Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων & Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Σχεδιασμός, υλοποίηση και αξιολόγηση συστήματος δημιουργίας, μεταφοράς και απομακρυσμένης εκτέλεσης διεργασιών

Διπλωματικη Εργασια

του

Αρβανίτη Χρήστου

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Αθήνα, Φεβρουάριος 2020

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων & Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Σχεδιασμός, υλοποίηση και αξιολόγηση συστήματος δημιουργίας, μεταφοράς και απομακρυσμένης εκτέλεσης διεργασιών

Διπλωματικη Εργασια

του

Αρβανίτη Χρήστου

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την κάτωθι τριμελή επιτροπή την 19η Φεβρουαρίου 2020.

Γεώργιος Γκούμας Επίκουρος Καθηγητής ΕΜΠ	Διονύσιος Πνευματικάτος Καθηγητής ΕΜΠ	Νεκτάριος Κοζύρης Καθηγητής ΕΜΠ

Αρβανίτης Χρήστος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Περίληψη

Τα συστήματα παράλληλης και κατανεμημένης επεξεργασίας αναπτύσσονται ραγδαία, με ολοένα και αυξημένη λειτουργικότητα, προκειμένου να κατανέμουν βέλτιστα τον υπολογιστικό φόρτο στους διαθέσιμους πόρους, προσφέροντας κλιμακωσιμότητα, διαθεσιμότητα αλλά και αξιοπιστία. Παράλληλα, οι τεχνολογίες αποθήκευσης και επαναφοράς διεργασιών στο Linux, οι οποίες συγκεντρώνουν μεγάλο ενδιαφέρον από την κοινότητα, εισάγουν νέες προοπτικές για δυναμική μεταφορά διεργασιών σε κόμβους ενός υπολογιστικού συστήματος, ακόμη και αν αυτοί δεν παρουσιάζουν πλήρη ομοιογένεια. Στην παρούσα διπλωματική, σχεδιάζεται και υλοποιείται ένα σύστημα μεταφοράς και απομακρυσμένης εκτέλεσης διεργασιών σε χώρο χρήστη, το οποίο στηρίζεται στο εργαλείο CRIU και παρέχει τη διεπαφή της κλήσης συστήματος fork(), εστιάζοντας στη διαφάνεια. Επιπλέον, αναπτύσσονται επεκτάσεις για υποστήριξη των πλέον διαδεδομένων μηχανισμών διαδιεργασιακής επικοινωνίας. Τέλος, αξιολογείται η υλοποίηση του συστήματος, συγκρίνοντας τον χρόνο ολοκλήρωσης της μεταφοράς και απομακρυσμένης εκτέλεσης της διεργασίας με παραμέτρους το χρησιμοποιούμενο σύστημα αρχείων, τη θέση του κατανεμημένου συστήματος αρχείων καθώς και την τεχνολογία διασύνδεσης των κόμβων του συστήματος. Παρατηρούνται ικανοποιητικοί χρόνοι εκτέλεσης, δεδομένων των περιορισμών της υλοποίησης και των χρησιμοποιούμενων εργαλείων, ενώ οριοθετούνται τα επόμενα στάδια βελτιστοποίησης του συγκεκριμένου έργου.

Λέξεις-κλειδιά: remote fork, process migration, CRIU, checkpoint, restore, απομακρυσμένη εκτέλεση, network IPC

Abstract

Parallel and distributed computing systems are rapidly emerging, providing functionality for load balancing, offering scalability, availability and fault tolerance. Alongside those systems, process checkpoint and restore technologies implement mechanisms to freeze, save and restore processes even on heterogeneous hardware. Throughout this diploma thesis, a system of process migration and remote execution is designed and implemented, based on CRIU tool. This system uses fork system call's semantics, focusing on end user transparency. Additionally, transparent mechanisms that provide communication over network are implemented for most of widespread inter-process communication mechanisms. The implemented system's performance is evaluated on a number of different parameters, including the computing system node interconnection technology, the underlying file-system type used and the location of the distributed file-system server, with promising results considering the current implementation states and restrictions imposed by the elements of the system. Finally, based on the evaluation results, we are stating future goals and implementation requirements needed in order to optimize the resulting implementation of this diploma thesis.

Keywords: remote fork, process migration, CRIU, checkpoint, restore, remote execution, network IPC

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω το Στέφανο Γεράγγελο για το χρόνο που αφιέρωσε και την ευκαιρία που μου έδωσε να καταπιαστώ με το συγκεκριμένο θέμα καθώς και με πληθώρα ενδιαφέροντων θεμάτων, δίνοντάς μου τα κατάλληλα ερεθίσματα και την απαραίτητη στήριξη καθ' όλη τη διάρκεια εκπόνησης της παρούσας διπλωματικής εργασίας. Θα ήθελα ακόμη να ευχαριστήσω το διδακτορικό φοιτητή Χρήστο Κατσακιώρη για την πολύτιμη και άμεση βοήθεια στο στάδιο αξιολόγησης της υλοποίησης.

Δε θα μπορούσα να συνεχίσω χωρις να εκφράσω την ευγνωμοσύνη μου στην οικογένειά μου, σε εκείνους τους ανθρώπους που συνέχισαν να με στηρίζουν με αυταπάρνηση όλα αυτά τα χρόνια, παρέχοντάς μου τα απαραίτητα μέσα ώστε να πραγματοποιήσω τους στόχους μου. Καθένας τους ξεχωριστά, με ενθάρρυνε να τολμήσω, να μην αποδεχτώ τη μετριότητα ως επακόλουθο των ελαττωμάτων μου και να αντιμετωπίζω με ψυχραιμία την κάθε πρόκληση. Τέλος θα ήθελα να ευχαριστήσω τα πρόσωπα εκείνα που όχι μόνο με στήριξαν ψυχολογικά όλη αυτή τη περίοδο αλλά με βοήθησαν να συνειδητοποιήσω τη σημαντικότητα των όμορφων στιγμών πέρα από την επιστήμη.

