Server Surveillance

By Mads Pedersen

This document shortly describes a small vacation project I did the Christmas of 2016. The motivation for this project is to make server level solutions health visible for a team’s members on a simple and easy way with a minimum of effect on the surveyed server. The reason for this is to find a cheaper alternative for subscription based products that can achieve the same – NewRelic as an example.

# Features

* Graph visualization of server’s connection
* Tests on server configuration
  + Check if App Pools is running
  + Check if Windows Services is running
  + Check if Windows Feature is enabled
  + Check if Scheduled Task is enabled
  + Check if a port is opened to a server or IP
  + HTTP Requests – checked for if GZip is enabled, SSL is active, and if the status code is correct
* Self-healing feature
  + If a test is failing it is possible to execute an implemented cure
  + Cures for:
    - Starting a stopped App Pool
    - Starting a stopped Windows Service
    - Enabling a Windows Feature
* Report generation
  + Generates a flat HTML document containing info about the setup
* Data collection of following statistics
  + Server
  + Redis
  + MongoDb
* Data graph representation
  + Server memory, cpu, and network in and out
  + Redis commands executed
  + Mongo actions (queries, inserts, updates and deletes) + Mongo network in and out
* Easy JSON configuration of the solution servers

# Basic introduction to execution

The program is simple. It takes a well-structured JSON document and executes the tests, the data collection and the requests tests in parallel and collect the results and generates the report file in a HTML format. The collected data is persisted to disk for a period of 60 min per default and is automatically deleted when it gets older than that. There is a plan for permanently storing the data in a data store (see future features). Currently it is rather expensive to execute all the tests at once on the target servers, to limit the CPU footprint for the application the test will not be executed as often as the data collection tasks. As per default the tests are executed every 60 minutes and the data collection tasks are executed either per 10 seconds or per one minute. In the future performance for the tests will be a priority to make sure that the tasks can be executed more frequently without having as big a footprint. The reports is generated every 60 seconds and the frontend is updated automatically through javascript.

The tasks of any kinds are executed in a thread with the “Idle” status so the CPU can down-prioritize the task at a time of high CPU load. The CPU footprint of the data collection is measured to be max 5% and a average on 3% CPU utilization at a 2 core server and basically no RAM usage to notice.

The configuration file has the following properties:

* Alias
  + A unique name for the configuration
* Header
  + The header used on the reports
* Description
  + Text to describe the configuration and solution setup, this is also displayed on the report
* Credentials
  + An array of objects containing: alias, username and password fields
* Servers
  + An array of servers
* Requests
  + An array of requests

A server is defined by the following fields:

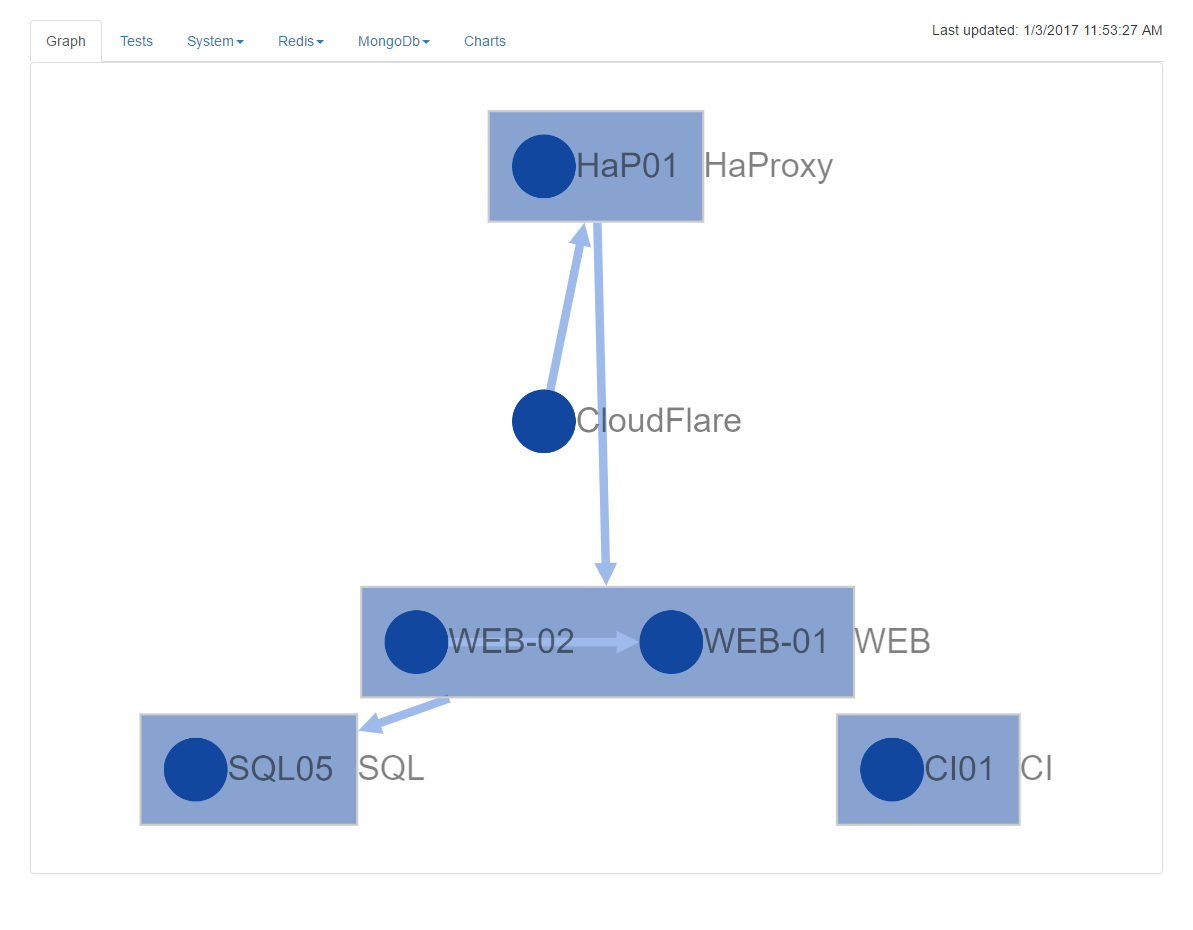
* CredentialAlias
  + Reference to one of the defined credentials in the configuration
* Label
  + A text defining the group the server is a part of
* DisplayName
  + The name displayed on the graph by the server node
* Name
  + The network name of the server or the IP of the server
* AppPools
  + Array of string defining the names of the app pools that is required to run
* ScheduledTasks
  + Array of string defining the names of the scheduled tasks that have to be enabled on the server
* Services
  + Array of string defining the names of the Windows Services that have to run on the server
* Features
  + Array of string defining the names of the Windows Features that have to be enabled on the server
* RequiredConnections
  + The connections defining the relationship to other servers, these objects are also used in the Port available test. A RequiredConnection object contains of the following fields: host, port and name
* Ignore
  + Array of aliases on actions that must be ignored for the server
* Additionals
  + Dictionary of different fields used for e.g. to log into the Redis for information collection

A Request object contains the following fields:

* Label
  + Short description of the URL
* Uri
  + The URI for the request
* SSL
  + Boolean defining if you should test for it or not
* Gzip
  + Boolean defining if you should test for it or not

