutti gli strumenti, Sensu server, Sensu client, Uchiwa e Grafana il più semplice possibile.

Il gruppo "Bebop", del quale faccio parte, che fornisce e mantiene questa infrastruttura centrale di monitoraggio, fornisce all'utente finale una serie di endpoint relativi a RabbitMQ e InfluxDB ai quali collegarsi, rendendo totalmente trasparente l'infrastruttura sottostante, delegando all'utente solo la personalizzazione del monitoraggio della propria infrastruttura, ossia la creazione di nuovi check, modifiche al sistema di notifica e creazione di nuove dashboard.

Tra i benefici riscontrati da questa infrastruttura, rispetto alle altre soluzioni mostrate, troviamo la grande agilità nella aggiunta di nuovi elementi da monitorare, la compatibilità dei sensori esistenti con il nuovo sistema e la piena integrazione con Puppet e Foreman.

Grazie a queste caratteristiche, è stato possibile migrare al nuovo sistema in tempi brevi, garantendo un perfetta interazione tra la configurazione di un servizio e il suo monitoraggio. Tramite questo approccio, la configurazione di un servizio e il suo monitoraggio possono andare di pari passo, riducendo gli errori umani, il tempo necessario alla integrazione di nuovi sistemi e la manutenzione di quelli già presenti.

Inoltre, grazie alla modularità del sistema di monitoraggio, alla possibilità di interazione con Sensu ed InfluxDB tramite API RESTful e alle dashboard presenti su Grafana e Uchiwa, abbiamo riscontrato una grande praticità nel controllo della infrastruttura e individuazione/risoluzione di problematiche.



# Capitolo 4

## Conclusioni e sviluppi futuri

Lo scopo di questa tesi era la descrizione di un progetto relativo alla implementazione di un sistema centralizzato di monitoraggio per il CNAF, in sostituzione di quello attualmente presente.

Il risultato di questo progetto è stato il setup di un infrastruttura di monitoraggio centralizzata, utilizzabile da tutti i reparti del CNAF in un setup altamente affidabile.

Il sistema ottenuto è in grado di reggere il carico di lavoro generato da più di 1500 entità collegate a tale infrastruttura e scalare orizzontalmente a seconda delle esigenze, garantendo al centro il monitoraggio di tutti i suoi servizi.

Tra i benefici riscontrati da questa nuova infrastruttura, abbiamo la grande agilità nella aggiunta di nuovi elementi e la piena integrazione con Puppet e Foreman, riducendo così errori umani e sforzo necessario per integrare nuovi sistemi.

Il tempo necessario alla migrazione dai vecchi sistemi è stato breve come previsto grazie alla piena compatibilità dei sensori esistenti con il nuovo sistema e alla facilità di creazione delle dashboard.

Inoltre, utilizzando tutti lo stesso sistema, è stato possibile condividere le proprie esperienze con gli altri reparti, andando a ridurre ulteriormente i tempi di apprendimento e setup.

Grazie alle nuove dashboard Grafana e Uchiwa abbiamo riscontrato una grande praticità nel controllo della infrastruttura e individuazione/-risoluzione di problematiche.

Tra gli sviluppi futuri, troviamo l'inclusione dei reparti rimanenti nella infrastruttura creata, con particolare riguardo al monitoraggio della rete e l'adozione di un sistema di analisi dei log così da poter incrociare dati di monitoraggio e allarmistica con i log di sistema e dei servizi, per garantire una copertura totale della infrastruttura.

In conclusione, ritengo che questa esperienza abbia avuto un esito positivo per il bagaglio culturale da me acquisito, sia per la parte puramente tecnica, sia per la parte legata alla interazione tra i vari reparti per la raccolta dei requisiti e la risoluzione di problematiche legate al passaggio dal vecchio al nuovo sistema.

# Appendice A

## Puppet

*Puppet* [6] è un software open source che permette la gestione automatizzata della configurazione di servizi tramite codice.

La possibilità di gestire la propria infrastruttura tramite questo approccio porta alcuni vantaggi:

- Coerenza degli ambienti;
- Tracciabilità delle modifiche;
- Maggiore facilità di replica delle configurazioni e limitazione degli interventi manuali;
- Riduzione del tempo di setup dei servizi e documentazione implicita della infrastruttura;
- Possibilità di versionare la propria infrastruttura tramite sistemi VCS (Version Control Systems);
- Possibilità di applicare un sistema di continuous integration per la propria infrastruttura.