Περιεχόμενα

Πίνακας Σχημάτων

Καταχωρήσεις

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο

Οι τεχνολογίες παράλληλης, κατανεμημένης επεξεργασίας (parallel, distributed execution) και κατανομής φόρτου (load balancing) αναπτύσσονται ραγδαία εδώ και δεκαετίες, κατά κύριο λόγο εξαιτίας των τεράστιων πλεονεκτημάτων που προσφέρουν στον υπολογισμό καθώς και στην παροχή υπηρεσιών, σε απόδοση, αξιοπιστία αλλά και σε κλιμακωσιμότητα. Επιτρέπουν στα συστήματα να δεσμεύουν δυναμικά πόρους και να κατανέμουν πόρους από ένα σύνολο διαθέσιμων συστημάτων, ξεπερνώντας τους πεπερασμένους πόρους ενός και μόνο μηχανήματος, καθιστώντας τα ικανά να εξυπηρετήσουν ακόμη και εκρηκτικές απαιτήσεις σε πόρους με το βέλτιστο τρόπο. Συνεπώς, εμφανίζεται πλέον η ευκαιρία για αποφυγή της πεπερασμένης κάθετης κλιμάκωσης (ένας κόμβος του συστήματος δεν μπορεί να αναβαθμίζεται επ άπειρον), για χάρη της οριζόντιας κλιμάκωσης, η οποία περιλαμβάνει απλώς την εγκατάσταση ενός νέου κόμβου στο σύστημα (Scalability).

Σε υπολογιστικά συστήματα κλίμακας, οποιοσδήποτε κόμβος που απαρτίζει το σύστημα ενδέχεται να αποτύχει. Στόχος της υλοποίησης οποιασδήποτε τέτοιας δομής είναι όχι η εξάλειψη της αναπόφευκτης αποτυχίας κάποιου κόμβου, αλλά η συνεχής εξυπηρέτηση και διαθεσιμότητα της υπηρεσίας παρά τα επιμέρους σφάλματα (Availability). Προς αυτήν την κατεύθυνση, η δυνατότητα αποθήκευσης της κατάστασης μιας διεργασίας καθώς και η μεταφορά της εκτέλεσής της σε άλλον κόμβο αυξάνει τη διαθεσιμότητα και την ανοχή σε σφάλματα (Fault tolerance) του συνολικού συστήματος.

Τα παραπάνω στοιχεία αποτελούν εφαλτήριο για αρκετά σενάρια χρήσης, όπως η μεταφορά και απομακρυσμένη εκτέλεση συγκεκριμένων απαιτητικών διεργασιών σε ισχυρότερα (σε υπολογιστικούς πόρους, χωρητικότητα αποθήκευσης ή εύρος δικτύου) συστήματα, η εκτέλεση διεργασιών workers ενός διακομιστή δικτύου όπως ο nginx σε διαφορετικούς κόμβους για βέλτιστη κατανομή του υπολογιστικού φόρτου, καθώς και η ενσωμάτωση της λογικής της μεταφοράς και απομακρυσμένης εκτέλεσης σε υπάρχοντα συστήματα κατανομής φόρτου.

Κατά τη διάρκεια της συγκεκριμένης διπλωματικής εργασίας σχεδιάζεται και υλοποιείται ένα σύστημα μεταφοράς και απομακρυσμένης εκτέλεσης διεργασιών για το λειτουργικό σύστημα Unix, το οποίο διατηρεί μία από τις πλέον χρησιμοποιούμενες και διαδεδομένες διεπαφή παράλληλης επεξεργασίας, το μοντέλο fork()/exec(). Ιδιαίτερη έμφαση δόθηκε στην διαφανή υλοποίηση (Transparency) του συστήματος, ώστε η διεπαφή να έρχεται σε πλήρη αντιστοιχία με την προϋπάρχουσα.

Τέλος, πλέον του κύριου συστήματος, αναπτύχθηκαν και πρόσθετα απομακρυσμένης διαδιεργασι-

ακής επικοινωνίας, τα οποία επιτρέπουν τη χρήση της απομακρυσμένης εκτέλεσης διεργασιών σε ακόμη περισσότερα σενάρια. Μια αρκετά ρεαλιστική περίπτωση σεναρίου χρήσης αποτελεί ένα κέλυφος όπως το bash το οποίο πλέον μπορεί να εκτελέσει piped διεργασίες σε διαφορετικά συστήματα.

1.2 Υπάρχουσες λύσεις

Τη στιγμή εκπόνησης της παρούσας διπλωματικής εργασίας δεν υπάρχουν λύσεις οι οποίες να προσφέρουν την διαφάνεια της παρούσας υλοποίησης. Πάραυτα, εντοπίζονται λύσεις οι οποίες προσφέρουν απομακρυσμένη εκτέλεση και μεταφορά διεργασιών ανάμεσα σε κόμβους ενός υπολογιστικού συστήματος. Αυτές οι λύσεις, υλοποιούν τη συγκεκριμένη λειτουργικότητα εισάγοντας ένα ακόμη επίπεδο στον πυρήνα, ώστε να παρέχουν ένα σύστημα μοναδικής εικόνας (SSI) ανάμεσα στους κόμβους του δικτύου, συναθροίζοντας τους συνολικούς υπολογιστικούς πόρους. Στη συνέχεια παραθέτουμε ορισμένες από αυτές τις λύσεις:

1.2.1 MOSIX/OpenMOSIX

Το MOSIX [**1**] (παλαιότερα ανοιχτού κώδικα, πλέον κλειστού κώδικα) και η πλέον αναπτυσσόμενη open source εκδοχή του, το OpenMOSIX [**2**], επιτρέπουν τη μεταφορά και την απομακρυσμένη εκτέλεση διεργασιών. Το MOSIX [**1**] αναπτύσσεται με κλειστό κώδικα, ενώ πλέον δεν απαιτεί τη χρήση τροποποιημένου πυρήνα και επιτρέπει εκτέλεση σε αρκετές διανομές των Linux.

Το OpenMOSIX [**2**] σταμάτησε να αναπτύσσεται τον Μάρτιο του 2008 και απαιτεί τροποποιημένες εκδόσεις του Linux πυρήνα, οπότε δεν αποτελεί βιώσιμη λύση για τα σύγχρονα υπολογιστικά συστήματα.

