Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων
Και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης Και Υπολογιστων

Πλατφόρμα Βασισμένη σε Οντολογίες για τη Διαχείριση Υποδομής Κέντρου Δεδομένων, με Προγραμματιστικό Έλεγχο

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ιωάννης Α. Ανδρουλιδάκης

Αθήνα, Ιούνιος 2017

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων Και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης Και Υπολογιστων

An Ontology-Based Platform for Data Center Infrastructure Management, with Programmatic Control

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ιωάννης Α. Ανδρουλιδάκης

Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23η Ιουνίου 2017.

. . . .	. . . .	. . . .
Νεκτάριος Κοζύρης	Νικόλαος Παπασπύρου	Γεώργιος Γκούμας
Καθηγητής ΕΜΠ	Αν. Καθηγητής ΕΜΠ	Επ. Καθηγητής ΕΜΠ

Αθήνα, Ιούνιος 2017

. . . .

Ιωάννης Α. Ανδρουλιδάκης
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Περίληψη

Στην παρούσα διπλωματική εργασία παρουσιάζουμε το σχεδιασμό και την υλοποίηση του *OntoMon*, μιας διαλειτουργικής πλατφόρμας για την παρακολούθηση και οπτικοποίηση ετερογενών υπολογιστικών συστημάτων. Το OntoMon είναι ένα πλαίσιο λογισμικού αφηρημένου περιεχομένου, χτίζεται πάνω σε Οντολογίες και δεν εξαρτάται από τη σημασιολογία του εκάστοτε συστήματος-στόχου. Θεμελιώνει τη δική του στοίβα λογισμικού, ελέγχει προγραμματιστικά τις δομικές του μονάδες και εισάγει ένα αφηρημένο μοντέλο αντικειμένων για την αναπαράσταση όλων των οντοτήτων. Βασική μας επιδίωξη είναι η ενεργή επίβλεψη και διαχείρηση υποδομής Κέντρων Δεδομένων μεγάλης κλίμακας σε πραγματικό χρόνο, με σκοπό τη διατήρηση της λειτουργικότητας.

Στη σημερινή εποχή, ο ρυθμός παραγωγής δεδομένων αυξάνεται εκθετικά, καθώς οι ψηφιακές υπηρεσίες και συναλλαγές παράγουν, περίπου, 2.5 exabytes καθημερινά. Συνυπολογίζοντας τις αέναες ανάγκες για συνέπεια και διαθεσιμότητα των δεδομένων αυτών, είναι προφανές ότι τόσο η ακαδημαϊκή κοινότητα, όσο και αυτή της Τεχνολογίας Πληροφοριών είναι αντιμέτωπες με την πρόκληση της αποδοτικής αποθήκευσης και διαχείρισης ψηφιακών δεδομένων μεγάλης κλίμακας. Η μαζική ανάκτηση και επεξεργασία τους απαιτεί τη συνεχή διαθεσιμότητα τόσο των φυσικών πόρων, όσο και των υπηρεσιών λογισμικού εντός των Κέντρων Δεδομένων, προκειμένου να αποφεύγονται κάθε είδους αποτυχίες ή απώλειες. Το γεγονός αυτό, καταδεικνύει την τεράστια σημασία των εργαλείων παρακολούθησης της υποδομής των Κέντρων Δεδομένων, προκειμένου οι διαχειριστές να μπορούν να αποφανθούν άμεσα για την τρέχουσα κατάσταση λειτουργίας των επιμέρους τμημάτων υλικού και λογισμικού, πραγματοποιώντας σχετικούς ελέγχους και εκτιμήσεις απόδοσης. Ακόλουθα, η βαθύτερη κατανόηση της συμπεριφοράς τους, αλλά και των εσωτερικών τους αλληλεπιδράσεων, οδηγεί σε αποδοτικότερη συνολική διαχείριση, περιορισμό του κόστους λειτουργίας, λήψη εμπεριστατωμένων αποφάσεων και, μακροπρόθεσμα, βελτίωση των παρεχόμενων υπηρεσιών.

Το OntoMon ακολουθεί μια πολυεπίπεδη αρχιτεκτονική, η οποία ενσωματώνει πολυάριθμα υποσυστήματα, αξιοποιώντας κλιμακώσιμες τεχνολογίες ανοικτού κώδικα. Έτσι, καταμερίζει τις εργασίες και προσθέτει λειτουργικότητα. Προκειμένου να διευκολύνει τη διαχείριση διαφορετικών υπολογιστικών συστημάτων, το OntoMon συλλέγει και επεξεργάζεται μετρικές απόδοσης από τμήματα υλικού και λογισμικού σε πραγματικό χρόνο, και πραγματοποιεί συγκρίσεις με τιμές-κατώφλια. Παράλληλα, υποστηρίζει μια δυναμική, ευέλικτη και πλήρως προσαρμόσιμη Διεπαφή Χρήστη, η οποία παρουσιάζει την τρέχουσα κατάσταση της παρακολουθούμενης υποδομής. Για να επικυρώσουμε τη λειτουργικότητα της πλατφόρμας μας, πραγματοποιήσαμε 2 διαφορετικές δοκιμές: στην πρώτη περίπτωση παρακολουθήσαμε τη φυσική υποδομή ενός νοητού Κέντρου Dεδομένων, ενώ στη δεύτερη περίπτωση εστιάσαμε σε ένα λογισμικό κατανεμημένης αποθήκευσης δεδομένων, εγκατεστημένο σε συστοιχία υπολογιστικών κόμβων.

Λέξεις-Κλειδιά

Οντολογίες, διαχείριση υποδομής κέντρου δεδομένων, αφηρημένο μοντέλο αντικειμένων, κατανεμημένη παρακολούθηση, χρονοσειρές, διαδικτυακή διεπαφή χρήστη, οπτικοποίηση πραγματικού χρόνου, προγραμματιστικός έλεγχος, SVG, Icinga, Influx, Grafana, Angular

Abstract

In this thesis, we present the design and implementation of *OntoMon*, an interoperable monitoring and visualization platform that is capable of managing heterogeneous computing systems. OntoMon is a content-agnostic framework that is founded upon Ontologies and does not rely on the semantics of each target system. We introduce a custom software stack that handles all core components programmatically and relies on an abstract object model that represents all entities. Our main objective is the proactive, real-time supervision and management of IT infrastructure housed in large-scale Data Centers, aiming at preserving operational efficiency.

In our times, the rate of data production is exponentially growing as digital services and digital transactions produce approximately 2.5 exabytes of data on a daily basis. Considering the everlasting needs for data consistency and availability, it is obvious that both the IT community and the academia are confronted with the tremendously challenging tasks of hyperscale storage and processing. Querying and analyzing massive amounts of data require full-time hardware and software availability, in order to prevent data loss or any kind of system failure. This indicates the great importance of Data Center infrastructure monitoring so that administrators can instantly verify the functionality of both physical and software assets. In this regard, performance checks and time-series analysis need to be conducted, in order to observe their overall behavior and interactions inside the underlying system. Hence, the deeper understanding of the IT infrastructure leads to more efficient management, cost-effective deployments, optimized decision making and, in the long run, enhanced QoS.

OntoMon follows a multi-tier architecture that integrates dedicated subsystems to separate concerns and employs various scalable open-source technologies to add functionality. In order to facilitate administration of diverse systems, OntoMon collects and aggregates performance metrics from both hardware and software assets in real time and performs respective threshold-based checks. Besides, it delivers a versatile, fully-dynamic and highly customizable User Interface that provides a summarized view of the monitored infrastructure and demonstrates its current state of operation. As a proof-of-concept, we studied two different use cases: the first use case targets the hardware assets housed in a Data Center, while the second one concentrates on a software-defined distributed multi-node storage cluster.

Keywords

Ontologies, DCIM, abstract object model, distributed monitoring, time-series, web user interface, real-time visualization, programmatic control, SVG, Icinga, Influx, Grafana, Angular

Preface

Σε αυτό το σημείο θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου Νεκτάριο Κοζύρη για τα ερεθίσματα, την έμπνευση, αλλά και τη δυνατότητα που μου προσέφερε να ασχοληθώ με τον τομέα των Υπολογιστικών Συστημάτων στη διπλωματική μου εργασία. Στη συνέχεια, οφείλω να ευχαριστήσω ιδιαιτέρως τους Βαγγέλη Κούκη και Κωνσταντίνο Βενετσανόπουλο για την άμεση ανταπόκρισή τους, την εμπιστοσύνη που μου έδειξαν και την άρτια συνεργασία μας κατά τη διάρκεια της συγγραφής της παρούσας διπλωματικής. Δίχως τις τεχνικές τους υποδείξεις, τις σχεδιαστικές ιδέες, αλλά και τη γενικότερη στήριξή τους αυτό το διάστημα η εκπόνηση της εργασίας αυτής δε θα ήταν εφικτή. Τέλος, ευχαριστώ θερμά πρωτίστως την οικογένειά μου και ακόλουθα τους φίλους μου για την αγάπη και την αμέριστη συμπαράστασή τους, υλική και ψυχολογική, καθ' όλα τα χρόνια της ακαδημαϊκής μου πορείας.

Ιωάννης Ανδρουλιδάκης
Ιούνιος 2017

Contents

List of figures

List of tables

1 Εισαγωγή

1.1 Σκοπός

Ο σκοπός της παρούσας εργασίας είναι ο σχεδιασμός και η υλοποίηση του *OntoMon*, μιας πολύπλευρης πλατφόρμας η οποία ενσωματώνεται σε ετερογενή υπολογιστικά συστήματα προσανατολισμένα είτε στο υλικό, είτε στο λογισμικό, με σκοπό την παροχή υπηρεσιών παρακολούθησης και οπτικοποίησης γενικού σκοπού σε πραγματικό χρόνο. Το πλαίσιο λογισμικού που περιγράφουμε είναι αφηρημένο ως προς το περιεχόμενο, καθώς υποστηρίζει διαφορετικού τύπου, ως προς τη σημασιολογία, υπολογιστικά συστήματα, οι οντότητες των οποίων περιγράφονται από μια καλώς ορισμένη Οντολογία. Επιπρόσθετα, υποστηρίζει μια πλήρως δυναμική και προσαρμόσιμη στις ανάγκες του χρήστη διαδικτυακή Διεπαφή, προκειμένου να διευκολύνει την επίβλεψη και τη διαχείριση υποδομής σε Κέντρα Δεδομένων μεγάλης κλίμακας.

1.2 Διατύπωση Προβλήματος-Κίνητρο

Στην εποχή μας, η ραγδαία αύξηση του ρυθμού παραγωγής ψηφιακής πληροφορίας έχει οδηγήσει στην επέκταση της φυσικής υποδομής των Κέντρων Δεδομένων, αλλά και στη σχεδίαση εξειδικευμένου λογισμικού, προκειμένου να είναι δυνατή η αποδοτική αποθήκευση και επεξεργασία δεδομένων μεγάλου όγκου(Big Data). Η επικράτηση των κατανεμημένων συστημάτων, των περιβαλλόντων εικονικοποίησης, αλλά και των υπηρεσιών νέφους σε υπολογιστικές πλατφόρμες μεγάλης κλίμακας έχει καταστήσει αναγκαία τη διαρκή παρακολούθηση και δειγματοληψία των δομικών τους μονάδων τόσο σε επίπεδο υλικού, όσο και λογισμικού, στοχεύοντας στην υψηλή απόδοση και την παροχή ποιοτικών υπηρεσιών στους χρήστες. Τα σημερινά Κέντρα Δεδομένων είναι πιο “πυκνά” από ποτέ, φιλοξενώντας ένα τεράστιο εύρος υπολογιστικών συστημάτων. Κατά συνέπεια, βάβες ή αστοχίες, σημεία συμφόρησης, καθώς και υπολειτουργία επιμέρους τμημάτων είναι συχνά φαινόμενα που αφορούν τη διαχείριση της υποδομής ενός Κέντρου Δεδομένων. Ακόλουθα, η κοινότητα Τεχνολογιών Πληροφορικής(IT) ήδη επενδύει στην ανάπτυξη μιας νέας κατηγορίας λογισμικού, γνωστή ως DCIM [1], η οποία περιλαμβάνει ένα σύνολο εργαλείων και υπηρε-

[1]Data Center Infrastructure Management

σιών για τη Διαχείριση Υποδομής Κέντρων Δεδομένων. Οι DCIM λύσεις πλέον αποτελούν αναπόσπαστο μέρος της στρατηγικής των IT επιχειρήσεων, καθώς οδηγούν σε βελτίωση της απόδοσης, αύξηση της ευελιξίας και καλύτερο έλεγχο των εξόδων. Συγκεκριμένα, τα DCIM εργαλεία παρέχουν αναλύση της απόδοσης των επιμέρους δομών ενός συστήματος σε πραγματικό χρόνο, απομόνωση υπο-συστημάτων για τον άμεσο εντοπισμό δυσλειτουργιών, ενώ συντελούν στη λήψη ποιοτικότερων αποφάσεων μέσω της ουσιώδους κατανόησης της συμπεριφοράς του συστήματος. Παράλληλα, ορισμένα από αυτά υποστηρίζουν οπτικοποίηση του συστήματος που παρακολουθούν, διαγράμματα, αλλά και ειδοποιήσεις πραγματικού χρόνου στο χρήστη σχετικά με συμβάντα εντός του συστήματος υπό παρακολούθηση. Όπως γίνεται αντιληπτό, η γραφική αναπράσταση των μετρικών απόδοσης του "συστήματος-στόχου", σε συνδυασμό με την απεικόνιση της εσωτερικής του τοπολογίας, οδηγούν σε μια εύληπτη και περιληπτική παρουσίασή του, καθώς και σε ένα πιο αξιόπιστο και βιώσιμο περιβάλλον λειτουργίας.

1.3 Υπάρχουσες Λύσεις & Εναλλακτική Προσέγγιση

Τα τελευταία χρόνια έχουν αναπτυχθεί πολυάριθμες DCIM υπηρεσίες, οι περισσότερες από τις οποίες κρίνονται ικανοποιητικές ως προς το τελικό αποτέλεσμα και τις δυνατότητες που προσφέρουν στον τελικό χρήστη. Ωστόσο, θεωρούμε πως υπάρχουν σημαντικά περιθώρια βελτιώσης κυρίως ως προς τη λογική της σχεδίασής τους, αλλά και τη μοντελοποίηση των συστημάτων στα οποία ενσωματώνονται. Πιο συγκεκριμένα, τα περισσότερα από τα υπάρχοντα DCIM εργαλεία στην αγορά είναι ιδιόκτητα και κλειστού κώδικα, γεγονός που αντιβαίνει στις σύγχρονες πρακτικές ανάπτυξης λογισμικού. Οι χρήστες εξαρτώνται από τους παρόχους, ενώ το σύστημα παρακολούθησης εγκαθιστάται ως "μαύρο κουτί", με την έννοια ότι δεν είναι δυνατή η πλήρης αντίληψη των εσωτερικών μηχανισμών λειτουργίας του. Ασφαλώς, κάτι τέτοιο δεν ευνοεί την προσαρμοστικότητα και την ευελιξία ως προς τις ανάγκες του χρήστη. Από την άλλη πλευρά, οι αντίστοιχες υλοποιήσεις ανοικτού κώδικα βασίζονται σε παλαιωμένες τεχνολογίες, ενώ αρκετές από αυτές δεν ανατπύσσονται ενεργά, με ελάχιστες εξαιρέσεις.

Επιπρόσθετα, τα περισσότερα εργαλεία διαχείρισης και παρακολούθησης Κέντρων Δεδομένων υιοθετούν συντηρητικές προσεγγίσεις αναφορικά με την αναπαράσταση των επιμέρους δομών ενός συστήματος μεγάλης κλίμακας. Αυτές συνήθως περιλαμβάνουν το σαφή ορισμό αντικειμένων στατικού τύπου, τα οποία συνήθως ανήκουν σε κάποια προκαθορισμένη κλάση και διατηρούν συγκεκριμένες ιδιότητες. Αυτό απορρέει από το γεγονός ότι οι υπάρχουσες DCIM λύσεις είναι σχεδιασμένες να παρακολουθούν ένα συγκεκριμένο σύστημα-στόχο, γνωρίζοντας εκ των προτέρων την εσωτερική του οργάνωση, καθώς και εξειδικευμένες πληροφορίες για τα δομικά του στοιχεία. Η εν λόγω προσέγγιση αποκλείει την επαναχρησιμοποίηση και τη διαλειτουργικότητα τμημάτων κώδικα, ενώ προϋποθέτει ειδικές μεθόδους χειρισμού και ελέγχου, προσαρμοσμένες στον τύπο του εκάστοτε αντικειμένου. Ως εκ τούτου, προάγεται μια αρχιτεκτονική βασισμένη σε κλάσεις και αυστηρώς τυποποιημένα αντικείμενα με δεδομένα χαρακτηριστικά, που όμως βρίσκουν εφαρμογή μόνο στο εν λόγω σύστημα-στόχο. Έτσι, περιορίζεται σημαντικά το εύρος των συστημάτων στα οποία μπορεί να ενσωματωθεί μια κοινή πλατφόρμα παρακολούθησης και οπτικοποίησης, χωρίς να πραγματοποιηθεί η χρονοβόρα επέκτασή της. Αντίθετα, το OntoMon μοντελοποιεί τις οντότητες και τις σχέσεις του συστήματος-στόχου σε αφηρημένα, ως προς το περιεχόμενο, και επαρκώς καθορισμένα, ως προς τη δομή, αντικείμενα, υλοποιώντας γενικού σκοπού χειριστές και μηχανισμούς επεξεργασίας που δεν εξαρτώνται από τη σημασιολογία. Ακολούθως, βασίσαμε την ανάπτυξη του OntoMon σε μαθηματικές Οντολογίες. Με δεδομένο ότι στόχος μας είναι παρακολούθηση πολυδιάστατων συστημάτων, θεωρούμε πως η χρήση οντολογιών διευκολύνει και εξυπηρετεί την περιγραφή ετερογενών συστημάτων-στόχων με εντελώς διαφορετικά χαρακτηριστικά, σε μια ενιαία και σαφώς ορισμένη δομή που προσαρμόζεται στις εκάστοτε ανάγκες.

Ακόμη, μελετώντας διάφορα λογισμικά για τη διαχείρηση υποδομής Κέντρων Δεδομένων, καταλήξαμε στο ότι αρκετά από αυτά έχουν υπερβολικά σύνθετη δομή, εξαρτήσεις και πολυπλοκότητα σχεδίασης. Μάλιστα, οι προσφερόμενες διαπροσωπείες χρήστη(UI) είναι αυστηρά καθορισμένες εκ των προτέρων ως προς τη δομή, βασιζόμενες στη στατική αναπαράσταση του συστήματος-στόχου, γεγονός που περιορίζει σημαντικά την εκφραστικότητα και την ευελιξία. Για το λόγο αυτό, η αρχιτεκτονική της προτεινόμενης πλατφόρμας παρακολούθησης αποτελείται από 3 διακριτά στρώματα, καθένα από τα οποία εσωκλείει ανεξάρτητες επιμέρους μονάδες που είναι επιφορτισμένες τη διεκπεραίωση σαφώς καθορισμένων λειτουργιών. Ενσωματώσαμε αρκετές τεχνολογίες ανοικτού κώδικα με στόχο την κλιμακωσιμότητα και την υψηλή επίδοση, ενώ φροντίσαμε όλες οι εσωτερικές επικοινωνίες μεταξύ των διαφορετικών επιπέδων να πραγματοποιούνται αποτελεσματικά, μέσω καθαρών `JSON` APIs. Παράλληλα, το UI που υποστηρίζει το OntoMon είναι πλήρως δυναμικό, κατασκευάζεται την ώρα της εκτέλεσης με βάση το εκάστοτε σύστημα-στόχο και είναι επαρκώς προσαρμόσιμο στις εκάστοτε ανάγκες του χρήστη ως προς την οπτικοποίηση της παρακολουθούμενης υποδομής και τη μορφή των ειδοποιήσεων.

1.4 Εφαρμογή-Έλεγχος

Προκειμένου να ελέγξουμε τη αποτελεσματικότητα της πλατφόρμας που αναπτύξαμε, προσομοιώσαμε τη λειτουργία ενός πρότυπου Κέντρου Δεδομένων. Συγκεκριμένα, δημιουργήσαμε ένα περιβάλλον ελέγχου μέσα στο οποίο εγκαταστήσαμε μια συστοιχία εικονικών υπολογιστών αποτελούμενη από 3 κόμβους. Για να αποδείξουμε ότι το OntoMon πράγματι μπορεί να παρακολουθήσει ετερογενή συστήματα, μελετήσαμε 2 διαφορετικά συστήματα-στόχους, τα οποία περιγράφηκαν από αντίστοιχες Οντολογίες που δόθηκαν ως είσοδος στην πλατφόρμα μας: στο πρώτο σενάριο παρακολουθήσαμε τη φυσική υποδομή ενός Κέντρου Δεδομένων, ενώ στο δεύτερο εστιάσαμε στις οντότητες επιπέδου λογισμικού ενός κλιμακώσιμου κατανεμημένου συστήματος αποθήκευσης δεδομένων. Και στις 2 περιπτώσεις παρεμβήκαμε στην κανονική λειτουργία των συστημάτων-στόχων, επιβάλλοντας υπολογιστικά φορτία και προκαλώντας προβλήματα συντονισμού. Ακολούθως, παρατηρήσαμε τις αυτόματες οπτικές ανανεώσεις στη Διεπαφής Χρήστη του OntoMon, καθώς και τις σχετικές ειδοποιήσεις σε πραγματικό χρόνο, καταδεικνύοντας την αλλαγή της κατάστασης λειτουργίας του συστήματος-στόχου.

2 Σχεδίαση

2.1 Σκεπτικό Σχεδίασης

Τα σύγχρονα υπολογιστικά συστήματα είναι πιο σύνθετα και πολύπλευρα από ποτέ, συνεπώς θεωρήσαμε απαραίτητο η πλατφόρμα μας να προσαρμόζεται εύκολα και αποτελεσματικά σε διαφορετικά ως προς το αντικείμενο συστήματα-στόχους. Συνεπώς, απορρίψαμε την προσέγγιση των αμετάβλητων αντικειμένων και δομών, καθώς και την επιβολή αυστηρών τύπων και σημασιολογίας κατά τη μοντελοποίηση των οντοτήτων υπό παρακολούθηση. Αντίθετα, εστιάσαμε την προσοχή μας στη σχεδίαση μιας αρθρωτής αρχιτεκτονικής, η οποία περιλαμβάνει ορισμένες βασικές δομικές μονάδες για τον καθορισμό του βασικού "σκελετού" της, καθώς και ορισμένες προσαρμόσιμες μονάδες για την εξυπηρέτηση των αναγκών διαχείρησης του εκάστοτε συστήματος-στόχου. Κύριος άξονας της σχεδίασής μας είναι η απαγκίστρωση από το περιεχόμενο και τη σημασιολογία του παρακολουθούμενου συστήματος, δηλαδή η επίτευξη διαλειτουργικότητας. Ακόλουθα, προχωρήσαμε στην υλοποιήση μιας πολύπλευρης και ταυτόχρονα ενιαίας πλατφόρμας παρακολούθησης για την υποβοήθηση της διαχείρισης διαφορετικών υπολογιστικών συστημάτων. Τα σημαντικότερα χαρακτηριστικά της προτεινόμενης σχεδίασης, καθώς και οι θεμελιώδεις αρχές που ακολουθήσαμε συνοψίζονται παρακάτω:

- **Ενσωμάτωση σε ετερογενή υπολογιστικά συστήματα:** το OntoMon είναι σχεδιασμένο ώστε να παρέχει υπηρεσίες παρακολούθησης και απεικόνισης διαφορετικών συστημάτων-στόχων χρησιμοποιώντας την ίδια βάση κώδικα, κάνοντας προσαρμογές μόνο σε συγκεκριμένα τμήματά του. Για το λόγο αυτό, έπρεπε να βρούμε ένα βολικό και ταυτόχρονα συνεπή τρόπο για την περιγραφή των διαφόρων συστημάτων-στόχων. Ως εκ τούτου, αποφασίσαμε να χρησιμοποιήσουμε μια Οντολογία η οποία θα λειτουργεί ως σημείο αναφοράς και θα ορίζει τις οντότητες, τις σχέσεις και την εσωτερική δομή του συστήματος-στόχου. Η οντολογική αυτή περιγραφή αποτελεί τη βάση πάνω στην οποία τελικά δομούνται όλα τα επίπεδα της πλατφόρμας μας. Για την ευκολότερη διαχείριση και επαλήθευση της Οντολογίας προχωρήσαμε στον καθορισμό ενός *οντολογικού σχήματος*, το οποίο περιλαμβάνει επαρκείς ιδιότητες για την πλήρη περιγραφή της υποδομής του εκάστοτε συστήματος-στόχου.

- **Αφηρημένη μοντελοποίηση:** σε συμφωνία με τη γενικής φύσεως οντολογική του βάση, σχεδιάσαμε το OntoMon με τέτοιον τρόπο, ώστε να αναπαριστά τις πραγματικές οντότητες υλικού και λογισμικού με τον πλέον αφηρημένο και ευέλικτο τρόπο: *αντικείμενα*. Η μοντελοποίηση των εννοιών του συστήματος-στόχου με αφηρημένα ως προς το περιεχόμενο, αλλά σαφώς καθορισμένα ως προς την εσωτερική δομή, αντικείμενα μας επέτρεψε να υλοποιήσουμε μεθόδους και χειριστές γενικού σκοπού που αντιμετωπίζουν όλα τα αντικείμενα με τον ίδιο ακριβώς τρόπο, ανεξάρτητα από τη σημασιολογία τους στον πραγματικό κόσμο. Τα αντικείμενα στο OntoMon είναι επεκτάσιμα και διαλειτουργικά, καθώς δεν ορίζονται συγκεκριμένοι τύποι ή κλάσεις αντικειμένων. Ακολούθως, όλες οι εσωτερικές επικοινωνίες μεταξύ των δομικών μερών της πλατφόρμας μας πραγματοποιούνται με τη λογική της δοσοληψίας "γενικού-τύπου" αντικειμένων και βασίζονται σε APIs που επιβάλλουν αναπαράσταση της πληροφορίας σε μορφή `JSON`.

- **Πολυχρηστικότητα & Προσαρμοστικότητα:** μία από τις σημαντικότερες επιδιώξεις του OntoMon είναι η προσαρμοστικότητα όλων των επιπέδων του στο εκάστοτε σύστημα-στόχος. Η προτεινόμενη σχεδίαση ενθαρρύνει την παρέμβαση του τελικού χρήστη κατά την αρχικοποίηση, υποστηρίζοντας την επέκταση της δομής των αντικειμένων, τον ορισμό εξειδικευμένων υπηρεσιών ελέγχου ή δειγματοληψίας, καθώς και πλήρη έλεγχο επί της οπτικοποίησης της παρακολουθούμενης υποδομής μέσα στο UI. Δεδομένου ότι η Διεπαφή Χρήστη του OntoMon βρίσκεται στην κορυφή της αρχιτεκτονικής και χτίζεται δυναμικά κατά το χρόνο εκτέλεσης, οι χρήστες μπορούν να κατασκευάσουν εξατομικευμένες ειδοποιήσεις και να ελέγξουν σε μεγάλο βαθμό τη συμπεριφορά των οπτικών στοιχείων της εφαρμογής, μέσω αυτοματοποιημένων μηχανισμών.

- **Κλιμακωσιμότητα & Ανθεκτικότητα Τεχνολογιών:** κατά την ανάπτυξη του OntoMon θέσαμε ως προτεραιότητα την ενσωμάτωση ποικίλων τεχνολογιών ανοικτού-κώδικα, οι οποίες να αποδίδουν ικανοποιητικά σε μεγάλη κλίμακα, προκειμένου να υποστηρίζονται συστήματα-στόχοι μεγάλου μεγέθους και πολύπλοκης δομής. Παράλληλα, επιθυμούμε η πλατφόρμα μας να είναι συγχρονισμένη με τις πιο πρόσφατες εξελίξεις στον τομέα της ανάπτυξης λογισμικού, γι'αυτό φροντίσαμε να χρησιμοποιήσουμε ανοικτά πρότυπα, όπως το `JSON` και το `SVG`, τα οποία εξασφαλίζουν συμβατότητα με πολυάριθμα εξωτερικά συστήματα Τεχνολογιών Πληροφορικής. Με αυτόν τον τρόπο, υποστηρίζουμε έμπρακτα τις αντίστοιχες κοινότητες και αποκλείουμε το ενδεχόμενο εξάρτησης από ιδιωτικούς παρόχους.

- **Πολυεπίπεδη Αρχιτεκτονική:** βασιζόμενοι στον προσδιορισμό ανεξάρτητων δομικών μονάδων και στοχεύοντας τόσο στο διαχωρισμό των αρμοδιοτήτων, όσο και στη διευκόλυνση της μελλοντικής συντήρησης της πλατφόρμας μας, προτείνουμε μια αρχιτεκτονική η οποία χωρίζεται σε ξεχωριστά επίπεδα οργάνωσης ανάλογα με την επιτελούμενη λειτουργία. Πιο συγκεκριμένα, κάθε στρώμα περιλαμβάνει πολυάριθμες διακριτές μονάδες, καθεμιά από τις οποίες είναι επιφορτισμένη με την παροχή συγκεκριμένων υπηρεσιών στο συνολικό σύστημα. Τα διαφορετικά στρώματα αλληλεπιδρούν με καλώς ορισμένα πρωτόκολλα επικοινωνίας τα οποία επιτρέπουν την απρόσκοπτη ροή δεδομένων από τα χαμηλότερα στα υψηλότερα στρώματα, και αντίστροφα. Η αποκεντρικοποιημένη αυτή προσέγγιση επιτρέπει την παραλληλοποίηση της ανάπτυξης της πλατφόρμας μας, επιταχύνει τον εντοπισμό σφαλμάτων και απλοποιεί την επίλυσή τους.

2.2 Διαχωρισμός Ρόλων

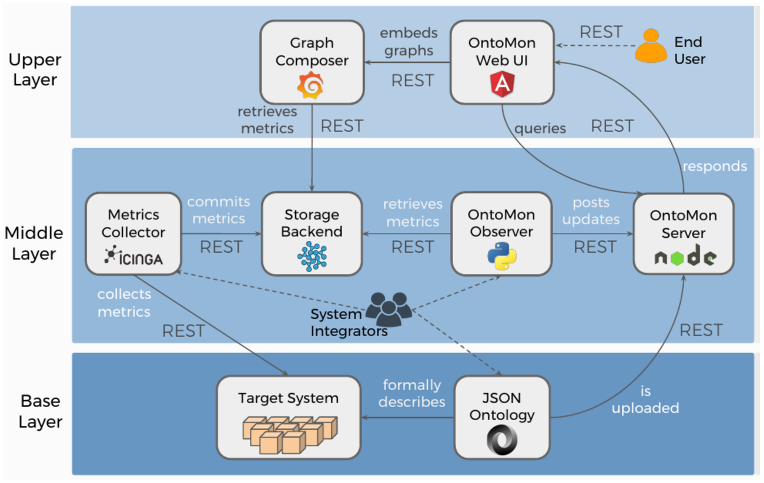
Σε αυτό το σημείο είναι χρήσιμο να επισημάνουμε τους ρόλους των χρηστών που αλληλεπιδρούν με την πλατφόρμα μας. Λόγω της κατανεμημένης σχεδίασης και της προσαρμοστικότητάς του, το OntoMon δεν προορίζεται για απευθείας χρησιμοποίηση από τον τελικό χρήστη, αλλά απαιτεί τη συμβολή ενός *προγραμματιστή ενσωμάτωσης συστημάτων* κατά την αρχική του ρύθμιση, προκειμένου να προσαρμοστεί στο εκάστοτε σύστημα-στόχο και να μπορέσει να εξυπηρετήσει αποτελεσματικά τις διαχειριστικές ανάγκες του τελικού χρήστη. Πιο συγκεκριμένα:

- **Προγραμματιστής Ενσωμάτωσης Συστημάτων:** πρόκειται για ένα άτομο ή ομάδα προγραμματιστών που αναλαμβάνει το συντονισμό και τη ρύθμιση όλων των επιμέρους τμημάτων της πλατφόρμας μας με σκοπό την παροχή εξειδικευμένων υπηρεσιών παρακολούθησης και οπτικοποίσης ενός συγκεκριμένου συστήματος-στόχου. Ο προγραμματιστής ενσωμάτωσης δουλεύει σε στενή συνεργασία με τον τελικό χρήστη προκειμένου να κατανοήσει την εσωτερική δομή του συστήματος-στόχου και να διαδόσει τη γνώση αυτή στην πλατφόρμα μας. Στόχος του είναι η πιστή αναπαράσταση των υποδομής προς παρακολούθηση, ο σαφής καθορισμός των υπηρεσιών διαχείρισης και η υλοποίηση της πολιτικής παρακολούθησης και του μηχανισμού ειδοποιήσεων, με βάση τις εκάστοτε ανάγκες. Όπως γίνεται αντιληπτό, κάτι τέτοιο απαιτεί τη βαθιά κατανοήση της αρχιτεκτονικής και των λειτουργιών του εργαλείου μας, ώστε να πραγματοποιθεί η παρέμβαση σε συγκεκριμένα δομικά μέρη.

- **Τελικός Χρήστης:** συνήθως πρόκειται για το διαχειριστή του συστήματος-στόχου ή ενός full-stack μηχανικού υπέυθυνου για την επίβλεψη της απόδοσης ολόκληρης της υποδομής. Υπό αυτό το πρίσμα, ο τελικός χρήστης ενημερώνει τον προγραμματιστή ενσωμάτωσης συστημάτων για τις υπηρεσίες παρακολούθησης και οπτικοποίησης που επιθυμεί από την πλατφόρμα μας. Τελικά, η αλληλεπίδρασή του με το OntoMon εστιάζεται αποκλειστικά στην πρόσβαση στη διαδικτυακή Διεπαφή Χρήστη για την επίβλεψη και τη διαχείριση του συστήματος-στόχου.

2.3 Αρχιτεκτονική

Σε αυτήν την ενότητα παρουσιάζουμε αναλυτικά τις δομικές αρχιτεκτονικές μονάδες του OntoMon, καθώς και τις σημαντικότερες σχεδιαστικές αποφάσεις που λάβαμε κατά τη διαδικασία ανάπτυξης της πλατφόρμας μας. Το OntoMon είναι ένα ιεραρχικά δομημένο πλαίσιο λογισμικού, το οποίο αποτελείται από 3 διακριτά στρώματα που αλληλεπιδρούν μεταξύ τους. Καθένα από τα στρώματα αυτά αντιπροσωπεύει μια διαφορετική άποψη της πλατφόρμας μας και, ακόλουθα, εσωκλείει διάφορες οντότητες λογισμικού που είναι υπέυθυνες για τη διεκπεραίωση συγκεκριμένων λειτουργιών. Προκειμένου να δώσουμε στον αναγνώστη μια πρώτη διαίσθηση σχετικά με τη δομή του OntoMon, παραθέτουμε μια υψηλού επιπέδου παρουσίασή της στο ακόλουθο σχήμα:



Σχήμα 1: *Στρώματα Αρχιτεκτονικής του OntoMon*

Όπως φαίνεται, η ροή των δεδομένων ξεκινά από το βασικό στρώμα, περνά στο μεσαίο στρώμα και καταλήγει στο υψηλό, όπου τελικά λαμβάνει χώρα η αλληλεπίδραση με τον τελικό χρήστη.

Βασικό Στρώμα

Το βασικό στρώμα καθορίζει και διατυπώνει την είσοδο για την πλατφόρμα μας. Αποτελείται από το Σύστημα-Στόχο, το οποίο ορίζεται ως ένα σύνολο από επιμέρους οντότητες οι οποίες αλληλεπιδρούν και συνεργάζονται για την επίτευξη ενός στόχου, και την Οντολογία, η οποία είναι μια επίσημη, καλά δομημένη και λεπτομερής περιγραφή της υποδομής και των επιμέρους σχέσεων που ορίζονται εντός του συστήματος-στόχου. Στην ουσία πρόκειται για μια σαφώς καθορισμένη αναπαράσταση των εννοιών του πεδίου ενδιαφέροντος, η οποία αντιστοιχίζει τα πραγματικά δομικά στοιχεία του συστήματος-στόχου σε εικονικά αντικείμενα εντός του OntoMon.

Σύστημα-Στόχος: Το πραγματικό υπολογιστικό σύστημα το οποίο παρακολούθει και οπτικοποιεί η πλατφόρμα μας. Υπεύθυνος για την οργάνωση και τη σχεδίασή του είναι, συνήθως, ο τελικός χρήστης. Το ακριβές αντικείμενο και οι υπηρεσίες του εν λόγω συστήματος μπορεί να ποικίλουν ανάλογα με το πεδίο εφαρμογής του, ωστόσο υποθέτουμε ότι σε κάποιο επίπεδο περιλαμβάνει υπολογιστές(servers), προκειμένου να καταστεί εφικτή η δειγματοληψία της επίδοσής τους από το μεσαίο στρώμα. Κάθε σύστημα-στόχος αποτελείται από πολυάριθμες οντότητες(assets) και τις μεταξύ τους διασυνδέσεις(relations). Συγκεκριμένα, μπορεί να είναι προσανατολισμένο στο επίπεδο του λογισμικού, όπως ένα κατανεμημένο σύστημα αποθήκευσης δεδομένων, στο επίπεδο του υλικού, όπως οι συσκευές και οι διατάξεις στο εσωτερικό ενός Κέντρου Δεδομένων, ή και στα δύο ταυτόχρονα. Η τελευταία συνδυαστική προσέγγιση περιγράφει ένα υβριδικό σύστημα, του οποίου η διαχείριση απαιτεί την παρακολούθηση σε όλα τα επίπεδα, από άκρη σε άκρη. Σε τέτοιες περιπτώσεις, η πλατφόρμα μας προσφέρει στον τελικό χρήστη τη δυνατότητα να συσχετίσει ρυθμίσεις επιπέδου υλικού με αντίστοιχες μετρικές απόδοσεις επιπέδου λογισμικού, ώστε να εξάγει χρήσιμα συμπεράσματα σχετικά με τις εσωτερικές αλληλεπιδράσεις του συστήματος-στόχου. Ενδεικτικά, κατανεμημένα συστήματα λογισμικού αλλά και συστοιχίες υπολογιστικών κόμβων μεγάλης κλίμακας αποτελούν αντιπροσωπευτικά παραδείγματα συστημάτων-στόχων στα οποία το OntoMon μπορεί να ενσωματωθεί αποτελεσματικά.

Οντολογία: Πρόκειται για μια αυστηρή και αυστηρά δομημένη οντολογική περιγραφή του συστήματος-στόχου, η οποία δίνεται ως είσοδος στην πλατφόρμα μας και αποτελεί τη βάση για τη λειτουργία της. Αρχικά, το OntoMon ελέγχει την εγκυρότητα της Οντολογίας, τη διαβάζει και στη συνέχεια ρυθμίζει τις σχετικές λειτουργίες παρακολούθησης και οπτικοποίησης. Προκειμένου να προτυποποιηθεί η δομή της εν λόγω Οντολογίας εισάγαμε ένα *οντολογικό σχήμα*, το οποίο μπορεί να θεωρηθεί ως το “αποτύπωμα” κάθε συστήματος-στόχου. Το σχήμα αυτό πρέπει να προσφέρει ένα ενιαίο και ταυτόχρονα διαλειτουργικό μηχανισμό για τη λεπτομερή περιγραφή διαφορετικών συστημάτων-στόχων. Συνεπώς, αποφασίσαμε να χρησιμοποιήσουμε το πρότυπο **`JSON`** για την υλοποίηση της εκάστοτε Οντολογίας, καθώς συνδυάζει τα χαρατηριστικά που επιζητάμε. Συγκεκριμένα, πρόκειται για ένα ανοικτό, ελαφρύ και ευρέως χρησιμοποιούμενο πρότυπο αναπαράστασης δεδομένων, το οποίο είναι αναγνώσιμο τόσο από τον άνθρωπο, όσο και από τη μηχανή. Στην πραγματικότητα είναι μια δομή “κλειδιού-τιμής”, η οποία ειναι φορητή, εξ’ορισμού επεκτάσιμη και συμβατή με τις περισσότερες γλώσσες προγραμματισμού. Το γεγονός αυτό διευκολύνει την ανταλλαγή πληροφορίας ανάμεσα σε εφαρμογές, καθώς και τον ορισμό εμφωλευμένων δομών σε πολλά επίπεδα.

Όσον αφορά την πλατφόρμα μας, η Οντολογία μοντελοποιείται ως ένας πίνακας από `JSON` αντικείμενα, καθένα από τα οποία αντιπροσωπεύει ένα δομικό στοιχείο του πραγματικού συστήματος-στόχου.Επιλέξαμε όλα τα αντικείμενα να ορίζονται στο ίδιο επίπεδο, χωρίς φωλιάσματα, ενώ οι μεταξύ τους εξαρτήσεις εκφράζονται με μαθηματικές σχέσεις σε αντίστοιχες ιδιότητες. Έτσι, η προτεινόμενη δομή της Οντολογίας προσομοιάζει σε αυτήν μιας επίπεδης “πισίνας” αντικειμένων(object pool), προσφέροντας υψηλή απόδοση όταν πολλά αντικείμενα ανακτώνται για μικρό χρονικό διάστημα. Επιπρόσθετα, για λόγους συνάφειας και ομοιομορφίας, προσδώσαμε στα `JSON` αντικείμενα της Οντολογίας συγκεκριμένα χαρακτηριστικά. Πρώτον, κάθε αντικείμενο πρέπει να ταυτοποιείται μοναδικά. Δεύτερον, κάθε αντικείμενο “περιέχεται σε” κάποιο άλλο προκειμένου να αποτυπωθεί η ιεραρχική οργάνωση του συστήματος-στόχου. Το πρώτο είναι το αντικείμενο-παιδί, ενώ το δεύτερο είναι το αντικείμενο-πατέρας. Τρίτον, κάθε αντικείμενο αντιστοιχίζεται σε μια προκαθορισμένη από τους χρήστες οπτική αναπαράσταση(αρχείο εικόνα), ώστε τελικά να απεικονιστεί από τη Διεπαφή Χρήστη του OntoMon. Ακόμη, επιβάλλαμε ένα σύνολο κανόνων αναφορικά με την τοπολογική οργάνωση των αντικειμένων εντός της Οντολογίας, ώστε να εξασφαλίσουμε την επιθυμητή ιεραρχική δομή:

- Κάθε αντικείμενο έχει ακριβώς ένα πατέρα, που ανήκει στην ίδια Οντολογία
- Υπάρχει μοναδικό αντικείμενο με `null` πατέρα, το αντικείμενο-ρίζα
- Ο γράφος που προκύπτει πρέπει να είναι συνδεδεμένος, δηλαδή να υπάρχει μονοπάτι που να συνδέει οποιουσδήποτε 2 κόμβους εντός του

Η παραπάνω σχεδίαση οδηγεί σε μια ακυκλική και συνδεδεμένη τοπολογία, δηλαδή ένα δέντρο με μοναδική ρίζα και πολλούς ενδιάμεσους κόμβους. Το οντολογικό αυτό δέντρο αντιπροσωπεύει την ιεραρχία εντός της Οντολογίας και είναι ο ακρογωνιαίος λίθος του OntoMon, αφού περιέχει όλη την απαραίτητη πληροφορία για το σύστημα-στόχο. Τελικά, ο προγραμματιστής ενσωμάτωσης συστήματος "ανεβάζει" το *Ontology.json* αρχείο στον εξυπηρετητή του OntoMon μέσω μιας απλής `HTTP POST` αίτησης.

Μεσαίο Στρώμα

Το μεσαίο στρώμα φιλοξενεί τις λειτουργίες παρακολούθησης και υλοποιεί τις πολιτικές ελέγχου λαμβάνοντας αποφάσεις. Αποτελείται από τα 4 ξεχωριστά δομικά μέρη τα οποία είναι αλυσιδωτά συνδεδεμένα. Αρχικά, οι agents του συστήματος παρακολούθησης συλλέγουν μετρικές απόδοσης και τις αποθηκεύουν ως χρονοσειρές σε μια εξειδικευμένη βάση δεδομένων. Στη συνέχεια, ο Παρακολουθητής ανακτά τις μετρικές αυτές, τις επεξεργάζεται και συμπεραίνει αναφορικά με την τρέχουσα κατάσταση του συστήματος-στόχου. Τελικά, ενημερώνει τον Server(εξυπηρετητή) του OntoMon σχετικά, προκειμένου η πληροφορία να διαθοδεί στο στρώμα παρουσίασης.

Συλλέκτης Μετρικών Απόδοσης: Το πρώτο βήμα για την παρατήρηση του συστήματος-στόχου είναι η εγκατάσταση και ρύθμιση του συστήματος δειγματοληψίας. Το εργαλείο αυτό είναι καίριας σημασίας, καθώς είναι υπεύθυνο για τη διενέργεια περιοδικών ελέγχων επίδοσης των δομικών μερών του συστήματος-στόχου και τη συλλογή σχετικών μετρήσεων και στατιστικών. Ο προγραμματιστής ενσωμάτωσης συστήματος αναλαμβάνει τον ορισμό υπηρεσιών ελέγχου προσαρμοσμένων στις ανάγκες των διαχειριστών. Από τη στιγμή που το OntoMon καλείται να διαχειριστεί απαιτητικά υπολογιστικά συστήματα μεγάλης κλίμακας, δώσαμε αρκετή προσοχή στην επιλογή

ενός κατάλληλου μηχανισμού παρακολούθησης, ο οποίο να μπορεί να ρυθμιστεί κατάλληλα και να πετύχει υψηλές αποδόσεις.
Ακόλουθα, επιλέξαμε το Icinga, ένα εργαλείο παρακολούθησης σχεδιασμένο για τη συλλογή μετρικών απόδοσης σε κατανεμημένα περιβάλλοντα με πολλούς κόμβους. Προχωρήσαμε σε μια ιεραρχική οργάνωση της συστοιχίας του Icinga, αποτελούμενη από master και client agents(“πράκτορες”), καθώς και τις αντίστοιχες ζώνες μέσα στις οποίες δραστηριοποιούνται. Μάλιστα, σχετικά με τα αντικειμένα που ορίζονται μέσα στη συστοιχία του Icinga(`Zones`,`Endpoints`,`Objects`), επιλέξαμε συγχρονισμό από πάνω προς τα κάτω, ώστε να διατηρήσουμε όλες τις κοινές, για τους clients, ρυθμίσεις στις master ζώνες, και να τις στείλουμε μαζικά. Με αυτόν τον τρόπο μειώνουμε την πολυπλοκότητα, επιταχύναμε τη διαδικασία της προσαρμογής του Icinga στις ανάγκες του εκάστοτε συστήματος-στόχου, καθώς δεν απαιτείται σύνδεση σε κάθε client agent ξεχωριστά για τη ρύθμιση των υπηρεσιών δειγματοληψίας. Παράλληλα, οι client agents εγκαθιστώνται στους διάφορους servers του συστήματος-στόχου και δουλεύουν αποκεντρωμένα, εκτελώντας τοπικά τους ελέγχους και πετυχαίνοντας κατανομή του φορτίου εργασίας σε όλο το πλάτος της συστοιχίας του Icinga. Συγκεκριμένα, οι client agents λαμβάνουν οδηγίες από τους υπεύθυνους master agents σχετικά με τους ελέγχους απόδοσης που πρέπει να πραγματοποιήσουν. Μόλις ολοκληρώσουν τη δειγματοληψία στέλνουν τα αποτελέσματα πίσω στον υπεύθυνο master agent, ο οποίος τα συγκεντρώνει ανά κατηγορία. Με τη σειρά τους, οι master agents καταθέτουν τις τιμές των μετρικών απόδοσης σε μια εξειδικευμένη βάση δεδομένων χρονοσειρών, την InfluxDB, για μελλοντική επεξεργασία. Το Icinga είναι συμβατό και υποστηρίζει πολυάριθμα backend σχήματα αποθήκευσης, μέσω του `Writer Module` που παρέχει, πραγματοποιώντας κατάλληλες ρυθμίσεις.
Όσον αφορά την επικοινωνία, το Icinga χρησιμοποεί τα πρωτόκολλα που ορίζει το `HTTP` API της Influx προκειμένου να συνδεθεί σε πραγματικό χρόνο στον εξυπηρετητή της πάνω από το δίκτυο και να αποστείλει χρόνο- και μέτα-δεδομένα σχετικά με την απόδοση των δομικών μερών του συστήματος-στόχου. Πιο αναλυτικά, θέτει `HTTP` αιτήσεις τύπου `POST` στο άκρο που “ακούει” και αποδέχεται συνδέσεις ο influx-δαίμονας. Η επιλογή μας να ενσωματώσουμε ένα ήδη υπάρχον σύστημα συλλογής και δειγματοληψίας μετρικών απόδοσης αποσκοπεί τόσο στη μελέτη και την κατανόηση ενός πραγματικού συστήματος που χρησιμοποιείται στην παραγωγή, όσο και στην επίτευξη κλιμάκωσης σε σύνθετα περιβάλλοντα.

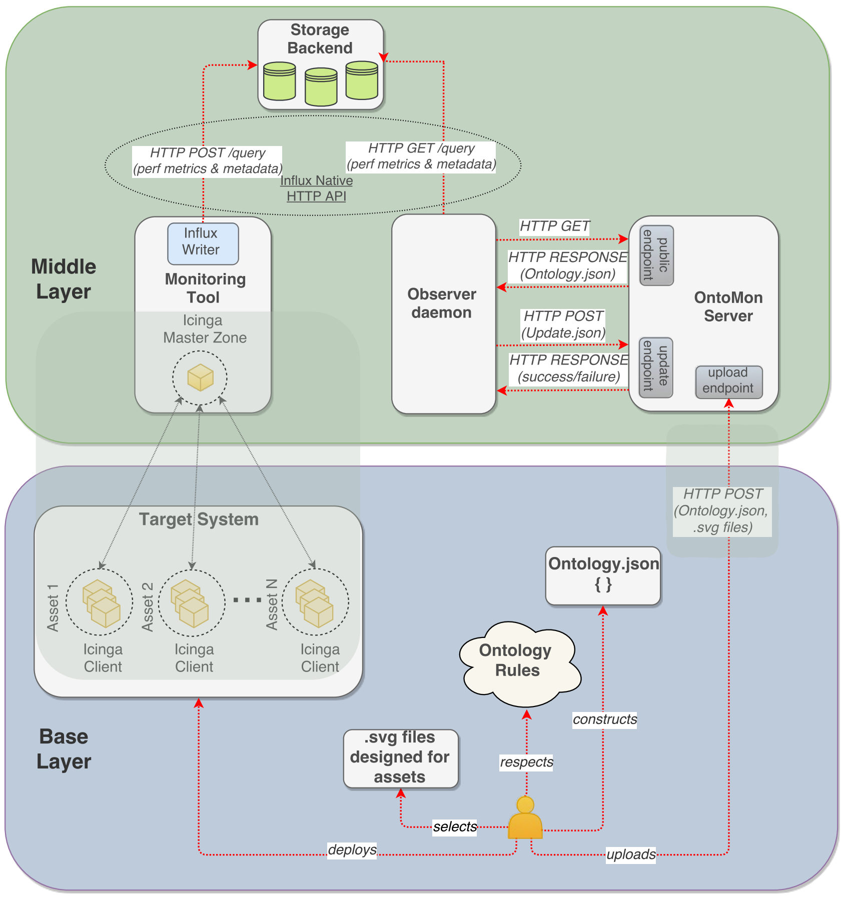
Αποθήκευση Χρονοδεδομένων Τα χρονοδεδομένα ορίζονται ως μια σειρά τιμών που δεικτοδοτούνται σε ίσα διαδοχικά χρονικά διαστήματα. Ακολούθως, μια βάση χρονοσειρών ορίζεται ως ένα εξειδικευμένο σύστημα λογισμικού το οποίο είναι σχεδιασμένο για τη βελτιστοποίηση της αποθήκευσης και διαχείρισης χρονοδεδομένων. Οι βάσεις χρονοσειρών υποστηρίζουν τις ανάγκες συσκευών ή υπηρεσιών που παράγουν δεδομένα σε διαρκή ροή. Εφόσον το σύστημα παρακολούθησης του OntoMon δειγματοληπτεί και συλλέγει μετρικές απόδοσης προερχόμενες από το σύστημα-στόχο σε τακτά χρονικά διαστήματα, έπρεπε να βρούμε έναν αποδοτικό τρόπο για την αποθήκευση και τη διατήρηση των χρονοδεδομένων αυτών για μελλοντική ανάλυση(πχ γραφήματα, ιστορική μελέτη, εντοπισμός μοτίβων κλπ). Για να επιλύσουμε το συγκεκριμένο ζήτημα αποφασίσαμε να ενσωματώσουμε στην πλατφόρμα μας της βάση χρονοσειρών InfluxDB, η οποία βασίζεται σε μια κατανεμημένη αρχιτεκτονική, προσφέρει υψηλή διαθεσιμότητα και μοιράζει το φόρτο εργασίας σε επιμέρους μονάδες, εφόσον εγκατασταθεί σε παραπάνω του ενός συνεργαζόμενους κόμβους. Για τις ανάγκες της παρούσας εργασίας ήταν αρκετό να εγκαταστήσουμε την InfluxDB σε έναν μόνο κόμβο, μειώνοντας την πολυπλοκότητα της τοπολογίας και πετυχαίνοντας μικρούς χρόνους απόκρισης. Συγκεκριμένα, προκειμένου να προσφέρει υψηλή επίδοση σε πολύπλοκα χρονοερωτήματα, η Influx βασίζει τους υπολογισμούς στις εξής παραδοχές: 1. οι λειτουργίες `delete` και `update` είναι περιορισμένες σε σχέση με την `write`, η οποία είναι η πιο συνήθης 2. τα χρονοδεδομένα αποθηκεύονται σε αύξουσα χρονολογική σειρά 3. σε καταστάσεις υψηλού φόρτου, δίνεται περισσότερη βαρύτητα στη διαθεσιμότητα από ότι στη συνέπεια.

Επιπρόσθετα, η Influx υποστηρίζει ένα εξειδικευμένο `HTTP` API πάνω σε δημοσίως γνωστά διαδικτυακά άκρα(πχ `/ping`,`/query`,`/write`), προκειμένου να διευκολύνει την απομακρυσμένη πρόσβαση των χρηστών στα χρονοδεδομένα που αποθηκεύει. Σε αυτήν της την ιδιότητα βασίσαμε τις περισσότερες επικοινωνίες που περιλαμβάνουν τη δοσοληψία μετρικών απόδοσης πραγματικού χρόνου. Ακόμη, παρόλο που η Influx είναι μια NoSQL βάση, εξυπηρετεί ερωτήματα σε μια προσαρμοσμένη γλώσσα ερωτημάτων, την InfluxQL. Η γλώσσα αυτή προσομοιάζει σε συντακτικό και δομή στην SQL, προσφέροντας ένα πιο οικείο περιβάλλον εργασίας. Η μορφοποίηση των χρονοδεδομένων περιλαμβάνει measurements, series, metrics, tags και άλλα πεδία, όπως αναλύεται παρακάτω.

OntoMon Παρακολουθητής Μια από τις πιο ουσιώδεις αποφάσεις που λάβαμε κατά το σχεδιασμό του OntoMon, ήταν η υλοποίηση ενός πλήρως ελεγχόμενου μηχανισμού παρακολούθησης και αποστολής ειδοποιήσεων, ορίζοντας προσαρμοσμένα APIs για την επικοινωνία. Συνεπώς, συμπεριλάβαμε στην αρχιτεκτονική μας μια ακόμη δομική μονάδα η οποία αναλαμβάνει να συμπεράνει την τρέχουσα κατάσταση των επιμέρους τμημάτων του συστήματος-στόχου με βάση τις πιο πρόσφατες μετρικές απόδοσης και, ακολούθως, να ενημερώσει το στρώμα οπτικοποίησης για την ανανέωση της Διεπαφής Χρήστη του OntoMon σε πραγματικό χρόνο. Πρέπει να σημειώσουμε, ότι υπάρχει σαφής διαχωρισμός μεταξύ των agents του Icinga και του Παρακολουθητή, καθώς οι πρώτοι απλώς συλλέγουν και εξάγουν σε πραγματικό χρόνο τις τιμές των μετρικών απόδοσης από το σύστημα-στόχο, ενώ ο δεύτερος είναι αφοσιωμένος στην ανάκτησή τους από τη βάση χρονοσειρών, την επεξεργασία τους, την πραγματοποίηση υπολογισμών, καθώς και τη βαθύτερη ανάλυση και αλληλολσυσχέτισή τους. Ο Παρακολουθητής του OntoMon είναι ένας δαίμονας υλοποιημένος σε `Python`, προσαρμόζεται στο εκάστοτε σύστημα-στόχο από τον προγραμματιστή ενσωμάτωσης και συγκεντρώνει όλη τη συλλογιστική και τις πολιτικές της παρακολούθησης που θέλει να επιβάλλει ο χρήστης. Πρακτικά, ο Παρακολουθητής συγκρίνει τις καταγραφόμενες μετρικές απόδοσης και τις ενδείξεις λειτουργίας των δομικών στοιχείων του συστήματος-στόχου με προκαθορισμένες σταθερές και στη συνέχεια δημοσιεύει σχετικές ενημερώσεις. Οι ενημερώσεις αυτές παράγονται περιοδικά και μπορούν να θεωρηθούν ως στιγμιότυπα του συστήματος-στόχου σε διαδοχικές χρονικές στιγμές. Περιέχουν αναλυτικές πληροφορίες και δεδομένα σχετικά με την τρέχουσα κατάσταση της παρακολουθούμενης δομικής μονάδας(asset), τις μετρικές απόδοσής της και την οπτικοποίηση της σχετικής ειδοποίησης εντός της Διεπαφής Χρήστη για την ενημέρωση του τελικού χρήστη. Κατά την αρχικοποίησή του, ο Παρακολουθητής απαιτεί τον προσδιορισμό των servers που πρόκειται να παρακολουθηθούν είτε σε επίπεδο λογισμικού, είτε υλικού, καθώς και τις αντίστοιχες υπηρεσίες ελέγχου. Η επικοινωνία μεταξύ του Παρακολουθητή και της βάσης χρονοσειρών του OntoMon πραγματοποιείται με συναρτήσεις-χειριστές οι οποίες είναι πλήρως συμβατές με το `HTTP` API της Influx. Για λόγους ομοιομορφίας, επιλέξαμε να μοντελοποιήσουμε τις παραγόμενες ενημερώσεις κατάστασης του συστήματος-στόχου σε μορφή `JSON` με συγκεκριμένες ιδιότητες. Δεδομένου ότι κάθε ενημέρωση αναφέρεται σε μια συγκεκριμένη δομική μονάδα του συστήματος-στόχου, ο Παρακολουθητής της πλατφόρμας μας πρέπει να γνωρίζει την αντιστοίχιση των δομικών μονάδων αυτών σε αντικείμενα της Οντολογίας. Κατά συνέπεια, ο Παρακολουθητής αλληλεπιδρά με τον Εξυπηρετητή του OntoMon ως εξής: αρχικά ανακτά το *Ontology.json* μέσω ενός `HTTP GET` ερωτήματος, στη συνέχεια πραγματοποιεί τους ζητούμενους ελέγχους και, τελικά, αποστέλλει ένα *Update.json* αντικείμενο μέσα στο σώμα ενός `HTTP POST` αιτήματος. Η σχεδίαση αυτή εξασφαλίζει πλήρη συμβατότητα με την οντολογική περιγραφή του βασικού στρώματος και εισάγει ένα προσαρμοσμένο API ειδοποιήσεων για την επικοινωνία με το ανώτερο στρώμα.