Il ruolo di Puppet nella gestione della infrastruttura risulta maggiore di quello di uno script lanciato al termine dell'installazione di una macchina: Puppet controlla la configurazione della macchina in maniera attiva, correggendo se necessario la configurazione e riavviando i servizi in caso di bisogno, garantendo la coerenza tra quello descritto dal codice e quello realmente presente sulla macchina.

### A.0.1 Come funziona

L'idea alla base di Puppet è la costruzione di moduli che gestiscano ognuno un aspetto della configurazione di un sistema, come per esempio Apache.

La community Puppet, tramite il portale Puppetforge, mette a disposizione una ampia gamma di moduli per i servizi più usati, come Apache, MySQL, PostgreSQL, Nginx, dando la possibilità all'utilizzatore finale di sfruttare moduli esistenti, evitando di doverli scrivere da zero, proponendo invece modifiche al codice del progetto usato in caso di necessità.

Per l'installazione di un modulo dal repository Puppetforge possiamo utilizzare il seguente comando:

---

```
1 puppet module install <autore>--<nomemodulo>
```

---

Il modulo verrà così installato al path `/etc/puppet/modules/<nomemodulo>`

La struttura di un modulo è la seguente:

---

```
1 |--- files
2 |--- Gemfile
3 |--- manifests
4 |--- metadata.json
5 |--- Rakefile
6 |--- README.md
7 |--- spec
8 |--- templates
9 |--- tests
```

---

- **Manifests:** contiene la definizione delle classi Puppet
- **Files:** contiene file statici che un nodo può scaricare
- **Templates:** contiene dei template che il modulo può usare

Un esempio di codice di una classe Puppet è il seguente:

---

```
1 class mymodule (
2   $myparam = 'pippo',
3 ){
4   $dir_base      = '/root/script_puppet'
5   $my_dir        = "${dir_base}/test2"
6   $my_file_name = 'file_test2'
```

```
7   $my_file      = "${my_dir}/${my_file_name}"
8
9   Exec {
10      cwd  => '/',
11      path => '/usr/bin:/usr/sbin:/bin:/sbin:/usr/local/bin:/
        usr/local/sbin',
12  }
13
14  cron { 'crontab_test1':
15      command => "echo \"ciao\" >> ${my_dir}/tmp_file ",
16      user    => 'root',
17      hour    => 15,
18      minute  => 30,
19  }
20
21  file { [$dir_base, $my_dir]:
22      ensure => directory,
23      mode   => '0644',
24      owner  => 'root',
25      group  => 'root',
26  }
27
28
29  file { $my_file:
30      ensure => file,
31      owner  => 'root',
32      group  => 'root',
33      content => 'Hello world',
34      notify => Exec["chmod_${my_file_name}"],
35  }
36
37  package { 'ntp':
38      ensure => present,
39  }
40
41  service { 'ntpd':
42      ensure => running,
43      enable => true,
44  }
45
46  exec { "chmod_${my_file_name}":
47      command      => "chmod 777 ${my_file} && echo \"pippo >>
        ${my_file} ",
48      environment => ['pippo=pluto'],
49      refreshonly => true,
50  }
51
52  notify {$myparam: }
53
54 }
```



---

Per eseguirlo, possiamo creare un file (manifesto) in cui includiamo il modulo:

**myfile.pp**

```
1 class {'mymodule':  
2   myparam => 'ciao',  
3 }
```

---

successivamente applichiamo il manifesto creato:

```
1 puppet apply myfile.pp
```

Le modalità di esecuzione di Puppet possono essere di due tipi:

- Masterless (mostrata nell'esempio precedente)
- Agent-master

La modalità agent-master richiede il setup di due componenti:

- Puppetmaster
- PuppetCA

In questa modalità:

- I moduli non sono installati su ogni client ma solo sui Puppet master
- L'esecuzione del client in questo caso avviene mediante il comando "puppet agent -t", di solito tramite crontab
- Ogni client richiede una richiesta di compilazione di un catalogo delle configurazioni al server, per poi applicare le configurazioni richieste localmente
- Ogni client, per poter comunicare con il server deve ottenere un certificato valido dalla PuppetCA

# Appendice B

## Foreman

*Foreman* [5] è un tool open source che permette la gestione completa del ciclo di vita di un server, pienamente integrato con Puppet e con i sistemi di virtualizzazioni più usati, con il quale è possibile interagire tramite una apposita CLI, API RESTful e una moderna UI (fig. B.1).

