ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μηχανισμός Ανάνηψης Εγκατάστασης Εφαρμογών σε Περιβάλλον Υπολογιστικών Νεφών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΧΑΡΔΟΥΒΕΛΗ ΠΑΝΑΓΙΩΤΗ-ΙΑΣΟΝΑ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2018

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΈΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μηχανισμός Ανάνηψης Εγκατάστασης Εφαρμογών σε Περιβάλλον Υπολογιστικών Νεφών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΧΑΡΔΟΥΒΕΛΗ ΠΑΝΑΓΙΩΤΗ-ΙΑΣΟΝΑ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 29η Μαρτίου 2018.

.....	.....	.....
Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Επίκουρος Καθηγητής Ε.Μ.Π.	Δημήτριος Τσουμάκος Αναπληρωτής Καθηγητής Ιονίου Πανεπιστημίου

Αθήνα, Μάρτιος 2018

.....

Χαρδούβελης Παναγιώτης - Ιάσων

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Στις μέρες μας, όλο και περισσότερες επιχειρήσεις και προγραμματιστές στρέφονται στη τεχνολογία υπολογιστικών νεφών για την εγκατάσταση των εφαρμογών τους. Υπενοικιάζοντας πόρους σε προσιτές τιμές και κατασκευάζοντας εικονικές μηχανές είναι σε θέση να εγκαταστήσουν μέσα σε λίγα λεπτά τις εφαρμογές τους στο διαδίκτυο χωρίς να χρειάζεται να επενδύσουν στην αγορά και την εγκατάσταση υπολογιστικών πόρων. Για την αυτοματοποίηση της διαδικασίας εγκατάστασης διάφορα εργαλεία έχουν προταθεί, τα οποία όμως δε λαμβάνουν υπόψιν την ασταθή φύση του νέφους και τα παροδικά σφάλματα που μπορεί να κάνουν την εμφάνιση τους κατά την εγκατάσταση και να την οδηγήσουν σε αποτυχία. Η Αύρα είναι ένα εργαλείο εγκατάστασης εφαρμογών στο νέφος που μοντελοποιώντας το πρόβλημα της εγκατάστασης μιας εφαρμογής σαν τη διάσχιση ενός κατευθυνόμενου ακυκλικού γράφου, επιχειρεί να ξεπεράσει την εμφάνιση τέτοιων σφαλμάτων, μέσω της επανεκτέλεσης τμημάτων του γράφου που απέτυχαν. Βασική προϋπόθεση για την επιτυχία αυτής της προσέγγισης είναι διαδοχικές επανεκτελέσεις μιας αλληλουχίας ενεργειών να οδηγούν πάντα την εγκατάσταση στην ίδια κατάσταση. Εντούτοις, αυτό δεν ισχύει πάντα καθώς πολλές ενέργειες που πραγματοποιούνται κατά την εγκατάσταση τροποποιούν πόρους που σχετίζονται με το σύστημα αρχείων και τη μνήμη.

Σκοπός αυτής της εργασίας είναι η επέκταση του συστήματος της Αύρας, ώστε να υποστηρίζει την επανεκτέλεση ενεργειών που αλληλεπιδρούν με πόρους που σχετίζονται με τη μνήμη. Χρησιμοποιώντας το CRIU, ένα εργαλείο που επιτρέπει την αποθήκευση και επαναφορά της κατάστασης μιας εφαρμογής διασφαλίζουμε ότι πριν από την επανεκτέλεση τέτοιων ενεργειών η εγκατάσταση βρίσκεται στην κατάσταση που επιθυμούμε.

λέξεις κλειδιά: εγκατάσταση εφαρμογής στο νέφος, Αύρα, CRIU, ταυτοδυναμία μνήμης, παροδικά σφάλματα

Abstract

Nowadays, more and more businesses and developers are turning to cloud computing and the services it provides, as a means to deploy their applications. The cloud enables them to allocate Virtual Machines on top of which they deploy their applications within minutes in a pay as you go manner, without the need to invest in expensive hardware. For the automation the deployment process, several tools have been proposed, but they do not take into consideration, the unstable, error prone nature of the cloud and the transient errors that can occur, failing the deployment. Aura is a cloud application deployment tool that formulates the deployment as a Directed, Acyclic Graph traversal and re-executes the parts of the graph that failed, thus overcoming the limitations presented by transient errors. A core prerequisite for this approach to work is that the re-execution of a script always leads to the same results. However, not all deployment scripts guarantee idempotent execution, since most of them update file system and memory related resources.

The purpose of this thesis is the extension of the Aura system to support the re-execution of scripts that interact with memory related resources. By using CRIU, a tool that allows us to checkpoint and restore the state of a process, we ensure that the deployment process rolls back to the desired state before scheduling the re-execution of not memory idempotent scripts.

keywords: cloud application deployment, Aura, CRIU, memory idempotence, transient errors

Ευχαριστίες

Θα ήθελα να ευχαριστήσω την οικογένεια μου για την υποστήριξη που μου παρείχαν κατά την εκπόνηση αυτής της διπλωματικής εργασίας και καθ' όλη τη διάρκεια των σπουδών μου.

Ιάσονας Χαρδούβελης
Αθήνα, $26_{η}$ Μαρτίου 2018

Πίνακας περιεχομένων

1 Εισαγωγή

1.1 Η τεχνολογία των υπολογιστικών νεφών και τα πλεονεκτήματα που προσφέρει

Στις μέρες μας, η μεγάλη διαθεσιμότητα σε δίκτυα μεγάλης χωρητικότητας, σε χαμηλού κόστους υπολογιστές και συσκευές αποθήκευσης, σε συνδυασμό με τη μείωση των τιμών σε μεγάλη κλίμακα της ηλεκτρικής ενέργειας, του εύρους ζώνης δικτύου και του λογισμικού οδήγησε στην ανάπτυξη της τεχνολογίας των υπολογιστικών νεφών, γνωστά και με τον όρο cloud computing. Το cloud computing είναι ένα μοντέλο υποδομών και λογισμικού που επιτρέπει την πρόσβαση σε κοινόχρηστες ομάδες διαρθρώσιμων πόρων, οι οποίοι προμηθεύονται κυρίως μέσω διαδικτύου.

Υπάρχουν πολλοί λόγοι για τους οποίους εταιρίες και επιχειρήσεις στρέφονται στο cloud computing. Ένας βασικός παράγοντας είναι ότι επιτρέπει στους ενδιαφερόμενους να ελαχιστοποιήσουν το αρχικό κόστος επένδυσης για απόκτηση υποδομών και υπολογιστικών πόρων, υπενοικιάζοντας τους, αντί να διαθέτουν μέρος του κεφαλαίου τους για αγορά υποδομών, χώρου και συντήρηση. Ακόμα, η δυναμική φύση του cloud και η δυνατότητα ελαστικής κλιμάκωσης που προσφέρει δίνει τη δυνατότητα στους χρήστες να αυξομειώνουν τους υπολογιστικούς τους πόρους ανάλογα με τις ανάγκες τους.

Άλλα βασικά χαρακτηριστικά που καθιστούν τη τεχνολογία του cloud computing ελκυστική είναι η απόδοση και η αξιοπιστία που προσφέρει. Οι υπηρεσίες που προσφέρονται εκτελούνται σε ένα δίκτυο ασφαλών κέντρων δεδομένων, τα οποία συντηρούνται συνεχώς και αναβαθμίζονται με τη τελευταία γενιά μηχανημάτων. Ακόμα, κάθε επιχείρηση έχει ανάγκη για συστήματα που ανταπεξέρχονται σε περιπτώσεις σφαλμάτων και καταστροφών. Οι πάροχοι cloud υπηρεσιών εγγυώνται σε μεγάλο βαθμό την ομαλή λειτουργία των υποδομών τους, ενώ δημιουργώντας πολλαπλά αντίγραφα ασφαλείας προσφέρουν σε επιχειρήσεις και χρήστες τη δυνατότητα ανάκαμψης σε περίπτωση καταστροφής.

Πέρα όλων των παραπάνω, η τεχνολογία του cloud computing βρίσκει μεγάλη εφαρμογή και στον τομέα της εγκατάστασης εφαρμογών. Υπολογιστικοί πόροι και αποθηκευτικά μέσα παρέχονται πλήρως εικονοποιημένα με το πάτημα ενός κουμπιού επιτρέποντας στους χρήστες να δημιουργούν εικονικές μηχανές και αποθηκευτικά μέσα όπου μπορούν να εγκαθιστούν τις εφαρμογές τους πλήρως λειτουργικά μέσα σε λίγα λεπτά. Ακόμα, η διαδικασία ανάπτυξης μιας εφαρμογής διευκολύνεται από το γεγονός ότι οι προγραμματιστές μπορούν να δοκιμάζουν

οποιαδήποτε αλλαγή στον κώδικα τους σε πραγματικό χρόνο, εξαλείφοντας τυχόν σφάλματα που σε άλλες περιπτώσεις μπορεί να περνούσαν απαρατήρητα μέχρι τη τελική έκδοση της εφαρμογής.

1.2 Η διαδικασία εγκατάστασης μιας εφαρμογής στο νέφος

Η εγκατάσταση μιας εφαρμογής είναι μια διαδικασία σαφώς ορισμένων ενεργειών που πολλές φορές απαιτεί το συγχρονισμό μεταξύ διαφορετικών οντοτήτων. Πολύπλοκες ενέργειες όπως αρχικοποίηση συσκευών αποθήκευσης, εκτέλεση σεναρίων εγκατάστασης πακέτων λογισμικού και των εξαρτήσεων τους και επεξεργασία αρχείων ρυθμίσεων εκτελούνται με συγκεκριμένη σειρά για την επίτευξη της εγκατάστασης μιας διεργασίας. Η σαφώς ορισμένη αλληλουχία των ενεργειών αυτών σε συνδυασμό με την προγραμματιζόμενη διαχείριση πόρων μέσω κλήσεων API που προσφέρουν οι πάροχοι cloud υπηρεσιών, επιτρέπουν την αυτοματοποίηση της διαδικασίας εγκατάστασης μιας εφαρμογής στο cloud.

Για την αυτοματοποίηση και το συγχρονισμό των ενεργειών που αναφέρθηκαν παραπάνω έχουν προταθεί διάφορα συστήματα. Το Openstack Heat [1] αποτελεί υπηρεσία της πλατφόρμας λογισμικού του Openstack [2], ενός εργαλείου ανοικτού κώδικα που χρησιμοποιείται για τη διαχείριση cloud πόρων και τη δημιουργία εικονικών μηχανών. Αντίστοιχα, το AWS CloudFormation [3] είναι μια δωρεάν υπηρεσία για τους πελάτες της πλατφόρμας cloud υπηρεσιών AWS που προσφέρει η Amazon. Με τη βοήθεια των παραπάνω εργαλείων, ο χρήστης ορίζει τον σκελετό της εφαρμογής που επιθυμεί να εγκαταστήσει και των πόρων που αυτή απαιτεί, σε αρχεία περιγραφικής γλώσσας (πχ json [4], yaml [5]). Στη συνέχεια, τα εργαλεία αυτά επικοινωνούν με τον πάροχο cloud υπηρεσιών μέσω των κατάλληλων κλήσεων API, παρέχοντας τους πόρους που ο χρήστης όρισε για της ανάγκες της εφαρμογής του και ρυθμίζοντας τη σχέση μεταξύ αυτών. Άλλα εργαλεία όπως το Vagrant [6] και το Juju [7] λειτουργούν σαν εξωτερική σουίτα λογισμικού που επικοινωνεί με τον πάροχο cloud υποδομών ως client.

Όλα τα προαναφερθέντα εργαλεία μοντελοποιούν τη διαδικασία εγκατάστασης μιας εφαρμογής ως έναν κατευθυνόμενο ακυκλικό γράφο εξαρτήσεων (Directed Acyclic Graph), οι κόμβοι του οποίου αντιπροσωπεύουν τη κατάσταση της εγκατάστασης και των δομικών στοιχείων που την αποτελούν (modules) και οι ακμές την εκτέλεση ενός σεναρίου ενεργειών εγκατάστασης (script). Έτσι, ο συνολικός γράφος εκφράζει τη σειρά με την οποία πρέπει να εκτελεστούν τα απαραίτητα σενάρια εκτέλεσης, αλλά και τις εξαρτήσεις μεταξύ διαφορετικών καταστάσεων. Η εκτέλεση ενός script συνεπάγεται την αλλαγή της κατάστασης ενός module της εφαρμογής και κατ' επέκταση όλης της εγκατάστασης, ενώ επιτυχής εκτέλεση ενός script προϋποθέτει ότι η εγκατάσταση έχει φτάσει σε ένα συγκεκριμένο στάδιο πριν την έναρξη της εκτέλεσης του.

Οι εξαρτήσεις μεταξύ των script που εκτελούνται σε κάθε module είναι κομβικής σημασίας. Για παράδειγμα, έστω ότι ο χρήστης επιχειρεί να εγκαταστήσει μια εφαρμογή αποτελούμενη από δύο module, έναν εξυπηρετητή (server) και μία βάση δεδομένων, με την οποία και επικοινωνεί για την αποθήκευση και ανάκτηση δεδομένων. Το script το οποίο θέτει σε λειτουργία τη διεργασία του server πρέπει να εκτελεστεί εφόσον η βάση δεδομένων βρίσκεται σε λειτουργία και έχουν πραγματοποιηθεί οι απαραίτητες ρυθμίσεις, συνήθως με την τροποποίηση των απαραίτητων αρχείων ρυθμίσεων, ώστε ο server να μπορεί να επικοινωνήσει με τη βάση. Πρόωρη εκτέλεση του συγκεκριμένου script θα οδηγήσει σε αποτυχία της εγκατάστασης της εφαρμογής απαιτώντας ανθρώπινη παρέμβαση για διόρθωση των λαθών.

Βασισμένα σε αυτό το μοντέλο, τα εργαλεία που αναφέρθηκαν επικοινωνούν αρχικά με τον πάροχο cloud υπηρεσιών για την παροχή των απαραίτητων πόρων και στη συνέχεια διατρέχουν το γράφο εξαρτήσεων της διαδικασίας εγκατάστασης, εκτελώντας τα απαραίτητα script στα αντίστοιχα εικονικά μηχανήματα.

Η επιτυχής ολοκλήρωση της διαδικασίας εγκατάστασης προϋποθέτει την επιτυχή εκτέλεση όλων των script που ορίζονται από το χρήστη. Εντούτοις, τα προαναφερθέντα εργαλεία με τη προσέγγιση που υλοποιούν δε λαμβάνουν υπ’όψιν την επιρρεπή σε σφάλματα φύση του cloud. Ιδανικά όλα τα script που δρομολογούνται για εκτέλεση για την εγκατάσταση μιας εφαρμογής εκτελούνται με επιτυχία, όμως στη πραγματικότητα η πολυπλοκότητα που εισάγει το λογισμικό εικονοποίησης συχνά οδηγεί σε παροδικές αποτυχίες. Αποτυχία συνδεσιμότητας, προσωρινά μη διαθέσιμες υπηρεσίες δικτύου (πχ DNS), μη σωστά προσαρτημένα συστήματα αρχείων είναι κάποια από τα προβλήματα που εμφανίζονται για μικρό χρονικό διάστημα και ύστερα εξαφανίζονται. Παρ’όλη την αβλαβή φύση τους λόγω της παροδικότητας τους, είναι ικανά να οδηγήσουν σε αποτυχία την εκτέλεση ενός script και κατ’ επέκταση όλης της εγκατάστασης. Για παράδειγμα, ας υποθέσουμε ότι ένα script αναλαμβάνει την εγκατάσταση μιας εφαρμογής μέσω ενός συστήματος διαχείρισης και εγκατάστασης πακέτων λογισμικού (πχ apt) και κατά την εκτέλεση του συμβαίνει ένα σφάλμα δικτύου. Το script θα τερματίσει τη λειτουργία του εσφαλμένα, καθώς τα απαραίτητα πακέτα δεν θα αποκτηθούν, οδηγώντας τη διαδικασία εγκατάστασης σε αποτυχία.

Τέτοια συμβάντα έχουν απαγορευτικό κόστος τόσο χρονικά όσο και στον προϋπολογισμό μιας επιχείρησης, ενώ ανεκμετάλλευτοι υπολογιστικοί πόροι που έχουν δεσμευτεί για την εγκατάσταση πιθανότατα απαιτούν ανθρώπινη παρέμβαση για την αποδέσμευση ή επαναρύθμιση τους. Μια λύση που έχει προταθεί για το πρόβλημα των παροδικών σφαλμάτων είναι η επανεκτέλεση ενός script σε περίπτωση που αποτύχει, με το σκεπτικό ότι κάποια επανεκτέλεση, εν τέλει θα επιτύχει καθώς το παροδικό σφάλμα θα έχει εξαφανιστεί. Ένα εργαλείο εγκατάστασης

εφαρμογών στο cloud που επιχειρεί να υπερπηδήσει τον περιορισμό των περιοδικών σφαλμάτων μέσω της επανεκτέλεσης είναι η Αύρα [8].

Η Αύρα αναλαμβάνει να προσπελάσει το αρχείο περιγραφής της εφαρμογής, το οποίο ορίζει ο χρήστης σε περιγραφική γλώσσα, να επικοινωνήσει με τον πάροχο cloud υπηρεσιών για την δέσμευση των απαραίτητων πόρων και στη συνέχεια να δρομολογήσει για εκτέλεση τα script εγκατάστασης στα αντίστοιχα module της εφαρμογής. Ακόμα, επιβλέπει τη διαδικασία εγκατάστασης απομονώνοντας script που εκτελέστηκαν ανεπιτυχώς και επαναδρομολογώντας τα, μέχρις ότου ολοκληρωθούν με επιτυχία. Ο συγχρονισμός που απαιτείται μεταξύ των διαφορετικών module διασφαλίζεται μέσω ενός μηχανισμού ανταλλαγής μηνυμάτων, που είναι υπεύθυνος για το πέρασμα ορισμάτων απαραίτητων για την εκτέλεση κάποιων script. Εκτέλεση ενός script προϋποθέτει ότι όλα τα ορίσματα που αναμένει από άλλα script, που έχουν οριστεί στη περιγραφή της εφαρμογής, έχουν παραληφθεί και ότι η εγκατάσταση έχει φτάσει σε ένα συγκεκριμένο σημείο.

Βασική προϋπόθεση για η επανεκτέλεση ενός script είναι να οδηγεί την εγκατάσταση στην ίδια κατάσταση, η πιο απλά να έχει πάντα τις ίδιες παρενέργειες. Αυτή η ιδιότητα ονομάζεται ταυτοδυναμία (idempotence). Μια δεύτερη εκτέλεση ενός script που δεν έχει αυτή την ιδιότητα πιθανότατα δεν θα οδηγήσει στα επιθυμητά αποτελέσματα. Για παράδειγμα, έστω ένα απλό script που διαβάζει ένα αρχείο του συστήματος αρχείων και στη συνέχεια το διαγράφει. Σε περίπτωση που εκτελεστεί δεύτερη φορά θα τερματίσει εσφαλμένα, καθώς το αρχείο που θα προσπαθήσει να διαβάσει θα έχει διαγραφεί.

Τα περισσότερα scripts εγκατάστασης προκαλούν παρενέργειες σε επίπεδο συστήματος αρχείων (file system) καθώς η εγκατάσταση διαφορετικών πακέτων λογισμικού, οι ρυθμίσεις και η διασύνδεση μεταξύ των προϋποθέτει την τροποποίηση των αντίστοιχων αρχείων ρυθμίσεων. Παρόλα παρενέργειες συναντώται και σε επίπεδο μνήμης, καθώς πολλές εφαρμογές στηρίζονται σχεδόν εξ ολοκλήρου στη μνήμη για τη λειτουργία τους. Επανεκτέλεση script που αλληλεπιδρούν με τη μνήμη ενός module πιθανόν να μην έχει το επιθυμητό αποτέλεσμα καθώς η κατάσταση της μνήμης ενδέχεται να έχει μεταβληθεί σε προηγούμενη, αποτυχημένη εκτέλεση του εν λόγω script. Χαρακτηριστικό παράδειγμα τέτοιου είδους εφαρμογών, αποτελούν εφαρμογές αποθήκευσης ζευγών κλειδιού-τιμής στη μνήμη και χρησιμοποιούνται ευρέως ως συστήματα κρυφής μνήμης (cache) βελτιώνοντας την απόδοση πληθώρας συστημάτων, καθώς αποθηκεύοντας δεδομένα στη μνήμη και όχι στο σύστημα αρχείων, προσφέρουν γρηγορότερη ανταπόκριση σε αιτήματα σε σχέση με μια βάση δεδομένων.

1.3 Αντικείμενο διπλωματικής εργασίας

Αντικείμενο αυτής της διπλωματικής εργασίας είναι η επέκταση του συστήματος της Αύρας ώστε να υποστηρίζει την επανεκτέλεση script που συνδιαλέγονται και μεταβάλουν τη μνήμη μιας εφαρμογής, μέσω ενός μηχανισμού αποθήκευσης και επαναφοράς της κατάστασης της διαδικασίας εγκατάστασης. Πιο συγκεκριμένα, script τα οποία επηρεάζουν τη κατάσταση της μνήμης ενός module αναγνωρίζονται και πριν τη δρομολόγηση της εκτέλεσης τους παράγεται ένα στιγμιότυπο των διεργασιών η μνήμη των οποίων πρόκειται να μεταβληθεί αλλά και της κατάστασης όλων των module που απαρτίζουν την εφαρμογή (πχ ποια script έχουν εκτελεστεί μέχρι εκείνη τη στιγμή). Στη συνέχεια, σε περίπτωση αποτυχίας της εκτέλεσης η διαδικασία εγκατάστασης παύεται (σταματάει η δρομολόγηση επόμενων script) και μπαίνει σε λειτουργία ο μηχανισμός επαναφοράς. Κατά την επαναφορά, η διεργασία της οποίας η κατάσταση αποθηκεύτηκε τερματίζεται και επανεκκινείται από το αποθηκευμένο στιγμιότυπο και η εγκατάσταση επανέρχεται στο στάδιο που βρισκότανε πριν τη δρομολόγηση του αποτυχημένου script, το οποίο εκτελείται ξανά.

Για την αποθήκευση και την επαναφορά μιας διεργασίας χρησιμοποιείται το εργαλείο CRIU [9]. Το CRIU είναι ένα εργαλείο του λειτουργικού συστήματος Linux που προσφέρει τη δυνατότητα παγώματος (checkpoint) και επαναφοράς (restore) εφαρμογών σε χώρο χρήστη. Στη διαδικασία του checkpoint η κατάσταση μιας εφαρμογής, δηλαδή ο εικονικός της χώρος διευθύνσεων, περιγραφητές αρχείων, sockets και άλλοι βασικοί πόροι, σώζεται σε μια ομάδα αρχείων στο δίσκο. Ο χρήστης μπορεί να χρησιμοποιήσει αυτά τα αρχεία ώστε να επαναφέρει την εφαρμογή στην κατάσταση που βρίσκονταν ακριβώς όταν εκτελέστηκε το checkpoint. Με αυτό τον τρόπο εγγυόμαστε ότι παρενέργειες στην κατάσταση της μνήμης μιας εφαρμογής που προήλθαν από την αποτυχημένη εκτέλεση ενός script αναιρούνται και μπορεί να εκτελεστεί ξανά.

1.4 Διάρθρωση της διπλωματικής εργασίας

Η παρούσα διπλωματική εργασία οργανώνεται ως εξής:

Κεφάλαιο 2: Σε αυτό το κεφάλαιο γίνεται μια εκτενής ανάλυση του συστήματος Αύρα το οποίο και επεκτείναμε και του CRIU, ώστε ο αναγνώστης να καταλάβει το τρόπο λειτουργίας τους.

Κεφάλαιο 3: Σε αυτό το κεφάλαιο διατυπώνεται λεπτομερώς το πρόβλημα που κληθήκαμε να αντιμετωπίσουμε, τα βήματα που ακολουθήθηκαν αλλά και τα προβλήματα που συναντήσαμε κατά την υλοποίηση. Πιο συγκεκριμένα, περιγράφεται ο τρόπος υλοποίησης των διαδικασιών αποθήκευσης και επαναφοράς της κατάστασης ενός και περισσότερων module, οι αναγκαίες επεκτάσεις που πραγματοποιήθηκαν στο σύστημα της Αύρας ενώ παραθέτεται και ένα εκτενές σενάριο χρήσης για κατανόηση του συνολικού τρόπου λειτουργίας του συστήματος. Τέλος, παρουσιάζεται ένα εναλλακτικό σενάριο χρήσης που ξεφεύγει από την έννοια της εγκατάστασης μιας εφαρμογής και αφορά τη ζωντανή μεταφορά μιας εφαρμογής σε περιβάλλον μεγαλύτερων υπολογιστικών δυνατοτήτων

Κεφάλαιο 4: Σε αυτή την ενότητα περιγράφονται τα πειράματα που πραγματοποιήθηκαν για την αξιολόγηση του συστήματος και παρουσιάζονται τα αποτελέσματα αυτών σε μια σειρά γραφημάτων.

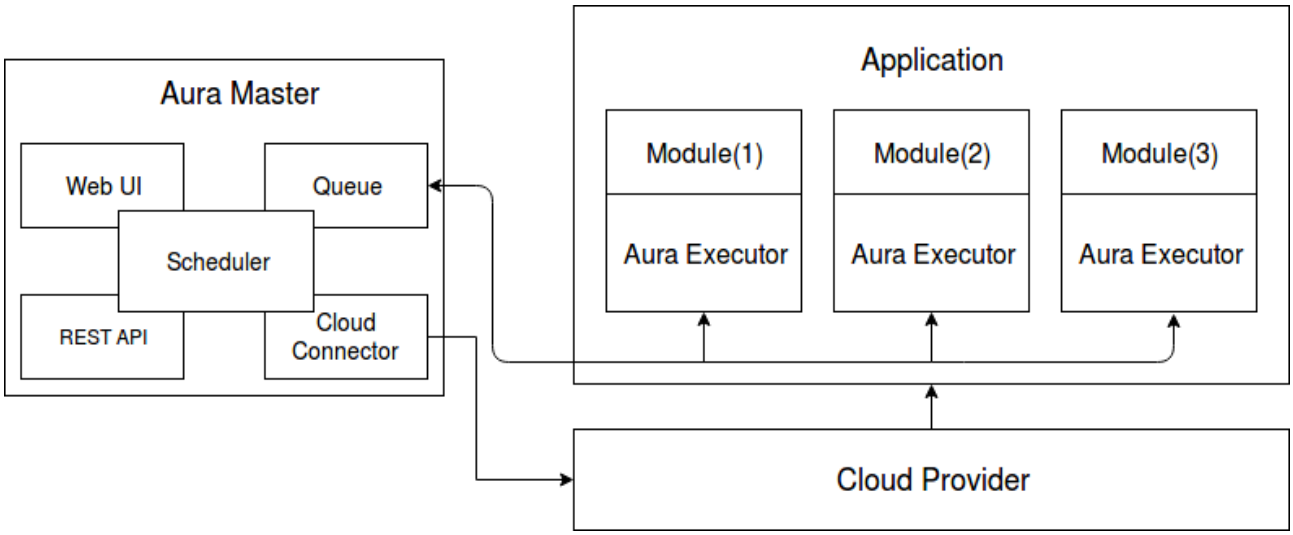
Κεφάλαιο 5: Εδώ παρουσιάζονται τα συμπεράσματα στα οποία καταλήξαμε.

2 Υπόβαθρο

2.1 Το σύστημα Αύρα

Η Αύρα είναι ένα εργαλείο εγκατάστασης εφαρμογών στην πλατφόρμα παροχής cloud υπηρεσιών του Openstack που προσφέρει ανάνηψη από παροδικά σφάλματα. Στο παρόν κεφάλαιο θα εμβαθύνουμε στον μηχανισμό της Αύρας και στο σύστημα ανάνηψης παρουσιάζοντας τη διαδικασία της εγκατάστασης μιας εφαρμογής στο σύννεφο.