OntoMon Εξυπηρετητής Ο εξυπηρετητής του OntoMon είναι ένα από τα κεντρικότερα δομικά του μέρη, καθώς πραγματοποιεί συναλλαγές με πολυάριθμες οντότητες και διανέμει δεδομένα σε όλα τα στρώματα της πλατφόρμας μας. Είναι η δομική μονάδα η οποία "γνωρίζει" ανά πάσα στιγμή την κατάσταση του συστήματος-στόχου. Πρόκειται για ένα ελαφρύ, κλιμακώσιμο και πλήρως λειτουργικό κομμάτι εκτελέσιμου κώδικα(script), υλοποιημένο εξ'ολοκλήρου σε `JavaScript`. Ο OntoMon εξυπηρετητής γεφυρώνει τα κενά μεταξύ των δομικών μερών της πλατφόρμας μας και είναι υπέυθυνος τόσο για το χειρισμό `HTTP` ερωτημάτων προς αυτόν, όσο και για την αποστολή στατικών αρχείων, όταν αυτά ζητηθούν. Πιο συγκεκριμένα, "ακούει" και αποδέχεται συνδέσεις σε πολλαπλά διαδικτυακά άκρα ταυτόχρονα, υποστηρίζοντας επικοινωνία με πολλούς clients σε πραγματικό χρόνο. Αναφορικά με την επικοινωνία με το βασικό στρώμα, ο εξυπηρετητής του OntoMon κοινοποιεί ένα συγκεκριμένο διαδικτυακό άκρο στο οποίο ο προγραμματιστής ενσωμάτωσης συστημάτων μπορεί να αποστείλει τα *Ontology.json* και τα *.svg* αρχεία που αντιστοιχούν στα αντικείμενα της Οντολογίας μέσω ενός `HTTP POST` ερωτήματος.

Μόλις τα αρχεία αυτά "ανέβουν" επιτυχώς, αποθηκεύονται τοπικά στον εξυπηρετητή και διατείθενται σε ένα άλλο, επίσης ευρέως γνωστό για την πλατφόρμα μας διαδικτυακό άκρο, ώστε να ανακτηθούν μετέπειτα από τα δομικά μέρη τόσο του μεσαίου, όσο και του υψηλού στρώματος μέσω `HTTP GET` αιτημάτων. Για παράδειγμα, ο OntoMon Παρακολουθητής αλλά και η Διεπαφή Χρήστη του OntoMon χρειάζονται το *Ontology.json* αρχείο, ώστε να προσαρμόσουν τις λειτουργίες τους στο σύστημα-στόχο. Βάση σχεδίασης, ο OntoMon εξυπηρετητής κρατά πάντα τις πιο πρόσφατες εκδόσεις των αρχείων αυτών, καθιστώντας δυνατή την αλλαγή τους ακόμα και κατά το χρόνο εκτέλεσης. Τέλος, υλοποιήσαμε μια εξειδικευμένη συνάρτηση χειρισμού των *Update.json* αντικειμένων που αντιπροσωπεύουν τις ενημερώσεις κατάστασης, η οποία τα σειριοποιεί σε ουρά, διαβάζει τα επιμέρους πεδία τους και τα “σερβίρει” σε ένα τρίτο εύρεως γνωστό διαδικτυακό άκρο. Παράλληλα, καταγράφει όλες τις αιτήσεις ενημέρωσης κατάστασης λειτουργίας σε ένα *Update.log* αρχείο, παρέχοντας ένα ανθρωπίνως αναγνώσιμο ιστορικό της συνολικής συμπεριφοράς του συστήματος-στόχου στο χρόνο.



Σχήμα 2: *Επικοινωνία μεταξύ βασικού και μεσαίο στρώματος*

Υψηλό Στρώμα

Στο υψηλό στρώμα της πλατφόμας πραγματοποιείται η οπτικοποίηση του συστήματος-στόχου, καθώς και η συγκεντρωτική παρουσίαση της τρέχουσας κατάστασής του. Το στρώμα αυτό αποτελείται από δομικά μέρη τα οποία συνεργάζονται μεταξύ τους, ανακτούν χρονοδεδομένα από το μεσαίο στρώμα και προσφέρουν στον τελικό χρήστη μια γενική εικόνα του συστήματος υπό παρακολούθηση. Ουσιαστικά, πρόκειται για την τελική έξοδο του OntoMon, με την οποία αλληλεπιδρά απευθείας ο τελικός χρήστης. Πιο συγκεκριμένα, φιλοξενεί μια διαδικτυακή Διεπαφή Χρήστη και έναν εξειδικευμένο Συνθέτη Γραφημάτων χρονοσειρών.

Διεπαφή Χρήστη Αποτελεί τον πυρήνα του υψηλού στρώματος και εκφράζει σε μεγάλο βαθμό την ευρύτερη λογική σχεδίασης του OntoMon, εκμεταλλευόμενη όλη την επεξεργασία που πραγματοποιείται στα κατώτερα στρώματα. Σκοπός της είναι η παροχή υπηρεσιών επίβλεψης και διαχείρισης της παρακολουθούμενης υποδομής στον τελικό χρήστη, μέσω μιας σύγχρονης και ευέλικτης διαδικτυακής εφαρμογής. Προκειμένου να εκμεταλλευτούμε τις δυνατότητες που προσφέρουν τα πιο σύγχρονα εργαλεία ανάπτυξης διαδικτυακών εφαρμογών και μοντέρνα διαδικτυακά APIs, αποφασίσαμε να υλοποιήσουμε τη Διεπαφή Χρήστη(UI) του OntoMon με την τελευταία έκδοση της Angular. Όπως αναφέραμε προηγουμένως, επιδιώξαμε να αναπτύξουμε μια Διεπαφή Χρήστη η οποία να διαχειρίζεται αποδοτικά αφηρημένου περιεχομένου συστήματα και να μη βασίζει τη λειτουργία της σε εξειδικευμένες πληροφορίες γι'αυτά. Έτσι, προχωρήσαμε σε μια πλήρως δυναμική υλοποίηση, η οποία στηρίζεται στην οντολογική περιγραφή του βασικού στρώματος και απεικονίζει οπτικά τα αντικείμενά της στο `DOM` του περιηγητή(browser). Ασφαλώς, η Διεπαφή Χρήστη αυτή πρέπει να είναι απόλυτα συμβατή και να γνωρίζει τα APIs στα οποία βασίζονται οι επικοινωνίες εντός της πλατφόρμας μας.
Αναφορικά με την οπτικοποίηση των αντικειμένων, αυτή πραγματοποιείται αυτοματοποιημένα μέσω προγραμματιστικού ελέγχου επί των `SVG` αρχείων, τα οποία μετασχηματίζονται και, τελικά, οργανώνονται σε μια εμφωλευμένη δομή. Είναι αξιοσημείωτο, ότι σε frontend επίπεδο καμία απόφαση δε λαμβάνεται με βάση κάποιο συγκεκριμένο τύπο αντικειμένου, αφού κατά σύμβαση όλα τα αντικείμενα του OntoMon είναι αφηρημένα ως προς το περιεχόμενο και αντιμετωπίζονται με ενιαίο τρόπο. Απα-

ραίτητη προϋπόθεση για την απεικόνιση του συστήματος-στόχου είναι η ανάκτηση τόσο της Οντολογίας(*Ontology.json*), όσο και των *.svg* αρχείων από τον Εξυπηρετητή του OntoMon, μέσω `HTTP` αιτήσεων τύπου `GET`.

Εκτός από την απεικόνιση του συστήματος-στόχου στον περιηγητή ιστού, η Διεπαφή Χρήστη του OntoMon συγκεντρώνει και αναπαριστά με γραφικό τρόπο όλες τις μετρικές απόδοσης, τις ενημερώσεις κατάστασης λειτουργίας, τα μεταδεδομένα και τα αρχεία καταγραφής που προέρχονται από το μεσαίο στρώμα. Στην προτεινόμενη σχεδίαση, η Angular εφαρμογή εισάγει μια υπηρεσία που πραγματοποιεί διαρκή "σφυγμομέτρηση" του συστήματος-στόχου, αποστέλλοντας περιοδικά `HTTP GET` αιτήσεις στον OntoMon εξυπηρετητή. Στις αιτήσεις αυτές ο εξυπηρετητής της πλατφόρμας μας απαντά στέλνοντας τις πιο πρόσφατες ανανεώσεις κατάστασης και μετρικές απόδοσης των δομικών μονάδων υπό παρακολούθηση, πάντα σε μορφή `JSON`. Με τη σειρά της, η Διεπαφή Χρήστη διαβάζει τα *Update.json* αντικείμενα καθώς φτάνουν από τον Εξυπηρετητή και στη συνέχεια επιτελεί τις εξής λειτουργίες:

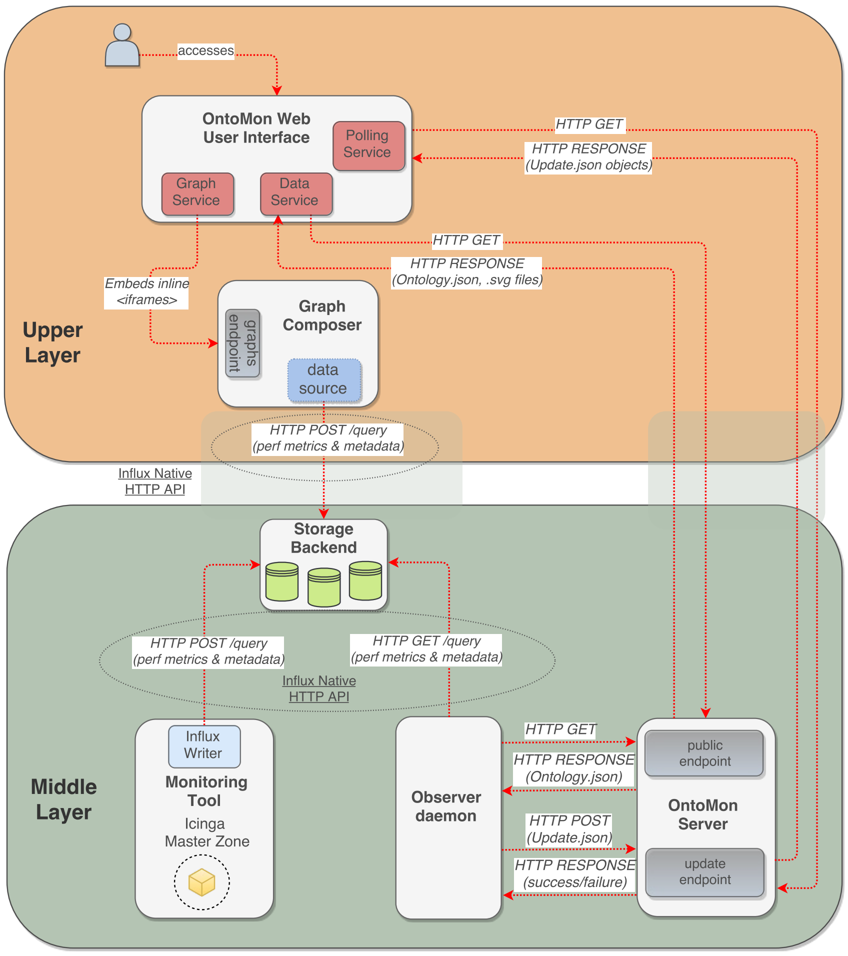
1. Παράγει ειδοποιήσεις σχετικά με τα συμβάντα που αφορούν μεμονωμένα δομικά μέρη του συστήματος-στόχου, προορισμένες για τον τελικό χρήστη. Η μορφή, το περιεχόμενο και η τελική εμφάνιση των ειδοποιήσεων αυτών εντός του UI του OntoMon καθορίζονται πλήρως από το περιεχόμενο συγκεκριμένων πεδίων του εκάστοτε *Update.json* αντικειμένου.

2. Παρουσιάζει συγκεντρωτικά τις τιμές των μετρικών απόδοσης κάθε δομικής μονάδας σε πραγματικό χρόνο, τόσο σε πίνακες, όσο και σε γραφικές παραστάσεις που ανανεώνονται αυτόματα καθώς καταφθάνουν νέα χρονοδεδομένα.

Μια σημαντική σχεδιαστική απόφαση που λάβαμε σχετικά με την αναπαράσταση της απόδοσης του συστήματος-στόχου σε γραφήματα, ήταν να απαλλάξουμε την Angular από την απαιτητική αυτή λειτουργία. Ως εκ τούτου, ενσωματώσαμε στην πλατφόρμα μας μια ξεχωριστή οντότητα η οποία αναλαμβάνει εξ'ολοκλήρου την κατασκευή και εξαγωγή διαδραστικών διαγραμμάτων απόδοσης με βάση τις μετρικές αποδόσεις του συστήματος-στόχου, και η οποία επικοινωνεί με τη Διεπαφή Χρήστη πάνω από το δίκτυο. Η απόφασή μας αυτή στοχεύει στο διαχωρισμό των ευθυνών εντός του υψηλού στρώματος, αλλά και στην αποτελεσματική κατανομή του συνολικού φόρτου εργασίας.

Συνθέτης Γραφημάτων Ένα από τα πλέον απαραίτητα χαρακτηριστικά μιας σύγχρονης Διεπαφής Χρήστη είναι η ικανότητα να συνοψίζουν και να παρουσιάζουν τη ζητούμενη πληροφορία με εύληπτο και ενστικτώδη τρόπο στον τελικό χρήστη, ώστε να καταστεί δυνατή η άμεση κατανόηση της τρέχουσας κατάστασης και η εξαγωγή σχετικών συμπερασμάτων. Θεωρούμε, λοιπόν, πως η ενσωμάτωση γραφημάτων για την οπτική αναπαράσταση των δεδομένων μιας εφαρμογής είναι βαρύνουσας σημασίας, καθώς απλοποιεί σημαντικά τον εντοπισμό προβλημάτων εντός του συστήματος-στόχου, ενώ διευκολύνει την ιστορική ανάλυση των μετρικών απόδοσης και την ανακάλυψη μοτίβων. Έτσι, διευκολύνεται η διαχείριση του συστήματος-στόχου και η απομονωμένη μελέτη της συμπεριφοράς συγκεκριμένων δομικών μονάδων στο χρόνο. Η τεχνολογία που επιλέξαμε για το Συνθέτη Γραφημάτων ήταν η Grafana, μια ανοικτού κώδικα πλατφόρμα οπτικοποίησης χρονοδεδομένων γραμμένη σε `Go` και `JavaScript`, η οποία υποστηρίζει πολυάριθμες λειτουργίες ανάλυσης, όπως μαθηματικές συναρτήσεις, φίλτρα, συνάθροιση κλπ. Η Grafana κατασκευάζει λεπτομερή και διαδραστικά γραφήματα, τα οποία σκιαγραφούν αποτελεσματικά τη συνολική απόδοση του συστήματος-στόχου. Ένα από τα σημεία-κλειδιά στο σχεδιασμό της Grafana, είναι η ανακατασκευή των γραφημάτων στην **πλευρά του πελάτη**, μειώνοντας τον υπολογιστικό φόρτο του εξυπηρετητή. Συγκεκριμένα, στην αρχική αίτηση του πελάτη ο εξυπηρετητής της Grafana απαντά με το βασικό `HTML` σκελετό του γραφήματος, ενώ το υπόλοιπο περιεχόμενο φορτώνεται δυναμικά χρησιμοποιώντας `JavaScript`. Τα γραφήματα αυτά βοηθούν τον τελικό χρήστη να αποκτήσει άμεσα μια ευρύτερη εικόνα του συστήματος-στόχου και να πραγματοποιήσει προβλέψεις για τη μελλοντική συμπεριφορά μεμονωμένων οντοτήτων. Ταυτόχρονα, η Grafana υποστηρίζει τον ορισμό χρηστών και ομάδων, καθώς και την ανάθεση ρόλων, ώστε η πρόσβαση στα dashboards που εξάγει να είναι πλήρως ελεγχόμενη.

Άλλο ένα πλεονέκτημα της Grafana είναι η συμβατότητα με πολυάριθμα backend περιβάλλοντα αποθήκευσης χρονοσειρών, τα οποία αποτελούν και την είσοδό του Συνθέτη Γραφημάτων. Στη σχεδίασή μας καθορίσαμε την Influx βάση χρονοδεδομένων ως την πηγή του Συνθέτη Γραφημάτων, ο εξυπηρετητής του οποίου ανακτά μέσω του `HTTP API` της Influx μετρικές και μεταδεδομένα από το σύστημα-στόχο σε τακτά χρονικά διαστήματα. Η δυνατότητα αυτή της Grafana επιτρέπει την απευθείας επικοινωνία μεταξύ του Συνθέτη Γραφημάτων και της Βάσης Χρονοσειρών, χωρίς να εμπλέκεται καθόλου στη διαδικασία αυτή η Διεπαφή Χρήστη. Έτσι, απαλλάσσουμε την πλατφόρμα μας από περιττές δωσοληψίες, μεταφορά δεδομένων μεγάλου όγκου και, κατά συνέπεια, υπερφόρτωση του δικτύου. Ακολούθως, καθώς ο δαίμονας της Grafana ανακτά χρονοδεδομένα από την Influx βάση, κατασκευάζει δυναμικά dashboards και panels για κάθε παρακολουθούμενη δονική μονάδα του συστήματος-στόχου(asset), τα οποία ανανεώνει αυτόματα καθώς καταφθάνουν νέες τιμές των μετρικών απόδοσης. Ταυτόχρονα, τα διαθέτει σε ένα ευρέως γνωστό διαδικτυακό άκρο για μελλοντική ενσωμάτωσή τους σε εφαρμογές. Πιο αναλυτικά, αντιστοιχίζει κάθε `panel` με ένα μοναδικό `URL`, το οποίο περιέχει μια παράμετρο που προσδιορίζει τη δομική μονάδα στην οποία αναφέρεται το εν λόγω γράφημα. Τελικά, η Angular εφαρμογή φορτώνει ως απομακρυσμένος πελάτης τα γραφήματα που αφορούν το σύστημα-στόχος κατά την ώρα της εκτέλεσης του OntoMon, ενσωματώνοντάς τα σε `HTML <iframes>`.



Σχήμα 3: *Επικοινωνία μεταξύ μεσαίου και υψηλού στρώματος*

3 Υλοποίηση

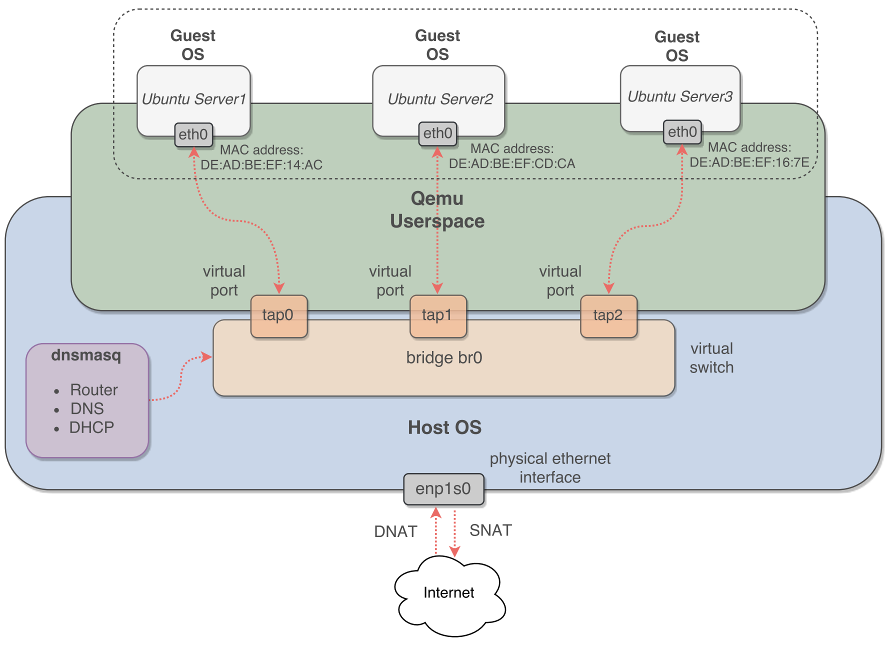
3.1 Πειραματική Διάταξη

Όπως ήδη αναφέραμε, το OntoMon ενσωματώνεται σε ετερογενή υπολογιστικά συστήματα τα οποία όμως σε κάποιο επίπεδο της οργάνωσής τους περιέχουν servers, οι οποίοι μπορούν αντιμετωπιστούν είτε ως οντότητε υλικού, είτε λογισμικού. Προκειμένου να πραγματοποιήσουμε τη μελέτη μας, λοιπόν, έπρεπε να εγκαταστήσουμε ένα πειραματικό περιβάλλον το οποίο να είναι συμβατό και με τους 2 αυτούς τύπους συστημάτων. Κατά συνέπεια, αποφασίσαμε να στήσουμε μια πλήρως ελεγχόμενη και λειτουργική συστοιχία εικονικών μηχανών, η οποία αντανακλά την εσωτερική δομή ενός υπολογιστικού συστήματος τόσο σε επίπεδο υλικού, όσο και λογισμικού ταυτόχρονα.Συγκεκριμένα, χρησιμοποιήσαμε το Qemu, μια ανοικτού κώδικα μηχανή εικονικοποίησης ώστε να ρυθμίσουμε από άκρη σε άκρη τη λειτουργία 3 υπολογιστικών κόμβων, διασυνδεδεμένων μεταξύ τους. Οι 3 αυτές εικονικές μηχανές φιλοξενούνται στο ίδιο φυσικό μηχάνημα(host). Για λόγους ομοιομορφίας εγκαταστήσαμε σε όλες τις εικονικές μηχανές και στο host μηχανήμα το ίδιο λειτουργικό σύστημα(Ubuntu 16.04.2), το οποίο υποστηρίζει λειτουργίες εικονικοποίησης.

Ρύθμιση Ιδιωτικού Δικτύου

Βασιζόμενοι στην υπόθεση ότι η συστοιχία εικονικών μηχανών μας προσομοιάζει σε οργάνωση και περιεχόμενο ένα πραγματικό υπολογιστικό περιβάλλον μέσα σε ένα Κέντρο Δεδομένων, έπρεπε να εξασφαλίσουμε τη δικτυακή επικοινωνία ανάμεσα στους κόμβους της. Έτσι, αποφασίσαμε να προχωρήσουμε στην υλοποίηση ενός προσαρμοσμένου στρώματος δικτύου μεταξύ των εικονικών μηχανών και του φυσικού μηχανήματος, καθορίζοντας μια τοπολογία με πρόσβαση στο διαδίκτυο πίσω από NAT. Κατά την υλοποίησή μας αυτή εκμεταλλευτήκαμε τις διεπαφές δικτύου που υποστηρίζουν το Qemu και το Linux. Αρχικά ορίσαμε μια διεπαφή γέφυρας(bridge) στο host μηχάνημα(br0) αναθέτοντάς της μια στατική IP διεύθυνση(10.0.0.254. Η εικονική αυτή διεπαφή επιπέδου λογισμικού προσομοιώνει τη λειτουργία ενός μεταγωγέα(switch), υλοποιείται μέσα στον πυρήνα του λειτουργικού συστήματος και αναλαμβάνει τη διαχείριση της δικτυακής κίνησης. Πάνω στη διεπαφή-γέφυρα του επιβλέποντος κόμβου(hypervisor) ρυθμίσαμε τη διεργασία dnsmasq, η οποία υλοποιεί υπηρεσίες δικτύου. Συγκεκριμένα κατευθύναμε τη δικτυακή κίνηση που προέρχεται από τις εικονικές μηχανές και αφορά τα DNS και DHCP πρωτόκολλα στη διεπαφή-γέφυρα με IP 10.0.0.254, η οποία αναθέτει στατικές IP διευθύνσεις εύρους 10.0.0.0/24 στις κάρτες δικτύου όλων των συνδεδεμένων κόμβων, αντιστοιχίζοντάς τις σε MAC διευθύνσεις. Ταυτόχρονα, απαντά σε DNS ερωτήματα και δρομολογεί όλα τα πακέτα που προορίζονται για το διαδίκτυο, λειτουργώντας ως πύλη(gateway) του ιδιωτικού δικτύου.

Για να πραγματοποιηθεί η σύνδεση μεταξύ της εικονικής κάρτας δικτύου και της διεπαφήςγέφυρας του host, κάθε Qemu διεργασία του userspace είναι υπεύθυνη να "σηκώσει" μια tap διεπαφή και να την συνάψει πάνω στη γέφυρα, ακριβώς όπως συνδέονται τα ethernet καλώδια σε ένα φυσικό μεταγωγέα. Για την αυτοματοποίηση της παραπάνω διαδικασίας προσαρμόσαμε κατάλληλα το qemu-if-up script, δημιουργώντας ένα ταίριασμα 3 tap διεπαφών στη διεπαφή-γέφυρα. Το τελευταίο βήμα για την επιτυχή λειτουργία του ιδιωτικού μας δικτύου αφορούσε τη ρύθμιση του host μηχανήματος. Πιο αναλυτικά, ενεργοποιήσαμε την προώθηση `IPv4` πακέτων εντός της διεργασίας διαχειριστή δικτύου, ενώ παράλληλα ορίσαμε έναν κανόνα για το τείχος προστασίας(firewall), ο οποίος επιτρέπει τη "μεταμφίεση" των πακέτων που καταφθάνουν στη διεπαφή-γέφυρα από το ιδιωτικό δίκτυο και προορίζονται για το διαδίκτυο.



Σχήμα 4: *Τοπολογία Συστοιχίας Εικονικών Μηχανών*

3.2 Προσδιορισμός Σχήματος Οντολογίας

Σε αυτήν την ενότητα θα παρουσιάσουμε τον ακριβή τρόπο φορμαλισμού της Οντολογίας, στοχεύοντας στην αναπαράστασή της με ένα σχήμα που οργανώνεται με το ανοικτό πρότυπο `JSON`. Κάθε αντικείμενο που ανήκει στην Οντολογία ακολουθεί την προτεινόμενη δομή, προκειμένου να διευκολυνθούν οι λειτουργίες διαβάσματος και ανάλυσης από την πλατφόρμα μας. Σύμφωνα με τις σχεδιαστικές μας αρχές, χρειαζόμαστε ένα αφηρημένο ως προς το περιεχόμενο, αλλά αυστηρό και καταληπτό μοντέλο δεδομένων, το οποίο δε θα περιέχει περιττή πληροφορία και ταυτόχρονα θα περιγράφει επαρκώς την οργάνωση και τις έννοιες του εκάστοτε συστήματος-στόχου. Το προτεινόμενο οντολογικό σχήμα φαίνεται παρακάτω:

```
{
```

```json
    "uuid": "c8bd7185-6349-4df5-8628-f87115222987",
    "name": "Ubuntu-Server-1",
    "label": "Server",
    "file": "Server.svg",
    "parent": "25e2ce69-2445-4b28-9a71-e7ca01bc57ea",
    "info": {
       "description": "Server asset"
     }
}
```

Listing 1: *Example of JSON Ontology object*

- **`uuid:`** ένα 128-bit αναγνωριστικό για το μοναδικό προσδιορισμό κάθε αντικειμένου, με στόχο την αποφυγή συγκρούσεων σε μεγάλη κλίμακα.
- **`name:`** πεδίο τύπου `string` που καθορίζεται από το χρήστη και περιέχει το όνομα κάθε αντικειμένου. Παρόλο που το OntoMon δεν πραγματοποιεί σημασιολογικό έλεγχο, η τιμή του πεδίου αυτού θεωρητικά αντιστοιχεί σε μια δομική μονάδα του συστήματος-στόχου.
- **`label:`** προσφέρει ομαδοποίηση παρόμοιων αντικειμένων εντός της Διεπαφής Χρήστη, χωρίς όμως να προσαρμόζεται οποιοσδήποτε άλλος χειρισμός τους ανάλογα με την τιμή του. Έτσι, σε καμία περίπτωση δε θα πρέπει να συγχέεται με την εισαγωγή τύπων στα αντικείμενα.
- **`parent:`** εκφράζει τις εξαρτήσεις ανάμεσα στα αντικειμένα της Οντολογίας, ορίζοντας σχέσεις "πατέρα-παιδιού". Η τιμή του πεδίου αυτού είναι είτε `null` για το αντικείμενο-ρίζα, είτε το `uuid` ενός υπάρχοντος αντικειμένου. Ακολουθώντας τις σχέσεις που καταδεικνύει η ιδιότητα αυτή κατασκευάζουμε το ιεραρχικό δέντρο της Οντολογίας του συστήματος-στόχου.
- **`file:`** ορίζει το αρχείο κλιμακούμενων διανυσματικών γραφικών (`SVG`) το οποίο θα χρησιμοποιηθεί από τη Διεπαφή Χρήστη για την προσαρμοσμένη οπτικοποίηση του συγκεκριμένου αντικειμένου της Οντολογίας. Το αρχείο αυτό πρέπει να είναι στη διάθεση του Εξυπηρετητή του OntoMon κατά την εκτέλεση, ενώ απαιτείται να είναι τύπου `.svg`, ώστε να είναι εφικτός ο προγραμματιστικός έλεγχός του. Επίσης, απαιτούμε ο `XML` κώδικας που υλοποιεί το αρχείο `SVG` να έχει ορισμένες ιδιότητες τις οποίες χρειαζόμαστε για τη χαμηλού επιπέδου επεξεργασία του. Στόχος μας είναι

να απεικονίσουμε τα SVG αρχεία με εμφωλευμένο τρόπο μέσα στο DOM του περιηγητή ιστού, ώστε να σκιαγραφήσουμε την οργάνωση του συστήματος-στόχου και να φανούν οι σχέσεις εξάρτησεις μεταξύ των δομικών του μερών. Επομένως, χρειαζόμαστε ένα μηχανισμό που να εκφράζει την ικανότητα των αντικειμένων να **περιέχουν** άλλα. Αποφασίσαμε να μοντελοποιήσουμε το μηχανισμό αυτό ως μια inline ιδιότητα των `.svg` αρχείων, εισάγοντας την έννοια των **`slots`**. Κάθε `slot` ορίζεται από ένα σημείο αναφοράς(πάνω αριστερά) και διαστάσεις μήκους και πλάτους:

<upper-left-point1-x>, <upper-left-point1-y>, <width>, <height>

Κάθε SVG με n `slots` μπορεί να περιέχει έως και n άλλα SVGs, ενώ η συσχέτιση των `slots` με την ιδιότητα `parent` μας δίνει μια αντιπροσωπευτική αναπαράσταση της εσωτερικής οργάνωσης της παρακολουθούμενης υποδομής. Επιπρόσθετα, προκειμένου να καταστεί εφικτή η προσαρμογή των οπτικών ειδοποιήσεων που παραδίδει η Διεπαφή Χρήστη απαιτούμε τον ορισμό ενός στοιχείου(element) κλάσης **`indicator`** στο σώμα του SVG αρχείου από τον προγραμματιστή ενσωμάτωσης. Η απόφαση μας αυτή αφορά τον εύκολο εντοπισμό του στοιχείου `indicator` κατά τη λειτουργία της οπτικοποίησης. Πιο αναλυτικά, σε αυτό το HTML στοιχείο θα πραγματοποιηθεί κάποια μεταβολή στην τιμή κάποιας από τις inline ιδιότητές του, αντιπροσωπεύοντας την ανανέωση της τρέχουσας κατάστασης λειτουργίας του αντίστοιχου δομικού μέρους του συστήματος-στόχου.

```
<svg xmlns="http://www.w3.org/2000/svg" width="100%" height="100%"
    slots="15,25,200,50,45,75,240,60">
    <g>
            <circle>
                <animate class="indicator" attributeName="fill"
                values="#62c36e;#36a242;#62c36e"/>
            </circle>
    </g>
</svg>
```

Listing 2: *SVG αρχείο με 2 slots και 1 indicator στοιχείο*

- **`info`:** περιέχει οποιεσδήποτε πληροφορίες θέλει να συμπεριλάβει ο χρήστης σχετικά με κάθε αντικείμενο της Οντολογίας, για την πιο πλήρη περιγραφή των οντοτήτων του συστήματος-στόχου. Συνήθως πρόκειται για ένα JSON αντικείμενο, με επιμέρους πεδία και τιμές, τα οποία παρουσιάζονται εντός της Διεπαφής Χρήστη για λόγους πληρότητας.

3.3 Υλοποίηση του OntoMon Εξυπηρετητή

Η υλοποίηση ενός εξυπηρετητή διαδικτύου(web server) είναι μια, γενικά, απαιτητική διαδικασία. Προκειμένου να επιταχύνουμε την ανάπτυξη του OntoMon και να μην προχωρήσουμε σε δικτυακές ρυθμίσεις χαμηλού επιπέδου, αποφασίσαμε να χτίσουμε τον Εξυπηρετητή του OntoMon πάνω στη μηχανή εκτέλεσης JavaScript Node.js, και συγκεκριμένα στο πλαίσιο λογισμικού Express.js. Το πλαίσιο αυτό βασίζεται σε ασύγχρονα συμβάντα τα οποία διαχειρίζεται ένα μοναδικό νήμα εκτέλεσης, σε αντίθεση με το παραδοσιακό Request/Response πολυνηματικό μοντέλο. Το Express.js μας επέτρεψε να υλοποιήσουμε με απλό τρόπο ένα κλιμακώσιμο Εξυπηρετητή, γραμμένο εξ'ολοκλήρου σε JavaScript, επωφελούμενοι τόσο από το μεγάλο εύρος υποστηριζόμενων βιβλιοθηκών υψηλού επιπέδου, όσο και από τα μοντέρνα διαδικτυακά APIs. Ο Εξυπηρετητής του OntoMon ενορχηστρώνει και εξυπηρετεί όλα τα αιτήματα σχετικά τα δεδομένα της πλατφόρμας μας, προσφέροντας δομημένα APIs επικονωνίας και διαδικτυακά άκρα για τις συνδέσεις των πελατών. Οι βασικές λειτουργίες που υλοποιήσαμε φαίνονται στον ακόλουθο πίνακα:

Διαδικτυακή Υπηρεσία	**Άκρο**	**Πελάτης**
Ανέβασμα .json, .svg αρχείων	`/upload`	Τελικός-Χρήστης
Διάθεση στατικών αρχείων	`/resources`	Διεπαφή Χρήστη, Παρακολουθητής
Χειρισμός ενημερώσεων	`/updates`	Διεπαφή Χρήστη, Παρακολουθητής
Διάθεση αρχείου καταγραφής	`/history`	Διεπαφή Χρήστη

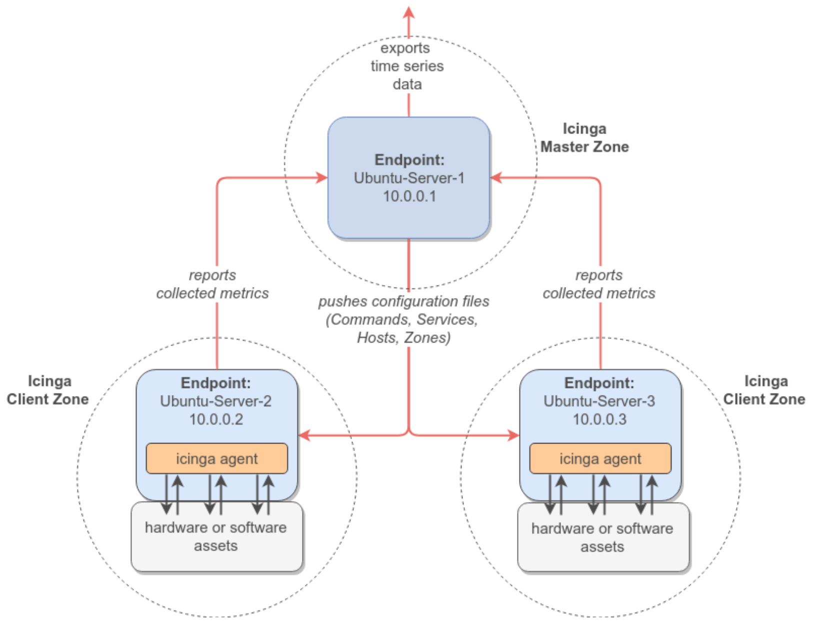
Πίνακας 1: *Υπηρεσίες του OntoMon Εξυπηρετητή*

Στο δικτυακό άκρο ontomonHost:8080/upload ο εξυπηρετητής μας παραλαμβάνει αρχεία μέσω απλών HTTP POST αιτημάτων, τα οποία στέλνει ο προγραμματιστής ενσωμάτωσης συστήματος. Τα αρχεία αυτά(*Ontology.json, .svg*) διατείθενται προς ανάκτηση στο δικτυακό άκρο ontomonHost:8080/resources. Ταυτόχρονα, προχωρήσαμε στην υλοποίηση μιας ειδικής συνάρτησης, η οποία διαχειρίζεται τα HTTP POST αιτήματα που αφορούν τις ενημερώσεις κατάστασης της παρακολουθούμενης υποδομής(*Update.json*) στο δικτυακό άκρο `ontomonHost:8080/updates`. Τέλος, προκειμένου να διατηρήσει το ιστορικό των ενημερώσεων κατάστασης λειτουργίας του συστήματος-στόχου, ο Εξυπηρετητής μας παράγει ένα αρχείο καταγραφής το οποίο διαθέτει στο άκρο `onotmonHost:8080/history`. Το αρχείο αυτό ανακτάται μετέπειτα από τη Διεπαφή Χρήστη, ώστε να ενημερώσει τον τελικό χρήστη σχετικά με την ιστορική εξέλιξη της συμπεριφοράς της παρακολουθούμενης υποδομής.

3.4 Συλλογή Μετρικών Απόδοσης με το Icinga

Προκειμένου να δειγματοληπτήσουμε σωστά το σύστημα-στόχο, προχωρήσαμε σε προσεκτική ρύθμιση της συστοιχίας του Icinga θέτοντας ως προτεραιότητες την υψηλή απόδοση και τη εύκολη συντήρηση και επέκταση σε μεγάλη κλίμακα. Παρόλο που το δοκιμαστικό περιβάλλον μας αποτελείται από 3 κόμβους, φροντίσαμε η υλοποίηση των υπηρεσιών συλλογής μετρικών απόδοσης να είναι κλιμακώσιμη σε πιο σύνθετες τοπολογίες και να αντανακλά πλήρως τις ανάγκες των διαχειριστών.
Αρχικά εγκαταστήσαμε σε όλους τους κόμβους του συστήματος-στόχου τα πακέτα λογισμικού του Icinga μέσω απομακρυσμένης `ssh` σύνδεσης και στη συνέχεια χρησιμοποιώντας το `Icinga node wizard` εγκαταστήσαμε τον αντίστοιχο Icinga agent σε κάθε κόμβο. Στον κόμβο που φιλοξενεί τον Icinga master agent ορίσαμε την επιθυμητή ιεραρχία από `Zones` και `Endpoints`: 1 master `Zone` με `Endpoint` τον κόμβο με `IP 10.0.0.1` και 2 ισότιμες client `Zones` με `Endpoints 10.0.0.2` και `10.0.0.3`, αντίστοιχα. Έτσι, καθένας από τους client agents γνωρίζει τον υπεύθυνο κόμβο στον οποίο θα καταθέτει τις μετρικές απόδοσης που συλλέγει, μέσω του Icinga API. Η εγγραφή των client agents στον master agent πραγματοποιήθηκε μετά από ταυτοποίηση CA πιστοποιητικών και CSR υπογραφών για λόγους ασφάλειας. Έτσι, πετύχαμε μια κατανεμημένη οργάνωση του συστήματος παρακολούθησης, στην οποία οι έλεγχοι απόδοσης των εικονικών server λαμβάνουν χώρα τοπικά, κατανέμοντας ομοιόμορφα το φόρτο εργασίας.
Στην ίδια κατεύθυνση, ο ορισμός των αντικειμένων του Icinga που προσδιορίζουν το περιεχόμενο και την πολιτική της παρακολούθησης του συστήματος-στόχου(πχ `Hosts, Services, CheckCommands`) πραγματοποιήθηκε στον Icinga master κόμβο, ώστε να θεμελιώσουμε τον από τα πάνω προς τα κάτω συγχρονισμό των ρυθμίσεων εντός της συστοιχίας του Icinga. Η δομή αυτή είναι εύληπτη και ταυτόχρονα αποδο-

τική, ενώ μπορεί να επεκταθεί αρκετά εύκολα προκειμένου να υποστηρίξει μεγαλύτερο αριθμό client κόμβων. Οι client agents παραλαμβάνουν αυτόματα πάνω από το δίκτυο τα αντικείμενα παρακολούθησης του Icinga, τα αντιγράφουν τοπικά και προχωρούν στην εκτέλεση των ζητούμενων ελέγχων απόδοσης των δομικών μερών του συστήματος-στόχου. Η συστοιχία του Icinga φαίνεται στο παρακάτω σχήμα:


exports time series data
Icinga Master Zone
Endpoint: Ubuntu-Server-1 10.0.0.1
reports collected metrics
pushes configuration files (Commands, Services, Hosts, Zones)
reports collected metrics
Icinga Client Zone
Endpoint: Ubuntu-Server-2 10.0.0.2
icinga agent
hardware or software assets
Endpoint: Ubuntu-Server-3 10.0.0.3
icinga agent
hardware or software assets
Icinga Client Zone

Σχήμα 5: *Icinga Monitoring Cluster*

Όλα τα αρχεία ρυθμίσεων του συστήματος παρακολούθησης είναι τοποθετημένα κάτω από τον κατάλογο `/etc/icinga2/zones.d` του Icinga master κόμβου, όπου έχουμε δημιουργήσει αντίστοιχους φακέλους για κάθε `Zone`. Όλοι οι κόμβοι εντός μιας ζώνης του Icinga λαμβάνουν ακριβώς τις ίδιες ρυθμίσεις και οδηγίες, οι οποίες περιέχονται στα σχετικά αρχεία `hosts.conf, services.conf` και `templates.conf`. Τα `CheckCommand` αντικείμενα προσδιορίζουν εντολές εντός του Icinga για την εκτέλεση κατάλληλων scripts που δειγματοληπτούν την απόδοση του εκάστοτε κόμβου και καταγράφουν μετρικές απόδοσης. Τα scripts αυτά έχουν τη μορφή plugins και συνήθως είναι γραμμένα σε `bash`, `Python` ή `Perl`. Για την παρακολούθηση των συστημάτων-στόχων που μελετήσαμε, χρειάστηκε να γράψουμε τα δικά μας plugins ή να προσαρμόσουμε ορισμένα ήδη υπάρχοντα στις ανάγκες μας. Από την άλλη, τα `Services` αντικείμενα καθορίζουν ποιό `CheckCommand` θα εκτελεστεί, με ποιά ορίσματα, κάθε πότε κλπ.

Αναφορικά με την εξαγωγή των μετρικών απόδοσης που συλλέχθηκαν σε πραγματικό χρόνο, το Icinga είναι απόλυτα συμβατό με την InfluxDB την οποία επιλέξαμε για την αποθήκευση των χρονοδεδομένων που διαχειρίζεται το OntoMon. Αρχικά ενεργοποιήσαμε το ενσωματωμένο `influxdb feature` του Icinga, το οποίο αρχικοποιεί τον `Influx Writer`, ένα αντικείμενο του Icinga σχεδιασμένο να επικοινωνεί με το backend περιβάλλον αποθήκευσης και να αποστέλλει περιοδικά τα αποτελέσματα των ελέγχων απόδοσης της παρακολουθούμενης υποδομής. Ακόλουθα, χρειάστηκε να προχωρήσουμε στην εξειδικευμένη ρύθμισή του, παρέχοντας ακριβείς πληροφορίες σχετικά με το δαίμονα της βάσης χρονοδεδομένων που εξυπηρετεί τις αιτήσεις αποθήκευσης. Συγκεκριμένα καθορίσαμε το όνομα του κόμβου, το δικτυακό άκρο, το όνομα της βάσης, το όνομα χρήστη και τον κωδικό πρόσβασης.

3.5 Αποθήκευση Χρονοσειρών με την InfluxDB

Σε αυτό το σημείο έχουμε εγκαταστήσει επιτυχώς το σύστημα δειγματοληψίας και συλλογής μετρικών απόδοσης, επομένως σειρά έχει η ενσωμάτωση της βάσης χρονοσειρών για τη μακροχρόνια διατήρηση των τιμών αυτών. Όπως και προηγούμενα, εγκαταστήσαμε τα πακέτα της πιο πρόσφατης έκδοσης της InfluxDB στην πειραματική μας διάταξη, αυτή τη φορά όμως στον επιβλέποντα κόμβο(hypervisor), καθώς εκεί εκτελείται ο Παρακολουθητής του OntoMon, ο οποίος χρειάζεται τις πιο πρόσφατες μετρήσεις επίδοσης του συστήματος-στόχου. Ως εκ τούτου, η ανάκτηση των μετρικών απόδοσης γίνεται άμεσα και τοπικά, χωρίς επιπλέον επιβάρυνση του δικτύου. Πέρα από αυτό, ρυθμίσαμε έναν NTP εξυπηρετητή για τον αποτελεσματικό συγχρονισμό των μετρήσεων και τη θέσπισης μιας χρονικής σύμβασης.
Στη συνέχεια καθορίσαμε την οντότητα `database` της Influx, προσδιορίζοντας το όνομά της αλλά και την την πολιτική κράτησης των χρονοδεδομένων σε αυτήν, δηλαδή το χρονικό διάστημα για το οποίο οι μετρικές απόδοσης θεωρούνται έγκυρες και παραμένουν διαθέσιμες προς ανάκτηση. Κατά τη λειτουργία της, η Influx εισάγει εσωτερικά τις έννοιες των `shards, groups` για την καλύτερη οργάνωσή της. Κάθε Influx `database` αποτελείται από `data points`, τα οποία είναι αντικείμενα με ιδιότητες `name, tags, timestamp` και `fields`, ενώ συνήθως είναι συσχετισμένα με κάποια χρονοσειρά μετρήσεων(`series` ή `measurement`).
Αποσκοπώντας στον πλήρη έλεγχο της δομής των μετρικών που συλλέγονται στο σύστημα-στόχο, έπρεπε να προσαρμόσουμε τις ρυθμίσεις του `InfluxDBWriter` αντικειμένου του Icinga στις ανάγκες μας. Έτσι, σκιαγραφήσαμε το πρότυπο των εγγρα-

φών που αποστέλλει ο συλλέκτης μετρικών απόδοσης στο backend περιβάλλον αποθήκευσης, καθορίζοντας επακριβώς κάθε πεδίο τους, συμπεριλαμβάνοντας μεταδεδομένα και τιμές-κατώφλια. Συγκεκριμένα θέσαμε ως όνομα του πεδίου `measurement` κάθε εγγραφής το όνομα του αντίστοιχου `CheckCommand` αντικειμένου που πραγματοποιεί την καταγραφή του. Παράλληλα, προσθέσαμε ένα μικρό σετ από `tags` τα οποία διευκολύνουν τα InfluxQL ερωτήματα ως προς τη συνάθροιση και το φιλτράρισμα. Τέλος, συμπεριλάβαμε το πεδίο `timestamp` το οποίο υπήρχε εξ' ορισμού, προκειμένου να γνωρίζουμε τους χρόνους δειγματοληψίας.

3.6 Παρακολουθητής του OntoMon

Από τη στιγμή που οι agents του Icinga είναι επιφορτισμένοι αποκλειστικά με την εκτέλεση περιοδικών ελέγχων και τη συλλογή των αντίστοιχων μετρικών απόδοσης από τις δομικές μονάδες του συστήματος-στόχου, έπρεπε να εισάγουμε μια εξειδικευμένη οντότητα για την υλοποίηση της εκάστοτε πολιτικής παρακολούθησης του συστήματος-στόχου, όπως αυτή επιβάλλεται από τον τελικό χρήστη και των προγραμματιστή ενσωμάτωσης. Δηλαδή χρειαζόμασταν μια διαρκώς διαθέσιμη διεργασία η οποία θα τρέχει στο παρασκήνιο, θα εφαρμόζει την εκάστοτε συλλογιστική επί των χρονοδεδομένων και θα ενημερώνει τη Διεπαφή Χρήστη για την τρέχουσα κατάσταση του συστήματος-στόχου. Ως εκ τούτου, αναπτύξαμε τον Παρακολουθητή του OntoMon ως ένα δαίμονα γραμμένο εξ' ολοκλήρου σε `Python`, με απώτερο στόχο να εξασφαλίζει 2 βασικά χαρακτηριστικά: **προσαρμοστικότητα** και **ευελιξία**. Εν προκειμένω, ο Παρακολουθητής της πλατφόρμας μας πρέπει να υποστηρίζει εξατομικευμένες υπηρεσίες παρακολούθησης για κάθε σύστημα-στόχο και ταυτόχρονα να καθορίζει την οπτική αναπαράσταση των αντίστοιχων ειδοποιήσεων που προορίζονται για τον τελικό χρήστη, χωρίς να εξαρτάται από τη σημασιολογία των δομικών μερών της υποδομής.

Ακόλουθα, η προτεινόμενη υλοποίηση βασίστηκε στην ανάπτυξη μηχανισμών και υπηρεσιών γενικού σκοπού που είναι σχεδιασμένες να προσαρμόζονται από τον προγραμματιστή ενσωμάτωσης στις ανάγκες ετερογενών συστημάτων-στόχων. Η προσέγγισή μας προσδιορίζεται από διαλειτουργικότητα των επιμέρους μεθόδων χειρισμού των χρονοδεδομένων, ενώ ενθαρρύνεται η ενεργή συνεισφορά των προγραμματιστών ενσωμάτωσης για την πλήρη προσαρμογή του πυρήνα εξαγωγής συμπερασμάτων του Παρακολουθητή και τον καθορισμό της “νοημοσύνης” του. Τα 3 πιο θεμελιώδη modules που οικοδομούν τον Παρακολουθητή της πλατφόρμας μας παρέχοντας τη ζητούμενη λειτουργικότητα είναι τα εξής:

- **`Observer.py`:** πρόκειται για τον κεντρικό σκελετό του Παρακολουθητή, καθώς εδώ βρίσκεται ο κώδικας-οδηγός ο οποίος συντονίζει τις διάφορες υπηρεσίες ελέγχου. Προσαρμόζοντας το συγκεκριμένο κώδικα, ο προγραμματιστής ενσωμάτωσης μπορεί να ορίσει ως παραμέτρους γραμμής εντολών(CLI) τα αναγνωριστικά των υπό παρακολούθηση servers του συστήματος-στόχου, προκειμένου να χρησιμοποιηθούν ως κλειδιά για την ανάκτηση των σχετικών μετρικών απόδοσης από τη βάση χρονοδεδομένων. Επιπρόσθετα, εντός του `Observer.py` καλούνται όλες οι συναρτήσεις ελέγχου επί των δομικών μονάδων του συστήματος-στόχου.

- **`Services.py`:** σε αυτό το module περιέχονται οι ορισμοί των εκάστοτε συναρτήσεων χειρισμού του Παρακολουθητή, στο σώμα των οποίων υλοποιούνται οι υπηρεσίες παρακολούθησης, ελέγχου και συμπερασμού τρέχουσας κατάστασης. Κάθε υπηρεσία ελέγχου αντιστοιχεί σε μια συνάρτηση-χειριστή. Για να διεκολύνουμε την προσαρμογή, παρέχουμε την αφηρημένη ως προς το περιεχόμενο συνάρτηση `get_measurement()`, η οποία χτίζει δυναμικά InfluxQL ερωτήματα τα οποία αποστέλλει μέσω `HTTP GET` αιτημάτων στον εξυπηρετητή της Influx βάσης χρονοδεδομένων και συγκεκριμένα στο δικτυακό άκρο `ontomonHost:8080/query`. Στη συνέχεια, διαβάζει την απάντηση του εξυπηρετητή, μοντελοποιεί τις τιμές των μετρικών απόδοσης που ανέκτησε σε μορφή λίστας και τελικά την προωθεί στην εκάστοτε συνάρτηση-χειριστή για την πραγματοποίηση συγκρίσεων με τιμές-κατώφλια τις οποίες έχει καθορίσει ο προγραμματιστής ενσωμάτωσης. Κάθε συνάρτηση-χειριστής απομονώνει ή συνδυάζει χρονοδεδομένα και εκτιμά την τρέχουσα κατάσταση του εκάστοτε δομικού μέρους του συστήματος-στόχου. Οι δυνατές τιμές της κατάστασης(`state`) του κάθε δομικού μέρους είναι `OK, WARNING` και `CRITICAL`, ενώ η πληροφορία αυτή συμπεριλαμβάνεται στη δομή ενημέρωσης που προορίζεται για το υψηλό στρώμα του OntoMon.