1.2.2 Kerrighed

Το Kerrighed [**3**], ακριβώς όπως τα MOSIX και OpenMOSIX, παρέχει μια ενοποιημένη οπτική ενός υπολογιστικού συστήματος. Αποτελείται από ένα σύνολο patches στον πυρήνα του Linux, βασίζεται στην έκδοση 2.6.30 του Linux kernel, ενώ παρέχει πληθώρα λειτουργιών πως διαμοιραζόμενο χώρο ονομάτων για αναγνωριστικά διεργασιών (PID), μεταφορά διεργασιών με ανοιχτά αρχεία και τμήματα διαμοιραζόμενης μνήμης (κατά το πρότυπο του System V).

1.2.3 Popcorn Linux

Το Popcorn Linux [**4**] στηρίζεται στην εκτέλεση replicated πυρήνων σε κάθε υπολογιστικό κόμβο, οι οποίοι επικοινωνούν μέσω ενός συστήματων μυνημάτων πυρήνα, το οποίο πυροδοτείται σε κάθε εκτέλεση της kexec κλήσης. Με αυτή τη μέθοδο παρέχεται η εικόνα ενός SSI, ακόμη και σε ετερογενή συστήματα. Σε κάθε αίτημα για συγκεκριμένες σελίδες της μνήμης, αυτές ενημερώνονται σε όλους τους πυρήνες του cluster συστήματος, ώστε να εξασφαλίζεται η απαραίτητη συνάφεια. Προκειμένου μια εφαρμογή να είναι συμβατή με τη παρούσα λύση, οφείλει να γίνει compile με τον κατάλληλο compiler που παρέχεται πό το ίδιο το λειτουργικό, επιδρώντας με αυτόν τον τρόπο στην διαφάνεια της λύσης. Απο τις προαναφερθείσες λύσεις, αυτή είναι η πλέον ενεργά αναπτυσσόμενη και ανοιχτού κώδικα.

1.3 Διάρθρωση της διπλωματικής εργασίας

Η παρούσα διπλωματική εργασία οργανώνεται σύμφωνα με την ακόλουθη μορφή:

- **Κεφάλαιο 2:** Σε αυτό το κεφάλαιο παρατίθεται το απαραίτητο υπόβαθρο για την κατανόηση των εννοιών και της υλοποίησης της απομακρυσμένης fork.
- **Κεφάλαιο 3:** Σε αυτό το κεφάλαιο αναλύεται η υλοποίηση της κλήσης απομακρυσμένου fork, καθώς και επιμέρους λύσεις που ακολουθήσαμε για τα προβλήματα που προέκυψαν κατά τη διάρκεια της ανάπτυξης της.
- **Κεφάλαιο 4:** Σε αυτό το κεφάλαιο περιγράφουμε τη διαδικασία αξιολόγησης της υλοποίησής της παρούσας διπλωματικής και παρουσιάζουμε τα συγκριτικά αποτελέσματα.
- **Κεφάλαιο 5:** Τέλος, διατυπώνουμε τα συμπεράσματά μας και ορίζουμε τη μελλοντική στοχοθεσία για την συνέχιση του έργου της παρούσας διπλωματικής.

Κεφάλαιο 2

Υπόβαθρο

2.1 Checkpoint - Restore Διεργασιών

Εφαρμογές που απαιτούν υψηλό αριθμό υπολογιστικών πόρων χρειάζονται συνήθως και αρκετό χρόνο εκτέλεσης. Ένα χαρακτηριστικό παράδειγμα τέτοιων εφαρμογών, που παραδοσιακά απαιτούσε υψηλούς χρόνους εκτέλεσης σε υπερ-υπολογιστικά συστήματα είναι οι προσομοιώσεις μοριακών δυναμικών, ρευστών καθώς και τα προβλήματα μαθηματικής βελτιστοποίησης. Επιπλέον, εφαρμογές που δεν κλιμακώνονται σωστά ενδέχεται να απαιτούν υψηλούς χρόνους εκτέλεσης, αφού μπορούν να χρησιμοποιήσουν αποδοτικά περιορισμένο αριθμό πόρων. Οι εφαρμογές που έχουν τέτοιες απαιτήσεις, εισάγουν στην εκτέλεση τον κίνδυνο αποτυχίας, είτε λόγω σφάλματος του υλικού, είτε λόγω σφάλματος κατά την ανάπτυξη του ίδιου του λογισμικού. Επιπροσθέτως, τα υπερυπολογιστικά συστήματα συχνά περιορίζουν την εκτέλεση οποιουδήποτε προγράμματος σε συγκεκριμένα όρια, εισάγωντας μια πολιτική ορθής χρήσης, προκειμένου να αποφευχθούν φαινόμενα υπερβολικής κατανάλωσης πόρων από μια και μόνο εκτέλεση. Σε κάθε μία απο τις παραπάνω περιπτώσεις, η εκτέλεση του προγράμματος αυτού θα τερματιστεί. Οι συνέπειες του τερματισμού είναι κοστοβόρες απο άποψη χρόνου εκτέλεσης και υπολογιστικών πόρων. Λύση σε αυτό το πρόβλημα είναι η αποθήκευση της κατάστασης του προγράμματος ανα τακτά χρονικά διαστήματα. Η διαδικασία αυτή ονομάζεται checkpointing και η κάθε αποθηκευμένη κατάσταση checkpoint [**5**]. Σε περίπτωση αποτυχίας εκτέλεσης, δε χρειάζεται πλέον η επανέναρξη του προγράμματος από το αρχικό στάδιο εκτέλεσης, αλλά η αποσφαλμάτωση και η επαναφορά από το τελευταίο λειτουργικό checkpoint. Η διαδικασία της επαναφοράς καλείται Restoring. Ανάλογα με την υλοποίηση, η κατάσταση μίας εφαρμογής αποθηκεύει κατά το checkpointing δεδομένα όπως ο εικονικός χώρος διευθύνσεων, η κατάσταση των καταχωρητών και οι περιγραφητές ανοιχτών αρχείων.