2.1.1 Αρχιτεκτονική του Συστήματος της Αύρας



Εικόνα 2-1: Η αρχιτεκτονική του συστήματος Αύρα

Στο παραπάνω διάγραμμα παρουσιάζουμε την αρχιτεκτονική της Αύρας, η οποία μπορεί να χωριστεί σε δύο μέρη: Στον Αύρα Master και τους Αύρα Executors. Ο Aura Master, συνήθως εγκατεστημένος σε μια εικονική μηχανή στο cloud είναι υπεύθυνος για τη διαδικασία εγκατάστασης της εφαρμογής. Το application αποτελείται από ξεχωριστά module, που χωρίς βλάβη της γενικότητας και για λόγους ευκολίας μπορούμε να υποθέσουμε ότι το καθένα από αυτά εγκαθίσταται σε μια ξεχωριστή εικονική μηχανή. Ως module λογισμικού ορίζουμε ένα συστατικό στοιχείο μιας μεγαλύτερης εφαρμογής, όπως ένα πακέτο λογισμικού, διαδικτυακή υπηρεσία ή πόρο, που περιλαμβάνει ένα σύνολο συναρτήσεων και δεδομένων και μπορεί να εγκατασταθεί με την εκτέλεση ενός συνόλου σεναρίων εγκατάστασης (configuration scripts). Ο cloud provider,

αναπαριστά το πρωτόκολλο επικοινωνίας με τον πάροχο, υπεύθυνο για την ανάθεση και διαχείριση των πόρων.

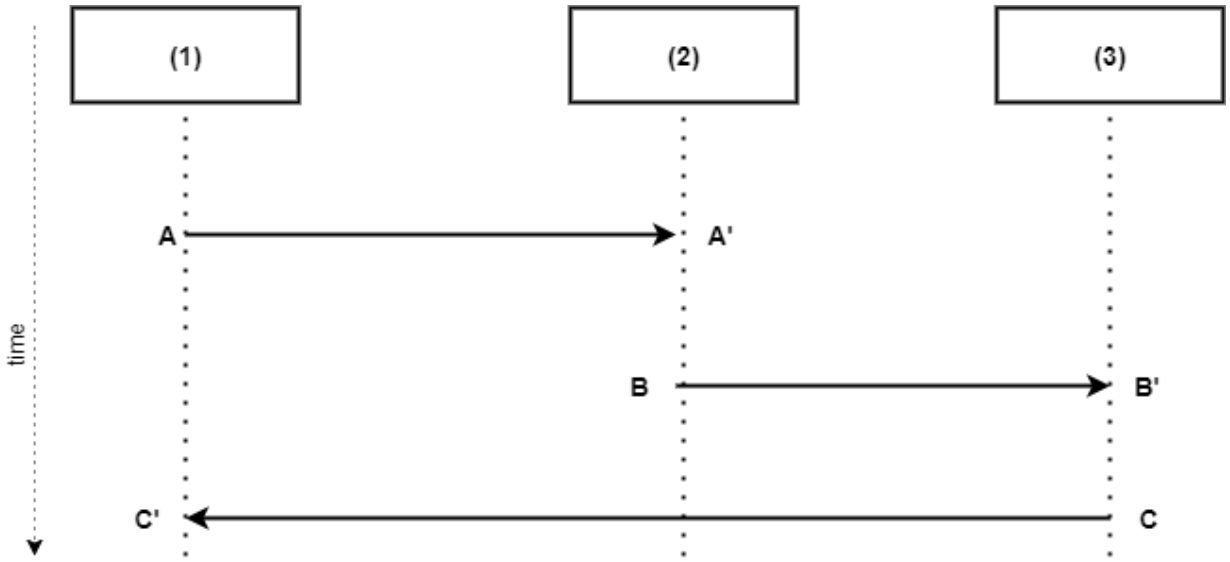
Εκτός από τον Αύρα Master, σημαντικό ρόλο παίζουν και οι εκτελεστές (Αύρα Executors), οι οποίοι συνδέονται στο κάθε module και είναι υπεύθυνοι για την εκτέλεση των απαραίτητων script εγκατάστασης στην αντίστοιχη εικονική μηχανή. Από την άλλη ο Αύρα Master λειτουργεί ως συντονιστής. Ο master επικοινωνεί με τον cloud provider μέσω του συντονιστή νέφους (cloud connector) ο οποίος είναι υπεύθυνος για την μετατροπή των υψηλού επιπέδου εντολών παροχής πόρων (π.χ. δημιουργία νέας εικονικής μηχανής) σε εντολές που υποστηρίζονται από τον πάροχο υπηρεσιών. Η ουρά (queue) αντιπροσωπεύει το μηχανισμό επικοινωνίας μεταξύ των διαφορετικών εκτελεστών καθώς όλα τα μηνύματα επικοινωνίας μεταξύ των module, απαραίτητα για την διαδικασία εγκατάστασης δημοσιεύονται εκεί. Οι εκτελεστές, παρακολουθούν συγκεκριμένες ουρές και καταναλώνουν μηνύματα που προορίζονται γι' αυτούς. Τέλος, ο δρομολογητής (scheduler) είναι υπεύθυνος για την παρακολούθηση και το συντονισμό σε πραγματικό χρόνο της συνολικής διαδικασίας εγκατάστασης.

2.1.2 Ο μηχανισμός ανταλλαγής μηνυμάτων ως μέσο συγχρονισμού

Για την επιτυχή αυτοματοποίηση της διαδικασίας εγκατάστασης μιας εφαρμογής στο cloud απαιτείται ο σωστός συγχρονισμός των ενεργειών και η επικοινωνία μεταξύ των modules. Αυτό επιτυγχάνεται με την ανταλλαγή μηνυμάτων, που αναφέραμε παραπάνω. Τα μηνύματα αυτά μπορεί να περιέχουν χρήσιμη πληροφορία, όπως διαδικτυακές διευθύνσεις (IP addresses), πόρτες δικτύου κλπ, για την εκτέλεση των script για τα οποία προορίζονται ή απλά να αποτελούν ενημερωτικά μηνύματα, όπου ο αποστολέας ενημερώνει ότι έφτασε σε κάποιο επιθυμητό σημείο της εκτέλεσης του με επιτυχία. Όταν ένα script κάποιου module περιμένει ένα μήνυμα ως είσοδο, παύει την εκτέλεση του μέχρι αυτό να ληφθεί επιτυχώς. Αντιθέτως, για την αποστολή ενός μηνύματος το υπεύθυνο module το δημοσιεύει στο μηχανισμό ανταλλαγής μηνυμάτων (στην περίπτωση μας μια ουρά ανταλλαγής μηνυμάτων στην οποία θα αναφερθούμε στη συνέχεια), με τη μορφή {παραλήπτης : μήνυμα} και συνεχίζει κανονικά την εκτέλεση του.

Στο παράδειγμα της εικόνας 2-2 παρουσιάζεται το χρονοδιάγραμμα της εγκατάστασης μιας εφαρμογής αποτελούμενη από τρία modules. Κατά τη διάρκεια της διαδικασίας, κάθε module εκτελεί τα δικά του σενάρια εγκατάστασης (που δηλώνονται με τις κάθετες διακεκομμένες γραμμές) και μπορεί να στέλνει ή να λαμβάνει μηνύματα σε άλλα module (που δηλώνονται με τα οριζόντια βέλη). Το module 2 αναστέλλει την εκτέλεση του script μεταξύ των σημείων Α' - Β, μέχρις ότου λάβει το μήνυμα που αποστέλλει το module 1 στο σημείο Α. Αντιθέτως, το module 1

στο σημείο Α, δημοσιεύει το μήνυμα στην ουρά και δρομολογεί κατευθείαν την εκτέλεση του script μεταξύ των σημείων A-C'.



Εικόνα 2-2

Με τον παραπάνω απλό μηχανισμό "μπλοκαρίσματος" για την αναμονή ενός μηνύματος επιτυγχάνεται ο απαραίτητος συγχρονισμός μεταξύ των διαφορετικών module, όσο η διαδικασία εγκατάστασης προχωράει, καθώς ένα module μπορεί να περιμένει κάποιο άλλο να φτάσει σε συγκεκριμένο σημείο της εκτέλεσης του, αναστέλλοντας τη λειτουργία του μέχρι να λάβει το κατάλληλο μήνυμα. Για παράδειγμα, έστω μια εφαρμογή αποτελούμενη από δύο module, έναν web server και έναν database server. Ο web server θα ξεκινήσει τη λειτουργία του, αφού πρώτα περιμένει την επιτυχή ρύθμιση του database server. Στις παρακάτω ενότητα θα κάνουμε μια εκτενή περιγραφή των κυριότερων στοιχείων της Αύρας και του τρόπου λειτουργίας της, που μπορεί να χωριστεί στις εξής βασικές κατηγορίες:

- Προσπέλαση του αρχείου περιγραφής της εφαρμογής (application description).
- Ανάθεση των απαραίτητων πόρων από τον πάροχο υπηρεσιών νέφους (cloud provider) και περαίωση της σύνδεσης στα εικονικά μηχανήματα μέσω ssh.
- Εκτέλεση του σεναρίου εγκατάστασης όπως αυτό ορίζεται από το αρχείο περιγραφής της εφαρμογής.

2.1.3 Προσπέλαση του αρχείου περιγραφής της εφαρμογής

Όλες οι απαραίτητες πληροφορίες για την εγκατάσταση μιας εφαρμογής στο σύννεφο εμπεριέχονται στο αρχείο περιγραφής της εφαρμογής, που ο χρήστης παρέχει σαν όρισμα στην Αύρα. Πρόκειται για ένα αρχείο μορφής JSON (JavaScript Object Notation), περιλαμβάνει τα διαπιστευτήρια του χρήστη για τη σύνδεση με τον πάροχο υπηρεσιών (cloud provider),

πληροφορίες για το είδος (flavor) των εικονικών μηχανών που απαιτούνται για την εγκατάσταση του κάθε module που απαρτίζει την εφαρμογή και την τοποθεσία των σεναρίων εγκατάστασης (scripts) που καλείται να εκτελέσει το κάθε module για την επίτευξη της εγκατάστασης της εφαρμογής. Αξίζει να αναφερθεί ότι στο συγκεκριμένο αρχείο περιγράφεται και η αναγκαία επικοινωνία μεταξύ των module, δηλαδή αναφέρεται για κάθε script αν αποτελεί αποστολέας ή παραλήπτης ενός μηνύματος. Παρακάτω, παρουσιάζεται ένα ενδεικτικό παράδειγμα του αρχείου περιγραφής.

```
{
    "modules": [{
        "name": "aura-one",
        "image_id":"8ae647b4-8357-4caa-9ed6-d9f3ef9d473e" ,
        "flavor_id": “11”,
        "scripts": [{
                "seq": 1,
                "file": "one/script1.sh",
                "output": ["aura-two/2"]
        },
        {
                "seq": 2,
                "file": "one/script2.sh"
        },
        {
                "seq": 3,
                "file": "one/script3.sh",
                "input": ["aura-two/2"]
        }]
    },
    {
        "name": "aura-two",
        "image_id": "8ae647b4-8357-4caa-9ed6-d9f3ef9d473e",
        "flavor_id": "11",
        "multiplicity":2,
        "scripts": [{
                "seq": 1,
                "file": "two/script1.sh"
        },
        {
                "input": ["aura-one/1"],
                "seq": 2,
                "file": "two/script2.sh",
                "output": ["aura-one/3"]
        }]
     }]
}
```

Το κάθε module της εφαρμογής περιέχει τα εξής πεδία:

name	Το όνομα του module
image_id	Το αναγνωριστικό της εικόνας που χρησιμοποιεί ο Openstack για τη δημιουργία της εικονικής μηχανής στην οποία εγκαθίσταται το module
flavor_id	Αναγνωριστικό που δηλώνει τις δυνατότητες του εικονικού μηχανήματος σε επεξεργαστικούς πυρήνες, αποθηκευτικό χώρο και μνήμη.
multiplicity (optional)	Δηλώνει τη πολλαπλότητα ενός module
scripts	Λίστα που περιέχει πληροφορίες για τα script που καλείται να εκτελέσει το κάθε module.

Το κάθε script έχει τις εξής ετικέτες:

seq	Η σειρά εκτέλεσης του κάθε script.
file	Η τοποθεσία του αρχείου.
input	Λίστα που περιέχει τα script από τα οποία αναμένεται είσοδος απαραίτητη για την εκτέλεση.
output	Λίστα που περιέχει τα script στα οποία θα σταλεί η έξοδος της εκτέλεσης

Τέλος, εκτός από την περιγραφή των module μιας εφαρμογής, το αρχείο περιέχει και τα απαραίτητα διαπιστευτήρια για τη σύνδεση της Αύρας με το API του Openstack, όπως φαίνεται παρακάτω.

```
"cloud-conf": {
               "version":"2",
               "username":"user_id",
                      "password":"secret_pass",
                      "project_id":"12345678avcdefg",
               "auth_url":"http://termi7.cslab.ece.ntua.gr:5000/v2.0",
               "network_name": "private-net"
       }
```

Η Αύρα κατασκευάζει μια δομή δεδομένων εύκολα προσπελάσιμη που περιέχει για κάθε module τα scripts που καλείται να εκτελέσει, το περιεχόμενο τους και τη σειρά εκτέλεσης τους. Ακόμα, σε περίπτωση που κάποιο script περιμένει είσοδο από κάποιο άλλο, η καταχώριση ενημερώνεται αντιστοίχως. Ιδιαίτερη αναφορά πρέπει να γίνει στην περίπτωση που για κάποιο module έχει οριστεί στο αρχείο περιγραφής πολλαπλότητα (multiplicity) μεγαλύτερη του ενός.

Η πολλαπλότητα δηλώνει ότι ένα module θα εγκατασταθεί περισσότερες της μίας φοράς σε ξεχωριστά εικονικά μηχανήματα. Όλοι οι σωσίες εκτελούν τις ίδιες ακριβώς ενέργειες, και περιμένουν μηνύματα από τους ίδιους αποστολείς, και στέλνουν μηνύματα στους ίδιους παραλήπτες. Για παράδειγμα, ας θεωρήσουμε ένα hadoop [10] cluster αποτελούμενο από έναν κόμβο αφέντη (master node) και τρεις κόμβους εργάτες (slave nodes). Στο αρχείο περιγραφής της εφαρμογής αρκεί να περιγράφεται μόνο ένα slave module με πολλαπλότητα τρία, καθώς όλοι οι slaves καλούνται να εκτελέσουν τις ίδιες ενέργειες, όπως για παράδειγμα την εγκατάσταση του hadoop και την εδραίωση σύνδεσης με τον master. Η δομή δεδομένων της Αύρας θα περιέχει εγγραφές για ένα master και τρία slave modules. Όπως γίνεται αντιληπτό, καταχωρήσεις που αφορούν τα configuration script του master που επικοινωνούν μέσω μηνυμάτων με αυτά των slaves πρέπει να ενημερωθούν αντιστοίχως, καθώς πλέον περιμένουν είσοδο / παράγουν έξοδο και από / για τα τρια αντίγραφα.

Μετά το τέλος της προσπέλασης του αρχείου περιγραφής της εφαρμογής, όλα τα απαραίτητα δεδομένα για την διαδικασία της εγκατάστασης βρίσκονται στη μνήμη της Αύρας.

2.1.4 Επικοινωνία με τον πάροχο υπηρεσιών και ανάθεση πόρων

Επόμενο βήμα αποτελεί η ανάθεση πόρων από τον πάροχο υπηρεσιών. Η κλάση CloudOrchestrator αναλαμβάνει τη σύνδεση με τον πάροχο, στην περίπτωση μας τον Openstack, με τη βοήθεια του novaclient python API [11], χρησιμοποιώντας τα διαπιστευτήρια του χρήστη, τα οποία παρέχονται από το αρχείο περιγραφής όπως αναφέρθηκε παραπάνω. Μόλις εδραιωθεί η σύνδεση, δημιουργείται ένα νήμα εκτέλεσης (thread) για κάθε module, με σκοπό την δημιουργία μιας εικονικής μηχανής (Virtual Machine) που θα το φιλοξενήσει. Ως ορίσματα παρέχονται το όνομα του module, το είδος (flavor) της εικονικής μηχανής, καθώς και το αναγνωριστικό της εικόνας της εικονικής μηχανής (image id) που προτιμούμε. Συγκεκριμένα, το image id αναφέρεται σε κάποια συγκεκριμένη εικόνα λειτουργικού συστήματος που ο Openstack δύναται να μας παρέχει εγκατεστημένο στο εικονικό μηχάνημα, ενώ το flavor στις υπολογιστικές δυνατότητες του VM.

Χρησιμοποιώντας την κατάλληλη κλήση API του Openstack, το κάθε νήμα εκτέλεσης δημιουργεί μια εικονική μηχανή και επιστρέφει, τερματίζοντας την εκτέλεση του μόλις ο Openstack είναι σε θέση να του επιστρέψει την IP διεύθυνση του VM. Αξίζει να αναφερθεί ότι η Αύρα παρέχει τη δυνατότητα εγκατάστασης σε υπάρχον εικονικό μηχάνημα, εφόσον παρέχεται η διεύθυνση IP του στο αρχείο περιγραφής, και στην περίπτωση αυτή το στάδιο ανάθεσης πόρων παραλείπεται για το συγκεκριμένο module.

2.1.5 Εκτέλεση του σεναρίου εγκατάστασης της εφαρμογής

Η εκτέλεση του σεναρίου εγκατάστασης μπορεί να χωριστεί στις εξής διαδικασίες:

- Δημιουργία νημάτων εκτέλεσης υπεύθυνα για τη δρομολόγηση των script του κάθε module.
- Εδραίωση της σύνδεσης των προαναφερθέντων νημάτων με τα αντίστοιχα εικονικά μηχανήματα.
- Εκκίνηση της διαδικασίας εγκατάστασης της εφαρμογής με εκτέλεση των απαραίτητων script.
- Παρακολούθηση της διαδικασίας για τυχόν σφάλματα.

Τα νήματα εκτέλεσης διεκπεραιώνουν την επικοινωνία της Αύρας με τα εικονικά μηχανήματα. Ουσιαστικά, πρόκειται για τους Aura Executors που περιγράψαμε στην αρχιτεκτονική του συστήματος. Κάθε νήμα αντιστοιχεί σε ένα module και αναλαμβάνει να εκτελέσει τα αντίστοιχα script στο εικονικό μηχάνημα όπου θα εγκατασταθεί το module. Για την αρχικοποίηση

των νημάτων απαιτούνται οι πληροφορίες εγκατάστασης του ανάλογου module, όπως αυτές ορίστηκαν στο αρχείο περιγραφής της εφαρμογής (scripts κλπ.), αλλά και το ιδιωτικό κλειδί για τη σύνδεση στα εικονικά μηχανήματα, την οποία περιγράφουμε στη συνέχεια. Τέλος, τα νήματα επικοινωνούν μεταξύ τους με έναν κοινό και απλό μηχανισμό ουράς ανταλλαγής μηνυμάτων.

Η ουρά ανταλλαγής μηνυμάτων είναι μια εμφωλευμένη δομή απο Python Dictionaries, στην οποία έχουν πρόσβαση όλα τα νήματα. Επειδή πρόκειται για διαμοιραζόμενη δομή την οποία ενδέχεται περισσότερα του ενός νήματα να επιχειρούν να τροποποιούν παράλληλα, έχουμε εισάγει το κατάλληλο κλείδωμα (lock), το οποίο εγγυάται ότι κάθε στιγμή μόνο ένα νήμα έχει τη δυνατότητα να τροποποιεί τα δεδομένα της ουράς, διασφαλίζοντας την ατομικότητα και την εγκυρότητα των δεδομένων. Η ουρά υποστηρίζει μεθόδους send, receive και block-receive.

Η μέθοδος send τροποποιεί τα δεδομένα της ουράς εισάγοντας νέα μηνύματα. Δέχεται ως ορίσματα τον αποστολέα του μηνύματος, τον παραλήπτη και το μήνυμα. Ο αποστολέας και ο παραλήπτης έχουν τη δομή <όνομα module>/<σειρά εκτέλεσης script>. Η μέθοδος send, διεκδικεί το κλείδωμα και εισάγει το καινούργιο μήνυμα στην ουρά δημιουργώντας μια νέα εγγραφή της μορφής αποστολέας --> μήνυμα για τον αντίστοιχο παραλήπτη.

Η μέθοδος receive παίρνει ως όρισμα τον αποστολέα και τον παραλήπτη του μηνύματος που ο παραλήπτης περιμένει, διεκδικεί το κλείδωμα, ελέγχει αν το μήνυμα έχει σταλεί μέσω της μεθόδου send από τον αποστολέα και σε περίπτωση επιτυχίας επιστρέφει το μήνυμα. Η μέθοδος block-receive αποτελεί επέκταση της receive δίνοντας την δυνατότητα στον καλών να αναστέλλει την εκτέλεση του για συγκεκριμένο χρονικό διάστημα ή και επ’ αόριστον, μέχρι το μήνυμα να ληφθεί.

Για τη σύνδεση των νημάτων εκτέλεσης με τα εικονικά μηχανήματα που φιλοξενούν τα module της εφαρμογής χρησιμοποιείται το paramiko [12], module της Python που υλοποιεί το πρωτόκολλο SSHv2. Για την περαίωση της σύνδεσης χρησιμοποιείται για πρακτικούς λόγους ζεύγος δημοσίου κλειδιού που προϋπάρχει στο image των εικονικών μηχανών. Το ιδιωτικό κλειδί που απαιτείται πρέπει να υπάρχει στο σύστημα που εκτελεί τον Aura Master και η τοποθεσία του δηλώνεται στο αρχείο περιγραφής.

Μετά την περαίωση της σύνδεσης ξεκινάει η διαδικασία εγκατάστασης της εφαρμογής. Το κάθε νήμα εκτέλεσης καλείται να εκτελέσει τα script του αντίστοιχου module μέσω της μεθόδου execute script η οποία επαναλαμβάνεται για κάθε script μέχρι να εκτελεστούν όλα με επιτυχία. Τα βήματα της μεθόδου συνοψίζονται ως εξής.

Αρχικά, όλα τα script πριν τη δρομολόγηση της εκτέλεσης τους βρίσκονται στην κατάσταση αναμονής (script status: Pending). Μόλις κάποιο script δρομολογηθεί για εκτέλεση (η σειρά εκτέλεσης έχει οριστεί από το αρχείο περιγραφής της εφαρμογής), περνά στη κατάσταση αναμονής εισόδου (script status: waiting for input), στην περίπτωση που περιμένει ένα ή

περισσότερα μηνύματα εισόδου από κάποιο άλλο module, η διαδικασία δεν ξεκινά μέχρι αυτά να ληφθούν.

Στη συνέχεια προχωράμε στην κατάσταση εκτέλεσης (script status: Executing), όπου το περιεχόμενο του script και τα μηνύματα εισόδου αντιγράφονται σε δύο ξεχωριστά αρχεία στο εικονικό μηχάνημα, επιστρέφοντας το όνομα αυτών. Το μόνο που απομένει είναι η εκτέλεση του αρχείου που βρίσκεται στο εικονικό μηχάνημα με όρισμα το αρχείο που περιέχει τα μηνύματα. Όπως έχουμε αναφέρει τα μηνύματα μπορεί να λειτουργούν απλά ως μέσο συγχρονισμού, χωρίς να περιέχουν πληροφορία αναγκαία για την εκτέλεση ή αντίθετα να αποτελούν ορίσματα, απαραίτητα για την επιτυχία του script. Γι αυτό το λόγο το αρχείο μηνυμάτων πάντα παρέχεται ως όρισμα, έστω και κενό αν δεν υπάρχουν μηνύματα και το script αναλαμβάνει να το προσπελάσει εφόσον χρειάζεται. Το νήμα δρομολογεί την εκτέλεση του απομακρυσμένου αρχείου, μετά το πέρας της οποίας επιστρέφονται δύο περιγραφητές αρχείων (file descriptors), έναν για τη ροή εξόδου (standard output stream) και έναν για τη ροή σφάλματος (standard error stream) που παρήχθησαν. Αξίζει να αναφερθεί ότι ενώ η μέθοδος που δρομολογεί την εκτέλεση επιστρέφει κατευθείαν η ανάγνωση των δύο περιγραφητών περιμένει την ολοκλήρωση του script.

Μόλις ολοκληρωθεί η εκτέλεση και προσπελαστούν οι περιγραφητές εξόδου και σφάλματος είμαστε σε θέση να γνωρίζουμε το αποτέλεσμα τις διαδικασίας. Σε περίπτωση επιτυχίας, το ρεύμα σφάλματος είναι κενό και μεταβαίνουμε στην κατάσταση επιτυχίας (script status: Finished). Αντιθέτως, σε περίπτωση αποτυχίας το ρεύμα σφάλματος δεν είναι κενό, μεταβαίνουμε στη κατάσταση σφάλματος (script status: Error) και η διαδικασία που περιγράφτηκε για την κατάσταση εκτέλεσης (script status: Executing) επαναλαμβάνεται.

Τελευταίο βήμα, αποτελεί η αποστολή του ρεύματος εξόδου σε όσα script αναμένουν μήνυμα από το script που μόλις εκτελέστηκε. Για κάθε παραλήπτη, εισάγουμε στον μηχανισμό της ουράς το μήνυμα όπως εξηγήθηκε παραπάνω. Τέλος, μεταβαίνουμε στη κατάσταση επιτυχούς τερματισμού της εκτέλεσης (script status: Done) και είμαστε έτοιμοι να δρομολογήσουμε το επόμενο script.

Ένα νήμα εκτέλεσης επαναλαμβάνει την παραπάνω διαδικασία για όλα τα script του αντίστοιχου module μέχρις ότου όλα να ολοκληρωθούν με επιτυχία και ανακοινώνει την ολοκλήρωση της δουλειάς του. Αντίστοιχα, η διαδικασία εγκατάστασης τερματίζει όταν όλα τα νήματα εκτέλεσης ολοκληρώσουν την εκτέλεση τους.

2.2 Το πρόγραμμα CRIU

Το CRIU είναι ένα εργαλείο του λειτουργικού συστήματος Linux που προσφέρει τη δυνατότητα παγώματος (checkpoint) και επαναφοράς (restore) εφαρμογών σε χώρο χρήστη. Στη διαδικασία του checkpoint η κατάσταση μιας εφαρμογής, δηλαδή ο εικονικός της χώρος διευθύνσεων, περιγραφητές αρχείων, sockets και άλλοι βασικοί πόροι, σώζεται σε μια ομάδα αρχείων στο δίσκο. Ο χρήστης μπορεί να χρησιμοποιήσει αυτά τα αρχεία ώστε να επαναφέρει την εφαρμογή στην κατάσταση που βρίσκονταν ακριβώς όταν εκτελέστηκε το checkpoint. Παρακάτω παρουσιάζουμε ορισμένα πλεονεκτήματα του CRIU και περιγράφουμε τη λειτουργία των διαδικασιών checkpoint και restore μιας διεργασίας. Στη συνέχεια της περιγραφής με τον όρο διεργασία θα αναφερόμαστε στο δέντρο διεργασιών που επιθυμούμε να παγώσουμε / επαναφέρουμε.

Ορισμένες από τις λειτουργικότητες που προσφέρει το CRIU σε σχέση με άλλα παρόμοια προγράμματα είναι τα παρακάτω:

- Είναι υλοποιημένο σε χώρο χρήστη.
- Είναι διαφανές στις εφαρμογές που αποθηκεύει, δηλαδή η εφαρμογή που πρόκειται να παγώσει δεν αντιλαμβάνεται ότι αποθηκεύεται η κατάστασή της και ως εκ τούτου δεν απαιτείται τροποποίηση του πηγαίου κώδικα της.
- Δεν υπάρχει κάποια επιπλέον επιβάρυνση στην εκτέλεση μιας εφαρμογής που έχει επαναφερθεί από παγωμένη κατάσταση.
- Δεν απαιτεί τροποποιημένο πυρήνα του Linux για εκδόσεις πυρήνα μετά του 3.11
- Είναι σχετικά απλό στην εγκατάσταση και στη χρήση του.
- Υποστηρίζεται η αποθήκευση και αποκατάσταση συνδέσεων δικτύου.