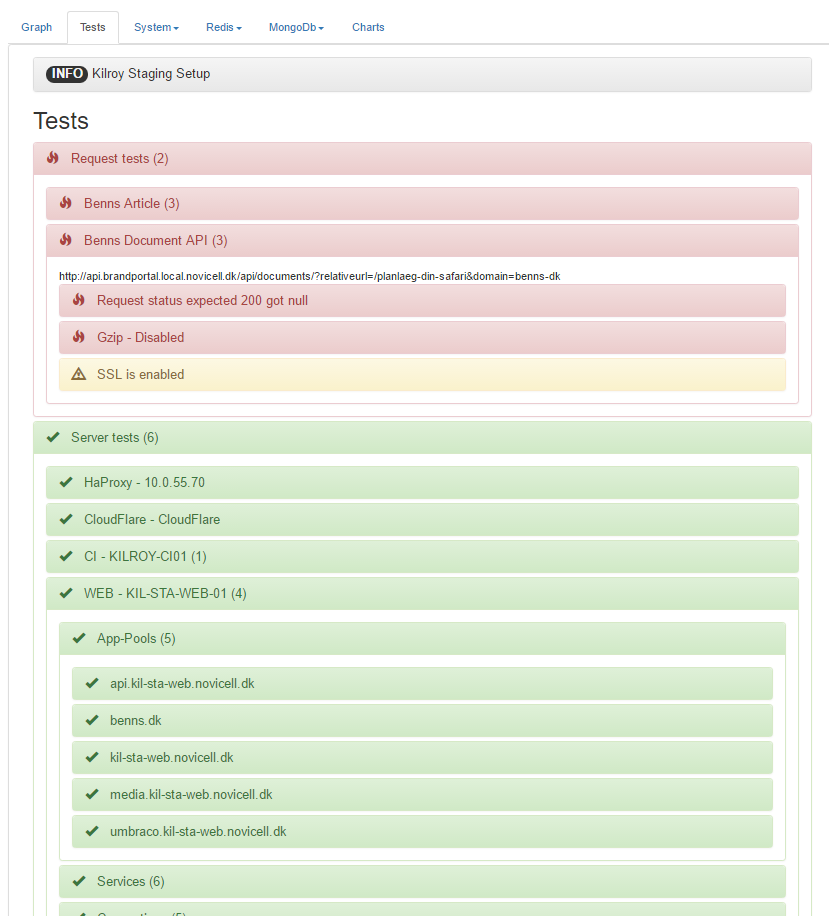
# User interface

The first tab in the report is representing the server level architecture graph that shows the different connections between the servers. This is generated based on the Port tests that specify which servers a specific server should have access to. If you have a server, e.g. CloudFlare, that you don’t have access to remotely, you are still able to include it into your JSON configuration. Just use the “Ignore” array to specify that it should not execute specific tests for this server.

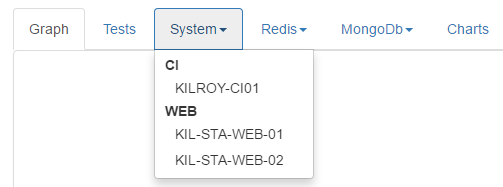
This is here that you can see how the label field for a server is used. If two servers have the same label they are placed in the same box ergo in the same grouping of servers. This way you can group servers into clusters of servers. The relationship arrows displays that a server is dependent on another server by direction.

The second tab shows the different tests executed in the configuration. There will be different colors indicating different results. If a group contains both red and yellow results then the total result will be an error (red) because it have precedence over the warning.

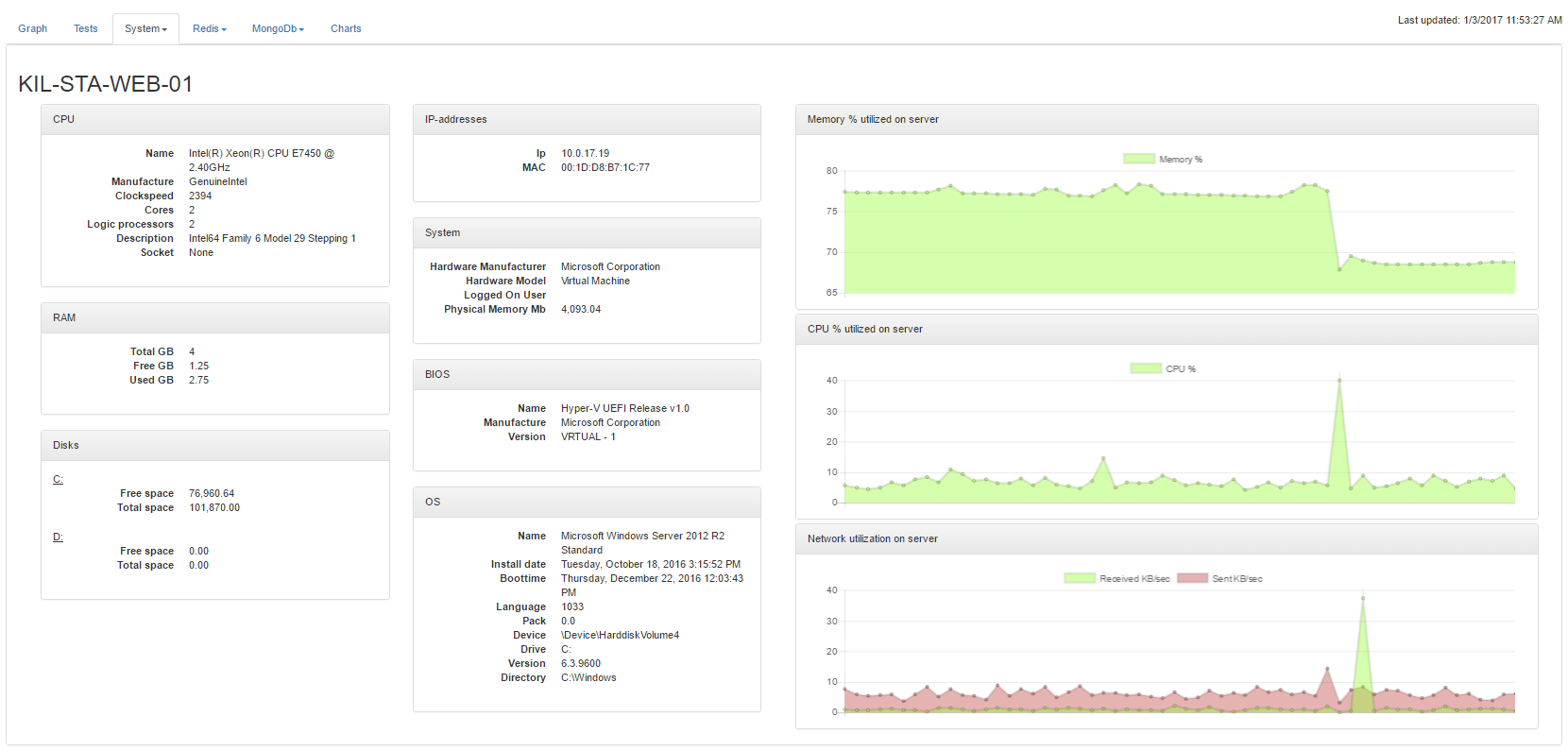
If a server have a service that is disabled by some reason and the program is configured to heal the setup then it will display the test for the disabled service as yellow. There will in this case also display a result of the healing action. If it went well the cure-result will be yellow and, else red.