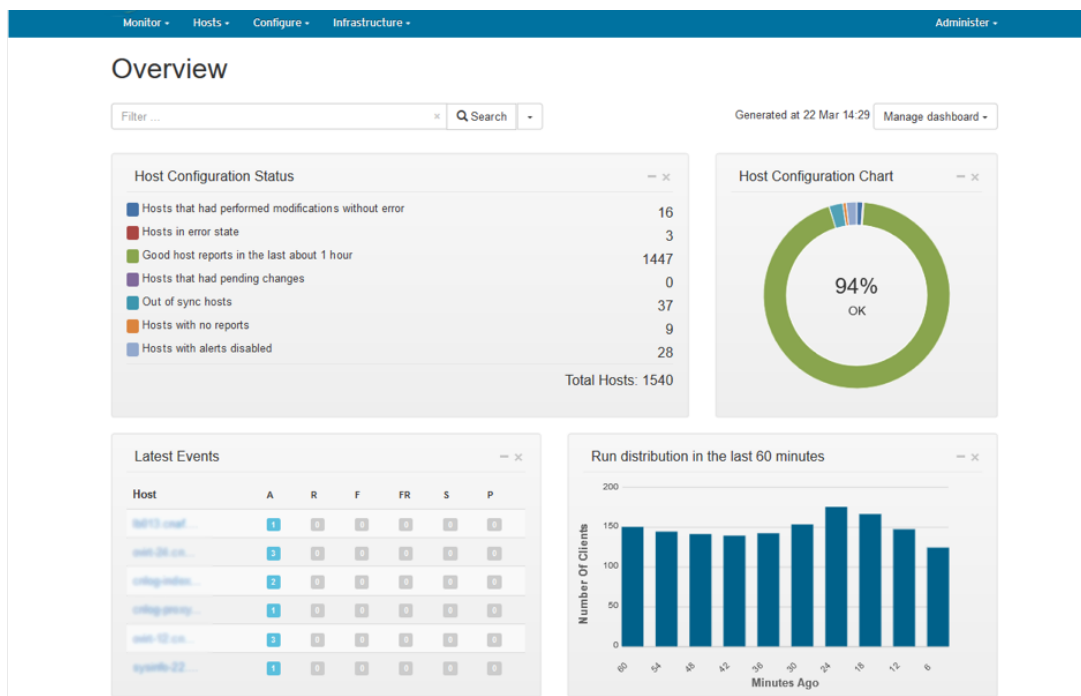


Figura B.1: Foreman UI

Nella interazione tra Foreman e Puppet, Foreman viene identificato come un ENC (External Node Classifier), ossia uno strumento utilizzato dai Puppetmaster per identificare classi e parametri associate ad uno specifico nodo e al quale inviare report relativi alle configurazioni applicate.

Foreman permette di raggruppare sotto una unica interfaccia:

- Installazione di macchine fisiche;
- Installazione di macchine virtuali;
- Gestione dei moduli Puppet;
- Creazione di gruppi logici di host;
- Gestione dei parametri associati a moduli Puppet, dando la possibilità di associare ad un dato gruppo di host determinati moduli, con specifici parametri;
- Monitoraggio dello stato delle installazioni dei sistemi;
- Monitoraggio dello stato delle configurazioni dei sistemi.

Oltre a quanto indicato, esso permette di interagire direttamente con alcune componenti necessarie alla installazione e configurazione come PuppetCA, DHCP, TFTP e DNS tramite l'apposito software Foreman smart-proxy (fig. B.2).