2.2.1 Η διαδικασία αποθήκευσης της κατάστασης μιας διεργασίας

Η διαδικασία του checkpoint μιας διεργασίας μπορεί να χωριστεί σε τρεις φάσεις: Πάγωμα του δέντρου διεργασιών της εφαρμογής, συλλογή της κατάστασης και των πόρων των διάφορων νημάτων της εφαρμογής και αποθήκευση τους σε αρχεία στο δίσκο και τέλος επαναφορά της διεργασίας σε λειτουργική κατάσταση ή τερματισμό της. Το CRIU αντλεί τις πληροφορίες που χρειάζεται από το /proc σύστημα αρχείων. Τέτοιου είδους πληροφορίες μεταξύ άλλων αποτελούν:

Πληροφορία	Κατάλογος αρχείων
Περιγραφητές αρχείων που χρησιμοποιεί η διεργασία	/proc/$pid/fd/ /proc/$pid/fdinfo/
Παράμετροι καναλιών δεδομένων (pipes, sockets)	/proc/$pid/fd/
Χάρτες μνήμης (memory maps)	/proc/$pid/maps /proc/$pid/map_files/

Πριν την έναρξη της διαδικασίας του checkpoint, πρέπει να σιγουρευτούμε ότι η κατάσταση της διεργασίας δεν θα αλλάξει. Αλλαγή της κατάστασης μπορεί να σημαίνει άνοιγμα νέων αρχείων, socket, αλλαγή συνεδρίας (session) κλπ ή και παραγωγή νέων διεργασιών-παιδιών τα οποία θα αποτύχουμε να παγώσουμε. Γι' αυτό το λόγο το δέντρο διεργασιών πρέπει να παγώσει στην κατάσταση την οποία βρίσκεται πριν την έναρξη της διαδικασίας. Ενώ θεωρητικά κάτι τέτοιο είναι απλό να επιτευχθεί με ένα σήμα stop (SIGSTOP signal), στη πράξη η διεργασία δεν πρέπει να αντιληφθεί κάποια αλλαγή στη κατάσταση της. Συγκεκριμένα, το σήμα SIGSTOP αν και δεν μπορεί να αγνοηθεί, είναι παρατηρήσιμο από τη διεργασία πατέρα. Γι' αυτό το λόγο χρησιμοποιείται είτε η κλήση συστήματος ptrace [13], είτε ο cgroup freezer [14], καθώς και τα δύο αυτά εργαλεία παρέχουν τη δυνατότητα παύσης της εκτέλεσης ενός δέντρου διεργασιών χωρίς αυτή να γίνει αντιληπτή από τις ίδιες τις διεργασίες.

Ο αναγνωριστικός αριθμός της διεργασίας (pid) παρέχεται σαν όρισμα από τη γραμμή εντολών (επιλογή –tree). Το CRIU χρησιμοποιεί αυτο το pid για να προσπελάσει τους καταλόγους /proc/$pid/task/ και /proc/$pid/task/$tid/children συλλέγοντας τις απαραίτητες πληροφορίες για το δέντρο διεργασιών και παγώνει τις διεργασίες μέσω της εντολής PTRACE_SEIZE της κλήσης συστήματος ptrace. Συγκεκριμένα, από το man page της ptrace:

Η κλήση συστήματος ptrace() παρέχει τη δυνατότητα σε μια διεργασία (tracer) να παρατηρεί και να ελέγχει την εκτέλεση μιας άλλης διεργασίας (tracee) και να τροποποιεί τη μνήμη της και τους καταχωρητές της. Με την εντολή PTRACE_SEIZE (πυρήνας Linux >3.4) ο tracer προσκολλάται στη διεργασία με το δοσμένο pid χωρίς να την σταματάει. Οι διεργασίες παιδιά σταματούν την εκτέλεση τους αναφέροντας σήμα SIGTRAP αντί για SIGSTOP χωρίς έτσι να γίνεται αντιληπτό το πάγωμα του δέντρου από τη διεργασία πατέρα.

Έχοντας πλέον παγώσει το δέντρο διεργασιών, το CRIU χρησιμοποιώντας τις πληροφορίες που συνέλεξε, αποθηκεύει την κατάσταση των παγωμένων εφαρμογών σε ένα σύνολο αρχείων. Για να το πετύχει αυτό είτε διαβάζει το /proc σύστημα αρχείων, είτε χρησιμοποιεί την τεχνική έγχυσης παρασιτικού κώδικα (Parasite Call Injection).

Στη πρώτη περίπτωση, πληροφορίες για τις διευθύνσεις εικονικής μνήμης (/proc/$pid/smaps), αρχεία φορτωμένα στη μνήμη (/proc/$pid/map_files), αριθμοί των ανοικτών περιγραφητών αρχείων (/proc/$pid/fd) και άλλες παράμετροι, όπως καταχωρητές που χρησιμοποιούνται από την διεργασία (/proc/$pid/stat) αποθηκεύονται στα κατάλληλα αρχεία.

Στην περίπτωση της έγχυσης παρασιτικού κώδικα, ενώ η διεργασία βρίσκεται σε παγωμένη κατάσταση, γίνεται έγχυση του παράσιτου και εκτέλεση μιας κλήσης συστήματος mmap μέσα από το χώρο διευθύνσεων της διεργασίας με τη βοήθεια της ptrace, ώστε να κατανεμηθεί μια περιοχή κοινής μνήμης μεταξύ του CRIU και της διεργασίας, που θα χρησιμοποιηθεί για τη στοίβα του παρασιτικού κώδικα και την ανταλλαγή πληροφοριών. Έπειτα, ο παρασιτικός κώδικας αντιγράφεται στο χώρο διευθύνσεων της διεργασίας και ο δείκτης εντολής (CS:IP) της μετατοπίζεται κατάλληλα, ώστε να γίνει εκτέλεση του παρασίτου, που δίνει τη δυνατότητα στο CRIU να αντιγράψει τα περιεχόμενα της μνήμης της διεργασίας στα κατάλληλα αρχεία. Όλες οι πληροφορίες που συγκεντρώνονται αποθηκεύονται σε αρχεία της μορφής google protocol buffer[15].

Τέλος όταν όλες οι απαραίτητες πληροφορίες αποθηκευτούν, ο παρασιτικός κώδικας αφαιρείται με τη χρήση της ptrace και το CRIU τερματίζει τη λειτουργία του, είτε σκοτώνοντας το δέντρο διεργασιών είτε αφήνοντας τη διεργασία να συνεχίσει τη λειτουργία της σαν να μη συνέβη τίποτα.

2.2.2 Η διαδικασία της επαναφοράς (restore)

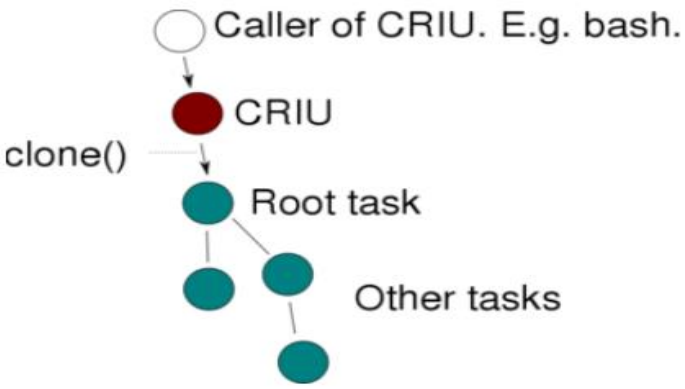
Κατά τη διαδικασία της επαναφοράς, το CRIU μεταλλάσσεται στις διεργασίες τις οποίες επαναφέρει. Αρχικά, πρέπει να γίνει επίλυση των διαμοιραζόμενων πόρων. Το CRIU προσπελαύνει τα αρχεία που δημιούργησε κατά το checkpoint, βρίσκοντας ποιες διεργασίες μοιράζονται πόρους. Αργότερα, αυτοί οι διαμοιραζόμενοι πόροι, όπως αρχεία, περιοχές κοινής μνήμης κλπ, επαναφέρονται από μια διεργασία κληρονομώντας τους στις υπόλοιπες. Έπειτα, το CRIU καλεί την κλήση συστήματος fork() πολλές φορές, μέχρι να αναπαράγει τη δομή του δέντρου διεργασιών που καλείται να επαναφέρει. Να σημειώσουμε ότι η επαναφορά των νημάτων εκτέλεσης της διεργασίας δεν πραγματοποιείται εδώ, αλλά σε επόμενο στάδιο.

Στη συνέχεια, το CRIU επαναφέρει βασικούς πόρους που χρησιμοποιούσε η διεργασία, ανοίγοντας αρχεία και sockets, προετοιμάζοντας τους χώρους ονομάτων (namespaces) και γεμίζοντας με δεδομένα ιδιωτικές περιοχές μνήμης. Χρονοδιακόπτες, διαπιστευτήρια, νήματα εκτέλεσης των διεργασιών καθώς και οι ακριβείς διευθύνσεις της πραγματικής μνήμης που αντιστοιχούν στην εικονική μνήμη τους επαναφέρονται στο τελευταίο στάδιο, για λόγους που εξηγούμε παρακάτω.

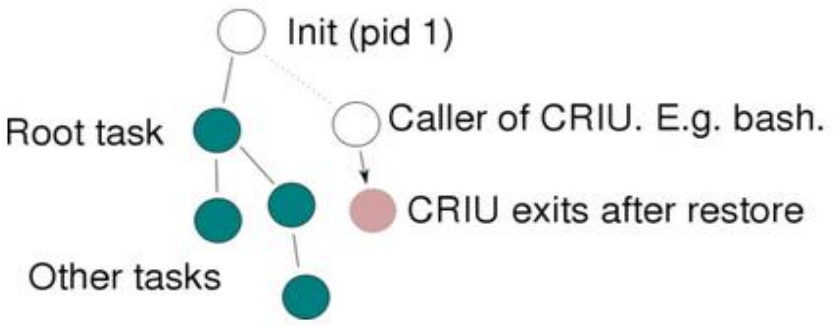
Όπως αναφέραμε στο δεύτερο στάδιο της επαναφοράς, το CRIU εκτελεί fork() αναπαράγοντας το δέντρο διεργασιών που επαναφέρει. Ως αποτέλεσμα, θα πρέπει να αντικαταστήσει τα δικά του περιεχόμενα της μνήμης με αυτά της διεργασίας, αφήνοντας στη μνήμη ένα κομμάτι κώδικα που θα πραγματοποιεί τις απαραίτητες κλήσεις munmap και mmap. Σε αυτό το σημείο λοιπόν επαναφέρουμε τη μνήμη της διεργασίας, τους χρονοδιακόπτες για να μην πυροδοτηθούν πρόωρα, και τα διαπιστευτήρια που επιτρέπουν στο CRIU προνομιούχες λειτουργίες όπως fork-with-pid και τα νήματα εκτέλεσης, τα οποία αν επαναφέραμε νωρίτερα δεν θα μπορούσαν να λειτουργήσουν λόγω της ξαφνικής αλλαγής της τοπολογίας της μνήμης τους.

Αξίζει να σημειωθεί ότι οι διεργασίες που επαναφέρονται έχουν το ίδιο PID με αυτό που είχαν τη στιγμή που συνέβη το πάγωμα τους, γεγονός που σημαίνει ότι ύπαρξη μιας τρίτης διεργασίας με αυτό το PID κατά τη διαδικασία του restore θα οδηγήσει σε αποτυχία.

Το δέντρο διεργασιών μετά τη διαδικασία επαναφοράς φαίνεται παρακάτω.



Το δέντρο διεργασιών που επαναφέραμε έχει πλέον ως πατέρα τη διεργασία του CRIU. Σε περίπτωση που επιθυμούμε η διεργασία του CRIU να τερματιστεί μετά την ολοκλήρωση της επαναφοράς και το δέντρο διεργασιών να κληρονομηθεί απο την Init, όπως φαίνεται στην παρακάτω εικόνα, αρκεί να παρέχουμε ως επιπλέον όρισμα το *--restore-detached* στην εντολή *criu restore*.



3 Μηχανισμός και Επεκτάσεις

Στο παρόν κεφάλαιο θα παρουσιάσουμε ένα σενάριο χρήσης το οποίο αναδεικνύει το πρόβλημα που προσπαθήσαμε να αντιμετωπίσουμε. Στη συνέχεια, θα αναφερθούμε στην αρχιτεκτονική του μηχανισμού τον οποίο προτείνουμε και στον εμπλουτισμό της Αύρας με τις λειτουργίες checkpoint και restore.

3.1 Παροδικά σφάλματα και η έννοια της ταυτοδυναμίας

Όπως έχουμε αναφέρει η πολυπλοκότητα του λογισμικού εικονοποίησης και της στοίβας υλικού των cloud υπηρεσιών μπορεί να οδηγήσει σε σφάλματα παροδικής φύσεως από την πλευρά του παρόχου. Προσωρινά μη διαθέσιμες υπηρεσίες δικτύου (πχ. DNS), απόρριψη δικτυακών πακέτων, έλλειψη συνδεσιμότητας και μη σωστά προσαρτημένα συστήματα αρχείων (πχ read-only file systems) είναι μερικά από αυτά και συνήθως εκδηλώνονται για μικρό χρονικό διάστημα. Εμφάνιση τέτοιων σφαλμάτων κατά τη διαδικασία εγκατάστασης μπορεί να οδηγήσει σε αδυναμία επιτυχούς εκτέλεσης ενός η περισσοτέρων configuration script.

Η υπάρχουσα έκδοση της Αύρας υιοθετεί την ιδέα της επανεκτέλεσης ενός script σε περίπτωση αποτυχίας, βασιζόμενη στην παροδική φύση των προβλημάτων που αναφέραμε. Όταν το πρόβλημα αποκατασταθεί, η εκτέλεση του script θα ολοκληρωθεί με επιτυχία και η διαδικασία εγκατάστασης μπορεί να συνεχιστεί. Η λύση της επανεκτέλεσης ως μηχανισμό ανάνηψης σφαλμάτων προϋποθέτει δύο βασικά χαρακτηριστικά: Πρώτον, τα τυχόν σφάλματα που προκύπτουν να είναι παροδικά, καθώς η επανεκτέλεση δεν θα έχει τα επιθυμητά αποτελέσματα σε περίπτωση επίμονων σφαλμάτων. Δεύτερον, η επανεκτέλεση ενός script πρέπει να οδηγεί την εγκατάσταση στην ίδια κατάσταση, η πιο απλά να έχει πάντα τις ίδιες παρενέργειες. Αυτή η ιδιότητα ονομάζεται ταυτοδυναμία (idempotence).

Στη γενική περίπτωση, εκτέλεση ενός script παραπάνω της μιας φοράς δεν έχει πάντα τις ίδιες παρενέργειες. Για παράδειγμα, έστω το παρακάτω script, το οποίο αναλαμβάνει να διαβάσει τα δεδομένα ενός αρχείου από το σύστημα αρχείων και μετά το διαγράφει.

```
#!/bin/bash


cat /tmp/file_A>> /etc/conf

rm /tmp/file_A
```

Το συγκεκριμένο script μπορεί να εκτελεστεί ακριβώς μια φορά. Κάθε διαδοχική εκτέλεση θα αποτύχει, καθώς το αρχείο έχει διαγραφεί, ως αποτέλεσμα των παρενεργειών της πρώτης εκτέλεσης.

Αν και τα περισσότερα scripts προκαλούν παρενέργειες σε επίπεδο συστήματος αρχείων (file system) το ίδιο πρόβλημα συναντάται και σε επίπεδο μνήμης, καθώς πολλές εφαρμογές στηρίζονται σχεδόν εξ ολοκλήρου στη μνήμη για τη λειτουργία τους. Χαρακτηριστικό παράδειγμα αποτελούν εφαρμογές αποθήκευσης ζευγών κλειδιού-τιμής στη μνήμη και χρησιμοποιούνται ευρέως ως συστήματα κρυφής μνήμης (cache) βελτιώνοντας την απόδοση πληθώρας συστημάτων, καθώς αποθηκεύοντας δεδομένα στη μνήμη και όχι στο σύστημα αρχείων, προσφέρουν γρηγορότερη ανταπόκριση σε αιτήματα σε σχέση με μια βάση δεδομένων. Δύο τέτοια χαρακτηριστικά παραδείγματα αποτελούν το redis [7] και το memcached [17] δύο εργαλεία αποθήκευσης ζευγών κλειδιού-τιμής στη μνήμη, που μπορεί να χρησιμοποιηθούν μεταξύ άλλων και σαν βάση δεδομένων. Κατά αναλογία με το προηγούμενο παράδειγμα έστω ένα script το οποίο ζητά τη τιμή ενός κλειδιού από τον redis server και στη συνέχεια το διαγράφει από τη βάση. Αντίστοιχα με την περίπτωση του file system αυτό το script μπορεί να εκτελεστεί μόνο μια φορά, καθώς επανεκτέλεση του δεν οδηγεί το module redis server και κατ' επέκταση όλο το deployment στην ίδια κατάσταση.

Παρατηρούμε ότι scripts που προκαλούν παρενέργειες σε επίπεδο συστήματος αρχείων, μνήμης ή και εξωτερικών πόρων μπορεί να μην επιτρέπουν την επανεκτέλεση τους. Στην παρούσα διπλωματική εργασία καλούμαστε να επιλύσουμε τον περιορισμό αυτόν σε επίπεδο μνήμης για το σύστημα της Αύρας, καθιστώντας δυνατή την επανεκτέλεση script που αλληλεπιδρούν με τη μνήμη στοιχείων ενός module, τροποποιώντας την κατάσταση της.

Για να ξεπεράσουμε τον περιορισμό τον οποίο αναλύσαμε, υιοθετούμε έναν μηχανισμό στιγμιότυπου (snapshot mechanism) του module η κατάσταση του οποίου πρόκειται να τροποποιηθεί από την εκτέλεση ενός script. Πιο συγκεκριμένα, με τη βοήθεια του CRIU, αποθηκεύουμε την κατάσταση του module σε αρχεία στο εικονικό μηχάνημα στο οποίο είναι εγκατεστημένο και προχωρούμε στην εκτέλεση του script. Σε περίπτωση αποτυχίας λόγω παροδικού σφάλματος, επαναφέρουμε το module στην κατάσταση την οποία έχουμε αποθηκεύσει επιτρέποντας έτσι επανεκτέλεση script που προκαλούν παρενέργειες στην κατάσταση του module.

Προϋποθέσεις

Για την ομαλή λειτουργία όσων θα περιγράψουμε παρακάτω προϋποθέτουμε ότι ο χρήστης χρησιμοποιεί για την εγκατάσταση της εφαρμογής του εικονικές μηχανές οι οποίες έχουν έκδοση του CRIU (>3.4) εγκατεστημένη και λειτουργική. Το CRIU μπορεί βεβαίως να εγκατασταθεί στα εικονικά μηχανήματα πριν την έναρξη του application deployment εμπλουτίζοντας το αρχείο περιγραφής της εφαρμογής με τα απαραίτητα script που θα εγκαταστήσουν το CRIU, επιμηκύνοντας όμως έτσι χρονικά τη διαδικασία. Ακόμα, απαραίτητη είναι μια σχετικά πρόσφατη έκδοση του πυρήνα του Linux (4.8 ή νεότερη) με τη ρύθμιση CONFIG_CHECKPOINT_RESTORE. Με αυτή την ρύθμιση το CRIU έχει πρόσβαση σε πολλές παραμέτρους του πυρήνα από χώρο χρήστη με την χρήση API.

3.2 Η διαδικασία του Checkpoint

Σε αυτή την ενότητα θα περιγράψουμε τον μηχανισμό παραγωγής ενός στιγμιότυπου της διαδικασίας εγκατάστασης (checkpoint), αλλά και το πότε και το πως αυτός τίθεται σε λειτουργία.

Η σχεδίαση του συστήματος βασίστηκε στο πότε θα θέλαμε η Αύρα να επιτελεί τη λειτουργία του Checkpoint. Μια απλή υλοποίηση θα υπαγόρευε εκτέλεση του μηχανισμού μετά από την εκτέλεση του κάθε script. Εντούτοις, η παραγωγή στιγμιότυπου είναι μια διαδικασία που μπορεί να διαρκέσει από μερικά δευτερόλεπτα μέχρι λεπτά ανάλογα με τον όγκο της μνήμης που χρησιμοποιεί η διεργασία που επιθυμούμε να αποθηκεύσουμε, καθιστώντας το χρονικό κόστος που προσθέτει η τόσο συχνή εκτέλεση της διαδικασίας στον συνολικό χρόνο του deployment απαγορευτικό. Μια καλύτερη προσέγγιση είναι η εκτέλεση του μηχανισμού μόνο πριν από scripts που δεν έχουν την ιδιότητα της ταυτοδυναμίας και τροποποιούν τη κατάσταση της μνήμης του module που μας ενδιαφέρει και κατ' επέκταση της εφαρμογής.

Ο εντοπισμός τέτοιων scripts είναι μια διαδικασία που δε χρίζει αυτοματοποίησης, ειδικά όταν μιλάμε για έναν καθολικό μηχανισμό. Ακόμα, επιθυμούμε οι επιπρόσθετες λειτουργίες που εισάγουμε στην Αύρα να είναι ανεξάρτητες του αρχείου περιγραφής της εφαρμογής, δηλαδή να μην περιέχει scripts εντολών του CRIU και σχετικά με την επιπλέον λειτουργικότητα που προσφέρουμε. Γι' αυτό το λόγο δίνουμε τη δυνατότητα στο χρήστη να ορίσει σε ποιο σημείο της εκτέλεσης επιθυμεί την παραγωγή στιγμιότυπου, εισάγοντας στο αρχείο περιγραφής πέντε νέες ετικέτες (tags).

Ετικέτα	Χρησιμότητα
“coordinator”	Ετικέτα που αναφέρεται σε module του οποίου την κατάσταση επιθυμούμε να αποθηκεύσουμε με τη βοήθεια του CRIU
“criu”:[“module name/script sequence”]	Ετικέτα που αναφέρεται σε script module ιδιότητας coordinator. Με τη χρήση της ο χρήστης δηλώνει ότι μετά την εκτέλεση του script μπαίνει σε λειτουργία ο μηχανισμός παραγωγής στιγμιότυπου. Η λίστα στο δεξί μέρος της ετικέτας περιλαμβάνει τα scripts από τα οποία περιμένουμε είσοδο για την εκκίνηση της εκτέλεσης.
“notify”:[“module name/script sequence”]	Script με αυτή την ετικέτα παράγουν μηνύματα για τα script που δηλώνονται στη λίστα. Η ετικέτα notify αφορά μηνύματα προς script με την ετικέτα criu (κατ’ αναλογία με από πάνω)
cp_scripts	Όταν ένα νήμα καλείται να εκτελέσει ένα script με αυτή την ετικέτα και συγκεκριμένα ακριβώς πριν την έναρξη της εκτέλεσης του, παράγει ένα αντίγραφο της κατάστασης της εκτέλεσης του και συγκεκριμένα της δομής των script που διατηρεί. Αμέσως μετά την ολοκλήρωση του checkpoint κάθε module πρέπει να ανανεώσει το αντίγραφο της κατάστασης του.
cp_queue	Αναφέρεται σε scripts που περιέχουν την ετικέτα criu. Δηλώνει την παραγωγή αντιγράφου της δομής της ουράς ανταλλαγής μηνυμάτων αμέσως μόλις ολοκληρωθεί η διαδικασία του checkpoint. Η ουρά αποτελεί κοινή δομή οπότε απαιτείται δημιουργία ενός μόνο αντιγράφου.

Με χρήση των κατάλληλων ετικετών, ο χρήστης επιλέγει το ακριβές σημείο της εκτέλεσης που θέλει να αποθηκεύσει ενώ το ίδιο αρχείο περιγραφής είναι συμβατό και με την κλασσική έκδοση της Αύρας, όπου οι επιπλέον ετικέτες απλά αγνοούνται. Στο παρακάτω κομμάτι ενός αρχείου περιγραφής, η εγγραφή “criu”:[“client/1”] δηλώνει ότι το module server, στο τρίτο κατά σειρά script της εκτέλεσης του και συγκεκριμένα αμέσως μετά το πέρας του, θα αποθηκεύσει την κατάσταση του, μόλις λάβει μήνυμα από το πρώτο script του module client.

```
"modules": [{
 "name": "server",
 "multiplicity":2,
 "coordinator":"",
 "image_id": "1c92ce6a-ee8f-4946-a848-c07c2fc1a836",
 "flavor_id": "14",
 "scripts": [{
        "seq": 1,
        "file": "server/create-criu.sh"
 },
 {
        "seq": 2,
        "file": "server/start-server.sh",
        "output": ["client/1","client/3","client/5"]
 },
 {
        "seq": 3,
        "file": "server/dummy.sh",
        "criu": ["client/1"],
        "output": ["client/2"],
        "cp_queue":""
 }
```

Αξίζει να σημειωθεί ότι στο συγκεκριμένο παράδειγμα, επιθυμούμε να παγώσουμε την κατάσταση της εκτέλεσης του module server μετά την ολοκλήρωση του script client/1. Αν αυτή η στιγμή δεν αντιστοιχεί σε κάποιο script μπορούμε απλά να εισάγουμε ένα άδειο (dummy) script, με τις κατάλληλες ετικέτες.

Η παραγωγή στιγμιότυπου του γράφου εκτέλεσης δεν περιλαμβάνει μόνο αποθήκευση της κατάστασης των module. Για να μπορεί η Αύρα να επαναδρομολογήσει την εκτέλεση σε περίπτωση σφάλματος της εγκατάστασης χρειάζεται η αποθήκευση της κατάστασης των scripts του κάθε module και της ουράς ανταλλαγής μηνυμάτων. Αναλυτικότερα, έστω για το application description της εικόνας η Αύρα, μέσω του αντίστοιχου thread δρομολογεί για εκτέλεση το script server/3 με ετικέτα “criu”:[“client/1”], του οποίου η κατάσταση μεταβαίνει απο Pending σε Waiting_to_dump, δηλώνοντας ότι περιμένει την ολοκλήρωση του script client/1. Όσο παραμένει σε αυτή την κατάσταση, διεκδικεί πρόσβαση στην κοινή ουρά ανταλλαγής μηνυμάτων μέχρι να λάβει το αντίστοιχο μήνυμα. Η ίδια διαδικασία επαναλαμβάνεται για όλα τα μηνύματα που έχουν οριστεί στο αρχείο περιγραφής.

Στη συνέχεια, το script μεταβαίνει στη κατάσταση Executing, όπου εκτελείται το περιεχόμενό του όπως περιγράψαμε στο Κεφάλαιο 2. Μετά την επιτυχή ολοκλήρωση του ξεκινά η διαδικασία του checkpoint. Αρχικά, με τη βοήθεια του CRIU αποθηκεύουμε την κατάσταση του module στο απομακρυσμένο μηχάνημα, εκτελώντας ένα αρχείο που έχουμε δημιουργήσει στο εικονικό μηχάνημα από την αρχή της διαδικασίας και παρουσιάζεται ενδεικτικά παρακάτω.

```
#!/bin/bash


CRIU=/root/criu-3.4/criu/criu

PID="$(pidof <<process>>)"

if [ -d server-dump ]
then
    rm -r /root/server-dump
fi
mkdir server-dump


$CRIU dump -vvv -o dump.log -t $PID --images-dir /root/server-dump --
shell-job --tcp-established && echo OK
```

Συνοπτικά, το script βρίσκει το PID της διεργασίας που θέλουμε να παγώσουμε και εφόσον αυτή είναι ενεργή κατασκευάζει τον κατάλογο όπου θα αποθηκευτούν τα αρχεία που θα παραχθούν από την εκτέλεση της εντολής criu dump. Η εντολή criu dump δέχεται ένα μεγάλο πλήθος ορισμάτων ανάλογα με τη διεργασία που καλείται να αποθηκεύσει. Στον παρακάτω πίνακα παρουσιάζονται συνοπτικά μερικά από τα βασικά. Αξίζει να σημειωθεί ότι στη γενική περίπτωση η χρήση του ορίσματος –leave-running αποθαρρύνεται, καθώς συνέχιση της λειτουργίας της διεργασίας μετά την αποθήκευσή της μπορεί να οδηγήσει σε ασυνέπειες μεταξύ των αρχείων που δημιουργούμε κατά του checkpoint και εξωτερικούς πόρους που μπορεί να χρησιμοποιεί η διεργασία, όπως αρχεία ή TCP συνδέσεις. Γι' αυτό το λόγο επιλέγουμε να τερματίζουμε τη διεργασία μετά την αποθήκευσή της και αμέσως μετά να εκτελούμε την εντολή *criu restore*, ώστε να την επαναφέρουμε.

Όρισμα	**Λειτουργία**
-t <pid>	Το pid της διεργασίας που επιθυμούμε να αποθηκεύσουμε
--shell-job	Απαραίτητο όρισμα για διεργασίες που έχουν δημιουργηθεί από το shell και δεν είναι αρχηγοί του session στο οποίο υπάρχουν. Κατά την επαναφορά, η διεργασία κληρονομεί χαρακτηριστικά όπως session και process group id, από τη διεργασία του CRIU.
--tcp-established	Δηλώνει την αποθήκευση TCP συνδέσεων που τυχόν χρησιμοποιεί η διεργασία
--leave-stopped / --leave-running	Όρίζει την κατάσταση της διεργασίας μετά την ολοκλήρωση της αποθήκευσης της

	κατάστασης της. Με το όρισμα –leave-running η διεργασία συνεχίζει κανονικά τη λειτουργία της, ενώ με το όρισμα –leave-stopped (προεπιλεγμένη λειτουργία) τερματίζει.