Δύο είναι τα βασικότερα είδη μηχανισμών checkpointing:

2.1.1 System Level Checkpointing

Το checkpointing σε επίπεδο συστήματος εκτελεί αποθήκευση και επαναφορά διεργασιών δημιουργώντας ένα πλήρες αντίγραφο της μνήμης της. Αυτό το είδος checkpointing δεν απαιτεί αλλαγές στον κώδικα της εκάστοτε εφαρμογής, ενισχύοντας τη διαφάνεια και την ευκολία χρήσης, ενώ η αποθήκευση της κατάστασης μπορεί, υπό προϋποθέσεις, να επιτελεστεί σε οποιαδήποτε στιγμή χωρίς πρότερη προετοιμασία της εφαρμογής. Το κύριο πλεονέκτημα αυτής της προσέγγισης είναι το γεγονός ότι δεν απαιτούνται αλλαγές στον κώδικα της εφαρμογής. Επιπλόν, σε πολλές περιπτώσεις, οι μηχανισμοί

checkpoint restore συνεργάζονται με τον πυρήνα ώστε να έχουν πρόσβαση σε δομές πληροφοριών για την κάθε διεργασία που αποθηκεύουν. Μειονέκτημα τους αποτελεί το μεγάλο μέγεθος του κάθε checkpoint, δεδομένου ότι αποθηκεύουν το σύνολο της μνήμης μιας διεργασίας [**5**].

2.1.2 Application Level Checkpointing

Στο checkpointing σε επίπεδο εφαρμογής, ο προγραμματιστής είναι υπεύθυνος να εισάγει τη λογική του checkpoint/restore στον κωδικα της εφαρμογής. Αντίθετα με τη προηγούμενη κατηγορία, απαιτούνται αλλαγές στον κώδικα, ενώ, αν η εφαρμογή έχει αναπτυχθεί χωρίς τη λογική του checkpoint restore, απαιτείται από τροποποίηση του κώδικα και αναπροσαρμογή (refactoring), μέχρι και ανάπτυξη εκ θεμελίων. Αν και αυτό αποτελεί ένα σημείο δυσχρηστίας, η συγκεκριμένη κατηγορία υλοποιήσεων τείνει να παράγει αρχεία κατάστασης μικρότερα σε μέγεθος απο αυτά του checkpointing σε επίπεδο συστήματος. Τα αρχεία κατάστασης περιλαμβάνουν μόνο το ελάχιστο ποσό πληροφορίας που απαιτείται για να επαναφερθει η διαδικασία, αποφεύγοντας την αποθήκευση της πλήρους μνήμης της εφαρμογής [**5**].

Στον Πίνακα 2.1 παραθέτουμε τις κυρίαρχες διαθέσιμες υλοποιήσεις Checkpoint Restore μηχανισμών ανά κατηγορία, καθώς και μια σύγκριση των βασικών αξιοσημείωτων τους λειτουργικοτήτων [**5**].

2.1.3 Chcekpoint/Restore in Userspace - CRIU

Από τις προαναφερθείσες λύσεις επιλέγεται το Checkpoint Restore In Userspace (CRIU) για την ανάπτυξη της υλοποίησης της παρούσας διπλωματικής εργασίας. Το CRIU υλοποιεί τη λειτουργικότητα Checkpoint/Restore σε επίπεδο συστήματος, αλλά σε χώρο χρήστη. Αυτό επιτρέπει να μην απαιτούνται μετατροπές ή προσθήκες στον πυρήνα του συστήματος, αν και εδώ πρέπει να σημειωθεί πως ο πυρήνας περιέχει μετατροπές ειδικά υλοποιημένες για το CRIU στο κύριο branch [**6**]. Επειδή είναι υλοποιημένο σε χώρο χρήστη, προσφέρει απόλυτη διαφάνεια στις εφαρμογές, οπότε άρμόζει στην υλοποιησή μας. Η διαδικασία του checkpoint στα πλαίσια του CRIU καλείται dump, ενώ η διαδικασία του restore παραμένει με την ίδια ονομασία, οπότε στο εξής θα χρησιμοποιούμε την ορολογία του CRIU στη μελέτη μας.

Το CRIU υποστηρίζει δύο μεθόδους εκτέλεσης των παρεχόμενων λειτουργιών. Ο χρήστης μπορεί είτε να εκτελέσει το /bin/criu εκτελέσιμο (swrk mode), είτε να εκκινήσει την CRIU υπηρεσία η οποία και εξυπηρετεί όλα τα αιτήματα προς το αυτό (RPC mode). Οι δύο αυτοί τρόποι δεν αναπτύσσονται παράλληλα και ο RPC βρίσκεται στο επίκεντρο τη στιγμή συγγραφής της παρούσας διπλωματικής. Επιπλέον, οι δύο αυτοί τρόποι δεν υλοποιούν το σύνολο της λειτουργικότητας που παρέχει το criu, όπως θα δούμε και στη συνέχεια. Συνεπώς, αποτελεί επιλογή του προγραμματιστή ο τρόπος που θα χρησιμοποιηθεί. Επιπλέον, στα πλαίσια του CRIU, παρέχεται βιβλιοθήκη διεπαφής για προγράμματα υλοποιημένα τόσο σε C [**7**] όσο και σε Python [**8**]. Οι βιβλιοθήκες αυτές, αποτελούν μια ευκολότερη διεπαφή για τους τρόπους επικοινωνίας με το CRIU και στηρίζονται στην ανταλλαγή μηνυμάτων μέσω protocol buffers [**9**].

Το CRIU διαθέτει συγκεκριμένα πλεονεκτήματα συγκριτικά με τις εναλλακτικές λύσεις, τα οποία και παρατίθενται ακολούθως:

Υλοποίηση	**CRIU**	**DMTCP**	**BLCR**	**SCR**
Τύπος	System level	System level	System level	Application level
Πυρήνας	Standard >3.11	Standard	Standard with loaded module	Standard
Προφόρτωση	Όχι	Ναι	Ναι	Όχι
Απαιίτηση αλλαγής κώδικα	Όχι	Όχι	Ναι	Ναι
Containers	Ναι	Όχι	Όχι	Όχι
Βιβλιοθήκες παράλληλης/ κατανεμημένης επεξεργασίας	Όχι (υπό ανάπτυξη)	Ναι	Ναι	Ναι

Πίνακας 2.1: Σύγκριση γνωστών βιβλιοθηκών Checkpoint/Restore [**11**][**5**]

- Εκτελείται σε χώρο χρήστη.
- Δεν απαιτεί τροποποίηση του κώδικα των εφαρμογών τις οποίες αποθηκεύει, εξυπηρετώντας την απαίτηση για διαφάνεια της υλοποίησής μας.
- Εκτελείται χωρίς κάποια τροποποίηση στον πυρήνα των κόμβων, δεδομένου ότι ικανοποιείται η απαίτηση για έκδοση πυρήνα μεταγενέστερη της 3.11.
- Υποστηρίζει την αποθήκευση και η επαναφορά συνδέσεων στο δίκτυο.
- Υποστηρίζει την αντικατάσταση συγκεκριμένων αρχείων, τα οποία ήταν ανοιχτά κατά τη διαδικασία του checkpoint.
- Διαθέτει βιβλιοθήκη συμβατή με προγράμματα γραμμένα σε C [**7**], γεγονός που μας επιτρέπει να το ενσωματώσουμε στην υλοποίησή.