The next tab is going to display information on the different servers there is remote PowerShell access to. Just to mention the navigation. For both the System, Redis and Mongo tabs the dropdown is grouped according to the Label field for the servers. This is just as in the before mentioned graph representation.

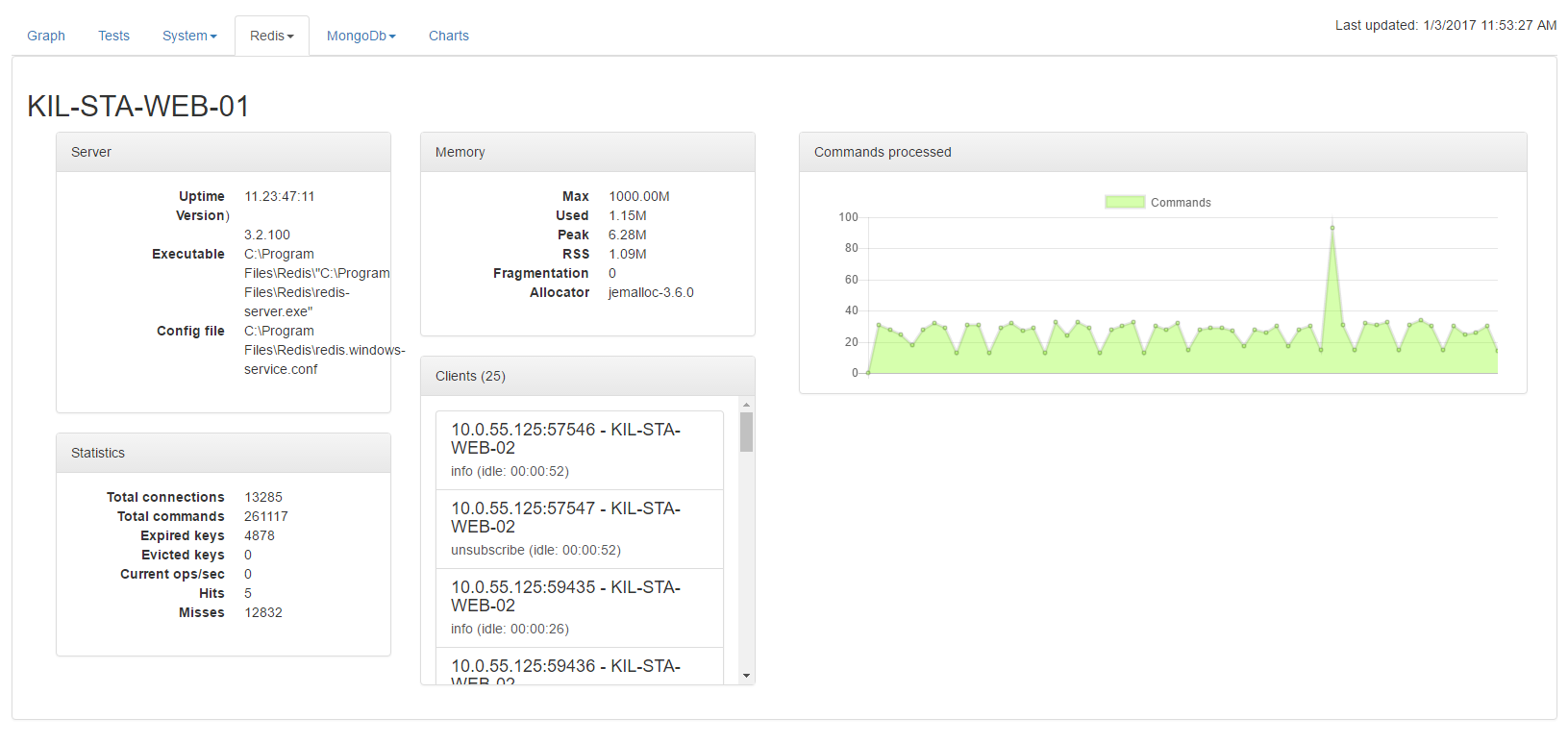


The next tab displays information about the server itself. This is about the hardware