Μετά την ολοκλήρωση της λειτουργίας του CRIU μεταβαίνουμε στην κατάσταση Sending_Output όπου στέλνουμε μηνύματα (στο παράδειγμα μας στον client/2), σηματοδοτώντας το πέρας της διαδικασίας του checkpoint. Τέλος, στην κατάσταση Done, αποθηκεύουμε την κατάσταση των script του module και της κοινής ουράς ανταλλαγής μηνυμάτων, ώστε να έχουμε τη δυνατότητα να επανεκκινήσουμε τη διαδικασία εγκατάστασης από αυτό το σημείο.

Η παραγωγή στιγμιότυπου της κατάστασης των scripts είναι μια διαδικασία που γίνεται τοπικά από το κάθε νήμα εκτέλεσης που αντιστοιχεί σε κάθε module. Όπως έχουμε αναφέρει, το κάθε νήμα διατηρεί μια ιδιωτική δομή με το περιεχόμενο και την τρέχουσα κατάσταση των script που καλείται να μεταφέρει και να εκτελέσει στο εικονικό μηχάνημα που διαχειρίζεται. Για τη δρομολόγηση ενός script προς εκτέλεση, το νήμα συμβουλεύεται αυτή τη δομή, επιλέγοντας το script με τον μικρότερη σειρά δρομολόγησης (ετικέτα “seq”) και κατάσταση Pending. Όπως είναι φυσικό, τα περιεχόμενα της συγκεκριμένης δομής μεταβάλλονται όσο η διαδικασία της εγκατάστασης προχωράει και στην περίπτωση σφάλματος σε μεταγενέστερο στάδιο όλα τα νήματα καλούνται να επανέλθουν στην κατάσταση που βρίσκονταν τη στιγμή που επιτελέστηκε το checkpoint, επαναφέροντας το στιγμιότυπο της δομής που έχουν αποθηκευμένο, ώστε μετά τη διόρθωση του σφάλματος να είναι σε θέση να δρομολογήσουν το σωστό script προς εκτέλεση. Η παραγωγή στιγμιότυπου της δομής πραγματοποιείται μετά την επιτυχή εκτέλεση script με ετικέτα criu ή ετικέτα cp_scripts. Η ύπαρξη δύο ετικετών δικαιολογείται αν αναλογιστούμε ότι ένα module οφείλει να αποθηκεύσει την κατάσταση του ακόμα και αν δεν είναι αυτό που επιτελεί τη λειτουργία του checkpoint.

Αντιθέτως, η δημιουργία στιγμιότυπου της ουράς ανταλλαγής μηνυμάτων πραγματοποιείται μόνο από ένα νήμα για μια δεδομένη χρονική στιγμή, καθώς αποτελεί διαμοιραζόμενη δομή ανάμεσα σε όλα τα νήματα. Για να διασφαλίσουμε την ακεραιότητα των δεδομένων, επιλέγουμε το νήμα που καλεί το CRIU. Συγκεκριμένα, διεκδικεί το κλείδωμα της δομής, διασφαλίζοντας ότι κανένα άλλο νήμα δεν θα ανανεώσει ή θα διαβάσει τα περιεχόμενα της ουράς και παράγει ένα ακριβές αντίγραφο των μηνυμάτων που περιέχει. Η δομή της ουράς έχει εμπλουτιστεί με ένα νέο πεδίο που δείχνει στο πιο πρόσφατο αποθηκευμένο στιγμιότυπο, το οποίο και αντικαθίσταται μετά από κάθε checkpoint. Για την επαναφορά, το νήμα που είναι υπεύθυνο για την διαδικασία του Restore αρκεί να μεταβάλλει το δείκτη της αυθεντικής ουράς ώστε να δείχνει στο αποθηκευμένο στιγμιότυπο.

Ο μηχανισμός της ουράς αποτελεί βασικό μηχανισμό για την επικοινωνία και το συγχρονισμό των module της εφαρμογής, όπως περιγράψαμε στο κεφάλαιο 2. Συγκεκριμένα, η έναρξη της εκτέλεσης ενός script δεν ξεκινά παρά μόνο όταν λάβει τα μηνύματα που περιμένει. Όπως είναι φυσικό, επαναφορά της διαδικασίας εγκατάστασης σε προγενέστερο στάδιο συνεπάγεται αποθηκευμένο αντίγραφο της ουράς ακριβώς τη στιγμή που ολοκληρώνεται το checkpoint, δηλαδή αμέσως μόλις το υπεύθυνο νήμα εισάγει στην ουρά τα μηνύματα που σηματοδοτούν το πέρας της διαδικασίας, επιτρέποντας στα υπόλοιπα module να συνεχίσουν τη λειτουργία τους.

3.3 Η διαδικασία Επαναφοράς

Σε αντίθεση με τη διαδικασία παραγωγής στιγμιότυπου του γράφου εκτέλεσης η οποία ορίζεται από το χρήστη σε συγκεκριμένα σημεία, η διαδικασία επαναφοράς έχει δυναμικό χαρακτήρα, καθώς ενεργοποιείται όποτε εντοπίζεται σφάλμα κατά την εκτέλεση ενός script οποιουδήποτε module. Ο μηχανισμός επαναφοράς οφείλει να εισάγει τις εξής λειτουργίες:

- Παρακολούθηση της διαδικασίας του deployment για τυχόν σφάλματα.
- Ενεργοποίηση του μηχανισμού σε περίπτωση σφάλματος και παύση της διαδικασίας του deployment.
- Επαναφορά του γράφου εκτέλεσης σε προηγούμενη αποθηκευμένη κατάσταση.
- Πληροφόρηση των νημάτων εκτέλεσης της Αύρας μόλις η επαναφορά ολοκληρωθεί και επανεκκίνηση του deployment από προγενέστερο στάδιο.

Παρακάτω θα εμβαθύνουμε στις παραπάνω λειτουργίες, αναλύοντας τις επεκτάσεις που πραγματοποιήσαμε στον υπάρχοντα μηχανισμό της Αύρας.

3.3.1 Επέκταση του υπάρχοντος μηχανισμού της Αύρας

Για την καλύτερη περιγραφή του μηχανισμού επαναφοράς και του πως αυτός ενεργοποιείται, θα ξεκινήσουμε από το στάδιο κατασκευής και αρχικοποίησης των νημάτων εκτέλεσης των module της εφαρμογής. Όπως έχουμε αναφέρει στο Κεφάλαιο 2, η Αύρα κατασκευάζει ένα νήμα εκτέλεσης για κάθε module. Η διαδικασία επαναφοράς απαιτεί την εκτέλεση του CRIU στο εικονικό μηχάνημα του κάθε module του οποίου η κατάσταση είχε αποθηκευτεί σε προηγούμενο στάδιο. Η απρόβλεπτη φύση των παροδικών σφαλμάτων και η

ανάγκη για συνεχή παρακολούθηση της διαδικασίας μας οδήγησε στην εισαγωγή επιπλέον daemon threads και συγκεκριμένα ένα για κάθε module στο οποίο εκτελούμε το CRIU, που είναι υπεύθυνα για την παρακολούθηση του deployment και εκκίνηση της διαδικασίας επαναφοράς σε περίπτωση σφάλματος. Συγκεκριμένα, τα thread αυτά συνδέονται στα εικονικά μηχανήματα των module πέραν των κλασσικών και περιμένουν μέχρι η Αύρα να αναφέρει ότι συνέβη κάποιο σφάλμα. Πλέον το κάθε νήμα εκτέλεσης, σε περίπτωση ανεπιτυχούς ολοκλήρωσης ενός script αλλάζει την κατάσταση ενός αντικειμένου health flag, τύπου threading Event, που συμβολίζει τη κατάσταση του deployment. Αλλαγή της κατάστασης σηματοδοτεί την έναρξη της διαδικασίας επαναφοράς κατά την οποία τα νήματα εκτέλεσης της Αύρας που είναι υπεύθυνα για την εγκατάσταση των διάφορων module παύουν τη λειτουργία τους και τα νήματα επαναφοράς ξεκινούν την εκτέλεση τους.

Συγκεκριμένα, τα daemon threads, στα οποία στη συνέχεια θα αναφερόμαστε ως watchers, παρακολουθούν την κατάσταση του αντικειμένου και στην περίπτωση που αυτή αλλάξει, τίθεται σε λειτουργία ο μηχανισμός επαναφοράς. Κατά την επαναφορά, ο κάθε watcher δημιουργεί νήματα εργάτες (worker threads), όσα και η πολλαπλότητα του module που καλείται να επαναφέρει σε προγενέστερο στάδιο και περιμένει να εκτελεστούν. Για λόγους ευκολίας θα αναλύσουμε την περίπτωση module πολλαπλότητας:1 (multiplicity:1) και θα αναφερθούμε στις διαφοροποιήσεις που προκύπτουν για διαφορετική πολλαπλότητα σε επόμενη ενότητα. Για την επαναφορά ενός module πολλαπλότητας 1, δημιουργείται ένα worker thread που καλείται να εκτελέσει ένα αρχείο που έχουμε τοποθετήσει στο εικονικό μηχάνημα και παρουσιάζεται στην εικόνα.

```
#!/bin/bash

CRIU=/root/criu-3.4/criu/criu

if pidof <<process we_want_to_restore>> >/dev/null
then

    kill -9 $(pidof <<process we_want_to_restore>>)


fi

cd /root/server-dump
$CRIU restore -vvv -o restore.log --images-dir
/root/server-dump \
--shell-job --tcp-established --restore-detached && echo
OK

exit 0
```

Το παραπάνω script ελέγχει αν η διεργασία που θέλουμε να επαναφέρουμε από αποθηκευμένο στιγμιότυπο λειτουργεί ακόμα και σε αυτή την περίπτωση της στέλνει σήμα τερματισμού. Στη συνέχεια εκτελεί την εντολή *criu restore* παίρνοντας ως όρισμα τον κατάλογο αρχείων που

δημιουργήσαμε κατά τη διαδικασία του checkpoint. Όταν η εντολή *criu restore* ολοκληρωθεί, η διεργασία θα βρίσκεται πάλι σε λειτουργία, στην κατάσταση στην οποία βρίσκονταν όταν συνέβη το checkpoint και το worker thread τερματίζει την εκτέλεση του. Η εντολή *criu restore* δέχεται ένα μεγάλο πλήθος ορισμάτων ανάλογα με τη διεργασία που καλείται να επαναφέρει και σχετίζονται σε μεγάλο βαθμό με τα ορίσματα που χρησιμοποιήθηκαν κατά την εντολή criu dump. Στον παρακάτω πίνακα παρουσιάζονται συνοπτικά μερικά από τα βασικά.

Όρισμα	**Λειτουργία**
--shell-job	Απαραίτητο όρισμα στην περίπτωση που χρησιμοποιήθηκε κατά την εντολή dump.
--tcp-established	Δηλώνει την επαναφορά TCP συνδέσεων που χρησιμοποιούσε η διεργασία κατά τη διάρκεια της αποθήκευσης. Απαραίτητο όρισμα στην περίπτωση που χρησιμοποιήθηκε κατά την εντολή dump.
--images-dir /path/to/images/	Ορίζει την τοποθεσία των αρχείων που παράχθηκαν κατά την εντολή dump
--restore-detached	Το CRIU τερματίζει τη λειτουργία του μετά την ολοκλήρωση του restore και αντί να γίνεται ο πατέρας την restored διεργασίας, την κληρονομεί στην init.

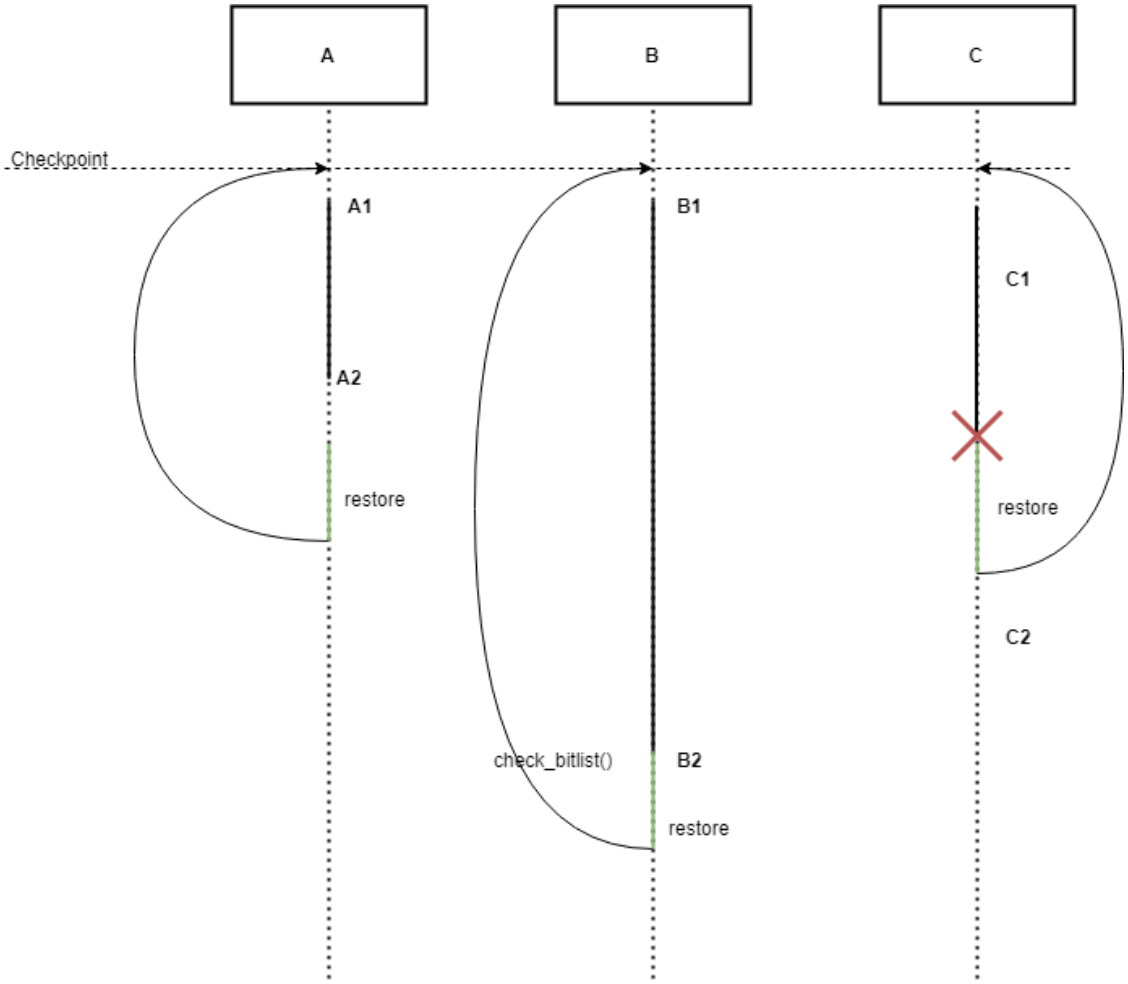
Στη συνέχεια, ο watcher επαναφέρει την αποθηκευμένη ουρά ανταλλαγής μηνυμάτων, ώστε με την συνέχεια της διαδικασίας να μην δρομολογηθούν λάθος script. Τέλος, χρησιμοποιούμε μια λίστα τιμών-σημαιών όπου το κάθε κελί αντιστοιχεί σε ένα νήμα εκτέλεσης για να ενημερώσουμε τα thread αυτά ότι συνέβη επαναφορά σε προγενέστερο στάδιο του γράφου εγκατάστασης. Ο watcher ενημερώνει την τιμή του κάθε κελιού και ύστερα επαναφέρει το αντικείμενο health flag στην αρχική του κατάσταση, σηματοδοτώντας την ολοκλήρωση της διαδικασίας του Restore. Η λειτουργία του watcher δεν τερματίζει καθώς επιστρέφει στο στάδιο παρακολούθησης σφαλμάτων.

Κατά τη διάρκεια της διαδικασίας που περιγράψαμε τα νήματα εκτέλεσης παύουν τη δρομολόγηση καινούργιων script περιμένοντας, να ολοκληρωθεί η επαναφορά. Όταν το health flag επιστρέψει στην κανονική του κατάσταση, το κάθε νήμα συμβουλεύεται τη λίστα που αναφέραμε και σε περίπτωση που η τιμή του αντίστοιχου κελιού έχει αλλάξει επαναφέρει τη δομή των script που είχε αποθηκεύσει κατά τη διαδικασία του checkpoint. Πλέον το κάθε νήμα δεν δρομολογεί σειριακά τα script, αλλά συμβουλεύεται τη δομή και επιλέγει το script με την ελάχιστη σειρά

δρομολόγησης και κατάσταση διάφορη του Done. Έτσι το επόμενο script που θα δρομολογηθεί μετά την επαναφορά, είναι αυτό που εκτελέστηκε αμέσως μετά την ολοκλήρωση του αντίστοιχου checkpoint.

3.3.2 Η χρησιμότητα της λίστας τιμών-σημαιών

Με μια πρώτη ματιά, ο μηχανισμός της λίστας φαντάζει περιττός είναι όμως αναγκαίος όπως φαίνεται από το παρακάτω παράδειγμα.



Εικόνα 3-1

Έστω ότι η εφαρμογή μας αποτελείται από τρία modules A, B και C. Ο B δρομολογεί ένα script μεγάλης διάρκειας εκτέλεσης τη χρονική στιγμή B1, κατά τη διάρκεια του οποίου το script που εκτελεί εκείνη την στιγμή ο C αποτυγχάνει, αλλάζοντας το health flag και πυροδοτώντας το μηχανισμό επαναφοράς. Σε αυτή τη φάση οι A και C δεν δρομολογούν άλλα script, ο watcher δημιουργεί ένα worker thread, όπως περιγράψαμε, ενώ ο B περιμένει την ολοκλήρωση του script που είχε δρομολογήσει. Στην περίπτωση που το worker thread ολοκληρώσει τη λειτουργία του πριν το script του module B, το νήμα που διαχειρίζεται το B δεν θα αντιληφθεί ότι πραγματοποιήθηκε

επαναφορά, καθώς το health flag θα έχει επιστρέψει στην αρχική του κατάσταση. Με τη χρήση της λίστας, διασφαλίζουμε ότι κάθε thread πριν τη δρομολόγηση ενός script θα ελέγξει αν έχει πραγματοποιηθεί επαναφορά σε προηγούμενο στάδιο εκτέλεσης, την οποία δεν αντιλήφθηκε λόγω εκτέλεσης κάποιου script.

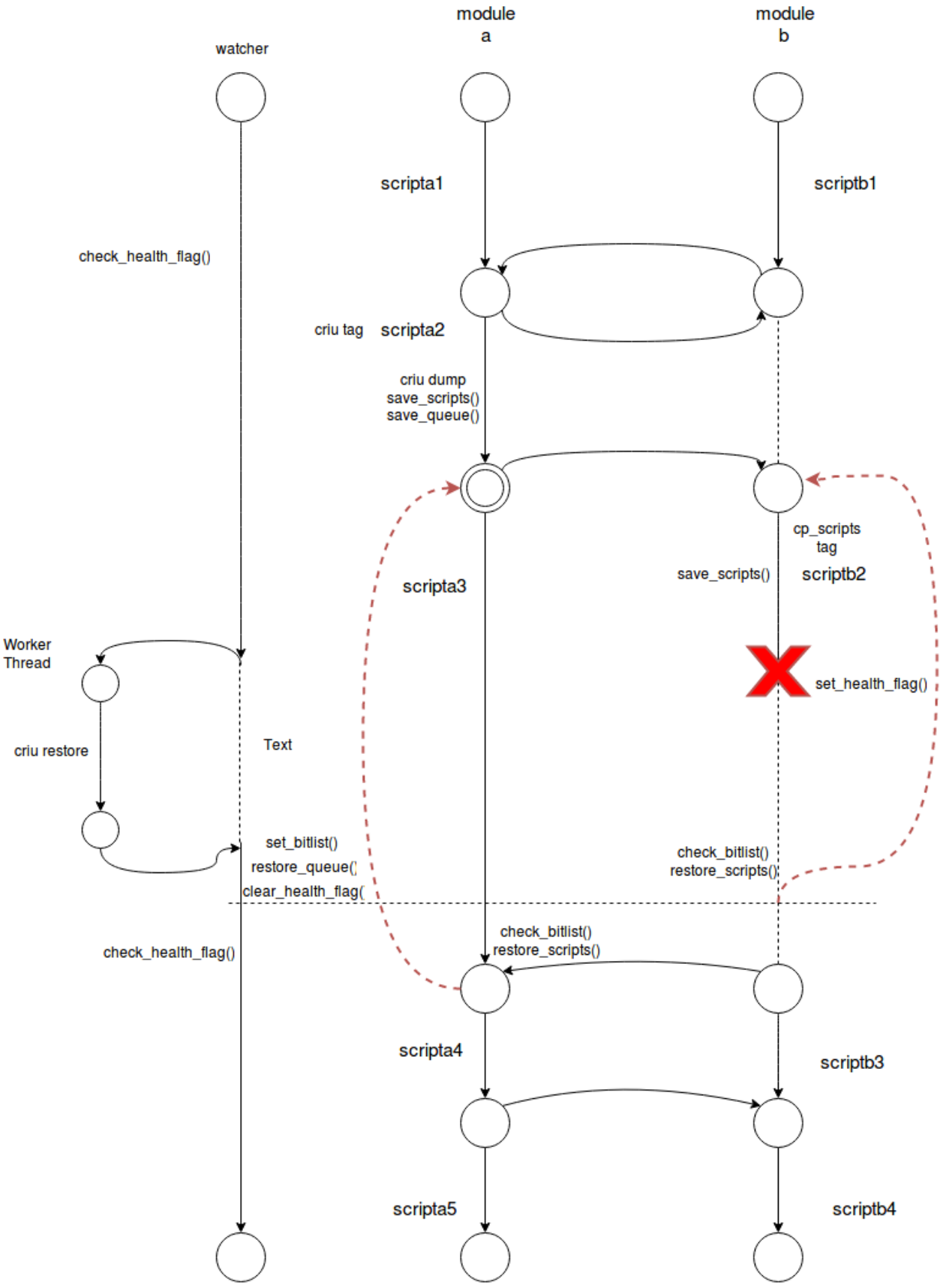
3.4 Ένα Συνολικό σενάριο Εγκατάστασης

Στον γράφο εκτέλεσης της εικόνας 3-2 παρουσιάζουμε ένα σενάριο εγκατάστασης όπου οι μηχανισμοί αποθήκευσης και επαναφοράς φαίνονται σε λειτουργία. Η εφαρμογή αποτελείται από δύο module που αλληλεπιδρούν μεταξύ τους, το a του οποίου την κατάσταση θέλουμε να αποθηκεύουμε και να επαναφέρουμε σε περίπτωση σφάλματος και του b. Οι κατακόρυφες μαύρες ακμές συμβολίζουν την εκτέλεση ενός script και οι καμπυλωτές οριζόντιες την ανταλλαγή μηνυμάτων. Τέλος, οι διακεκομμένες μαύρες ακμές υποδεικνύουν άεργο χρόνο κατά τον οποίο ένα thread περιμένει την άφιξη ενός μηνύματος για να συνεχίσει τη λειτουργία του.

Η Αύρα κατασκευάζει τα νήματα εκτέλεσης a και b καθώς και ένα watcher νήμα, υπεύθυνο για την επαναφορά του module a. Η επιτυχής εκτέλεση των scripta1, scriptb1 σηματοδοτεί την δρομολόγηση του scripta2 το οποίο ενεργοποιεί το μηχανισμό του checkpoint, λόγω της ετικέτας criu. Η κατάσταση του module σώζεται σε αρχεία, ενώ το αντίστοιχο νήμα δημιουργεί αντίγραφα της κατάστασης των αρχείων που καλείται να εκτελέσει και του μηχανισμού της ουράς, αμέσως μόλις εισάγει σε αυτόν το μήνυμα που προορίζεται για το module b. Η διαδικασία συνεχίζεται κανονικά, με την έναρξη των scripta3 και scriptb2, το οποίο πριν μεταβεί σε κατάσταση executing, αποθηκεύει την κατάσταση των script του module b. Κατά την εκτέλεση του scriptb2, παρατηρείται σφάλμα και το thread b αλλάζει την κατάσταση του αντικειμένου health flag. Το watcher thread παρατηρεί αυτή την αλλαγή, ξυπνάει και δημιουργεί το worker thread που θα καλέσει το CRIU για το module a. Κατά τη διάρκεια της διαδικασίας *criu restore*, το νήμα b αναμένει την αλλαγή του αντικειμένου health flag ώστε να συνεχίσει να δρομολογεί scripts ενώ το νήμα a συνεχίζει την εκτέλεση του rerstore χωρίς να έχει αντιληφθεί ότι επιτελείται η διαδικασία του restore.

Το worker thread ολοκληρώνει την λειτουργία του και επιστρέφει δίνοντας τη σκυτάλη στο watcher thread που ενημερώνει τη δομή της λίστας, επαναφέρει την κατάσταση της ουράς στην κατάσταση που βρίσκονταν κατά το checkpoint και αλλάζει την κατάσταση του health flag ανακοινώνοντας το τέλος του restore. Οι κόκκινες διακεκομμένες ακμές συμβολίζουν την αλλαγή της κατάστασης των module μετά το restore. Συγκεκριμένα το thread b, παρατηρεί την αλλαγή του health flag, ελέγχει το κελί της λίστας που του αντιστοιχεί και επαναφέρει το τοπικό αντίγραφο της κατάστασης των script που είχε δημιουργήσει αμέσως μετά το checkpoint. Στην συνέχεια

δρομολογεί την ξανά την εκτέλεση του scriptb2. Από την άλλη το νήμα a τελειώνει την εκτέλεση του scripta3 με επιτυχία και ετοιμάζεται να δρομολογήσει το scripta4. Όμως ελέγχοντας την τιμή του κελιού της λίστας που του αντιστοιχεί, ενημερώνεται ότι ο μηχανισμός της επαναφοράς έχει επαναφέρει την κατάσταση της εγκατάστασης σε προγενέστερο στάδιο. Έτσι, και το νήμα a, επαναφέρει το παλαιότερο αντίγραφο της κατάστασης των script και δρομολογεί ξανά για εκτέλεση το scripta3 και η διαδικασία συνεχίζεται κανονικά.



Εικόνα 3-2: Ένα συνολικό σενάριο εκτέλεσης εγκατάστασης

3.5 Η περίπτωση πολλαπλότητας ενός module

Η πολλαπλότητα (multiplicity) ενός module δηλώνει την ύπαρξη ενός η περισσοτέρων αντιγράφων του. Αν και το κάθε module εγκαθίσταται σε διαφορετικό εικονικό μηχάνημα αλληλεπιδρά με τα άλλα module της εφαρμογής όπως ακριβώς ορίζεται στο αρχείο περιγραφής για το πρωτότυπο. Για παράδειγμα, έστω το αρχείο περιγραφής της εικόνας. Η Αύρα θα δημιουργήσει δύο ξεχωριστά module για το aura-two, με όνομα aura-two1 και aura-two2. Το πρώτο script του aura-one θα παράξει έξοδο και για τα δύο module και αντίστοιχα το script3.sh του aura-one θα περιμένει είσοδο και από τα δυο module πριν ξεκινήσει την εκτέλεση του.

```
       "modules": [{
        "name": "aura-one",
        "scripts": [{
               "seq": 1,
               "file": "one/script1.sh",
               "output": ["aura-two/2"]
        },
        {
               "seq": 2,
               "file": "one/script2.sh"
        },
        {
               "seq": 3,
               "file": "one/script3.sh",
               "input": ["aura-two/2"]
        }]
    },
    {
        "name": "aura-two",
        "multiplicity":2,
        "scripts": [{
               "seq": 1,
               "file": "two/script1.sh"
        },
        {
               "input": ["aura-one/1"],
               "seq": 2,
               "file": "two/script2.sh",
               "output": ["aura-one/3"]
        }]
    }]
```

Για τους μηχανισμούς παραγωγής στιγμιότυπου και επαναφοράς, προσεγγίζουμε τα module πολλαπλότητας σαν ένα ενιαίο. Συγκεκριμένα, για τη διαδικασία του checkpoint, αν ένα module πολλαπλότητας καλείται να αποθηκεύσει την κατάσταση του θεωρούμε ότι όλα τα αντίγραφα θα προβούν στην ίδια ενέργεια. Τα module ξεκινούν την εκτέλεση του CRIU παράλληλα. Οι υπόλοιπες ενέργειες που περιγράψαμε σειριοποιούνται, εισάγοντας κλειδώματα τύπου Reentrant

Lock στη δομή της ουράς ανταλλαγής μηνυμάτων. Μετά το τέλος της εκτέλεσης του CRIU το κάθε νήμα που αντιστοιχεί σε module που αποθηκεύουμε, διεκδικεί το κλείδωμα της ουράς, ειδοποιεί τα υπόλοιπα module, αποθηκεύει την κατάσταση των script του και της ουράς και απελευθερώνει το κλείδωμα. Αυτό επαναλαμβάνεται μέχρι όλα τα νήματα να αποθηκεύσουν την κατάσταση των script τους. Ιδανικά η κατάσταση της ουράς θα έπρεπε να αποθηκεύεται μόνο μια φορά, από το τελευταίο νήμα που θα αποκτήσει την κυριότητα το κλειδώματος αλλά η δομή είναι σχετικά μικρή και το υπολογιστικό κόστος πρακτικά αμελητέο, ενώ όσον αφορά τη μνήμη δεν έχουμε πολλαπλά αντίγραφα αφού όλα τα νήματα γράφουν στην ίδια τοποθεσία, αντικαθιστώντας το προηγούμενο αντίγραφο.