- **`Dispatcher.py`:** αυτό το κομμάτι κώδικα αναλαμβάνει την κατασκευή του *Update.json* αντικειμένου για την αναπαράσταση της τρέχουσας κατάστασης της παρακολουθούμενης υποδομής, όπως αυτή προέκυψε από την αντίστοιχη συνάρτηση-χειριστή. Ουσιαστικά μέσα στη συνάρτηση `dispatcher()` υλοποιείται το εξατομικευμένο

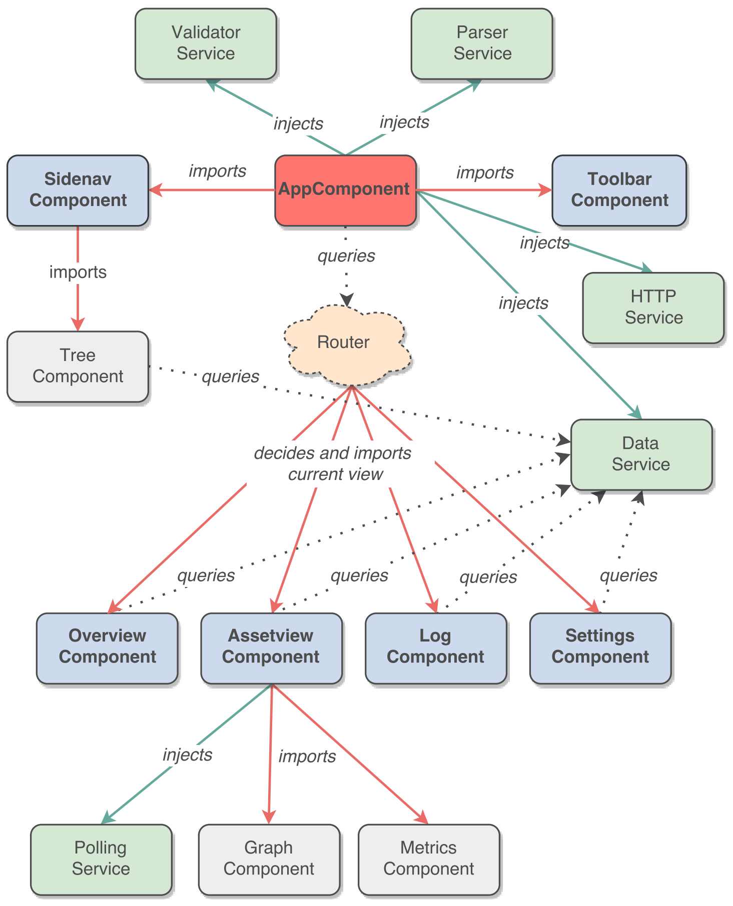
API ειδοποιήσεων του OntoMon, επομένως στο σημείο αυτό ο προγραμματιστής ενσωμάτωσης καθορίζει τον τύπο της οπτικοποίησης των ειδοποιήσεων της Διεπαφής Χρήστη που επιθυμεί ο τελικός χρήστης. Μόλις ολοκληρωθεί, λοιπόν, η κατασκευή του *Update.json* αντικειμένου, ο Παρακολουθητής στέλνει ένα `HTTP POST` αίτημα στον Εξυπηρετητή του OntoMon στο δικτυακό άκρο `onotmonHost:8080/updates`, ώστε να δημοσιεύσει την πιο πρόσφατη εικόνα της παρακολουθούμενης υποδομής και αυτή εν συνεχεία να αντικατοπτριστεί στην Angular εφαρμογή του υψηλού στρώματος της πλατφόρμας μας. Το πρότυπο του μηχανισμού ειδοποιήσεων που υλοποιήσαμε παράγει μεμονωμένες ενημερώσεις κατάστασης λειτουργίας που αναφέρονται σε συγκεκριμένα αντικείμενα της Οντολογίας και περιέχουν την τρέχουσα κατάστασή τους, τις πιο πρόσφατες τιμές των μετρικών απόδοσής τους, καθώς και λεπτομερείς οδηγίες σχετικά με την οπτικοποίηση της ενημέρωσης στο UI:

```
{
   "uuid": "ff2802db-15d1-42a6-bcfe-0dd3af12c9c7",
   "name": "icinga2-client1",
   "timestamp": "2017-05-15 00:56:12",
   "state": "OK",
   "metrics": {
      "cpu_load1": 0.5,
      "cpu_load5": 0.7,
      "cpu_load15": 0.8
   },
   "type": {
         "element_class": "indicator",
         "attribute_name": "values",
         "attribute_value": color
      },
   "description": "No issues were detected."
}
```

Listing 3: *Update.json Αντικείμενο Ενημέρωσης Κατάστασης*

3.7 Ανάπτυξη της Διεπαφής Χρήστη

Δεδομένου ότι η Διεπαφή Χρήστη της πλατφόρμας μας είναι η μοναδική οντότητα της πλατφόρμας μας με την οποία αλληλεπιδρά ο τελικός χρήστης, αφιερώσαμε αρκετό χρόνο και προσπάθεια κατά την ανάπτυξή της προκειμένου να προσφέρει μια συνοπτική και ταυτόχρονα ολοκληρωμένη εικόνα του παρακολουθούμενου συστήματος-στόχου, αλλά και να είναι εναρμονισμένη με τις αρχές σχεδίασης των κατώτερων στρωμάτων. Για την κατασκευή της επιλέξαμε την τελευταία έκδοση του πλαισίου διαδικτυακού λογισμικού Angular, υλοποιώντας την πλήρως σε `Typescript`, ένα υπερσύνολο της `JavaScript` που ενσωματώνει επιπλέον λειτουργίες και έννοιες όπως κλάσεις και τύπους μεταβλητών. Αναφορικά με τη σχεδίαση της εμφάνισης και του βασικού σκελετού ακολουθήσαμε τις βέλτιστες πρακτικές και συστάσεις των δημιουργών της Angular, ενώ αναφορικά με το περιεχόμενο επιλέξαμε να μην υπερφορτώσουμε την εφαρμογή μας με πληροφορία, αλλά να εστιάσουμε στην ουσιώδη αναπαράσταση της παρακολουθούμενης υποδομής με κατανοητό τρόπο.
Αρχικά, εφόσον οι Angular εφαρμογές βασίζονται κατά κόρον σε `Components`, αναπτύξαμε 4 βασικές όψεις για τη Διεπαφή Χρήστη σε 4 διακριτά `Components`: *overview, assetview, log, settings*. Η overview όψη παρουσιάζει ολόκληρο το σύστημα-στόχο, ενώ η όψη assetview επικεντρώνεται σε κάποιο συγκεκριμένο δομικό μέρος του. Ταυτόχρονα, υλοποιήσαμε και ορισμένα βοηθητικά `Components` τα οποία ενσωματώνονται στα βασικά και προσθέτουν επιπλέον λειτουργικότητα, όπως η απεικόνιση των γραφημάτων και του ιεραρχικού δέντρου της Οντολογίας.
Εκτός αυτών, προχωρήσαμε στον ορισμό `Services` εντός της Διεπαφής Χρήστη, δηλαδή εκειδικευμένων κλάσεων-υπηρεσιών οι οποίες ενσωματώνονται στα `Components` και είναι επιφορτισμένες με την περάτωση συγκεκριμένων λειτουργιών. Τα πιο σημαντικά `Services` της Διεπαφής Χρήστη του OntoMon αφορούν τη διαχείριση αιτήσεων `HTTP` για την ανάκτηση των *Ontology.json* και *.svg* αρχείων, την επαλήθευση της εγκυρότητας των απαιτούμενων δομών, καθώς και την αρχικοποίηση των εσωτερικών της μεταβλητών.
Για τη μείωση της πολυπλοκότητας και τον καλύτερο διαχωρισμό των αρμοδιοτήτων χωρίσαμε τις λειτουργίες της Διεπαφής Χρήστη του OntoMon σε 3 κύκλους:



Σχήμα 6: *Διεπαφή Χρήστη OntoMon: Αρχιτεκτονική*

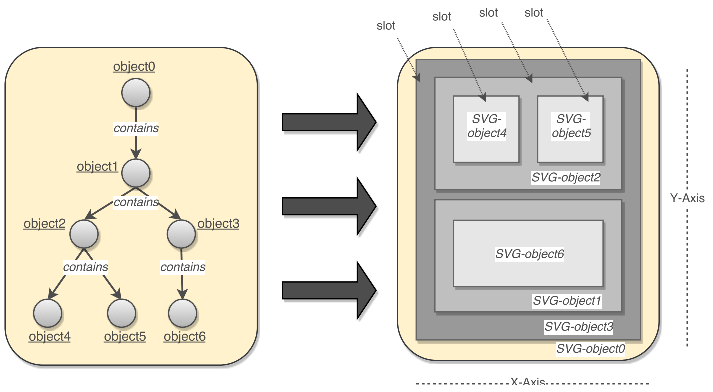
Κύκλος1: Διάβασμα και Έλεγχος Οντολογίας Σε πρώτη φάση τα `Components` και τα `Services` της εφαρμογής υλοποιούν τη θεμελιώδη οντολογική βάση του OntoMon. Αρχικά ανακτάται το αρχείο της Οντολογίας,*Ontology.json*, από το δικτυακό άκρο `ontomonHost:8080/resources` του Εξυπηρετητή του OntoMon μέσω της `HttpService` και στη συνέχεια προωθείται στην `ValidatorService` για έλεγχο. Μόλις ολοκληρωθεί ο έλεγχος εγκυρότητας της Οντολογίας ως προς το σχήμα και την εσωτερική της οργάνωση σε τοπολογία δέντρου, η υπηρεσία `ParserService` τη διαβάζει διεξοδικά και αρχικοποιεί το λεξικό `Controller`, μια εσωτερική δομή τύπου `Object` για την αναπαράσταση των πληροφοριών και των δεδομένων που αφορούν κάθε αντικείμενο της Οντολογίας. Ο `Controller` είναι η πιο σημαντική δομή της εφαρμογής μας, η οποία δεικτοδοτεί τα πεδία της με βάση το `uuid` κάθε αντικειμένου, ενώ στην ουσία επεκτείνει δυναμικά τα πεδία των `JSON` αντικειμένων της Οντολογίας κατά την επεξεργασία τους. Σε αυτό το σημείο πρέπει να επισημάνουμε πως απαιτείται ο **ασύγχρονος** χειρισμός των παραπάνω λειτουργιών ώστε τα `Components` που εξαρτώνται από κάποια ασύγχρονη λειτουργία να “ξυπνήσουν” όταν τα δεδομένα που χρειάζονται είναι έτοιμα. Για το λόγο αυτό χρησιμοποιήσαμε τόσο τα ενσωματωμένα `Events` της Angular, όσο και τη βιβλιοθήκη `rxjs` της `JavaScript`. Για παράδειγμα, το `TreeComponent` το οποίο οπτικοποιεί το δέντρο της Οντολογίας κάνοντας χρήση της βιβλιοθήκης `D3.js`, μπορεί να λειτουργήσει μόνο αν έχει ήδη ανακτηθεί, ελεγχθεί και διαβαστεί το αρχείο *Ontology.json*. Έτσι, ορίσαμε την υπηρεσία `DataService`, η οποία αναλαμβάνει τη διάθεση των δεδομένων σε όλα τα μέρη της εφαρμογής μας και να “πυροδοτήσει” τα σχετικά `Events`.

```
Object {
    "uuid": "c8bd7185-6349-4df5-8628-f87115222987",
    "name": "Ubuntu-Server-1",
    "label": "Server",
    "file": "Server.svg",
    "parent": "25e2ce69-2445-4b28-9a71-e7ca01bc57ea",
    "state": "OK",
    "lastUpdated": "2017-05-15 00:56:12",
    "description": "No issues were detected.",
    "slotAvailability": [false, true, true, true],
    "children": ["9c5d5b73-68ac-4a9d-92a7-7c60e492a7bd"],
    "info": {
        "OS": "Ubuntu 16.04.2",
        "brand": "IBM",
        "chassis_type": "Rackmount",
        "description": "Server asset",
        "form_factor": "7U",
        "model": "88861TU",
        "series": "BladeCenter S"
    },
    "metrics": {
        "cpu_load1": 0.5,
        "cpu_load5": 0.7,
    }
}
```

Listing 4: *Διεπαφή Χρήστη: Αντικείμενο εντός του Controller*

Κύκλος2: Οπτικοποίηση των SVG Μόλις ολοκληρωθεί επιτυχώς η φάση αρχικοποίησης η εφαρμογή μας ξεκινά να οπτικοποιεί το σύστημα-στόχο. Αρχικά, και πάλι η υπηρεσία `HttpService` ανακτά όλα τα SVG αρχεία που αντιστοιχούν στα αντικείμενα της Οντολογίας μέσω σχετικών αιτήσεων στο άκρο `onotmonHost:8080/resources` στο οποίο δέχεται αιτήσεις "ακούει" ο Εξυπηρετητής του OntoMon. Τα `.svg` αρχεία αυτά τα έχει "ανεβάσει" προηγουμένως ο προγραμματιστής ενσωμάτωσης, ενώ η ανάκτησή τους λαμβάνει χώρα κάθε φορά που φορτώνεται είτε η *overview* είτε η *assetview* σελίδα, σε περίπτωση που αυτά ανανεώθηκαν στο ενδιάμεσο.
Για την επίτευξη υψηλής επίδοσης αποφασίσαμε να παρέμβουμε απευθείας στο HTML DOM που παράγει η Angular στον περιηγητή ιστού και να προσθέτουμε εκεί δυναμικά τα SVG αρχεία, μεταχειρίζοντάς τα ως μπλοκ XML κώδικα. Η προσέγγιση αυτή μας επέτρεψε να κατασκευάσουμε προσαρμοσμένους HTML κόμβους κατά το χρόνο εκτέλεσης της εφαρμογής και να αποκτήσουμε πλήρη έλεγχο επί αυτών. Ωστόσο, το σημείο κλειδί κατά την οπτικοποίηση του συστήματος-στόχου ήταν η οργανωμένη διάσχιση του ιεραρχικού δέντρου της Οντολογίας, οι κόμβοι του οποίου είναι, πρακτικά, τα αντικείμενα του Controller. Συγκεκριμένα, με τη μέθοδο `traversal()` διασχίζουμε με BFS το δέντρο της Οντολογίας ανά επίπεδα, βρίσκουμε ποιο SVG αντιστοιχεί στον τρέχοντα κόμβο μέσω του `uuid` του και το προσθέτουμε ως φύλλο στο υπάρχον HTML DOM δέντρο του περιηγητή ιστού. Επιλέγοντας κατά βούληση τον κόμβο-αφετηρία για το BFS είναι εύκολο να απεικονίσουμε συγκεκριμένα δομικά μέρη του συνολικού συστήματος-στόχου, διατρέχοντας το αντίστοιχο υπόδεντρο.
Η παραπάνω διαδικασία παρέχει έναν αξιόπιστο και συνεπή τρόπο απεικόνισης της Οντολογίας του βασικού στρώματος μέσα στη Διεπαφή Χρήστη του OntoMon, προσθέτοντας τους SVG κόμβους με σωστή σειρά(πρώτα οι parent κόμβοι και στη συνέχεια οι child κόμβοι). Στο σημείο αυτό, ωστόσο, προέκυψε το ζήτημα της σωστής τοποθέτησης των SVGs μέσα στην εκάστοτε σελίδα της εφαρμογής μας, προκειμένου το συνολικό οπτικό αποτέλεσμα να είναι ουσιώδες και να αντοποκρίνεται στην πραγματικότητα, όπως αυτή καθορίζεται μέσα στην Οντολογία. Για την αντιμετώπιση του ζητήματος αυτού, εκμεταλλευτήκαμε τόσο την inline ιδιότητα `slot` που ορίστηκε από τον προγραμματιστή ενσωμάτωσης εντός του κώδικα των *.svg* αρχείων, όσο και την ιδιότητα `parent` των αντικειμένων της Οντολογίας. Τα βήματα που ακολουθούμε κατά την απεικόνιση ενός αντικειμένου είναι τα εξής: 1. εντοπισμός του αντικειμένου-πατέρα 2. έλεγχος της διαθεσιμότητας των `slots` του αντικειμένου-πατέρα 3. διάβα-

σμα της τιμής του πρώτου διαθέσιμου `slot`(σήμειο αναφοράς, διαστάσεις) 4. υπολογισμός μαθηματικών συντελεστών και εφαρμογή μετασχηματισμών στο `SVG` υπό τοποθέτηση(μετακίνηση και κλιμάκωση)



Σχήμα 7: *Αντιστοίχιση δέντρου Οντολογίας σε εμφωλευμένους `HTML DOM` κόμβους*

Κύκλος3: Ειδοποιήσεις σε Πραγματικό Χρόνο Κατά την τελευταία φάση της λειτουργίας της, η Διεπαφή Χρήστη του OntoMon επικεντρώνεται στην ποιοτική παρουσίαση της τρέχουσας κατάστασης του συστήματος-στόχου και την απεικόνιση της απόδοσης κατάστασης των δομικών του μονάδων. Το πρώτο βήμα για την επίτευξη των στόχων αυτών ήταν η ανάκτηση των μετρικών που συγκέντρωσε το μεσαίο στρώμα σε πραγματικό χρόνο. Ως εκ τούτου, ενσωματώσαμε στη Διεπαφή Χρήστη άλλη μια υπηρεσία, την `PollingService`, την οποία ρυθμίσαμε κατάλληλα ώστε να στέλνει περιοδικά `HTTP GET` αιτήματα στο άκρο `onotmonHost:8080/updates` του Εξυπηρετητή του OntoMon με σκοπό την ανάκτηση των πιο πρόσφατων ενημερώσεων κατάστασης σχετικά με το σύστημα-στόχο, όπως αυτές δημοσιεύονται από τον Παρακολουθητή της πλατφόρμας μας. Κάθε *Update.json* αντικείμενο που παραλαμβάνει η Angular εφαρμογή αντιστοιχεί σε μια ανανέωση κατάστασης ενός αντικειμένου της Οντολογίας και, συνεπώς, πρέπει να αντικατοπτριστεί εντός της Διεπαφής Χρήστη, προκειμένου να ενημερωθεί ο τελικός χρήστης.

Όσον αφορά τις μετρικές απόδοσης, η Διεπαφή Χρήστη διαβάζει το πεδίο `metrics` του *Update.json* αντικειμένου και αποθηκεύει τοπικά τις πιο πρόσφατες μετρήσεις απόδοσης. Στη συνέχεια, το `MetricsComponent` τις συγκεντρώνει σε ένα συνοπτικό πίνακα 2 στηλών, σε δομή κλειδιού-τιμής, ο οποίος προσφέρει μια γρήγορη εποπτεία στους διαχειριστές. Παράλληλα, το `GraphComponent` ενσωματώνει διαδραστικά διαγράμματα τα οποία αναπαριστούν με γραφικό τρόπο τα χρονοδεδομένα που κατέ-

γραψε το Icinga και αποθήκευσε η InfluxDB. Τα διαγράμματα αυτά κατασκευάζονται από τον Συνθέτη Γραφημάτων του υψηλού στρώματος και φορτώνονται δυναμικά μέσα στην κατάλληλη σελίδα της Διεπαφής Χρήστη, μέσω `HTTP` επικοινωνίας, όπως περιγράψαμε προηγουμένως. Ακόμη, φροντίσαμε να οπτικοποίησουμε σε ξεχωριστή σελίδα τα αρχεία καταγραφής(log files) τα οποία δημιουργεί ο OntoMon εξυπηρετητής, παρουσιάζοντας σε χρονολογική σειρά τις αλλαγές κατάστασης λειτουργίας των επιμέρους δομικών μονάδων του συστήματος-στόχου.
Η τελευταία, αλλά ίσως πιο σημαντική υπηρεσία της Διεπαφής Χρήστη αφορά την έγκαιρη παράδοση και εξατομικευμένη οπτικοποίηση των ειδοποιήσεων αλλαγής κατάστασης του συστήματος-στόχου. Για την αυτοματοποίηση της διαδικασίας αυτής, εισάγουμε τη μέθοδο `updateUI()`, η οποία εκτελείται ασύγχρονα, μόνο όταν καταφθάσει στο υψηλό στρώμα κάποια ενημέρωση από τον Παρακολουθητή του OntoMon, καταδεικνύοντας ότι η τρέχουσα όψη κάποιας δομικής μονάδας του συστήματος-στόχου πρέπει να ανανεωθεί. Αφού διαβαστεί το εκάστοτε *Update.json* αντικείμενο, και συγκεκριμένα τα πεδία `uuid`, `state` και `type`, αρχικά εμφανίζεται μια περιγραφική ειδοποίηση στο επάνω δεξιά μέρος της οθόνης, υποδεικνύοντας ποιό δομικό μέρος αντιμετωπίζει πρόβλημα. Στη συνέχεια, εντοπίζεται ο κατάλληλος `SVG` κόμβος του `HTML DOM` δέντρου του περιηγητή ιστού που αντιστοιχεί στο `uuid` της ενημέρωσης και, εντός αυτού, εφαρμόζονται στο στοιχείο κλάσης `indicator` οι μεταβολές ιδιοτήτων και τιμών που περιγράφονται στο *Update.json* αντικείμενο.

3.8 Σύνθεση Γραφημάτων με Grafana

Όπως αναφέραμε προηγουμένως, η ενσωμάτωση γραφικών παραστάσεων στη Διεπαφή Χρήστη είναι κομβικής σημασίας για τη συνολική εμπειρία του τελικού χρήστη, αλλά και την αποτελεσματικότερη διαχείριση της παρακολουθούμενης υποδομής. Προκειμένου να πετύχουμε μείωση του υπολογιστικού φόρτου για την Angular εφαρμογή, αναθέσουμε την κατασκευή των διαγραμμάτων απόδοσης του συστήματος-στόχου σε μια ξεχωριστή μονάδα λογισμικού, εγκαθιστώντας την πιο πρόσφατη έκδοση της Grafana(4.3) στον επιβλέποντα κόμβο(hypervisor) κόμβο. Στη συνέχεια προχωρήσαμε στη ρύθμιση των απαιτούμενων παραμέτρων προκειμένου ο Συνθέτης Γραφημάτων της πλατφόρμας μας να αποκτήσει τη ζητούμενη λειτουργικότητα. Αρχικά προσδιορίσαμε τις πληροφορίες για τη σύνδεση με το backend περιβάλλον αποθηκευσης χρονοσειρών, κάτω από τον κατάλογο `/etc/grafana/`, θέτοντας τη βάση InfluxDB ως την πηγή εισόδου χρονοδεδομένων:

```
   ...
# Protocol, access domain, port
protocol = http
domain = localhost
http_port = 3000
root_url = http://localhost:3000
# Basic AUTH for login
enabled = true
   ...
```

```
{
   "Name": "Cluster Metrics",
   "Default": true,
   "Type": "InfluxDB",
   "Url": "http://ontomonHost:8086",
   "Access": "proxy",
   "Enable_http_auth": false,
   "Details": {
      "Database": "icinga2",
      "User": "icinga2",
      "Password": "secret"
   }
}
```

Listing 5: *Ρύθμιση πηγής εισόδου στη Grafana*

Ο δαίμονας της Grafana επικοινωνεί απευθείας μέσω HTTP αιτημάτων με τον εξυπηρετητή της Influx, ανακτώντας τα πιο πρόσφατα δεδομένα σε τακτά χρονικά διαστήματα. Με τα δεδομένα αυτά χτίζει γραφήματα πραγματικού χρόνου, τα οποία ανανεώνονται καθώς έρχονται νέα δεδομένα, σκιαγραφόντας έτσι τη συμπεριφορά των επιμέρους δομικών μονάδων σε ένα δεδομένο χρονικό παράθυρο.

Το δεύτερο και πιο ουσιαστικό μέρος της ρύθμισης του Συνθέτη Γραφημάτων αφορούσε το στήσιμο των `panels` μέσα στο `dashboard` που εξάγει η Grafana. Κάθε `panel` εσωκλείει το διάγραμμα μιας συγκεκριμένης μετρικής απόδοσης και απαιτεί τον προσδιορισμό ενός InfluxQL ερωτήματος για την απόκτηση σχετικών χρονοδεδομένων. Τα InfluxQL ερωτήματα που διατυπώσαμε έπρεπε να είναι ευέλικτα και να προσαρμόζονται σε διαφορετικές δομικές μονάδες, καθώς δειγματοληπτήσαμε όλους τους κόμβους της συστοιχίας εικονικών μηχανών πάνω στις ίδιες μετρικές απόδοσης. Ακόλουθα, ορίσαμε `template` μεταβλητές(πχ `asset,interval`), τις οποίες συμπεριλάβαμε στα InfluxQL ερωτήματα ως παραμέτρους. Έτσι, καταφέραμε να προσαρμόσουμε τα ίδια ακριβώς `panels` στα χρονοδεδομένα διαφορετικών servers, αποφεύγοντας την επανάληψη πανομοιότυπου κώδικα. Μάλιστα, εκμεταλλευόμενοι το μεγάλο εύρος υποστηριζόμενων τύπων γραφημάτων(πχ bars,lines,histograms,singlestats,pie charts,gauges κλπ) οπτικοποιήσαμε αποδοτικά την απόδοση κάθε δομικού μέρους, ανάλογα με τον τύπο του. Τέλος, ρυθμίσαμε τη Grafana να εξάγει καθένα από τα `panels` αυτά μέσω ενός ξεχωριστού `URL`, αποσκοπώντας στην ενσωμάτωσή τους σε `<iframe>` στοιχεία της Διεπαφής Χρήστη του OntoMon.

4 Πειραματική Αξιολόγηση

Σε αυτήν την ενότητα παρουσιάζουμε την εμπειρία μας από την εφαρμογή του OntoMon στην πράξη. Ο στόχος μας ήταν να βεβαιωθούμε ότι η προτεινόμενη σχεδίαση είναι ικανή να αποφέρει τα ζητούμενα αποτελέσματα, αλλά και ότι οι ανοικτού κώδικα τεχνολογίες που επιλέξαμε συνεργάζονται αρμονικά. Επιπλέον, προκειμένου να πιστοποιήσουμε τη διαλειτουργικότητα της πλατφόρμας μας, μελετήσαμε 2 διαφορετικά σε λογική συστήματα-στόχους, τα οποία περιγράφουμε στη συνέχεια.

4.1 Βήματα Ενσωμάτωσης και Ελέγχου

Ιδανικά θα θέλαμε να δοκιμάσουμε τη λειτουργία του OntoMon σε ένα πραγματικό Κέντρο Δεδομένων μεγάλης κλίμακας, ωστόσο κάτι τέτοιο δεν ήταν εφικτό στα πλαίσια της παρούσας εργασίας. Ως εκ τούτου, το περιβάλλον το οποίο χρησιμοποιήσαμε για τις δοκιμές μας ήταν η συστοιχία των 3 εικονικών μηχανών που εγκαταστήσαμε κατά τη διαδικασία της υλοποίησης στον επιβλέποντα κόμβο. Επιπρόσθετα, λαμβάνοντας υπ'όψην ότι το OntoMon εστιάζει σε υπολογιστικά συστήματα τόσο στο επίπεδο του υλικού, όσο και λογισμικού, καθώς και ότι τα 2 σενάρια παρακολούθησης που μελετήσαμε δεν ήταν αντικροούμενα, αλλά συμπληρωματικά, ήταν δυνατό να χρησιμοποιήσουμε τις ίδιες ακριβώς εικονικές μηχανές και στις 2 περιπτώσεις, μειώνοντας τους χρόνους ανάπτυξης και εξοικονομώντας πόρους στο φυσικό μηχάνημα. Τα βήματα για την ενσωμάτωση των συστημάτων-στόχων και την πραγματοποίηση του ζητούμενου ελέγχου συνοψίζονται παρακάτω:

1. Αρχικά συντάσσουμε μια Οντολογία σε μορφή `JSON` η οποία ακολουθεί τις σχεδιαστικές αρχές που παρουσιάσαμε και περιγράφει αναλυτικά τις οντότητες και τις σχέσεις του συστήματος-στόχου που πρόκειται να παρακολουθήσει το OntoMon

2. Στη συνέχεια, δρώντας ως προγραμματιστές ενσωμάτωσης συστήματος, προχωράμε στον καθορισμό των υπηρεσιών παρακολούθησης, ελέγχου και ειδοποιήσεων στο μεσαίο στρώμα. Συγκεκριμένα, προσδιορίζουμε τα αντικείμενα που απαιτεί το Icinga και υλοποιούμε τις αντίστοιχες συναρτήσεις-χειριστές εντός του Παρακολουθητή του OntoMon.

3. Μόλις βεβαιωθούμε ότι οι επιμέρους μονάδες συνεργάζονται αποτελεσματικά, εισερχόμαστε στη Διεπαφή Χρήστη του OntoMon ως τελικοί-χρήστες και αναμένουμε μια αντιπροσωπευτική οπτική αναπαράσταση του συστήματος-στόχου, καθώς και της τρέχουσας κατάστασης των δομικών του στοιχείων που παρακολουθεί η πλατφόρμα μας.

4. Τέλος, με σκοπό την επαλήθευση της αποτελεσματικότητας του μηχανισμού παράδοσης ειδοποιήσεων σε πραγματικό χρόνο, παρεμβήκαμε στην κανονική λειτουργία του συστήματος-στόχου και προσομοιώσαμε καταστάσεις αποτυχίας ή υψηλού φόρτου στις οποίες μπορεί να βρεθεί το σύστημα-στόχος, προσδοκώντας σχετικές ενημερώσεις κατάστασης εντός της Διεπαφής Χρήστη του OntoMon.

4.2 Παρακολούθηση Φυσικής IT Υποδομής

Σε αυτήν τη δοκιμή ο στόχος μας είναι η παρακολούθηση και οπτικοποίηση της φυσικής(hardware) υποδομής στο εσωτερικό ενός Κέντρου Δεδομένων. Επειδή τα σύγχρονα Κέντρα Δεδομένων εσωκλείουν πολυάριθμες φυσικές οντότητες, επιλέξαμε να

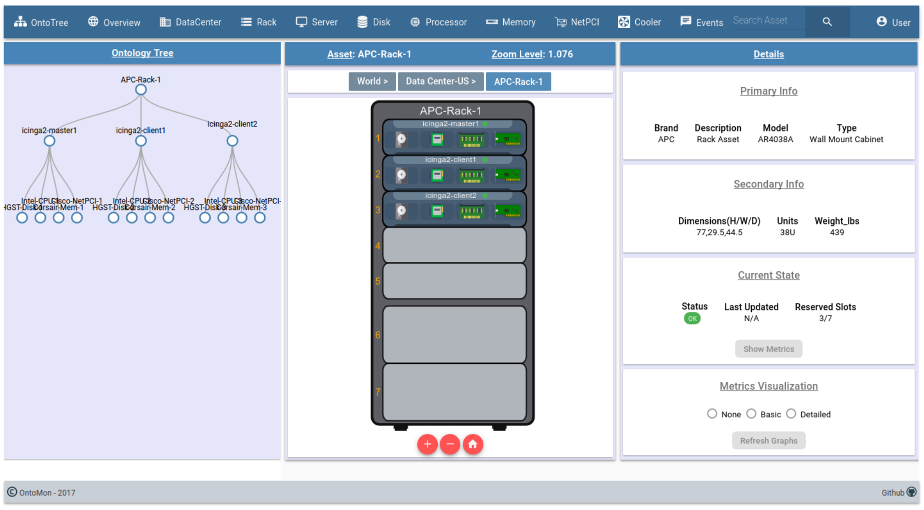
εστιάσουμε στις σημαντικότερες από αυτές, ορίζοντας την ακόλουθη ιεραρχία: Ρίζα, Κόσμος, Κέντρα Δεδομένων, Ράφια, Εξυπηρετητές, Μονάδες Επεξεργασίας, Δίσκοι, Μνήμες και Κάρτες Δικτύου. Θεωρούμε πως οι φυσικές οντότητες αυτές επαρκούν για την ολοκληρωμένη δειγματοληψία ενός υπολογιστικού συστήματος σε επίπεδο υλικού, παρουσιάζοντας τις δυνατότητες της πλατφόρμας μας. Η οργάνωση του συγκεκριμένου συστήματος-στόχου βασίστηκε σε πληροφορίες που βρήκαμε σχετικά με τη δομή των Κέντρων Δεδομένων στο διαδίκτυο, και ακολούθως πραγματοποιήσαμε μια αντιστοίχιση των πραγματικών φυσικών μονάδων σε εικονικές.
Η Οντολογία που συντάξαμε για αυτή τη δοκιμή περιέχει 1 Κέντρο Δεδομένων, 2 racks, 5 εξυπηρετητές και πολυάριθμα αντικείμενα επεξεργαστών, μνημών, δίσκων και καρτών δικτύου. Ως τελικοί χρήστες αποφασίσαμε να παρακολουθήσουμε την υποδομή αυτή ως προς τους εξυπηρετητές, για τους οποίους ορίσαμε τις ακόλουθες υπηρεσίες παρακολούθησης στο Icinga:

Physical Asset	Icinga Services	Metrics
CPU	`check_load, check_procs, cpu_stats`	`procs num, cpu usage, average load(1m/5m/15m)`
Disk	`check_disk, io_stats`	`disk usage & capacity per partition, reads/s, writes/s, tps, iowait`
Memory	`check_memory`	`memory usage & capacity, swap usage & capacity`
Network	`check_traffic, check_http, check_ssh, cluster_zone`	`transmit/s & receive/s per interface, hostalive, ssh time, connection to cluster`

Πίνακας 2: *Υπηρεσίες παρακολούθησης για τη φυσική υποδομή*

Όσον αφορά τον Παρακολουθητή του OntoMon, πρώτον τροποποιήσαμε το `observer.py` module για να προσδιορίσουμε τα ονόματα των servers που πρόκειται να παρακολουθήσουμε αλλά και να καλέσουμε τις αντίστοιχες συναρτήσεις-χειριστές. Δεύτερον, επεκτείνοντας το module `services.py` υλοποιήσαμε τις συναρτήσεις `memory_check()`, `disk_check()`, `cpu_check()` και `network_check()`, στο σώμα τον οποίων βρίσκεται η λογική και οι έλεγχοι για τις ενημερώσεις κατάστασης. Σχετικά με την οπτι-

κοποίηση των ειδοποιήσεων αλλαγής κατάστασης εντός της Διεπαφής Χρήστη, αρκεστήκαμε σε μια απλή αλλαγή χρώματος(κόκκινο, πράσινο, κίτρινο) του `indicator` στοιχείου του αντίστοιχου `SVG` κόμβου, ανάλογα με την τρέχουσα κατάσταση λειτουργίας του(`OK`,`WARNING`,`CRITICAL`).
Στη συνέχεια εισήλθαμε στη Διεπαφή Χρήστη του OntoMon μέσω ενός εξυπηρετητή ιστού, ώστε να δούμε αν η προσφερόμενη οπτικοποίηση ανταποκρίνεται στις απαιτήσεις του υποκείμενου συστήματος-στόχου. Πράγματι, παρατηρούμε την εμφωλευμένη οργάνωση των `SVGs` στη σελίδα της εφαρμογής, ενώ κάθε δομική μονάδα βρίσκεται στην αναμενόμενη θέση. Αυτό αποδεικνύει την ορθότητα των υπολογισμών και των μετασχηματισμών που εφάρμοσε η Angular εφαρμογή κατά την επεξεργασία του δέντρου της Οντολογίας και την προσθήκη των αντίστοιχων κόμβων στο `DOM`.
Στην κορυφή ομαδοποιούνται τα αντικείμενα της Οντολογίας σε μενού, στα αριστερά φαίνεται το ιεραρχικό δέντρο της Οντολογίας του συστήματος-στόχου, ενώ στα δεξιά ο τελικός χρήστης βλέπει επιπρόσθετες πληροφορίες σχετικά με τη δομική μονάδα που απεικονίζει η σελίδα στην οποία έχει πλοηγηθεί.



Primary Info

Brand	Description	Model	Type
APC	Rack Asset	AR4038A	Wall Mount Cabinet

Secondary Info

Dimensions(H/W/D)	Units	Weight_lbs
77,29.5,44.5	38U	439

Current State

Status	Last Updated	Reserved Slots
OK	N/A	3/7

Show Metrics

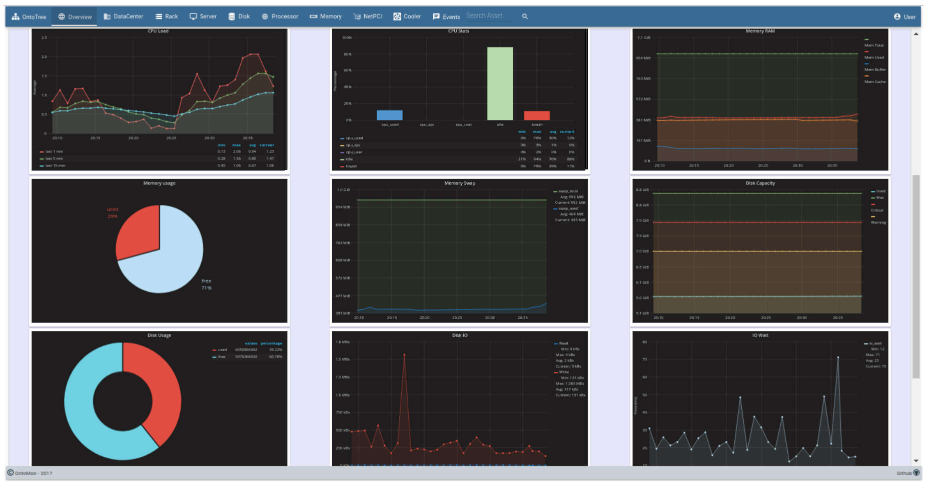
Metrics Visualization

None Basic Detailed

Refresh Graphs

Σχήμα 8: *Διεπαφή Χρήστη: οπτικοποίηση φυσικής υποδομής*

Για την πραγματοποίηση του ελέγχου του μηχανισμού ειδοποιήσεων, χρησιμοποιήσαμε το εργαλείο `stress(1)` του Unix, το οποίο είναι ικανό να γεννήσει και να επιβάλλει φόρτο εργασίας σε υπολογιστικούς κόμβους. Πιο συγκεκριμένα, συνδεθήκαμε μέσω `ssh` σε μια από τις 3 εικονικές μηχανές του νοητού Κέντρου Δεδομένων μας και επιβαρύναμε τη λειτουργία της με ασυνήθιστα μεγάλο φόρτο εργασίας ως προς τον επεξεργαστή, τη μνήμη και το δίσκο. Αναμένουμε, λοιπόν, ενημέρωση της κατάστασης του συγκεκριμένου εξυπηρετητή σε `CRITICAL` όσο διαρκεί το stress τεστ, επα-

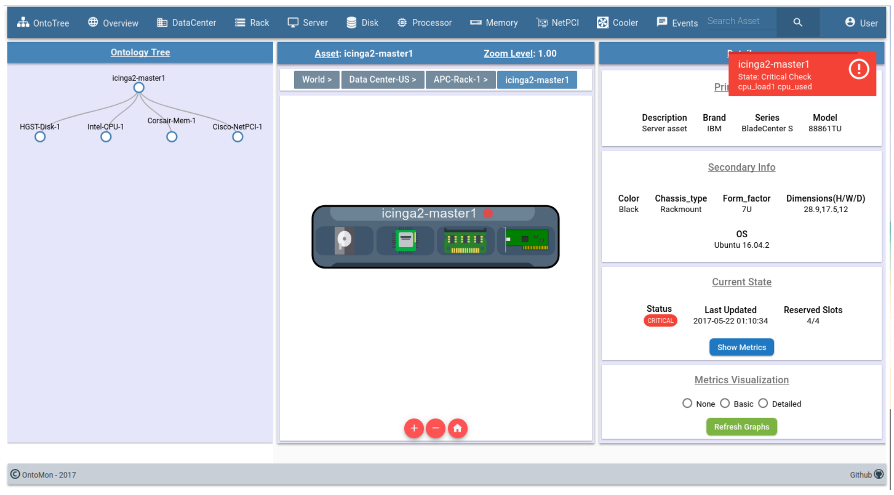


Σχήμα 9: *Διεπαφή Χρήστη: γραφική αναπαράσταση μετρικών απόδοσης*

ναφορά σε κατάσταση OK όταν ολοκληρωθεί, και σχετικές ειδοποιήσεις στο UI του OntoMon. Όπως φαίνεται και στα στιγμιότυπα που παραθέτουμε, η πλατφόρμα μας επιβεβαιώνει τις υποθέσεις μας στην πράξη, παραδίδοντας ειδοποιήσεις σε πραγματικό χρόνο οι οποίες ανταποκρίνονται στην τρέχουσα κατάσταση της παρακολουθούμενης υποδομής. Έτσι, διευκολύνεται σημαντικά το έργο της διαχείρισης του Κέντρου Δεδομένων.

```
# spawn 8 workers spinning on sqroot() for 45 seconds
stress --quiet --cpu 8 --timeout 45
# spawn 8 workers spinning on malloc()/free() for 45 seconds
stress --quiet --vm 8 --vm-bytes 256M --timeout 45
```

Listing 6: *Command to stress hardware assets*



Asset: icinga2-master1 **Zoom Level**: 1.00

icinga2-master1
State: Critical Check
cpu_load1 cpu_used

Description	Brand	Series	Model
Server asset	IBM	BladeCenter S	88861TU

Secondary Info

Color	Chassis_type	Form_factor	Dimensions(H/W/D)	OS
Black	Rackmount	7U	28.9,17.5,12	Ubuntu 16.04.2

Current State

Status	Last Updated	Reserved Slots
CRITICAL	2017-05-22 01:10:34	4/4

Metrics Visualization

Σχήμα 10: *Διεπαφή Χρήστη: ειδοποίηση κατάστασης CRITICAL*

4.3 Παρακολούθηση Πλατφόρμας επιπέδου Λογισμικού

Στη δεύτερη δοκιμή μας αποφασίσαμε να εστιάσουμε στην παρατήρηση ενός υπολογιστικού συστήματος θεμελιωμένου στο επίπεδο του λογισμικού(software) και όχι του υλικού, προκειμένου να βεβαιωθούμε ότι το OntoMon είναι πράγματι ικανό να διαχειριστεί ετερογενή συστήματα μέσω της αφηρημένης μοντελοποίησης αντικειμένων. Με αυτόν τον τρόπο μπορέσαμε να βγάλουμε χρήσιμα συμπεράσματα σχετικά με το εύρος των συστημάτων-στόχων στα οποία μπορεί να ενσωματωθεί το OntoMon. Η πλατφόρμα λογισμικού που επιλέξαμε να παρακολουθήσουμε ήταν ένα Ceph cluster, δηλαδή μια συστοιχία κόμβων πάνω στους οποίους τρέχει το Ceph, ένα σύστημα λογισμικού για την κατανεμημένη και κλιμακώσιμη αποθήκευση δεδομένων. Ένα Ceph cluster αποτελείται από πολυάριθμες οντότητες λογισμικού, όπως Monitors, OSDs και MDSs, καθεμιά από τις οποίες προσφέρει συγκεκριμένες υπηρεσίες στο cluster και υλοποιείται με ένα Ceph δαίμονα-διεργασία. Παράλληλα, η οργάνωση της εσωτερικής του δομής βασίζεται στον ορισμό Pools, Placement Groups και Objects, η παρακολούθηση των οποίων κρίνεται αναγκαία για την επίβλεψη ενός Ceph cluster. Το βασικό χαρακτηριστικό του Ceph είναι ότι πετυχαίνει υψηλή επίδοση ακόμη και χωρίς εξειδικευμένες συσκευές αποθήκευσης σε επίπεδο υλικού, ενώ οι λειτουργίες πυρήνα σχετικά με τη λήψη αποφάσεων αλλά και την αποθήκευση, τοποθέτηση και αναπαραγωγή των αντικειμένων μέσα στο object store του(RADOS) πραγματοποιούνται σε επίπεδο λογισμικού σύμφωνα με τον αλγόριθμο *CRUSH*. Χρησιμοποιώντας το εργαλείο `ceph-deploy` εγκαταστήσαμε το Ceph στην ήδη υπάρχουσα συστοιχία εικονικών μηχανών, ορίζοντας 1 Monitor κόμβο για το συντονισμό των λειτουργιών εντός του cluster και 2 OSD κόμβους για την εξυπηρέτηση των αιτήσεων αποθήκευσης και ανάκτησης αντικειμένων από το RADOS. Δεδομένου ότι το Ceph cluster μας αποτελείται από 2 OSDs και εξ'ορισμού διατηρεί παραπάνω από ένα αντίγραφα κάθε αντικειμένου στο object store, ορίσαμε τον παράγοντα αναπαραγωγής(replication factor) του ίσο με 2 προκειμένου να είναι εφικτό να φτάσει σε `active+clean` κατάσταση.
Όπως και στο πρώτο σενάριο, το πρώτο βήμα για την ενσωμάτωση του OntoMon στο συγκεκριμένο σύστημα-στόχο περιλαμβάνει τη σύνταξη ενός *Ontology.json* αρχείου, δηλαδή μιας αυστηρώς δομημένης Οντολογίας η οποία να περιγράφει αναλυτικά την εσωτερική του οργάνωση. Η συγκεκριμένη Οντολογία εισάγει την εξής ιεραρχία: **Ρίζα,Κόσμος,Κέντρα Δεδομένων,Συστοιχίες,Κόμβοι,Ceph-Monitors** και **Ceph-OSDs**. Πιο συγκεκριμένα, η τοπολογία μας αποτελείται από 1 Κέντρο Δεδομέ-

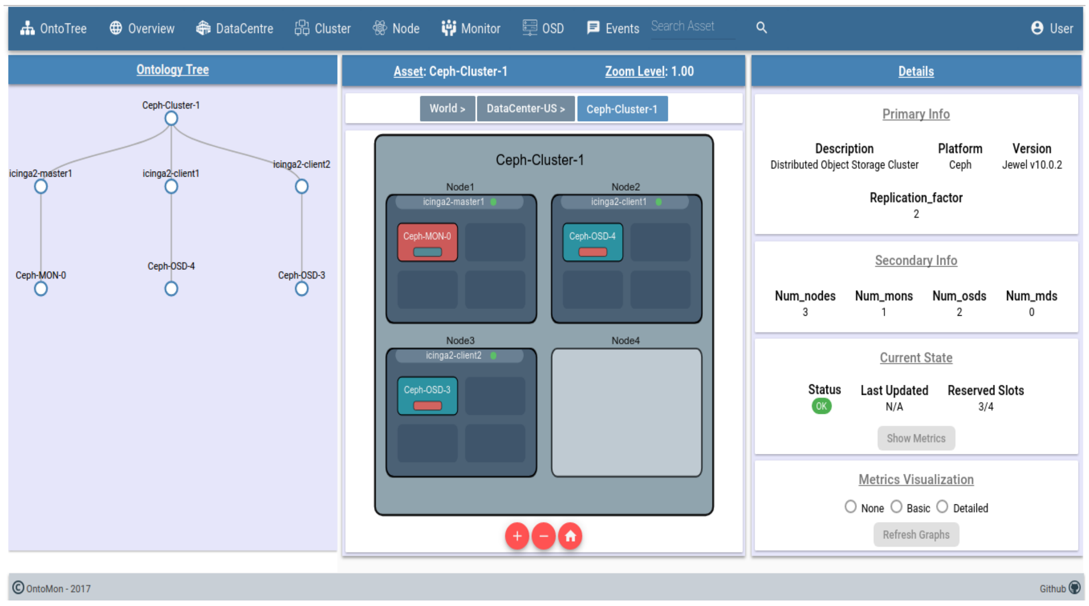
νων, 1 Ceph cluster, 3 Υπολογιστικούς Κόμβους, 1 Ceph-Monitor και 2 Ceph-OSDs. Σε αυτό το σημείο αξίζει να σημειώσουμε πως αντί για το Ceph, θα μπορούσαμε να παρακολουθήσουμε οποιαδήποτε άλλη πλατφόρμα λογισμικού, αρκεί να υπήρχε διαθέσιμη μια συμβατή οντολογική περιγραφή της. Μόλις ολοκληρωθεί η σύνταξη της Οντολογίας επιλέγοντας τα επιθυμητά `SVG` αρχεία για την αναπαράσταση των δομικών μονάδων λογισμικού, τα αποστέλλουμε στον Εξυπηρετητή του OntoMon. Ακόλουθα, δρώντας ως προγραμματιστές ενσωμάτωσης, επεκτείνουμε το μεσαίο στρώμα του OntoMon και ορίζουμε στο Icinga τις παρακάτω υπηρεσίες ελέγχου, οι οποίες δειγματοληπτούν λεπτομερώς το Ceph cluster καταγράφοντας τις κατάλληλες μετρικές απόδοσης για το συμπερασμό της τρέχουσας κατάστασης και συμπεριφοράς του:

Software Asset	Icinga Services	Metrics
Monitor	`check_ceph_daemon, check_ceph_mon`	`daemon_state,monmap_-epoch,mon_-total,quorum_-in,quorum_out`
OSD	`check_ceph_daemon, check_ceph_osd`	`daemon_state,read_-bytes_sec,write_-bytes_sec,in_osd,out_-osd,up_osd,down_-osd,apply_lat,commit_-lat`
Cluster	`check_ceph_health, check_ceph_df`	`health_-status,epoch,num_-pgs,pg_-states,objects_per_-pool,total_GB,total_-used_GB,raw_used_pct`

Πίνακας 3: *Υπηρεσίες παρακολούθησης για τη υποδομή λογισμικού*

Ταυτόχρονα, τροποποιούμε και πάλι το βασικό σκελετό του Παρακολουθητή του OntoMon, με σκοπό τον ορισμό των αντίστοιχων συναρτήσεων-χειριστών. Επί παραδείγματι, στο module `observer.py` καθορίσαμε τα `FQDN` των κόμβων του Ceph cluster τους οποίους πρόκειται να παρακολουθήσει η πλατφόρμα μας, ενώ φροντίσαμε να συμπεριλάβουμε και τις κλήσεις των συναρτήσεων ελέγχου με τα κατάλληλα ορίσματα. Εν συνεχεία, επεκτείναμε το `services.py` module υλοποιώντας τις συναρτήσεις-χειριστές **`check_ceph_health(), check_ceph_daemon(), check_ceph_df(), check_-ceph_mon()`** και **`check_ceph_osd()`**, οι οποίες παραλαμβάνουν τα σχετικά χρονοδεδομένα, επιτελούν συγκρίσεις και τελικά συμπεραίνουν τα επίπεδα απόδοσης του συστήματος-στόχου.

Όπως και προηγούμενα, εισερχόμαστε στη Διεπαφή Χρήστη του OntoMon και επιβεβαιώνουμε πως η μηχανή οπτικοποίησης λειτούργησε με απόλυτη επιτυχία. Τα αντικείμενα της Οντολογίας εμφανίζονται εμφωλευμένα στις σελίδες της εφαρμογής, παρουσιάζοντας μια γενική εικόνα του παρακολουθούμενου συστήματος λογισμικού. Συγκεκριμένα, η πλοήγηση στη σελίδα `/assetview/Ceph-Cluster-1` απεικονίζει συνολικά το Ceph cluster και προσφέρει ένα dashboard για την παρακολούθησή του σε πραγματικό χρόνο. Για να ελέγξουμε την αποκρισιμότητα της Διεπαφής Χρήστη, χρησιμοποιήσαμε το ενσωματωμένο εργαλείο `rados bench` του Ceph, το οποίο προσομοιώνει μαζικές εγγραφές και διαβάσματα αντικειμένων στο Ceph cluster. Πράγματι, τα αντίστοιχα διαγράμματα που ενσωματώνονται από τον Συνθέτη Γραφημάτων ανανεώθηκαν αυτόματα, παρουσιάζοντας ακριβείς τιμές απόδοσης, όπως αυτές προέκυψαν από τη δοκιμή μας.



Asset: Ceph-Cluster-1 Zoom Level: 1.00

Primary Info

Description	Platform	Version
Distributed Object Storage Cluster	Ceph	Jewel v10.0.2

Replication_factor: 2

Secondary Info

Num_nodes	Num_mons	Num_osds	Num_mds
3	1	2	0

Current State

Status	Last Updated	Reserved Slots
OK	N/A	3/4

Metrics Visualization: None Basic Detailed

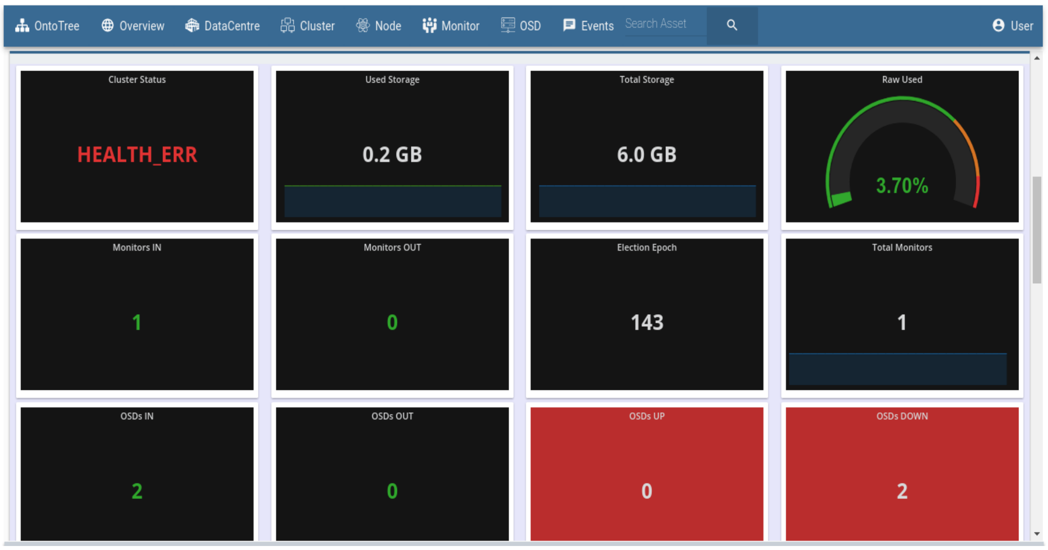
Σχήμα 11: *Διεπαφή Χρήστη: οπτικοποίηση υποδομής λογισμικού*

Τέλος, αποφασίσαμε να προσομοιώσουμε μια κατάσταση αποτυχίες του συστήματος-στόχου, ώστε να βεβαιωθούμε για την ετοιμότητα του μηχανισμού ειδοποιήσεων. Ως εκ τούτου, συνδεθήκαμε σε έναν OSD κόμβο του Ceph cluster και σταματήσαμε χειροκίνητα την αντίστοιχη διεργασία.

```
osd@icinga2-client1:~$ systemctl stop ceph-osd@4.service
osd@icinga2-client1:~$ systemctl start ceph-osd@4.service
```

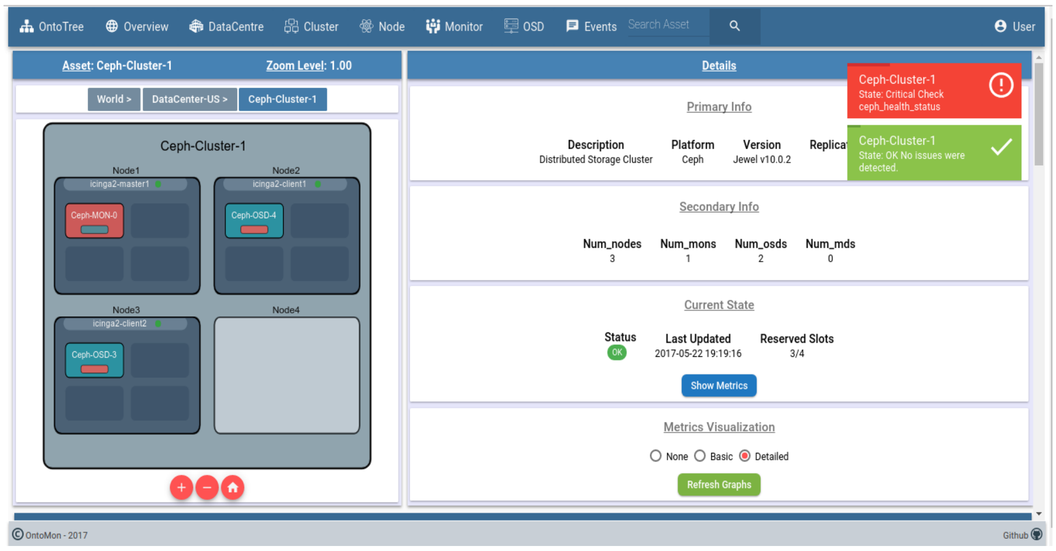
Listing 7: *Χειρισμός ceph-osd δαίμονα*

Επειδή, πλέον, οι ενεργοί OSDs του συστήματος-στόχου είναι 1 και όχι 2, η υγεία του Ceph cluster αναμένεται να λάβει την τιμή `HEALTH_ERR`. Ωστόσο, μόλις “ξανασηκώσουμε” το συγκεκριμένο `osd-daemon` περιμένουμε νέα ενημέρωση κατάστασης και επαναφορά της υγείας του Ceph cluster σε `HEALTH_OK`. Τα επόμενα στιγμιότυπα επιβεβαιώνουν όλες τις υποθέσεις που περιγράψαμε και πιστοποιούν ότι η αφηρημένη μοντελοποίηση που χρησιμοποιεί το OntoMon για τις υπηρεσίες παρακολούθησης και οπτικοποίησης έχουν πράγματι εφαρμογή σε ετερογενή υπολογιστικά συστήματα.



Cluster Status	Used Storage	Total Storage	Raw Used
HEALTH_ERR	0.2 GB	6.0 GB	3.70%
Monitors IN	Monitors OUT	Election Epoch	Total Monitors
1	0	143	1
OSDs IN	OSDs OUT	OSDs UP	OSDs DOWN
2	0	0	2

Σχήμα 12: *Διεπαφή Χρήστη: Ceph dashboard και ενδείξη σφάλματος*



Asset: Ceph-Cluster-1 Zoom Level: 1.00

Details

Primary Info

Description: Distributed Storage Cluster
Platform: Ceph
Version: Jewel v10.0.2
Replica

Secondary Info

Num_nodes: 3
Num_mons: 1
Num_osds: 2
Num_mds: 0

Current State

Status: OK
Last Updated: 2017-05-22 19:19:16
Reserved Slots: 3/4

Metrics Visualization

None Basic [✓ Detailed]

Ceph-Cluster-1
State: Critical Check ceph_health_status

Ceph-Cluster-1
State: OK No issues were detected.

Σχήμα 13: *Διεπαφή Χρήστη: επιτυχής επαναφορά κατάστασης Ceph cluster*

5 Συζήτηση-Συμπεράσματα

Σύνοψη

Στην παρούσα εργασία είχαμε την ευκαρία να μελετήσουμε εργαλεία διαχείρισης υποδομής Κέντρων Δεδομένων και να εξοικιωθούμε με τις έννοιες της παρακολούθησης και οπτικοποίησης υπολογιστικών συστημάτων μεγάλης κλίμακας σε πραγματικό χρόνο. Το κίνητρό μας να αναπτύξουμε τη δική μας πλατφόρμα, το *OntoMon*, ήταν κυρίως η έλλειψη εργαλείων ανοικτού κώδικα που να είναι ικανά να διαχειριστούν ταυτόχρονα ετερογενή συστήματα προσανατολισμένα είτε στο υλικό, είτε στο λογισμικό. Συγκεκριμένα, υλοποιήσαμε ένα ευέλικτο και προσαρμόσιμο πλαίσιο λογισμικού(framework) παρακολούθησης, το οποίο δεν εξαρτάται από το εννοιολογικό περιεχόμενο του εκάστοτε συστήματος-στόχου.

Το βασικό χαρακτηριστικό του OntoMon είναι ότι βασίζει την οργάνωση και τις εσωτερικές λειτουργίες του σε μια Οντολογία, δηλαδή σε μια ολοκληρωμένη και αυστηρώς δομημένη περιγραφή του εκάστοτε συστήματος-στόχου. Η αρχιτεκτονική σχεδίαση που αναλύσαμε περιλαμβάνει 3 ξεχωριστά στρώματα που συνεργάζονται και ενσωματώνουν πολυάριθμες κλιμακώσιμες τεχνολογίες ανοικτού κώδικα. Συγκεκριμένα, το OntoMon χρησιμοποιεί το Icinga ως συλλέκτη μετρικών απόδοσης πραγματικού χρόνου, την InfluxDB ως βάση αποθήκευσης χρονοσειρών, τη Grafana για τη σύνθεση γραφημάτων και την Angular για την ανάπτυξη μιας διαδικτυακής διεπαφής χρήστη. Παράλληλα, προκειμένου να καταστεί δυνατή η παρακολούθηση και η οπτικοποίηση του συστήματος-στόχου σε πραγματικό χρόνο, καθώς και η αποστολή ενημερώσεων στον τελικό χρήστη σχετικά με την τρέχουσα κατάσταση λειτουργίας του, αναπτύξαμε ορισμένες μονάδες διασύνδεσης που συνδέεουν τα στρώματα και καθιστούν δυνατή την προσαρμογή της πλατφόρμας μας στη συλλογιστική παρακολούθησης που επιθυμεί ο τελικός χρήστης. Όλες οι ενδιάμεσες επικοινωνίες βασίζονται σε καλώς ορισμένα APIs τα οποία αναπαριστούν την πληροφορία σε μορφή `JSON`.Η πλατφόρμα μας στοχεύει κυρίως σε κατανεμημένα συστήματα με πολυάριθμους κόμβους, παρέχοντας μια συνοπτική και ταυτόχρονα αντιπροσωπευτική εικόνα της υποδομής υπό παρακολούθηση. Μάλιστα, η ενσωμάτωση διαδραστικών γραφημάτων στη Διεπαφή Χρήστη διευκολύνει τον εντοπισμό σημείων συμφόρησης και συντελεί στη βαθύτερη κατανόηση των αλληλεπιδράσεων των επιμέρους μονάδων του συστήματος-στόχου. Ακόμη, ο προσαρμοσμένος στις ανάγκες του χρήστη μηχανισμός παράδοσης ειδοποιήσεων που αντιπροσωπεύουν αλλαγές στην κατάσταση λειτουργίας, αποτυπώνει τις πιο πρόσφατες αλλαγές ή τα γεγονότα που συνέβησαν στο σύστημα-στόχο σε ένα συγκεκριμένο χρονικό πλαίσιο.

Προκειμένου να πιστοποιήσουμε ότι το OntoMon πετυχαίνει αποτελεσματικά το σκοπό για τον οποίο σχεδιάστηκε, μελετήσαμε 2 ετερογενή συστήματα τα οποία περιγράψαμε με αντίστοιχες Οντολογίες, τις οποίες δώσαμε ως είσοδο στην πλατφόρμα μας. Στο πρώτο σενάριο, χρησιμοποιήσαμε το OntoMon για να ανιχνέυσουμε την οργάνωση και την επίδοση της φυσικής υποδομής ενός νοητού Κέντρου Δεδομένων. Από την άλλη, στο δεύτερο σενάριο αποφασίσαμε να εγκαταστήσουμε και να μελετήσουμε την απόδοση ενός κατανεμημένου συστήματος αποθήκευσης δεδομένων σε επίπεδο λογισμικού, και συγκεκριμένα την υγεία ενός Ceph cluster.

Και στις 2 περιπτώσεις η πλατφόρμα μας έδωσε τα αναμενόμενα αποτελέσματα όσον αφορά την παρακολούθηση, την οπτικοποίηση και την παράδοση ειδοποιήσεων στο χρήστη σε πραγματικό χρόνο. Συγκεκριμένα, το OntoMon ανταποκρίθηκε ικανοποιητικά σε όλα τα ερεθίσματα που του δώσαμε και σε όλες τις προσομοιώσεις πραγματικού περιβάλλοντος που πραγματοποιήσαμε. Κατά συνέπεια, τόσο η προτεινόμενη σχεδίαση, όσο και η υλοποιήση μας κρίνονται επιτυχημένες στα πλαίσια της εκπόνησης της παρούσας εργασίας. Παρόλο που ο στόχος μας δεν ήταν η ανάπτυξη μιας πλατφόρμας προορισμένης για χρήση στην παραγωγή, θεωρούμε πως το OntoMon εισάγει μια εναλλακτική προσέγγιση στον τομέα της παρακολούθησης υπολογιστικών συστημάτων, πάνω στην οποία θα μπορούσε να βασιστεί η ανάπτυξη επαγγελματικών

IT εργαλείων παρακολούθησης στο μέλλον.

Μελλοντικές Επεκτάσεις

Αν και η πλατφόρμα μας ήδη υποστηρίζει τις πιο βασικές δυνατότητες παρακολούθησης και διαχείρισης πολυδιάστατων συστημάτων-στόχων, είναι σαφές πως υπάρχουν αρκετά περιθώρια βελτίωσης της ποιότητας των παρεχόμενων υπηρεσιών. Τα μελλοντικά μας σχέδια σχετικά με το OntoMon, ακολούθως, αφορούν:

- **Αυστηρότερος έλεγχος της Οντολογίας:** στην παρούσα σχεδίαση, οι έλεγχοι εγκυρότητας και οι περιορισμοί που επιβάλλει το προτεινόμενο οντολογικό σχήμα για την περιγραφή του συστήματος-στόχου είναι κάπως περιορισμένοι. Μια λύση θα μπορούσε να είναι η ενσωμάτωση του `JSON-Schema` στο βασικό στρώμα `http://json-schema.org/`, ενός λεξιλογίου που υποστηρίζει την εισαγωγή σχολίων(annotations) μέσα σε `JSON` αντικείμενα. Έτσι, θα μπορούσε να πραγματοποιηθεί ενδελεχής έλεγχος ως προς τη δομή και το συντακτικό της Οντολογίας, καθώς και να προστεθούν μεταδεδομένα για λόγους πληρότητας.

- **Υποστήριξη πολλαπλών backend βάσεων:** προς το παρόν η πλατφόρμα μας υποστηρίζει μόνο τη βάση χρονοσειρών InfluxDB. Ωστόσο, θα μπορούσαμε να επεκτείνουμε το μεσαίο στρώμα ώστε να υποστηρίζονται και άλλες τεχνολογίες για την αποθήκευση χρονοσειρών, όπως TSTB, Elasticsearch και Prometheus, δίνοντας στον τελικό χρήστη τη δυνατότητα να επιλέξει όποια επιθυμεί.

- **Ενεργητική διαχείριση υποδομής:** μια σημαντική προσθήκη στις παρεχόμενες υπηρεσίες του OntoMon θα μπορούσε να είναι η υλοποίηση ενός μηχανισμού που θα συμπληρώνει τον υπάρχοντα μηχανισμό ειδοποιήσεων και ο οποίος θα επέτρεπε την άμεση παρέμβαση του τελικού χρήστη στο σύστημα-στόχο μέσω της Διεπαφής Χρήστη. Συγκεκριμένα, ο τελικός χρήστης αφού ενημερωθεί για την τρέχουσα κατάσταση λειτουργίας της υποδομής θα μπορεί να προχωρήσει σε σχετικές ενέργειες διαχείρισης(πχ κλείσιμο server, επανεκκίνηση υπηρεσιών δικτύου, ρύθμιση δαιμόνων λογισμικού κλπ) οι οποίες θα αναφέρονται σε συγκεκριμένες δομικές μονάδες. Μάλιστα, η διαδικασία αυτή μπορεί να αυτοματοποιηθεί εκπαιδεύοντας το OntoMon να πραγματοποιεί αυτόνομα τις επιθυμητές ενέργειες μέσω αλγορίθμων μηχανικής μάθησης. Για παράδειγμα, εάν ένας δίσκος παρουσιάζει αργή απόκριση και υψηλές θερμοκρασίες, είναι αρκετά πιθανό να προκληθεί απώλεια δεδομένων στο άμεσο μέλλον, οπότε το OntoMon θα μπορούσε να προχωρήσει αυτόνομα στην αποσύνδεσή του από το λειτουργικό σύστημα με σκοπό την αντικατάσταση ή τη συντήρησή του.

- **Ασφάλεια:** στα πλαίσια της εργασίας αυτής δε θέσαμε σε πολύ υψηλή προτεραιότητα την ασφάλεια των επικοινωνιών και την κρυπτογράφηση των δεδομένων που καλείται να διαχειριστεί το OntoMon. Ωστόσο στις μέρες μας και τα 2 αυτά ζητήματα είναι βαρύνουσας σημασίας, προκειμένου να εξασφαλίζεται η ακεραιότητα και η αξιοπιστία ενός σύνθετου IT συστήματος. Ακόλουθα, σχεδιάζουμε την ενσωμάτωση εξειδικευμένων υπηρεσιών ασφάλειας σε όλα τα στρώματα του OntoMon.