Οφείλουμε να σημειώσουμε πως το CRIU δεν αποτελεί πανάκεια όσον αφορά το ζήτημα του Checkpoint/Restore διεργασιών. Κατά τη διάρκεια της υλοποίσης διαπιστώθηκαν ζητήματα τα οποία και αντιμετωπίσαμε είτε με συγκεκριμένη μεθοδολογία στην ίδια την υλοποίηση, είτε με τροποποιήσεις/προσθήκες στον κώδικα του CRIU. Συγκεκριμένα:

- Οι διεργασίες, για λόγους διαφάνειας, επαναφέρονται με ακριβώς το ίδιο αναγνωριστικό διεργασίας (Process Identifier - PID). Αυτό το γεγονός οδηγεί σε σφάλματα, αν κατά την επαναφορά υπάρχει διεργασία που έχει λάβει το ίδιο αναγνωριστικό [**10**].
- Λόγω της κλίμακας του project και της τρέχουσας στοχοθεσίας της ομάδας ανάπτυξής του προς υλοποιήσεις με χρήση RPC, η βιβλιοθήκη C [**7**] δεν είναι ανανεωμένη για αρκετές λειτουργίες.

2.2 Κατανεμημένα συστήματα αρχείων

Με τον όρο κατανεμημένα συστήματα αρχείων αναφερόμαστε στα συστήματα εκείνα, τα οποία παρέχουν πρόσβαση σε ένα σύνολο αρχείων και καταλόγων σε πολλαπλούς διασυνδεδεμένους κόμβους. Τα

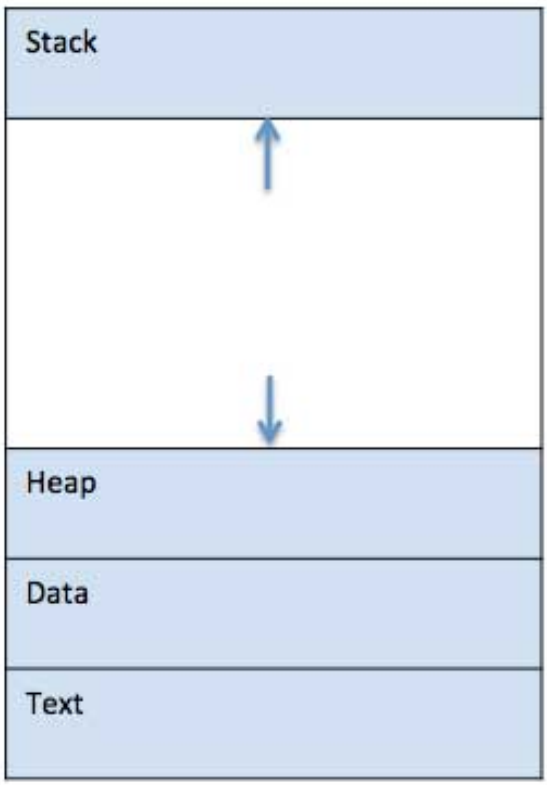
συστήματα αυτά παρουσιάζουν διαφοροποιήσεις ως προς τη διεπαφή, τη συμβατότητα και την απόδοση των ενεργειών εγγραφής και ανάγνωσης. Αντί να παρέχουν κοινή πρόσβαση σε μία συσκευή αποθήκευσης, υλοποιούν ένα δικτυακό πρωτόκολλο το οποίο δίνει αυτή τη διεπαφή στον τελικό χρήστη [**12**]. Στα πλαίσια της συγκεκριμένης διπλωματικής εργασίας, θα ασχοληθούμε με το NFS, χωρίς αυτό να σημαίνει πως δεν υφίστανται άλλες αξιόλογες υλοποιήσεις κατανεμημένων συστημάτων αρχείων (GlusterFS, OpenIO, LizardFS κ.α.).

Το NFS είναι ένα πρωτόκολλο κατανεμημένου συστήματος αρχείων, το οποίο παρέχει πλήρη διαφάνεια στη διεπαφή πρόσβασης αρχείων για τους απομακρυσμένους χρήστες. Οι υλοποιήσεις NFS παρέχουν ένα σύστημα το οποίο στηρίζεται στην ύπαρξη daemons σε κάθε client οι οποίοι μεταφράζουν τα αιτήματα πρόσβασης σε αρχεία σε RPC κλήσεις μέσω συγκεκριμένων αρθρωμάτων του πυρήνα, και daemons σε κάθε server οι οποίοι δέχονται αυτές τις RPC κλήσεις και ικανοποιούν καταλλήλως τα αιτήματα αυτά.

2.3 Διεργασίες & μηχανισμός fork

2.3.1 Η διεργασία

Μια διεργασία, σε κάθε λειτουργικό σύστημα, είναι ένα πρόγραμμα το οποίο εκτελείται. Αποτελείται από το τμήμα κειμένου, το οποίο είναι ο κώδικάς του προγράμματος, τη τρέχουσα δραστηριότητα, η οποία υποδεικνύεται απο τον μετρητή προγράμματος, καθώς και από τα περιεχόμενα των καταχωρητών του επεξεργαστή. Επιπλέον, πεεριλαμβάνει τη στοίβα της διεργασίας, η οποία αποθηκεύει προσωρινά δεδομένα, ένα τμήμα δεδομένων, προορισμένο για τις καθολικές μεταβλητές, καθώς και έναν σωρό, ο οποίος βρίσκεται στη μνημη και δεσμεύεται δυναμικά κατά τον χρόνο εκτέλεσης της διεργασίας [**12**]. Η δομή μιας διεργασίας παρουσιάζεται στο σχήμα 2.1.