Τα reentrant locks είναι ένα εργαλείο συγχρονισμού που προσφέρει η python και υλοποιούν τη λειτουργικότητα των κλασσικών κλειδωμάτων, με τη διαφορά ότι μπορούν να αποκτηθούν πολλές φορές από το ίδιο νήμα εκτέλεσης. Πιο συγκεκριμένα, οι καταστάσεις ενός κλειδώματος (κλειδωμένο / ξεκλείδωτο) εμπλουτίζονται με έννοιες όπως κυριότητα του κλειδώματος και επίπεδο εμφώλευσης. Τα νήματα επιχειρούν να αποκτήσουν την κυριότητα του κλειδώματος μέσω της αντίστοιχης μεθόδου του. Αν το κλείδωμα είναι ξεκλείδωτο ένας από όλους τους διεκδικητές γίνεται ο κυρίαρχος του κλειδώματος, ενώ οι υπόλοιποι μπλοκάρουν, περιμένοντας την αποδέσμευση του. Σε αντίθεση με τα κλασσικά κλειδώματα, τα reentrant locks δίνουν τη δυνατότητα στον κυρίαρχο του κλειδώματος να εκτελέσει επιπλέον εμφωλευμένες κλήσεις δέσμευσης και αποδέσμευσης του, πράξη που στην περίπτωση των κλασσικών κλειδωμάτων δεν είναι εφικτή. Για παράδειγμα έστω ότι ένα thread εκτελεί το κομμάτι κώδικα της εικόνας 3-3. Το thread αναστέλλει τη λειτουργία του μέχρι να αποκτήσει την κυριότητα του κλειδώματος. Στην περίπτωση που πρόκειται για κλασσικό κλείδωμα η δεύτερη απόπειρα κλειδώματος θα μπλοκάρει απεριόριστα παρά το γεγονός ότι εκτελείται από το thread που έχει την κυριότητα του κλειδώματος. Αντιθέτως, στην περίπτωση των Reentrant Locks η κλήση θα εκτελεστεί με επιτυχία καθώς το thread έχει αποκτήσει ήδη το κλείδωμα. Σημειώνεται ότι κάθε απόπειρα κλειδώματος πρέπει να συνοδεύεται απο την αντίστοιχη κλήση αποδέσμευσης, ενώ το κλείδωμα απελευθερώνεται όταν εκτελεστεί και η τελευταία αποδέσμευση.

```
lock.aquire()
#lock is now locked
lock.aquire()

#do stuff

lock.release()
lock.release()

#lock is now unlocked
```

Εικόνα 3-3

Ο λόγος που τα επιλέξαμε για να αντικαταστήσουμε τα κλασσικά κλειδώματα της ουράς είναι επειδή θέλαμε τα νήματα που υλοποιούν τον μηχανισμό του checkpoint να έχουν αποκλειστικότητα της ουράς για κάποιες περιόδους και παράλληλα να μπορούν να χρησιμοποιήσουν τις μεθόδους της, οι οποίες επίσης διεκδικούν τα ίδια κλειδώματα. Επιτυγχάνουμε λοιπόν τη λειτουργικότητα που επιθυμούμε χωρίς να είναι αναγκαίο να τροποποιήσουμε σε βάθος τον αντίστοιχο κώδικα, ή να εισάγουμε επιπλέον κλειδώματα στη δομή.

Για τον μηχανισμό της επαναφοράς, επιθυμούμε ένα watcher thread να διαχειρίζεται όλα τα module πολλαπλότητας. Γι αυτό το λόγο τροποποιήσαμε το στάδιο αρχικοποίησης των νημάτων της Αύρας. Όπως έχουμε αναλύσει, η Αύρα παράγει ένα αντικείμενο για κάθε module της εφαρμογής, μεθόδους του οποίου εκτελεί το αντίστοιχο νήμα εκτέλεσης. Module με την ετικέτα coordinator στο αρχείο περιγραφής δηλώνουν δημιουργία ενός watcher thread υπεύθυνο για τις ενέργειες του CRIU στο αντίστοιχο module σε περίπτωση σφάλματος. Επεκτείνοντας την αρχική ιδέα, με τη δημιουργία ενός watcher thread, που πρακτικά υλοποιεί τη διαδικασία της επαναφοράς μιας εφαρμογής, διατρέχουμε όλα τα υπόλοιπα module και σε περίπτωση ύπαρξης αντιγράφων λόγω πολλαπλότητας δημιουργούμε μια λίστα αναφορών στα αντίστοιχα αντικείμενα, ως όρισμα του watcher thread. Με αυτόν τον τρόπο κατα τη διάρκεια του deployment, ένας watcher παρακολουθεί για τυχόν σφάλματα που επηρεάζουν όλα τα module-αντίγραφα και κατά την εκκίνηση του μηχανισμού επαναφοράς, δημιουργεί νήματα-εργάτες για το δικό του module και αυτά των αντιγράφων, μέσω της λίστας ορισμάτων τα οποία και περιμένει να ολοκληρώσουν το έργο τους. Ο watcher προχωρά στις ενέργειες επαναφοράς της ουράς ενημέρωσης της λίστας και αλλαγή της κατάστασης του health flag μόνο όταν όλα τα thread-εργάτες ολοκληρώσουν την εκτέλεση τους, επιτυγχάνοντας έτσι την παραλληλοποίηση της διαδικασίας σε κομμάτια που είναι εφικτό και συνέχιση της μόνο όταν όλα τα module έχουν επανέλθει σε λειτουργική κατάσταση.

3.5.1 Αστοχίες του CRIU

Όπως έχουμε αναφέρει το CRIU επιχειρεί να επαναφέρει μια διεργασία με το pid που είχε όταν αποθηκεύτηκε η κατάσταση της. Στο λειτουργικό σύστημα Linux το δέντρο διεργασιών μεταβάλλεται διαρκώς χωρίς απαραίτητα τη μεσολάβηση του χρήστη. Για παράδειγμα, ο πυρήνας δημιουργεί kworker processes απαραίτητες για τη λειτουργία του συστήματος (πχ χειρισμός Interrupts ή system calls). Το γεγονός αυτό, σε συνδυασμό με τον τυχαίο τρόπο με τον οποίο ο πυρήνας αναθέτει pid σε καινούργιες διεργασίες εγγυμονεί τον κίνδυνο αποτυχίας της εντολής *criu restore* σε περίπτωση που το pid της διεργασίας τη στιγμή που συνέβη το *criu dump* πλέον έχει

ανατεθεί σε κάποια άλλη. Η προφανής λύση είναι ο τερματισμός της διεργασίας με το ζητούμενο pid, το οποίο όμως δεν είναι σωστό για τη σταθερότητα του συστήματος, ούτε είναι πάντα εφικτό.

Μια άλλη προσέγγιση είναι η χρήση των Linux namespaces [18]. Τα namespaces είναι χαρακτηριστικό του πυρήνα που προσφέρει απομόνωση και εικονικούς πόρους συστήματος, όπως pids, hostnames, συνδεσιμότητα και συστήματα αρχείων. Διαφορετικοί πόροι σχετίζονται με διαφορετικό τύπο namespace. Κάθε διεργασία σχετίζεται με το namespaces στα οποία δημιουργείται και μπορεί να χρησιμοποιεί τους πόρους που σχετίζονται με αυτά. Έτσι, εκτελώντας τη διεργασία του CRIU σε ένα καινούργιο pid namespace της παρέχουμε μια απομονωμένη οπτική του συστήματος, όπου όλα τα pid είναι διαθέσιμα. Πιο συγκεκριμένα, δημιουργούμε καινούργια mount και pid namespaces και στη συνέχεια δημιουργούμε τη διεργασία πατέρα, αντίστοιχη της init . Στη συνέχεια προσαρτούμε το /proc σύστημα αρχείων στο καινούργιο namespace καθώς είναι απαραίτητο για τη λειτουργία του CRIU. Τέλος, εκτελούμε την εντολή criu restore επαναφέροντας την αποθηκευμένη διεργασία στο καινούργιο namespace.

Η παραπάνω διαδικασία ενσωματώνεται στο criu-ns helper script [19] το οποίο και χρησιμοποιούμε τόσο για την παραγωγή στιγμιότυπου μιας διεργασίας όσο και για την επαναφορά, λύνοντας έτσι προβλήματα που προκύπτουν από τη σύγκρουση pids.

3.6 Σενάριο χρήσης: Ζωντανή μεταφορά εφαρμογής σε νέο εικονικό μηχάνημα

Σε αυτήν την ενότητα παρουσιάζεται ένα σενάριο χρήσης της έκδοσης της Αύρας που προτείνουμε που ξεφεύγει από την έννοια του application deployment. Εκμεταλλευόμενοι τη δυνατότητα σύνδεσης της Αύρας σε προϋπάρχοντα εικονικά μηχανήματα μέσω του API του Openstack και της λειτουργικότητας που εισάγει το CRIU πραγματοποιούμε ζωντανή μεταφορά μιας εφαρμογής σε νέο εικονικό μηχάνημα μεγαλύτερων δυνατοτήτων. Πιο συγκεκριμένα, έχοντας πρώτα στήσει μια εφαρμογή client server αρχιτεκτονικής βασισμένη στο εργαλείο redis, επιχειρούμε τη μεταφορά του module server σε νέο εικονικό μηχάνημα μεγαλύτερης μνήμης και εδραίωση συνδεσιμότητας με το module client. Σκοπός μας είναι η αύξηση της διαθέσιμης μνήμης της εφαρμογής, χωρίς να είναι αναγκαίο να χάσουμε τα δεδομένα που είχε στη μνήμη της εξαρχής (application level resizing).

Το παραπάνω είναι εφικτό για δύο λόγους. Πρώτον, το CRIU μας δίνει τη δυνατότητα να αποθηκεύσουμε τη κατάσταση της μνήμης της εφαρμογής και να την επαναφέρουμε σε νέο εικονικό μηχάνημα. Δεύτερον, μέσω του μηχανισμού της Αύρας και του γράφου εξαρτήσεων που υλοποιεί, εκτελούμε τα scripts που χρειάζονται για τη διαδικασία που περιγράψαμε με τη σειρά που απαιτείται στα διάφορα module που απαρτίζουν την εφαρμογή, ώστε μετά τη μεταφορά, η εφαρμογή να είναι λειτουργική. Για παράδειγμα, αφού πραγματοποιηθεί η επαναφορά της εφαρμογής σε νέο εικονικό μηχάνημα, το module client πρέπει να ενημερωθεί με την IP του νέου μηχανήματος όπου λειτουργεί η εφαρμογή. Τα βήματα της διαδικασίας παρουσιάζονται συνοπτικά παρακάτω:

- Παραγωγή στιγμιότυπου του module server και αποθήκευση του στο δίσκο.
- Παραγωγή ζεύγους δημοσίου κλειδιού και αποστολή του στο καινούργιο module.
- Αντιγραφή των αρχείων που παράχθηκαν κατά την αποθήκευση της κατάστασης του redis στο καινούργιο module.
- Ενημέρωση του κατάλληλου image file με την IP διεύθυνση του καινούργιου module.
- Επαναφορά της εφαρμογής στο καινούργιο εικονικό μηχάνημα.
- Αποστολή της καινούργιας διεύθυνσης IP στο module client.
- Τερματισμός της παλιάς εφαρμογής redis server.

Η παραγωγή στιγμιότυπου γίνεται με χρήση της εντολής *criu dump*, όπως έχουμε περιγράψει και παραπάνω. Για την παραγωγή ζεύγους δημοσίου κλειδιού χρησιμοποιούμε την εντολή *ssh-keygen* με τα κατάλληλα ορίσματα. Το δημόσιο και το ιδιωτικό κλειδί αποστέλλονται μέσω του μηχανισμού της ουράς ως είσοδο στο αντίστοιχο configuration script του καινούργιου module. Αφού γίνουν οι απαραίτητες ρυθμίσεις τα αρχεία που παρήχθησαν κατά την αποθήκευση της κατάστασης της διεργασίας αντιγράφονται στο καινούργιο εικονικό μηχάνημα μέσω ssh. Όταν όλα τα αρχεία μεταφερθούν επιτυχώς, ενημερώνουμε το image file files.img. Το αρχείο αυτό περιέχει απαραίτητες πληροφορίες για τους πόρους που χρησιμοποιεί η διεργασία. Πιο συγκεκριμένα, μας ενδιαφέρουν εγγραφές που αναφέρονται σε sockets τύπου PF_INET που υλοποιούν το πρωτόκολλο IP. Τέτοιες εγγραφές περιέχουν ως διεύθυνση πηγής (source address) την IP διεύθυνση του εικονικού μηχανήματος όπου αρχικά εκτελούνταν η διεργασία redis server. Επαναφορά της διεργασίας στο νέο μηχάνημα σε αυτή τη φάση θα κατέληγε σε αποτυχία, αφού το CRIU δεν θα μπορούσε να επαναφέρει τα αντίστοιχα sockets καθώς η διεύθυνση IP του μηχανήματος έχει πλέον αλλάξει.

Ενημερώνοντας το πεδίο διεύθυνσης πηγής (source address) με την IP του καινούργιου μηχανήματος όπου θέλουμε να επαναφέρουμε την εφαρμογή, δίνουμε την εντύπωση στο CRIU ότι το socket υπήρχε κατά τη φάση της αποθήκευσης και έτσι δημιουργείται κατά την επαναφορά. Για την τροποποίηση του αρχείου files.img χρησιμοποιούμε το εργαλείο CRIT [20] το οποίο είναι περιλαμβάνεται στην εγκατάσταση του CRIU. Πιο συγκεκριμένα, το files.img είναι αρχείο γλώσσας μηχανής και τύπου google protocol buffer. Μέσω της εντολής *crit show* μετατρέπουμε το criu binary σε τύπου json, ώστε να είναι εύκολα προσπελάσιμο και κατανοητό. Στη συνέχεια, αντικαθιστούμε όλες τις εγγραφές που αναφέρονται στην διεύθυνση του παλιού μηχανήματος με την IP του καινούργιου και τέλος με την εντολή *crit encode* μετατρέπουμε το αρχείο json που τροποποιήσαμε πάλι σε τύπου criu binary, ώστε να είναι προσπελάσιμο απο το CRIU.

Για το τελευταίο βήμα της διαδικασίας, επαναφέρουμε τη διεργασία redis server μέσω της εντολής *criu restore* στο καινούργιο εικονικό μηχάνημα και μόλις ολοκληρωθεί η επαναφορά, αποστέλλουμε την καινούργια διεύθυνση IP στο module client. Πλέον ο client μπορεί να επικοινωνήσει με το καινούργιο server και μπορούμε να τερματίσουμε τη διεργασία στο παλιό μηχάνημα.

Βασική προϋπόθεση για την ομαλή λειτουργία όσων περιγράψαμε παραπάνω, είναι το ίδιο σύστημα αρχείων μεταξύ του εικονικού μηχανήματος στο οποίο εκτελούνταν αρχικά η διεργασία redis server και του μηχανήματος στο οποίο την επαναφέρουμε. Αρχεία και πόροι που χρησιμοποιούνταν από τη διεργασία κατά τη στιγμή της αποθήκευσης της κατάστασης της οφείλουν να υπάρχουν και στο καινούργιο μηχάνημα για επιτυχή επαναφορά. Για παράδειγμα, στο αρχείο files.img εκτός από περιγραφητές αρχείων που αναφέρονται σε sockets, εγγραφές τύπου

“reg” αναφέρονται σε τοποθεσίες αρχείων που χρησιμοποιούσε η διεργασία, όπως το εκτελέσιμο της εφαρμογής, διάφορες βιβλιοθήκες κλπ. Όλες αυτές οι τοποθεσίες αρχείων πρέπει να είναι συμβατές με αυτές του καινούργιου εικονικού μηχανήματος. Γι’ αυτό το λόγο έχουμε δημιουργήσει όλα τα εικονικά μηχανήματα βασισμένοι στο ίδιο στιγμιότυπο εικονικής μηχανής που έχουμε ανεβάσει στον Openstack εξασφαλίζοντας έτσι ότι όλα τα μηχανήματα έχουν πανομοιότυπο σύστημα αρχείων. Αντίθετα, μπορούμε ελεύθερα να επιλέξουμε διαφορετικό flavor για κάθε εικονικό μηχάνημα ορίζοντας έτσι διαφορετικά χαρακτηριστικά σε ότι αφορά τις δυνατότητες, όπως RAM, cpu, disk space.

Κατά την εκτέλεση του σεναρίου της ζωντανής μεταφοράς, παρατηρήθηκε ότι η επιτυχία της επαναφοράς της εφαρμογής εξαρτάται και από τα χαρακτηριστικά του επεξεργαστή των μηχανημάτων. Πιο συγκεκριμένα, κατά τη δημιουργία νέων εικονικών μηχανών ο Openstack επιλέγει αυθαίρετα μεταξύ διαφορετικών availability zones. Τα availability zones είναι στην ουσία σύνολα ξενιστών (hosts), οι οποίοι φιλοξενούν τις εικονικές μηχανές.

Το CRIU κατά τη φάση της αποθήκευσής της κατάστασης μιας εφαρμογής στο δίσκο, παράγει μεταξύ άλλων και ένα αρχείο με τις δυνατότητες του επεξεργαστή. Κατά την επαναφορά σε καινούργιο μηχάνημα, ελέγχει αν ο νέος επεξεργαστής υποστηρίζει τις ίδιες δυνατότητες, ώστε η εφαρμογή να μπορεί να λειτουργήσει κανονικά. Στις περισσότερες περιπτώσεις δεν υπάρχει πρόβλημα, εκτός αν κάποια εφαρμογή έχει μεταγλωττιστεί με κάποια βελτιστοποίηση (optimization) που βασίζεται σε ιδιαίτερα χαρακτηριστικά του επεξεργαστή. Στην περίπτωση μεταφοράς της διεργασίας redis server από εικονικό μηχάνημα του temporary availability zone σε νέο μηχάνημα που ανήκει στο core, παρατηρήθηκε ότι το CRIU αρνείται να επαναφέρει τη διεργασία λόγω ασυμβατότητας του floating point unit μεταξύ των επεξεργαστών και συγκεκριμένα της δυνατότητας xsave. Για να προσπεράσουμε αυτόν τον περιορισμό υποδεικνύουμε στο CRIU να αγνοήσει τη συγκεκριμένη δυνατότητα, με το συμπληρωματικό όρισμα –cpu-cap=^fpu κατά τη διαδικασία της επαναφοράς.

4 Πειραματική αξιολόγηση

Σε αυτό το κεφάλαιο, επιχειρούμε να αξιολογήσουμε την απόδοση του συστήματος ανάνηψης της διαδικασίας εγκατάστασης μιας εφαρμογής από παροδικά σφάλματα. Συγκεκριμένα, επιθυμούμε να δείξουμε ότι το σύστημα που προτείνουμε:

- Υποστηρίζει την αποθήκευση και την επαναφορά εφαρμογών διαφόρων μεγεθών σε ικανοποιητικό χρόνο.
- Λειτουργεί σε ικανοποιητικό χρόνο για εφαρμογές που αποτελούνται από module πολλαπλότητας μεγαλύτερης του ενός.
- Είναι ορθό, παράγοντας εν τέλει το σωστό αποτέλεσμα για πολύπλοκους γράφους εκτέλεσης ενώ απόδοσή του είναι αποδεκτή για διαφορετικούς ρυθμούς σφαλμάτων.
- Δεν αφορά μόνο μια συγκεκριμένη εφαρμογή, καθώς διεξήχθησαν πειράματα για διαφορετικές εφαρμογές χωρίς να απαιτείται τροποποίηση του πηγαίου κώδικα.

4.1 Περιβάλλον διεξαγωγής πειραμάτων

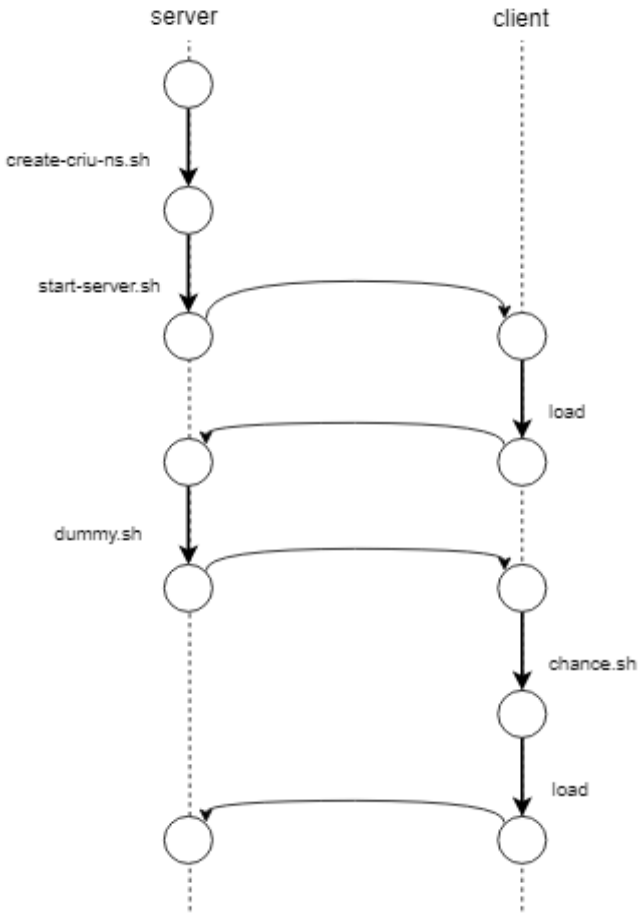
Για την πειραματική αξιολόγηση της εργασίας μας και θέλοντας να εξετάσουμε τη συμπεριφορά του προτεινόμενου μηχανισμού σε διαφορετικές τεχνολογίες storage, χρησιμοποιήσαμε δυο διαφορετικά περιβάλλοντα εκτέλεσης. Το πρώτο είναι ένα Openstack cluster (version Icehouse) που αποτελείται από 6 nodes που αθροίζουν 160 cores, 512GB RAM, χρησιμοποιούν τοπικά προσαρτημένους δίσκους (κάθε node έχει 2 x 1 SATA 2.5" 1TB σε RAID0). Το δεύτερο περιβάλλον εκτέλεσης, το οποίο χρησιμοποιήθηκε μόνο για τα πειράματα πολλαπλότητας module ίσης με 6, χρησιμοποιεί νεότερη έκδοση Openstack (Pike), αποτελείται από 8 nodes που αθροίζουν 320 cores, 2.5TB RAM και χρησιμοποιεί το κατανεμημένο σύστημα αποθήκευσης CEPH για τις εικόνες των VMs. Το CEPH αποτελείται από 8 OSDs και 1 monitor και κάθε OSD διαθέτει 4 x SATA 3.5" 3TB σε RAID5 και 400GB NVMe για cache. Με την αξιολόγηση του μηχανισμού σε δυο διαφορετικά περιβάλλοντα εκτέλεσης, μπορούμε να δούμε την επιρροή που αυτή ασκεί στις λειτουργίες dump/restore κατά την εγκατάσταση μιας εφαρμογής. Το server module φιλοξενείται σε εικονικό μηχάνημα εξοπλισμένο με 6GB RAM, και 10 GB σκληρού δίσκου. Χρησιμοποιήθηκε η έκδοση του CRIU 3.4 η έκδοση του redis 4.0.1 και η έκδοση του memcached 1.5.3. Όσον αφορά το λειτουργικό σύστημα των εικονικών μηχανών χρησιμοποιήσαμε Ubuntu Linux 16.04 με πυρήνα λειτουργικού 4.8.0-45-generic. Τέλος, για τη διεξαγωγή των πειραμάτων δημιουργήσαμε μια εικόνα εικονικής μηχανής που περιείχε όλα τα παραπάνω, μιας και οι χρόνοι που μας ενδιαφέρει

πρωτίστως να αξιολογήσουμε την απόδοση του συστήματος κατά τις λειτουργίες αποθήκευσης της κατάστασης και επαναφοράς.

Για την επίδειξη και την αξιολόγηση του συστήματος επαναφοράς της διαδικασίας εγκατάστασης μιας εφαρμογής σε προγενέστερο στάδιο που προτείνουμε χρησιμοποιούμε δύο εφαρμογές αποθήκευσης ζευγών κλειδιού-τιμής στη μνήμη, το redis και το memcached. Οι δύο αυτές εφαρμογές είναι παρόμοιες στη χρήση τους και επιλέχθηκαν καθώς στηρίζονται στη μνήμη και όχι στην αποθήκευση δεδομένων στο δίσκο. Ακόμα, αποτελούν χαρακτηριστικά παραδείγματα, καθώς χρησιμοποιούνται ευρέως ως συστήματα κρυφής μνήμης (cache) βελτιώνοντας την απόδοση πληθώρας εφαρμογών, όπως το Facebook το Instagram και το Twitter, καθώς αποθηκεύουν δεδομένα στη μνήμη και όχι στο σύστημα αρχείων, προσφέροντας έτσι γρηγορότερη ανταπόκριση σε αιτήματα σε σχέση με μια βάση δεδομένων.

4.2 Οι διαδικασίες αποθήκευσης και επαναφοράς ως προς το χρόνο

Σκοπός του πρώτου πειράματος είναι η μέτρηση του χρόνου εκτέλεσης του CRIU για την αποθήκευση της κατάστασης και την επαναφορά των δύο εφαρμογών για διαφορετικό όγκο δεδομένων. Ο γράφος εγκατάστασης της εφαρμογής παρουσιάζεται συμπυκνωμένος στην παρακάτω εικόνα, καθώς επαναλαμβάνεται.