configurations as in CPU, RAM, storage etc. Along with that there is representations of the used RAM and CPU along with the usage of Network in and out.



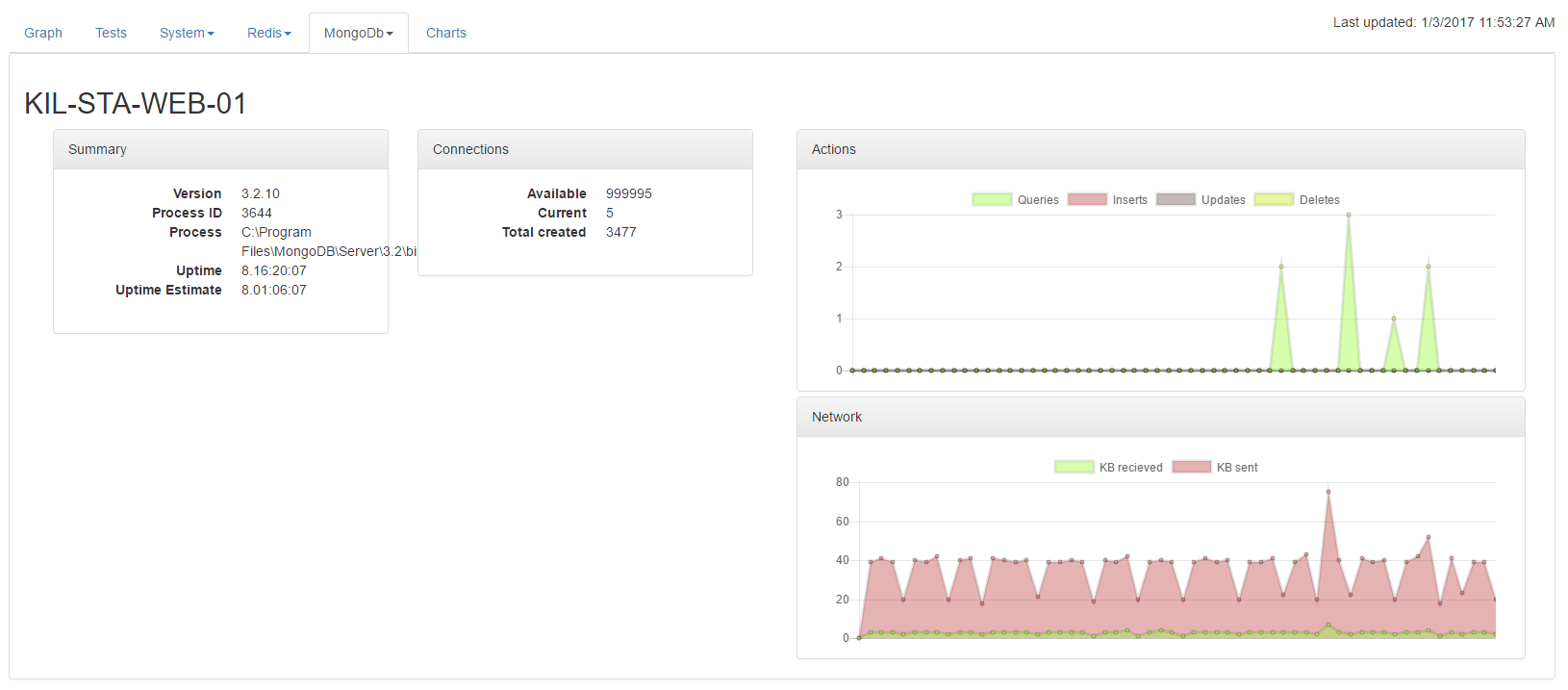
The next two tabs is almost the same as this, the difference is the context of the views. Where the prior view is concerned about the server itself the next two have process specific concerns and contains statistics about Redis and MongoDb (se future features)