- **Βελτιωμένος χειρισμός αρχείων καταγραφής:** όλα τα σύγχρονα υπολογιστικά συστήματα καταγράφουν τόσο το ιστορικό τους, όσο και σημαντικά γεγονότα ή αποτυχίες αναφορικά με τη λειτουργία τους σε αρχεία καταγραφής. Υποστηρίζουμε πως είναι δυνατό να εξαχθεί χρήσιμη πληροφορία σχετικά με τη συμπεριφορά της κάθε δομικής μονάδας και των μεταξύ τους σχέσεων από τα αρχεία αυτά, εφόσον εφαρμοστεί κατάλληλου επιπέδου ανάλυση. Για το λόγο αυτό σκοπεύουμε να ενσωματώσουμε ένα εργαλείο όπως το Logstash στο OntoMon, αφού η πλατφόρμα μας περιέχει αρκετά συστήματα επιπέδου λογισμικού που δημιουργούν αρχεία καταγραφής. Πιο συγκεκριμένα, το Logstash είναι ικανό να συναθροίσει, να διαβάσει και να μετασχηματίσει στην επιθυμητή μορφή σύνθετα και πολυάριθμα αρχεία καταγραφής καθώς αυτά δημιουργούνται.

1 Introduction

In this chapter, we outline the scope of our work. We first provide a quick overview of the problem we are trying to solve and argue about its importance. Next, we shortly describe some existing solutions and highlight potential problems in achieving their goal. We move on to illustrate our proposed design and how it fits in as a solution, meeting all requirements. Finally, we conclude with an early preview of some promising results and the structure of the document.

1.1 Problem Statement

The primary objective of this thesis is the design and implementation of a versatile, general-purpose, real-time monitoring and visualization platform, named *OntoMon*, that integrates well with heterogeneous computing systems. The proposed framework is **content-agnostic**, in the sense that it runs equally well on more than one target systems, either hardware- or software-oriented, without relying on their semantics. OntoMon is founded upon a formal ontological description of the target system that is about to be monitored and introduces an abstract object model to represent all entities. Besides, in order to facilitate the supervision and management of IT infrastructure, it supports a dynamic and highly customizable Web User Interface, that is generated on-the-fly and offers a holistic overview of the monitored target system.

With the prevalence of cloud and distributed computing, constant instrumentation and proactive administration of components inside large-scale deployments has become a necessity in order to preserve high performance and quality of services. In large scale environments, such as Data Centers that host multi-node computer clusters, it is common that administrators have to deal with dense and diverse computing systems that need constant supervision, due to asset failures, performance deterioration or under-utilization of resources. Thus, close attention must be paid to the operation of the underlying infrastructure, referring to both physical devices and software services. Although existing monitoring solutions fullfill their goal, most of them are designed to operate on specific target systems and have a rather narrow spectrum of application. Thereby, the ultimate goal of OntoMon is to offer an interoperable and unified monitoring solution, that alleviates the problem of developing system-specific monitoring software to manage individual computing platforms. In this regard, the proposed framework is designed to timely detect asset-related issues or abnormal behavior, and deliver respective real-time notifications to the end-user.

1.2 Motivation

In the era of Big Data, the exponential production rate of digital information, together with the needs for data consistency and availability pose a challenging task on both the academia and the industry: efficiently storage and processing of data in the petascale. Therefore, it is essential that Data Centers, as well as all the entire IT infrastructure that is housed within them, operates smoothly and seamlessly, in order to achieve high levels of performance and sufficiently cover the current needs of the IT market.

Furthermore, the worldwide demand for new and more powerful IT-based applications, combined with the economic benefits of consolidating software and physical assets, has led to the expansion of Data Centers in both size and complexity. Modern Data Centers are more dense than ever, housing a huge variety of computing platforms, varying from Cloud and Virtualization environments, to IoT applications. However, in recent years, the environmental laws impose serious space, energy and power limitations on Data Centers, making their administration even more sophisticated than before. As improvements in processing engines, networking topologies and storage equipment continue along their impressive paths, the IT community has recognized the significant importance of efficient tracing and orchestration of the inner components of large systems. Consequently, rise has been given to the development of new toolset in the field of IT, referred to as DCIM [1]. These tools are fully dedicated on the supervision of Data Center infrastructure, providing real-time asset management and integrated services to fulfill their design purposes.

Today, the area of DCIM is rapidly emerging, aiming at enhancing the efficiency, agility and robustness of Data Centers. The majority of storage vendors and enterprises are largely investing in this field, mainly driven by economy-of-scale benefits and the endeavour of improving storage technologies. Data Center asset management encompasses more than simply locating a specific asset. It additionally involves gaining a deeper insight into the behavior of individual devices and services, by correlating their configuration with real-time performance metrics. Once properly deployed, DCIM solutions provide administrators with direct and clear visibility of the underlying infrastructure, such as server functionality, network connectivity or software availability. Data center assets, either hardware- or software-defined, can be conveniently located and instrumentated. In this way, provisioning of new assets can be timely scheduled, so that system downtimes can be reduced, as the latter are usually translated in unpredictable costs for the enterprise. For those IT organizations that consider device intelligence and service agility as top priorities, the effective usage of DCIM software is becoming a strategic necessity. Increasingly, a wide range of software tools and accelerated subsystems are combined to offer enhanced IT monitoring, management, and control, with regard to an accurate, real-time, end-to-end view of IT service delivery. Furthermore, DCIM software allows for efficient management of risks, distribution of workloads away from failure points, and more cost-effective deployments.

Taking the aforementioned into consideration, it is essential to build a versatile and adaptable DCIM framework that is capable of profiling different kinds of assets inside modern Data Centers, regardless of their type. Hence, real-world entities inside Data Centers should be represented in an abstract manner, while their visualization should provide a holistic and intuitive overview of their current state of operation. In this regard, the preferences of the end-user must be reflected, while avoiding the definition of content-specific objects with static types or classes. The more generic and versatile the implementation, the broader the area of application. Of course the traditional features of DCIM software can be preserved or extended, allowing administrators and engineers to make well-informed decisions about planning, forecasting and driving

[1]Data Center Infrastructure Management

automated systems to manage resources. In this context, we argue that visualization of the Data Center layout, graphical representation and analysis of real-time performance metrics, along with respective push notifications and alerts will definitely lead to a more reliable, secure and sustainable operating environment.

1.3 Existing Solutions

Today, there is a great variety of DCIM software solutions available on the market of IT, promising to face common system administration challenges. Varying from proprietary, closed-source tools to free, open-source solutions, most of them introduce some core features, such as *asset discovery and tracking, real-time data collection, capacity planning, performance evaluation, alerting* and others.

Despite their satisfactory performance and impressive capabilities, we argue that the integration of DCIM software into the enterprise should adopt a different logic regarding design. To begin with, the majority of most high-end monitoring solutions are entirely commercial and closed-source. While these platforms are, generally, efficient and qualitative, such as Sunbird [2] and Hyperglance [3], it is claimed that closed-source monitoring solutions discourage end-users from deeply understanding their inner architecture and components, resulting in non-customized, "black box" deployments. In addition, solely depending on proprietary DCIM software can lead to vendor lock-in, resulting in hardware conflicts or incompatibilities. At the same time, the open-source alternatives are rather stationary and admittedly limited in number, employing outdated technologies. Most of them are based on the LAMP [4] stack, such as openDCIM [1] and Racktables [2]. Only a handful of them are actively developed today, like Netbox [3], a promising DCIM tool currently designed by Digitalocean which greatly inspired us to implement our own custom monitoring platform.

We also suggest that there is room for an alternative approach regarding the modeling of the monitored infrastructure inside DCIM software. The majority of current solutions are founded upon a rather conservative representation of supervised assets, explicitly introducing static methods or type-specific handlers to manipulate

[2]`https://www.sunbirddcim.com/`
[3]`https://www.hyperglance.com/`
[4]Linux Apache MySQL PHP

them. Knowing beforehand the internal structure and specific information about the basic building blocks of the target system, they propose class-oriented architectures and encourage strong typing of internal objects. Therefore, their implementation is intimately coupled with the semantics of the targeted system, while its expansion is time-consuming. DCIM solutions abiding by these principles are obliged to define specific attributes and behavior of individual components in advance, according to their type, ruling out reusability and interoperability of code. Thus, hard-coded support is needed throughout their implementation. In essence, every object is treated as an instantiation of a pre-defined asset prototype, with standard characteristics that are usually determined by the respective software vendor. This design decision decisively narrows the breadth of supported target systems and use cases. Furthermore, it poses serious limitations regarding the level of customization by the end-user, so that specialized needs can be served, either in the monitoring or the visualization layer. In our work, we reject the approach of immutable objects and classes; instead, we propose generic and dynamic manipulation of abstract objects, along with programmatic control and definition of well-structured APIs.

Besides, since OntoMon attempts to monitor diverse target systems, we decided to structure it upon *Ontologies*. To our knowledge, there are no existing DCIM software solutions based on Ontologies, in terms of determining the structure and concepts of the monitored target system. We argue that mathematical Ontologies are a powerful tool that can be effectively used to define the entities and relations of the infrastructure being monitored. An Ontology provides a unified, yet interoperable structure to describe heterogeneous target systems, comprising of different assets. Enabling end-users to generate the respective ontological description of their system and providing it as input to OntoMon allows for modular design and great adaptation to personalized administrator needs. Therefore, we regard Ontologies as the cornerstone of the proposed design, offering support for dynamic content and higher customization capabilities, as it is further discussed later on.

Last but not least, after studying various DCIM platforms, we concluded that many of them are rather complex in terms of structure and have multiple dependencies in their deployment, as they attempt to incorporate diverse services into a single solution. Meanwhile, most of them deliver a pre-configured User Interface for the supervision of the infrastructure, that is built upon the static representation of the monitored assets, resulting in reduced flexibility and expressivity. Conversely, the proposed framework is based on a multi-level architecture, introducing 3 discrete layers that successfully integrate open-source technologies to offer functionality. The User Interface provided by OntoMon is fully-dynamic, generated on-the-fly based on the ontological description of the target system and is highly customizable in terms of asset visualization and alerting.

1.4 Results

In order to evaluate the overall functionality of OntoMon, we configured a testing environment to simulate real Data Center infrastructure and operations. We studied 2 different use cases, involving both hardware- and software-based monitoring. Therefore, we needed a testbed suitable for both cases, in order to showcase the versatility of our platform with regard to the target systems it can manage. Thus, we set up a 3 node VM cluster. The target system of each use case was represented by a different Ontology, that was provided to OntoMon as input. As such, we assumed a notional Data Center topology containing various assets and introduced the following scenarios:

- **Monitoring the physical infrastructure of a Data Center:**
  In this scenario, the Ontology describes the hardware assets of the notional Data Center. In our test we designedly imposed heavy computational workloads on the physical components of specific nodes belonging to our virtual cluster and waited for the corresponding updates in the Web UI of OntoMon. Indeed, we observed an automatic UI update concerning the visualization of the respective asset, indicating its performance degration. Alongside, a push notification was delivered offering additional details about the issue that occured.
- **Monitoring the software components of a software-defined Cluster:**
  In this scenario, we initially deployed a Ceph distributed storage cluster on the existing virtual cluster. This time, the Ontology described the hierarchy of the software assets inside the Ceph cluster, while the Web UI of OntoMon accommodated a Ceph-specific supervision dashboard. As in the previous scenario, we interfered in the normal operation of the cluster and forced a Ceph storage daemon to stop. Subsequently, we verified that OntoMon delivers respective

notifications and the automatic visual updates in the User Interface, indicating the failure.

1.5 Thesis Structure

The structure followed in this thesis is described below:

- **Chapter 2:** presentation of the theoretical background and concepts that our application is founded upon
- **Chapter 3:** analysis of the core architectural principles and design decisions from a higher-level perspective
- **Chapter 4:** demonstration of the focal points of our implementation, reference to the problems we faced during the development process, as well as the proposed workarounds
- **Chapter 5:** evaluation and testing of our solution
- **Chapter 6:** concluding remarks and possible future extensions

2 Background

In this chapter, we provide all the necessary background knowledge required to fully understand the design and implementation of the monitoring platform proposed in this work. Initially, we review the basic principles of Ontologies along with their integration in modern software systems. In the next section, we thoroughly analyze the concept of System monitoring and inspect the main features of the Icinga distributed monitoring platform. After that, we discuss the importance of distributed storage solutions today, present their most important aspects and concentrate on the structure of the Ceph storage sluster. Furthermore, we provide an overview of the web technologies we used to build the User Interface of our OntoMon. More specifically, we provide an inclusive description of Scalable Vector Graphics and debate why this web standard could be part of the future web. Last but not least, we examine the contribution of Web Frameworks in the field of web development and outline the architecture of a typical Angular application.

2.1 Ontologies

2.1.1 Definition

The word "Ontology" generates a lot of controversy and has been given different definitions in the literature. Etymologically, the compound word "Ontology" combines *onto-*, which derives from the Greek *ὄν, ὄντος*(="being"), present participle of the verb *εἰμί*(="be"), and *-logia* from *λόγος*(="logical discourse"). In this thesis, we adopt the

definition given by Thomas R. Gruber in [5]: ”In the context of knowledge sharing, the term ontology means an explicit specification of a conceptualization”. Meanwhile, in discussion to [6] he states: ”In the context of computer and information sciences, an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse. The representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members). The definitions of the representational primitives include information about their meaning and constraints on their logically consistent application”.
A conceptualization denotes an abstract semantic structure that is used to encode implicit knowledge of a certain domain. Accordingly, its specification takes the form of the definitions of representational vocabulary(classes, relations, and so forth), which provide meanings for the vocabulary and formal constraints on its coherent use. Essentially, an Ontology is a formal description of the entities, objects, concepts, and relationships that are presumed to exist among them in a given area of interest, for an agent or a community of agents. Ontologies can be regarded as taxonomic hierarchical views of the world that we wish to represent for some purpose. They concentrate on providing a notion of shared understanding upon the perception of a domain of knowledge, delivering a controlled and well-structured vocabulary. There is also an expectation that the features of the proposed model in an Ontology should closely resemble the real world. An Ontology is characterized by a common syntax(symbols, expressions), explicit semantics, hierarchical organization of concepts and content reasoning.

2.1.2 Types

Today, Ontologies are closely associated with automated reasoning and are identified as the basic building blocks of modern knowledge systems. Scientists in the field of artificial intelligence categorize Ontologies in the following types:

- Domain Ontologies
- Upper Ontologies
- Hybrid Ontologies

Domain Ontologies A domain Ontology focuses on a specific part of the world and attempts to model its basic concepts by providing particular meanings of the terms applied to that domain. The representation of domain Ontologies is rather specific and eclectic. Thus, incompatibility issues may occur among different Ontologies that describe the entities and relations of the same domain. With the expansion of systems that are founded upon domain Ontologies, it is common to merge multiple narrow domain Ontologies into a broader representation of concepts. It is also frequent that great diversity exists among Ontologies dealing with the same domain, depending on description languages, intended usage, and, ultimately, human perception of the same domain. Present-day systems are capable of automatically merging domain Ontologies that are founded upon the same top-level Ontology. This is feasible because they use the same set of general-purpose terms to specify the semantics of entities and relations inside the respective domain. This automation feature is of significant importance in the field of information systems, since manual merging domain Ontologies is a very complex and time-consuming process.

Upper Ontologies An upper Ontology (also known as top-level or foundation Ontology) is an Ontology that encloses general purpose terms, such as "object", "property", "relation" "function" etc, that are common across multiple domains. Upper Ontologies are conceptually well-founded and use a semantically transparent standard glossary. To represent the meaning of terms, Ontologies contain categories that are organized in an *is-a* hierarchy, ultimately forming a *taxonomy*. The key role of an upper Ontology is to provide an application-independent description of the world and guarantee interoperability among a large number of domain-specific ontologies, thus facilitating the integration of semantics. For this, a common ontological foundation is proposed upon which the domain specific Ontologies can formulate their definitions. Therefore, the terms described in domain Ontologies can be regarded as specialized extensions of the general purpose terms defined in an upper Ontology. In essence, the entities and attributes introduced by upper Ontologies must be abstract enough to allow specification of concepts in multiple and diverse domains. This indicates that domain specific Ontologies can share a common starting point. In this way, the development of new Ontologies is greatly facilitated and simplified. In addition, upper level Ontologies are regarded as richly axiomatized, since they apply formal restrictions and rules upon the abstract terms they introduce in the form of axioms. These restrictions are, subsequently, inherited by the domain Ontologies and determine their core structure. Hence, upper level Ontologies provide the means to verify the consistency and the conditions of domain specific Ontologies, mainly in terms of the general categories upon which they are founded. Some of the most widely accepted Upper Ontologies are DOLCE [1], SUMO [2], and GFO [3].

Hybrid Ontologies A hybrid Ontology, as its name suggests, is a special type of ontology that borrows features from both domain and upperOontologies. It includes an "upper" part that defines concepts, terms and relations and a "lower" part that is more domain-specific and provides high extensibility. Hybrid Ontologies usually offer rich semantics and high expressivity.

2.1.3 Core Components

Every Ontology is assumed to comprise of numerous different components. The names of these components may differ among Ontologies, depending on the representation language used to formulate them or the purpose of creation. Despite this, most Ontologies share a typical structure and set of basic building blocks. To this end, we list the most important ontological components below, as stated in `Wikipedia` [7]:

- **Individuals:** the basic, "ground level" components of an Ontology. They can be either concrete or abstract.
- **Classes:** abstract groups, sets or collections of objects that classify individuals, other classes, or a combination of both. Ontologies can vary based on whether classes can contain other classes, a class can belong to itself, there is a universal class, etc. In general, classes can be very different in terms of attributes and content. It is also possible that a class inside an Ontology can be subsumed by another class; the former is referred to as the subsuming class(or supertype), while the latter as a subclass (or subtype) of the former.

[1] http://www.loa.istc.cnr.it/old/DOLCE.html
[2] http://www.adampease.org/OP/
[3] https://en.wikipedia.org/wiki/General_formal_ontology

- **Attributes:** aspects, properties, features, characteristics or parameters, that designate objects or classes. Each attribute can be a class or an individual, while its value of can be a simple or complex data type.
- **Relationships:** formal specifications that determine how objects or entities are related to each other. Commonly, a relation is subject to a specific type or class that describes the essence of the specified connection between the objects. Relations usually retain a uni- or bi-directional form of expression (e.g. *ObjectA is defined as child of Object B*), that intuitively describes the semantics and dependencies inside the domain. An important type of relation is the subsumption relation, defining classified objects.
- **Restrictions:** Formally stated descriptions of what must be true in order for some assertion to be accepted as input. Mostly attached to relations.
- **Rules:** Statements in the form of an *if-then* sentence that describe the logical inferences that can be drawn from an assertion in a particular form.
- **Axioms:** Assertions in a logical form that together with the rules, comprise the overall theory that the Ontology describes in the application domain.

2.1.4 The Semantic Web

The Semantic Web [8] represents the evolution of the Web as we know it today and is an initiative mainly supported by the World Wide Web Consortium. The Semantic Web aims at providing extensions of the current web standards so that computer systems can automatically process and integrate information and resources available on the Web, via trusted network interactions. In this direction, collaborative efforts have been made to enrich abstract data on the Web with explicit meaning and semantics, in order to dissolve ambiguities and avoid collisions. Therefore, the development of modern data sharing technologies, as well as the specification of strong vocabularies that involve rules, axioms and data handlers, are absolute necessities.

The W3C [4] community regards domain Ontologies as the cornerstone of the Semantic Web, offering a technological stack to facilitate its definition and development. The aforementioned stack consists of numerous languages that are designed to pro-

[4]`https://www.w3.org/`

vide a meaningful description of data on the Web, such as RDF[5], OWL [6] and XML [7]. Since its birth, the Web has greatly depended on hyperlinks and `HTML` documents, a markup convention capable of integrating both text and multimedia objects into its code. Combinations of these languages can be used in order to supplement the current and future Web documents with additional semantics or specifications in terms of content. One can regard this as an extension of `HTML`-based resources, as well as Web-accessible databases. What's more, this approach outlines the foundation of Linked Data, a concept intimately related to the Semantic Web. In practice, the Semantic Web already applies to web resources, by embodying machine-readable metadata about cross-referenced web content. More specifically, it standardizes format and determines how web resources are related to each other. In this manner, fully-automated agents can mimic human deductive reasoning and inference, efficiently inferring facts or retrieving information for web endpoints.

The Semantic Web relies on a common framework that allows data to be shared and reused across interconnected applications and web services. Its ultimate goal is the integration of semantics into large scale data and, in the long run, the aggregation of knowledge spread across the Web. In the Semantic Web, it is quite common to use the term "Vocabulary" instead of "Ontology". Although there is no clear distinction between an Ontology and a Vocabulary, the former is used for complex and formal collection of terms, while the latter involves looser or occasional formalism. The main roles of Ontologies, in the applications of the Semantic Web are summarized below:

- **Contribution to data integration**, by formal definitions and dissolution of ambiguities

- **Organization of knowledge**, by effectively managing large collections of formalized data

- **Definition of metadata terms**, by determining the core semantics of the web content

[5]Resource Description Framework
[6]Web Ontology Language
[7]Extensible Markup Language

2.1.5 Why use an Ontology?

The most important reasons for developing an Ontology are given below:

Analyzing domain knowledge
As stated in [9], "ontological analysis clarifies the structure of knowledge. Given a domain, an Ontology forms the heart of any knowledge representation system for that domain, capturing its intrinsic conceptual structure. These conceptualizations are required so that a vocabulary for representing knowledge can be formulated. Thus, the first step in devising an effective knowledge representation system, and vocabulary, is to perform an effective ontological analysis of the field, or domain." Unless a declarative and formal specification of entities and relations existing in a domain of knowledge is given, no coherent or extensive analysis can be performed. As a result, weak or dubious analyses might lead to conflicting or inconsistent knowledge bases. For example, ontological analysis of a domain can prove the distinction between a category and a role.

Sharing of common understanding
Undeniably, one of the most important reasons for generating an Ontology is the distribution of knowledge in a specific domain and the common understanding of concepts among both humans and software agents. Given a thorough analysis on a specified area of interest, one can arrive at a representational set of conceptualizations of entities and their inner relations. Hence, the resulting Ontology comprises of a well-defined syntax that encodes all knowledge related to the domain and associates terms with concepts. Accordingly, this representation language (or vocabulary) can be shared with the community, so that anyone having similar needs or is interested in the respective field can refer to specific concepts in a standardized manner. In addition, sharing of Ontologies on the Web encourages the extraction, aggregation and query of information by both end-users and services, acting as a single source of truth for the given domain of knowledge. However, while a commitment to an Ontology guarantees consistency, it does not guarantee the completeness, with respect to queries and assertions using the vocabulary defined in the Ontology.

Enable reuse of domain knowledge

The dominance of Ontologies throughout time is certainly due to their capability of reusing domain knowledge. In scientific research, it is quite common that different domains need to represent the same or similar notions. Let's assume that numerous groups of researchers wish study the same knowledge domain. As long as a formal ontological representation of the respective domain exists, all groups of researchers can simply base their work upon it or extend it to meet their specialized needs. Therefore, shared Ontologies can be used as the foundation upon which new domain-specific Ontologies can be built. In this direction, the duplication of the domain-specific analysis procedure is eliminated, while redundancy is greatly reduced. Meanwhile, integrating or investigating pre-defined Ontologies along with the corresponding vocabularies can prove both time- and effort-saving for scientists.

Explicit domain assumptions

Since the perception of the world constantly changes with time, it is essential that knowledge systems and related platforms are kept updated with the latest knowledge. As Ontologies make explicit assumptions about the domain of study, they provide a convenient mechanism for mutating these assumptions at any time. Hard-coded most knowledge assumptions in the fields of programming and computing systems are quite difficult to locate, grasp and update. Contrariwise, the knowledge assumptions made in Ontologies are rather straightforward and held in a single domain-specific repository, simplifying the update process even for inexperienced users.

Separating domain from operational knowledge

Ontologies are commonly used to separate domain knowledge, which refers to the concepts in the area of interest, from operational knowledge, which refers to the understanding of the manipulation mechanisms. By design, Ontologies belong to the semantic specification level and are, generally, independent of data modeling strategies and platform- or language-specific implementations. They provide the required concept specifications and the theoretical foundation, upon which programs or systems are built. In this manner, interoperability is facilitated, as heterogeneous services are capable of integrating the same formalized domain knowledge representation.

2.1.6 Ontology-based Software Systems

It is widely admitted, that the most significant role of Ontologies, with regard to information systems and software developing, is to determine a well-structured data model capable of representing concepts in an abstract manner. This model is independent of specific logical or physical design patterns and aims at facilitating data acquisition, translation, refinement and uniformal sharing among seemingly unrelated systems or services. In particular, Ontologies are widely applicable in applications that query, cross-check or merge information from diverse sources across the Web. Today, an increasing number of software systems are built upon a common, converged, universally accepted representation of domain knowledge in the form of ontological descriptions. Such practice results in the definition of strongly linked tools and mechanisms that encourage reusability of components and simplified maintenance in the future.

The majority of software systems and applications undertake specific tasks and offer dedicated services to end-users, usually related to a well-specified domain of knowledge. Given an ontological description of this particular domain, developers can benefit from a formalized model in the development process, which enables acceleration of iterations and optimization of individual components, as well as the interplay between them. Specifically, an Ontology introduces specific vocabulary, semantics and logic related to the domain and, thus, it improves system consistency and coherence. Besides these, such model can optimize architecture, automate processes via integration of information and contribute to a deeper understanding of the workload distribution inside the application, leading to efficiency improvements on the software side. Modern ontology-based design patterns involve specification of dynamic parts, reusability of software entities and automatic translation of high-level models into executable programs. In this direction, the development of a software system that relies on Ontologies focuses on having as few hard coded components as possible and deriving everything from a single source. In most cases, a concrete user interface consists of a subset of repeated interface elements, having different meaningful components. As a result, the number of procedural components developed in the course of designing an interface becomes considerably less. Meanwhile, specificying the underlying concepts of the model could provide a solid basis for future extensions. Hence, evolving the system is then a matter of updating this input source and having the updates propagate through the rest of the system.

Bearing in mind the purpose of this thesis, along with the fact that, nowadays, User Interfaces are regarded as core components of modern software systems, we emphasize our analysis on the design of ontology-based User Interfaces. Multiple efforts have been made aiming at bridging the gap between the generation of Ontologies and software engineering methodologies. For the sake of completeness, we demonstrate the approach described in [11], so that the reader can gain some intuition on the subject.

In this paper, an ontology-based framework is proposed, aimed at dynamically building User Interfaces. In general, ontology-based tools provide an alternative approach to software development, compared to the traditional model-based one. Their main feature is the automatic translation of the declarative, high-level models of interface components into executable programs. Most model-based approaches to the development of User Interfaces suffer from certain limitations. Mainly, they require the definition of individual, targeted, statically modeled components, discouraging the prevalence of a universally accepted standard. As a result, components that are identical by meaning and effect might have different names or conflicting interpretations, according to the terminology used to define them. Also, for every model definition, specific principles and mechanisms are used, imposing difficulties in the process of linking and transfering among models. Finally, a detailed description of the application program must be provided, making the development and maintenance even more difficult. Therefore, ontology-based approaches extend model-based solutions by introducing the following concepts and individual components:

- Information and attributes regarding each model of a user interface must be represented in the form of an ontology model. This model might need to be modified if conditions of using software change.
- Universally accepted ontological models must be widely available for use and maintenance. Thus, models of different domain ontologies can be hosted on the Web, in order to facilitate further development and integration of semantics in applications.
- There is a clear distinction between the core application and the corresponding User Interface, in the sense that they should be independently designed and implemented. However, a concrete communication layer must be defined for their interaction with a common set of variables. This architecture separates the tasks of

model description and linking and reduces development and maintainability costs.

- The designer of the User Interface should be provided with a set of tools aimed at the implementation, integrating inputs, functions and UI elements reflecting the requirements of users.

Domain Concept System Model The domain concept system provides the notional background regarding the broader domain of interest. In particular, it comprises of formal and well-structured definitions of the core concepts of the domain, along with their inter-relations. Explicit representation of these definitions leads to a domain Ontology. Thus, the domain concept model can be reduced to a domain Ontology that offers a well-formulated terminology to study behavior and correspondence of entities. A standard structure is usually followed, aiming at general use and simplification of the design of UIs. Besides, the domain concept system model is responsible for appropriately expressing inputs and outputs of the system, intellectually decoding the actions of users.

The Display Model User Interfaces usually consist of a presentation layer, an interaction layer and the respective relations among objects. The display model is closely related to the concept system models, in terms of content and information, as well as to the application program model, in terms of variables and values. Any interaction between the end-user and the application model is carried out by views and display aspects defined in the display model, transmitting information in both directions. Hence, there is a correspondance between the domain concept system and the display aspects of the UI. Modifications made during the life cycle of the system, either to the concept system or the application program, must be mapped to the display model of the interface. Essentially, the display aspects can be regarded as formal definitions relying on the concept system model. Their structure, description and purpose must be general, while they should also be extensible, to support new objects and properties.

The Application Program Model The application program of a software system is the core component that is responsible for solving tasks based on the given input. Any process that is fulfilling a task can be considered as part of the application program. Ideally, the application program should contain minimal information, boiling down to the determination of a set of variables used to exchange information with the UI layer. Description of the application program variables can be related to the domain concept system, the implementation language and mathematical terms. The design of the application program model is inseparable from the design of the interface.

2.1.7 Representation

Ontologies are typically formulated in languages which are closer in expressive power to logical formalisms, such as the predicate calculus. These languages allow abstraction from data structures and implementation strategies. This enables the Ontology designers to impose semantic constraints without forcing any particular encoding strategy. Thereafter, Ontologies are ranked at the *semantic* level, whereas database schemas are models of data at the *logical* or *physical* level. Subsequently, languages used to formulate ontologies differ from languages used to model databases.

An Ontology can be regarded as a set of relation definitions between terms. In practice, ontological descriptions involve formal "is-a", "part-of", "connected-to" relations, as well as class taxonomies and class-instance relations. The relations inside an Ontology are formal, in the sense that they are expressed in some formal description language. The main goal of description languages is to guarantee conceptual clarity and consistency. Accordingly, combining relations can lead to implicit conclusions, given that certain terms are related. Higher level approaches provide a set of tools and mechanisms to implement an Ontology, such as frames or simplified description logics. Languages belonging to this level allow the definition of classes, instantiation of objects and building of relations among the. Yet, some designers prefer even more formalized tools, so they choose fully-constrained logical theories to represent ontological relations. Such theories contain modal, first or higher order logics to express intended usage of terms and inter-connections. When working with informally described Ontologies, it is very difficult to automate processing in computing systems, since these are mainly based on natural language descriptions. While more intuitive and simplified, less formal description of concepts does not represent the objective of depending on Ontologies in software systems. Conversely, strict, standardized and well-formatted descriptions can be automatically processed by reasoners and encode their intended meaning. Furthermore, sharing and extending Ontologies that contain formally structured definitions is a lot easier than manipulating abstract ones. Best practices suggest that a correct modeling language should be used depending on the application, that would not limit the designers during the development process and would allow a high-level of expressivity and detail.

What really matters is the utilization Of ontologies to provide the representational machinery with which to instantiate domain models in knowledge bases, make queries to knowledge-based services, and represent the results of calling such services. Having a central, interoperable, reusable, and comprehensive metamodel during the development of a software system proves useful when flexibility and interoperability are needed. Additionally, sticking to widely accepted standards in representing metamodels and transformations, greatly facilitates access and manipulation of data.

2.2 System Monitoring

2.2.1 Overview

In the extremely fast-paced world of IT, where interacting with large and complex computing systems is the norm, monitoring the underlying infrastructure is the key for the effective supervision and administration of every platform. Today's data centers are more dense and complex than ever, containing physical, virtual, cloud-based and legacy servers. Monitoring services are capable of reliably modeling information and statistics about these assets in a single data structure, so that it can be actively studied and analyzed based on numerous uses cases and scenarios. This process outlines the overall behavior of individual components in a large system, along with their inner relations. It also enables humans to track down weaknesses, improve security and optimize performance. Therefore, integrated monitoring and management software is an emerging field of the enterprise that promises to bridge the gap between IT and facilities, ultimately aiming at facilitating the administration of Data Centers.

Indeed, in current production environments, systems are always seeking for consistency and durability, in order to provide high-quality services to the end-users. In addition, efficiently dealing with failures or system outages is of paramount importance, as downtimes are translated in huge financial and business costs. In smaller, self-contained systems it is easier to detect issues and have an overview of the platform state. Growing in scale, though, means more complicated architectural design, multiple inter-dependent components and, thus, numerous risk areas that might cause faults or slowdown. Therefore, it is vital to deal with inconsistencies efficiently, in the sense of quickly isolating the root cause of the failure and actively restoring the system back to a functional state as soon as possible. This can lead to cost-effective architectures that increase the operational efficiency of clusters and extend the lifetime of their components. Picking the right infrastructure for the right purpose certainly leads to higher ROI [8] values in the long term.

Monitoring services enable sophisticated infrastructure analytics, by simplifying operational processes and delivering meaningful information about the building blocks of big clusters. By summarizing performance data, they intend to maintain system

[8]Return On Investment $= \frac{gain-cost}{cost}$

availability, distribute workloads and minimize latencies. At the same time, monitoring platforms contribute to decision making regarding IT resources, since the collected data and statistics can be used to design predictive models about infrastructure(e.g. optimal workload distribution, scheduling of future hardware upgrades, resource sharing policies etc). When instrumentation points are carefully chosen, administrators can observe the high-level state of the system, along with its delivery to end-users. A good monitoring system should always address the following questions: “which part failed?” and “what caused that failure?” In this direction, it is much easier to effectively isolate and troubleshoot any existing issues related to infrastructure. It is notable that monitoring is applicable both in physical and virtual resources across different vendors.

However, intense monitoring systems or instrumentation platforms might prove counterproductive if they are not properly implemented or configured. In such cases, it is common that large amounts of storage are required, together with high resource utilization or interference in normal operations. It is a responsibility of the administrators to fine tune the monitoring system so that it serves its intended purpose. For example, in remote monitoring approaches, there is an additional network load to track and record metrics. Extra layers of interaction add more complexity to the existent system.

2.2.2 Features of Monitoring Solutions

The majority of monitoring platforms and software shares a set of common features. Some of the main ones are listed below:

Real-time Tracking Monitoring tools provide system administrators with a concise, yet holistic overview of how well a system is currently performing, by manipulating real-time performance metrics and statistical data. Efficiency indicators of individual assets can be regarded as the “heartbeat” of the target system, outlining its overall status and health. Collection of such metrics, along with proper aggregation and refinement, can indicate service availability, functionality of equipment and identification of possible bottlenecks. Moreover, monitoring checks and policies can be adequately configured to target specific sub-systems, in order to accumulate component- or service-specific measurements. Observation of the underlying infrastructure in real-time, allows administrators to correlate running services and server workload with system profiling results, ultimately helping them discover allocation patterns and perform trend analysis on-the-fly. Besides, effective change management and awareness of current configuration contribute to rapid scaling and more intelligent deployments.

Energy & Power Management By installing energy sensors and the respective supporting hardware on the power infrastructure of a data center, monitoring software can accurately calculate power and performance measurements, such as PUE [9] and DCIE [10]. Data Centers operating at 1.5 PUE or lower are considered efficient. What's more, these measurements can be combined with fluid dynamic analysis in order to define models that allow prediction of energy consumption when infrastructure changes or gets updated. For example, this can lead to optimized air flow, cooling system and equipment placement inside the building. Rapid and unanticipated turnarounds in heat densities may impose additional burdens on the physical infrastructure, resulting in performance deterioration and increased risks of overloading and outages.

Remote Access Administration As ROBO[11] and edge data centers are constantly gaining popularity, many administrators rely on remote server monitoring. This means accessing the data center assets that are connected to the network via RESTful APIs and performing custom diagnostic checks locally. In case an issue occurs and the system is unable to automatically resolve it, human intervention might be needed to investigate the situation and tackle the problem. With remote administration, it is possible that no physical presence is needed inside the data center to manage infrastructure.

Visualization & Alerting It is quite common for monitoring software to provide an administration panel or dashboard, that aggregates and analyzes performance data and statistics. Such interfaces usually employ visualization tools that, given a collection of time-series performance data, produce comprehensive and interactive graphs. Graphical representation of asset performance can accurately demonstrate the current status of a complex system in a rather intuitive way. In this respect, administrators are capable of timely detecting issues or gaining insight into the underlying system. Thus,

[9]Power Usage Effectiveness
[10]Data Center Infrastructure Efficiency
[11]Remote Office Branch Office

this can lead to enhanced control of resources and qualitative decision making. In addition, dashboards can be configured to deliver real-time notifications intended for humans, which are usually pushed to the system in the form of a ticket queue. These alerts are raised when pre-defined threshold values are hit, targeting specific hosts, services or even the entire cluster. Alerting can be determined by raw performance metrics, or derived statistical parameters through filtering and aggregation(e.g. ratios, averages, dispersion). Noteably, notification thresholds and frequency must be carefully specified, so that alerts are sent only when a solid indication of real issues exist. Thereby, proactive monitoring alerts provide a better understanding of the internal working of the monitored system and are considered essential for resolving issues or system failures before they impact end-users.

Reporting & Logs System monitoring platforms are capable of collecting and reading logs from multiple machines and keeping track of server activity throughout time. This enables administrators to perform complex queries or aggregation functions upon log data, isolating specific time windows or services. Also, record keeping of important events (e.g. a server failure, a spike in bandwidth utilization, etc.) can be combined with running services or user requests, leading to ad-hoc conclusions about the system.

2.2.3 Monitoring and DCIM

According to the definition given in Wikipedia [14], "Data Center Infrastructure Management (DCIM) represents any set of tools(including software programs as well as hardware devices in the form of computer parts) that help organize and manage the information stored in a Data Center. The energy required to organize and store large amounts of data can be used with greater efficiency if the infrastructure of that data is carefully and appropriately managed. Thus, DCIM represents a class of IT products and services designed to assist the growing global demand for storage of digital data and information".

In our days, DCIM solutions involve both software- and hardware-based approaches. The former focus on devising efficient storage algorithms and implementing scale-out platforms to manage large sets of unstructured data. The latter concentrate on enhancing the organization and capabilities of the physical infrastructure enclosed in a

Data Center, such as extension of the capacity, prevention of overheating, network upgrades and others. The DCIM marketplace is rapidly evolving since its identification as a missing component of the IT stack. Currently, there are multiple proprietary and open-source DCIM solutions available, aiming at optimizing performance and reliability of Data Center deployments. Integrating DCIM software into the lifecycle management of data centers offers the following benefits:

- Data safety, accuracy and consistency
- Accurate detection and visualization of hardware and software assets
- Automation of control and asset provisioning
- Complete control over running services and respective configurations
- Observation of power availability and reduction of energy consumption
- Deep understanding of inner system structure and component interrelations
- Supervision and orchestration of operations inside the data center, aligning IT to the needs of the enterprise

The latest tendency in the IT community and the leading storage vendors recommends SaaS [12] versions of DCIM software, that is delivered to end-users via cloud and offers all core monitoring services. It is obvious, that DCIM is intimately related with monitoring and visualization platforms, that provide system administrators with a deeper insight of the entire infrastructure, such as racks, servers, switches, storage devices, power generators and cooling systems. Therefore, DCIM deployments might require setting up specialized software, hardware and sensors. Monitoring individual assets inside a data center focuses on improving the conditions under which the physical infrastructure operates, contributing to higher reliability and efficiency. Data and statistics coming from the real-time monitoring process can be used for intelligent decision making, capacity planning and business analytics. DCIM Suites are also capable of reducing costs associated with mismatched supply/demand and, consequently, improve operational efficiency by accurately supporting remediation.

Fundamentally, DCIM is about adding interactivity and visibility to the physical world that traditional IT runs upon. Based on the analysis stated in [13], we summarize the most critical capabilities of a well-rounded DCIM tool below:

[12]Software as a Service

- Power & Environmental Monitoring
- Reporting & Visualization
- Resource & Workflow Management
- IT Asset Monitoring & Management
- Predictive Analysis, Modeling & Simulation
- Integration with Virtualization Platforms - virtualization platforms are widely used in IT production evnironments today, achieving high performance at dramatically lower costs. Server and network virtualization is a software-based solution to extend the capabilities and capacity of IT applications without making additional investments in physical equipment. Virtualized operating systems and applications are encapsulated in individual, isolated software containers upon which administrators retain full control. Virtualization guarantees rapid provisioning cycles and application co-ordination. Hence, modern DCIM should support such platforms, interact with their services and APIs, as well as include corresponding Virtual Machine operations(create, remove, reboot, shutdown) in their management dashboards to enhance user control.

2.2.4 Monitoring Perspectives

The most basic categorization of monitoring solutions regards the way they are deployed in the system that is abou to be monitored:

Agent-based vs Agentless

In the field of IT management, there has always been a debate between agent-based and agent-less monitoring approaches. A *monitoring agent* is a service or daemon, that runs on a specific node of the system, performs script checks and ultimately collects and reports performance data, statistics, event logs, metadata and trace information. Undoubtedly, at some level or depth, every monitoring platform assigns the provisioning of assets and the aggregation of respective results to some software entity. It is noticeable, that several IT enterprises support a combination of both approaches to observe underlying infrastructure in a more flexible manner. Manually configuring how infrastructure tiers are monitored leads to a fully customized deployment that serves all specified user needs. Of course, each of the aforementioned approaches has its benefits and shortcomings, so it is important to understand their differences in design and logic:

- **Ease of Deployment & Maintenance**
  In agent-based approaches, a monitoring agent has to be deployed on each remote server of the cluster to collect metrics locally. In large scale computing environments with thousands of nodes, this process can prove very time- and effort-consuming. Evaluating or upgrading such system also requires hard effort by the administrators, since agent functionality on every node of the cluster must be verified. On the contrary, agent-less approaches are generally easier to deploy, as only a central authority is responsible for orchestrating the monitoring process, initiating all actions inside the system, so no deployment of additional agents is required. As a result, monitoring in an agent-less manner is, generally, more cost-effective.

- **Network Overhead**
  Agent-oriented technologies are very bandwidth-efficient, since performance data is collected and processed in the same host that the agent is running. The only interaction of the agent with the "outer world" is with regard to the transmission of the performance measurements to the presentation layer. Conversely, agent-less solutions impose additional network load to the targeted system, as raw performance data must be propagated to the remote collector agent. Connections are established at a rate of the most frequently collected data, while in most cases there is a constant polling process going on.

- **Security**
  Agent-based monitoring provides a high level of security as far as the communication policy and the transfer of data are concerned. The reason for this is that agent communicates with the operating system or user applications internally, so no additional network configuration to reach the Internet is needed(e.g. Proxy Servers, Firewall rules, Network Adress Translation etc). Contrariwise, in single agent deployments, the remote data collector must be given domain administration privileges in order to be allowed to connect to the target servers, communicate on specific network ports and successfully retrieve real-time metrics(e.g. via `SSH,SNMP` [13]).

- **Availability**

[13]Simple Network Management Protocol

Monitoring using multiple agents offers high availability since in the case of a network failure no performance data is lost, nor the monitoring stales. This could only happen if the specific agent of a node encounters some issue. On the other side, a network failure could bring down a whole agent-less monitoring system, as there is no way to transmit the collected data without network connectivity. Of course, it is also possible that the central monitoring agent might fail, due to heavy workloads and escalation of traffic.

- **Level of Monitoring**
  Agent-based monitoring solutions provide a deeper and broader overview of subsystems, compared to agent-less ones, since the latter are limited by application- or system-related monitoring capabilities.

In short, agent-less monitoring has reduced deployment times, consumes fewer resources and is designed for centralized environments that are capable of supplying monitoring platforms with large amounts of network bandwidth. In this case, all monitored assets are connected to that network. On the other side, agent-based monitoring is based on pull technology, is less dependent on network connectivity and offers monitoring capabilities and administrative control. At the same time, it guarantees that each sub-system operates separately as an independent component.

Monitoring software can also be categorized based on content and objective. There are two main different aspects regarding these parameters:

- **System Performance Perspective:**
  In this approach, systems might be experiencing issues regarding performance and need better utilization of the available resources. Services might be failing often or the overall system might not be running at full potential. Performance monitoring involves:

  1. Performing monitoring and profiling checks in order to identify the nature and scope of the component that is under-performing
  2. Collection of data, systematic analysis and course of actions(e.g. configuring, tuning, etc) that aim at resolving the issue
  3. Re-apply monitoring checks to verify that the degradation issue is resolved

  Though short in duration, the above process is often iterative, as these steps might be executed multiple times until the system reaches the desired level of performance

- as system resources interact with each other, their performance and utilization are highly interrelated. Thus, the elimination of one resource bottleneck might lead to the detection of another one, that was not yet noticed(chaining).

- **Capacity Planning Perspective:**
  In this approach, system performance and efficiency are considered satisfactory and require preservation at their current levels. That's why the duration of the monitoring process is longer, aiming at determining the variance in the utilization of system resources. Knowing the frequency and rates of change, administrators are capable of devising long-term plans regarding resource management. The key difference from performance monitoring lies in the facts that monitoring is constantly performed and the level of detail is considerably lower. Collecting data and measuring performance regularly over a longer period of time leads to a broader view of the infrastructure from which one can extract useful information about the inner attribute of a large scale cluster. Aggregation and correlation of performance data coming from multiple sources promise to facilitate automated management of IT applications. Consequently, administrators can accurately divide resources in categories, based on the mean-variance of their workload, and determine the impact of running services upon system resources.

Finally, it is useful to clarify two basic monitoring approaches in terms of the policy of the monitoring process:

- **Pull-based Monitoring**
  In this approach, multiple monitoring agents are installed on the assets being monitored, probing them at a regular interval. Their goal is to run pre-configured checks to measure performance, generate reports based on these metrics and produce corresponding real-time alerts or notifications. External services or systems can collect telemetry data from the agent. Hence, monitoring is essentially performed by the agent itself, while communication is facilitated by client libraries. The main advantage of pull-monitoring is that it provides a standard mechanism for services to expose their targets and a well-known format to pull data from. Nevertheless, pull based methods are not so performant on event-driven environments, like individual API requests or unexpected behavior of the infrastructure.

- **Push-based Monitoring**

The push-based method allows for "reverse" monitoring, in the sense that the data collected by the agent installed on the targets, is pushed to a publicly available storage repository, commonly a time series database that supports access via some RESTful `HTTP` API. Pushing data into a time series database usually requires an extra step of formatting the metrics into the line protocol of the database, before real-time metrics are flushed. Subsequently, monitoring is essentially performed by an external service that does not belong to the target system. Such service queries the database, retrieves time series data and applies further processing upon them.

Depending on the scope of each application, either the push- or pull-based monitoring approach might be more suitable. However, there are also cases that require the combination of both in a hybrid deployment, in order to bring the best out of both worlds and optimize the administration process.

2.2.5 Icinga Distributed Monitoring

About the Icinga Project

Icinga [18] is a distributed, scalable and extensible monitoring system that checks the availability of resources, notifies end-users about current issues and provides extensive business intelligence data. It is cross-platform and highly customizable, since it executes user-defined checks in the form of scripts, aiming at automation of operations inside the targeted system. Icinga is an original fork of the Nagios [19]. With an updated core, compared to Nagios, it offers great flexibility by providing additional features and integrations with modern technologies. It relies on a highly modular architecture that combines core capabilities with add-ons and plugins, as necessary.
Icinga is an entirely open-source monitoring solution, which is actively developed and supported by a large community. Due to its multithreaded design, it performs exceptionally well in large scale systems, efficiently monitoring thousands of nodes at the same time. We have studied and deployed the latest stable release, which is 2.6.

Why Icinga?

- **Distributed environment:** Icinga is designed to perform checks in a distributed fashion, issuing checks across the entire platform. The nodes of an Icinga cluster are capable of monitoring themselves and collecting performance data. Icinga instances autonomously replicate configuration and program states in real-time, ensuring data integrity. Besides, Icinga nodes can be configured in groups for high availability, in case of node failure. Hence, the monitoring process is not dependent on a single central monitoring server, though metrics are ultimately propagated to the root of the hierarchy tree for further analysis. Distributed monitoring guarantees higher stability and redundancy; a failing component can be replaced by one of its peers, without disrupting the normal operation of entire monitoring system.

- **Object-based configuration:** deploying and configuring Icinga requires the definition of macros, commands, services, notifications, hosts and other monitoring concepts. All these entities are described, handled and understood in the form of objects by Icinga. Users can extend the default templates by defining custom properties or adding conditional behaviors in order to meet their needs. This object modeling makes the configuration process more comprehensible and intuitive, separating concerns and responsibilities.

- **Compatibility with monitoring plugins:** Icinga maintains backward compatibility with all existing Nagios plugins and configurations. Thereby, a numerous predefined services and checks are available online for free use.

- **Efficient integrations:** Icinga is a monitoring platform that offers compatibility with various technologies that are closely related to effectively storing performance data and status information. More specifically, it provides connectors to PostgreSQL, MySQL and Oracle databases. To achieve that, it introduces native Icinga Data Output(IDO) modules that are responsible for formatting and writing time- series metrics to various storage backends. Icinga supports platforms that efficiently handle time series, like Graphite, InfluxDB and TSDB and others that handle logs, like Elastic and Gelf.

- **Web Interface & HTTP API:** the web administration panel of Icinga is a lightweight, PHP-based, standalone software component, that shows the current status of hosts and services inside the Icinga cluster. It can be configured to send alert notifications to the user when performance checks return *WARNING* or *CRITICAL* status. It is easily extendable and can support multiple backgrounds simultaneously. What's more, Icinga can communicate with external services via a RESTful HTTP API, so users can define configuration objects, check asset availability or monitoring data via `GET` or `POST` calls. Results can be returned in `XML` or `JSON` format, while the Icinga API supports filtering(choosing columns to be returned), aggregation and count operations. User authorization can be performed either using cookies or simply as an in-request parameter.

- **Security & Encryption:** Monitoring servers are always given a certain level of trust in order to query remote systems. The communication channels of the Icinga API are secured by SSL X.509 certificates and public key cryptography. Furthermore, they can be initiated by either endpoint and work behind a NAT boundary.
- **Automated Deployments:** Icinga supports Chef, Ansible, Saltstack and Puppet platforms for automated management and setups of monitoring systems
- **Powerful CLI:** administrators can greatly benefit from the command line tools that Icinga offers, aiming at validating configurations and automating the deployment process

Icinga Cluster Architecture

From a higher-level perspective, Icinga consists of 3 fundamental components:

- **Icinga Core**, which manages monitoring tasks, receives check results and statistics from plugins
- **IDODB**, which effectively stores performance data in specific formats
- **Icinga Web**, which provides a real-time administration dashboard

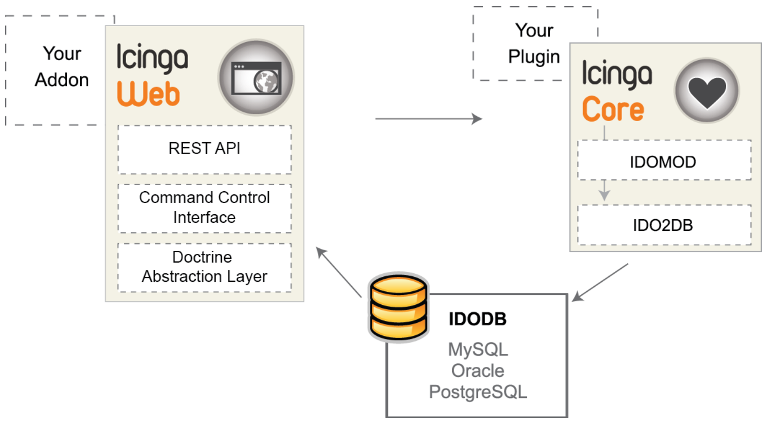


Figure 2.1: *Icinga Architectural Components*

Configuration Design

In order to formalize the structure of the cluster and perform the monitoring checks, Icinga introduces certain entities that are represented as objects in the configuration files and interfaces. `CheckCommands`, `Services`, `Hosts` and `Templates` are the most important ones and can describe almost anything inside a cluster, varying from network services like `DHCP`[14] `,SMTP,SSH` etc. to switches and server nodes.

CheckCommands

`CheckCommand` objects are used to describe monitoring checks performed by Icinga. Their definition always contains a `command` attribute, which holds a full path to the check plugin to be executed. Optionally, the `arguments` attribute is determined, holding any command line arguments needed for the proper execution of the check script. `CheckCommand` objects are usually referenced by `Host` or `Service` objects, while parameters can be passed to them on-the-fly. Accessing attributes of other objects requires `runtime macros`, that dynamically retrieve current values.

Services

`Service` objects must always contain the `check_command` attribute, that specifies the filesystem path where the command to be executed by the service is saved, as well as the `host_name` attribute, which holds the host object associated with the service. Services can be in any of the following states: *OK*, *WARNING*, *CRITICAL* and *UKNOWN*, depending on the output value of the corresponding command.

Hosts

`Host` objects may contain already defined services or commands to be executed directly. Since host objects usually represent server nodes, it is quite common to contain an `address` field that holds the associated network address. Hosts can either be in an *UP* or *DOWN* state.

Templates

Built-in `Template` objects are provided by Icinga for all object types. They introduce various default attributes, along with predefined values that users are likely to find useful. Objects, as well as templates, can import an arbitrary number of templates. If necessary, attributes inherited from a template can be overridden inside

[14]Dynamic Host Configuration Protocol

the object definition. In addition to built-in attributes, users can define custom ones under the `vars` attribute, which is actually a dictionary. Valid value types include `string,number,boolean,array` and `function`. Loop iteration over arrays or dictionaries and “if-then-else” conditions are also supported. Icinga templates and objects share a common namespace, so no duplicate names are accepted.

Rules

`Rules` automate and extend the definition of configuration objects since they describe logical expressions and patterns that determine the application of services and commands. Decision making can rely on attribute values, existence of objects, comparisons with constants, etc. It is usual, that several objects need to be correlated with other objects and external attributes. Manually managing large sets of configuration objects that match on a common pattern, is both time-consuming and error-prone.

The official Icinga documentation provides an extensive description of all Icinga objects. For the scope of this thesis, it is sufficient to present the definition of the most important ones:

```
# Definition of Host Object
object Host "server1" {
   import "generic-host"
   address = "10.0.0.1"
   check_command = "hostalive"
}
# Definition of Service Object
object Service "ping4" {
   import "generic-service"
   check_command = "ping4"
   # Rule applying to all hosts with 'address' attribute defined
   assign where host.address && host.vars.os == "Linux"
}
# Definition of CheckCommand Object,
object CheckCommand "ping" {
   command = [ PluginDir + "/check_ping", "-H", "$ping_address$" ]
   arguments = {
      "-w" = "100,5%"
```

```
            "-c" = "250,10%"
            "-p" = "5"
    }
    # runtime macro refering to host <address> attribute
    vars.ping_address = "$address$"
}

# Definition of Template Service Object
template Service "generic-service" {
    max_check_attempts = 3
    check_interval = 5m
    retry_interval = 1m
    enable_perfdata = true
}

# Definition of Template Service Object
template Service "generic-host" {
    max_check_attempts = 5
    check_interval = 1m
    retry_interval = 30ms
    check_command = "hostalive"
}
```

Listing 2.1: *Definition of Icinga Objects*

Distributed Design

The Icinga cluster is a distributed monitoring environment that supports load balancing and high availability among cluster nodes. Icinga defines certain roles for the nodes that belong to the cluster, in order to establish a hierarchical topology and assign duties. This design highly adaptable to any scale(size, complexity), simplifies node configuration and provides great flexibility to administrators. In **??** the reader can observe the tree structure of the Icinga cluster, consisting of 3 types of nodes-master, satellite and client- which are bidirectionally connected:

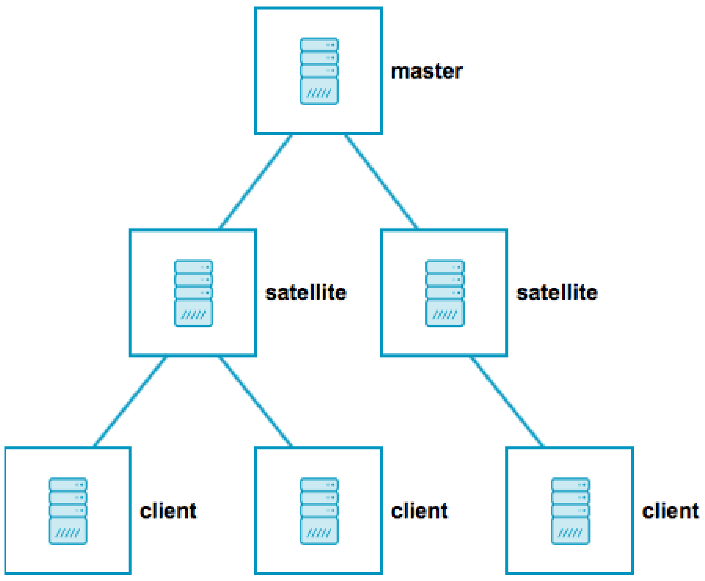


Figure 2.2: *Distributed Setup of Icinga Cluster*

Roles

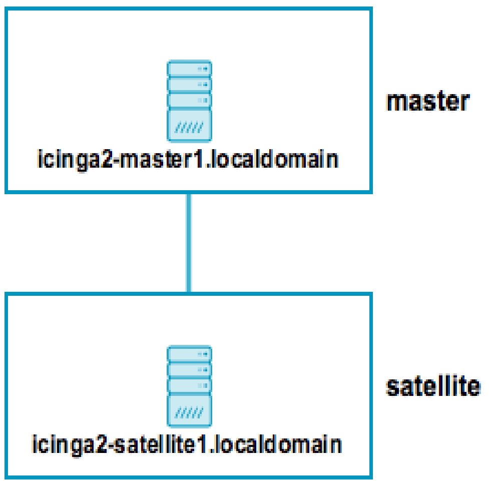
- **Master:**
  - has no parent, only child nodes(root)
  - aggregates performance metrics from clients and commits them into the storage backend
  - serves Icinga Web administration dashboard
- **Satellite:**
  - has a parent and a child node
  - executes checks on its own or delegates check execution to child nodes
  - receives configuration files for `Hosts, Services,` etc. from the master node
  - runs independently from the master node, while its presence is not mandatory inside the cluster
- **Client:**
  - only has a parent node - leaf of the tree
  - receives command execution events from its parent node and executes the specified checks
  - sends check execution results and performance data to the parent node

Zones & Endpoints

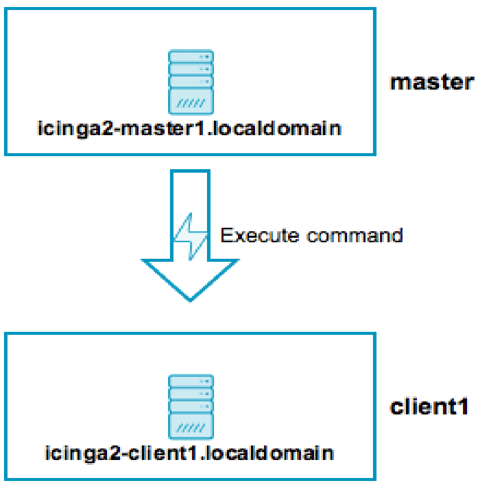
For more convenient organization of nodes that share the same role, Icinga introduces the concept of **`Zones`**, creating an abstraction grouping layer. In this way Icinga organizes the cluster in a hierarchical structure, by defining `Zone` objects that depend on “parent-child” relationships. Child `Zones` are not allowed to push, but only receive configurations and updates from their parents, by setting the `accept_config=true` option in the respective definition. This option instantiates the Icinga API Listener object. In order to preserve consistency in operations, a `Zone` should have no interferance with other zones, while every defined host or service object can belong to one zone only.The members of a `Zone` are called **`Endpoints`** and are also configured as objects. A `Zone` usually consists of multiple `Endpoints` that co-operate to offer specific services. All members of the same zone trust each other and can be configured to share the total workload (distributed monitoring) or replace each other in case of failure(high availability). Besides, all communications between `Endpoints` are secured with SSL certificates to prevent system vulnerabilities.

```
# Definition of Endpoint object
object Endpoint "icinga2-master1" {
   host = "192.168.56.101"
}
# Definition of Endpoint object
object Endpoint "icinga2-client1" {
   host = "192.168.56.102"
}
# Definition of Zone object
object Zone "master-zone1" {
   endpoints = [ "icinga2-master1" ]
}
# Definition of Zone object
object Zone "client-zone1" {
   endpoints = [ "icinga2-client1" ]
   parent = "master-zone1"
}
```

Listing 2.2: *Definition of Icinga Zones and Endpoints*



(**a**) *master + satellite*



(**b**) *master + client*

Figure 2.3: *Icinga Zone Hierarchy*

Configuration Modes

Icinga supports two different modes to synchronize configurations, depending on the direction that configuration files, checks, updates and notifications flow: **Top →Down** and **Bottom →Up**. We focus and analyze only the **Top →Down** configuration option, since it is the recommended and most popular approach across most Icinga deployments. This mode is suitable for topologies that need all the configuration files to be stored in the master `Zones` and get pushed or synced into satellite or client `Zones`. The definition of the configuration objects takes place only once inside the master node under the `/etc/icinga2/zones.d` directory, while each `Zone` is individually configured regarding hosts, services and checks. Then, configuration files are pushed down to satellites or clients that run their own local scheduler and send check results back to the master. No manual restart of the Icinga daemon is required on the child nodes since validation and syncing are done automatically. `Endpoints` that belong to the same `Zone` receive the exact same configuration from the corresponding master `Zone`. Thus, monitoring large scale clusters that consist of thousands of nodes can be greatly simplified and accelerated. For additional flexibility, Icinga usually introduces a **`Global Zone`** object, that is used for sharing Icinga objects across separate `Zones` of a single Icinga cluster. In this manner, individual or unrelated `Zones` share a common set of Icinga configuration objects, like `Commands,Templates,Users` and `Groups`, to perform monitoring.