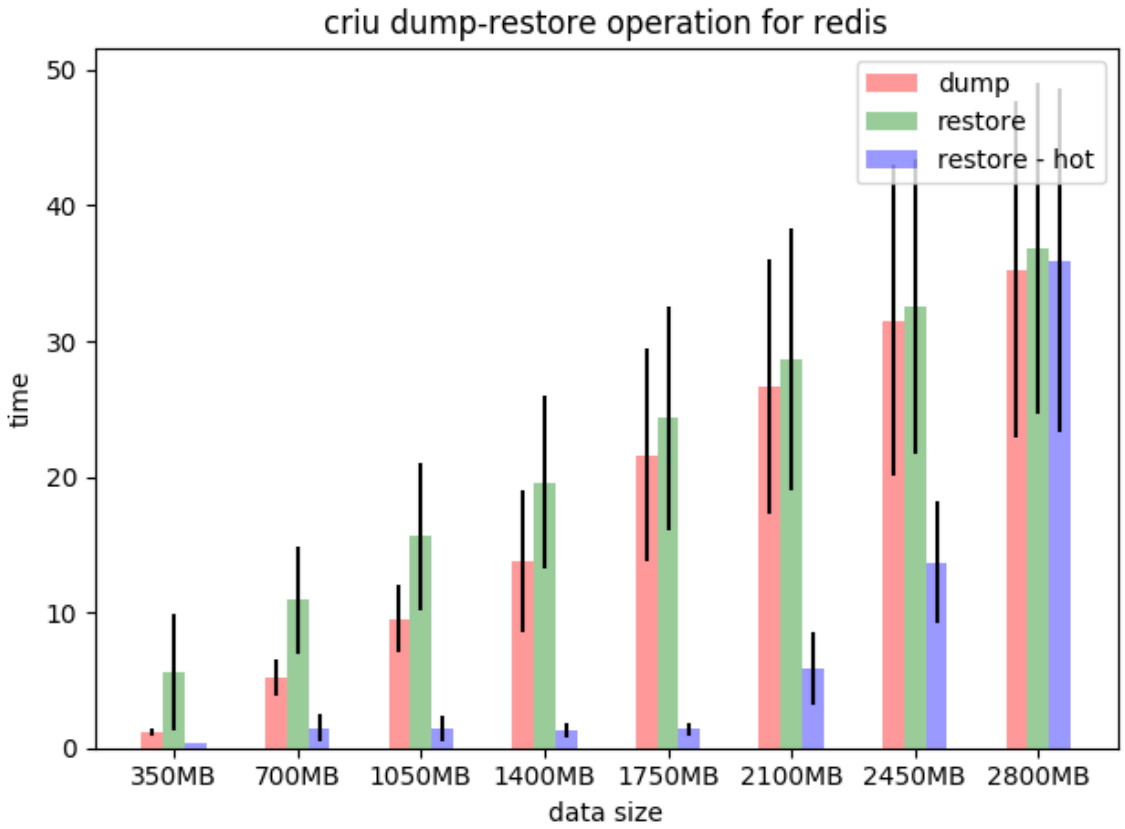


Εικόνα 4-1: Γράφος εκτέλεσης της εγκατάστασης του πειράματος

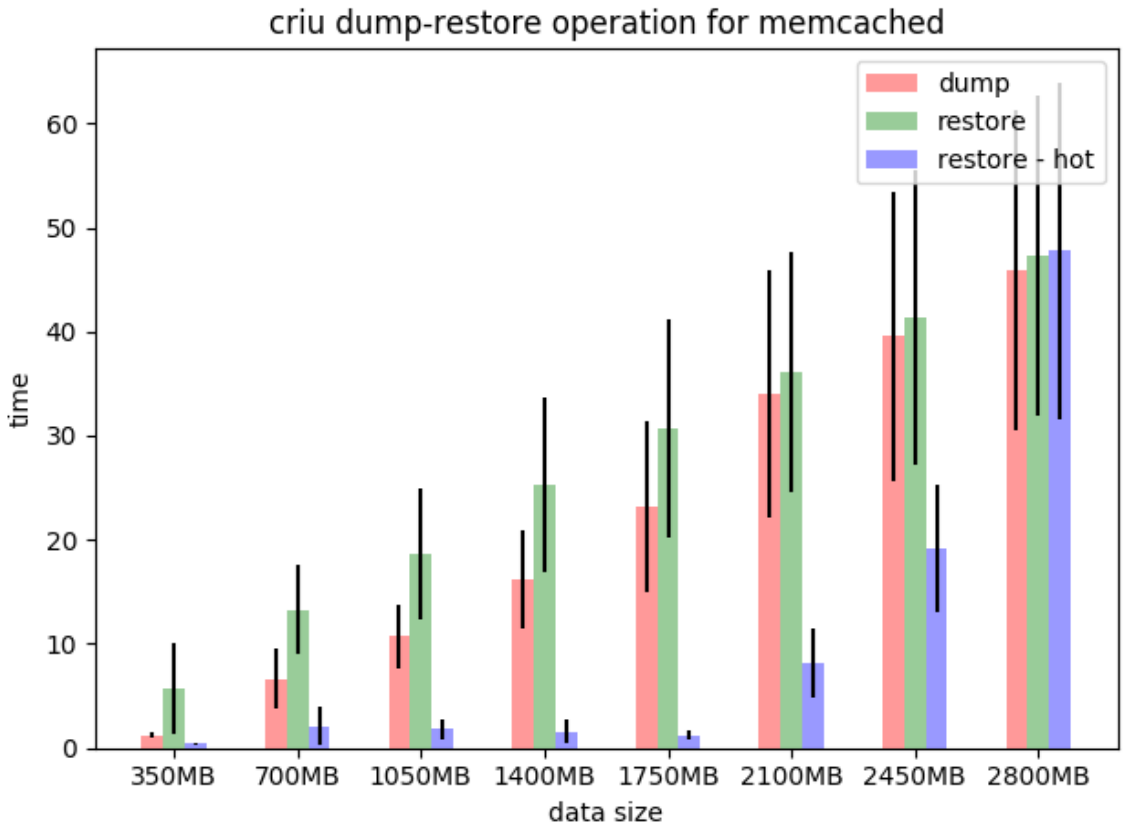
Η εφαρμογή μας αποτελείται από δύο modules, έναν server και έναν client, καθένα από τα οποία εκτελούνται σε δικό τους μηχάνημα. Μετά την εκκίνηση της εφαρμογής του server, η διεύθυνση IP του μηχανήματος αποστέλλεται ως είσοδο στα script του client, μέσω του μηχανισμού ανταλλαγής μηνυμάτων της Αύρας. Το script load, αναλαμβάνει να στείλει δεδομένα της μορφής ζεύγους κλειδιού-τιμής για αποθήκευση στον redis server μέσω της εφαρμογής redis client. Η τιμή του κάθε κλειδιού είναι ένα string μεγέθους 350 KB, ενώ εισάγουμε στο server 1000 κλειδιά, δηλαδή 350 MB δεδομένων. Μετά το πέρας της εκτέλεσης του, δρομολογείται το script dummy.sh του module server, που αν και κενό σε περιεχόμενο, εκκινεί την εκτέλεση του προγράμματος CRIU και την αποθήκευση της κατάστασης της εφαρμογής του server στο δίσκο, λόγω του αντίστοιχου tag στο αρχείο περιγραφής της εφαρμογής. Όπως έχει αναφερθεί στο κεφάλαιο 3, η διαδικασία αποθήκευσης τερματίζει την εκτέλεση της εφαρμογής, οπότε είναι αναγκαίο να την επαναφέρουμε σε λειτουργία μέσω του CRIU. Σημειώνεται ότι κατά τη συγκεκριμένη επαναφορά, τα δεδομένα της εφαρμογής διατηρούνται στη κρυφή μνήμη του συστήματος με αποτέλεσμα πολλές από τις σελίδες μνήμης της εφαρμογής που το CRIU καλείται να επαναφέρει από το δίσκο να υπάρχουν στην κρυφή μνήμη, επιταχύνοντας έτσι τη διαδικασία.

Στη συνέχεια, δρομολογείται η εκτέλεση του script chance.sh του client, το οποίο κατά τη πρώτη του εκτέλεση αποτυγχάνει. Η Αύρα εντοπίζει αυτή την αποτυχία και εκκινεί το μηχανισμό επαναφοράς στο πιο πρόσφατο αποθηκευμένο στιγμιότυπο της εφαρμογής του server. Κατά τη συγκεκριμένη επαναφορά φροντίζουμε να διαγράψουμε τα δεδομένα της κρυφής μνήμης, ώστε να παρατηρήσουμε το χρόνο που απαιτείται από το CRIU για να επαναφέρει την εφαρμογή από το δίσκο. Μετά την επιτυχή επαναφορά του server επαναδρομολογείται η εκτέλεση του script chance.sh, η δεύτερη του εκτέλεση του οποίου επιτυγχάνει. Εν συνεχεία, ο client δρομολογεί ξανά το script load εισάγοντας άλλα 350MB δεδομένων στο server. Η διαδικασία που περιγράφεται παραπάνω επαναλαμβάνεται 8 φορές, όπου σε κάθε βήμα εισάγουμε στο redis server 350 MB επιπλέον δεδομένων, αποθηκεύουμε το καινούργιο στιγμιότυπο στο δίσκο μέσω του CRIU, προσομοιώνουμε ένα τεχνητό σφάλμα και επαναφέρουμε την εφαρμογή μετρώντας τους χρόνους εκτέλεσης του CRIU για την αποθήκευση της κατάστασης της εφαρμογής, την επαναφορά της και την επαναφορά της με άδεια κρυφή μνήμη. Ο γράφος εκτέλεσης της διαδικασίας παρουσιάζεται παρακάτω, όπου παραλείπονται τα επαναλαμβανόμενα βήματα.

Το συγκεκριμένο πείραμα πραγματοποιείται για τις εφαρμογές redis και memcached, ώστε να συγκρίνουμε τη συμπεριφορά τους κατά τις διαδικασίες αποθήκευσης της κατάστασης τους και επαναφοράς.



Εικόνα 4-2: Redis server multiplicity 1, average values



Εικόνα 4-3: Memcached server multiplicity 1, average values

Στις παραπάνω γραφικές παραστάσεις παρουσιάζεται ο χρόνος σε δευτερόλεπτα (κατακόρυφος άξονας) που απαιτείται για την ολοκλήρωση της αποθήκευσης (κόκκινο χρώμα), της επαναφοράς (πράσινο) και της επαναφοράς με γεμάτη μνήμη (μπλε) της εκάστοτε εφαρμογής, ως προς τον όγκο των δεδομένων της μνήμης της (οριζόντιος άξονας).

Από τα παραπάνω διαγράμματα προκύπτει ότι ο χρόνος των διαδικασιών αποθήκευσης της κατάστασης των εφαρμογών και της επαναφοράς τους είναι ανάλογος του πόση μνήμη καταλαμβάνουν, γεγονός απόλυτα λογικό. Κατά την αποθήκευση το CRIU αναλαμβάνει να αντιγράψει τα περιεχόμενα της μνήμης της εφαρμογής σε αρχεία στο δίσκο και κατά την επαναφορά, να διαβάσει τα αρχεία και να τα αντιγράψει στη μνήμη, οπότε ο χρόνος των διαδικασιών αυτών αυξάνεται με το μέγεθος των δεδομένων.

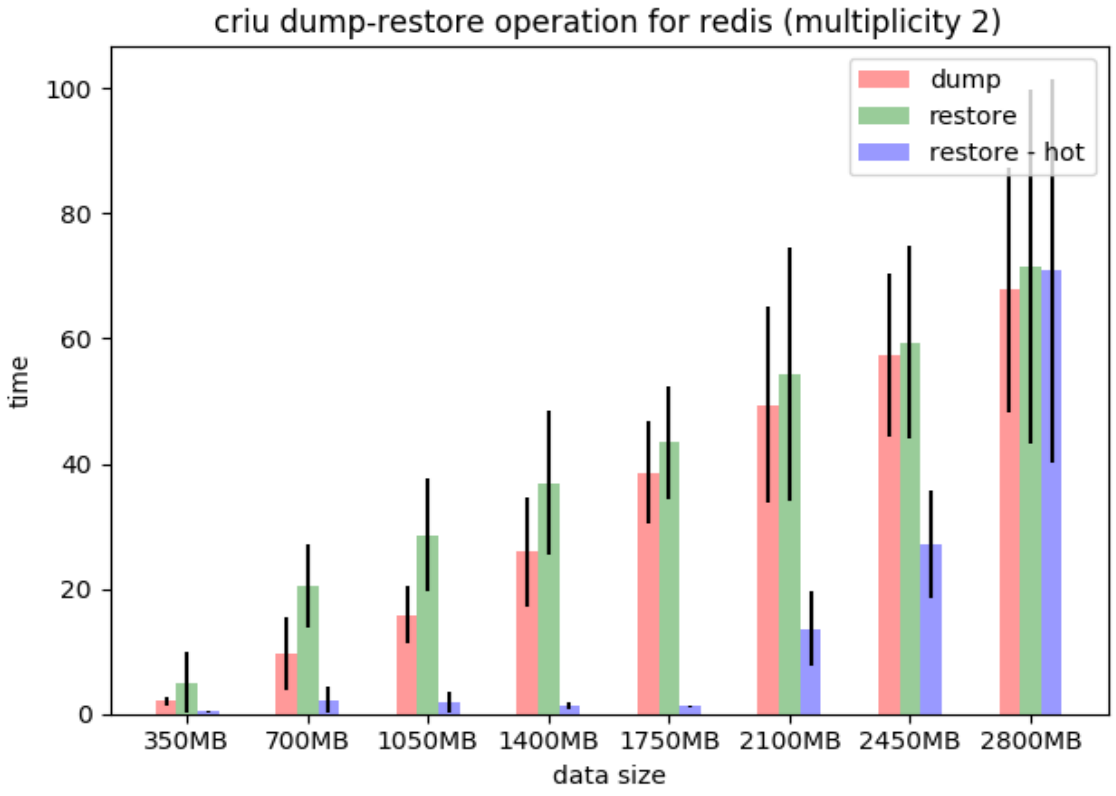
Όσον αφορά την επαναφορά εφαρμογής με ζεστή μνήμη παρατηρούμε ότι πραγματοποιείται αισθητά πιο γρήγορα, καθώς η μεταφορά δεδομένων από τη μνήμη είναι πολύ πιο γρήγορη από την μεταφορά δεδομένων από το δίσκο. Για σχετικά μικρό μέγεθος δεδομένων (~1,75 GB) η επαναφορά εκτελείται σε πολύ σύντομο χρονικό διάστημα. Για μεγαλύτερα μεγέθη δεδομένων, παρατηρείται βαθμιαία αύξηση του χρόνου που απαιτείται, καταλήγοντας οι τρεις διαδικασίες να απαιτούν παραπλήσιο χρόνο.

Συγκρίνοντας το χρόνο που απαιτεί η αποθήκευση με αυτόν της επαναφοράς, παρατηρούμε ότι για μικρά μεγέθη δεδομένων η αποθήκευση εκτελείται πιο γρήγορα, ενώ όσο αυξάνεται το μέγεθος της εφαρμογής οι δυο διαδικασίες απαιτούν παραπλήσιο χρόνο. Κατά την αποθήκευση, τα δεδομένα της μνήμης γράφονται στο δίσκο (write operation) και αντίστοιχα κατά την επαναφορά μεταφέρονται από το δίσκο στη μνήμη (read operation). Αν και ο χρόνος ενός read operation είναι πραγματικός, το λειτουργικό σύστημα υιοθετεί τη πρακτική της καθυστέρησης πολλαπλών write operations με σκοπό τη σειριοποίηση τους, ώστε η συνολική διαδικασία εγγραφής να πραγματοποιηθεί ταχύτερα. Συγκεκριμένα, στη περίπτωση εγγραφής δεδομένων στο δίσκο, το υποσύστημα εισόδου εξόδου (I/O subsystem) αρκείται στο να αναγνωρίσει ότι θα πραγματοποιήσει την εγγραφή, χωρίς ουσιαστικά να την πραγματοποιήσει. Όταν όμως η write cache γεμίσει το σύστημα είναι αναγκασμένο να πραγματοποιήσει την πραγματική εγγραφή των δεδομένων στο δίσκο. Γι αυτό το λόγο, για μεγάλο μέγεθος δεδομένων παρατηρείται παραπλήσιος χρόνος για την ολοκλήρωση των δύο διαδικασιών.

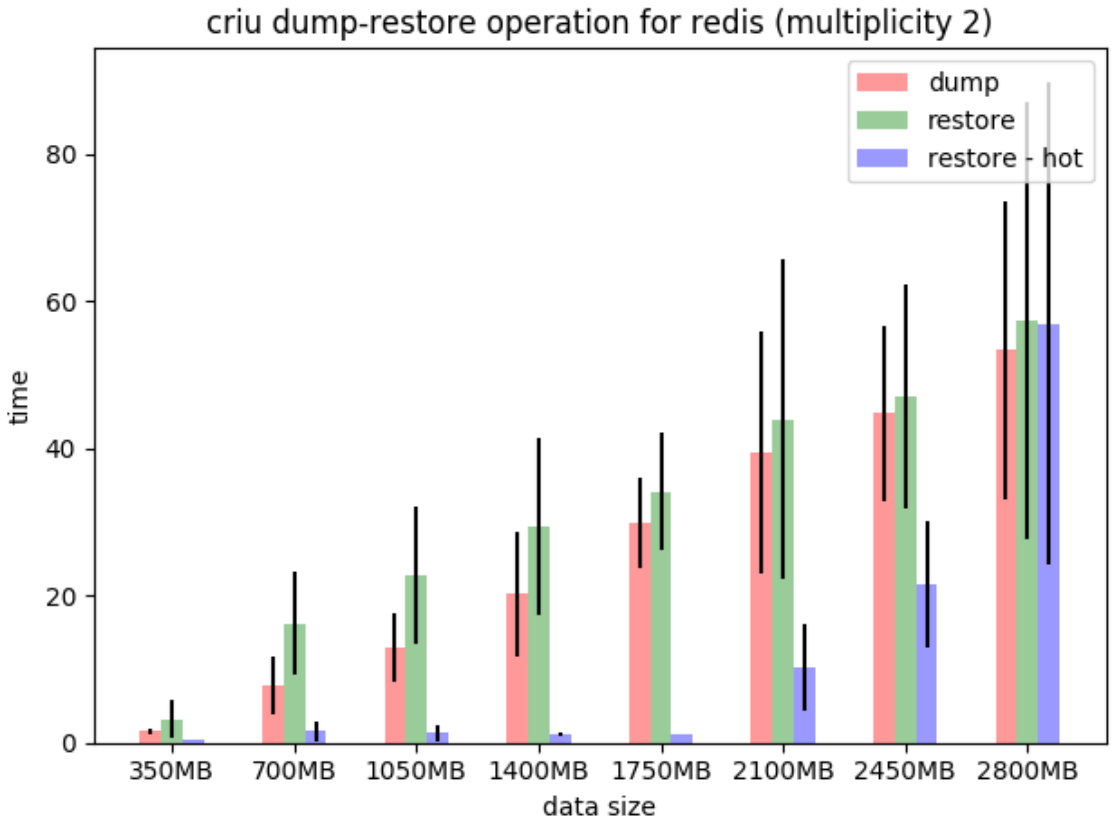
Τέλος, συγκρίνοντας τη συμπεριφορά του redis και του memcached, θα μπορούσαμε να πούμε ότι ενώ για μικρά μεγέθη παρατηρούνται παραπλήσιοι χρόνοι, στην περίπτωση του memcached χρειάζεται λιγότερος χρόνος για την ολοκλήρωση της αποθήκευσης και της επαναφοράς της εφαρμογής, της τάξεως των 5 με 10 δευτερολέπτων.

4.3 Οι διαδικασίες αποθήκευσης και επαναφοράς ενός module ως προς το χρόνο για διαφορετικές πολλαπλότητες

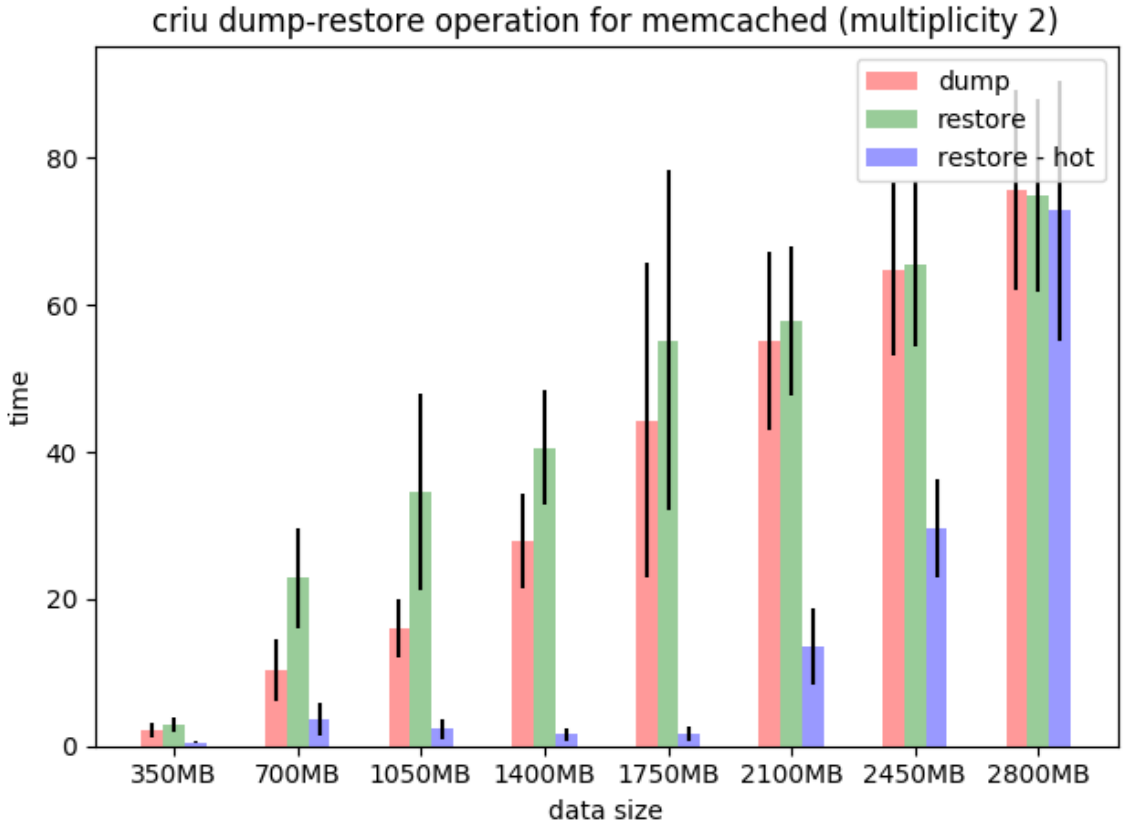
Για το επόμενο πείραμα, εκτελέσαμε το σενάριο εγκατάστασης του προηγούμενου πειράματος για διαφορετικές πολλαπλότητες του server. Όπως περιγράφηκε στο κεφάλαιο 3, στην περίπτωση module πολλαπλότητας μεγαλύτερης του ενός, η αποθήκευση και η επαναφορά εκτελούνται παράλληλα, ενώ η εγκατάσταση προχωράει μόνο όταν το κάθε module έχει ολοκληρώσει την αντίστοιχη διαδικασία. Στις παρακάτω γραφικές παραστάσεις απεικονίζονται οι χρόνοι που απαιτούνται για την εκτέλεση του παραπάνω σεναρίου εκτέλεσης για τις εφαρμογές redis και memcached, για πολλαπλότητα 2, 4, 6 του module server. Πιο συγκεκριμένα, από ένα σύνολο 15 εκτελέσεων του σεναρίου εγκατάστασης για κάθε εφαρμογή και για τις πολλαπλότητες που αναφέραμε, παραθέτουμε στο πρώτο διάγραμμα τη μέση τιμή του μέγιστου χρόνου που διήρκεσε η κάθε διαδικασία ως ακραία περίπτωση και το μέσο χρόνο που απαιτείται για την ολοκλήρωση της διαδικασίας της αποθήκευσης και της επαναφοράς των εφαρμογών redis και memcached.



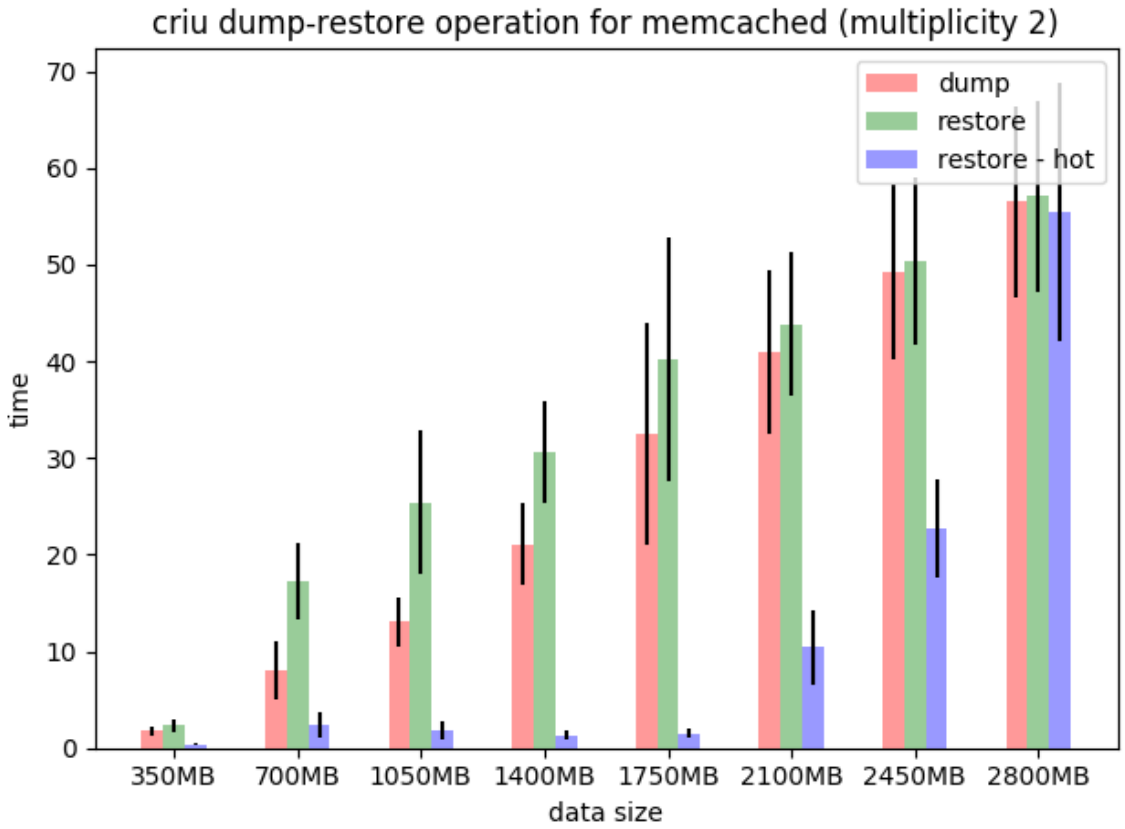
Εικόνα 4-4: Redis server multiplicity 2, max values



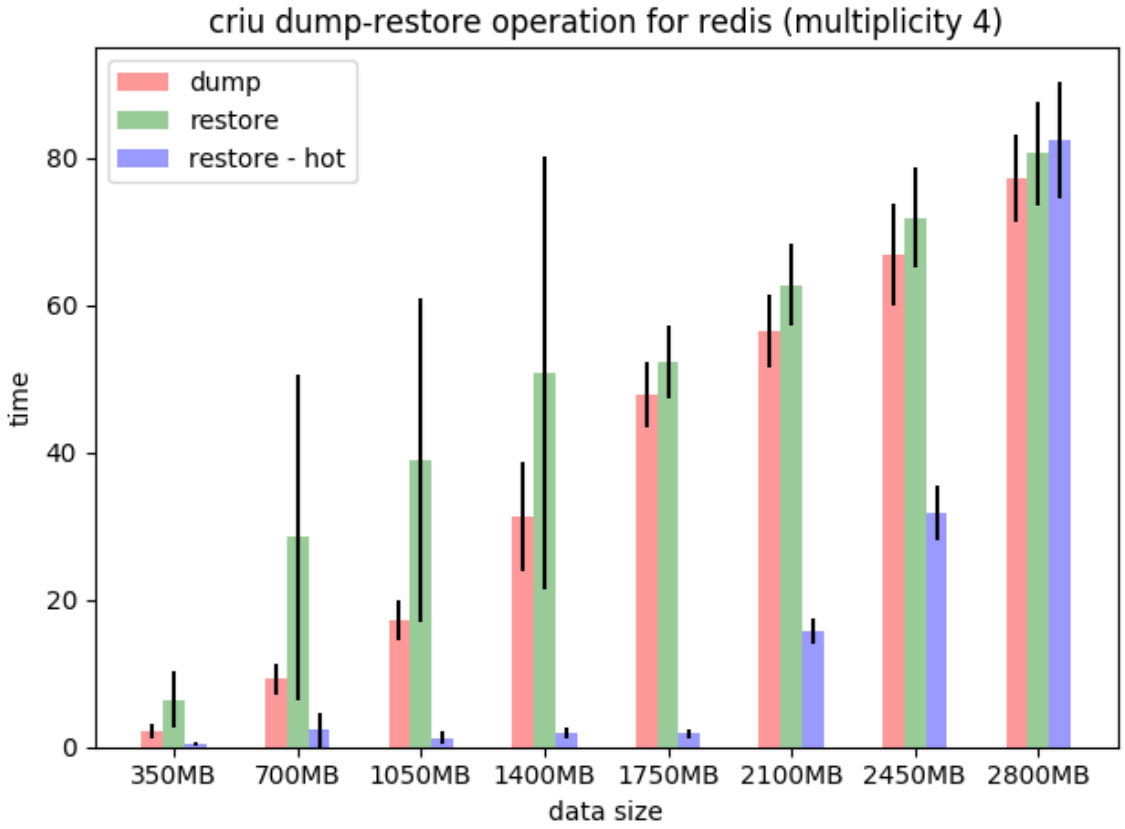
Εικόνα 4-5: Redis server multiplicity 2, average values