## Redis

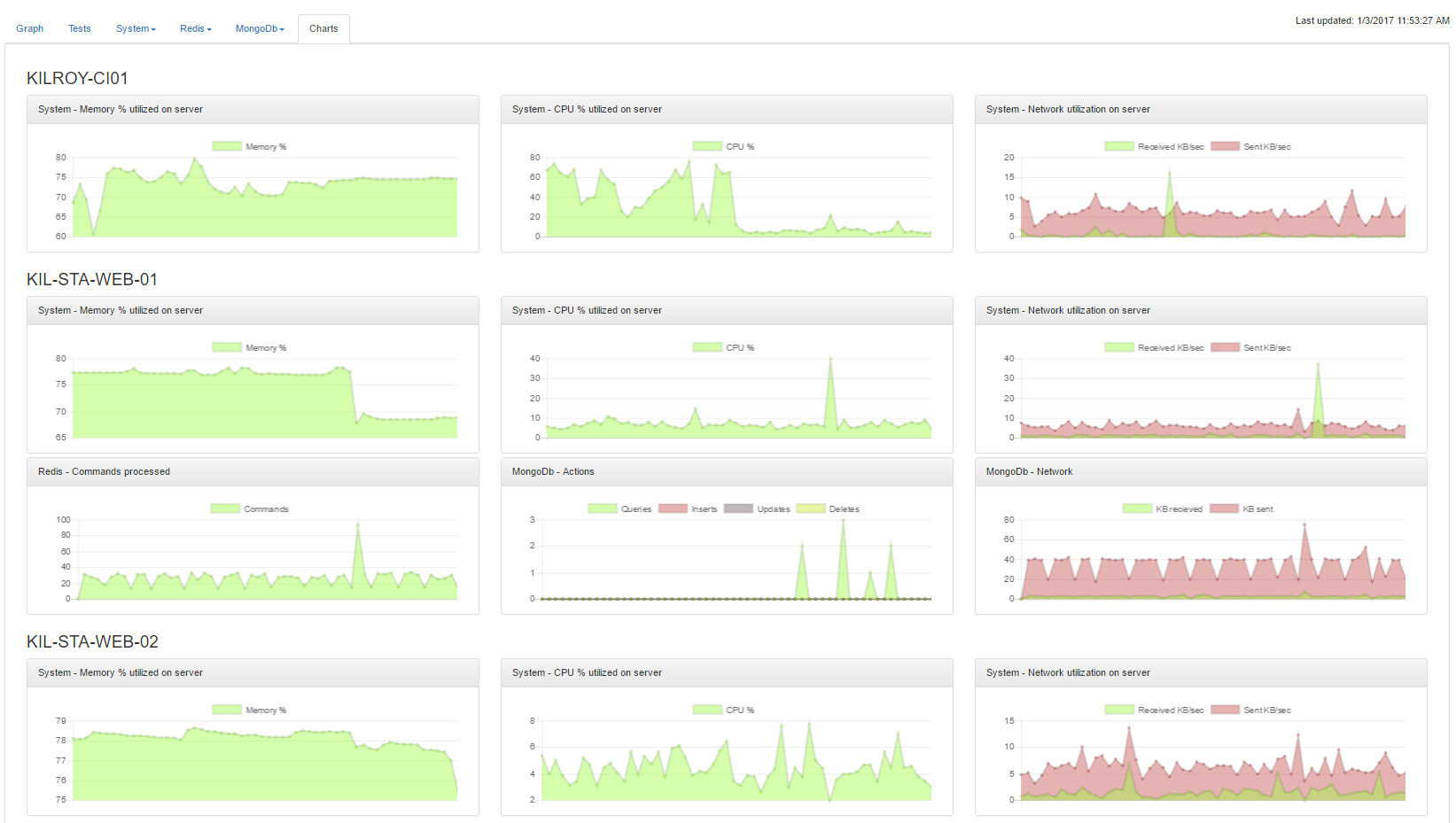


The redis view concerns it self about displaying how manu commands gets executed against the process and also the display of RAM usage for the process (not in this screencap). As a difference it also displays the different clients currently connected to the server instance.

## MongoDb



In the mongo server representation is currently consisting of basic server information, which is going to be expanded in the future to contain even more valuable information about the state. Along with this it contains two graphs: one that shows the current count of actions on the server and a network utilization chart. There will in the future also be a few other graphs based on the metric functionality from the Mongo API



The final view is a collected version of all the charts on the different server tabs. The reason for this view is to make all the data visible through a media that all members of the team can grasp, even management should be able to understand and use them. The reason for this is also to be able to highlight a graph that is out of its baseline (see future features).

# Future features

This section describes the ideas for features that can be implemented in the future to make the project more usable in other contexts.

## SQL data collection

To be able to show the statistics from a SQL (MSSQL) instance and though that display if there is any issues in the way the server is queried. This is also where it could be preferable to show recommended indexes and display the slowest queries executed.

## RabbitMQ data collection

Collect information about messages processed on different queues and display how the RabbitMQ configuration currently is. The list of queues and clients along with the load on the different queues.

## HaProxy data collection

Displaying data about in and out going traffic along with which server configuration is up and which is not.

## Permanent data storage

There have to be a way in the future to store the data for future usage. The way this could be done is by using a NoSQL data store e.g. MongoDB.

## Baseline charts

Calculate baselines based on the charts dataset so that alerts can be generated and acted opon.