2.3 Distributed Data Storage

2.3.1 The current picture

It is beyond doubt that, in the time of Big Data, digital technologies have prevailed in almost every aspect of human life. The massive amounts of unstructured data that are globally produced on a daily basis, raise the demands for massive capture, storage, analysis and curation. Consequently, both the research community and the enterprise, concentrate their efforts on the design of cost efficient storage systems that are capable of facing this challenge, as storage is at the core of every IT-related service. Even though centralized storage solutions are barely 20-25 years old, they are no longer able to keep up with the exponential data growth.

Traditional storage solutions involve disk arrays, multiple controllers and a lot of disk trays attached to a SAN [15] in order to provide data storage services to clients. The disk trays are connected to the storage controllers and clients are able to perform data operations through those controllers. Scaling up the capacity and performance of such systems requires the constant provisioning of new hardware, together with proper configuration. As expected, this approach suffers from throughput, capacity and bandwidth bottlenecks, as controllers get overloaded over time. The typical solution is to buy additional controllers and disk arrays, moving some load to the new hardware. Of course, this solution is expensive and definately not flexible, as it is limited to the terabyte scale and does not fit the current demands of the industry for elasticity and high availability in the petascale. Therefore, it is argued that we have reached the stage at which conventional ways of storing data using stand-alone network attached devices have been declared dead, as they no longer serve their intended purpose.

Despite the fact that commodity storage hardware is rapidly improving, future-proof storage solutions do not lie in faster drives or higher network bandwidth. Instead, a new concept of storing data is introduced suggesting an alternative approach, referred to as *distributed data stores*. This concept proposes a software-defined storage(SDS) solution, in which all storage-related tasks are handled programmatically by software entities which are entirely decoupled from the underlying physical hardware. Software-defined architectures reject centralized, networked storage resources and es-

[15]Storage Area Network

tablish a clustered, dynamic, server-based system that relies on sharing each server's local storage resources. Such storage solutions employ powerful computational algorithms, that distribute workloads across available resources and guarantee data consistency and redundancy. Currently, distributed data storage platforms are the most promising approach to satisfying the ever-growing demands for the efficient storage of data in large scale.

2.3.2 Distributed Storage Definition

According to Wikipedia, "A distributed data store is a computer network where information is stored on more than one nodes, often in a replicated fashion. It usually refers to either a distributed database where users store information on a number of nodes, or a computer network in which users store information on a number of peer network nodes".

Distributed databases are mostly non-relational databases that support quick access of data over a large number of nodes. Some distributed databases provide rich query capabilities to the end-users, while others are rather limited due to a key-value store semantics. Meanwhile, in peer network data stores, the user can usually reciprocate and allow other users to use their computer as a storage node as well. Information may or may not be accessible to other users depending on the design of the network. A distributed storage system can relate to any of the 3 types of storage: block, file or object. The goal is to achieve high reliability and ubiquitous availability of data. Moving to a software-defined model ensures a smooth future transition to hybrid and multi-cloud operations. In order to gain deeper insight into distributed data stores, it is useful to observe their core characteristics, along with their main differences compared to traditional storage solutions:

- **Philosophy**
  The main difference between traditional storage techniques(like SAN or NAS) and distributed data storage ones(like HDFS, GFS, Ceph) is that the former mostly depend on the underlying hardware(hardware defined), while the latter do not require specific hardware support(software defined). This means that SAN & NAS storage solutions propose a hardware-specific, limited, proprietary deployment, whereas a software defined storage cluster can be deployed using non-specialized commodity

hardware(servers, drivers, network etc), separating logical aspects of storage from the physical components. Software-defined storage typically includes a form of storage virtualization that separates the physical storage hardware (data plane) from the data storage management logic or intelligence(control plane).

- **Cost**
  Considering the always increasing demand for storage, the philosophy and design of distributed storage solutions make them very cost-effective with regard to the IT infrastructure needed to set up a reliable alrge-scale storage system. Software-defined data storage, being hardware-independent, has a positive impact on IT deployments, as it converges storage, increases the utilization of servers and restricts power and space consumption by applying strong computational models. As a result, it considerably reduces the cost per GB comparing to conventional storage solutions, especially when managing data in the petascale. In addition, the system administration costs are greatly reduced since considerably less infrastructure entities need to be managed.

- **Flexibility & Automation**
  Traditional SAN & NAS come with limitations in terms of scalability, as they can support a maximum number of controllers and number of drives connected to each controller. On the contrary, distributed data stores are *scale-out*, since they have theoretically unlimited capacity being expandable by design. One can keep adding new nodes to the software-defined storage cluster and the cluster will keep increasing in capacity and performance. This is also known as horizontal scaling. Besides, the management of SDS is very flexible, as it provides the capabilities to add new nodes or remove faulty ones on-the-fly, using some CLI programming tool. The fact that there is no downtime during these processes greatly facilitates the maintenance and expansion of software-defined storage clusters.

- **Reliability**
  The majority of distributed storage solutions are fault-tolerant, meaning they are likely to continue their operation even though one or more components have failed. In addition, they are capable of replicating the data stored for any unpredicted failures. Due to their distributed nature and design, data is not stored on a single node; instead, it is stored in multiple nodes across the cluster, depending on user-defined policies and replication factor. It is also quite common that they regularly perform

data sanity checks to prevent data corruption. Since data movement around the cluster is relatively expensive, some SDS technologies adopt pooling approaches that suggest leaving data in place and creating an upper mapping layer to it that spans arrays. In distributed data storage, it is the function of software-defined storage system that ensures that everything is divided, distributed, replicated and, in case of failure, rectified with intelligent algorithms.

- **Multi-tenancy**
  Software-defined storage solutions are designed to manage heavy cloud workloads and, therefore, provide multi-tenant functionality. Multi-tenancy involves delegation of management responsibilities, isolation of users and restriction of access based on security policies.

2.3.3 Ceph Distributed Data Store

As a test case scenario for OntoMon we decided to deploy a Ceph distributed data storage cluster and monitor its health. Ceph [20] is a software-defined platform that belongs to next-generation storage solutions, combining various core characteristics:

- *Software-defined*: agile deployment, faster hardware upgrades and lower costs
- *Enterprise-features*: replication, snapshots, thin provisioning, auto-tiering and self-healing guarantee efficiency in operations, protection and consistency of data
- *Unified storage*: combines block, file and object interfaces offering great versatility, as most other storage solutions are block-, file-, or object-only interfaces
- *Scale-out*: incrementally expand the capacity of the cluster, based on current storage needs, simply by adding commodity nodes
- *Open-source*: collaboration, customizability, quality, interoperability and security

What is Ceph?

Ceph is an open-source software-defined storage platform, that implements object storage on a single distributed computer cluster. It offers a truly unified, highly scalable and reliable storage solution by providing interfaces for object-, block- and file-level storage. Ceph primarily aims for completely distributed data operations without a single point of failure, scalable to the exabyte level, and freely available. Using an algorithm called *CRUSH* [16], it effectively stores, retrieves and replicates data across various cluster nodes. Thus, it offers replica-based data protection, by retaining multiple copies of data across multiple cluster nodes in the case of failure. This makes it fault-tolerant, while using commodity hardware that require no specific support. Due to its design principles, the system is both self-healing and self-managing, aiming to minimize administration time and other costs, without any centralized bottlenecks.

The object manipulation algorithm used by Ceph liberates storage clusters from the scalability and performance limitations imposed by a centralized data table mapping. It replicates and re-balances data within the cluster dynamically, while delivering high-performance and infinite scalability. In traditional architectures, clients talk to a centralized component (e.g. a gateway, broker, API etc.), which acts as a single point of entry to a complex subsystem. This imposes serioues limitations to both performance and scalability, while introducing a single point of failure, if the centralized component goes down. Ceph claims to achieve maximum uptime, due to its de-centralized design and intelligent storage daemons. Also, Ceph does not rely on RAID technology for data protection; rather, it uses replication and erasure coding. Regarding networking, Ceph communicates over a regular `TCP/IP` network. Ceph has been rapidly evolving in the area of cloud storage and is grabbing center stage both with the major open source cloud platforms, like OpenStack, CloudStack etc. and the leading developers in both storage and Linux space, such as Canonical, Red Hat and SUSE.

Architectural Components

The architectural structure of a Ceph cluster, as well as its fundamental components, are presented in Figure 2.4 and Figure 2.5. The former reviews the design of Ceph from a high-level perspective, while the latter depicts some more technical details:

RADOS [17]

RADOS [22] is the cornerstone of a Ceph cluster and is the component that implements distributed object storage. All data issued to a Ceph cluster is, eventually, broken up into objects before it gets written. The RADOS layer is responsible for ma-

[16]Controlled Replication Under Scalable Hashing

[17]*Reliable Autonomic Distributed Object Store*

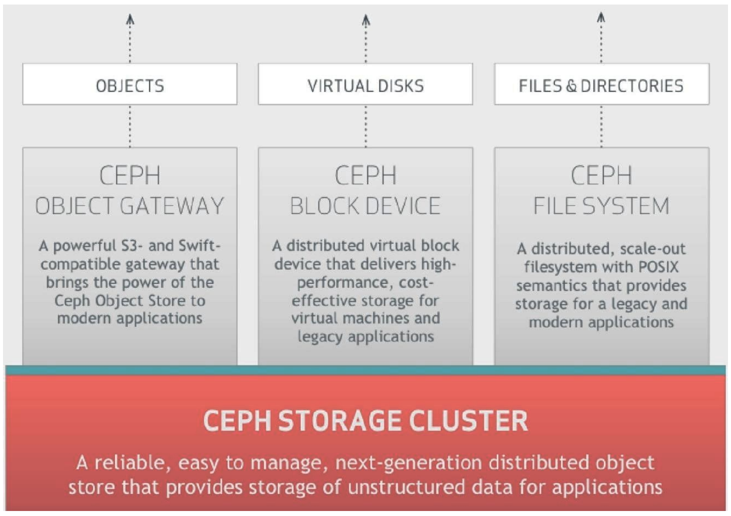

OBJECTS
VIRTUAL DISKS
FILES & DIRECTORIES
CEPH OBJECT GATEWAY
A powerful S3- and Swift-compatible gateway that brings the power of the Ceph Object Store to modern applications
CEPH BLOCK DEVICE
A distributed virtual block device that delivers high-performance, cost-effective storage for virtual machines and legacy applications
CEPH FILE SYSTEM
A distributed, scale-out filesystem with POSIX semantics that provides storage for a legacy and modern applications
CEPH STORAGE CLUSTER
A reliable, easy to manage, next-generation distributed object store that provides storage of unstructured data for applications

Figure 2.4: *High-level Ceph Cluster Architecture*

nipulating and storing these objects irrespective of their data type. In this direction, every data transaction from users is translated into an internal object operation in RADOS. RADOS makes sure that data always remains in a consistent and reliable state, by performing data replication, failure detection and recovery, as well as migration and rebalancing across cluster nodes. Ceph manages everything as an object underneath, by assigning every object a unique identifier and storing it in a flat address space. An object in RADOS contains both binary data and metadata, formalized as a set of key-value pairs. RADOS is rather flexible and generic, so it is ideal for building more complex systems on top of it. A RADOS cluster consists of multiple nodes upon which the Ceph daemons are running. The most important ones are OSDs [18] and Monitors:

- **OSDs** are running on top of a filesystem(e.g.`xfs,btrfs,ext4`), are associated with a specific path of the filesystem (e.g. `/var/lib/ceph/osd-1`) or a mounted storage device (e.g. `/dev/sda/ceph/osd-1`) mounted to the filesystem, and mainly perform read and write operations upon that storage location. Ceph OSDs manage disk block allocation internally, exposing an interface that allows others to read and write named objects of variable-size in a flat namespace(i.e. no hierarchy of directories). A Ceph cluster can support thousands of OSDs that serve objects to clients, working in a peer-to-peer fashion. There are 2 types of OSDs: *primary* and *secondary*. Each object in RADOS is assigned to a primary OSD that handles all read and write operations for the clients, while secondary OSDs are under the control of the pri-

[18]Object Storage Devices

mary. OSDs communicate and collaborate with other object storage devices, in order to perform replication and recovery tasks. More specifically, OSDs heartbeat each other, so when an OSD fails, its “peers” inform the Monitor. RADOS considers two dimensions of OSD liveness: whether the OSD is reachable, and whether it is assigned data by CRUSH. An unresponsive OSD is initially marked **down** and if it does not quickly recover, it is marked **out** of the data distribution. Then, another OSD joins to re-replicate its contents. What’s more, OSD nodes are responsible for performing the mapping of objects to blocks, a task traditionally done at the file system layer in the client. This behavior allows the local entity to best decide how to store an object. It is recommended to run one OSD per physical storage device.

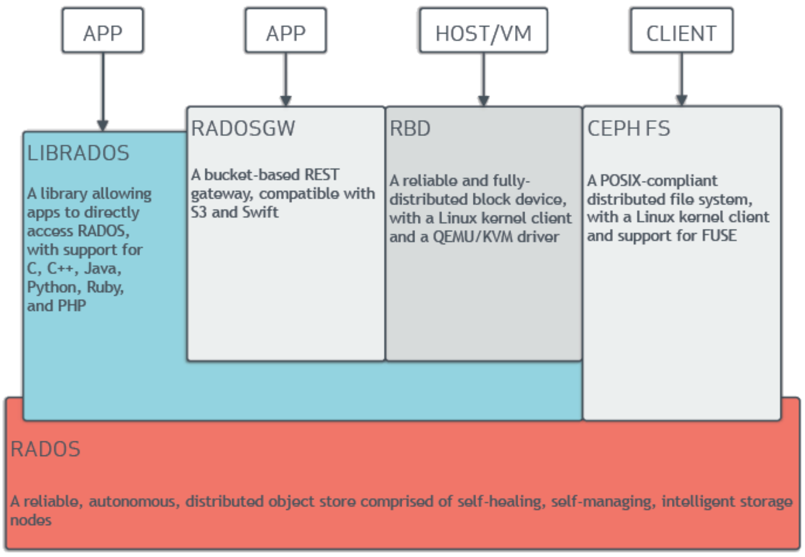

APP
APP
HOST/VM
CLIENT
LIBRADOS
A library allowing apps to directly access RADOS, with support for C, C++, Java, Python, Ruby, and PHP
RADOSGW
A bucket-based REST gateway, compatible with S3 and Swift
RBD
A reliable and fully-distributed block device, with a Linux kernel client and a QEMU/KVM driver
CEPH FS
A POSIX-compliant distributed file system, with a Linux kernel client and support for FUSE
RADOS
A reliable, autonomous, distributed object store comprised of self-healing, self-managing, intelligent storage nodes

Figure 2.5: *Technical Ceph Cluster Architecture*

- **Monitors**, which are responsible for keeping track of the health of the cluster, managing the cluster membership and orchestrating the data placement elections. Monitors do not serve objects to clients; instead, they maintain a master copy of the map of the cluster, representing its current state. Before a Ceph client reads or writes data to the cluster, a copy of the latest cluster map should be obtained from the Monitor node. Therefore, Monitors receive timely reports regarding the current state of OSDs that belong to the cluster. When object storage devices fail or new devices are added, Monitors get informed about these changes and, once agreed upon them, they issue an updated valid version of the cluster map, that records the most recent changes. Differences in map epochs are significant only when they vary between two communicating OSDs, which must agree on their proper roles with respect to the particular PG the I/O references. This property allows RADOS to distribute map updates lazily by combining them with existing inter-OSD messages, effectively shifting the distribution burden to the OSDs. Aiming at consistency and durability, active Monitors form a *quorum*, implementing PAXOS [24], a family of algorithms that provide consensus for distributed decision making. To avoid split-brain situations, a majority of monitors must always be available(i.e. odd number of quorum members) in order to agree on updates of the cluster map and guarantee that no monitor distributes the old map to the clients.

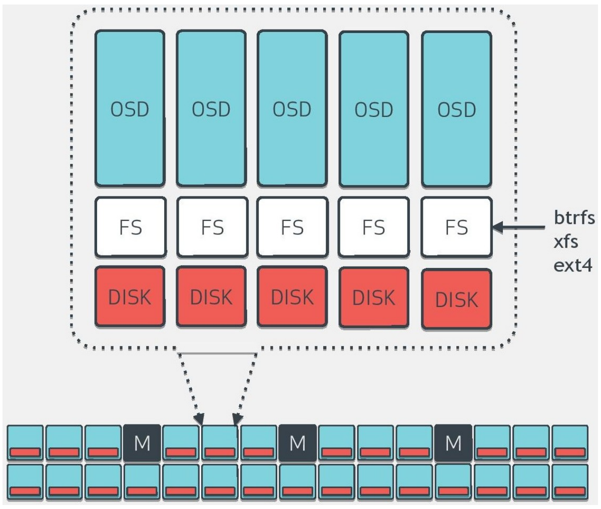


Figure 2.6: *Overview of RADOS*

LIBRADOS

As its name suggests, *LIBRADOS* is a set of software libraries, or an API, that provides a native interface for low-level access to the services of RADOS, available in `C,C++,Java,PHP` and `Python`. The communication with the storage cluster is implemented via `TCP` sockets, introducing no additional `HTTP` overhead, so it is really direct and fast. LIBRADOS provides rich functionality over objects, supporting file-like operations on them and as well as a locking primitive, called "watch-notify". It is also highly customizable, as user `C++` code can be injected to manipulate object classes.

RADOSGW

RADOS Gateway is a FastCGI service that provides a RESTful API to store objects and metadata to the RADOS cluster and can be used along with any FastCGI capable webserver. It is a thin-layer `HTTP` gateway, that sits on top of LIBRADOS with its own data formats, and maintains its own user database, authentication, and access control. The RADOS Gateway supports two interfaces:

- **S3-compatible:** provides object storage functionality with an interface that is compatible with the Amazon S3 RESTful API,
- **Swift-compatible:** provides object storage functionality with an interface that is compatible with the OpenStack Swift API

Since a unified namespace is used, web applications that are connected to RADOS Gateway can read and write data using either API. Buckets, accounts and authentication are also integrated and supported.

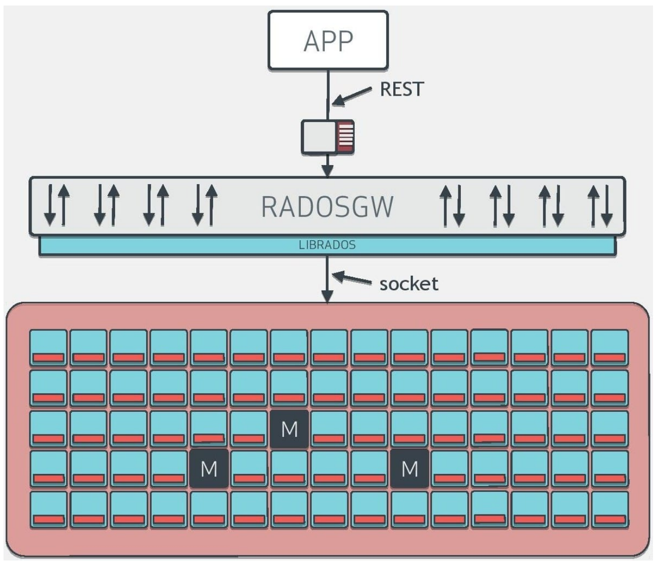


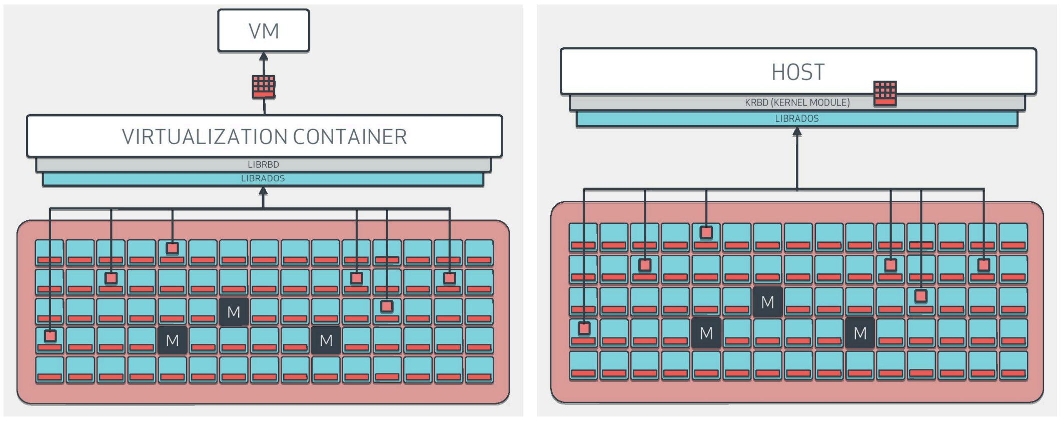
Figure 2.7: *Rados Gateway for RESTful services*

RBD[19]

RBD is the core component that provides the block storage interface of Ceph. It is a reliable and fully distributed block device that supports Linux kernel clients, along with the QEMU/KVM virtualization driver. Ceph block devices are thinly provisioned, largely resizeable and store data in multiple OSDs across the cluster. Block-based storage interfaces are the most common way to store data with rotating media such as hard disks, CDs, floppy disks etc. This means block device images are broken into smaller chunks of volumes that are stored as objects in RADOS. By striping images across the cluster, Ceph improves read access performance for large block device images and

[19]Rados Block Device

decouples VMs from the host machine. The block device can be virtualized, providing block storage to virtual machines in various virtualization platforms. Ceph Block Devices leverage RADOS capabilities such as snapshotting and copy-on-write clones, that are widely used in VM environments. For example, one can create a "golden" image inside a Ceph cluster and then use COW cloning to bring up multiple VMs using RBD-based image snapshots. As shown in Figure 2.8, there are 2 different ways of accessing the RBD interface of Ceph:



(a) *librbd* **(b)** *KRBD*

Figure 2.8: *Rados Block Device access*

1. Use **librbd**, which is a library that links the virtualization container with Ceph, providing devices as disks to Virtual Machines. Ceph Block Devices integrate well with the QEMU virtualization engine. Librbd is linked to LIBRADOS, in order to communicate with the RADOS cluster, with the virtualization container, in order to provide the virtual disk volume to the VM. The main objective of librd is to gather chunks of space from RADOS, formalize them as a single block of storage and make it available to the VM. The main advantage of this architecture is that, since the image and all data related to the virtual machine are stored in the RADOS cluster, we can easily migrate the VM by suspending it on one container and then bringing it up on a different container on-the-fly. QEMU can access an image as a virtual block device directly via librbd. This performs better because it avoids an additional context switch on the hypervisor, and benefits from RBD caching.
2. Use **KRBD**, which is a Linux kernel module [20] that makes RADOS block storage available on a Linux server or host. This module grants higher flexibility to the

[20] `http://elixir.free-electrons.com/linux/latest/source/drivers/block/rbd.c`

clients, as RADOS images can be exported as block devices to the host. The process involves loading the RBD module and then mapping a block device image name to a kernel module, by specifying some related info.

CephFS

CephFS is a distributed, POSIX-compliant filesystem that runs on top of the same distributed cluster that provides object and block storage interfaces. CephFS allows data to be stored in files and directories and provides a traditional file system interface with known POSIX semantics. The Ceph Filesystem requires at least one Ceph Metadata Server daemon(MDS) to be running on some node of the RADOS Cluster.In the backend, the MDS handles all data related to the filesystem as objects and saves them in RADOS. Interacting with CephFS requires communication with the MDS involving permissions, ownerships, timestamps, along with the specification of file and directory hierarchies. Once the needed semantics are provided to, data is retrieved and delivered to the client by the OSD. The Metadata Server does not handle any data, it just stores and manages the POSIX semantics. CephFS is horizontally scalable, since when the number of metadata server increases, the workload is split among themselves providing a high-availability setup. Another feature that the Ceph FS introduces is file sharing, as multiple clients can work on the same file simultaneously.

Data Placement

One of the most intriguing and powerful features in Ceph lies in the way it faces the challenge of efficiently managing data at large scale. Essentially, this refers to the mechanism that is responsible for computing the placement and storage of a growing amounts of data across the cluster. Ceph avoids lookup of data locations in some central directory, in order to facilitate addition or removal of nodes. Thus, it aims at moving as few objects as possible, while still maintaining balance across new cluster configurations. Ceph focuses on reducing periods of high IO, as it is designed to balance IO traffic, by distributing the workload to sufficiently many nodes. To achieve this, Ceph implements the *CRUSH* algorithm to store and retrieve data, by delegating the computation of the explicit storage location of each object inside the cluster to the client. In order to explore CRUSH, it is required to analyze the concepts of *pools, placement groups* and *OSDs*, as presented in the official Ceph docs [25]. Without loss of generality, the problem of data placement can be reformulated into an "object-to-osd mapping" problem.

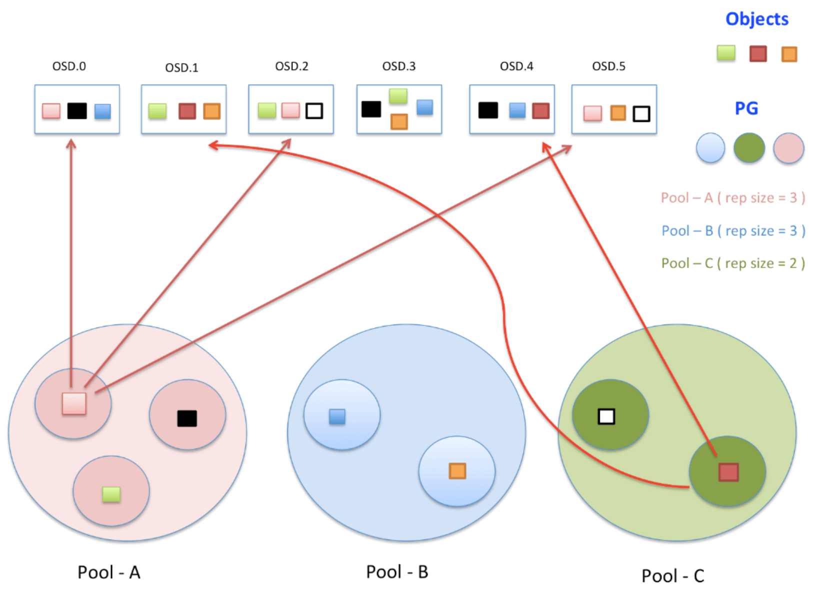

Objects
PG
OSD.0
OSD.1
OSD.2
OSD.3
OSD.4
OSD.5
Pool – A (rep size = 3)
Pool – B (rep size = 3)
Pool – C (rep size = 2)
Pool - A
Pool - B
Pool - C

Figure 2.9: *Overview of Ceph Data Placement*

Pools

Pools are a logical group for storing objects inside Ceph. They provide the highest level of abstraction that partitions Ceph global storage. Hence, this abstraction can be used to define the configuration of storage policies, such as the level of resilience(replication factor), the number of placement groups, the CRUSH ruleset, ownerships, etc. In Ceph, each object will be stored in a concrete pool, so the pool identifier(a number) and the name of the object(a string) are used to uniquely identify the object in the system. When a pool is created, it is assigned a number of placement groups (PGs). Storing data in a pool requires user authentication with permissions for the specific pool. Ceph is also capable of taking snapshots of pools.

Placement Groups

Placement groups(PGs) define the second level of storage abstraction inside a Ceph cluster, and actually are shards or fragments of a logical object pool that places objects as a group into OSDs. Though invisible to clients, PGs are very important entities inside a Ceph cluster, as they dynamically map objects to OSDs and are, thus, regarded as the distribution unit in Ceph. The objective of placement groups is to aggregate objects within a pool, since tracking per-object placement and metadata adds disproportionate computational workloads. In addition, aiming at higher reliability, Ceph spreads

PGs across multiple OSDs inside the cluster. It is notable, that every single Placement Group is guaranteed to always be consistent and, therefore, could be regarded as a sub-cluster. By convention, the first OSD mapped to a PG is the primary OSD, while the rest ODSs mapped to the same PG are secondaries. Every object inside a Ceph cluster is mapped to exactly one PG. In general, more PGs improve performance and lead to a well-balanced storage system. However, increasing the number of OSDs, requires careful specification of `pg_num` value, since it significantly influences both the behavior of the cluster and the durability of stored data. A recommended general rule to evenly distribute data across the Ceph cluster is given below:

$$TotalPGs = \frac{OSDs * 100}{ReplicationFactor}$$

Figure 2.10: *Choosing the right number of PGs*

CRUSH Map

CRUSH empowers Ceph clients to communicate with OSDs directly, rather than through a centralized server by performing client-side calculations. Hence, clients need to provide an overview of the current cluster state to CRUSH algorithm in order to perform any data operation inside the Ceph cluster. The CRUSH map aggregates OSDs into physical locations and contains a list of rules that tell the CRUSH algorithm how it should replicate data in the pools of the Ceph cluster. This is actually a hierarchy of nodes(also referred to as "buckets") and leaves, that are defined by their type. Hierarchies are, in general, arbitrary in order to serve each user needs. However, the leaf nodes always represent OSDs and each leaf belongs to a node or bucket.

The CRUSH rules allow modeling of the actual physical topology of the Ceph deployment, in terms of data centers, racks, shelves, hosts etc. New elements can be defined in the hierarchy, while failure domains(e.g. a shared power source, a shared network, etc.) can be targeted to handle errors more efficiently. In essence, the CRUSH map contains information that helps tracking down the physical locations where the Ceph stores redundant copies of data. It is quite common for larger clusters, that each pool may have its own CRUSH ruleset, replication strategies or distribution policies. CRUSH models the underlying physical structure of assets and can, therefore, identify sources of failure.

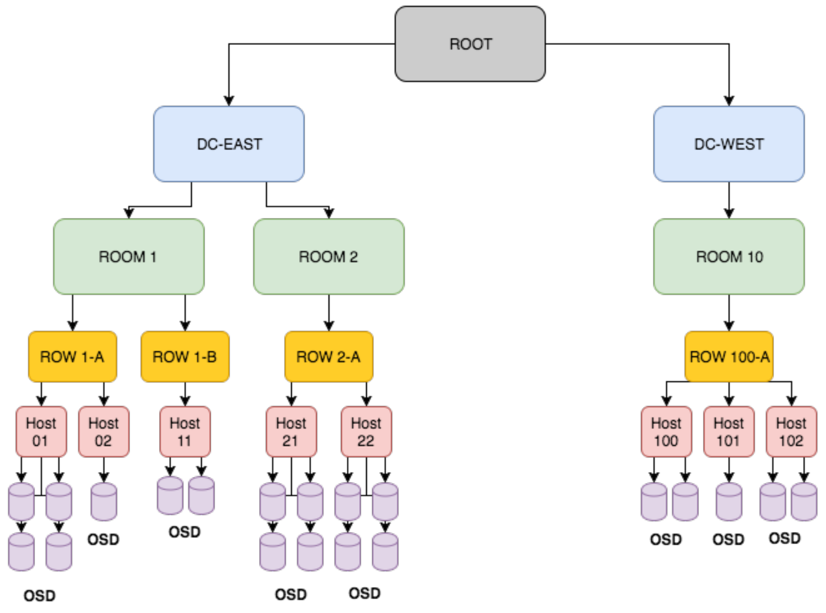


Figure 2.11: *Example of a cluster topology*

CRUSH Algorithm Overview

The CRUSH algorithm that Ceph uses was proposed in [23]. Within storage systems, its goal is to distribute object replicas randomly among available storage resources and lead to a probabilistically balanced distribution, by uniformly mixing old and new data together. This approach has the critical advantage that, on average, all devices will be similarly loaded, allowing the system to perform well under any potential workload. CRUSH is a deterministic, pseudo-random function that maps an input value, typically an object or object group identifier, to a list of devices that will store the desired replicas of the object. For that, CRUSH only needs a compact, hierarchical cluster map that describes the devices available in the storage cluster, along with the knowledge of the replica placement policy. This description usually comes in the form of a hierarchy that contains storage devices, weights and placement rules. In addition, CRUSH ensures stability and efficient reorganization of data, moving around a very limited number of objects in case storage devices are added or removed. Being aware of the underlying physical infrastructure, organization policies and rule sets, CRUSH can effectively detect sources of failure and maintain the desired distribution of data across the cluster.

Key CRUSH properties

- ***Avoids failed devices***, by taking care of the data placement itself and managing where to place objects. Replication across multiple devices acts as a safety valve in case of unexpected failures.
- ***Works as a function***, allowing clients to do calculations on their side and directly communicate with storage devices, without the need of a central proxy. This eliminates any bottlenecks and increases cluster performance and scalability.
- ***Pseudo-random***, meaning that the set of devices that sharing replicas for one item appears to be independent of all other items. This prevents concurrent, correlated failures from causing data loss.
- ***Fast***, as placement calculation takes up microseconds, even for large clusters.
- ***Deterministic***, since identical inputs produce identical outputs.

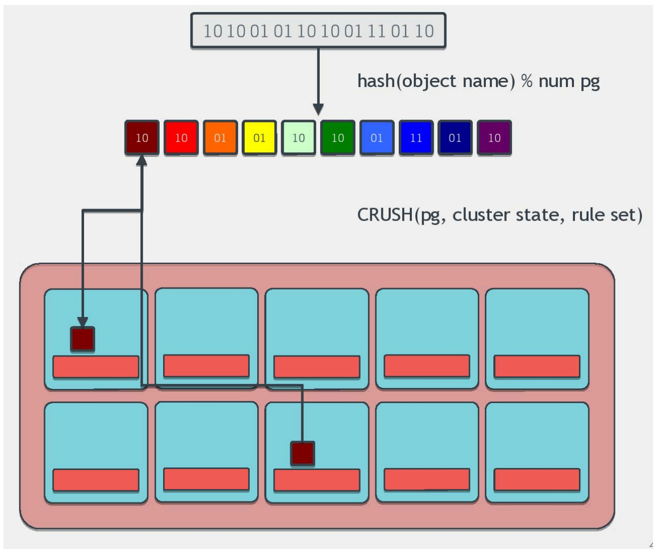

10 10 01 01 10 10 01 11 01 10
hash(object name) % num pg
10
10
01
01
10
10
01
11
01
10
CRUSH(pg, cluster state, rule set)

Figure 2.12: *CRUSH calculation overview*

Using CRUSH as a Ceph client

Having in mind the above, we can proceed and present the way a Ceph client discovers the placement of a specific object inside a Ceph cluster. All the calculations listed below take place in the **client-side**, imposing no additional overhead to the Ceph cluster. Of course, these calculations can be performed in other any entity(OSD, MDS) verifying that CRUSH is fully distributed in design. Once the client discovers which storage device is currently associated with the specific object, he directly connects to that OSD and performs any object-related operation.

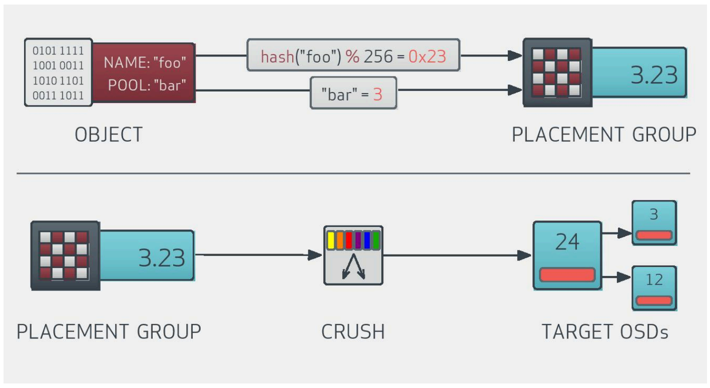


Figure 2.13: *CRUSH 2-step calculation*

As shown in the figure above, the **object name** gets hashed and then a `modulo` calculation into `pg_num` is performed. Given the pool that the object belongs to, the pool id is appended to the result of the hash function, ultimately giving the id of the placement group of the object. In order to locate the target OSDs, the PG id is passed to CRUSH, along with the latest OSD cluster map(cluster state, hierarchy & placement rules) that the client obtains from the Monitors. CRUSH performs various calculations and responses with an ordered list of n OSDs, for n-way replicated storage policy. The length of the list, n, must be equal to the replication factor, while the calculated locations are explicit and not user-selected. All write operations coming from the client are directed to the primary OSD of the respective PG that the object belongs to. The object version gets updated and then the primary propagates the write operation to all additional replica OSDs(secondaries). In this way, CRUSH simultaneously solves both the problem of effective distribution and instant location of data. This method results in a statistically even distribution of objects into different OSDs available across the cluster.

2.3.4 Monitoring a Ceph Cluster

When interacting with a Ceph cluster, in order to guarantee high performance and scalability, it is crucial to supervise the underlying infrastructure, to diagnose issues, handle common errors and ensure that every component is operating as expected. Monitoring a Ceph cluster from a higher level ensures that all Ceph daemons(Monitors, OSDs, MDSs) are up and running, while administrators can retrieve specific information about individual Ceph processes, using some of the native CLI tools. If a daemon is not running or under-performing, it is vital to detect the source causing the issue. For example, if an OSD is marked down, that could imply that an error concerning its corresponding disk has occurred. Similarly, if an OSD is marked out, network connectivity issues may exist. Therefore, dedicated monitoring services should be introduced to keep track of cluster behavior, states and infrastructure-wide historical data, such as throughput and latency. Besides, a Ceph cluster should also be monitored with regard to the physical assets and storage devices it employs(e.g. CPU usage, memory availability, disk stats, etc.), so that a full-stack overview of the deployment can be concluded. Hardware metrics can be correlated with the performance of the Ceph cluster, leading in a deeper understanding of dependencies, the discovery of patterns and, in the long run, reduction of recovery times and costs. In addition, administrators can fine-tune individual components, by pinpointing bottlenecks and appropriately provisioning resources. Some of the most important aspects of monitoring and investigating Ceph's operations are given below:

- **Cluster Health**
  Indicates the overall status of the Ceph Cluster: "HEALTH_OK" status obviously means that the Ceph cluster is up and running, and that there are no observable issues that were detected by Ceph's built-in self-diagnosis. Other possible statuses are "HEALTH_WARN" and "HEALTH_ERR". The built-in command line tools can provide more detailed information about the health of the cluster.

- **Cluster Capacity**
  Total storage availability is a must-watch metric, as near-full clusters may face performance issues. Also, even distribution of objects across the cluster requires a balance between the number of OSDs, PGs and replicas, so storage capacity should be monitored at every level(cluster, pool, placement group, OSD).In some cases, for example, instead of simply aggregating metrics from divergent sources together, it is useful to monitor each pool separately. This means running pool-specific checks that store performance statistics like read/write throughput, capacity usage, the number of stored objects, etc. in pool-level resolution. System administrators can leverage this feature to gather details about each pool usage pattern and quickly spot overloaded pools to tune the application for better performance.

- **Monitors**
  Since Monitors are the dedicated masters of the cluster, admins should know which

Monitors currently form the quorum, as well as the latest monitor map epoch. It is important to get timely alerts when monitor nodes go down, in order to avoid deadlocks during the voting process of the monitor quorum

- **OSDs**
  As mentioned before, OSDs can be in "UP, DOWN, IN" and "OUT" states. It is important to be aware when an OSD is simultaneously "DOWN + IN", as Ceph is likely to experience difficulties in recovering or migrating data from this node and manual intervention might be needed. In addition, OSDs should be monitored in terms of capacity, as near-full OSDs may break the data placement balance of the cluster.

- **MDSs**
  In the case of CephFS, it is mandatory to ensure MDS nodes are active and responsive, as they co-ordinate every file-based operation on RADOS.

- **Placement Group states**
  Proactive monitoring of PGs can lead to the detection of inconsistencies in the Ceph cluster. Knowing the states of PGs(*active, clean, inactive, unclean, undersized, degraded or stale*) indicates which operations are currently taking place inside the cluster and helps understanding and isolation problems.

- **Cluster throughput & IOPS**
  Throughput is defined as the maximum rate of production or processing. Thus, its value can depict the data rates delivered by the Ceph Cluster and outline how well it performs. This measurement is closely related to the number of IO operations performed per second.

- **Cluster latency**
  If caching is enabled, Ceph does not write all updates directly to disk. A transaction is marked complete, after the operation is successfully committed to the journal - the time required for this step is reported as "commit latency". Data is permanently stored when actual data is written to disk - the time required to flush the operation to disk is reported as "apply latency". Observing these times ensures that the system is performing quickly enough and is sufficiently robust to data loss.

- **CPU, Memory utilization**
  Ceph daemons are quite demanding, regarding CPU and memory resources. There-

fore, it is essential to watch usage and availability percentages of processors and memory chips, to ensure that each cluster node is functional. Furthermore, spikes in cpu and memory usage might indicate congestion or unusual workload in specific storage components.

- **Disk statistics**
  Disks, being the physical storage devices where user data is ultimately placed, should be extensively monitored to prevent data loss or even predict future failures. High temperature, slow IO and high latencies might identify a failing disk. Timely equipment replacement assures that the corresponding OSD can continue performing its tasks normally.

- **Network activity**
  In a large cluster, network traffic congestion is a frequent phenomenon that can burden performance. Simple `ping` checks show if cluster nodes are connected and responsive. Furthermore, tracking of overall throughput, exchanged packets, incoming or outcoming traffic, response times and the number of transactions can certainly verify if the network stack of the cluster is operating normally.

2.4 Scalable Vector Graphics

2.4.1 About SVG

Scalable Vector Graphics [26] is an `XML`-based vector image format for two-dimensional graphics, with comprehensive support for interactions and animations. SVG documents are dynamic in nature and, generally, highly customizable. Their content is stylable, scalable to different display resolutions, and can be viewed stand-alone, mixed with `HTML` content, or embedded within other `XML` languages by using respective namespaces. The World Wide Web Consortium (W3C) originally developed the SVG specification in 1999 as an open standard. The latest release is 1.1, while SVG version 2 is currently under development and still in draft. There are, also, SVG profiles that are designed for mobile.

2.4.2 Overview

SVG is a format that is completely independent of the resolution of the rendering device, typically a display monitor. Its platform consists of two parts: an `XML`-based file format and API for graphical applications. SVG is a standard that is used in multiple business areas including Web graphics, animations, user interfaces, mobile applications, graphics interchange and high-quality design. This is proved by the fact that all major modern web browsers provide SVG rendering support. Sophisticated applications of SVG are possible by accessing SVG Document Object Model(`DOM`), which provide complete access to all elements, attributes and properties. In addition, a rich set of event handlers, such as `onmouseover`, `onclick`, `onload` etc. can be assigned to any SVG graphical object to provide interactivity. The MIME [21] type for SVG is `image/svg+xml`, its namespace is `http://www.w3.org/2000/svg` and its public identifier is `PUBLIC"-//W3C//DTD SVG 1.1//EN"`. SVG integrates well with other W3C specifications and standards, such as: `XML,HTML,CSS` and `SMIL`[22]. By conforming to multiple open standards, SVG becomes more powerful and enables users to incorporate vector graphics into their Web sites. At this point, it is essential to analyze the core characteristics of the SVG standard:

Scalable
In terms of graphics, scalability means not being limited to a single, fixed, pixel size and, hence, adapting to multiple resolutions without sacrificing quality. On the Web, scalable means that a particular technology can grow to a large number of files, users or applications. SVG, being a graphics technology intended for the Web, is scalable in both senses. SVG graphics are scalable to different display resolutions while the same SVG graphic can be placed at different sizes on the same Web page. Another important parameter to note regarding SVG scalability is that their content can both be viewed as a stand-alone graphic and be referenced or included inside other SVG graphics. Thereby, it is possible to build a complex illustration in smaller parts, perhaps by several people.

Vectorized
Vector graphics use geometric objects such as lines, curves and polygons to represent

[21]Multi-purpose Internet Mail Extensions
[22]Synchronized Multimedia Integration Language

images. They are based on vectors, which lead through locations called control points or nodes. This gives greater flexibility compared to raster-only formats (such as PNG and JPEG) which have to store pixel-specific information for the enclosed graphic. Typically, vector formats can also integrate raster images and combine them with vector information such as clipping paths to produce a complete illustration. Since all modern displays are raster-oriented, vector graphics need to be rasterized before they are rendered; this process is computationally intensive and usually takes place on the client side. Of course, raster-based graphics are by definition compatible with raster-oriented displays. The SVG standard provides control over the rasterization process, such as anti-aliasing low-quality vector implementations.

Stylable

Scalable Vector Graphics take advantage of style sheets in terms of presentational control, flexibility, faster download and improved maintenance. SVG extends this control and offers great customization capabilities. The combination of scripting, manipulation of DOM and generation of CSS code is often termed "Dynamic HTML" and is widely used for animation, interactivity and presentational effects over vector images. SVG allows the same script-based manipulation of the DOM tree and the style sheet.

XML-based

XML is a standard for structured information exchange, has become extremely popular and is both widely and reliably implemented. Being written in XML, SVG builds upon this strong foundation and gains many of the advantages XML offers. More precisely, it has a sound basis for internationalization, powerful structuring capabilities, an object model, etc. By building on existing, cleanly-implemented specifications, XML-based grammars are open to implementation without a huge reverse engineering effort.

Namespaced

SVG is also intended to be used as one component in a multi-namespace XML application. This multiplies the power of each of the namespaces used, to allow innovative new content to be created. For example, SVG graphics may be included in a document which uses any text-oriented XML namespace - including XHTML. SVG can be regarded as a general-purpose component for multi-namespace environments that need to use graphics.

2.4.3 Fundamental Features

Some of the most important features that the SVG standard supports are listed below:

- **Graphical Objects**

  SVG models graphics at the level of objects rather than individual points. Therefore SVG provides a general path element that can be used to create any graphical object, by outlining curved or straight lines. As some geometrical shapes are quite common in applications, SVG provides them as standard graphical objects, like `<rect>`,`<circle>`,`<ellipsis>`,`<line>`. SVG provides fine control over the coordinate system in which graphical objects are defined and the transformations that will be applied during rendering. Essentially, `XML` is used to describe what the image should look like by utilizing predefined attributes. Coordinate system inside the SVG are determined by the `viewport` and `viewBox` attributes.

(a) *circle* **(b)** *rectangle* **(c)** *star*

Figure 2.14: *SVG Shapes*

- **Text & Fonts**

  Graphically rich material is often highly dependent on the particular font used and the exact spacing of the glyphs. In many cases, designers convert text to outlines to avoid any font substitution problems. This means that the original text is not present and thus searchability and accessibility suffer. In response to feedback from designers, SVG includes font elements so that both text and graphical appearance are preserved.

- **Painting & Colors**

  Graphical objects can be modified in terms of coloring by assigning their fill and stroke properties to 3 or 6-digit hex or RGB values. Gradients, patterns and transparency are also supported.

- **Filtering & Effects**
  SVG allows the declarative specification of filters, either singly or in combination, which can be applied to the client-side when the SVG is rendered. These are specified in such way that graphics are still scalable and displayable at different resolutions. Common filters involve blurring, lighting, color adjustments, etc.

- **Scripting**
  All aspects of an SVG document can be accessed and manipulated using scripts in a similar way to HTML. The default scripting language is `ECMAScript`, which is closely related to `JavaScript` and there are defined `DOM` objects for every SVG element and attribute. Scripts are enclosed in `<script>` elements. They can run in response to pointer events, keyboard events and document events as required.

- **Transformations**
  The transformations supported by SVG are translation, rotation, skewing, scaling, as well as matrix operations. All transformations are summed up in an element's `transform` attribute can be chained simply by concatenating them, separated by whitespace. When using transformations upon an element, a new coordinate system is established. Thus, user-specified units for the element and its children might not follow the 1:1 pixel mapping, as they might be skewed, translated or scaled based on the applied transformation.

- **Animations**
  SVG has built-in support for animations, using `SMIL`, but it is also compatible with `CSS` animations and specifically `keyframes`. Animations can be triggered either declaratively or via script-based manipulation of the document aiming at visualizing combinations of transforms. Main SVG animations focus on translation, scaling, rotation, motion, paths, etc. Control values of attributes can be specified, along with specific timing, repeat count and duration. Also, SVG elements can visualize multiple animations, either simultaneously or serially.

- **Metadata**
  In accord with the W3C's Semantic Web initiative, SVG allows authors to provide metadata about SVG content. The main facility is the `<metadata>` element, where the document can be described using Dublin Core metadata properties (e.g. title, creator/author, subject, description, etc.). Other metadata schemas may be used as

well. In addition, SVG defines `<title>` and `<desc>` elements where authors may also provide plain-text descriptive material within an SVG image to help indexing, searching and retrieval by a number of means.

2.4.4 Vector vs Raster

In current Web there is a strong debate concerning the usage of the two main types of image files: *Raster* and *Vector*. Raster images consist of pixels and are created with pixel-based programs or captured with a camera or scanner. They are very commonly used today (e.g. photography), including popular format extensions like `jpg/jpeg`,`png`,`bmp`, and `gif`. Raster file formats store data related to each individual pixel, while the number of pixels is finite. Though rich in detail and precisely editable, raster images are resolution dependent, meaning they cannot scale up to an arbitrary resolution without loss of apparent quality. For example, when enlarged, a blurring effect occurs, as computers automatically fill with arbitrary colors the extra pixels, not knowing the exact desired shade. This interpolation of data causes the image to appear blurry since many pixels are wrongly-colored. Once the image is created at a certain dimension, enlarging the image is not feasible without losing quality(pixelation). Anti-aliasing techniques can be used, but they are not adequetely efficient in big zoom levels. In addition, raster images are usually large files(depending on the `dpi`) and require a considerable amount of processing power to be handled. Compression of raster images also causes partial loss of data (e.g. straight lines might appear "jagged" after compression).

On the contrary, vector images do not store per-pixel information; instead, they are based on mathematical formulas and geometrical primitives to draw shapes, using points, lines and curves. They are produced by vector-based software and have been recently gaining more attention. The construction of such images relies on the defining a set of control points and strong mathematical models used to link them. Given a vector image input, a viewing program interprets the vector image as a set of instructions that maps a series of grid points(coordinates) to the place where the lines or curves are to be drawn. This guarantees smooth and crispy rendering no matter the zoom level. For example, a raster image of a "`1x1`" square at 300 `dpi` will have 300 individuals pieces of information, while a vector image will only contain four points, one for each corner. Accordingly, the computer will use mathematics to connect these points and fill in all of the missing information. Vector images are quite popular in font and logo design, including file formats like `svg`,`pdf` and `ai`.

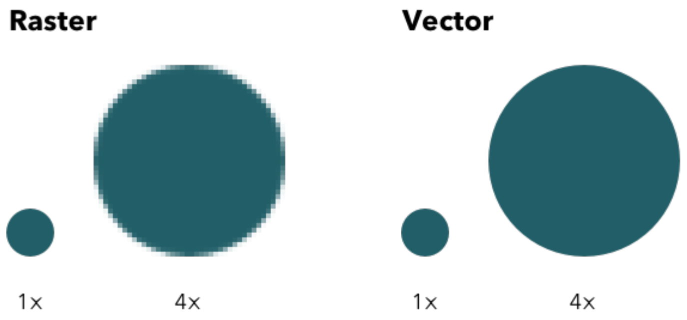


Figure 2.15: *Raster vs Vector*

Rasterization Since modern display monitors and printers are raster/bitmapped devices, vector image formats have to be converted to raster format (bitmap/pixel arrays) before they can be rendered or printed. Intuitively, this implies that discrete fragments must be created from continuous surfaces. During this process, in practice, each grid point (or fragment) of the vector image corresponds to one pixel in the frame buffer and this is represented in one pixel of the screen. These can be colored and possibly illuminated. Raster devices correlate pixels directly to a certain number of bits in memory. The size of the (output) bitmap/raster format file generated by the conversion will depend on the resolution required, but the size of the (input) vector file generating the bitmap/raster file will always remain the same. Thus, it is easy to convert from a vector file to a range of bitmap/raster file formats, but it is quite more difficult to go in the opposite direction, especially if subsequent editing of the vector picture is required.

2.4.5 Why use SVG?

Taking the aforementioned into consideration, we believe there are several good reasons for choosing SVG as the image format for the needs of our monitoring application. More specifically SVG:

- is more powerful and flexible compared to other image formats used on the web, as its nature enables higher accessibility via programmatic control and internal code manipulation
- offers infinite scalability and higher quality at any size of display with no data loss

- as vector graphics are not composed of pixels they are resolution-independent. This goes back to the nature of vector graphics, which is visualizing the original mathematic equation that creates a consistent shape every time, computing the path between points.

- is object-based and thus *self-aware*, in the sense that, each element "knows" its attributes and, as such, can be altered or morphed at run-time. Also, it supports single object or layer modification without affecting other objects in the image.
- is file-size efficient and portable, as it more coarse-grained in design and produces smaller files that are easier to store and transmit over networks.

2.5 Web Application Frameworks

2.5.1 Overview

It is undeniable, that Web development is a field of software engineering that is constantly gaining in popularity, since more and more systems and services are becoming web-oriented. As the Web matures, there have been major changes in the challenges that developers face daily, mainly regarding the sophistication of modern web applications. Clients have growing demands from web-hosted services, varying from improved performance and stability, to additional features and shorter development times. As a result, the process of developing services and applications intended for the Web can become rather complex and confusing, if the right toolbox is not selected.

In general, a framework can be regarded as any real or conceptual structure that facilitates the building or the expansion of a layered structure into something meaningful. In the field of computer systems, a Web Application Framework is a set of resources and tools designed for software developers, aimed at facilitating the building and management of web applications, web services and websites. It provides pre-defined templates, high-level APIs, modular libraries and built-in functions to the developers, for accessing data resources and adding capabilities to their applications. At the same time, web application frameworks effectively hide much of the boilerplate code needed during initialization, abstracting low-level concerns from developers. It must be clarified, that a web development framework is not a standalone software system - instead,

it is usually built upon an existing infrastructure or stack, that usually involves operating systems, database management systems, web servers and others.
The main reason for using a web framework is the automation of the development process. Specifically, it absolves hard-coding every individual feature of a large system while it offers a structured and tested design for the application. Essentially, much of the application functionality is deferred to the framework, compared to directly calling methods provided by a library. In this regard, web frameworks are founded upon the principle of *not reinventing the wheel*, leading to valuable time and effort savings. The majority of modern web frameworks are entirely open-source, actively developed and supported by large programming communities.

2.5.2 Web Application Architectures

Application frameworks may have different foundations and principles, regarding their approach to design applications and interfaces. For the scope of this thesis, it is sufficient to examine two basic architectural approaches, along with two fundamental design patterns:

MVC Architecture

MVC stands for Model-View-Controller and is a traditional software design pattern, that is widely used when building user interfaces. Its primary objective is to isolate business logic from the user interface, by separating concerns into three interconnected units. This allows for decoupling of components and parallelization of tasks during the development process.

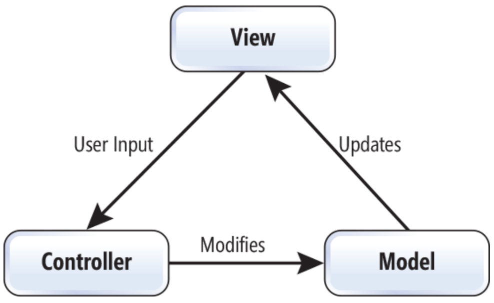


Figure 2.16: *MVC components and interactions*

- **Model**
  The Model is the central component of the pattern, holding application-related information in the form of data and business rules. It is usually represented by objects that manage application logic, behavior and details with regard to the problem domain. The Model is responsible for storing updated data and properties coming from the Controller. The current content of the Model is always presented in the View via updates.

- **View**
  The View is the layer that is closer to the user, consisting of the elements forming the user interface. Multiple views of the same information might exist simultaneously, providing any data outputs or visual representations that the application produces(e.g. buttons, charts, forms, etc). The View is the component that directly interacts with the user, perceiving actions and events.

- **Controller**
  The Controller is responsible for interacting with both the Model and the View, managing the communication between them. It controls the data flow in both directions: when user action events (e.g. mouse clicks, keystrokes, etc) are captured and propagated from the View, it first runs the corresponding handlers and then issues corresponding updates to the Model, regarding properties, methods, or entities.

In short, using the MVC pattern during the development process of both server- and client-side applications, has the following benefits:

- Parallel development of the application, since logically related operations and actions are grouped together
- Each component is independent and does not block other components
- Reusability of code, since developers can build or refactor applications based on already designed stand-alone components
- Ease of maintainability and future modifications

CB Architecture

Component-based architecture is a design concept that encapsulates individual components of a complex system into self-managed subsystems. In this approach, components are the basic building blocks of the interface and each of them offers a specific service to the system. Components perform tasks independently from each other and are usually implementing dynamic features, such as client-side request handling, auto-refresh and re-render of document elements. Because of these principles, component-based applications are regarded highly modular and highly cohesive. Frameworks adopting this development strategy are capable of handling components in a very efficient and performant way. Furthermore, CBA ensures consistent component design, since it requires that all APIs and methods related to a component must reside in its structure. In practice, components are implemented in the form of classes, objects or related mechanisms. Of course, developers can maintain multiple instances of these classes and reuse their features in different parts of the interface.
The main difference between CBA and MVC architectures can be tracked down to the way that each architecture splits the responsibilities of the application. MVC frameworks separate concerns *horizontally*: this means that all features needed by each template of the UI are implemented at different levels of the architecture. Conversely, CBA frameworks separate concerns *vertically*, in the sense that each component of the application retains its design, logic and corresponding methods at the same level of the architecture.

SPA vs MPA

Since web applications are gradually replacing desktop applications(e.g. social networks, mail services, cloud storage, etc.), two main design patterns have emerged with regards to web application development: *Multi-Page Application* (MPA) and *Single-Page Application* (SPA). Depending on the purpose, content and the scope of the application, either model might be more suitable.

- **Single-Page Model**
  Applications that adopt the SP model work inside web browsers without any page reload or refresh during runtime. All contents and resources of a web page are

loaded dynamically only once during startup, mainly using AJAX[23]. While in use, only session-bound data is transmitted. This approach aims at minimizing waiting times, reducing network bandwidth utilization and providing a more "native-like" user experience. SPA applications are, generally, fast, streamlined and rather easy to debug, by monitoring network operations and investigating DOM elements. What's more, SPA is capable of effectively caching local storage and, consequently, still be operational offline. On the downside, SPA applications might be more vulnerable to security threats, such as client-side script injections via `XSS` (Cross-Site Scripting). Also, SEO[24] is a quite complex task for SPA applications.