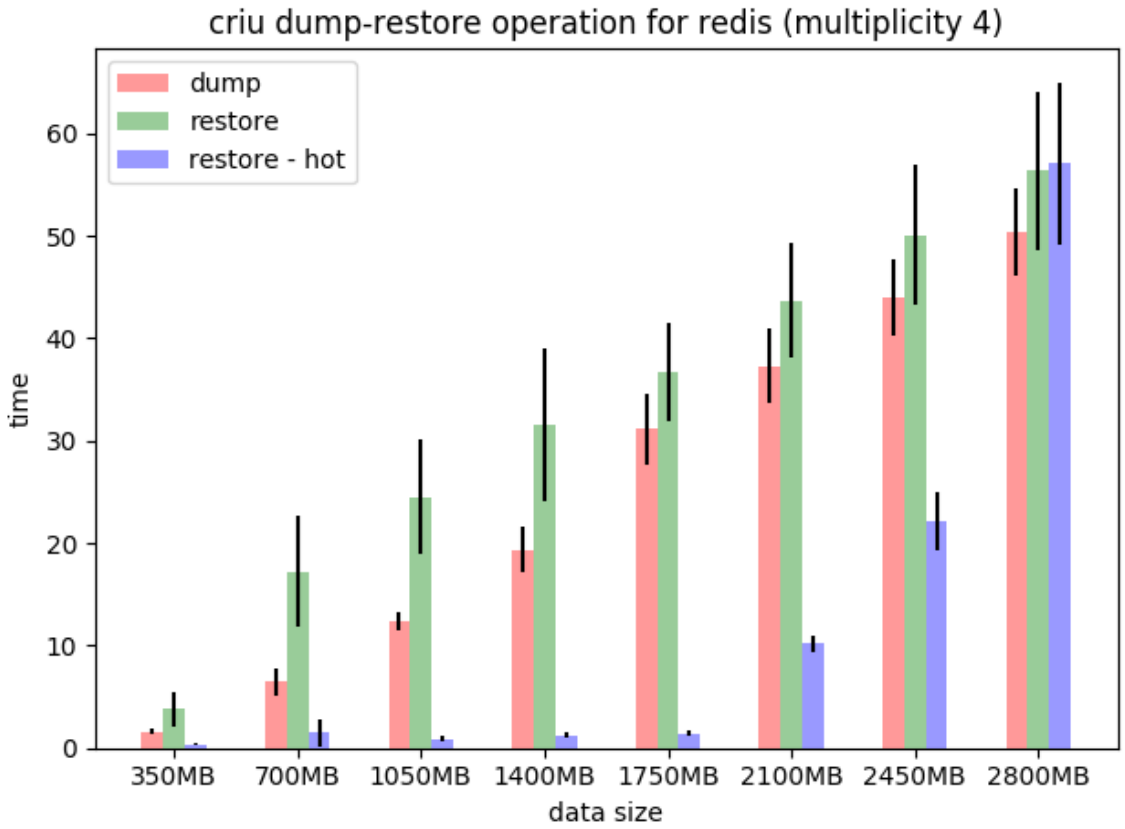
Εικόνα 4-6: Memcached server multiplicity 2, max values



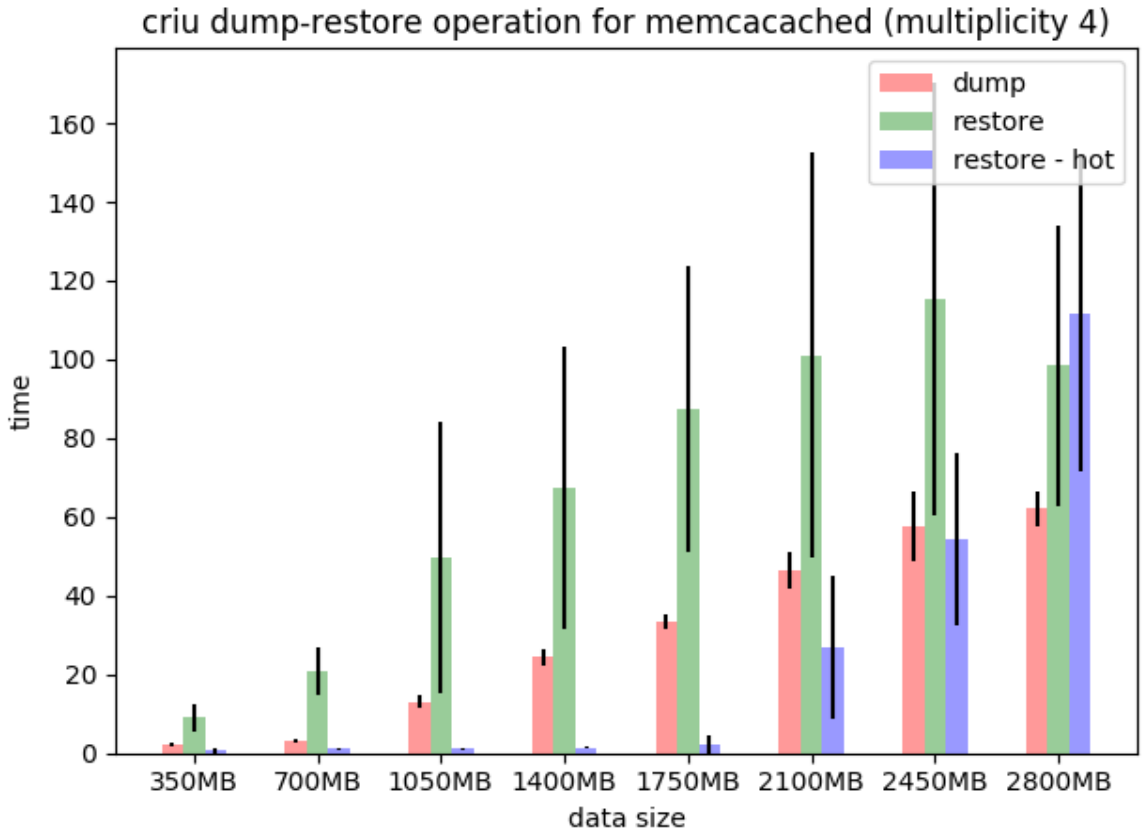
Εικόνα 4-7: Memcached server multiplicity 2, average values



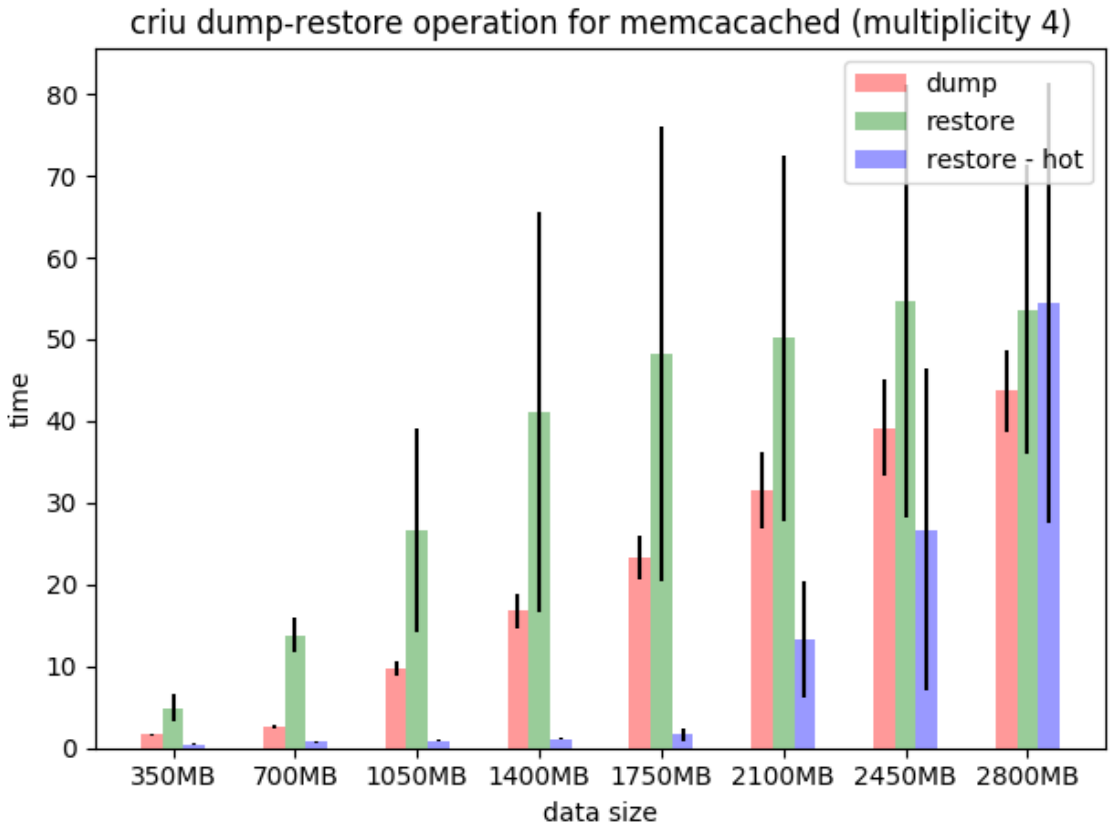
Εικόνα 4-8: Redis server multiplicity 4, max values



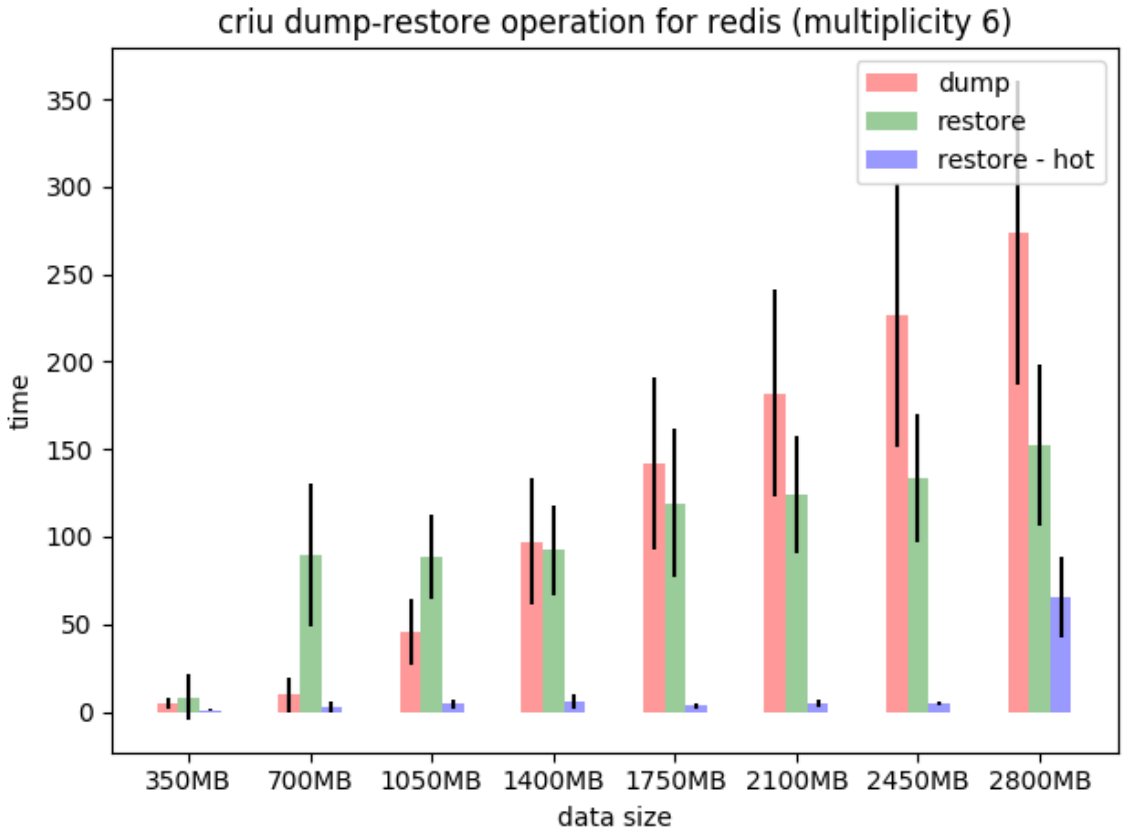
Εικόνα 4-9: Redis server multiplicity 4, average values



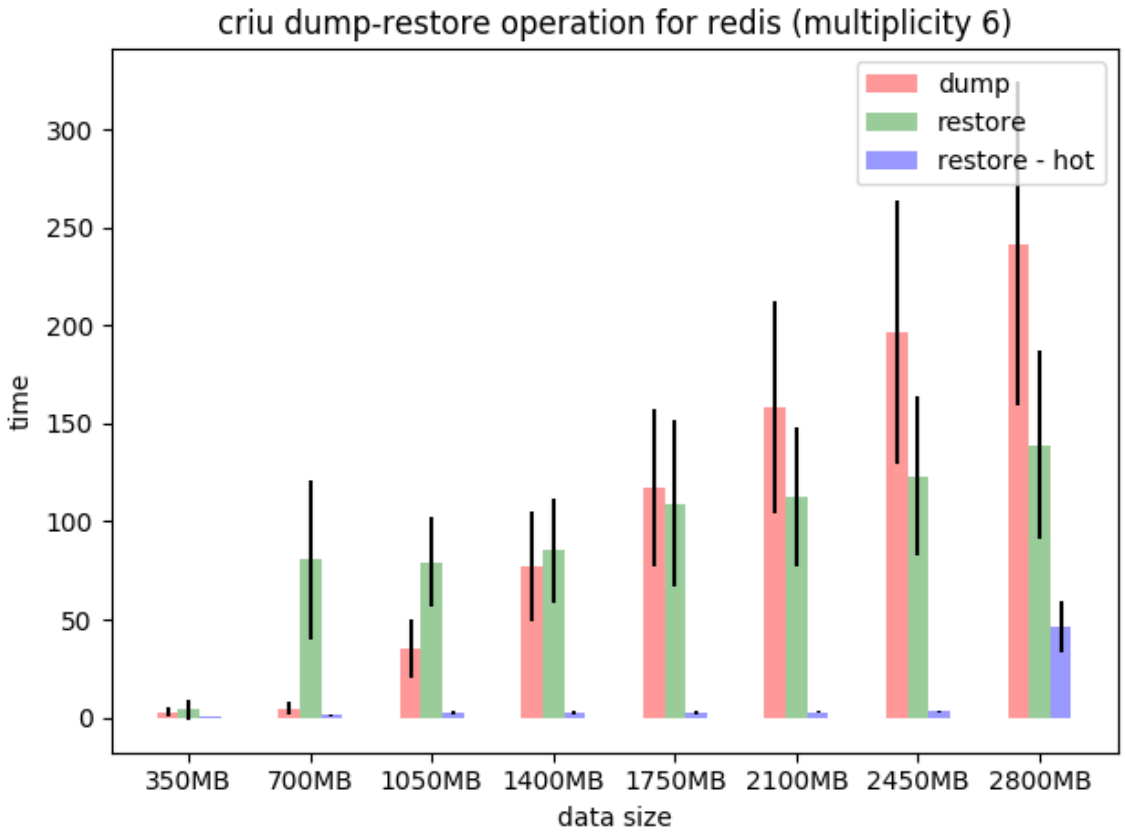
Εικόνα 4-10: Memcached server multiplicity 4, max values



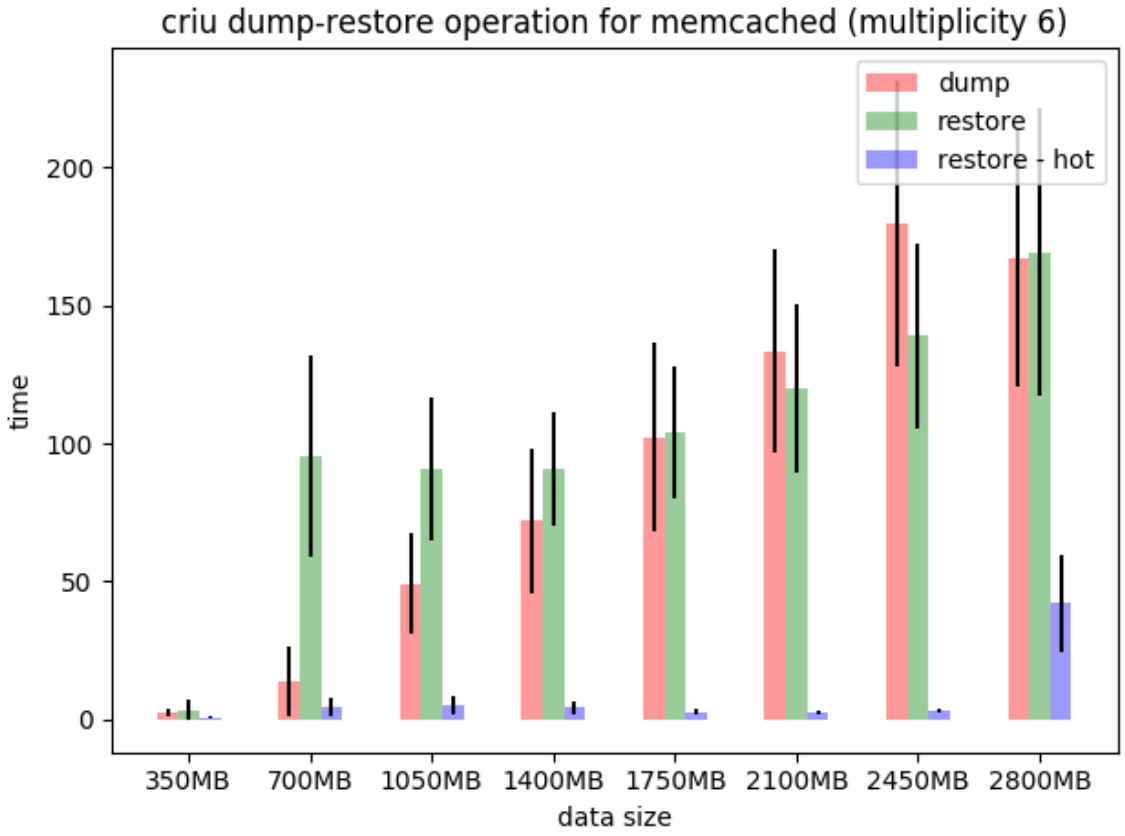
Εικόνα 4-11: Memcached server multiplicity 4, average values



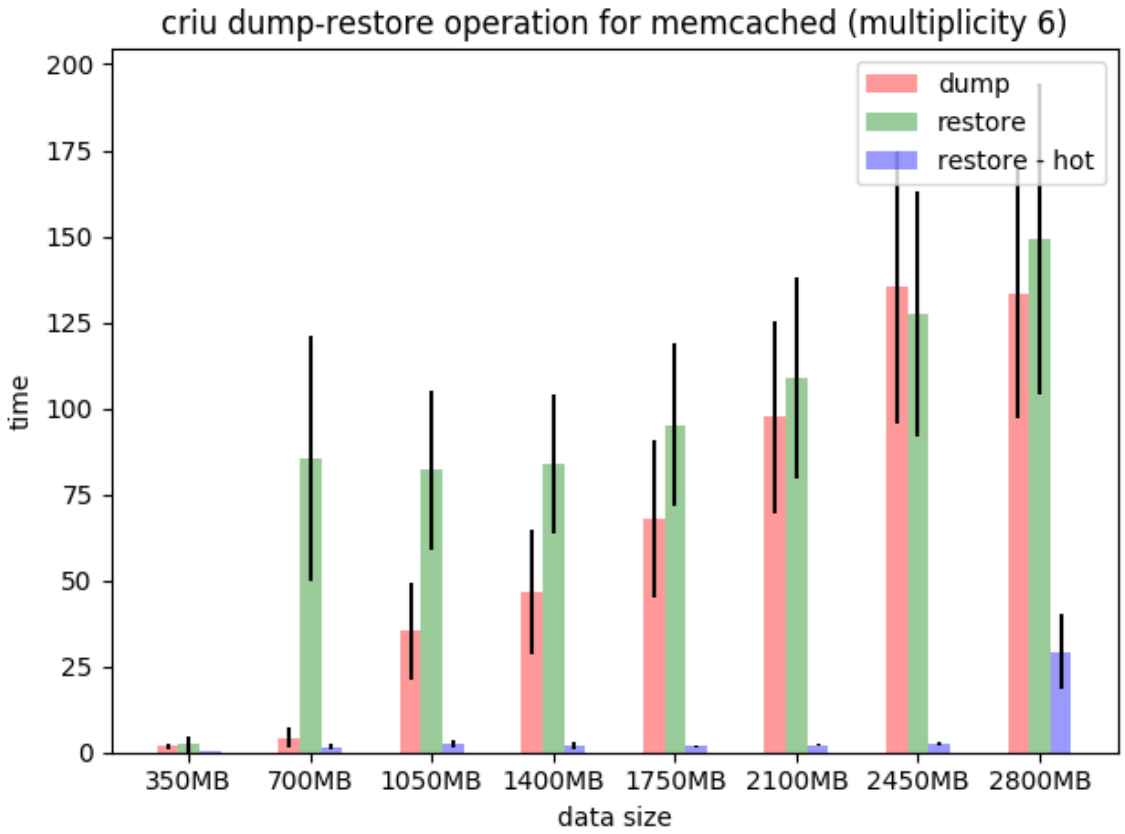
Εικόνα 4-12: Redis server multiplicity 6, max values



Εικόνα 4-13: Redis server multiplicity 6, average values



Εικόνα 4-14: Memcached server multiplicity 6, max values



Εικόνα 4-15: Memcached server multiplicity 6, average values

Όπως και για το προηγούμενο πείραμα, έτσι και εδώ ο κατακόρυφος άξονας αντιπροσωπεύει το χρόνο που απαιτήθηκε για την ολοκλήρωση των διαδικασιών αποθήκευσης, επαναφοράς και επαναφοράς με ζεστή μνήμη, για τα μεγέθη δεδομένων εφαρμογής που αναγράφονται στον οριζόντιο άξονα.

Με μια πρώτη ματιά, είναι εμφανές ότι αύξηση της πολλαπλότητας του module την κατάσταση του οποίου αποθηκεύουμε και επαναφέρουμε στην περίπτωση σφάλματος προκαλεί αύξηση του χρόνου που απαιτείται, παρ' όλο που όπως έχει περιγραφεί στο κεφάλαιο 3 οι διαδικασίες αυτές πραγματοποιούνται παράλληλα. Η βασική αιτία αυτής της συμπεριφοράς είναι οι περιορισμοί του υλικού στο οποίο λειτουργούν τα εικονικά μηχανήματα.

Μολονότι τα εικονικά μηχανήματα που δημιουργούμε και εγκαθιστούμε τις εφαρμογές συμπεριφέρονται σαν αυτόνομες μονάδες, στη πραγματικότητα μοιράζονται πόρους του ίδιου ξενιστή (host) ή συμπλέγματος (cluster), γεγονός που έχει επιπτώσεις στην απόδοση. Στη περίπτωση μας, δύο ή περισσότερα εικονικά μηχανήματα επιχειρούν να γράψουν παράλληλα δεδομένα στο δίσκο για την αποθήκευση της κατάστασης της εφαρμογής και αντίστοιχα να διαβάσουν δεδομένα από αυτόν για την επαναφορά της. Αν και αυτές οι ενέργειες (I/O operations) είναι φαινομενικά ανεξάρτητες, στη πραγματικότητα σειριοποιούνται στην υποκείμενη μονάδα αποθήκευσης, προκαλώντας έτσι καθυστέρηση. Μάλιστα, για την περίπτωση πολλαπλότητας 6, η

καθυστέρηση ήταν απαγορευτική για τη διεξαγωγή του πειράματος, με αποτέλεσμα να χρειαστεί να εκτελεστεί το πείραμα σε σύμπλεγμα εξοπλισμένο με καλύτερες μονάδες αποθήκευσης το οποίο ήταν σε θέση να διαχειριστεί τη παράλληλη εκτέλεση I/O operations μεγάλου όγκου δεδομένων από 6 διαφορετικά modules. Ακόμα και σε αυτή την περίπτωση παρατηρείται ότι ο χρόνος που απαιτήθηκε για την ολοκλήρωση των διαδικασιών αποθήκευσης και επαναφοράς για όλα τα μεγέθη δεδομένων, στο σενάριο πολλαπλότητας 6 είναι πολύ μεγαλύτερος από τον αντίστοιχο για μικρότερες πολλαπλότητες.

Όσον αφορά την ιδιαίτερα μεγάλη τυπική απόκλιση των μετρήσεων σε ορισμένα διαγράμματα, δηλώνει ότι σε κάποιοι χρόνοι είναι πολύ μεγαλύτεροι από τους υπόλοιπους. Αυτό μπορεί να οφείλεται στο πόσα εικονικά μηχανήματα εκτελούνται εκείνη τη χρονική στιγμή στο cluster προκαλώντας συμφόρηση στους διαύλους δεδομένων του (I/O bus) και σε άλλα απρόβλεπτα φαινόμενα, όπως διακοπές ρεύματος, καταστροφή ενός δίσκου κλπ.

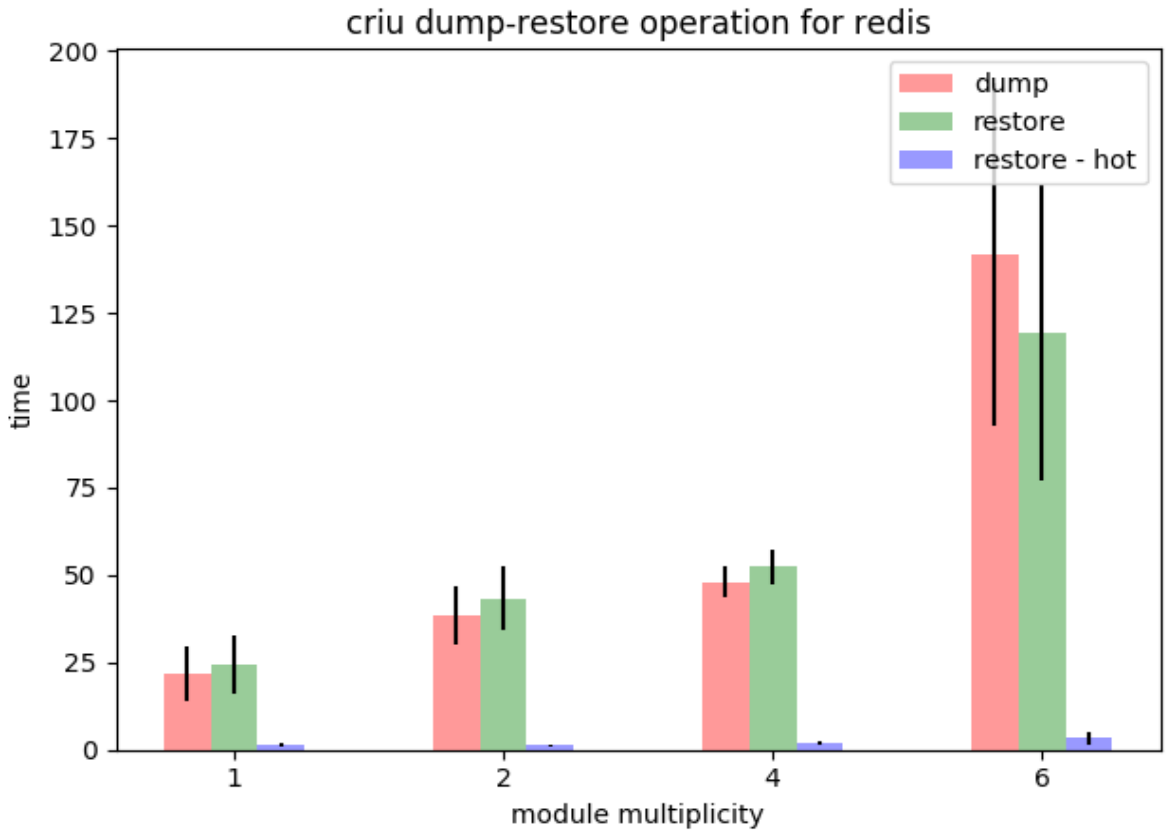
Αν και οι εφαρμογές πολλαπλότητας 2 και 4 παρουσιάζουν παρόμοια συμπεριφορά με αυτή που περιγράφηκε στο προηγούμενο πείραμα για πολλαπλότητα 1, ιδιαίτερο ενδιαφέρον παρουσιάζουν οι διαφορές που παρατηρούνται για τα πειράματα που εκτελέστηκαν στο βελτιωμένο cluster. Αρχικά παρατηρούμε ότι ενώ στα προηγούμενα διαγράμματα, η διαδικασία της επαναφοράς είναι πιο χρονοβόρα από αυτή της αποθήκευσης, στην περίπτωση πολλαπλότητας module ίση με 6 και του νέου cluster αυτό δεν ισχύει. Ο λόγος είναι το διαφορετικό υλικό αποθήκευσης και το λογισμικό το οποίο το ελέγχει. Όπως αναλύθηκε στις παρατηρήσεις του πρώτου πειράματος ο χρόνος ολοκλήρωσης αναγνώσεων και εγγραφών στη μνήμη εξαρτάται σε σημαντικό βαθμό από το λογισμικό που διαχειρίζεται το I/O subsystem και το μέγεθος της write cache.