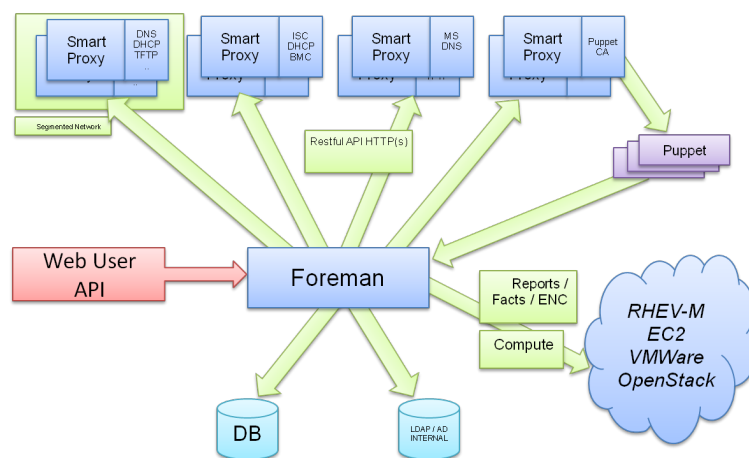


Figura B.2: Architettura Foreman

# Bibliografia

- [1] Centro nazionale per la ricerca e lo sviluppo nelle tecnologie informatiche e telematiche. <https://www.cnaf.infn.it/>.
- [2] A frameworks for reliable, scalable, distributed computing and data storage. <http://hadoop.apache.org/>.
- [3] High performance tcp/httpd load balancer. <http://www.haproxy.org/>.
- [4] Javascript mvw framework. <https://angularjs.org/>.
- [5] Open source complete life cycle systems management tool for provisioning, configuring and monitoring of physical and virtual servers. <https://theforeman.org/>.
- [6] Open source configuration management tool. <https://puppet.com/>.
- [7] Open source dashboard for sensu. <https://uchiwa.io/>.
- [8] Open source dashboards composer. <http://grafana.org/>.
- [9] Open source in-memory database. <http://redis.io/>.
- [10] Open source message broker. <https://www.rabbitmq.com/>.
- [11] Open source monitoring solution. <https://sensuapp.org/>.
- [12] Open source network monitoring solution. <https://www.nagios.org/>.

- [13] Open source network monitoring solution. <http://www.zabbix.com/>.
- [14] Open source programming language. <https://golang.org/>.
- [15] Open source time-series database. <http://www.mrtg.org/rrdtool/>.
- [16] Open source time-series database. <https://graphiteapp.org/>.
- [17] Open source time-series database. <https://www.influxdata.com/>.
- [18] Open source time-series database. <http://opentsdb.net/>.
- [19] A routing software written in c. <http://www.keepalived.org/>.
- [20] Search engine based on lucene. <https://www.elastic.co/>.
- [21] Time series databases comparison. <https://blog.dataloop.io/top10-open-source-time-series-databases>.
- [22] A unix daemon that collects, transfers and stores performance data of computers and network equipment. <https://collectd.org/>.
- [23] Worldwide lhc computing grid. <http://wlcg.web.cern.ch/>.
- [24] Babik Marian, Fedorko Ivan, Hook Nicholas, Lansdale Thomas Hector, Lenkes Daniel, Siket Miroslav, and Waldron Denis. Lemon - lhc era monitoring for large-scale infrastructures. 2011.

# Elenco delle figure

1.1	CNAF datacenter . . . . .	2
1.2	CNAF WAN . . . . .	3
2.1	Architettura Nagios . . . . .	11
2.2	Funzionamento NRPE . . . . .	12
2.3	Funzionamento NSCA . . . . .	12
2.4	Nagios UI . . . . .	14
2.5	Architettura Lemon . . . . .	15
2.6	Lemon Web . . . . .	16
2.7	Zabbix UI . . . . .	19
2.8	Collectd UI . . . . .	20
3.1	Infrastruttura di monitoraggio . . . . .	26
3.2	Sensu workflow . . . . .	29
3.3	Redis Sentinel . . . . .	46
3.4	RabbitMQ exchange . . . . .	48
3.5	Fanout exchange . . . . .	49
3.6	Direct exchange . . . . .	49
3.7	Sensu exchange . . . . .	50
3.8	RabbitMQ admin plugin . . . . .	52
3.9	Esempio HAProxy RabbitMQ . . . . .	53
3.10	HAProxy RabbitMQ . . . . .	54
3.11	HAProxy Redis . . . . .	55
3.12	Uchiwa UI . . . . .	61
3.13	Uchiwa aggregate check . . . . .	62
3.14	Uchiwa details . . . . .	62
3.15	Grafana query composer . . . . .	63

3.16 Grafana template variable . . . . .	64
3.17 Grafana dashboard . . . . .	64
B.1 Foreman UI . . . . .	73
B.2 Architettura Foreman . . . . .	74

# Elenco delle tabelle

1.1	CNAF in numeri . . . . .	4
2.1	Tabella degli stati Nagios . . . . .	11
2.2	InfluxDB vs Graphite [21] . . . . .	22
2.3	OpenTSDB vs Elasticsearch [21] . . . . .	22
3.1	Tabella degli stati Sensus . . . . .	32





# Ringraziamenti

Desidero ringraziare il Professor. Fabio Panzieri e il Dottor. Andrea Ceccanti per avermi dato questa possibilità ed essere stati una guida durante la stesura della tesi.

Ringrazio inoltre i colleghi del CNAF per avermi dato la possibilità di partecipare a questo progetto, in particolare i membri del gruppo "Be-bop" Diego Michelotto e Giuseppe Misurelli con i quali ho collaborato e condiviso questa esperienza.

Un ringraziamento particolare alla mia famiglia per avermi supportato e avermi dato la possibilità di seguire questo percorso universitario.

Ringrazio Elisa per essermi sempre stata vicina nei momenti difficili e avermi supportato e spronato durante questi anni.

Ringrazio infine tutti gli amici che mi hanno incoraggiato e supportato.