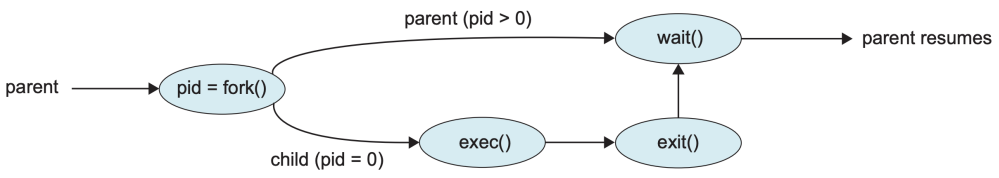


Σχήμα 2.1: Δομή διεργασίας στη μνήμη [**12**]

2.3.2 Δημιουργία διεργασίας και κλήση συστήματος fork

Στα βασισμένα στο UNIX λειτουργικά συστήματα, κάθε διεργασία προσδιορίζεται από τον προσδιοριστή διεργασίας της (PID), ο οποίος είναι ένας μοναδικός ακέραιος αριθμός. Κάθε νέα διεργασία δημιουργείται από τη κλήση συστήματος fork(). Η νέα διεργασία αποτελείται από ένα αντίγραφο του χώρου διευθύνσεων της αρχικής. Οι δύο διεργασίες πλέον εκτελούνται ανεξάρτητα μετά την εντολή fork(). Συγκεκριμένα, ο κωδικός επιστροφής της fork() είναι μηδέν για τη νέα (θυγατρική διεργασία), ενώ ο μη μηδενικός προσδιοριστής διεργασίας της θυγατρικής επιστρέφεται στη γονική. Η θυγατρική διεργασία κληρονομεί ιδιότητες χρονοπρογραμματισμού καθώς και πόρους όπως ανοικτούς περιγραφητές αρχείων από τη γονική [**12**].

Στη συνέχεια, η γονική διεργασία μπορεί να εκτελέσει ξανά fork() και να δημιουργήσει νέες θυγατρικές ή αν δεν έχει τι άλλο να κάνει όσο εκτελούνται οι θυγατρικές να καλέσει τη κλήση συστήματος wait() και να περιμένει τον τερματισμό των θυγατρικών διεργασιών που δημιούργησε, δηλαδή την κλήση exit() από αυτές [**12**]. Η διαδικασία αυτή απεικονίζεται στο Σχήμα 2.2.



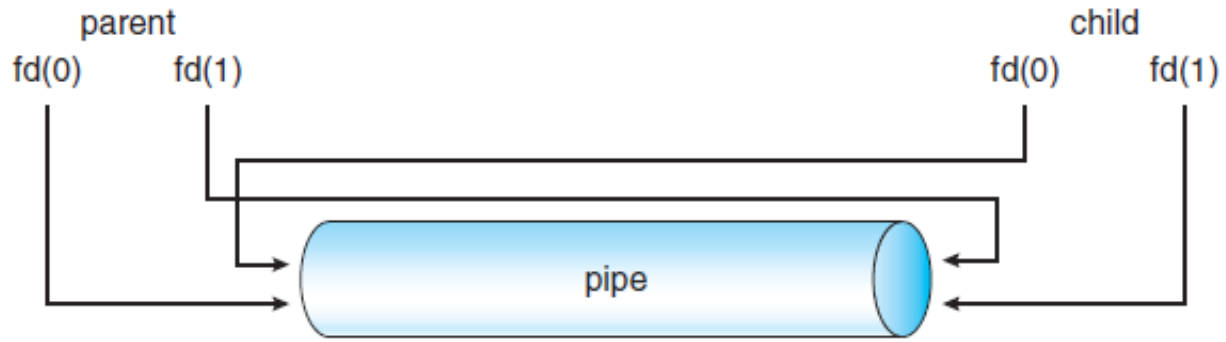
Σχήμα 2.2: Δημιουργία διεργασίας με την κλήση συστήματος fork()

2.4 Μηχανισμοί διαδιεργασιακής επικοινωνίας

Στην ενότητα αυτή μελετώνται οι μηχανισμοί διαδιεργασιακής επικοινωνίας (Inter-Process Communication - IPC) με τους οποίους ασχολείται η παρούσα υλοποίηση. Η ύπαρξή τους είναι απαραίτητη, αφού, υπό κανονικές συνθήκες, δύο διεργασίες, ακόμη και αν έχουν σχέση γονέα παιδιού, δεν μοιράζονται τμήματα μνήμης. Υπάρχουν δύο βασικά μοντέλα διαδιεργασιακής επικοινωνίας, το κοινής μνήμης (shared memory) και το βασιζόμενο στο πέρασμα μηνυμάτων (message passing). Στο μοντέλο κοινής μνήμης ορίζονται περιοχές μνήμης κοινές για τις συνεργαζόμενες διεργασίες, οπότε και αυτές διαβάζουν ή γράφουν δεδομένα στις περιοχές αυτές. Στο μοντέλο του περάσματος μηνυμάτων οι διεργασίες ανταλλάσσουν μηνύματα μέσω ενός ή περισσότερων συμφωνημένων διαύλων [**12**].