- **Multi-Page Model**
  The MP model represents the traditional approach to designing applications. When any update is issued to the application, like a user action or data submission, the application propagates a request back to the server. As soon as the server successfully handles the request, it responds to the application and a new page gets rendered in the browser. As expected, MPA-based applications are larger than SPA-based ones and consist of multiple UI levels. Also, it is very common that the same or similar pages of an application will be repeatedly requested so, ultimately, duplicate `HTML` code is going to be transferred and loaded. Usage of `AJAX` requests during the communication between the server and the application limits the amounts of markup transferred over the wire, since in most requests only fragments of the application actually need to be refreshed. Moreover, MPA-based applications require the development of both the front- and the back-end services, resulting in a more complex and time-consuming implementation.

2.5.3 Angular

About

Angular [27] is an open-source, component-based, front-end web application platform. It is the full-platform successor of the widely popular AngularJS framework, providing core libraries, multiple capabilites and a powerful compilation engine. Angular introduces `Typescript`, which is a typed superset of `Javascript` designed to

[23]Asynchronous JavaScript + XML
[24]Search Engine Optimization

scale. Being cross-platform, it can run in heterogeneous systems, while it is optimized for developer productivity and performance, supporting end-to-end testing. Due to the various modern features it introduces, Angular is considered as one of the most promising and future-proof frameworks of our days, largely inline with the W3C Web Component specifications [28]. It is a well-documented framework, currently developed and supported by Google, in collaboration with a large community of individual contributors.

Angular delivers single page applications that mainly follow the component-based architecture. Everything inside the application can be regarded as a component. However, each individual component is independent and implemented in an MVC-like logic. Due to structural dependencies, developers are constantly working on improving compatibility with emerging standards. Componentization of web applications enables developers to precisely modularize, test and determine both behavioral and presentation layers. In addition, Angular offers great versatility regarding target applications, as it is a framework designed also for mobile; supporting server-side or web worker rendering of `HTML`, it leverages performance gains and fits the needs of portable devices. Angular is entirely written in `TypeScript`, which offers familiar syntax and semantics, along with support for the latest and evolving `JavaScript` features coming from `ECMAScript2015`, such as `types,classes,decorators,async functions` and others. These features provide developers with a richer API with stronger tools, while compiling to clean `JavaScript` code that can be executed on various engines. Of course, `Typescript` code is fully compatible with `JavaScript` libraries, as long as the corresponding types for the APIs are defined. The design of Typescript is focused on optimizing performance, together with automating the development workflow. The CLI of Angular is also inline with these principles, as it is a well-rounded tool, that simplifies the generation of code, configures typings and deploys the application.

Core Concepts

- **Components**
  Components are the cornerstone of every Angular application. Patches of screen or UI elements are always associated with components that contain their logic and control their behavior. Components follow a tree-based structure, since every An-

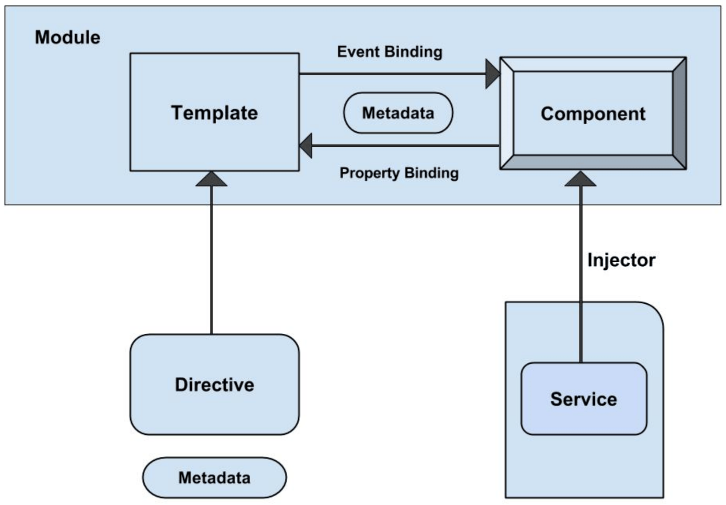


Figure 2.17: *Overview of Angular Architecture*

gular application is based on a root component that contains all other components in subsequent levels. In such a structure, tree nodes represent components and edges represent the way components are connected or interact with each other. They are implemented using a class based model that introduces a view and and the associated controller code. The corresponding properties and methods of each component are defined in the body of the class. Component controllers manage the model, in terms of data, and update accordingly the current state of the view. Commonly, a component is tightly coupled with a `template`, which is a set of native `HTML` and custom directives that tell Angular how to display this specific component and render it in the browser. Component templates can be defined either inline, or externally via `templateUrl`. In addition, components include metadata that determine the configuration(e.g. `stylesheets,providers,directives,selectors`) and the processing of the class, using decorators. Angular Components are designed to be portable and reusable in different parts of the application. Communication between the template and the body of the component can be implemented using the following mechanisms:

- **DOM →Component**: event binding inside DOM elements and corresponding handlers in the body of the Component
- **DOM ←Component**: property/data binding or interpolation, associating attributes of the DOM elements with properties of the class

- **DOM ←→Component**: `ng-model` directive inside the DOM, which is enables the bidirectional flow of data

Besides, since components frequently interact and exchange data with each other, a dedicated mechanism is available to define input and output properties, using `@Input` and `@Ouput` decorators, respectively.

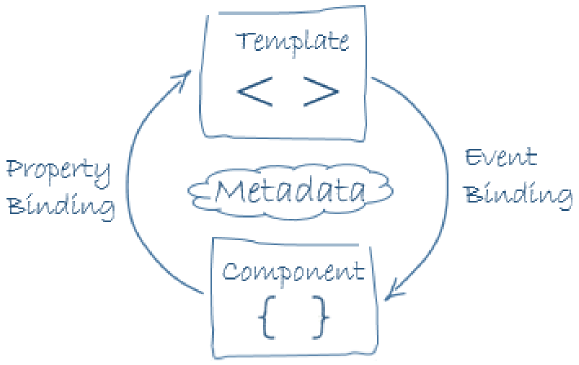


Figure 2.18: *Angular Component*

- **Services**

  Services are complementary to Components and are entities of code that efficiently perform a well-defined task. For example, one service might be responsible for communicating with a web server via HTTP, while a different service might be defined to digest and visualize data coming from in the body of the server response. Thus, services are quite broad as a concept, encompassing any data structure, function or feature the application needs. Typically, a service is a narrow, bounded class that serves specific operational purposes. The importance of defining services is revealed as the complexity of the application grows: when components are confronted with multiple tasks and complicated logic, they can import services to separate concerns. This eventually leads to improved readability, higher reusability and easier maintenance of code. Services factor application logic and are made available to components through the *dependency injection* mechanism.

- **Modules**

  The structure of Angular applications is, generally, modular. A Module is cohesive group of code that is bundled together and integrated with other modules to run applications. Modules mostly consist of components and services, and usually export methods and classes that other modules might find useful. They essentially play the role of libraries and provide additional functionality to applications when imported.

- **Dependency Injection**

  Angular introduces Dependency Injection(DI) as a standardized tool to import dependencies into components, modules or other services. In essence, it is a design pattern that passes an object to different components across the application, and creates new instances of its class and its inner dependencies. An *injector* mechanism maintains a container with the service instances needed by the application. Each time a *provider* is declared in the component's metadata section, the injector returns an instance of the requested service. When the service is resolved, the constructor of the component's class is called with that service as an argument.

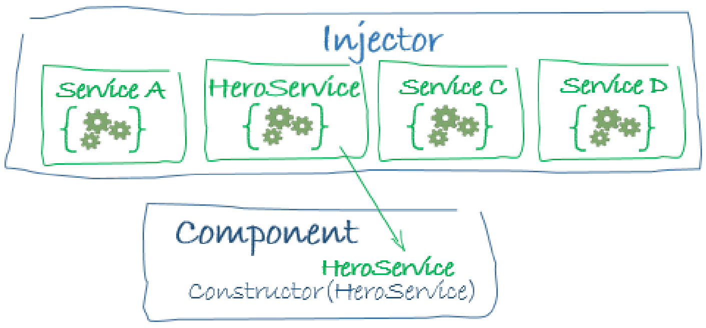


Figure 2.19: *Angular Dependency Injection*

- **Directives**

  To ensure templates are both flexible and dynamic, Angular provides Directives, which are classes that control the transformation of the DOM during the rendering process. Directives contain metadata that is attached to the component class by the `@Directive` decorator. Angular supports 3 different types of directives, while custom ones can also be defined:

  - **Directive-with-a-template:** this is the typical definition of components - the `@Component` decorator is a template-extended `@Directive` decorator.
  - **Structural directives:** these directives are the ones updating the `DOM`, by removing, adding and replacing elements(`ngIf`,`ngFor`,`ngSwitch` and others).
  - **Attribute directives:** these directives are responsible updating the appearance and behavior of `DOM` elements. In templates they look like regular `HTML` attributes.

Why Angular?

Designed for Performance

Angular applications set high performance as a priority by design. Compared to other web application frameworks, it provides various strategies and mechanisms to overcome performance bottlenecks and limitations. At first, Angular does not generate raw `HTML` code and provide it to browsers for parsing; instead, its engine builds the `DOM` nodes directly, entirely skipping the step of `HTML` parsing by the browser. Thus, rendering of the application is a matter of attaching ready-to-use nodes to the `DOM` tree of the browser. At the same time, since the rendering module is isolated, intensive computations can be executed in dedicated worker threads.

The concept of an independent renderer allows for customization of templates and higher performance. The Angular compiler offers two alternatives: *Just-In-Time*(JIT), which compiles the application code every time it is loaded in the browser using different sets of libraries, and *Ahead-Of-Time*(AOT), which compiles the application code only once at build time using one set of libraries. While JIT is the default pattern and widely used in development, AOT offers multiple benefits and is heavily used in production environments. More specifically, AOT:

1. Ensures fast rendering, as the browser loads a pre-compiled version of the application,
2. Effectively reduces the application size, since `JavaScript` payload is compressed and there is no need to download the Angular compiler and
3. Avoids template errors, as they are reported and detected during the build step.

In addition, Angular applies optimal planning concerning the detection of changes in the `DOM` tree, by avoiding unnecessary checks and focusing only on the parts of the model that were possible to change. As shown in Figure 2.20 the change detection graph of Angular represented by a directed tree, making our system much more predictable. By default, Angular will conservatively check all nodes of the tree - however it is possible to prune multiple paths of the tree and avoid recursive scans when dealing with immutable objects. For example, a Component depending only on immutable input attributes can be skipped during checking. Alternatively, *Observables* can be used for asynchronous handling of updated values, only when they are emitted.

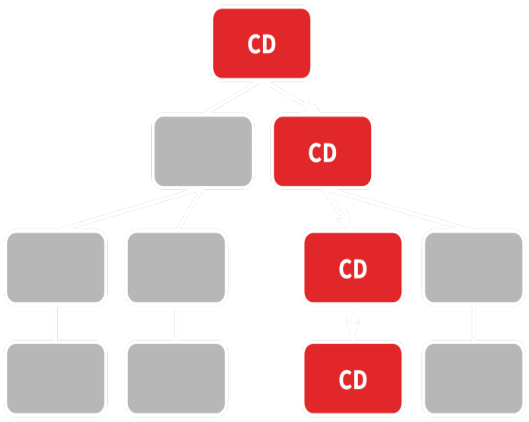


Figure 2.20: *Angular Change Detection*

Aligned with Web Components

The design decision of the Angular team to adopt a component-based approach is definately an attempt to incorporate Web Components, a set of future Web standards and specifications, into modern applications. Notably, Angular encourages developers to create their own custom Web elements during the development of web applications, such as `HTML tags,pipes,events,decorators` etc. Furthermore it supports flexible `Shadow DOM` manipulation, in the sense that interactions and boundaries between `DOM` trees can be established, enabling improved functional encapsulation of documents. Addition of properties and mixins of the `Shadow DOM` is also possible. Finally, in Angular applications `HTML` documents can be included or reused in other `HTML` documents - this feature improves code readability and extends functionality.

Modularity

Angular is a highly modular framework that builds upon well-defined, independent entities that serve specific purposes. Organization and grouping of related components into modules produces cohesive blocks of functionality that can be reused across the application. Modules are flexible, extendable and provide additional capabilities to the application, resembling external libraries. Angular implements many of its core APIs and features in the form of native modules(such as `FormsModule, HttpModule, RouterModule`. Each module has a narrow application domain, to separate concerns and follows a typical workflow, to guarantee efficacy. Besides, this approach expedites the process of error detection and correction, since each functionality of the application is associated with a specific module. Another benefit of using modules is the *lazy loading* feature; Angular can load modules on-demand, initially loading only the core and necessary features that the user expects. This method can effectively decrease startup time and enhance the user experience.

Reactive Programming

Angular is a framework that is capable of implementing applications that are based on functional & reactive programming. This type of programming relies on *asynchronous data streams*, which are sequences of data made available over time. Therefore, function-oriented programs store their state on streams, and not inside their application code. This approach is very performant, less error-prone and rarely produces side effects. Though functional programming has only recently been adopted as a best practice in the world of frontend development, Angular already supports, as it is compatible with the `Rxjs` [25] library and internally implements `Observables`, as part of its public API. **Rxjs** is a library that implements `Observables` for `JavaScript`. An `Observable` is a new asynchronous development primitive whose name is steaming from the *Observer* design pattern. The common practice when working with `Observables` is to define sources that emit streams of data, subscribe to these sources, be notified when new values arrive and react accordingly. It is also quite common to combine multiple data streams in order to create new, more complex ones. This event-driven style of programming is an alternative to the traditional object-oriented on of MV* frameworks and is considered very promising for the future. Among others, some standard functional operators that are widely applied to data streams are the following: `map,filter,subscribe,let,flatMap,reduce,merge,combineLatest`.

[25]Reactive Extensions for JavaScript

3 Architecture & Design

In this chapter, we thoroughly analyze the architecture of OntoMon from a high-level perspective, along with the fundamental design decisions we made during the development process. At first, we argue the importance of building a general-purpose monitoring and visualization platform and highlight the design principles that differentiate it from existing monitoring solutions. Then, we examine each individual layer of OntoMon separately and present the open-source technologies we employed for every inner component. In addition, we demonstrate how different layers communicate with each other and provide an extensive description of all intermediate core APIs.

3.1 Design Rationale

As already stated in section 2.2, monitoring software plays a crucial role in the management of large-scale, multi-node, clustered environments that require constant supervision. The IT industry pays great attention to DCIM tools, recognizing them as indispensable components of the IT production stack. Subsequently, numerous monitoring solutions are currently available on the market, sufficiently covering administrator needs. So what is the motivation behind developing yet another monitoring platform?

Having studied and deployed multiple DCIM platforms we concluded that even though existing solutions fulfil their goal, their design principles impose certain limitations regarding broadness and scalability. Despite their high performance and efficiency, the vast majority of existing system monitoring solutions have a rather narrow area of application, in the sense that their design and implementation are targeted at the supervision of specific target systems. In most cases, the underlying modeling of the assets being monitored is rather static since it relies on pre-defined classes, attributes and visualizations that are entirely determined by vendors. What's more, the process of adding, updating or removing entities depends on filling forms or instantiating templates, that contain asset-specific or hardcoded handling. This implies that such solutions rely on mechanisms that communicate with *strongly-typed* APIs, manipulating assets as instances of immutable classes, significantly limiting expressivity and customization. Since modern computing environments are rather multifarious, comprising of diverse hardware and software entities, we argue that a unified, adaptable, general-purpose monitoring platform could leverage complexity and facilitate administration.

In our workflow, we were greatly inspired by the approach adopted in Tendrl [4], a unified software defined controller that transparently manages diverse storage platforms. Currently, Tendrl supports monitoring and provisioning of Ceph and Gluster, though more SDS platforms are planned to be supported in the future . The main objective in the architecture of Tendrl is to maintain a self-sufficient Core Stack, that integrates with different systems and is not affected by possible failures. Tendrl deploys agents on storage nodes, stores state information in abstract `YAML` files with no pre-defined schema, represents all entities as objects and exports a stateless User Interface that provides integration with external systems. What's more, Tendrl mainly works with interfaces and abstract definition files and is, thus, independent of the underlying monitored system.

In this regard, we decided to structure our platform in a similar logic. The main characteristics and design principles of OntoMon are listed below:

- **Integration with Diverse Target Systems:** OntoMon is a platform aimed at monitoring and visualizing diverse target systems, without depending on system-specific concepts or semantics. Our goal was to deliver a **content-agnostic** framework, that is built upon a modular codebase that can be easily configured to manage heterogeneous target systems in a standardized way. Thus, we did not focus its design based on content, but on interoperable placeholders for mutable content. To achieve this, we had to find a convenient, yet consistent way of formulating the description of target systems, either hardware- or software-based. Therefore, we decided to set an Ontology as the foundation of our platform, so that any target system can be determined in detail, in terms of entities and relations. An Ontology is capable of providing a detailed, accurate and consistent overview of the target system. Hence, to automate processing and simplify validation, we introduced a well-rounded schema for the ontological description presenting the inner structure of the target system. This schema involves a certain number of general-purpose attributes that we consider sufficient for explicitly specifying the infrastructure that is about to be monitored. Our decision to build a monitoring framework based on Ontologies also guarantees a strong theoretical background that eliminates vagueness.

- **Abstract Object Model:** In accordance with its ontological foundation, OntoMon represents every concept or asset defined in the target system in the most abstract and flexible model: **Objects**. Working with abstract objects enabled us to introduce generic handlers, focus interactions on interfaces and treat every entity the same, regardless of its semantic content. Objects in OntoMon are extendable, interoperable and integrated into all architectural layers. Notably, no asset-specific actions or hardcoded support are needed, since objects in OntoMon are not associated with specific *types*. Therefore, we decided to keep all communications among the inner components of OntoMon aligned with this generic object logic, by implementing comprehensive `JSON`-oriented APIs and exporting meaningful standard endpoints.

- **Elasticity & Customizability:** One of the main objectives of OntoMon is to provide a versatile core that supports customization on all levels, so that the targeted system can be properly managed, based on user needs. Our design encourages user intervention during initial configuration, aiming at satisfying sensitivity preferences. Users are empowered to extend structure of objects, perform specialized performance checks or analysis upon assets and take full control of the visualization of their infrastructure. Since OntoMon provides general-purpose monitoring services of diverse target systems, we developed a dynamic User Interface that is generated on-the-fly and sits on the top of the rest of our system. In particular, users can benefit from self-descriptive APIs, construct personalized notifications and take full control of the visual elements representing the monitored assets. The sought-after flexibility of our framework is defined in terms of automatically adapting to changes in the target system.

- **Scalable, future-proof, open-source Technologies:** OntoMon employs various open-source technologies in order to provide functionality to individual components. We have taken great care in selected technologies that are scalable by design and bundled them together with some custom middleware, so that they serve their intended purpose inside our platform. Given the fact that a target system is likely to encompass thousands of physical or software assets, together with corresponding configurations and interdependencies, it is vital to set scalability as a priority. We argue that incorporating free software, actively developed platforms and widely accepted open standards, like `JSON` and `SVG`, facilitate integration with external IT systems, prevents vendor lock-in situations and contributes to respective communities.

- **Multilayered Architecture:** Focusing on modularity, separation of concerns and simplification of future maintenance, we propose a multi-level architecture for OntoMon. More specifically, we focused on developing stand-alone, dedicated and performant components, each assigned to a specific task or service inside our platform. Different layers are glued together with well-defined communication mechanisms, that allow to smoothly streamline the flow of data across the platform. This decentralized approach enables the parallelization of the development process and faster detection of potential issues, since each feature is implemented in a specific layer and component.

3.2 Distinction of Roles

At this point, it is necessary to clarify the roles of users interacting with the OntoMon framework. Due to its versatility and modularity, OntoMon is a monitoring platform that is not intended to be used by end-users directly; in practice, the presence and collaboration of a **systems integration engineer** are needed, so that our tool can manage diverse target systems. More precisely:

- The **Systems Integrator**(SI) is an individual or group of developers responsible for regulating and aggregating component subsystems into a whole, while ensuring functionality and automation. A system integrator works in conciliation and cooperates closely with the end-user, so that the various subsystems deliver the intended functionality. It is also quite common that, in modern software-defined platforms, the integrator engineer might be required to provide a customized implementation of software services or components, extending them to serve a personalized purpose. As far as OntoMon is concerned, the system integrator undertakes the task of aligning specific parts of the framework to conform to the underlying infrastructure. In particular, it the duties of the system integrator involve generating the Ontology, configuring the monitoring system and implementing the logic of the inductive reasoner component that initiates the notification process. As expected, the integrator should be aware of the OntoMon architecture and deeply understand its concepts.

- The **End-User** is usually a system administrator or a full-stack developer that is responsible for supervising computing infrastructure and observing the overall performance of the target system. In this regard, the end-user forwards the respective administration needs to the system integrator, so that the monitoring tool can be customized based on his preferences. In our case, the end-user merely interacts with OntoMon via its Web User Interface, which enables real-time asset tracking and supervision of the target system.

3.3 Architectural Overview

In this section, we provide a detailed description of the architectural components of our platform, together with the most important design decisions we made during the development process. OntoMon is a hierarchical monitoring and visualization platform, that consists of three distinct layers that interact with each other. Each of these layers illustrates a different aspect of our framework and, in turn, comprises of various software-defined components that are responsible for delivering discrete operations in a specific area of concern. To provide the reader with some basic intuition, we provide a high-level overview of the architectural design of OntoMon in Figure 3.1:

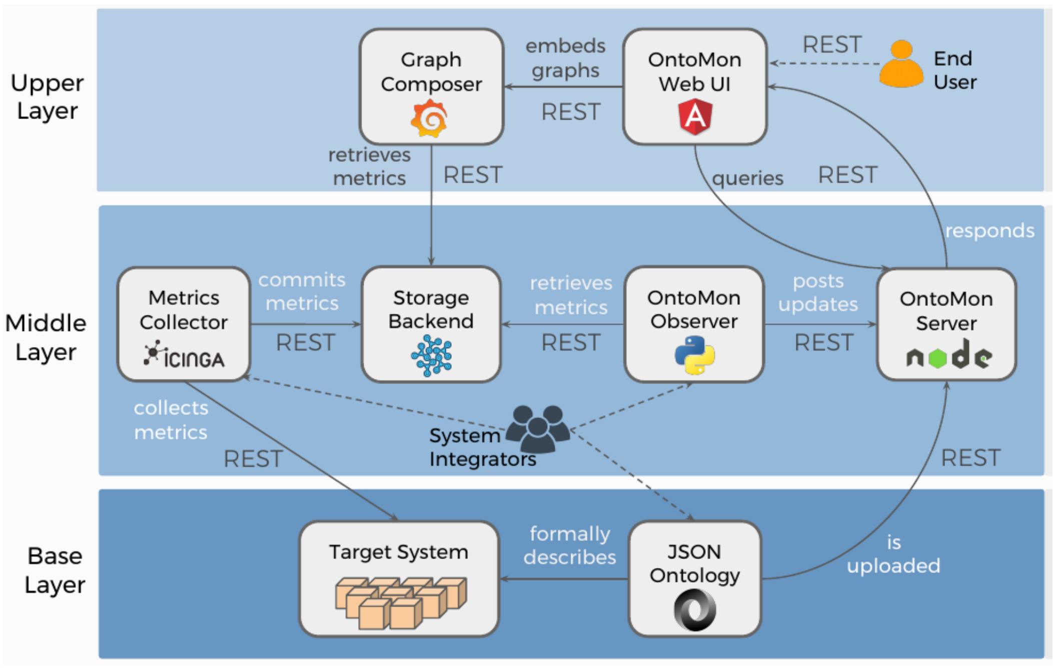


Figure 3.1: *OntoMon Architecture Layers*

Obviously, the data flow starts in the bottom layer, passes through the middle layer and terminates in the upper layer, demonstrating the order of operations. Every layer has certain responsibilities and implements specific features of OntoMon, by tuning and orchestrating numerous individual components:

Layer	**Inner Entities**
Upper	Target System, JSON Ontology
Middle	Monitoring Tool, Storage Backend, OntoMon Observer, OntoMon Server
Bottom	OntoMon Web Interface, Graph Composer

Table 3.1: *OntoMon Architectural Components*

3.3.1 The Base Layer

The base layer determines and formulates the input given to our platform. It consists of the **Target System**, which is a set of related assets that cooperate and are about to be monitored, and the **Ontology**, which is a formal, structured and detailed description adequately determining entities and relations inside the target system, based on certain principles. The Ontology is a representation of the domain of interest, as it captures the basic concepts of the target system, mapping its real-world entities and their relations to well-structured ontology objects.

- **Target System**
  This is the actual computer system that is going to be monitored and visualized by our platform. It is entirely designed and deployed by the end-user for testing or production purposes. Target systems may vary since OntoMon is capable of monitoring divergent platforms, regarding content. However, there is a common assumption that, in all cases and at some depth, the underlying system contains *servers*, that are accessible by the middle layer, both to perform profiling and initially install the collector agents. Every target system is assumed to consist of multiple entities, that are also referred to as **assets**, along with relations or inter-dependencies that exist among them. These assets usually follow a hierarhical structure. Namely, the target system can be hardware-oriented, like a data center along with the physical assets it encompasses, or software-oriented, such as a distributed storage cluster that relies on multiple daemons or services. It is also possible, that the target system combines both approaches; in this scenario, when hardware and software assets are concurrently monitored and visualized, the end-user is provided with a fully detailed overview of his deployment and can observe the whole stack end-to-end. In addition, useful conclusions can be made regarding the system since usage statistics of physical assets and resource efficiency can be correlated with software performance metrics. Indicatively, multi-node distributed software environments or large-scale clusters are good examples of target systems that OntoMon integrates well with.

- **Ontology**
  OntoMon is founded upon Ontologies since the input it requires is a formal ontological description of the target system. It first parses, validates and processes the input Ontology, and subsequently proceeds to the monitoring and visualization operations. In order to formalize the definition of Ontologies that are intended for our platform, a standardized *ontological schema* had to be introduced. This conceptual schema is a blueprint of every entity in the target system and, therefore, had to be concise, uniform, well-defined and formally structured. Thus, we decided to use the **`JSON`** [1] format for the representation of Ontologies, as it offers various benefits. `JSON` is a lightweight open-standard file format, that is both human-readable and

[1]JavaScript Object Notation

performant. In essence, a `JSON` object is a key-value structure that is fully customizable by definition. The fact that it is a portable, language-independent and browser-compatible data format makes it ideal for storing and exchanging data among applciations. It also very flexible, as it supports both single- and multi-level object structures. In our design, we defined the Ontology as a `JSON Array` that encloses multiple `JSON` objects as entries: each object inside the Ontology corresponds to a real-world entity of the target system. In general, these objects are defined by following a specific structure, which can be, though, aribtrarily extended by the systems integrator, who is the creator of the *Ontology.json* file. There are no nested definitions of Ontology objects, while their inter-dependencies are determined by mathematical relations, described in respective attributes. Hence, the proposed `JSON` formatted Ontology can be regarded as a collection of objects that are abstract in terms of semantics and are organized in a structure that resembles that of a flat *object pool*. Thus, there is no nesting of objects, as all objects of the Ontology are placed at the same level. This design pattern is suitable when dealing with a large number of objects that are repeatedly requested for short periods of time.

For coherence and uniformity reasons, we decided that Ontology objects should have certain characteristics. At first, each object of the Ontology must be uniquely identified. Secondly, these objects must define a notional hierarchical structure that represents the organization of the corresponding real-world entities inside the target system. We presume that this hierarchical structure is expressed by *parent-child* relations that exist among objects. Child objects are *contained in* parent objects. Thirdly, each object must be associated with a specific visual representation(i.e. an image file) so that our platform can manipulate and render it in the presentation layer. The image files that are associated with the Ontology objects are determined by the end-user and can be of arbitrary content, since no conceptual checks are applied by our framework. However, they need to specify certain attributes, as it is thoroughly analyzed in chapter 4. In order to preserve the consistency of the proposed ontological schema, we had to introduce 3 basic integrity constraints regarding the topological structure of the objects forming the Ontology:

– Each object has exactly 1 parent, which is an object that exists in the Ontology

– There is only 1 object inside the Ontology with `null` parent (root object)

  - The resulting graph must be connected, meaning there must exist a path from any node to any other node inside the graph

The above design principles guarantee that the target system is represented by a graph that is both *acyclic* and *connected* - essentially a **Tree** - with a single root node and multiple intermediate or leaf nodes. This `JSON`-formatted, tree-structured Ontology is the cornerstone of OntoMon. When ready, the system integrator uploads the *Ontology.json* file containing all crucial information about the target system to the OntoMon Server, at a well-known endpoint, so that our framework can use it as input.

3.3.2 The Middle Layer

The middle layer hosts the core monitoring and reasoning operations. It contains 4 discrete, but complementary components, each of which is responsible for delivering a specific task. These components are organized in a serial manner. At first, the Icinga monitoring tool collects performance metrics by isolating individual assets of the target system and sends time series data to the Influx storage backend. Then, our custom Observer, which is implemented as a Python daemon, queries the time series database, performs value checking and reports live-status updates to the OntoMon Server, an Express web server.

- **Metrics Collector**
  The first step in order to observe and accurately manage the target system is to carefully set up the monitoring system. This is the fundamental component that applies asset-specific performance checks upon the target system entities and extracts crucial information about them at a given interval. End-users or administrators can define fully customized checks to serve their needs. Undoubtedly, choosing the right metrics collector for our platform was a decision of significant importance. Since OntoMon is designed to manage demanding, large-scale systems, we had to employ a monitoring mechanism that would reduce the configuration overheads and offer the expected functionality. Subsequently, we decided to deploy Icinga, a highly performant monitoring solution for distributed environments consisting of a large number of nodes. We decided to deploy a distributed Icinga cluster, that synchronizes `Objects` and `Zones` in a top→bottom fashion, in order to abstract much of the

configuration complexity, and also to execute checks locally on each node, aiming at distributing monitoring workloads across the cluster. The Icinga clients, running on the specified servers of the target system, are responsible for executing the checks and report the respective real-time metrics back to the Master Zone, as soon as they perceive them. Another important reason for choosing Icinga is its out-of-the-box integration with a wide variety of time series storage backends. Hence, exporting the collected performance data directly to a specialized database of our choice was a matter of installing and configuring the respective `Writer Module` of Icinga. The primary role of the Icinga cluster we deployed is that of the **collector** component, that aggregates and commits times series performance data to a publicly available storage backend.

In terms of communication, the Icinga Master Zone interacts with the time series storage backend (InfluxDB) via a RESTful API, mainly sending the obtained real-time metrics together with useful metadata over the network: as soon as metrics arrive from the Icinga clients, the authoritative Icinga node sends an `HTTP` request to the pre-defined web endpoint that the Influx server daemon is and accepting connections and responding to. The format and the content of the requests are standardized by the native `HTTP` API that the Influx protocol defines. More technical details regarding this communication are discussed in section 4.4 and section 4.5. Our initiative behind setting up an Icinga cluster instead of developing our own monitoring tool was, on the one side, to observe and understand the inner structure and operation of a real-world provisioning tool used in production environments and, on the other side, to employ a tested, distributed and highly customizable collector of real-time data, that by design seeks for performance and scalability.

- **Time-Series Storage Backend**

  Time series (data) is defined as a sequence of data points, typically consisting of measurements that are indexed by equally spaced points in time. Thus, a time series database is a dedicated software system that is designed for optimized storage and handling of time series data. Time series databases enable applications and services to easily scale and support thousands of IoT devices or time series data in a continuous flow. Since OntoMon periodically collects performance metrics and statistics at a user-defined interval, we need an efficient way to store real time series data coming from the target system for long-term usage. Collected metrics should be retained in order to retrieve them in chunks over time, produce corresponding performance graphs and conduct further analysis for trend or pattern identification. The ultimate goal of such database is to effectively store daily, monthly or yearly data, keep it available and support complex time range or filtering queries. Consequently, we decided to set up an Influx server to efficiently undertake storage operations. InfluxDB is based on a distributed architecture, offers high availability and shares the total workload among resources when deployed on multiple hosts. For the scope of this thesis, we argue that a single instance of the Influx server is sufficient to cover our needs, since the low operational complexity of a single-node environment makes it really fast. Nevertheless, just like any other database, InfluxDB, entails some trade-offs aiming at optimizing performance. Its storage engine is built upon the following principles:

- If the exact same data is sent multiple times, no duplicate data is saved
- Since `delete` and `update` operations upon time series data are rare comparing to `write` operations, the functionality of the former is rather restricted
- Most `write` operations regard the most recent data, so time series data is stored in time ascending order
- Under heavy load conditions, consistency might be sacrificed so that multiple clients can be served simultaneously
- Focus on data aggregation and large data sets, rather than individual points

What's more, InfluxDB is easily accessible by clients, as the `influx-server` daemon exports a well-designed `HTTP` API for performing requests upon specific well-known endpoints(`/ping`,`/query`,`/write`. Furthermore, another key feature of InfluxDB that encouraged as to select it as the storage backend of OntoMon, is that despite being a NoSQL database, it supports an SQL-like query language to formalize queries(InfluxQL), facilitating interactions with a rather intuitive and familiar syntax. InfluxDB saves data in a particular format, suggesting *measurements, series, metrics, tags* along with other fields, as it is demonstrated in section 4.5.

- **OntoMon Observer**

  Even though Icinga is a monitoring system that is capable of sending notifications

to the end-user, we preferred to build our own custom Observer that would not depend on the Icinga API. Our initiative was to gain full control over the alerting mechanism, which originates at the middle layer and terminates in the upper layer of OntoMon. The objective of the Observer we designed is to inform the upper layer about the current state of the assets inside the target system so that real-time notifications or alerts can be triggered in the web application. Since the Icinga agents are committed on merely collecting and exporting performance metrics, we needed a dedicated component that would perform reasoning, correlations and, perhaps, more complex analysis over data obtained from the storage backend. Hence, we introduced the Observer, which is a stand-alone component that is implemented as a daemon, is written in `Python` and runs independently from the rest of the system. All core logic, policies and reasoning upon monitoring are implemented in this component. Its primary responsibility is to periodically compare the latest values of the collected metrics with pre-defined thresholds and programmatically publish status updates concerning the individual assets that are currently monitored. Typically, these updates can be considered as instantaneous snapshots of the system being supervised, containing information that determines the notification events in the upper layer. Meanwhile, the system integrator can customize updates to explicitly specify the way that the status change events will be visualized inside the User Interface of OntoMon. In order to facilitate integration with diverse systems, we designed the OntoMon Observer using abstract and dynamic modules.
Upon startup, the Observer daemon requires a list of hosts that belong to the target system and enclose the assets(either software or hardware) that are about to be monitored, along with a corresponding set of check services. Next, the Observer component queries the storage backend about all performance data related to these assets, with a view to obtaining the latest statistics or measurements. The communication between the Observer and the InfluxDB is, again, based on remote `HTTP` requests and is implemented with functions and handlers that are compatible with the `HTTP` API that Influx exports. Ultimately, when the Influx server successfully responds, the Observer performs the following tasks:

1. Digestion and refinement of the received performance data
2. Comparison of metric values against pre-defined thresholds
3. Generation of the update outlining the current state of the asset, based on the

checks performed

4. Deliver the update to upper layer, involving asset state, latest metrics and visualization guidelines

For API uniformity purposes, we concluded that these updates should also be modeled as `JSON` objects, with formalized structure and pre-defined attributes. On the grounds that each `JSON` update object strictly refers to a particular object of the Ontology and, hence to a real-world asset, the Observer component is expected to be aware of the `JSON`-formatted Ontology and, specifically, the mapping of real-world infrastructure and concepts to Ontology objects. Hence, the Observer retrieves the *Ontology.json* file from the OntoMon Server via an `HTTP GET` request and, as soon as the *Update.json* is constructed and validated, the Observer dispatches it inside the body of an `HTTP POST` request to a well-known endpoint that the OntoMon Server is managing. In this way, a custom status notification API is defined, that is fully compatible with the given ontological description of the target system.

- **OntoMon Server**

  The OntoMon Server is a central component of our platform, that interacts with multiple entities and distributes data across the architectural layers. In essence, it is a very fast and lightweight web server(script) that is entirely implemented in `JavaScript`. The OntoMon Server bridges the gap between components, as it takes over the handling of `HTTP` requests and the sharing of static files related to our system. It can be thought as the *glue* component, that provides services to various other entities and knows the current state of the target system at any time. It concurrently listens to multiple web endpoints, accepts remote connections and supports real-time communication with various clients. As far as the communication with the base layer is concerned, the OntoMon Server exports a well-known endpoint to the integrator of the system, so that the required files(*Ontology.json, .svg files* can be uploaded via simple `HTTP POST` requests. As soon as these files are successfully uploaded, the server of OntoMon stores them locally and delivers them in another publicly available endpoint. It is notable, that even though these files are necessary for the startup of our system, they might be updated later, even during runtime. By default, the OntoMon Server is designed to always keep the latest versions of files uploaded with the same name, by overwriting content. Legacy files can be also kept in history with proper configuration of the server.

All the components of our platform that need these files for their operations, can instantly retrieve them by sending an `HTTP GET` request to the specified endpoint. For example, both the Observer component, as well as the User Interface component require the *Ontology.json* file to properly function, so they inevitably have to issue requests to the OntoMon Server every time they run. In particular, the Web UI of OntoMon also needs to be aware of the *Update.json* and the `.svg` files. Last but not least, the OntoMon Server is responsible for accepting `HTTP POST` requests from the Observer daemon, containing `JSON`-formatted updates about the current state and the visualization details regarding assets in their body. Our server parses the *Update.json* objects it receives, serializes them in a queue and serves them in a third, well-known endpoint. Besides, it tracks down these update requests and generates concise, human-readable logs to retain a timeline of the target system behavior.

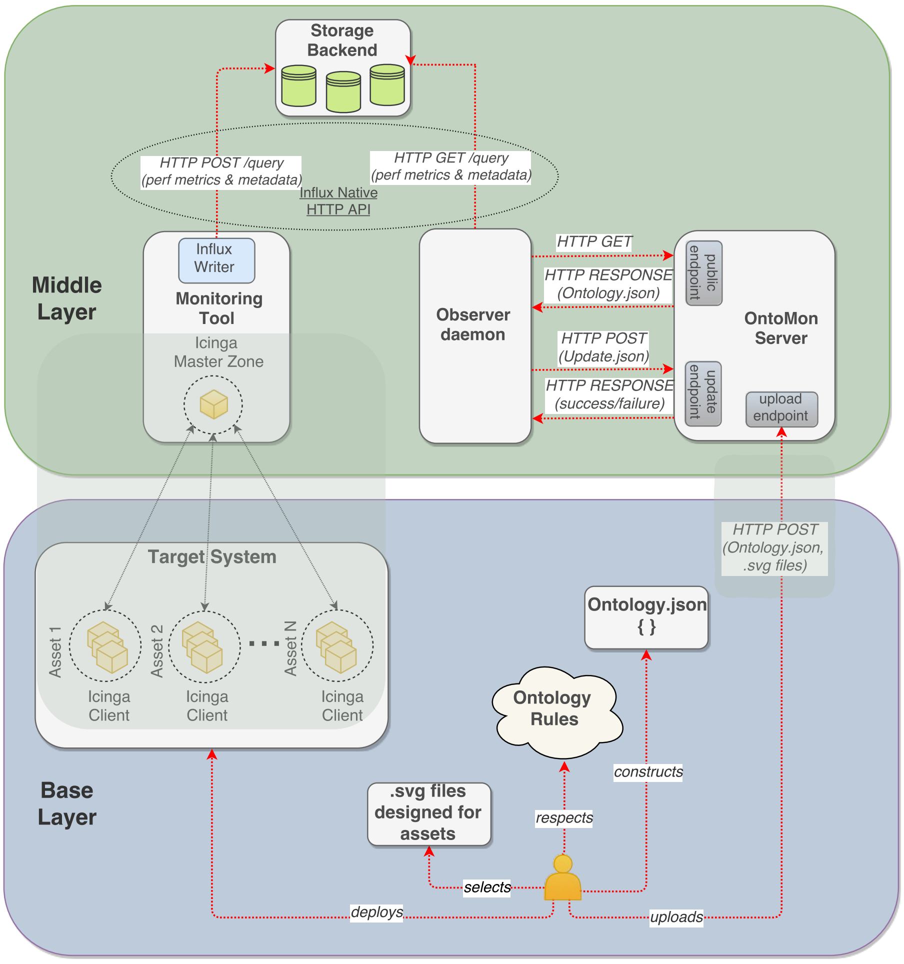


Figure 3.2: *Communication between Base and Middle Layers*

3.3.3 The Upper Layer

The upper layer of our platform implements the visualization of the target system and the presentation of the monitoring metrics. It accommodates higher-level components that co-operate with the underlying layers in order to offer an "all-inclusive" overview of the target system to the end-user. This layer is of vital importance, as it provides direct interaction with the end-user. More precisely, it includes the OntoMon Web Interface, a web application that aims at visualizing the current state of the monitored target system. In addition, it includes the Graph Composer, a dedicated component that produces interactive time-series data charts.

- **OntoMon Web Interface**

  The Web UI of OntoMon is the fundamental component of the upper layer, taking advantage of all the processing performed in the lower layers. It is the output of our platform, offering supervision and management capabilities to the end-user. The Web User Interface component is a flexible web application that is built upon the latest version of the widely popular Angular framework, in order to familiarize with a production-ready web development framework and benefit from the cutting-edge capabilities it offers. The main reason for designing a web UI for our platform in the first place, was to enhance end-user experience in terms of asset management, guarantee remote access via the Web and benefit from modern Web APIs and standards. As stated previously, we focused our attention on developing a **content-agnostic**, fully dynamic user interface, that renders objects directly to the browser `DOM`, solely based on their pre-defined attributes stated in the Ontology. The Angular web application is the top-level component of our platform and is fully compatible with all inner APIs of OntoMon. The visualization of the monitored infrastructure involves the programmatic control and manipulation of the `SVG` files that correspond to the assets of the target system. More specifically, `SVGs` are usually transformed or adjusted, in order to be organized in a nested structure, but no front-end action depends on typesetting the asset they represent, since they are abstract by convention. Prerequisite for the proper visualization of the object hierarchy, is the acquisition of both the *Ontology.json* and the *.svg* files from the OntoMon Server, via targeted `HTTP GET` requests.

  Apart from the visualization of the target system, the Web User Interface retrieves and aggregates performance metrics, status updates, metadata and logs from the middle layer. In the proposed design, the Angular application injects a polling service that periodically dispatches `HTTP` requests to the OntoMon Server to obtain the latest updates and, thus, actively sample the current status and performance data of each monitored asset. The OntoMon Web Interface parses the *Update.json* objects and produces customized real-time notifications and alerts, based on the specified guidelines. In this way, resources can be efficiently isolated and analyzed in-depth. Based on the `JSON`-formatted status updates, our UI delivers solid indications of asset-related issues that might occur to the end-user, along with a comprehensive presentation of the related measurements. We provide different views of the same performance data, such as tables and graphs, that are automatically updated as new time-series data arrives from the underlying system. In respect to the generation of real-time graphs, we decided to delegate this task to a dedicated platform, aiming at a separation of concerns and distribution of load. Therefore, the Web UI is not burdened with the computationally intensive operation of building graphs; instead, it is able to embed pre-built, dynamic and easily pluggable graphs into its views.

- **Graph Composer**

  The primary objective of building a monitoring web interface is to offer a compact, yet analytical overview of the underlying infrastructure and highlight the core aspects of the monitored infrastructure. The most instinctive and comprehensive way of demonstrating performance, utilization of resources or any asset-related statistics, is to represent real-time data into graphs or numerical language. All modern monitoring dashboards ship with powerful graphical representations of collected metrics, so that administrators can effortlessly gain meaningful insight of the monitored system, at a glance. We argue that this is a well-engineered approach that allows for fast issue detection, historical analysis of performance and discovery of patterns. Thereafter, it facilitates asset management and restriction of attention on individual components. In general, graphs and dashboards are designed to adapt to end-user needs, so that the enclosed information is presented in the most intuitive and comprehensible way. The goal is to expose interactions between series and correlate seemingly unrelated data. In this way, issues can be effectively handled as soon as they appear.

  Considering the above, in order to maximize the understanding of the target system and make high-quality decisions about the management and provisioning of the infrastructure, we decided to introduce a dedicated component that takes care of the visualization of the performance metrics in graphs. The respective technology we employed is Grafana, an open-source data visualization platform written in `Go` and `JavaScript`, that is optimized for presenting, querying and examining time series. It supports a wide variety of data analytics operations, such as mathematical functions, filtering, aggregation etc. Grafana produces detailed and interactive graphs that are easy to follow and provide a concise description of the specified asset performance. One of its key features is the **client-side** rendering of graphs, alleviating server nodes from heavy processing workloads. Thus, the client only receives the bare bone `HTML` code, along with a `JavaScript` file that ultimately loads the rest of the content dynamically. Based on these graphs, end-users not only obtain a holistic and intuitive view of the target system behavior in a certain time window, but are also empowered to conduct an automated inspection of future anomalies. To our knowledge, this mechanism is one of the most powerful features that the user interface of a monitoring tool can offer, as statistical models can be defined aiming at predicting asset availability or equipment replacement times. What's more, Grafana offers a built-in RBA [2] that allows the specification of users, groups and permissions upon dashboards.

Grafana is compatible with multiple storage backends, such as Graphite, TSDB, Prometheus and InfluxDB as input sources. Hence, wiring the proposed Graph Composer to the already deployed system was a trivial task. We set up the `grafana-server` and configured it to perform queries about time series data on the existing InfluxDB storage backend of the middle layer, via its rich `HTTP` API and native query language. This topology has the benefit that the collected performance data is not transmitted back and forth between the Web User Interface and the Influx server; instead, the Grafana daemon queries the InfluxDB for time series in a specified time frame. As data arrives at a given interval, Grafana initially composes, and then updates the corresponding graphs, while serving them locally at a pre-defined endpoint. In order to import these graphs into the User Interface of OntoMon, we implemented a dedicated service inside the Angular application, that is responsible for fetching and embedding real-time performance graphs as raw `HTML iframes`,

[2]Role Based Administration

fetching them from the well-known URL that Grafana exports them. Grafana produces a separate dashboard for each monitored asset, comprising of multiple standalone panels. Since these dashboards are asset-specific, we configured the URL that Grafana exports to contain an `asset` parameter, in order to designate the desired asset. This parameter is mapped to a template variable internally handled by Grafana.

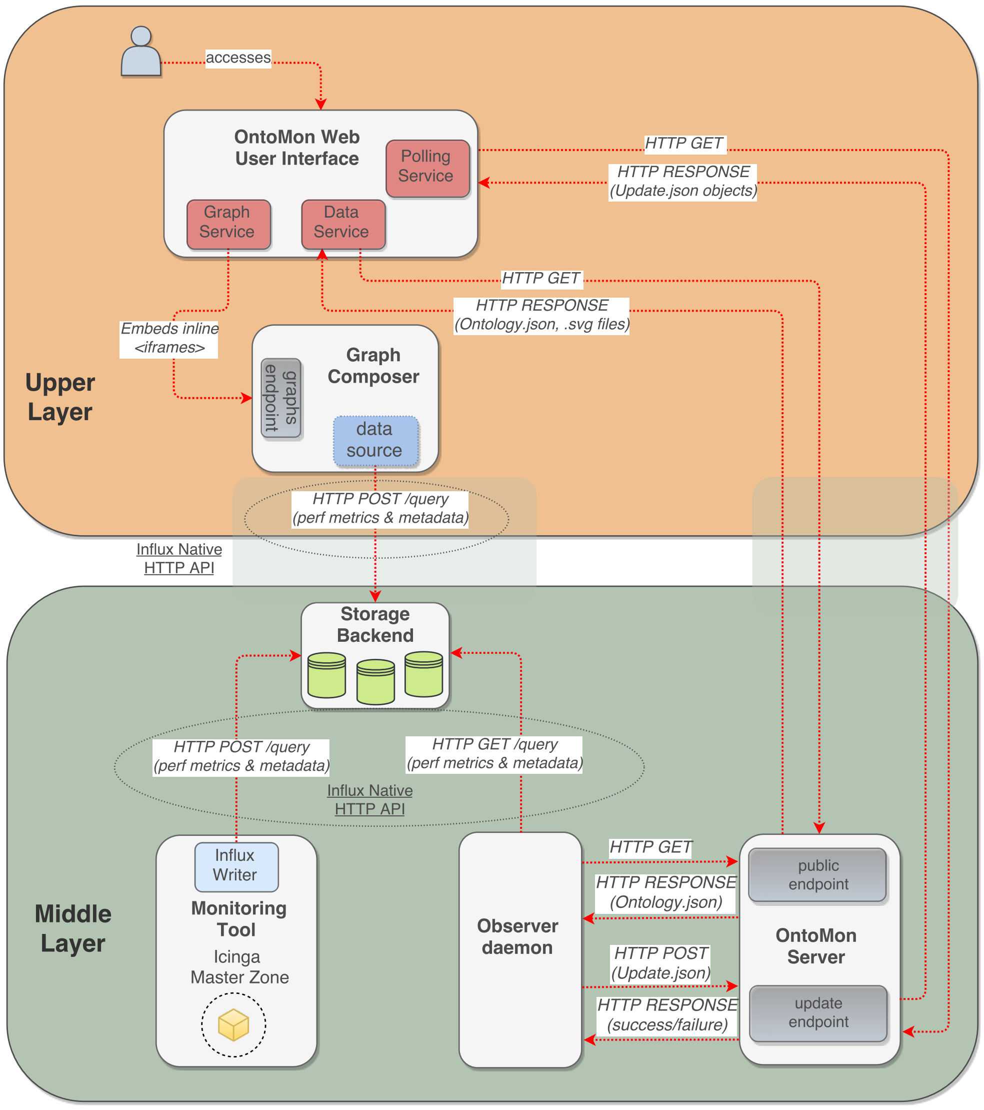


Figure 3.3: *Communication between Middle and Upper Layers*

4 Implementation

In this chapter, we methodically present the building of OntoMon. After demonstrating the theoretical background and the architectural design of our framework, we allow the reader to follow the steps we took during the development process, grasp the logic behind our code and observe how we overcame the difficulties we faced. In each iteration, we isolate a specific component of OntoMon platform, refer to the technologies and algorithms we used and illustrate the key points of the proposed implementation.

4.1 Experimental Target System

As already mentioned, OntoMon is a monitoring platform that sits on top of heterogeneous target systems. However, at some depth, we assume that every target system accommodates **computer machines** that are either treated as hardware or software entities. Based on this premise, we had to set up a testbed environment that would act as a candidate target system for our platform and serve for both hardware- and software-oriented case approaches. Therefore, we decided to deploy a small computer cluster, consisting of 3 virtual machines that are interconnected with each other. For this purpose, we decided to use Qemu [1], a free open-source machine emulator and virtualizer, that facilitated the deployment of our cluster. The Qemu virtualization engine enabled us to perform a full-stack configuration of the VMs and make any needed adjustments in their settings. At first, all 3 virtual machines were attached on the same physical host

[1]`http://www.qemu.org/`

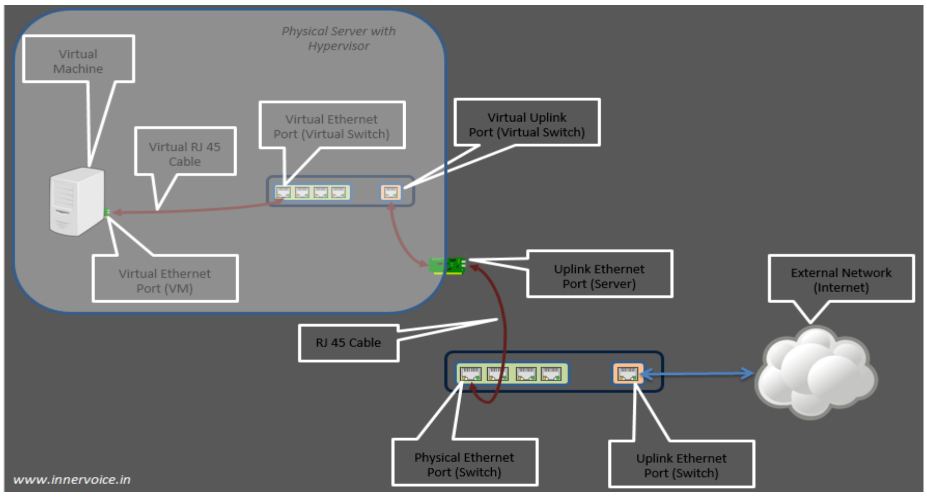
system, running `Ubuntu 16.04.2`, an operating system that has virtualization capabilities and integrates well with Qemu. For uniformity reasons, we decided to install the same guest operating system, `Ubuntu Server 16.04.2`, on all nodes across the testing cluster.

Private Network configuration

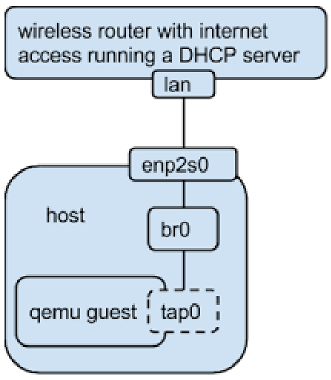
Bearing in mind that this cluster resembles a real-world distributed computing system, we had to ensure network connectivity both among the nodes(internal operations and protocols) and to the Internet(installation of software packages, utility services, remote access). Therefore, we decided to set up a fully-customized networking layer, employing network interfaces provided by Linux and Qemu. Our aim was to build a private network behind `NAT`, that could access the outer world as a usual home environment. At first, we created a bridge interface [2] (`br0`) on the hypervisor system and assigned it with a static IP address(`10.0.0.254`). A Linux bridge is really a *virtual switch* implemented inside the kernel, that carries incoming and outcoming traffic. Next, we extended the configuration of the `dnsmasq` [3] running on the hypervisor, a lightweight service providing network infrastructure for small networks. Primarily, we modified the configuration file of the `dnsmasq` service. We set the default *Router* and *DHCP* server options on `10.0.0.254` and also determined the `dchp-range` of the private network to be `10.0.0.0/24`, explicitly mapping physical MAC addresses to IP addresses, ensuring uniqueness. In this way, each server receives an IP address that is bound to the MAC address of its (virtual) network card. At the same time, all network traffic coming from the virtual machines will be routed to the `br0` interface of the hypervisor, that acts as the gateway of the private network.
However, in order to forward network packets from the virtual to the physical world, we had to set up a mechanism to redirect all network traffic to the hypervisor: tap interfaces. In short, tap interfaces are created by Qemu processes and are software entities that act as *virtual ports* that are hooked on a Linux bridge, just like ethernet ports on a real physical switch. They exist only in `kernelspace` and their goal is to provide an endpoint for the emulated NICs [4] of the VMs, so that a connection can be established to forward ethernet frames. Each virtual server needs a single tap interface

[2] `http://man7.org/linux/man-pages/man8/bridge.8.html`
[3] `http://www.thekelleys.org.uk/dnsmasq/doc.html`
[4] Network Interface Cards

as network backend. Thus, we configured Qemu through its CLI options to create a new tap interface during the bootup of the VMs, along with a `qemu-if-up` script that automatically attaches the newly created virtual ports to the `br0` interface of the hypervisor. Spinning up n VMs for our cluster results in a n to 1 port matching inside the bridge interface of the hypervisor.


Physical Server with Hypervisor
Virtual Machine
Virtual RJ 45 Cable
Virtual Ethernet Port (Virtual Switch)
Virtual Uplink Port (Virtual Switch)
Virtual Ethernet Port (VM)
Uplink Ethernet Port (Server)
External Network (Internet)
RJ 45 Cable
Physical Ethernet Port (Switch)
Uplink Ethernet Port (Switch)
www.innervoice.in

(a) *vNET topology*



(b) *vNet diagram*

Figure 4.1: *Virtual Bridge Networking*

The last step was to proceed with the configuration of the host system and the bridge interface. At first, since we are setting up a Linux router for our private virtual network, we had to enable `IPv4` Packet Forwarding on the hypervisor, as in most modern Linux distributions this feature is disabled by default. Subsequently, we edited the `/etc/network/interfaces` file inside the host system, in order to permanently provide details about the definition of the `br0` interface, such as static IP address, Netmask and others. Finally, we determined some Firewall rules using the `iptables` [5] service: our initiative was to redirect traffic coming from the virtual private `10.0.0.0/24` network to the physical `enp1s0` ethernet interface of the host, so that it could be dealt with as usual host traffic. Precisely, we configured `NAT` [6] Masquerading upon `enp1s0`, which is a method of hiding or remapping one IP address space(usually private) into another(usually public), by modifying the source and destination fields inside the header of the IP packets. The topology discussed above is known as *bridged networking using NAT*. The most notable points in the configuration of the private network for our cluster are provided below:

```
$ cat /etc/NetworkManager/dnsmasq.d/cluster.conf
interface=br0
bind-interfaces
dhcp-range=10.0.0.0,static
dhcp-host=DE:AD:BE:EF:14:AC,10.0.0.1
dhcp-host=DE:AD:BE:EF:CD:CA,10.0.0.2
dhcp-host=DE:AD:BE:EF:16:7E,10.0.0.3
dhcp-option=option:router,10.0.0.254
dhcp-option=6,10.0.0.254
```

Listing 4.1: *dnsmasq configuration*

```
# bridge setup
$ brctl addbr br0
# show bridge while VMs are running
$ brctl show
bridge name  bridge id     STP enabled  interfaces
br0          8000.526a92e3be49 no       tap0
                                        tap1
                                        tap2
# enable and verify IP packet forwarding
$ echo 1 > /proc/sys/net/ipv4/ip_forward
$ cat /proc/sys/net/ipv4/ip_forward
```

[5] `https://linux.die.net/man/8/iptables`
[6] Network Adress Translation

```
1
# SNAT to forward traffic to hypervisor ethernet interface
iptables -t nat -A POSTROUTING -o eth0 -s 10.0.0.0/24 -j MASQUERADE
```

Listing 4.2: *IP forwarding on hypervisor*

```
# /etc/network/interfaces
auto br0
iface br0 inet static
    bridge_ports none
    bridge_stp off
    bridge_maxwait 0
    address 10.0.0.254
    netmask 255.255.255.0
```

Listing 4.3: *Bridge definition*

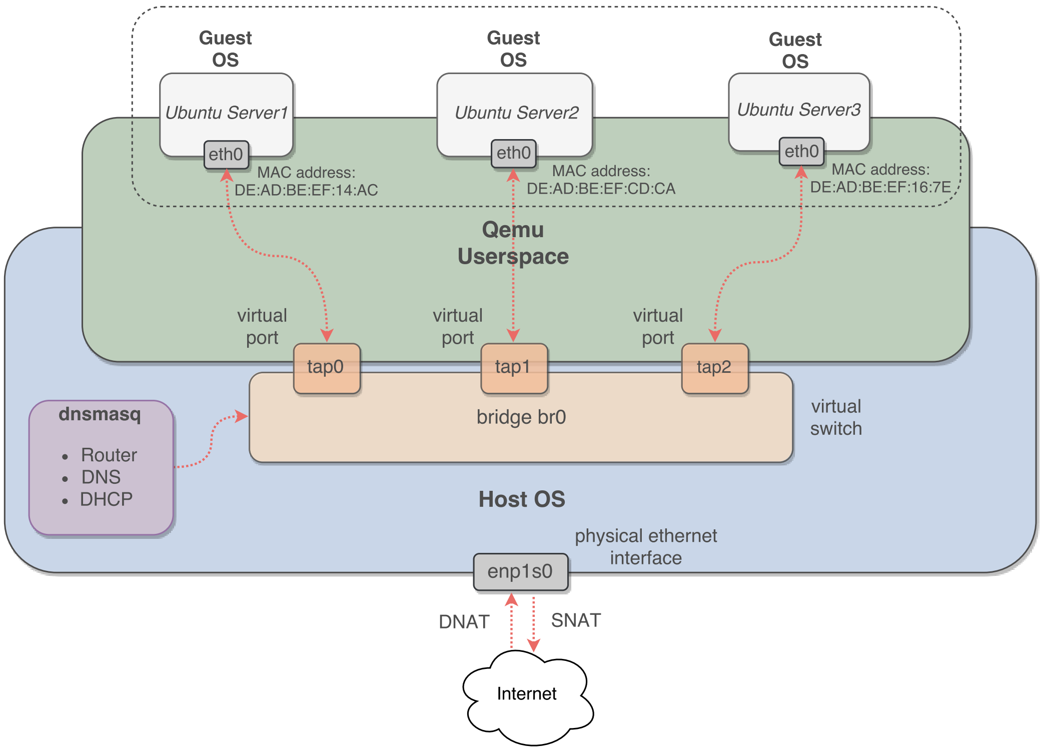


Figure 4.2: *Cluster Topology*

4.2 Ontological Schema Definition

In this section, we demonstrate how we formalized the Ontology component and provide an extensive description of the proposed *ontological schema*. As already stated, one of our primary design decisions was to formulate the ontological description of the target domain as a collection of `JSON` objects. Every object belonging to the Ontology must follow a specific structure, in order to facilitate parsing and validation. We needed an abstract, in terms of content, yet formal and apprehensible object model that would be free of superfluous information, reflecting the design principles described in subsection 3.3.1. Therefore, we argue that the proposed object attributes meet the aforementioned requirements, as presented below:

```json
{
  "uuid": "c8bd7185-6349-4df5-8628-f87115222987",
  "name": "Ubuntu-Server-1",
  "label": "Server",
  "file": "Server.svg",
  "parent": "25e2ce69-2445-4b28-9a71-e7ca01bc57ea",
  "info": {
     "description": "Server asset"
  }
}
```

Listing 4.4: *Example of JSON Ontology object*

- **UUID**
  In order to guarantee that the each `JSON` object inside the Ontology can be uniquely referenced and identified, we employed the Universally Unique Identifier standard. This is a 128-bit number, expressed in 32 hexadecimal digits, separated by hyphens in form $8-4-4-4-8$ that is used to identify information about components of computer systems. Their main advantage lies in the fact that there is no need for consulting a central authority to assign individual unique identifiers to our objects; instead `UUIDs` can be easily produced by pre-defined mathematical functions. By introducing a `uuid` attribute into

our schema, we do not have to rely on user-defined names or labels for the registration and management of the Ontology entities. In this way, we effectively avoid object collisions and contradictions, since `UUIDs` promise to minimize the probability of producing duplicate identifiers, even in large scale. Another important thing to note is that we entirely abstract the generation of `UUIDs` for objects from the system integrators, as it is a procedure that does not concern the conceptual design of the Ontology. Instead, we internally manipulate the input Ontology and create mappings between object names and `UUIDs`.