## Better frontend

Right now it is a flat and non-interactive HTML file that is the generated result from the program. This could be a more interactive frontend based on a MVC website.

## Install as a Windows Service

Take the current executable program and implement as a windows service that could run in the background and at startup. That way it could be installed on a server in the solutions server setup that is constantly running and collecting data.

## Alerting

When the baseline is implemented the minimum alerting system should be a Slack channel integration displaying the start and end of an event.

# Appendix A – Configuration file example

{

"Alias": "MPE",

"Header": "MPE Staging Setup",

"Description": "This will take the configuration for the MPE staging infrestructure, hosted at Novicell, and run tests based on the configuration file. This includes checking on if app-pools, windows-services, scheduled tasks are running and enabled along with ensuring that windows features are enabled. Besides this it will check if a port to another server is oppened. A final set of tests is based on URLs that get requested, it will the check if compression and SSL is enabled along with the correct status code",

"Credentials": [{

"Alias": "default",

"Username": "default",

"Password": "default"

}],

"Servers": [{

"CredentialAlias": "mpe",

"Label": "MPE",

"DisplayName": "MPE",

"Name": "MPE-SS",

"AppPools": ["test.mpe.ss"],

"ScheduledTasks": [],

"Services": [

"MongoDB",

"OctopusDeploy Tentacle",

"Redis",

"RabbitMQ",

"WAS",

"W3SVC"

],

"Features": ["Web-Server",

"Web-WebServer",

"Web-Common-Http",

"Web-Default-Doc",

"Web-Dir-Browsing",

"Web-Http-Errors",

"Web-Static-Content",

"Web-Http-Redirect"],

"RequiredConnections": [{

"Host": "127.0.0.1",

"Port": 80,

"Name": "HTTP"

}],

"Ignore":[]

"Additionals": {

"Redis.Password": "",

"Redis.Port": 6379,

"MongoDB.Username": "",

"MongoDB.Password": "",

"MongoDB.AuthenticationDatabase": "",

"MongoDb.Database": ""

}

}],

"Requests": [{

"Label": "Google front page",

"Uri": "https://google.com",

"Ssl": true,

"StatusCode": 200,

"Gzip": true

}]

}