2.4.1 Διοχετεύσεις

Μια διοχέτευση (pipe) λειτουργεί ως ένας δίαυλος επικοινωνίας για δύο διεργασίες. Υπήρξε από τους πρώτους μηχανισμούς IPC στα συστήματα UNIX. Η υλοποίηση μας, που θα περιγραφεί σε επόμενη ενότητα, υποστηρίζει μόνο τον απλούστερο τύπο διοχετεύσεων, τις κοινές/ανώνυμες διοχετεύσεις. Οι κοινές διοχετεύσεις είναι μονόδρομες και οι δύο διεργασίες, πρωτού τις χρησιμοποιήσουν, πρέπει να αναλάβουν είτε ρόλο καταναλωτή (εκείνος που εκτελεί ανάγνωση στη διοχέτευση), είτε παραγωγού (εκείνος που εκτελεί εγγραφή στη διοχέτευση). Σε συστήματα UNIX μια διοχέτευση κατασκευάζεται με τη συνάρτηση pipe(int fd[]), η οποία επιστρέφει στην πρώτη θέση του πίνακα fd το άκρο εγγραφής και στη δεύτερη το άκρο ανάγνωσης. Τα δύο άκρα αντιστοιχούν σε δύο ξεχωριστά αρχεία, οπότε ο πίνακας όρισμα της pipe() είναι ένας πίνακας ακεραίων ώστε να φιλοξενήσει δύο ξεχωριστούς περιγραφητές αρχείων[**12**]. Η πρόσβαση στα pipes γίνεται μέσω συνηθισμένων κλήσεων read() και write(), στο πνεύμα της βασικής αρχής του UNIX πως όλες οι δομές είναι αρχεία [**13**]. Τα δεδομένα που γράφονται στο άκρο εγγραφής του pipe μπορούν να διαβαστούν από το άκρο ανάγνωσης (Σχήμα 2.3) Επιπλέον, η πρόσβαση στα δεδομένα γίνεται με FIFO σειρά.



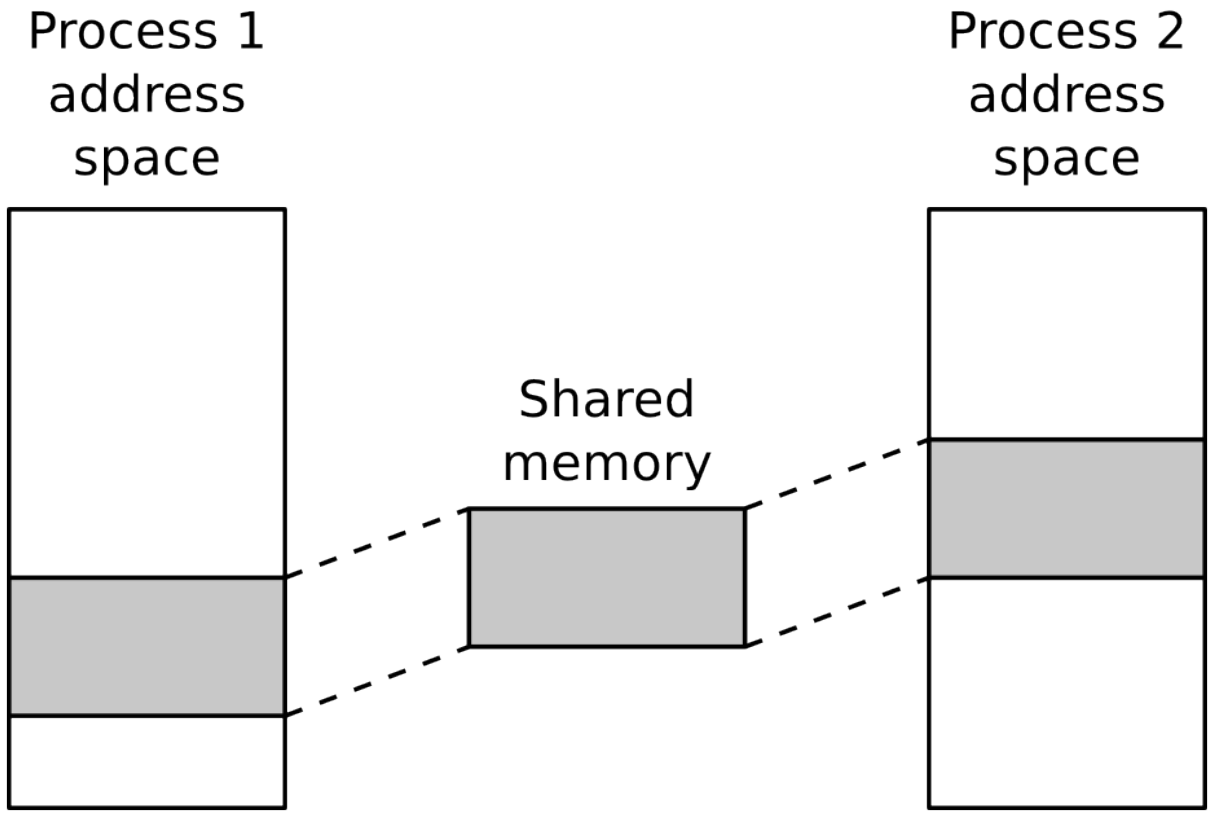
Σχήμα 2.3: Διοχέτευση (pipe) ανάμεσα σε διεργασίες γονέα παιδιού

2.4.2 Σήματα

Τα σήματα είναι ένας από τους παλαιότερους μηχανισμούς διαδιεργασιακής επικοινωνίας του UNIX. Χρησιμοποιούνται για να ειδοποιήσουν μια ή περισσότερες διεργασίες, ασύγχρονα, για κάποιο γεγονός.

2.4.3 Κοινή μνήμη

Οι διεργασίες που επικοινωνούν μέσω αυτού του μηχανισμού, εγκαθιδρύουν μια περιοχή κοινής μνήμης. Τυπικά, μια διεργασία ορίζει μια περιοχή μνήμης της ως κοινή και κάποια άλλη επισυνάπτει (attach) αυτό το τμήμα στο δικό της χώρο διευθύνσεων, όπως φαίνεται στο Σχήμα 2.4. Οι διεργασίες είναι υποχρεωμένες να διασφαλίσουν πως δεν θα εκτελέσουν εγγραφή στην ίδια διεύθυνση μνήμης ταυτόχρονα. Επιπλέον, με χρήση κοινής μνήμης για διαδιεργασιακή επικοινωνία, προκύπτουν προβλήματα συνέπειας κοινής μνήμης και προκειμένου αυτά να αντιμετωπιστούν εισάγονται καθυστερήσεις κατά το διαμοιρασμό **[12]**. Οι δύο υλοποιήσεις κοινής μνήμης που θα περιγράψουμε εμφανίζουν εξαιρετικές ομοιότητες, με τις κύριες διαφορές να εντοπίζονται στη διεπαφή που προσφέρεται στον προγραμματιστή, οπότε επιλέξαμε να τις υλοποιήσουμε στο σύστημα που προτείνουμε.