- **Name**
  This attribute is a `string` specified by the integrator-user and is, essentially, a conventional name assigned to each object defined in the Ontology. While entirely arbitrary in terms of context, it is expected to refer to the real-world entity of the target system it represents. The value of this attribute is parsed by the OntoMon Web Interface component and is used for adding additional information during the visualization process.

- **Label**
  Since the User Interface of OntoMon is content-agnostic, we needed a generic property based on which we could dynamically aggregate and group objects inside UI elements of the Angular application(e.g. menus, search fields, etc). Again, the value of the `label` attribute is a `string` of arbitrary context. However, we consider as best practice that assets of the same type in the real-world system are given the same label value, so that the categorization inside the User Interface is solid and meaningful. The `label` attribute must not be confused with objects types, models or classes, as no label-specific action is taken upon objects by the monitoring or visualization layers. In other words, the `label` attribute does not add any semantics to the object.

- **Parent**
  The `parent` attribute, as its name suggests, is an attribute that implements our principle of building a hierarchical structure among the objects of the Ontology. The `parent` attribute is also a `string` field that points to a `UUID` assigned to an existing object of the `JSON` array. By following the `parent` attribute of each object, we gradually construct the individual levels of the Ontology tree

and observe the complexity of the target system. Only a single object inside the Ontology is allowed to have `null` as `parent` and it is the *root object*. Essentially, the `parent` attribute is a foundational property of the ontological objects, as it encloses the mathematical formula that interconnects Ontology entities. Unlike the "is ancestor of" relation, the "has parent" relation is not transitive and, of course, not symmetric.

- **File**
  One of the primary objectives of OntoMon is to empower end-users to fully customize the visualization of their system. In this direction, we had to figure out an efficient way of mapping the objects inside the Ontology to optical elements that will appear in the various views of the User Interface. Accordingly, we introduced the `file` attribute, which is a `string` holding the image file that will be associated with the respective object. Since the visualization layer of our platform performs low-level operations upon image files, we require that end-users exclusively specify and upload `.svg` files for each defined object, so that transformations can be applied without sacrificing quality or detail, depending on the display.
  Furthermore, in order to facilitate the visualization of nested `SVG` elements inside the `DOM` of the browser, we needed a conceivable, yet operable way of depicting the parental relations that exist among objects. In practice, we had to come up with a mechanism that efficiently expresses the capability of objects to contain others. Our initial approach was to introduce **anchor points** inside the `SVGs`: parent objects would have `sockets` at certain X-Y coordinates, while children objects would have `plugs` at their central point. By convention, plug anchors fit inside sockets anchors so a relative placement among `SVGs` would be feasible. Soon enough, we realized that this approach was rather restrictive for our application, as it does not allow full control over transformed objects at runtime and has a quite rambling implementation. Therefore, we modeled the "parent-child" in a similar, but more elegant and flexible way. Instead of determining anchor points, we proposed the concept of **`slots`**. Typically, `slots` can be regarded as conceptual instating areas of fixed, pre-defined dimensions. They are implemented as a custom element attribute, that is directly encapsulated into the `XML/SVG` code. More precisely, its value is a `string` consisting of 4 comma

separated values, in the following form:

"*<upper-left-point1-x>, <upper-left-point1-y>, <width>, <height>, <upper-left-point2-x>, <upper-left-point2-y>, <width>, <height>, ...*"

As expected, the `SVG` files corresponding to parent objects in the Ontology must include a number of different slots in their code, that is equal to the number of their children. An `SVG` element with n slots can accommodate up to n child `SVG` elements. By convention, objects that are placed at the leaves of the ontology tree must set the empty string as the value of their `slots` attribute, indicating they contain no objects. The combination of `slots` with the `parent` attribute of the Ontology objects, is the base of our visualization engine. The main benefit of working with `slots` is that end-users of `SVG` file designers can accurately determine the exact position and size at which every asset of the target system will be rendered. Such functionality expands customization capabilities and provides an intuitive overview of the hierarchical relations among real-world assets. Finally, since the OntoMon UI performs asynchronously gets informed about status updates of assets, we searched for an elegant way of depicting these status updates upon `SVGs`. Our proposed solution involves the definition of an element specified by the system integrator, that is classified as `indicator`, inside the body of the `SVG` file. In this manner, status updates can be reflected as visual animations, shape updates, color variations, etc upon this indicator element, showcasing that an issue has occurred regarding the respective asset. It is also possible to define multiple indicators with different visual notifications.

```
<svg xmlns="http://www.w3.org/2000/svg" width="100%" height="100%"
    slots="15,25,200,50,45,75,240,60">
    <g>
            <!-- SVG markup here -->
            <circle>
                <animate class="indicator" attributeName="fill"
                  values="#62c36e;#36a242;#62c36e"/>
            </circle>
    </g>
</svg>
```

Listing 4.5: *SVG element with 2 slots*

- **Info**
  The `info` attribute is an optional, and rather complementary, attribute for our schema. It is deliberately defined so that end-users together with integrators can fill in additional information or details about the assets of the target system, such as hardware specifications or custom configurations. Ordinarily, the `info` attribute is itself a `JSON` object that comprises of arbitrary fields and values. The values of these fields are presented in the OntoMon User Interface component, when the corresponding asset is viewed by the end-user. Combining specific asset options or details with real-time graphs and metrics can prevent performance degradation, leading to a fine-tuned and well-operating system.

4.3 Development of the OntoMon Server

Developing a web server is, in general, a challenging and complicated task. However, our platform designates quite specific tasks and requirements from its internal server. Hence, in order to accelerate the development process and slide over a lot of boilerplate code, we settled for building a robust web server on top of the `Node.js` platform, a powerful `JavaScript` execution engine written in `C`. This platform does not follow the traditional *Request/Response* multi-threaded model; instead, it is built upon an event-driven approach that relies on the "Event Loop" Component, a single-threaded process that instantly processes non-blocking requests coming from clients. Only blocking requests are assigned to separate threads so that resource utilization is reduced and more concurrent requests can be handled by the server.

Mainly, we decided to utilize the fast and mature Express [29] application framework, that allows for writing scalable, yet simplified server-side code in pure `JavaScript`. The development process is rather quick and dry, while the Express web application runs as a standalone web server that abstracts concerns and is not dependent on Apache or Nginx. In addition, Express supports numerous built-in `HTTP` utility methods and middleware, facilitating the construction of solid RESTful APIs. Furthermore, there

is a great variety of high-level libraries available, that are designed to plug into Express and implement basic concepts like routing, serving of static files, authentication, data encryption, database integration and others. Thereafter, this state-of-the-art software stack allowed us to easily spin up a reliable and performant web server, without having to concern about complex low-level networking concepts.
OntoMon Server is a central component, that is responsible for orchestrating communication and exchange of data across our platform. It is publicly available and exposes a set of well-known `HTTP` API endpoints. More specifically, we configured the OntoMon Server to run on the hypervisor node, providing the various web services and resources to respective clients, as shown in Table 4.1:

Web Service	**Endpoint**	**Client**
Upload .json and .svg files	`/upload`	End-User
Serve static files	`/resources`	OntoMon WebUI, OntoMon Observer
Handle status update requests	`/updates`	OntoMon WebUI, OntoMon Observer
Generate and serve log file	`/history`	OntoMon WebUI

Table 4.1: *OntoMon Server API*

As shown in the table above, our web server listens and responds to specific endpoints. At first, it acts as a storage repository that supports file uploading via simple `HTTP POST` requests at `ontomonHost:8080/upload`. It is notable, that these files are persistently stored in the local memory of the OntoMon Server host and are not disposed after the termination of every application session, as end-users are likely to reuse the same *Ontology.json* and *.svg* multiple times. Thus, there is no need to re-upload identical data in order to produce the same OntoMon dashboard. In this way, files are always available for remote access during the runtime of our system, by sending `HTTP GET` requests to `ontomonHost:8080/resources`, either for `SVGs` or `Ontology.json`.

Besides, the OntoMon Server has a quite active role in the alerting mechanism of our framework. More specifically, we set up a web service at `ontomonHost:8080/updates`, that accepts `HTTP POST` requests of `JSON`-formatted asset status updates and also responds to `HTTP GET` requests with the latest status updates available. This service retains a time-sorted queue of *Update.json* objects, in the exact same form they were pushed by the OntoMon Observer. When an *Update.json* object arrives at the `/update` endpoint, the OntoServer pushes the object to the queue and then adds a corre-

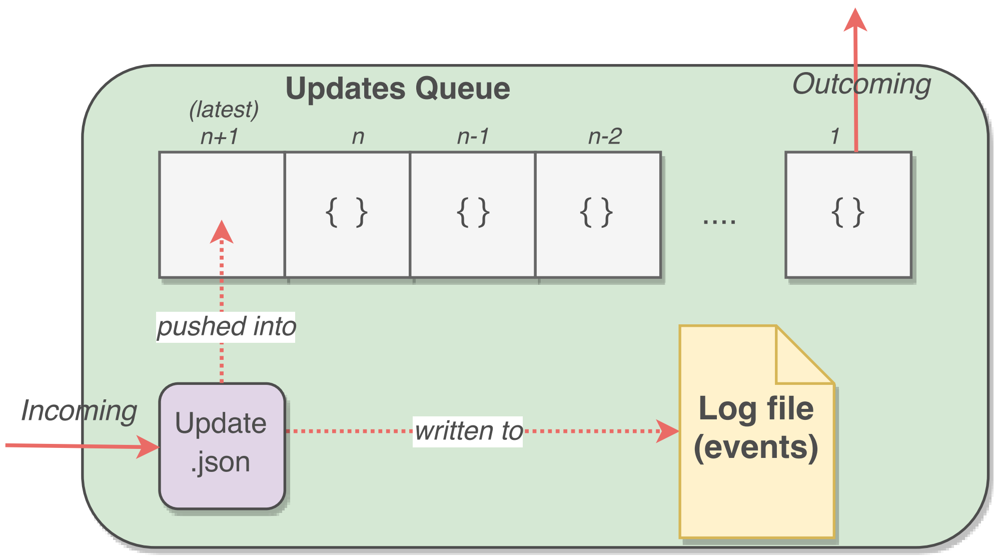


Figure 4.3: *OntoMon Server: Status Update mechanism*

sponding entry to the generated *Update.log* file, keeping track of status update events in chronological order. The *Update.log* file is statically served at `ontomonHost:8000/history`, so it can later be accessed by the Angular application.
Obviously, the OntoMon Server is a standalone component that is decoupled from the rest of the system and can be deployed on a remote node, as long as it is reachable via the network. A simplified version of the OntoMon Server implementation is provided below:

```
// create the Express.js application
var app = express()
// accept POST requests to upload files
app.post('/upload', upload.single('filename'), function(request,
    response, next){
   response.setHeader('Content-Type', 'text/plain');
   response.write('Upload was successful.');
   response.status(204);
   response.end();
});
// accept GET requests for asset status updates
app.get('/updates', function(request, response){
    response.setHeader('Content-Type', 'application/json');
    if (queue.length > 0) {
        var obj = queue.shift();
    } else {
        var obj = {};
    }
    response.json(obj);
    response.status(200);
    response.end();
});
// accept POST requests to update asset status
app.post('/updates', function(request, response){
    handleUpdatePOSTRequest(request);
    response.setHeader('Content-Type', 'text/plain');
    response.write('POST request was successful.');
    response.status(204);
    response.end();
});

function handleUpdatePOSTRequest(req){
    var filePath = __dirname + '/log/update.log';
    var uuid = req.body['uuid'], name = req.body['name'];
    var timestamp = req.body['timestamp'],state = req.body['state'];
    var type = req.body['type'], metrics = req.body['metrics'];
    var description = req.body['description'];
    var log_string = '[' + timestamp + ']: ' + name + ' - ' + state
        + ' - ' + description + '\n';

    // push to queue and log file
    log.push(req.body);
    queue.push(req.body);
    fs.appendFile(filePath, log_string, function (err) {
        console.log(err);
    });
    return;
```

```
}
```

Listing 4.6: *OntoMon Server: file upload and Updates queue*

4.4 Metrics Collection with Icinga

The setup and the configuration of the Icinga monitoring tool required our close attention, so that the data collection mechanism would be efficient and sustainable in large scale environments and reflect administrator needs.

- **Icinga Cluster Deployment**

  We began with the installation of the Icinga software packages on all cluster nodes that were about to be monitored. Since two different types of Icinga daemons had to be installed on individual nodes, we executed the `Icinga node wizard` utility command on every node, in order to automate the installation process. In OntoMon, we worked with the latest Icinga 2.6.3 Release:

```
root@ontomonHost:~# wget -O -
    https://packages.icinga.com/icinga.key | apt-key add -
root@ontomonHost:~# echo 'deb https://packages.icinga.com/ubuntu
    icinga-xenial main' >/etc/apt/sources.list.d/icinga.list
root@ontomonHost:~# apt-get update && apt-get install icinga2
```

Listing 4.7: *Installation of Icinga packages*

```
root@ontomonHost:~# icinga2 node wizard
Welcome to the Icinga 2 Setup Wizard!
Please specify if this is a satellite setup ('n' installs a
    master setup) [Y/n]:
```

Listing 4.8: *Running icinga node wizard on master node*

At first, we set up the Icinga Master at the hypervisor node, by specifying its FQDN [7] and accepting the default settings regarding the Icinga API(e.g. port). During the

[7]Full Qualified Domain Name

installation process, configuration files and directories were generated on the Master node, along with a CA certificate, a private key and a CSR for authentication purposes. Afterwards, we set up the Icinga Clients at `10.0.0.1`, `10.0.0.2` and `10.0.0.3`, respectively, following a similar procedure. Due to the fact that every Icinga Client is accounting to some Icinga Master, we specified the FQDN of the corresponding Master node when prompted by the `icinga node wizard`, so as to carry out the hierarchical structure. It is also worth mentioning, that the registration of a Client to some Master is secured by ticket certification: the Icinga Master generates a client-specific ticket that is based on its FQDN, distributes it to the concerned Icinga Client who then submits it back to the Master. Once identified, the client subscription is considered successful and the Icinga cluster is functional, comprising of nodes that "talk" the same API and have built trust relations among them.

- **Icinga Cluster Configuration**
  As already mentioned in subsection 3.3.2, our goal is to configure the Icinga monitoring platform in a distributed **Top→Down** manner. This configuration mode is decidedly more convenient and robust for multi-node environments, that enclose a large number of hosts which need to be monitored. Even though the following setup was designed for the target system described in section 4.1, its logic and principles certainly provide a solid basis for configuring monitoring in more complex topologies. Bearing in mind the concept of Icinga `Zones`, we managed to build a hierarchical structure by determining a single authoritative Master Zone(`127.0.0.1`) and three sibling -but discrete- Client Zones(`10.0.0.1`,`10.0.0.2` and `10.0.0.3`), that pertain to the Master Zone. Thus, the Icinga tree consists of 1 root and 2 leaf nodes. We argue that the discussed classification of our cluster nodes greatly simplifies the monitoring process, is comprehensible and explicitly separates concerns. Its most powerful feature, thereafter, lies in the fact that we had to define the configuration objects only once inside the Master Zone, while synchronization takes place automatically by pushing down the configuration to thousands of Clients in a distributed fashion. Replication of configuration objects requires network connectivity and trusted communication between Master and Client, as preconditions. Indisputably, this hierarchical design fits extremely well in clusters that comprise of a large number of client nodes which need to be monitored on the same physical or

software assets. The overview of our testing topology is portrayed in Figure 4.4:

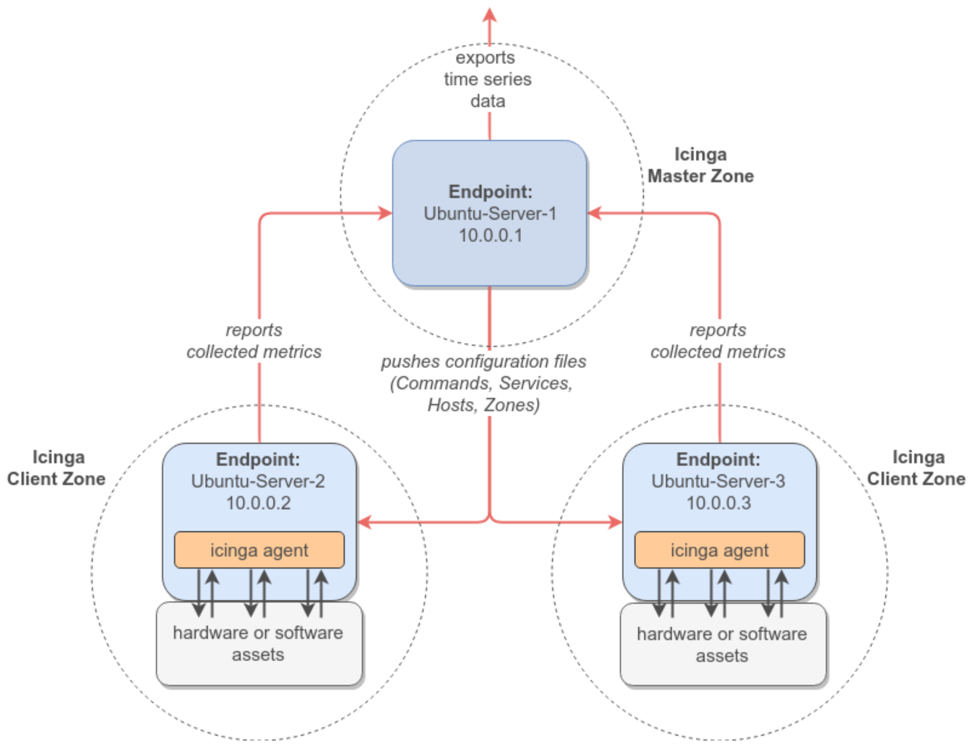


Figure 4.4: *Icinga Monitoring Cluster*

In order to regulate the monitoring process, we proceeded with the definition of various Icinga configuration objects, such as `Hosts,Services` and `CheckCommands`. As administrators, we do not have to define any configuration objects on the client nodes; instead, these nodes only receive the configuration objects coming from the authoritative Master and store them locally. Therefore, it is sufficient to provide an indicative presentation of the proposed configuration of the Master node. To begin with, all configuration files are stored under `/etc/icinga2` directory. Zone and Endpoint objects, along with their hierarchy, are defined in the `zones.conf` file. In the **Top →Bottom** configuration mode, the `/zones.d` directory recursively includes configuration folders that are named after `Zone`. These folders contain declaration files(e.g. `hosts.conf,services.conf`), in which Icinga configuration objects, such as Hosts and Services, are denoted. Each Icinga `Zone` is aware only of the objects defined in its respective folder. In this manner, nodes belonging to the same `Zone` receive the exact same configuration. However, the `/zones.d` directory might also contain a special configuration folder named `/global-templates`. Every Icinga object that is defined inside this folder is globally available in all `Zones`, including the Master one. We found this auxiliary feature extremely useful, as it provides and automated way of synchronizing `Templates, Services, Check-Commands` and other objects across irrespective `Zones` of the cluster.

```
object Endpoint "icinga-master-1" {
   host = "10.0.0.1"
}
object Endpoint "icinga-client-1" {
   host = "10.0.0.2"
}
object Endpoint "icinga-client-2" {
   host = "10.0.0.3"
}
object Zone "icinga2-master1" {
   endpoints = [ "icinga-master-1" ]
}
object Zone "icinga2-client1" {
   endpoints = [ "icinga-client-1" ]
   parent = "icinga-master1"
}
object Zone "icinga2-client2" {
   endpoints = [ "icinga-client-2" ]
   parent = "icinga-master1"
}
object Zone "global-templates" {
   global = true
}
```

Listing 4.9: *Definition of Zones and Endpoints(zones.conf)*

```
object CheckCommand "cpu_stats" {
   import "plugin-check-command"
   command = [ PluginDir + "/check_cpu_info" ]
   arguments = {
      "-w" = "88"
```

```
        "-c" = "93"
    }
}
```

Listing 4.10: *Definition of CheckCommand(commands.conf)*

```
apply Service "cpustat" {
    import "generic-service"
    check_command = "cpu_stats"
    assign where host.name == "icinga-client1"
}
```

Listing 4.11: *Definition of Service(services.conf)*

Plugins

Meanwhile, another crucial implementation aspect of the monitoring system was the designation of the checks that are going to be executed by the Icinga agents. Thanks to the integration of Icinga with most plugins of the `Nagios` platform, we were able to utilize various open-source performance checks, published in Nagios Exchange [8] and Icinga Exchange [9]. These plugins allow for generic monitoring of hosts, devices, services and applications. As we expected, however, our test cases called for instrumentation of asset-specific attributes, that were only partially compatible with any pre-defined plugin. Subsequently, we studied the structure and format of various Nagios plugins and followed it to define our custom performance check plugins. To this end, we made modifications or wrote from scratch various plugins, with respect to the test cases we studied. For example, we configured plugins that collect performance data from physical assets(e.g. `cpu_load,disk_capacity,memory_usage,network_utilization`), as well as others that measure the efficiency of software assets(e.g.`ceph_cluster_health, cluster_IOPS, cluster_throughput, ceph_daemon_state`). An analytical guide for writing custom Icinga Plugins that collect the desired performance data is available in the official Icinga documentation.

[8]`https://exchange.nagios.org`
[9]`https://exchange.icinga.com`

```
root@ontomonHost:~# ./check_http -H icinga2-client1 -w 10 -c 20
HTTP OK: HTTP/1.1 200 OK - 85997 bytes in 0.293 second response
    time |time=0.293414s;10.000000;20.000000;0.000000
    size=85997B;;;0
```

Listing 4.12: *Remote execution of HTTP plugin*

Integration with InfluxDB

At this point, we have successfully deployed the Icinga monitoring platform on the nodes of our cluster and configured it to aggregate real-time performance statistics from all hosts at a regular interval. However, in order to effectively store these metrics into the Influx time series database, we have to enable the native `influxdb` feature of Icinga. This feature instantiates an `Influx Writer` object, that is responsible for determining the structure of the output data. In particular, it transforms the collected key-value metrics into an `influx`-compatible format, by dynamically tagging instances of executed service checks. Its configuration is pretty straightforward and requires specific inht formation about the target storage backend, such as `host,port,database,username` and `password`. In addition, we were able to take full control over the format in which the time series data is exported, as the `Influx Writer` is capable of explicitly specifying the fields of the InfluxDB entries, as well as reporting metadata and threshold values. Combining the metadata that describe service checks(`execution_time, latency,max_check_attempts, max_check_attempts, etc`) with the thresholds settings provides a deep insight about downtimes or critical states of services. The configuration file of the `InfluxDBWriter` is shown in Listing 4.16:

```
root@ontomonHost:~# icinga2 feature enable influxdb
```

Listing 4.13: *Enabling the InfluxDBWriter in Icinga*

4.5 Metrics Storage with InfluxDB

Our next concern was to deploy the time series storage backend. We decided to also set up the latest stable version of the `influx-server`(1.2) on the `ontomonHost` node, so that the Icinga Master node can directly perform `write` requests and the network traffic congestion can be moderated. Therefore, we installed the respective `.deb` package files, along with the NTP [10] server, so that the timestamps of the data series would be correctly synchronized:

```
root@ontomonHost:~# wget
   https://dl.influxdata.com/influxdb/releases/influxdb_1.2.2_amd64.deb
root@ontomonHost:~# sudo dpkg -i influxdb_1.2.2_amd64.deb
root@ontomonHost:~# sudo apt-get install ntp
```

Listing 4.14: *InfluxDB and NTP installation*

Our intention was to gain full control over the format in which time series are stored in the storage backend. Thus, we had to understand the storage engine and schema that Influx is using. By knowing the exact structure and APIs used during data exchange process, we would be able to conveniently orchestrate both the request methods and the response handlers inside all consumer components of OntoMon. The top level container of InfluxDB is the `database`, that comprises of several components. Under the hood, data is stored in `shards`, which are in turn organized in `shard groups`, but this layer is almost transparent to the user. Every Influx `database` is characterized by its name and a `retention policy`, which determines both the replication factor and the time period after which data expires and is no longer considered valid(flush of the database). A `retention policy` is, actually, a bucket that divides a timeline into smaller segments. Hence, any incoming data points will fall into the corresponding segment, which is a retention policy bucket. The `database` contains multiple `data points`, which comprise of objects with the following attributes: `name,tags,timestamp` and `fields`. `Data points` are always associated with `series` or `measurements` that are created on the fly based on the aforementioned attributes. Supposing a collector agent is performing disk checks on a server and report-

[10]Network Time Protocol

ing the collected metrics to an InfluxDB, we can query the `Influx` server and get the following `JSON` response:

```
{
   "database" : "test-db",
   "retentionPolicy" : "24h",
   "points" : [{
      "name" : "disk",
      "tags" : {
         "server" : "test-server", "unit" : "1"
      },
      "timestamp" : "2015-03-16T01:02:26.234Z",
      "fields" : {
         "total_usage" : 100, "used" : 40, "free" : 60
      }
   }]
}
```

Listing 4.15: *Sample Influx server response*

Our approach involved proper configuration of the Icinga `InfluxDBWriter` object, in order to connect to the storage backend and perform `writes` of time series data. Firstly, we assigned the `measurement` of each data point the same name as the respective `CheckCommand` that collects it. Secondly, we introduced a small set of dedicated `tags`, that are used as keys in the InfluxQL queries, simplifying filtering and aggregation of data. No particular option was specified for the `timestamp` field, as it is automatically managed and encapsulated into data points.

```
object InfluxdbWriter "influxdb" {
    host = "127.0.0.1"
    port = 8086
    database = "icinga2"
    username = "icinga2"
    password = "supersecret"
        service_template = {
           measurement = "$service.check_command$"
        tags = {
            fqdn = "$host.name$"
            hostname = "$host.name$"
            service = "$service.name$"
        }
    }
    enable_send_thresholds = true
    enable_send_metadata = true
}
```

Listing 4.16: *InfluxDBWriter configuration*

4.6 Development of the OntoMon Observer

The monitoring stack of OntoMon has its foundation on the *push monitoring* method, since the deployed Icinga agents only push real-time metrics into the Influx database, and do not perform any alerting themselves. In this direction, though, soon arrised the need to develop a custom, highly available middleware that would be complementary to the Icinga platform, perpetually consuming the time series data and notifying the upper layer of our platform about any incidents regarding the assets of the target system. Hence, our approach was to design and implement the OntoMon Observer as a daemon that would silently run in the background, understand the Influx API and bridge the gap between the low-level monitoring stack and the user-level interface application. We implemented it entirely in `Python`, a high-level programming language that fully covered our needs. Throughout the development of the Observer daemon, we concentrated on ensuring two main characteristics: **modularity** and **adaptivity**. The logic behind this decision steams from the great diversity in both target systems and monitoring needs, as well as the ability to define custom visual notifications. Undeniably, different target systems comprise of conceptually divergent assets that are likely to have very specific performance indicators or attributes. However, even given the exact same target system, it is quite common that end-users or administrators have largely different perspectives and requirements from the monitoring platform. In such cases, the Observer is not aware of the content or policy of the performance check a user wishes to perform upon the target system, resulting in non-uniform requests.

Consequently, we needed a well-structured and standardized way of processing data: our implementation builds upon general-purpose concepts and mechanisms, providing a basic core of functionality. Nonetheless, in order to fulfil the user expectations, the abstract methods and handlers of the OntoMon Observer need proper specification by the system integrator, so as to serve the needs of different target systems. For the scope of this thesis, it was enough to direct methods and handler towards pre-defined monitoring services of fundamental physical and software assets, as it is further described in chapter 5. Furthermore, users are encouraged to further customize the Observer component to fit their needs, by following the proposed structure. For example, they can define arbitrary `CLI` arguments, combinatorial metric checks or priority policies. We argue that this approach can lead to a more versatile and flexible Observer component. The proposed implementation of the Observer comprises of 3 distinct components:

- `Observer.py`
  This module contains the central driver code for our Observer daemon and determines the check services upon which the Influx server will be queried so that the latest corresponding metrics can be periodically retrieved.

```python
import threading
import services
import constants
import argparse

# support for CLI arguments
parser = argparse.ArgumentParser()
parser.add_argument("--hosts", nargs='+', help="Enter hosts to
    be monitored")
args = parser.parse_args()
...

def execute_checks(hosts):
    for host in hosts:
        if services.host_alive(host):
            # Observe cpu asset of given host
            services.cpu_check(host)
            # Observe memory asset of given host
            services.memory_check(host)
            # Observvce disk asset of given host
            services.disk_check(host)
        else:
            print host, 'is down-Cannot get real time metrics'
    threading.Timer(constants.CLUSTER_HEARTBEAT, execute_checks,
        [args.hosts]).start()
```

Listing 4.17: *Simplified version of Observer.py*

- `Services.py`

  This module implements the core logic of the OntoMon Observer, as it includes definitions of handler functions for each monitoring service applied to assets in the `observer.py` file. Our daemon is responsible for communicating via `HTTP` with the storage backend in order to fetch the corresponding real-time performance metrics from `ontomonHost:8086/query`. The respective InfluxQL requests are dynamically generated at runtime by the general-purpose `get_measurement()` function we provide. This function also parses the `JSON` body of the response coming from the Influx storage backend and returns a `list` containing the desired performance metrics. The next task regards the digestion and refinement of data and is implemented inside the body of the handler functions. For example, users might need isolated or aggregated values(e.g. mean usage of a resource) in order that proceed to the performance evaluation phase. In this phase, the parsed performance data is passed to respective reasoners, which are responsible for performing comparisons against pre-defined notification thresholds. Essentially, these reasoners illustrate the monitoring and alerting policies that system integrators integrate into OntoMon. Consequently, the Observer concludes on the current state of individual monitored assets. This threshold-based technique to detect anomalies can be regarded as a form of service level agreement (SLA), which is practically a commitment between the service provider and the end-user. Based on the comparison, it can be any of the following: `OK, WARNING` or `CRITICAL`. It is important to carefully determine the notification thresholds so that no time or attention is spent on non-issues or instantaneous changes. The last step involves propagating the results of the evaluation phase to the OntoMon Server in the form of asset state updates, using the `dispatcher` module.

```python
import httplib, urllib, json
...
def get_measurement(hostname, measurement, metrics, fields, time):
    metrics_number = len(metrics)
    metrics_string = ",".join(metrics )
    fields_number= len(fields)
    fields_string = ",".join(fields)
    if metrics_number > 1:
        query=("select metric,"+fields_string+" from "+measurement+
            " where fqdn='" + hostname +
            "' and time >= now()-"+time+" and (metric='" + metrics[0]+"'")
        metrics.pop(0)
        for m in metrics:
            query+=" or metric='"+m + "'"
        # aggregators
        query += ") order by time desc limit "+str(metrics_number)
    else:
        query=("select metric,"+fields_string+" from "+measurement+
                " where fqdn='"+hostname+"' and time >= now() - "+time)
        # aggregators
        query+=" order by time desc limit 1"

    params=urlencode({"q": query, "db": constants.DATABASE_NAME})
    headers={"Content-type": "application/x-www-form-urlencoded",
               "Accept": "application/json"}
    conn_query =
        httplib.HTTPConnection(constants.INFLUX_HOST_ENDPOINT)
    conn_query.request("POST",
        constants.INFLUX_QUERY_ENDPOINT,params, headers)
    r = conn_query.getresponse()
    ...
```

Listing 4.18: *Simplified version of get_measurement()*

```
import constants
import dispatcher

# observe usage percentage of memory asset
def memory_check(hostname):
   ...
   # get metrics by specifying host, measurement, metrics, fields and time window
    mem_measurement_list = get_measurement(
      hostname, "memory", ["MUSED", "MTOTAL", "SUSED", "STOTAL"], ["value"], "5m")
    for m in mem_measurement_list:
        mem_dict[m['metric']] = m['value']
    # data analysis
    mem_timestamp = mem_measurement_list[0]["time"]
    mem_used_MB = round(mem_dict["MUSED"]/(1024*1024), 2)
    mem_total_MB = round(mem_dict["MTOTAL"]/(1024*1024), 2)
    mem_used_percentage = round(100*mem_used_MB/mem_total_MB, 2)
    mem_free_percentage = 100-mem_used_percentage
    ...
    if (mem_used_percentage >= constants.MEM_USAGE_CRITICAL):
        issue_description = "Check: mem_used_percentage"
        mem_status = "CRITICAL"
    elif (mem_used_percentage >= constants.MEM_USAGE_WARNING):
        issue_description = "Check: mem_used_percentage"
        mem_status = "WARNING"
    ...
    mem_metrics = {
        'mem_used': str(mem_used_MB) + 'MB',
        'mem_total': str(mem_total_MB) + 'MB',
        'mem_used_%': str(mem_used_percentage) + '%',
        'mem_free_%': str(mem_free_percentage) + '%'
    }
    if (mem_status != "OK"):
        issue_description = "Check " + issue_description
    else:
        issue_description = "No issues were detected."
    # Send latest memory usage status to OntoMon Server
    dispatcher.send_update(hostname, mem_timestamp, mem_status, mem_metrics,
        issue_description)
```

Listing 4.19: *Simplified Service example*

- `Dispatcher.py`

  In this module, we implement functions that formalize the structure of the `JSON` objects describing the current state of the monitored assets, as well as the latest values of metrics and direct guidelines regarding the visualization of the alert in the UI of OntoMon. The `dispatcher()` function serially publishes updates at `ontomonHost:8080/updates`. Technically, this is where our custom **alerting API** is specified, in the sense that we introduce a standardized format for the objects representing a change in the status of an asset. At the same time, we employ a `regex`-based validation of attribute payloads, so as to ensure that every `Update.json` object is constructed as expected. The proposed notification protocol is expressed in the `JSON` object presented in 4.6:

```
{
    "uuid": "ff2802db-15d1-42a6-bcfe-0dd3af12c9c7",
    "name": "icinga2-client1",
    "timestamp": "2017-05-15 00:56:12",
    "state": "OK",
    "metrics": {
        "cpu_load1": 0.5,
        "cpu_load5": 0.7,
        "cpu_load15": 0.8
    },
    "type": {
            "element_class": "indicator",
            "attribute_name": "values",
            "attribute_value": color
        },
    "description": "No issues were detected."
}
```

Listing 4.20: *Update.json Object*

In our custom alerting chain, each *Update.json* object is associated with a specific asset and check service, while it is a responsibility of the system integrator to determine its content. Accordingly, the `timestamp` field holds the time of the most recent data sampling, the `state` field retains the most recent state of the respective asset, as deduced by the reasoner. Furthermore, the `metrics` field contains a nested `JSON` object with key-value pairs of performance metrics names and values. The `type` attribute is a quite important one, as it explicitly defines the way the respective status change will be depicted in the Angular application: it is a nested `JSON` object whose fields determine which element and attribute will be updated inside the corresponding `SVG`. Finally, the `description` attribute provides useful insight to quickly observe which metrics are, possibly, causing performance degradation.

4.7 Development of the OntoMon User Interface

As in any monitoring platform, the implementation of the Web UI of our framework was of significant importance. Due to the fact that the User Interface is the only visible component of OntoMon, we focused its design on providing a solid, comprehensive and consistent overview of all the underlying computations made by our engine. We decided to build our web application on top of Node.js: we used the latest stable version (4.0) of Angular, along with `TypeScript` 2.1. As far as appearance is concerned, we followed some widely accepted guidelines and laid emphasis on simplicity and functionality. From our perspective, a successful UI should not be overloaded with information; instead, it should be concise and dedicated, while eliminating confusion and facilitating usability.

According to the aforestated principles, we designed the User Interface of OntoMon as a single page application, that serves 4 different pages: *overview, assetview, log* and *settings*. The *overview* page is the default one, and provides a holistic view of the target system. On the other hand, the *assetview* page targets a specific asset and provides all related information, performance metrics and graphs. Each page is implemented in a different Angular `Component`. Besides, we decided to construct additional auxiliary `Components` that would be responsible for managing various complementary UI elements, such as a header toolbar, a side navigation bar with the Ontology tree, a pop-up dialog window with performance metrics and the panels coming from the Graph Composer. Meanwhile, following the best practices concerning the architectural design of Angular applications, we introduced task-specific `Services`, that abstract various concerns from the main `Components` and extend their functionality. Such concerns involve the retrieval and validation of the *Ontology.json* and *.svg* files, as well as the local storage and distribution of data among `Components`. Hence, we greatly simplified the structure of our application and facilitated its future maintenance by defining clean and reusable code entities.

In order to reduce the complexity during the development process of OntoMon Web Interface, we decided to work in an iterative fashion:

- **Cycle1: Ontology parsing and validation**
  In this phase, we conduct all the pre-processing needed to build our application. As expected, the `Components` and `Services` active in this phase implement the fundamental ontological concepts of OntoMon. At the same time, they perform the initialization of the basic data structures used across our application, by hitting the `ontomonHost:8080/resources` endpoint. Initially, the `AppComponent` retrieves the *Ontology.json* file using the `HttpService` and forwards it to the `Validator Service`. Our validation mechanism employs the `D3.js` library to manipulate the `JSON` data and build the hierarchical tree structure, based on the `parent` attribute of each object of the Ontology. After ensuring that no cycles or undefined objects exist inside the Ontology, the `TreeData` array is initialized and the `ParserService` is invoked. The latter parses the contents of the *Ontology.json* file into the `Controller` object, which is later ex-

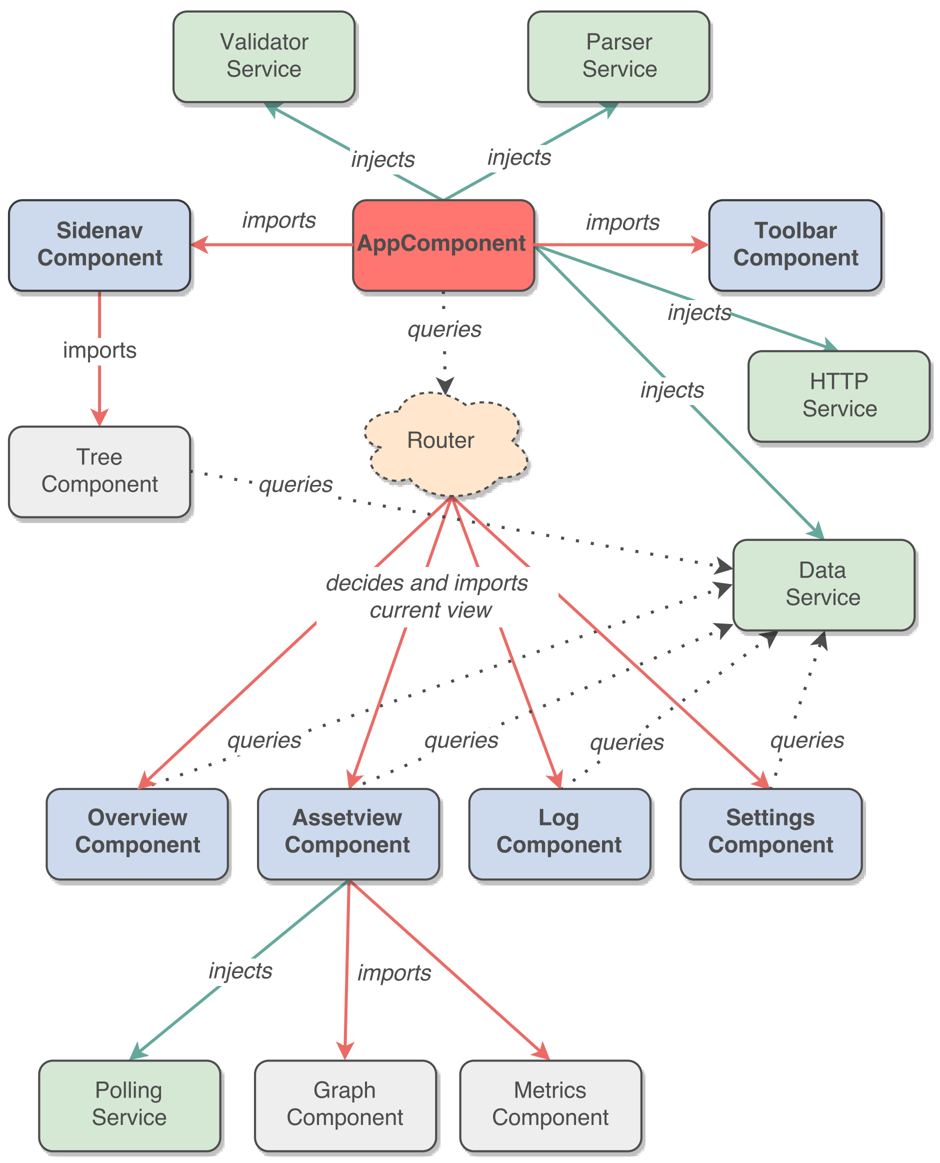


Figure 4.5: *OntoMon UI: Angular application architecture*

tended with additional information about each individual asset. It is notable, that the entries of the `Controller`, which is the most crucial data structure inside our UI, is indexed by the `uuid` attribute of the objects. Both the `TreeData` and the `Controller` are stored inside the `DataService`, so that data is made available to all components of our application by simply injecting the `DataService`.

Since the aforementioned operations involve client-server communication or end-to-end parsing of large files, the methods implementing them are mainly **asynchronous**. Thus, in order to effectively synchronize our application, we needed a *listener/notifier* mechanism. Subsequently, we decided to utilize the native `EventEmitter` class, along with the `rxjs/Observable` package. In this way, we were able to timely instantiate `Components` and trigger their methods at the right time. More specifically, when the `DataReady` event was emitted, the `TreeComponent` proceeded to the reactive visualization of the Ontology Tree using the `treemap` structure that the `D3.js` supports. Similarly, the `ToolbarComponent` was able to group the objects of the Ontology based on their `label` attribute, and dynamically build drop down menus used for asset-oriented routing inside the application.

```
Object {
    "uuid": "c8bd7185-6349-4df5-8628-f87115222987",
    "name": "Ubuntu-Server-1",
    "label": "Server",
    "file": "Server.svg",
    "parent": "25e2ce69-2445-4b28-9a71-e7ca01bc57ea",
    "state": "OK",
    "lastUpdated": "2017-05-15 00:56:12",
    "description": "No issues were detected.",
    "slotAvailability": [false, true, true, true],
    "children": ["9c5d5b73-68ac-4a9d-92a7-7c60e492a7bd"],
    "info": {
        "OS": "Ubuntu 16.04.2",
        "brand": "IBM",
        "chassis_type": "Rackmount",
        "color": "black",
        "description": "Server asset",
        "dimensions(H/W/D)": {
            "depth": 12,
            "height": 28.9,
            "width": 17.5
        }
        "form_factor": "7U",
        "model": "88861TU",
        "series": "BladeCenter S"
    },
    "metrics": {
        "cpu_load1": 0.5,
        "cpu_load5": 0.7,
        "cpu_load15": 0.8
    }
}
```

Listing 4.21: *Formulation of Ontology objects inside the Controller*

- **Cycle2: SVG visualization**

  After the initialization phase successfully completes, our application proceeds with the visualization of the monitored assets, strictly following the hierarchy specified in the ontological description. Again the first step is to reach the OntoMon Server at `ontomonHost:8080/resources`, but this time obtain all the `.svg` files that the end-user has already uploaded. As already mentioned, each `.svg` file is associated with a specific object of the Ontology and denotes its visual representation inside the OntoMon User Interface. However, it is possible that adjustments are made by end-users upon the target system and, consequently in the `Ontology.json` and `.svg` files. Subsequently, in order to make our platform adaptive to such changes, both the `Overview` and the `Assetview Components` are configured to make respective `HTTP` requests to the OntoMon Server each time they are loaded, always obtaining the most recently updated resources.

  As soon as all *.svg* files arrive, they are immediately stored locally and the visualization process is initiated. Aiming at preserving the performance and scalability of our application, we made the pretentious decision to directly manipulate the browser `DOM`, handling `SVGs` as raw `XML` code. With this approach, we were able to create fully-controlled `HTML` nodes on-the-fly and extend the `DOM` Tree that the Angular application has already defined. However, the key point of our implementation lies in the well-defined Ontology Tree: by calling the `traversal()` method, the view `Components` perform a `BFS`[11] traversal of the current Ontology Tree, starting from the root node. As every node of the Ontology Tree

[11]Breadth First Search

is an entry of the `Controller` object, we can easily retrieve the associated `SVG` via its `uuid` attribute and append it to the existing `DOM` Tree as a leaf node. Similarly, visualizing a specific sub-system of the initial target system is a matter of traversing the corresponding subtree of the Ontology Tree.

At this point, we have a both reliable and consistent way of visualizing the Ontology objects, since their respective `SVGs` are processed and rendered in the correct order: gradually, scanning the Ontology tree in horizontal levels, guarantees that parent elements will be attached first to the `DOM Tree`, followed by their children. Still, we had to tackle the issue of precisely translating and scaling the nested `.svg` files, so that they appear at the anticipated position in the display and exactly fit inside their parents. To achieve that, we parsed the `slot` attribute of the user-defined `SVGs` and modeled slots as containers that have a concrete reference point and fixed dimensions. The number of slots, along with their exact dimensions and availability are also stored in the `Controller` object of our application, extending the already existing entries. Next, we call the native `getBBox() JavaScript` method upon the `SVG` that is about to be rendered. This method returns the `width` and the `height` of a notional rectangle that exactly encloses this specific `HTML` element. Relying on this information, we calculated the necessary scale and displacement factors and subsequently applied the proportional transformations upon each `SVG` element. In particular, we delegated these transforms to `<animation>` elements that we appended to `SVGs`, making the UI of OntoMon more graceful and user-friendly.

- **Cycle3: Real-time data and alerting**

  As far as the last phase of operation is concerned, we invested much time and effort on the presentation layer and specifically on the display of performance metrics that the underlying system has collected. Since this is the actual output of the instrumentation that our monitoring platform has performed, the core mission of our User Interface is to provide end-users with an instant overview of their assets and give them qualitative insight regarding their current state and functionality. The precondition to achieving this was to actually acquire the most recent performance metrics gathered by the Icinga agents. Hence, we employed a `PollingService` that we scheduled to periodically query the OntoMon Server at `ontomonHost:8080/updates` via `HTTP GET` requests about

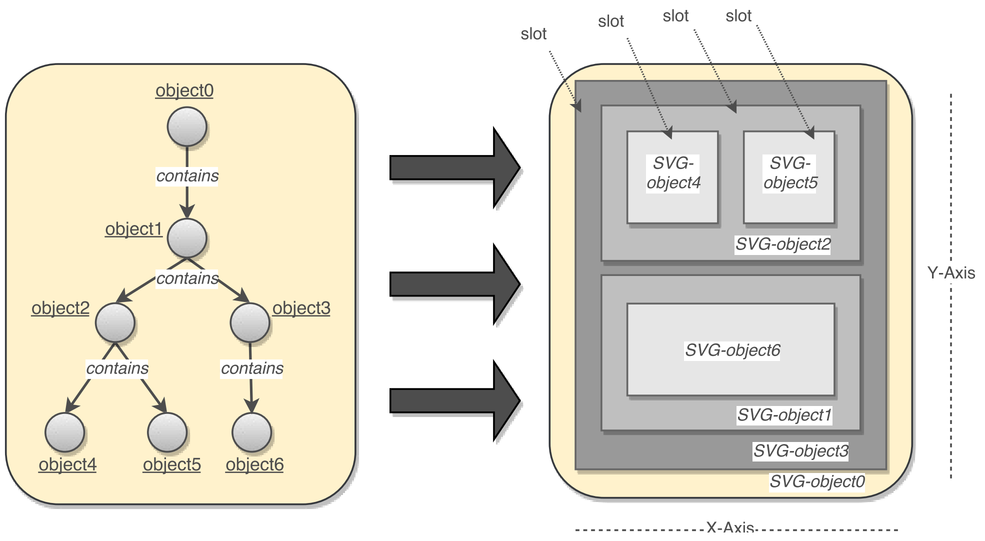


Figure 4.6: *Ontology Tree mapped to nested HTML DOM nodes*

pending target system updates. Every *Update.json* object returned by the OntoMon Server represents the most recent known state of the respective asset. Accordingly, parsing the values of the *Update.json* object attributes supplied our application with extensive details about the visualization of the alert, together with its latest metrics and state of operation. Both the `state` and the `metrics` attributes are also reposited in respective fields of the `Controller`. Aiming at comprehensively demonstrating the different check services executed on each host, we incorporated this information into two equivalent but alternative views: the `MetricsComponent` and the `GraphComponent`. These components inject the `DataService` holding the latest real-time data of each asset in the `Controller` data structure. The former summarizes performance indicators into a concise 2-column table for quick examination, while the latter communicates with the `grafana-server` server via `HTTP` and embeds real-time graphs of the same metrics, as described in section 4.8. We argue that the aggregation of all asset-specific metrics into a single view allows for efficient inspection and proper management of the monitored infrastructure. Moreover, we configured the Web UI of OntoMOn to periodically dispatch `HTTP GET` requests to the `ontomonHost:8080/log` web endpoint, in order to obtain an aggregated log of all events that were recorded during the runtime by the OntoMon Server. This log file(*Update.log*) is visualized inside the template of the `LogComponent`, demonstrating the timeline of the status updates of the target system in chronological order.

The last feature we had to design was the OntoMon notification mechanism, which is mainly implemented in the devoted `updateUI()` method we defined. The main problem that this method alleviates, is the inconsistency between our current visualization of the target system and the actual state of the real-world assets. Initially, we presume that all assets of the target system, either software or hardware, are operating as expected. Though, while our application pulls metrics and updates from the OntoMon server the current view presented to the end-user must be updated in order to reflect any issues that might have occurred inside the target system. The `updateUI()` method is triggered asynchronously by subscribing to `Observables`. As a result, every time our application receives a new *Update.json* object, it actively checks whether the state of the given asset has changed since the last time its state was reported by the monitoring system. If so, it takes the following actions: firstly, it sends an alarm to the end-user, in the form of a push notification which appears in the upper right corner of the screen and, secondly, using the `querySelector` method upon the `HTMLdocument` of the `AssetviewComponent`, it locates the `indicator` element of the "dirty" `SVG` and performs a pre-determined visual update of it, which in our implementation is a simple change of coloring.

4.8 Graph Composition with Grafana

In the proposed design, the generation of real-time graphs is of significant importance, so we had to deliberately configure Grafana. In the first place, we installed the related software packages(latest stable 4.3 version) on a the `ontomonHost` node so that a single, dedicated process of the OS would manage the composition of time series graphs:

```
root@ontomonHost:~# wget https://s3-us-west-2.amazonaws.com/
  grafana-releases/release/grafana_4.3.0_amd64.deb
root@ontomonHost:~# apt-get install -y adduser libfontconfig
root@ontomonHost:~# sudo dpkg -i grafana_4.3.0_amd64.deb
```

Listing 4.22: *Installation of Grafana*

Afterwards, we proceeded to the regulation of the `grafana` daemon, mainly to determine its core options and input data sources. Plotting time series with Grafana, requires that its data puller is capable of reaching and querying the storage backend via `HTTP`. In our case, we had to specify certain fields inside the configuration file of Grafana, located at `ontomonHost:/etc/grafana/`, such as:

```
   ...
# Protocol, access domain, port
protocol = http
domain = localhost
http_port = 3000
root_url = http://localhost:3000
# Basic AUTH for login
enabled = true
# Database for storing users and settings
type = sqlite3
host = 127.0.0.1:3306
name = grafana
user = root
   ...
```

Listing 4.23: *Grafana Server configuration*

Alongside, we defined a new data source for `Grafana`, named "Cluster Metrics", containing all real-time data that Icinga is collecting from the nodes of the target system. Specifically, we determined the following core attributes:

```
{
   "Name": "Cluster Metrics",
   "Default": true,
   "Type": "InfluxDB",
   "Url": "http://ontomonHost:8086",
   "Access": "proxy",
   "Enable_http_auth": false,
   "Details": {
      "Database": "icinga2",
        "User": "icinga2",
        "Password": "secret"
    }
}
```

Listing 4.24: *Grafana Data Source configuration*

The Grafana daemon issues HTTP requests at a regular interval at the specified target Url, pulling the latest performance metrics related to the monitored assets from InfluxDB. The last step was to set up the *panels* inside the Grafana *dashboard*. Each panel is the graphical representation of a specific performance measurement, providing insight into the behavior of the respective asset over time. The core configuration of a panel requires the definition of InfluxQL queries upon the storage backend, so that the desired metrics are returned by the influx-server. Since we monitored multiple hosts on the same assets using the same check services, we had to find an efficient way of avoiding duplicate panels and redundant definitions. Therefore, we introduced **template** variables, such as asset, so that the exact same panels and queries are applicable on different assets and present the respective performance values. Two characteristic examples of the queries we generated are provided below:

```
-- Query on hardware performance
SELECT mean("value") FROM "cpu_load"
WHERE "fqdn" =~ /^$hostname$/
AND "metric" = 'load_1min' AND $timeFilter
GROUP BY time($interval) fill(null)

-- Query on software performance
SELECT mean("value")
FROM "ceph_osd_stats"
WHERE "hostname" =~ /^$hostname$/ AND "metric" =~ /_bytes_/
    AND $timeFilter
GROUP BY time($__interval), "metric" fill(null)
```

Listing 4.25: *Example InfluxQL queries used in Grafana*

The first query retrieves the cpu_load measurement monitored on the node with `fqdn` tag equal to the template variable `hostname`. Then it filters the result by the `load_1min` metric, as long as it was collected in the `timeFilter` time window, and selects its mean value. Then, it aggregates the results along the time scale, by organizing them into groups determined by an `interval`. Analogously, the second query retrieves the `ceph_osd_stats` measurement monitored on the `hostname` node, filters by the `bytes` metric and the `timeFilter` time window, selects the mean value and aggregates the result. As far as the display options are concerned, Grafana supports rich and diverse graphs to visualize the obtained time series data. In particular, we selected `bars,points,lines,singlestats,pie charts,histograms` and `gauges` to effectively illustrate each performance metric.

Finally, in order to import the generated panels into the Web User Interface of OntoMon, these graphical components must be served at a be publicly available endpoint. Hence, we took advantage of the *dashboard sharing* feature of Grafana that generates direct `links` to individual Grafana panels. These `links` include the user-specified time range and template variables as parameters. These links can lead either to interactive Grafana graphs or snapshots.

```
<iframe
   src="http://ontomonHost:3000/dashboard/db/dashboard-0?
     orgId=2&var-asset=icinga2-client1&panelId=2&from=now-15m&to=now"
   width="450" height="200">
</iframe>
```

Listing 4.26: *Import of Grafana panels as iframes*

We decided to use these `links` inside `<iframe>` elements included in the `HTML` code of our `Angular` application. In this way we benefit from **client-side rendering** feature that Grafana supports, which involves rendering web content inside the browser using `JavaScript`: initially, only the bare minimum `HTML` is transferred from the server, while the rest of the content is dynamically loading by `script` elements and dependency libraries on the client(e.g. `Phantom.js`).

All implementation code is available in `https://github.com/gozek/Ontomon`

5 Experimental Evaluation

In this chapter, we will describe our experience from using OntoMon in practice. Our goal is to test the behavior of our monitoring and visualization platform in diverse use case scenarios and draw useful conclusions regarding our design and deployment choices. As a proof-of-concept, we generated two different inputs for OntoMon and observed the corresponding outputs. In section 5.2 we define a testing environment that is monitored with regard to its hardware assets, while in the section 5.3 we focus on the software assets of a different target system. Throughout this chapter, we guide the reader with comprehensive screenshots and analysis of the respective configurations.

5.1 Testbed Environment & Checkpoints

In order to confirm that the platform we deployed is properly operating and delivers meaningful output to the user, we decided to deploy OntoMon in a testing environment that resembles real-world systems in both architecture and design. Ideally, we would like to test our platform on real Data Centers, however, this was not feasible in the scope of this thesis. Hence, the testbed we used for both test cases was the neat and convenient 3-node VM cluster that we previously deployed as the target system in section 4.1. In addition, we decided to use the hypervisor node as the `ontomon-Host`, which accommodates the full OntoMon stack and all the core components of our platform, including the OntoMon Server, the OntoMon Observer and OntoMon Web Interface. Since the test cases we studied were not conflicting but complementary to each other, we used the exact same virtual nodes to emulate two different use case scenarios for our tool, in order to reduce resource utilization on the physical host. In general, the procedure we followed in both scenarios can be summarized in the steps below:

- We first generate a `JSON`-formatted Ontology that describes the target system we are going to supervise, clearly specifying its assets and inner relations by following the proposed design principles. We also upload the *Ontology.json* and all corresponding *.svg* files to the OntoMon Server.
- We then proceed to the configuration of the monitoring layer based on each test case, acting as system integrators, so as to determine which assets of the target system will be monitored and which performance checks will be executed. Therefore, we configure all Icinga objects and implement the respective handler functions inside the Observer.
- Next, we access the Web Interface of OntoMon, acting as end-users, and expect to see a representational visualization of the target system described in the Ontology, along with the current state of the monitored assets.
- Finally, aiming at cross-checking the alerting mechanism and the automatic updates of the UI, we focus on specific assets, fake some failure or performance degradation upon them and anticipate to observe the corresponding real-time notifications and performance evaluation inside the UI of OntoMon.