Μια ακόμα εμφανής διαφορά είναι ο χρόνος που απαιτείται για την επαναφορά με “ζεστή” μνήμη που δεν ακολουθεί το μοτίβο των προηγούμενων περιπτώσεων, δηλαδή πολύ γρήγορη επαναφορά για μεγέθη δεδομένων μέχρι και 1,75 GB, λίγο πιο αργή για 2,1 και 2,45 GB και περίπου ίση χρονικά με την αποθήκευση και την επαναφορά με καθαρή μνήμη για 2,8 GB. Στη περίπτωση του νέου cluster ο χρόνος που απαιτείται για επαναφορά μέχρι και για μέγεθος μνήμης μέχρι και ~2,45GB είναι ελάχιστος ενώ παρατηρείται μια μικρή αύξηση στο επόμενο βήμα. Ο λόγος είναι ότι τα εικονικά μηχανήματα στα οποία εκτελέστηκε το module server ήταν εξοπλισμένα με 8GB RAM, σε αντίθεση με αυτά του πρώτου cluster που διέθεταν 6GB RAM. Μεγαλύτερη χωρητικότητα της μνήμης, σημαίνει ότι μεγάλο μέρος της μένει αχρησιμοποίητο και χρησιμοποιείται από το λειτουργικό σύστημα για να αποθηκεύει δεδομένα που χρησιμοποιήθηκαν πρόσφατα και ενδέχεται να ξαναζητηθούν (caching). Έτσι όταν κατά τη διαδικασία της επαναφοράς, διαβάζονται τα αρχεία που το CRIU έγραψε κατά τη διάρκεια της αποθήκευσης της κατάστασης της εφαρμογής στο δίσκο, μεγάλο μέρος αυτών βρίσκεται ήδη στη μνήμη.

Από τις μετρήσεις που πραγματοποιήσαμε για τα προηγούμενα πειράματα παραθέτουμε ενδεικτικά τα παρακάτω διαγράμματα που απεικονίζουν το χρόνο που απαιτήθηκε για τις διαδικασίες αποθήκευσης, επαναφοράς και επαναφοράς με "ζεστή" μνήμη του module server για μέγεθος δεδομένων εφαρμογής 1,75 GB. Το διάγραμμα 4-16 αφορά την εφαρμογή redis και το 4-17 το memcached, με τον οριζόντιο άξονα να αντιπροσωπεύει την πολλαπλότητα του module στο οποίο εκτελείται το CRIU και τον κατακόρυφο να απεικονίζει το χρόνο που απαιτήθηκε. Παρατηρείται ότι για το συγκεκριμένο μέγεθος δεδομένων, η επαναφορά με "ζεστή" μνήμη πραγματοποιείται ταχύτατα για όλες τις πολλαπλότητες μιας και τα δεδομένα των εφαρμογών υπάρχουν ήδη στη μνήμη.

Ακόμα, στη περίπτωση του redis, η αποθήκευση και η επαναφορά πραγματοποιούνται αισθητά πιο γρήγορα για πολλαπλότητες 1,2 και 4 σε σχέση με την περίπτωση πολλαπλότητας 6 όπου απαιτείται σχεδόν ο διπλάσιος χρόνος. Τέλος για τα πειράματα που πραγματοποιήθηκαν στο νέο cluster, η διαδικασία της αποθήκευσης είναι πιο χρονοβόρα από αυτή της επαναφοράς, για λόγους που εξηγήθηκαν παραπάνω.

Στη περίπτωση του memcached, οι χρόνοι που παρατηρούνται συγκριτικά με αυτούς του redis είναι καλύτεροι, εκτός από αυτόν της επαναφοράς για πολλαπλότητα 4.



Εικόνα 4-16: Redis server module multiplicity vs time for 1,75GB of data

Εικόνα 4-17: Memcached server module multiplicity vs time for 1,75GB of data

4.4 Εκτέλεση της διαδικασίας εγκατάστασης με μεταβλητή πιθανότητα σφάλματος

Στόχος του επόμενου πειράματος είναι η μελέτη της συμπεριφοράς του συστήματος που προτείνουμε για σενάρια εγκατάστασης αποτελούμενα από script με τυχαία πιθανότητα σφάλματος. Συγκεκριμένα, μας ενδιαφέρει να δείξουμε ότι:

- Η διαδικασία εγκατάστασης τερματίζει επιτυχώς ανεξάρτητα της πιθανότητας σφάλματος ενός script.
- Η διαδικασία εγκατάστασης ολοκληρώνεται σε ικανοποιητικό χρόνο.
- Στο τέλος της εγκατάστασης η εφαρμογή έχει τα δεδομένα και τη συμπεριφορά που θα είχε σε περίπτωση που δε συνέβαινε κανένα σφάλμα.

Για το σκοπό αυτό εκτελούμε το σενάριο εγκατάστασης της εικόνας 4-18. Η εφαρμογή μας αποτελείται από έναν server και δύο client που εκτελούν τα ίδια script, εισάγοντας παράλληλα δεδομένα στο server. Κατά την εκτέλεση του, οποιοδήποτε εκ των script ενδέχεται να αποτύχει με κάποια πιθανότητα που ορίζεται για τις ανάγκες του πειράματος, προκαλώντας την αφύπνιση του μηχανισμού επαναφοράς του module server σε προγενέστερη αποθηκευμένη κατάσταση. Σε αντίθεση με τα παραπάνω πειράματα, όπου η αποτυχία ενός script ήταν προγραμματισμένη να συμβαίνει ελεγχόμενα, εδώ το κάθε script ενός client αποτυγχάνει τυχαία.

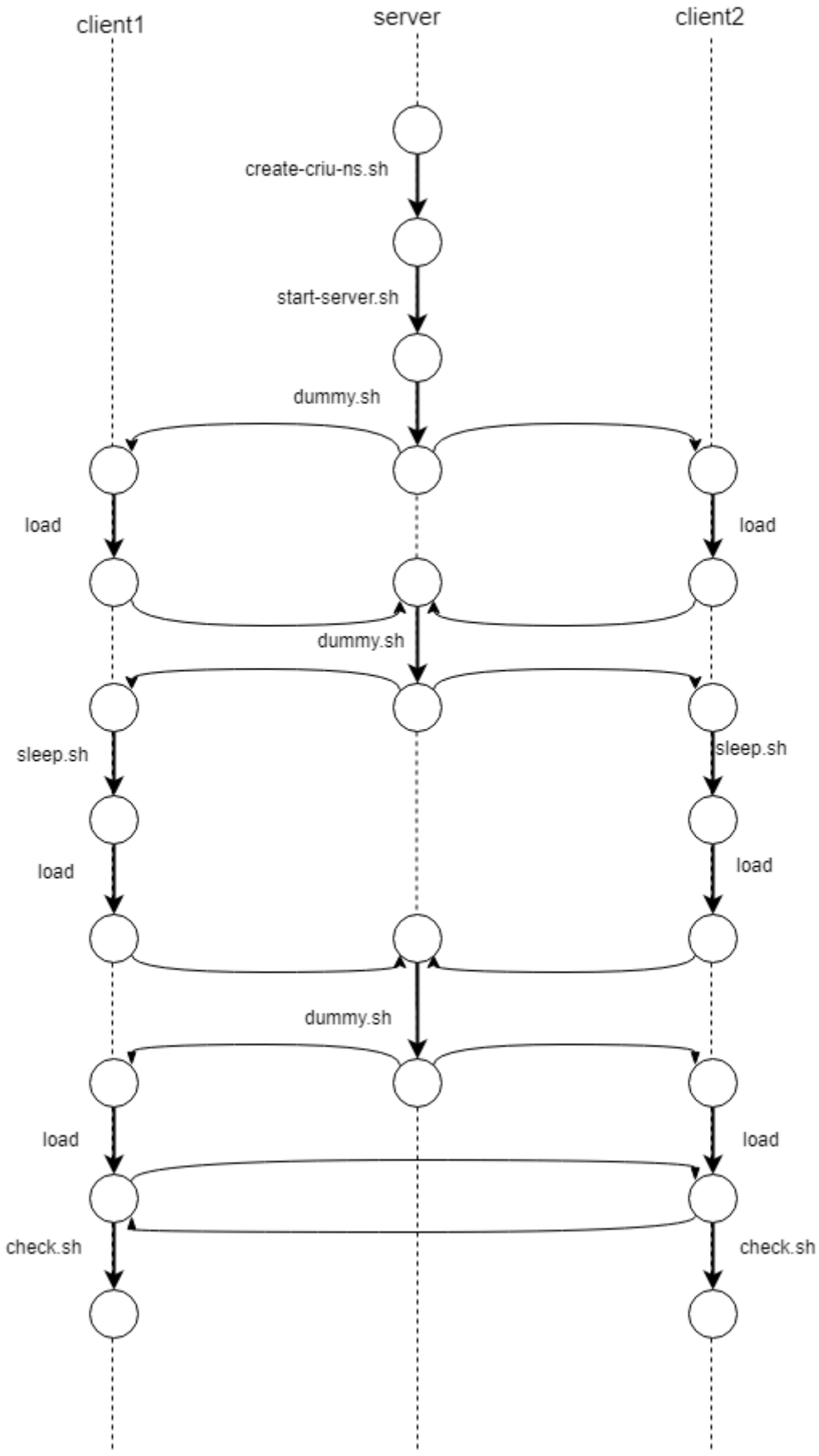
Αρχικά εκκινείται η διεργασία του server και παράγεται ένα στιγμιότυπο, ώστε να υπάρχει ένα αρχικό checkpoint, στο οποίο η εγκατάσταση θα μπορεί να επιστρέψει σε περίπτωση που αποτύχει κάποιο εκ των δύο script που εκτελούνται στον επόμενο βρόγχο. Εφόσον οι δύο client εκτελέσουν επιτυχημένα το script load, η εγκατάσταση προχωρά παρακάτω όπου παράγεται ένα νέο checkpoint της εγκατάστασης. Σε αντίθετη περίπτωση, πραγματοποιείται επαναφορά στο στιγμιότυπο που έχουμε αποθηκεύσει και η εκτέλεση των script load επαναδρομολογείται, ανεξαρτήτως αν κάποιο εκ των δύο script είχε εκτελεστεί με επιτυχία.

Ο δεύτερος βρόγχος αν και πιο σύνθετος ακολουθεί την ίδια λογική. Κάθε client εκτελεί το script sleep.sh και στη συνέχεια το load, εισάγοντας νέες τιμές στο server. Συνέχιση της διαδικασίας εγκατάστασης και τη παραγωγή νέου στιγμιότυπου του server προϋποθέτει επιτυχία και των τεσσάρων script. Μετά τη δημιουργία του τελευταίου στιγμιότυπου (dummy.sh) οι δύο client επιχειρούν να εισάγουν τιμές για τελευταία φορά. Όπως και παραπάνω, αποτυχία προκαλεί επαναφορά της εφαρμογής του server ενώ μετά την επιτυχή εκτέλεση το script check.sh ελέγχει την εγκυρότητα των δεδομένων και ότι μετά το πέρας της εκτέλεσης ο server περιέχει τις σωστές τιμές.

Το σενάριο εγκατάστασης που περιγράφτηκε εκτελείται για τις εξής πιθανότητες αποτυχίας:

Πιθανότητα αποτυχίας script ακραίων βρόγχων	**Πιθανότητα αποτυχίας script ενδιάμεσου βρόγχου**
0,2	0,105
0,3	0,163
0,4	0,225
0,5	0,292
0,6	0,367

Η παραπάνω διαφοροποίηση συμβαίνει ώστε να επιτύχουμε ίδια πιθανότητα επιτυχίας για κάθε βρόγχο.

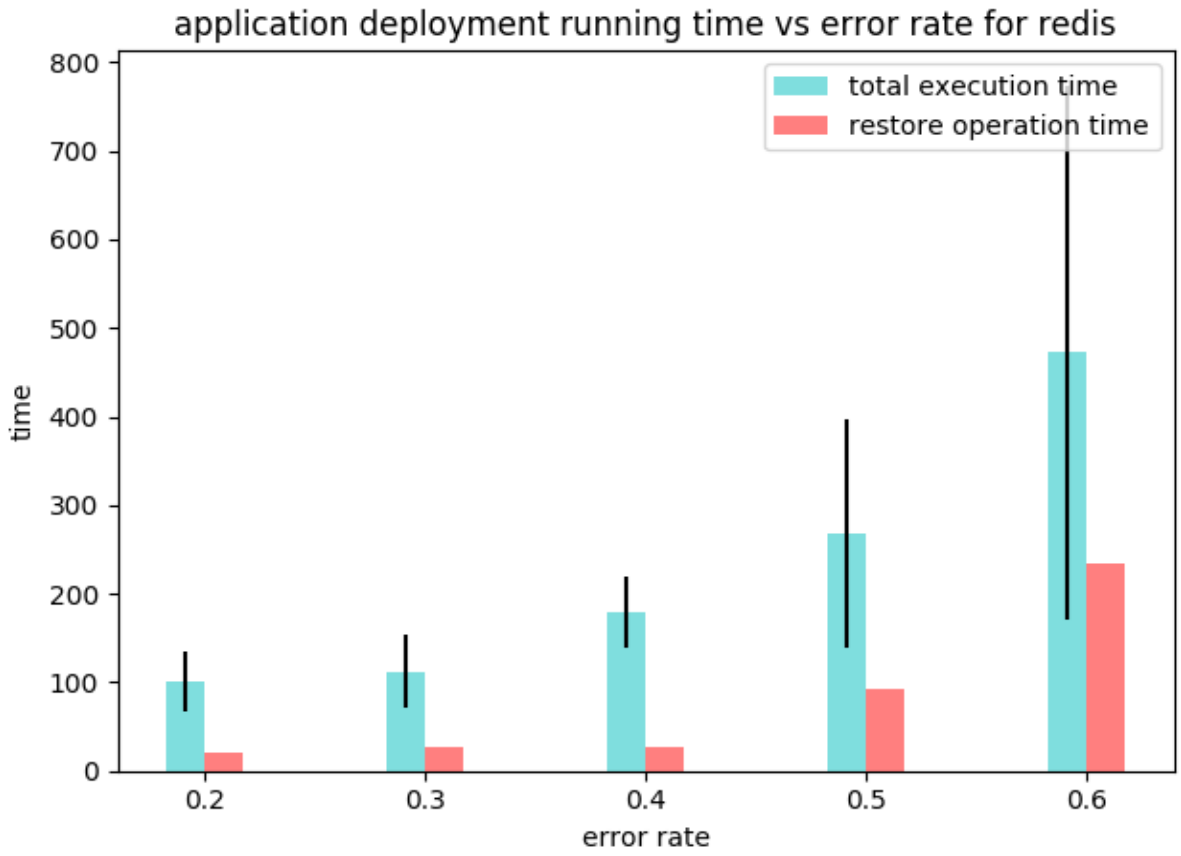


Εικόνα 4-18: Ο γράφος εκτέλεσης της εγκατάστασης του πειράματος

Στα διαγράμματα 4-19, 4-20 απεικονίζεται ο χρόνος εκτέλεσης του σεναρίου εγκατάστασης που παρουσιάστηκε παραπάνω για διαφορετικές πιθανότητες αποτυχίας των script του module client. Όπως είναι αναμενόμενο ο χρόνος εκτέλεσης είναι ανάλογος της πιθανότητας αποτυχίας, καθώς αυξάνονται οι επαναφορές που εκτελούνται. Η επιβάρυνση που εισάγουν οι διαδικασίες αποθήκευσης και επαναφοράς στο συνολικό χρόνο εκτέλεσης δίνονται από το παρακάτω τύπο:

$$(freeze\ time\ +\ restore\ time)/total\ execution\ time$$

Script Error Rate	**Restore overhead percentage**
0.2	0.604
0.3	0.647
0.4	0.664
0.5	0.714
0.6	0.777



Εικόνα 4-19: Redis running time vs error rate

Εικόνα 4-20: Memcached running time vs error rate

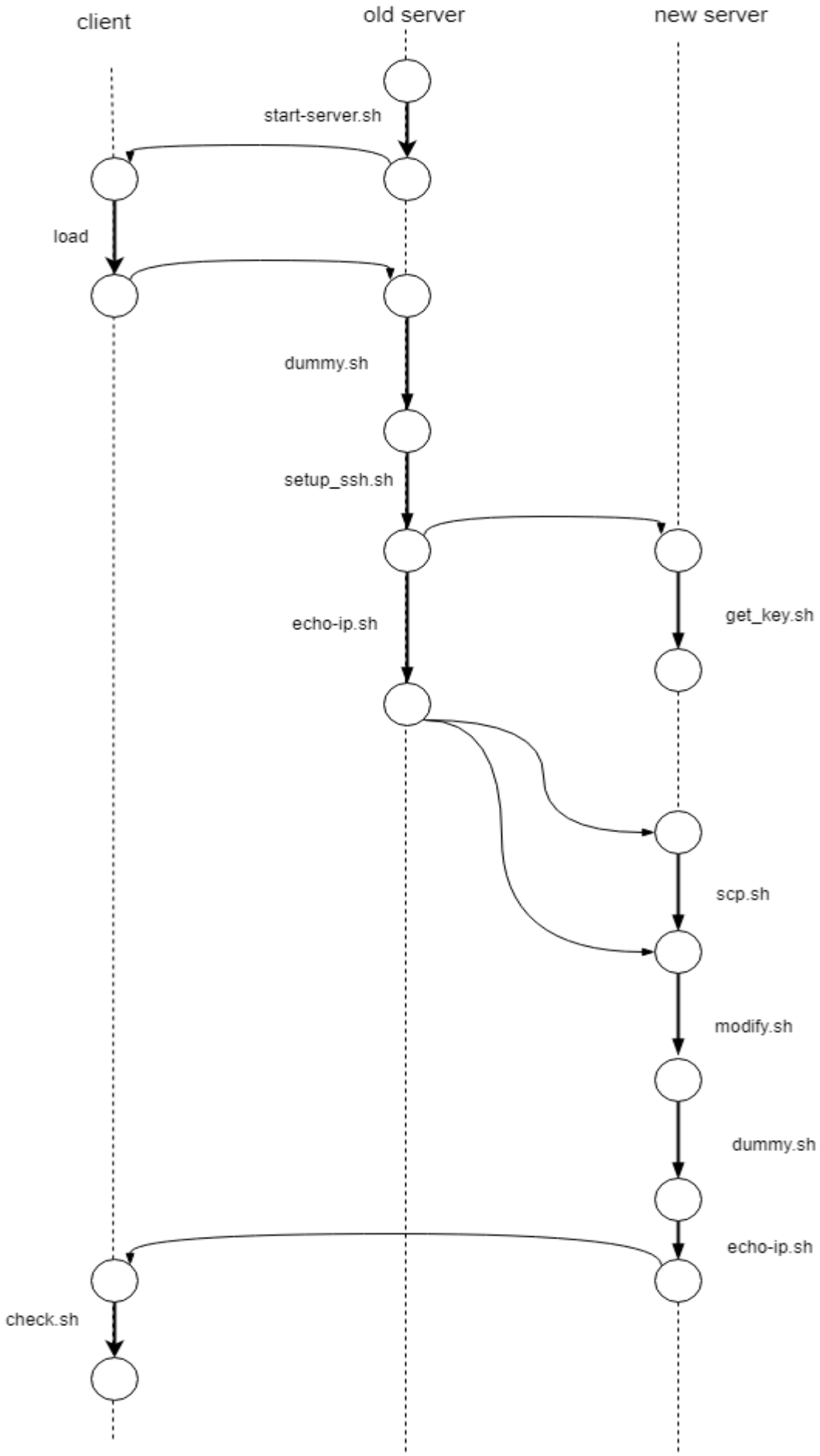
4.5 Ζωντανή μεταφορά εφαρμογής σε νέο εικονικό μηχάνημα μεγαλύτερων δυνατοτήτων

Σε αυτή την ενότητα παρουσιάζονται τα αποτελέσματα του πειράματος που πραγματοποιήσαμε για το σενάριο χρήσης που περιγράφηκε στην ενότητα 3.6. Πραγματοποιήθηκε μεταφορά της εφαρμογής redis server από εικονικό μηχάνημα χωρητικότητας μνήμης 4GB στο οποίο εκτελούνταν αρχικά σε νέο μηχάνημα μνήμης 8GB. Ο γράφος της διαδικασίας εγκατάστασης φαίνεται στην εικόνα 4-21.

Αρχικά, εκκινείται η διεργασία του redis server στο module old server και ο client εισάγει δεδομένα μέσω του script load. Μετά την ολοκλήρωση της εισαγωγής των δεδομένων, η κατάσταση της εφαρμογής αποθηκεύεται στο δίσκο (script dummy.sh) σε αρχεία που πρόκειται να μεταφερθούν στο module new server. Στη συνέχεια, πραγματοποιείται η παραγωγή ενός ζεύγους δημοσίου κλειδιού το οποίο μετά τις κατάλληλες ρυθμίσεις αποστέλλεται στο καινούργιο module, για την επίτευξη συνδεσιμότητας μεταξύ των δύο μηχανημάτων και την μεταφορά των αρχείων που δημιούργησε το CRIU. Η τελευταία ενέργεια του old server είναι να γνωστοποιήσει την IP διεύθυνση του στο νέο module.

Αφού πραγματοποιηθούν οι κατάλληλες ρυθμίσεις στο νέο module (get_key.sh), τα αρχεία που παράχθηκαν κατά την αποθήκευση της κατάστασης της διεργασίας αντιγράφονται στο νέο εικονικό μηχάνημα, μέσω της εντολής scp (secure copy) και ενημερώνονται όπου είναι απαραίτητο με την διεύθυνση IP του νέου module (modify.sh). Τέλος, ο new server επαναφέρει τη διεργασία μέσω της εκτέλεσης ενός dummy script, που φέρει την ετικέτα “criu” στο αρχείο περιγραφής της εφαρμογής, όπως περιγράφηκε εκτενώς στο κεφάλαιο 3. Αξίζει να σημειωθεί ότι ενώ κανονικά δρομολόγηση ενώ τέτοιου script προκαλεί αρχικά την αποθήκευση της διεργασίας και στη συνέχεια την επαναφορά της, στη περίπτωση μας η διαδικασία αποθήκευσης παραλείπεται καθώς δεν υπάρχει ενεργή εφαρμογή και πραγματοποιείται κατευθείαν η διαδικασία επαναφοράς.

Το μόνο που απομένει είναι το module new server να ενημερώσει το module client με την νέα IP διεύθυνση στην οποία μπορεί να επικοινωνήσει με την εφαρμογή του server. Ο client επιβεβαιώνει ότι υπάρχει συνδεσιμότητα και ότι τα δεδομένα που υπήρχαν στον server πριν τη μεταφορά έχουν διατηρηθεί (check.sh) και πλέον η διεργασία του server που εκτελείται στο αρχικό module μπορεί να τερματιστεί.



Εικόνα 4-21: Ο γράφος εγκατάστασης του πειράματος

5 Συμπεράσματα

Στις μέρες μας, ολοένα περισσότερες επιχειρήσεις και προγραμματιστές στρέφονται στο IaaS (infrastracture as a service) που προσφέρουν πάροχοι cloud υπηρεσιών, ως έναν γρήγορο και οικονομικό τρόπο ανάπτυξης των εφαρμογών τους. Η πολυπλοκότητα των σημερινών διαδικτυακών εφαρμογών πολυπλέκει σε μεγάλο βαθμό τη διαδικασία εγκατάστασης μιας εφαρμογής στο cloud καθιστώντας την αυτοματοποίηση της πολύ σημαντική.

Σε αυτή τη διπλωματική εργασία προτείνουμε την επέκταση της Αύρας, ώστε να υποστηρίζει την επανεκτέλεση αρχείων εκτέλεσης που επηρεάζουν τη κατάσταση της μνήμης μια εφαρμογής. Η Αύρα είναι ένα σύστημα αυτοματοποίησης της διαδικασίας εγκατάστασης μιας εφαρμογής στο cloud, με δυνατότητα ανάνηψης από παροδικά σφάλματα που οφείλονται στην ασταθή φύση του. Χρησιμοποιώντας το CRIU, ένα πρόγραμμα του λειτουργικού συστήματος Linux που επιτρέπει την αποθήκευση της κατάστασης μιας διεργασίας και την επαναφορά της, δημιουργούμε ένα στιγμιότυπο της κατάστασης του γράφου εγκατάστασης της εφαρμογής πριν την εκτέλεση script που μεταβάλλουν τη κατάσταση της μνήμης της εφαρμογής και κατ' επέκταση όλης της εγκατάστασης.

Για τη δημιουργία ενός στιγμιότυπου και τη δυνατότητα επαναφοράς της διαδικασίας εγκατάστασης σε αυτό, αποθηκεύουμε τη κατάσταση της μνήμης της εφαρμογής στο δίσκο, ενώ παράγουμε και ένα αντίγραφο της ουράς ανταλλαγής μηνυμάτων που χρησιμοποιείται για την επικοινωνία και το συγχρονισμό μεταξύ των διαφορετικών module της Αύρας. Εσφαλμένη εκτέλεση ενός script πυροδοτεί το μηχανισμό επαναφοράς, ο οποίος αναλαμβάνει να επαναφέρει την εφαρμογή στην κατάσταση που έχουμε αποθηκεύσει επανεκκινώντας την εγκατάσταση από εκείνο το σημείο και δρομολογώντας το αποτυχημένο script ξανά για εκτέλεση.

Για την αξιολόγηση του μηχανισμού που προτείνουμε εκτελέσαμε τρία διαφορετικά σενάρια εγκατάστασης, χρησιμοποιώντας δύο δημοφιλείς εφαρμογές, το redis και το memcached, που βασίζονται σχεδόν ολοκληρωτικά στη μνήμη τους. Τα αποτελέσματα των πειραμάτων δείχνουν ότι η βελτιωμένη έκδοση της Αύρας ανταποκρίνεται με επιτυχία σε διαφορετικά σενάρια εγκατάστασης και για εφαρμογές μεγάλου όγκου δεδομένων και αποτελούμενες από πολλά διαφορετικά module ενώ καταφέρνει να εκτελέσει με επιτυχία σενάρια εγκατάστασης ακόμα και σε ασταθή περιβάλλοντα, όπου οι πιθανότητες αποτυχίας ενός script είναι αρκετά μεγάλες. Από τη διεξαγωγή των πειραμάτων, παρατηρήσαμε ότι ο μηχανισμός παραγωγής στιγμιότυπου και επαναφοράς σε αυτό σε περίπτωση σφάλματος εισάγει επιπλέον επιβάρυνση στο συνολικό χρόνο εκτέλεσης της εγκατάστασης, το μέγεθος της οποίας επηρεάζεται γραμμικά από τον αριθμό των σελίδων (memory pages) της εφαρμογής. Αξίζει να επισημανθεί ότι το σύστημα που προτείνουμε

επιτυγχάνει πάντα την ανάνηψη από ένα παροδικό σφάλμα και εγγυάται την επιτυχία της εγκατάστασης, σε αντίθεση με άλλα εργαλεία εγκατάστασης εφαρμογών στο cloud.

Τέλος, παρουσιάζουμε έναν εναλλακτικό τρόπο χρήσης του συστήματος, που ξεφεύγει από την έννοια της εγκατάστασης. Εκμεταλλευόμενοι το γεγονός ότι η Αύρα μπορεί να συνδεθεί σε ήδη ενεργά εικονικά μηχανήματα καθώς και τη δυνατότητα του CRIU να επαναφέρει μια εφαρμογή σε διαφορετικό περιβάλλον από αυτό στο οποίο δημιουργήθηκε, πραγματοποιούμε τη ζωντανή μεταφορά μιας εφαρμογής σε νέο περιβάλλον, με σκοπό την αύξηση των δυνατοτήτων της.

6 Βιβλιογραφία

[1] Openstack Heat. https://wiki.openstack.org/wiki/Heat

[2] Openstack. https://www.openstack.org/

[2] AWS CloudFormation. http://aws.amazon.com/cloudformation/

[3] JSON (JavaScript Object Notation). https://www.json.org/

[4] yaml. http://yaml.org/

[6] Vagrant. https://www.vagrantup.com/

[7] Juju. https://jujucharms.com/

[8] Αύρα (Aura). https://github.com/giagiannis/aura/

[9] CRIU. https://criu.org/Main_Page

[10] Hadoop. http://hadoop.apache.org/

[11] python novaclient. https://github.com/openstack/python-novaclient

[12] Paramiko. http://www.paramiko.org/

[13] Ptrace system call. http://man7.org/linux/man-pages/man2/ptrace.2.html

[14] Cgroup freezer.
https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt

[15] Google Protocol Buffers. https://developers.google.com/protocol-buffers/

[16] Redis. https://redis.io/

[17] Memcached. https://memcached.org/

[18] Linux Namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html

[19] criu-ns helper script.
https://github.com/checkpoint-restore/criu/blob/criu-dev/scripts/criu-ns

[20] CRIT. https://criu.org/CRIT