5.2 Physical IT Infrastructure Monitoring

In this test case, our goal is to monitor and visualize the physical infrastructure inside a Data Center. Since modern Data Centers enclose innumerable physical assets, we decided to restrict our attention to certain types of entities, such as: **World, Data Centers, Racks, Servers, CPUs, Disks, Memories** and **NICs**. We grounded this decision upon the fact that the aforementioned entities are of significant importance when measuring the performance of large-scale computer clusters. Alongside, these entities were sufficient and diverse enough to demonstrate the monitoring and visualization capabilities of our platform. In essence, we had to create a mapping between physical assets and virtual equipment. Therefore, we came up with a notional description of assets and their placement inside a Data Center, depending on documentation available on the Web. The ontological description we generated was confined, yet complete regarding content. We defined several objects, determined their parent relations and associated each object with an `SVG` file:

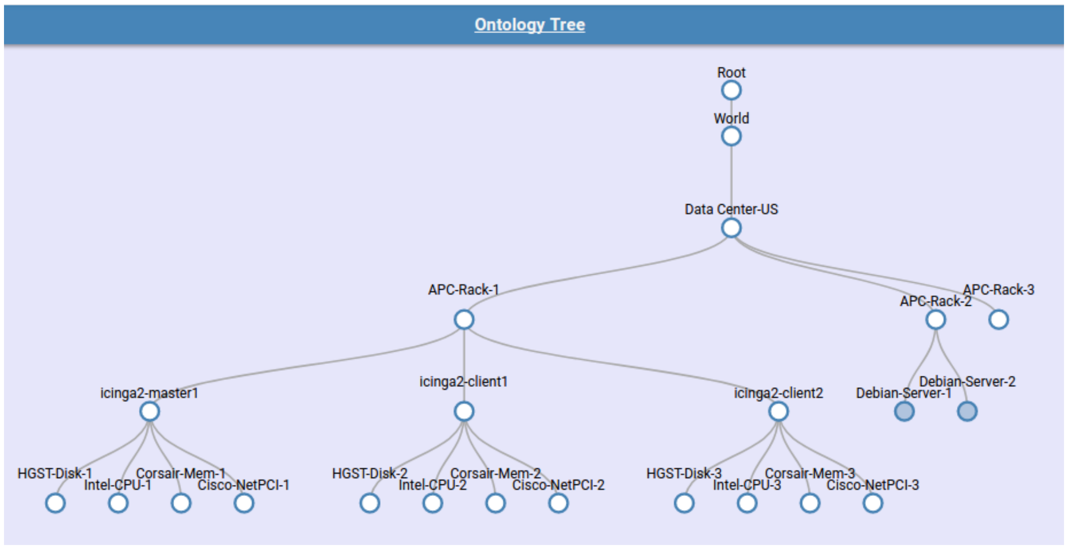


Figure 5.1: *OntoMon UI: Ontology Tree of physical infrastructure*

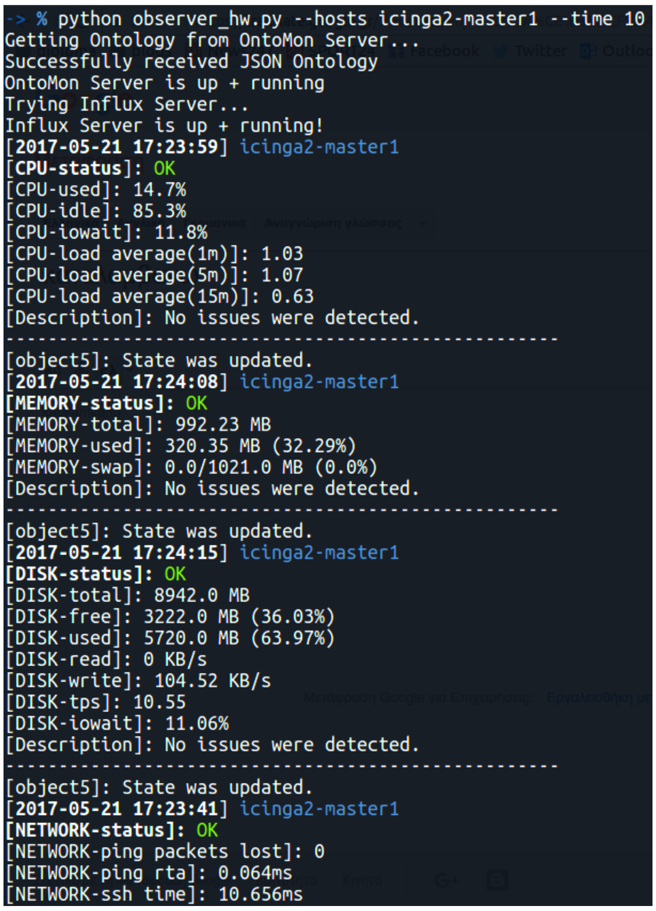
As shown in Figure 5.1, the *Ontology.json* file includes a single Root object, 1 Data Center, 3 Racks, 5 Servers and multiple CPUs, Memories, Disks and NetPCIs attached to the Server assets. Every edge of the Ontology Tree denotes a `parent` relation between assets. Our virtual cluster is represented in the leftmost subtree, hosted in the APC-Rack-1 cabinet and consisting of the 3 server nodes we are targeting. Even though we concentrated our analysis on the supervision of 3 Server assets, the scope of the proposed Ontology revealed the capability of OntoMon to efficiently monitor even larger deployments. Moving to the configuration of the monitoring layer, in order to effectively measure performance and timely detect failures of server instances, we had to collect indicative metrics that reflect how well assets are currently performing. Based on our assumption that each server contains 4 physical assets, we configured various respective Icinga `Services` to perform their instrumentation, as shown in Table 5.1.

We can confirm that the compiled time series data coming from the hardware assets are stored in the desired format in the Influx storage backend, by querying the `influx-server` from a terminal. Hence, the next step was to regulate the OntoMon Observer to enable consumption and reasoning over the collected metrics. To achieve that, we had to modify the core implementation of the Observer and extend its processing ca-

Physical Asset	Icinga Services	Metrics
CPU	`check_load, check_procs, cpu_stats`	`procs num, cpu usage, average load(1m/5m/15m)`
Disk	`check_disk, io_stats`	`disk usage & capacity per partition, reads/s, writes/s, tps, iowait`
Memory	`check_memory`	`memory usage & capacity, swap usage & capacity`
Network	`check_traffic, check_http, check_ssh, cluster_zone`	`transmit/s & receive/s per interface, hostalive, ssh time, connection to cluster`

Table 5.1: *Performance checks for physical server assets*

pabilities to support hardware monitoring. At first, we adjusted the `observer.py` module by adding two command line arguments: the `--hosts` argument determines which hosts are going to be watched by the OntoMon Observer and the `--time` argument specifies the time range of the queries issued on the time-series storage backend. Secondly, we implemented the `services_hw.py` module, where we defined various dedicated handler functions that receive and manipulate performance metrics associated with individual physical assets. In particular, we introduced the `memory_check(), disk_check(), cpu_check()` and `network_check()` functions and set the corresponding `OK, WARNING` and `CRITICAL` thresholds for the scheduled checks. Each performance value is compared to the threshold we have already defined and the result determines the current state of the asset. As far as the visual notifications are concerned, we provided very simple instructions inside the *Update.json* object, only changing the coloring of the `indicator` element of the respective `SVG`. For debugging purposes, we also configured the Observer to output monitoring results per service in the terminal. Furthermore, by measuring the system resources utilized by OntoMon Observer during runtime, we ascertained that we implemented a low-overhead daemon that efficiently serves the intended purpose. Indicatively, using native Linux tools we measured `0.25%` of `CPU` and `8.7MB` of `RAM`.



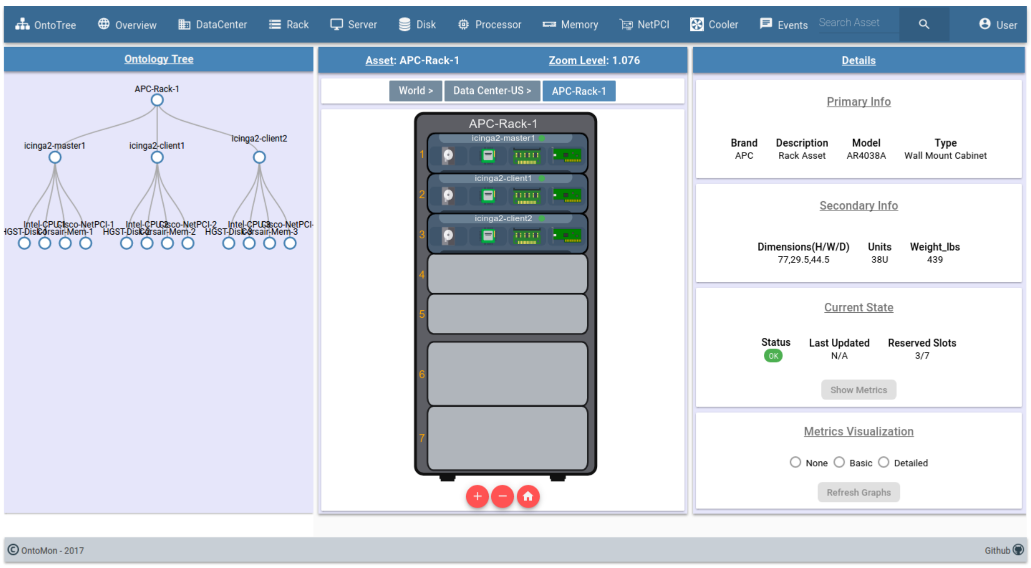
```
-> % python observer_hw.py --hosts icinga2-master1 --time 10
Getting Ontology from OntoMon Server...
Successfully received JSON Ontology
OntoMon Server is up + running
Trying Influx Server...
Influx Server is up + running!
[2017-05-21 17:23:59] icinga2-master1
[CPU-status]: OK
[CPU-used]: 14.7%
[CPU-idle]: 85.3%
[CPU-iowait]: 11.8%
[CPU-load average(1m)]: 1.03
[CPU-load average(5m)]: 1.07
[CPU-load average(15m)]: 0.63
[Description]: No issues were detected.
---------------------------------------------------
[object5]: State was updated.
[2017-05-21 17:24:08] icinga2-master1
[MEMORY-status]: OK
[MEMORY-total]: 992.23 MB
[MEMORY-used]: 320.35 MB (32.29%)
[MEMORY-swap]: 0.0/1021.0 MB (0.0%)
[Description]: No issues were detected.
---------------------------------------------------
[object5]: State was updated.
[2017-05-21 17:24:15] icinga2-master1
[DISK-status]: OK
[DISK-total]: 8942.0 MB
[DISK-free]: 3222.0 MB (36.03%)
[DISK-used]: 5720.0 MB (63.97%)
[DISK-read]: 0 KB/s
[DISK-write]: 104.52 KB/s
[DISK-tps]: 10.55
[DISK-iowait]: 11.06%
[Description]: No issues were detected.
---------------------------------------------------
[object5]: State was updated.
[2017-05-21 17:23:41] icinga2-master1
[NETWORK-status]: OK
[NETWORK-ping packets lost]: 0
[NETWORK-ping rta]: 0.064ms
[NETWORK-ssh time]: 10.656ms
```

Figure 5.2: *OntoMon Observer: monitoring hardware assets of single host*

```
$ curl -G 'http://ontomonHost:8086/query?pretty=true' \
\ --data-urlencode "db=icinga2" --data-urlencode "q=SHOW MEASUREMENTS"
{  "results": [{
      "series": [{
         "name": "measurements",
         "columns": ["name"],
         "values": [
            ["cluster-zone"], ["cpu_stats"], ["disk"],["hostalive"],
            ["http"], ["io_stats"], ["load"], ["memory"], ["ping4"],
            ["network_traffic"], ["procs"],["ssh"],["swap"]]
      }]
   }]
}
```

Listing 5.1: *Influx server JSON response*

Subsequently, we accessed the User Interface of OntoMon. Our first objective was to verify that the optical representation of the Data Center inside the web application corresponds to the ontological description we provided to OntoMon. More specifically, we navigated to the `assetview/APC-Rack-1` page, where all 3 server assets should be visible. Indeed, the `SVGs` associated with the 3 servers we supervise are clearly depicted inside the rack `SVG`, placed in different rows. This also expected, as we correlated each row of the rack with a single `slot` in its `SVG`. Server `SVGs` additionally enclose the CPU, Memory, Disk and NetPCI `SVGs` that we previously defined. Notably, each physical asset is scaled and translated to occupy exactly one `slot` of its parent asset, indicating that the relative transformations we performed were accurate. Our second objective was to observe the presentation layer of the Web Interface, select a specific server and observe its behavior through concentrated tables and reactive graphs. From our point of view, the representation of performance time series data is both insightful and qualitative, helping the end-user attain a deeper and clearer perception of its current performance. To this end, we regard the integration of `Grafana` as the graph composer of our Web UI successful.



Asset: APC-Rack-1 Zoom Level: 1.076

Details

Primary Info

Brand	Description	Model	Type
APC	Rack Asset	AR4038A	Wall Mount Cabinet

Secondary Info

Dimensions(H/W/D)	Units	Weight_lbs
77,29.5,44.5	38U	439

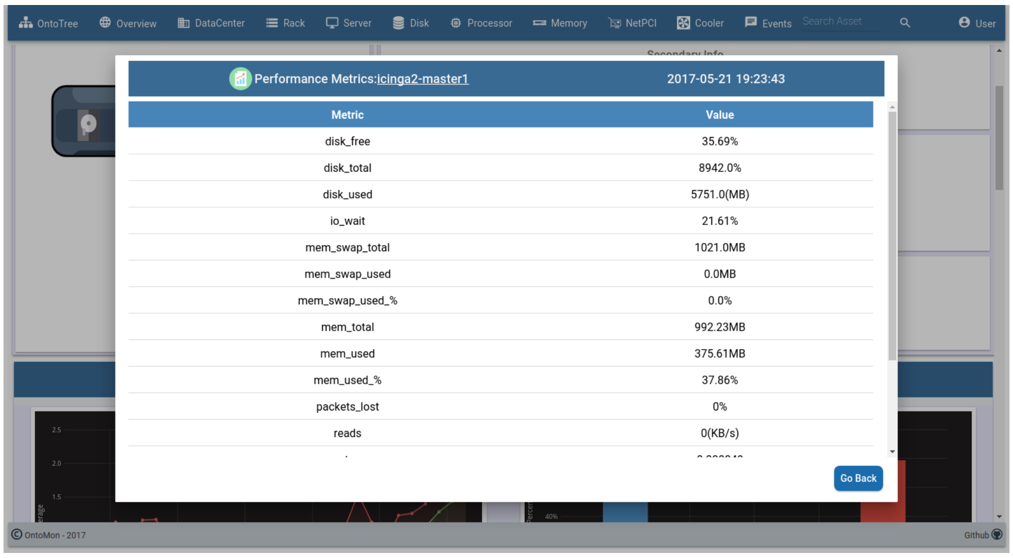
Current State

Status	Last Updated	Reserved Slots
OK	N/A	3/7

Metrics Visualization

© OntoMon - 2017

Figure 5.3: *OntoMon UI: visualization of nested hardware assets*



Performance Metrics:icinga2-master1 2017-05-21 19:23:43

Metric	Value
disk_free	35.69%
disk_total	8942.0%
disk_used	5751.0(MB)
io_wait	21.61%
mem_swap_total	1021.0MB
mem_swap_used	0.0MB
mem_swap_used_%	0.0%
mem_total	992.23MB
mem_used	375.61MB
mem_used_%	37.86%
packets_lost	0%
reads	0(KB/s)

Figure 5.4: *OntoMon UI: performance metrics in table*

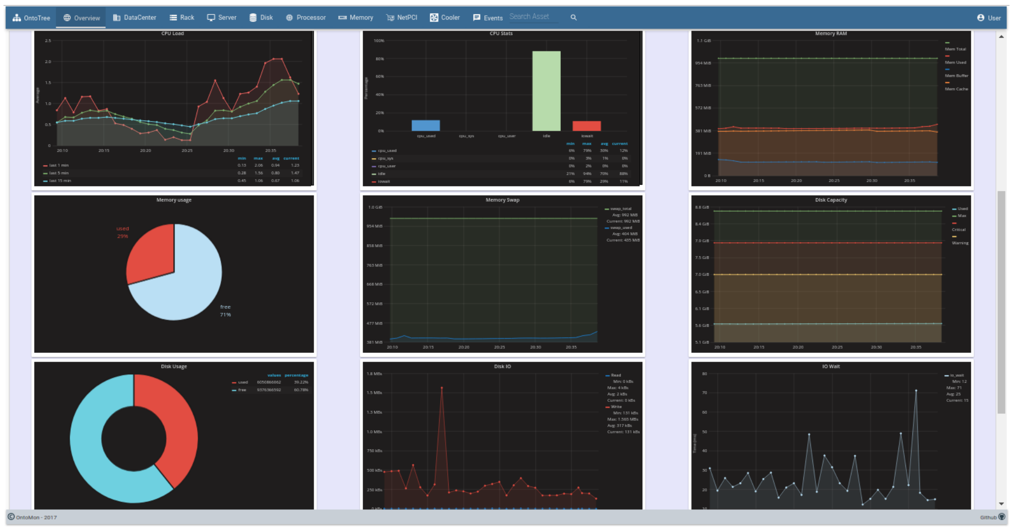


Figure 5.5: *OntoMon UI: graphical representation of performance metrics*

Last but not least, we needed to certify that the alerting mechanism is working as expected. Thus, we decided to simulate a real-world environment, where underperformance of software services or failures of physical devices are routine. We decided to impose heavy workload regarding `CPU`, `memory` and `disk` on one of the 3 virtual machines we are supervising, by using the native stress(1) Unix tool. As its name suggests, this is a lightweight tool that imposes certain types of computational stress upon Unix-like systems. Thereafter, we connected to the `icinga2-master1` host via `ssh` and executed the following commands as `root`. Our objective was to make the monitored target system operate under abnormal circumstances.

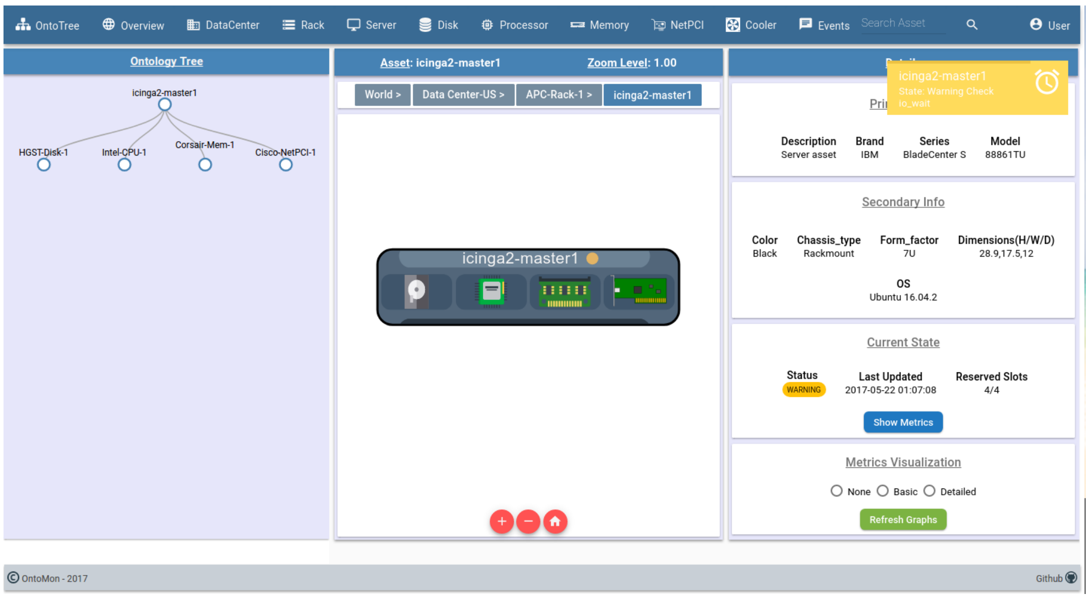
```
# spawn 8 workers spinning on sqroot() for 45 seconds
stress --quiet --cpu 8 --timeout 45

# spawn 8 workers spinning on malloc()/free() for 45 seconds
stress --quiet --vm 8 --vm-bytes 256M --timeout 45

# spawn 4 workers spinning on sync() and 4 workers on
    write()/unlink() for 45 seconds
stress --quiet --io 4 --hdd 4 --hdd-bytes 1G --timeout 45
```

Listing 5.2: *Command to stress hardware assets*

Since resource utilization, data traffic and latency times dramatically increase in both stress tests, we expect that current performance statistics collected from the server exceed the pre-defined threshold values that determine the state(`WARNING, CRITICAL`) of the `icinga2-master1` asset. Accordingly, we wait for corresponding real-time alerts in the User Interface of OntoMon, indicating that performance issues that have just occurred. We also anticipate that after the timeout of the `stress` test expires, the state of the `icinga2-master1` server asset will revert back to `OK`. As shown in the screenshots below, the UI we implemented successfully delivers the requisite notifications, facilitating the identification of performance bottlenecks of the monitored asset. Each notification is colored based on the current status of each asset, while the `indicator` element of the respective `SVG` gets automatically updated. Thus, administrators are timely informed about performance degradation of physical components and effectively guided on how to examine the related metrics, aiming at resolving the problem as soon as possible. Undeniably, this mechanism leads to reduced management times and deeper understanding of the underlying system that is under supervision. Finally, Figure 5.6 and Figure 5.7 demonstrate the performance lifecycle of the server and verify that the Web UI of OntoMon is fully functional according to our design specifications.



Asset: icinga2-master1 **Zoom Level**: 1.00

icinga2-master1
State: Warning Check
io_wait

Description	Brand	Series	Model
Server asset	IBM	BladeCenter S	88861TU

Secondary Info

Color	Chassis_type	Form_factor	Dimensions(H/W/D)
Black	Rackmount	7U	28.9,17.5,12

OS
Ubuntu 16.04.2

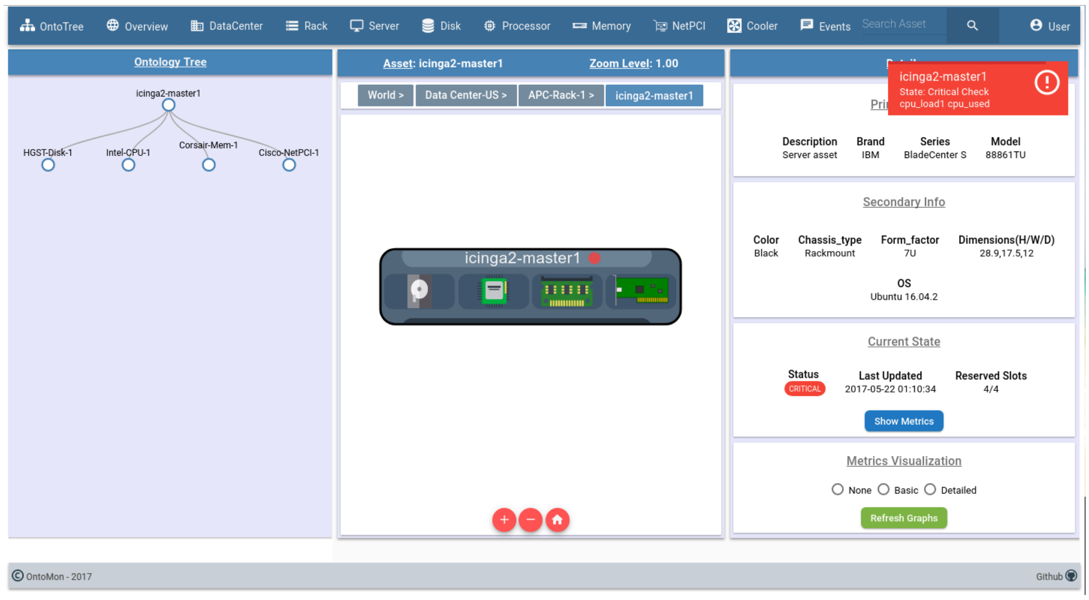
Current State

Status	Last Updated	Reserved Slots
WARNING	2017-05-22 01:07:08	4/4

Metrics Visualization

( ) None ( ) Basic ( ) Detailed

Figure 5.6: *OntoMon UI: notification of WARNING state*



Asset: icinga2-master1 **Zoom Level**: 1.00

icinga2-master1
State: Critical Check
cpu_load1 cpu_used

Description	Brand	Series	Model
Server asset	IBM	BladeCenter S	88861TU

Secondary Info

Color	Chassis_type	Form_factor	Dimensions(H/W/D)
Black	Rackmount	7U	28.9,17.5,12

OS
Ubuntu 16.04.2

Current State

Status	Last Updated	Reserved Slots
CRITICAL	2017-05-22 01:10:34	4/4

Metrics Visualization

( ) None ( ) Basic ( ) Detailed

Figure 5.7: *OntoMon UI: notification of CRITICAL state*

5.3 Software Defined Storage Monitoring

In order to reason about the correctness of our solution and verify that OntoMon is actually content agnostic, we decided to study a different use case, focusing on a software-defined target system. The main reason behind this choice was to showcase the versatility of our tool and prove that it is capable of extending monitoring on software clusters as well. At the same time, we would draw useful conclusions regarding the breadth of the target systems that our platform supports. Therefore, in this test case, we decided to test OntoMon upon a Ceph Cluster, that provides scalable and distributed object storage. This scenario can be regarded as a variation of the previous one, with the key difference that we now target software - and not physical - assets inside a Data Center. Of course, a Ceph Cluster requires disparate supervision and management compared to the physical infrastructure we monitored in the first test case. However, we followed analogous steps in order to observe and administrate the software-defined target system. As already stated in section 2.3, a Ceph Cluster comprises of multiple software components and daemons. For our testing purposes, it was sufficient to deploy a minimal 3-node Ceph Storage Cluster with 1 Monitor and 2 OSDs. As expected, we installed Ceph (Jewel v10.2.0) on the aforementioned 3-VM testbed environment, where all nodes can communicate with each other inside their private network. To accelerate the deployment process, we used the `ceph-deploy` tool of Ceph, since our needs did not call for any complex or detailed manual configuration. We executed `ceph-deploy` on the hypervisor node and granted current user with `root` privileges on the cluster nodes, in order to allow installation of `ceph packages` via `ssh`. Furthermore, since our Ceph cluster encloses only 2 Object Storage Daemons, we had to set its replication factor equal to 2, so that the Ceph Cluster could reach an `active+clean` health state in this topology.
Bearing in mind that different target systems are represented by different Ontologies, we had to generate an *Ontology.json* file that would accurately describe the hierarchy of entities inside the Ceph Cluster. The hypothesis upon which we rely our analysis is that the deployed Ceph Cluster "lives" inside a notional Data Center, that is capable of hosting multiple clusters. The key differentiation from the previous scenario lies in the type of assets that are enclosed in the Data Center, as we now ignore physical equipment and concentrate on ceph-related assets instead. Accordingly, the following

entities are introduced: **World, Data Centers, Clusters, Nodes, Ceph-Monitors** and **Ceph-OSDs**. Besides, we associated each of the aforementioned entities with separate `.svg` files, aiming at a more intuitive visualization inside the UI. The respective ontological description is shown in Figure 5.8. It is important to point out that we could have tested OntoMon on any multi-node software platform, and not only on a Ceph Cluster.

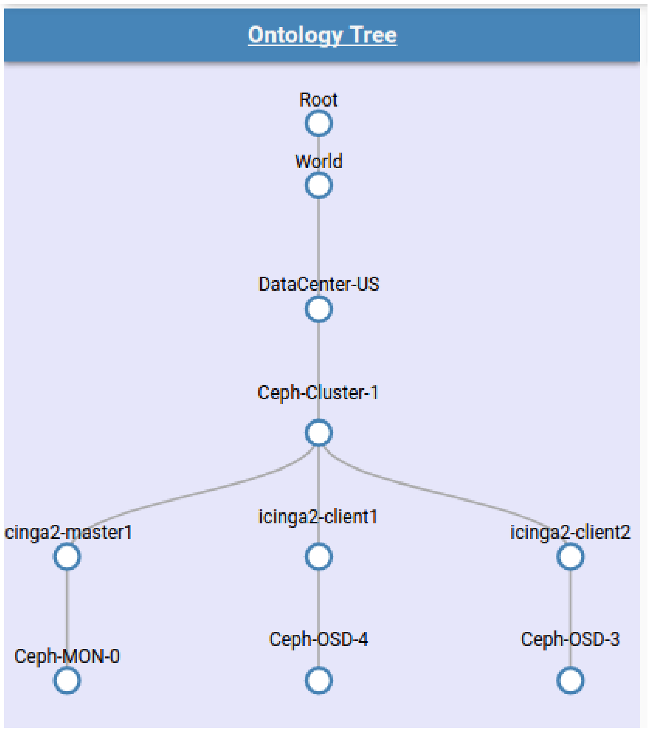


Figure 5.8: *OntoMon UI: Ontology Tree of software-defined cluster*

Our next concern was to regulate the monitoring environment in order to observe and thoroughly examine how the Ceph Cluster behaves. Admittedly, determining the correct metrics to monitor inside a Ceph Cluster is rather difficult even for experts in the context of today's complex and dynamic computer systems. Still, we had to make some assumptions and choose which metrics are necessary to monitor, while setting the right values of the thresholds for different scenarios. Subsequently, we implemented dedicated monitoring services to collect real-time performance data and perform calculations. During this process, we utilized the native `CLI tools` of Ceph that efficiently summarize the current status of the cluster and indicate possible issues. In particular, we combined multiple check commands and implemented combinatorial services. When operating in high-scale, it is important to provide administrators with an instant cluster-wide overview that outlines the general performance, rather than focusing on individual nodes of the cluster. Based on this principle, we initially concentrated on global metrics, such as:

- `Cluster health`: shows current overall health of the Ceph Cluster. If its value is other than `HEALTH_OK`, then the cluster is probably facing some issues.
- `Total & used capacity`: the storage availability of the Ceph Cluster. Near-full clusters are more likely to underperform.
- `Throughput`: how many operations are performed per second by the Ceph Cluster
- `Latency`: the average time interval between stimulation and response required by the OSDs to either *apply*(flush to disk) or *commit*(append to journal) an operation.

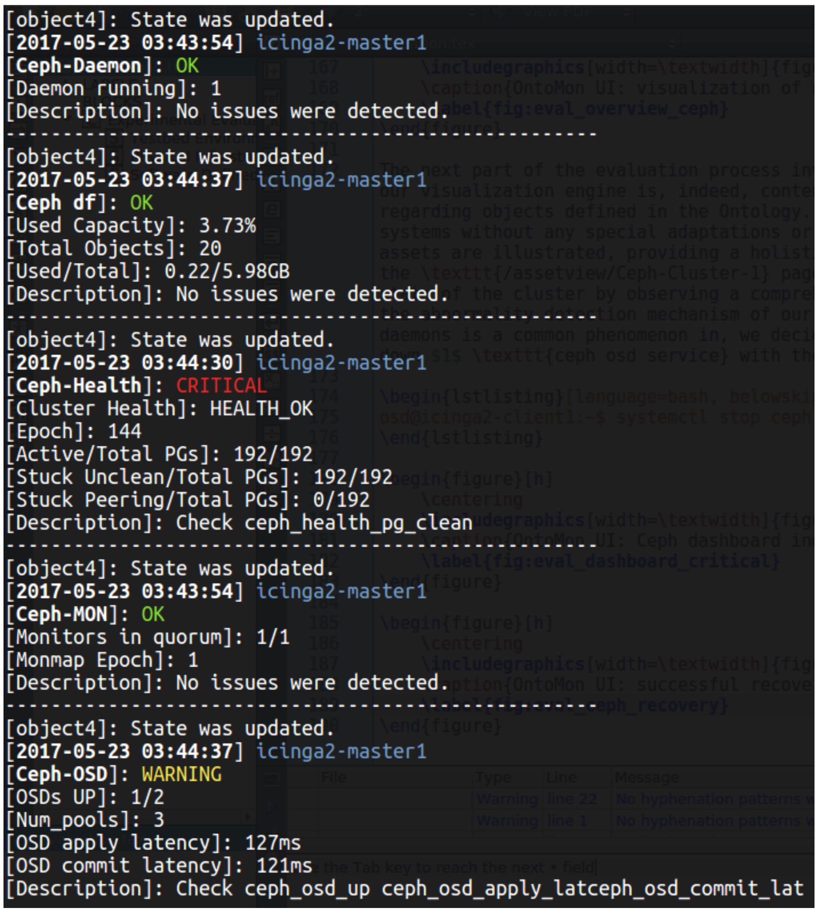
However, effective supervision of a Ceph Cluster also involves awareness of its basic building blocks, meaning the status of Monitors, OSDs, MDSs, Pools, Placement groups, etc. Therefore, we introduced additional coarse-grained check services that report critical asset-specific information and statistics:

- `UP, DOWN OSDs & Monitors`: availability of hosts running either OSD or Monitor daemons
- `IN, OUT OSDs & Monitors`: presence of specific OSD or Monitor inside the Ceph Cluster
- `Total and per pool objects`: the total objects stored in the cluster, along with their allocation in pools. Keeping the Ceph Cluster balanced in terms of storage load can lead to higher performance.
- `Throughput & IOPS per pool`: pool specific statistics that depict reads and writes in both `bytes` and operations per second.
- `State of Placement groups`: indicates how many placement groups are currently `active,degraded,stale,undersized,unclean` etc.
- `Total number of PGs`: this is a crucial value for every Ceph Cluster, greatly influencing performance.

Software Asset	Icinga Services	Metrics
Monitor	`check_ceph_daemon, check_ceph_mon`	`daemon_-state,monmap_-epoch,mon_-total,quorum_-in,quorum_out`
OSD	`check_ceph_daemon, check_ceph_osd`	`daemon_state,read_-bytes_sec,write_-bytes_sec,in_-osd,out_osd,up_-osd,down_osd,apply_-lat,commit_lat`
Cluster	`check_ceph_health, check_ceph_df`	`health_-status,epoch,num_-pgs,pg_-states,objects_per_-pool,total_-GB,total_used_-GB,raw_used_pct`

Table 5.2: *Performance checks for software assets*

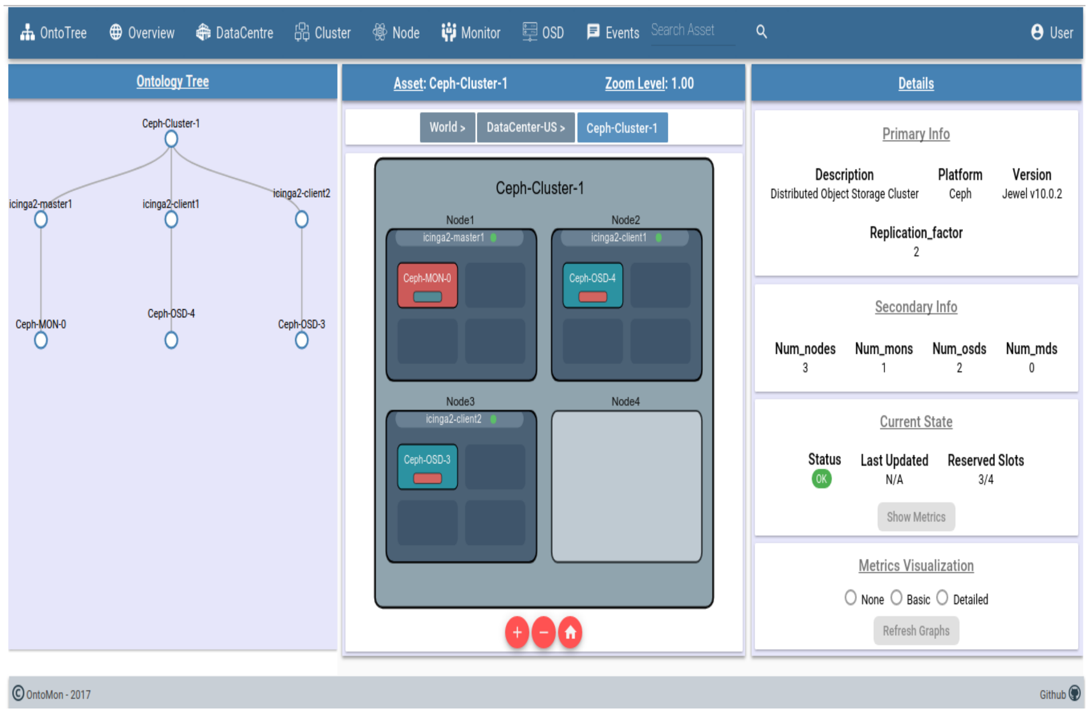
Again, acting as system integrators, we extended the implementation of the OntoMon Observer and adjusted the respective `Python` modules to serve the monitoring needs of the current software-defined target system. Firstly, we modified the `observer.py` module and added three command line arguments, `--mons,--osds` and `--time`, in order to specify the `FQDNs` of the respective Ceph cluster hosts and the time window for the retrieval of the performance metrics. Alongside, we implemented a `services_-sw.py` module based on the default `services.py` one, which encloses all respective handler functions for digesting time-series data and performing threshold checks. When these functions are ultimately called with the needed parameters, they perform time-series data processing, proceed to pre-defined threshold-based comparations and conclude on the current status of each monitored software asset. At the same time, we configured them to dynamically construct the *Update.json* object that holds the most recent metrics, state and notification guidelines and dispatch it to the OntoMon Server.



```
[object4]: State was updated.
[2017-05-23 03:43:54] icinga2-master1
[Ceph-Daemon]: OK
[Daemon running]: 1
[Description]: No issues were detected.
---------------------------------------------------
[object4]: State was updated.
[2017-05-23 03:44:37] icinga2-master1
[Ceph df]: OK
[Used Capacity]: 3.73%
[Total Objects]: 20
[Used/Total]: 0.22/5.98GB
[Description]: No issues were detected.
---------------------------------------------------
[object4]: State was updated.
[2017-05-23 03:44:30] icinga2-master1
[Ceph-Health]: CRITICAL
[Cluster Health]: HEALTH_OK
[Epoch]: 144
[Active/Total PGs]: 192/192
[Stuck Unclean/Total PGs]: 192/192
[Stuck Peering/Total PGs]: 0/192
[Description]: Check ceph_health_pg_clean
---------------------------------------------------
[object4]: State was updated.
[2017-05-23 03:43:54] icinga2-master1
[Ceph-MON]: OK
[Monitors in quorum]: 1/1
[Monmap Epoch]: 1
[Description]: No issues were detected.
---------------------------------------------------
[object4]: State was updated.
[2017-05-23 03:44:37] icinga2-master1
[Ceph-OSD]: WARNING
[OSDs UP]: 1/2
[Num_pools]: 3
[OSD apply latency]: 127ms
[OSD commit latency]: 121ms
[Description]: Check ceph_osd_up ceph_osd_apply_latceph_osd_commit_lat
```

Figure 5.9: *OntoMon Observer: monitoring software assets*

The next part of the evaluation process involves interaction with the Web UI of OntoMon. Initially, we can confirm that our visualization engine is, indeed, content-agnostic, as the exact same implementation delivers diverse optical results regarding objects defined in the Ontology. Meanwhile, it proves the capacity of our platform to integrate into various systems without any special adaptations or adjustments. As shown in Figure 5.10, our User Interface illustrates software-defined assets, providing a holistic overview of the monitored Ceph Cluster and its enclosed nodes. Navigating to the `/assetview/Ceph-Cluster-1` page of the web application allows administrators to gain insight into the current state of the cluster by observing a comprehensible real-time dashboard. In order to test the responsiveness of the Ceph dashboard we developed, we executed the most frequent operations that users and services perform upon a Ceph Cluster: `read` and `write` of data. As already mentioned, every transaction with regard to the Ceph Cluster is internally handled as an object operation in RADOS. Hence, we decided to use the native bench utility of `rados` to perform respective benchmark operations. In Figure 5.11 it is obvious that our dashboard is operating as expected, updating the performance graphs of `Throughput` and `Latency` on-the-fly, as fresh metrics arrive from the monitoring layer.



Primary Info

Description: Distributed Object Storage Cluster
Platform: Ceph
Version: Jewel v10.0.2
Replication_factor: 2

Secondary Info

Num_nodes: 3
Num_mons: 1
Num_osds: 2
Num_mds: 0

Current State

Status: OK
Last Updated: N/A
Reserved Slots: 3/4

Metrics Visualization

None Basic Detailed

Figure 5.10: *OntoMon UI: visualization of nested software assets*

```
# Write fixed size object to pool 'foo' for 60 sec
rados bench -p foo 60 write

# Sequentially read objects from pool 'foo' for 30 sec
rados bench -p foo 30 sec
```

Listing 5.3: *Benchmarking RADOS*

Finally, we had to verify whether the anomaly detection mechanism of our platform is functional for software assets, as well. Since malfunction of storage daemons is a common phenomenon in distributed storage clusters and large-scale deployments, we decided to reproduce an OSD failure. Thus, we connected to the `icinga2-client2` OSD node via `SSH` and manually stopped the respective 1 `ceph osd service` for a short period of time by running the following Ceph administration command:

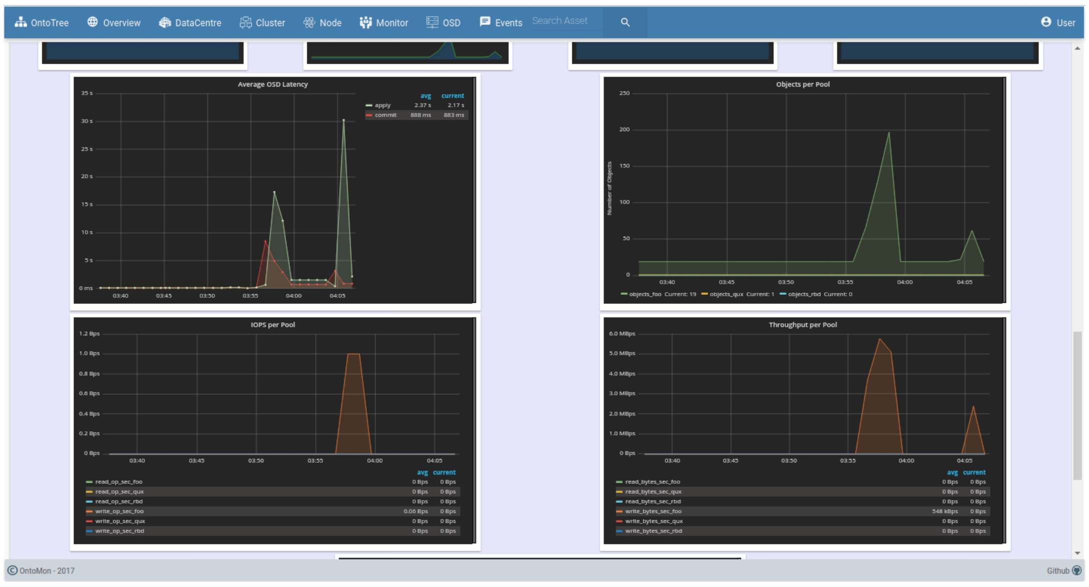
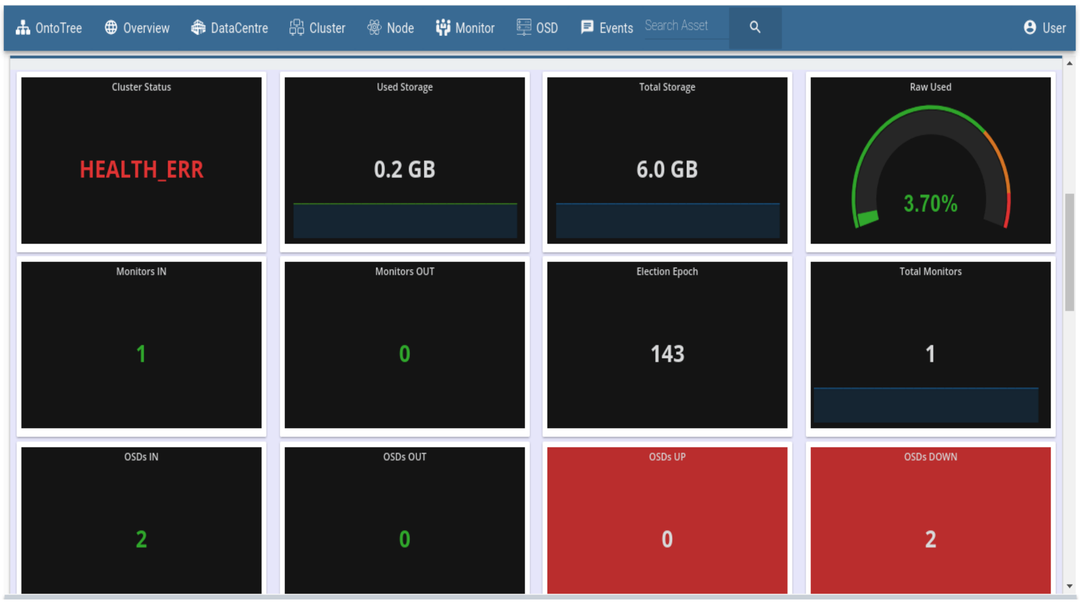

OntoTree
Overview
DataCentre
Cluster
Node
Monitor
OSD
Events
Search Asset
User
Average OSD Latency
avg current
apply 2.37 s 2.17 s
commit 888 ms 883 ms
Objects per Pool
Number of Objects
objects_foo Current: 19
objects_qux Current: 1
objects_rbd Current: 0
IOPS per Pool
avg current
read_op_sec_foo 0 Bps 0 Bps
read_op_sec_qux 0 Bps 0 Bps
read_op_sec_rbd 0 Bps 0 Bps
write_op_sec_foo 0.06 Bps 0 Bps
write_op_sec_qux 0 Bps 0 Bps
write_op_sec_rbd 0 Bps 0 Bps
Throughput per Pool
avg current
read_bytes_sec_foo 0 Bps 0 Bps
read_bytes_sec_qux 0 Bps 0 Bps
read_bytes_sec_rbd 0 Bps 0 Bps
write_bytes_sec_foo 548 kBps 0 Bps
write_bytes_sec_qux 0 Bps 0 Bps
write_bytes_sec_rbd 0 Bps 0 Bps
OntoMon - 2017
Github

Figure 5.11: *OntoMon UI: RADOS throughput and latency graphs*

```
osd@icingaClient1:~$ systemctl stop ceph-osd@4.service
```

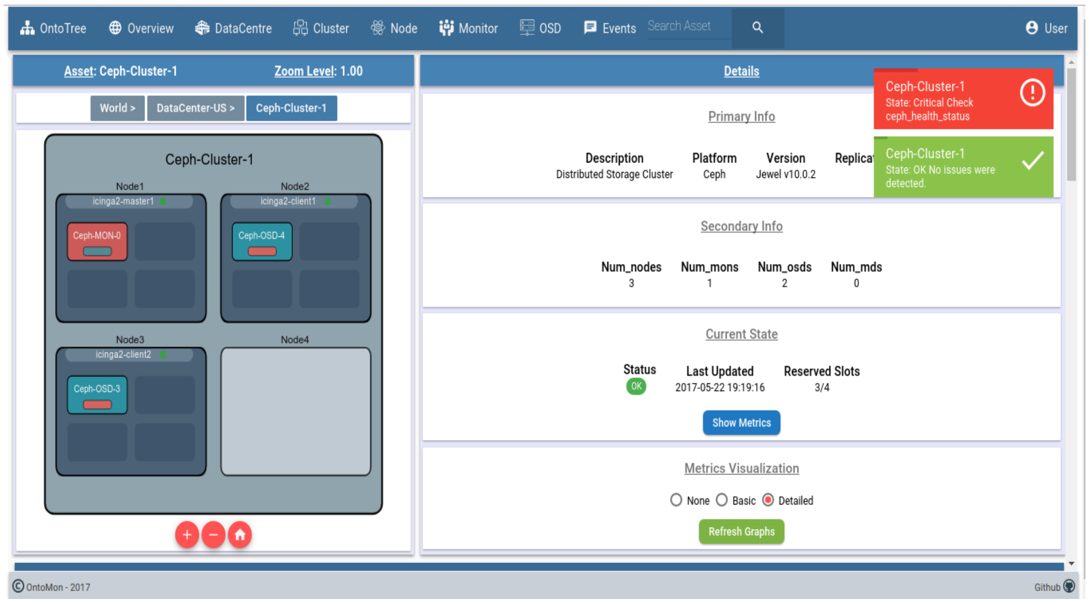
Listing 5.4: *Command to bring down OSD daemon*

Since we have deployed only 2 OSDs and ask for duplicate replicas of all objects across the nodes of the cluster, the awaited results involve a push notification in the Web UI of our platform, indicating that the Ceph Cluster is no longer in `HEALTH_OK` state. Ceph will try to rebalance and self-heal, but its state will remain unclean and all `Placement groups` will be stuck degraded. As soon as we restore the `ceph osd service` inside the `icinga2-client2` node and the needed relocations take place, the Ceph Cluster will recover and achieve an `active+clean` health state. The testing procedure analyzed above is evidently presented in the screenshots provided below. In Figure 5.12 we observe as end-users that the current state of our Ceph cluster is `HEALTH_ERR`, since at least 1 OSD daemon is down. Meanwhile, in Figure 5.13, we observe the push notifications delivered by OntoMon's mechanism regarding the degradation and recovery of the Ceph cluster.



Cluster Status: HEALTH_ERR
Used Storage: 0.2 GB
Total Storage: 6.0 GB
Raw Used: 3.70%
Monitors IN: 1
Monitors OUT: 0
Election Epoch: 143
Total Monitors: 1
OSDs IN: 2
OSDs OUT: 0
OSDs UP: 0
OSDs DOWN: 2

Figure 5.12: *OntoMon UI: Ceph dashboard indicating Cluster error*



Asset: Ceph-Cluster-1 Zoom Level: 1.00

Details

Primary Info

Description	Platform	Version	Replica
Distributed Storage Cluster	Ceph	Jewel v10.0.2	

Secondary Info

Num_nodes	Num_mons	Num_osds	Num_mds
3	1	2	0

Current State

Status	Last Updated	Reserved Slots
OK	2017-05-22 19:19:16	3/4

Metrics Visualization

None Basic [✓ Detailed]

Ceph-Cluster-1
State: Critical Check ceph_health_status

Ceph-Cluster-1
State: OK No issues were detected.

Figure 5.13: *OntoMon UI: notifying successful recovery of Cluster*

6 Conclusion

In this chapter we provide a short synopsis of our work, reviewing some focal points of the development process. Next, we list some future extensions and additional features that could make OntoMon even more powerful and robust, once implemented and integrated into our platform.

6.1 Concluding Remarks

The goal of this thesis was to study current DCIM solutions and familiarize with the concepts of monitoring and visualization in large-scale computer systems. In the challenging times of excessive data growth, we consider real-time system instrumentation and performance analysis as a must-have service for every IT-related platform. The initiative for our work was the lack of advanced open-source DCIM tools that can effectively supervise and manage both software- and hardware-defined systems. Thus, we decided to develop our own monitoring and visualization tool, named *OntoMon*.

The core feature of OntoMon lies in the fact that it requires a standard-formatted Ontology as input, describing the target system that is going to be monitored. The reason behind this foundation is to broaden the range of supported platforms and grant versatility. Besides, OntoMon is a multi-layered tool that combines various scalable open-source technologies to build a solid and well-rounded stack. Specifically, we employed Icinga as the collector agent, InfluxDB as the time series storage backend, Grafana as the graph composer and Angular to develop a dynamic content-agnostic Web User Interface. To accomplish our goals of proactive monitoring, as well as real-time alerting, we also introduced several middleware components that facilitate cross-layer communications and implement inner APIs.

The main candidate target systems for our platform are large scale, multi-node, distributed environments. By using OntoMon, administrators can observe both physical and software infrastructure of Data Centers, directly isolate specific assets and detect performance bottlenecks or abnormalities in running services. Visualization of assets offers an instant and holistic overview of the current deployment, while observation of real-time graphs can lead to pattern discovery and deeper understanding of system behavior. Meanwhile, thanks to our custom alerting protocol, the OntoMon WebUI interacts with the OntoMon Server and timely delivers push notifications regarding the current state of the target system or any recent events of interest. The core logic of the alerting mechanism is implemented in a `Python` daemon, the OntoMon Observer.

As a proof of concept for our platform, we studied two different target systems that were described in respective Ontologies. In the first use case, we used OntoMon to monitor the status and performance of hardware assets enclosed inside a notional Data Center, whose structure and topology closely resemble a real-world one. Conversely, in the second use case we decided to monitor a software-defined storage platform and, thus, deployed a simple, yet functional, 3-node Ceph Cluster. In both scenarios we observed satisfactory results in terms of real-time monitoring, visualization and alerting of the respective target systems. In particular, OntoMon efficiently reacted to every stimulation we provided, delivering meaningful insight into the state of both target systems in real time. Thereafter, the proposed design and implementation can be considered successful, as our platform had the expected behavior. Even though the objective of this thesis was not the implementation of a production-ready monitoring and visualization platform, we argue that the core logic and principles proposed in our framework certainly provide a sound basis for the development of versatile and well-rounded IT infrastructure monitoring solutions in the future.

6.2 Future Work

Even though OntoMon supports a wide variety of features in both visualization and monitoring layers, several enhancements regarding its current capabilities could be discussed, as well as additional. In addition, aiming at constantly improving the quality of services, we demonstrate our future plans for OntoMon:

- **Stricter Ontology definition and validation:** we could introduce a more detailed schema for the ontological description of the target system, in the sense that the currently employed schema imposes very few restrictions and rules. Since we introduced `JSON`-formatted Ontologies, we could leverage this issue by using the promising `JSON Schema` [1], a vocabulary that describes existing data format with annotations and performs complete structural validation of JSON documents. This `schema` is also capable of extending already defined schemas available on the Web, while it would also simplify the verification of syntax and the incorporation of metadata.
- **Add multiple storage backends:** instead of only supporting InfluxDB as the storage backend for the time-series monitoring data, we could also integrate storage platforms, such as TSTB, Prometheus and Elasticsearch, in the middle layer and let the end-user select the one he prefers. Of course, this implies the parallel extension of the OntoMon Observer component and the implementation of respective handler functions.
- **Extend monitoring capabilities:** although OntoMon already covers the most crucial checks regarding supervised assets, we could perform low-level, specialized checks and maybe correlate software- and hardware-based instrumentation to extract useful information about the performance and the inter-dependencies of the target system.
- **Implementation of Action Chain:** currently OntoMon offers monitoring and visualization of Data Center assets, along with an alerting mechanism that informs administrators when issues or state updates occur. We could extend the functionality of OntoMon to actively provision the behavior of the target system, by training it with artificial intelligence techniques that would trigger predefined system administration actions. This could lead to a fully-automated management of the underlying system, of course under human supervision and coordination. For example, when a disk suffers from slow response times and high

[1] `http://json-schema.org/`

temperature, OntoMon could automatically unmount it from the filesystem, so that it can be examined or replaced. Similarly, in case of network failure or expiry of request timeouts, a reboot of a server or router could be automatically triggered to resolve the issue.

- **Security:** for the scope of this thesis, security of communications and encryption of transferred data were low priorities in our design. However, in order to secure sensitive performance metrics against interceptions or interference from third-parties, respective services should be implemented to guarantee integrity. In addition, we plan to introduce a strong authentication system inside the OntoMon Web Interface, so that only verified users are granted access rights.
- **Enhanced log manipulation:** a key feature of modern platforms is the efficient handling of large and complex log files. Bearing in mind that OntoMon supports a wide variety of software-defined target systems, we could integrate a management tool like Logstash [2] in order to formalize and digest logs in a streaming fashion. `Logstash` can pull logs from various sources(e.g. web applications, data stores, etc.) and supports aggregation, parsing and transformation operations upon any type of log data. It is also compatible with Grafana, the graph composer we already have employed.

[2]`https://www.elastic.co/products/logstash`

Bibliography

[1] OpenDCIM project
http://www.opendcim.org/

[2] Racktables project
https://github.com/RackTables/racktables/

[3] Netbox project
https://github.com/digitalocean/netbox/

[4] Tendrl project
https://github.com/Tendrl/

[5] Thomas R.Gruber: *Toward Principles for the Design of Ontologies Used for Knowledge Sharing* Stanford Knowledge Systems Laboratory, Palo Alto, CA 94304, 1993
https://dl.acm.org/citation.cfm?id=219701

[6] Ling Liu and M. Tamer Özsu: *Encyclopedia of Database Systems* Springer Publishing Company, 2009
https://www.springer.com/us/book/9780387355443

[7] Ontologies in Information Science
https://en.wikipedia.org/wiki/Ontology_(information_science)

[8] World Wide Web Consortium-The Semantic Web *Semantic Web*
https://www.w3.org/standards/semanticweb/

[9] B. Chandrasekaran, John R. Josephson, V. Richard Benjamins: *What Are Ontologies, and Why Do We Need Them?* IEEE Educational Activities Department, Ohio State University, 1999
`http://dx.doi.org/10.1109/5254.747902`

[10] Ben Liu, Hejie Chen, Wei He: *Deriving User Interface from Ontologies: A Model-based Approach* Department of Computer Science, Beijing Institute of Printing, China, 2005
`http://ieeexplore.ieee.org/abstract/document/1562946/?reload=true`

[11] Kleshchev Alexander, Gribova Valeriya: *From an ontology-oriented approach conception to User Interface development* International Journal "Information Theories & Applications", Vol.10, 2007
`https://www.semanticscholar.org/paper/From-an-Ontology-oriented-Approach-Conception-to-U-Alexander-Valeriya/36d000a11eefa2816f6fe6180d1ad6cb88689dd0`

[12] Paulheim, Heiko: *Ontology-based Modularization of User Interfaces* SAP Research CEC Darmstadt, Bleichstrasse 8, 64283 Darmstadt, Germany, 2009
`http://doi.acm.org/10.1145/1570433.1570439`

[13] April Adams, David J. Cappuccio, Jay E. Pultz, Neha Kumar, Tiny Haynes: *Critical Capabilities for Data Center Infrastructure Management Tools*, 2014
`https://www.gartner.com/doc/3240917/critical-capabilities-data-center-infrastructure`

[14] Data Center Infrastructure Management
`https://en.wikipedia.org/wiki/Data_center_infrastructure_management`

[15] L.A.Barroso, J.Clidaras, U.Hölzle: *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines* Morgan & Claypool Publishers, 2013 Editor: Mark D. Hill, University of Wisconsin, Madison
`http://dx.doi.org/10.2200/S00516ED2V01Y201306CAC024`

[16] J.Moore, J.Chase, K.Farkas, P.Ranganathan: *Data Center Workload Monitoring, Analysis, and Emulation* Eighth Workshop on Computer Architecture Evaluation using Commercial Workloads, 2005
`https://pdfs.semanticscholar.org/bda2/0125ce93bbffdc752894313194139b9c93d7.pdf`

[17] A. Memari, J.Vornberger, J.M.Gómez, W.Nebel: *A Data Center Simulation Framework Based on an Ontological Foundation* Proceedings of the 28th EnviroInfo Conference, Oldenburg, Germany, 2014 Springer International Publishing, 2016
`http://dx.doi.org/10.1007/978-3-319-23455-7_3`

[18] Icinga project
`https://www.icinga.com/`

[19] Nagios project
`https://www.nagios.org/`

[20] Weil, Sage A. and Brandt, Scott A. and Miller, Ethan L. and Long, Darrell D. E. and Maltzahn, Carlos *Ceph: A Scalable, High-performance Distributed File System* Berkeley, CA, USA, 2006
`http://dl.acm.org/citation.cfm?id=1298455.1298485`

[21] Ceph Distributed Data Store
`http://ceph.com/`

[22] Weil, Sage A. and Leung, Andrew W. and Brandt, Scott A. and Maltzahn, Carlos *RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters* Storage Systems Research Center, University of California, Santa Cruz, 2007
`http://doi.acm.org/10.1145/1374596.1374606`

[23] S.Weil, S.Brandt, E.Miller, C.Maltzahn *CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data* Storage Systems Research Center, University of California, Santa Cruz, 2006
`http://doi.acm.org/10.1145/1188455.1188582`

[24] Roberto De Prisco, Butler Lampson, Nancy Lynch *Revisiting the paxos algorithm* Theoretical Computer Science, Volume 243, Issues 1–2, 2000
`https://doi.org/10.1016/S0304-3975(00)00042-6`

[25] Ceph official docs-Architecture
`http://docs.ceph.com/docs/master/architecture/`

[26] Scalable Vector Graphics(SVG) 1.1 Web Standard
`https://www.w3.org/TR/SVG/`

[27] Angular web application framework
`https://angular.io/`

[28] W3C standards-Web Components
`https://www.w3.org/standards/techs/components#w3c_all`

[29] Express.js project
`https://expressjs.com/en/4x/api.html`