Σχήμα 2.4: Κοινή μνήμη ανάμεσα σε δύο διεργασίες

2.4.3.1 Κοινή μνήμη POSIX

Σε συστήματα POSIX, ο μηχανισμός κοινής μνήμης προϋποθέτει την απεικόνιση αρχείων στη μνήμη δυο (ή περισσότερων) διεργασιών. Μια διεργασία δημιουργεί ένα αντικείμενο κοινής μνήμης μέσω της κλήσης συστήματος shm_open(), ορίζοντας τόσο το αναγνωριστικό όνομα του, το οποίο θα χρησιμοποιηθεί από τις υπόλοιπες διεργασίες για πρόσβαση στο ίδιο αντικείμενο, όσο και λοιπές παραμέτρους όπως δικαιώματά πρόσβασης στο αρχείο. Το αρχείο αυτό αποθηκεύεται σε ένα ειδικό σύστημα αρχείων, το tmpfs. Στη συνέχεια, η ίδια διεργασία ορίζει το μέγεθος του αντικειμένου/αρχείου σε bytes χρησιμοποιώντας τη συνάρτηση ftruncate(). Ακολούθως, με τη mmap() απεικονίζει το αρχείο στη

μνήμη της διεργασίας, λαμβάνοντας έναν δείκτη ώστε να προσπελαύνει αυτό το τμήμα κοινής μνήμης. Τέλος, καταργεί την απεικόνιση του αρχείου στον εικονικό χώρο διευθύνσεων με τη munmap(), κλείνει το αρχείο κοινής μνήμης χρησιμοποιώντας τον περιγραφητή του και στη συνέχεια εκτελεί τη shm_unlink() για να διαγράψει το αντικείμενο κοινής μνήμης, το οποίο είναι kernel persistent και αν δεν τερματίσουν ή εκτελέσουν unlink() όλες οι διεργασίες που έχουν πρόσβαση σε αυτό παραμένει μέχρι την απενεργοποίηση του μηχανήματος **[12]** **[14]**.

2.4.3.2 Κοινή μνήμη System V

Στα συστήματα System V, κάθε αντικείμενο κοινής μνήμης αναγνωρίζεται από έναν μοναδικό ακέραιο αριθμό, με τον ίδιο τρόπο που η κλήση συστήματος open() επιστρέφει έναν ακέραιο περιγραφητή αρχείου. Μια διεργασία αρχικά υπολογίζει ένα αναγνωριστικό κλειδί (το οποίο ομοίως με το POSIX χρησιμοποιείται για την αναφορά από όλες τις διεργασίες σε αυτό το αντικείμενο) που παράγεται μέσω της συνάρτησης ftok()**[15]**. Στη συνέχεια, δημιουργείται ή ανοίγει ένα αντικείμενο κοινής μνήμης με χρήση της shmget(), η οποία επιστρέφει το μοναδικό ακέραιο για αυτό το αντικείμενο. Κατά την κλήση της shmget() ορίζονται παράμετροι σχετικοί με το αντικείμενο κοινής μνήμης, όπως δικαιώματα πρόσβασης και το μέγεθος του τμήματος μνήμης. Έπειτα, η απεικόνιση του τμήματος αυτού στη μνήμη της διεργασίας γίνεται μέσω της shmat() η οποία επιστρέφει, όπως η mmap() προηγουμένως, έναν δείκτη για προσπέλαση του τμήματος από τη διεργασία. Όταν μια διεργασία εκτελέσει τις επιθυμητές ενέργειες στο τμήμα κοινής μνήμης καταργεί την απεικόνιση του στη μνήμη της διεργασίας καλώντας τη shmdt() **[16]** **[15]**.

2.5 Προφόρτωση

Η προφόρτωση αποτελεί μία λειτουργικότητα που παρέχεται από τον dynamic linker (ld), διαθέσιμη στα περισσότερα UNIX based συστήματα. Επιτρέπει τη φόρτωση ενός shared object από το χρήστη, πρωτού φορτωθούν όλες οι υπόλοιπες shared libraries που συνδέονται με ένα εκτελέσιμο. Η προφόρτωση αποτελεί σημαντικότατο εργαλείο στη προσπάθειά μας για πλήρη διαφάνεια της υλοποίησής μας. Προκειμένου να επιτευχθεί αυτό, οφείλει να διατηρεί την ίδια υπογραφή οποιασδήποτε συνάρτησης επιθυμεί να προφορτώσουμε. Επιπλέον, ο προγραμματιστής έχει την υποχρέωση να χρησιμοποιήσει την ίδια πολιτική σχετικά με τις τιμές επιστροφής και το χειρισμό των σφαλμάτων, ώστε να μην απαιτείται αλλαγή στον κώδικα των εκάστοτε εφαρμογών.

Κεφάλαιο 3

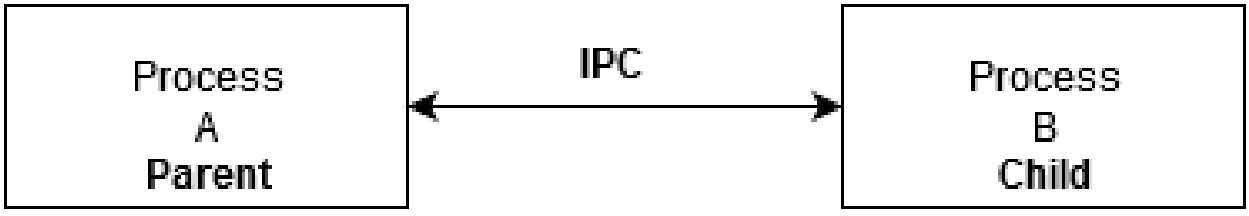
Υλοποίηση

Στο κεφάλαιο αυτό θα περιγράψουμε κάθε στάδιο της υλοποίησης του συστήματος μεταφοράς και απομακρυσμένης εκτέλεσης διεργασιών (ή αλλιώς rfork()) καθώς και τις προκλήσεις ή τη μη υλοποιημένη λειτουργικότητα σε κάθε ένα από αυτά.

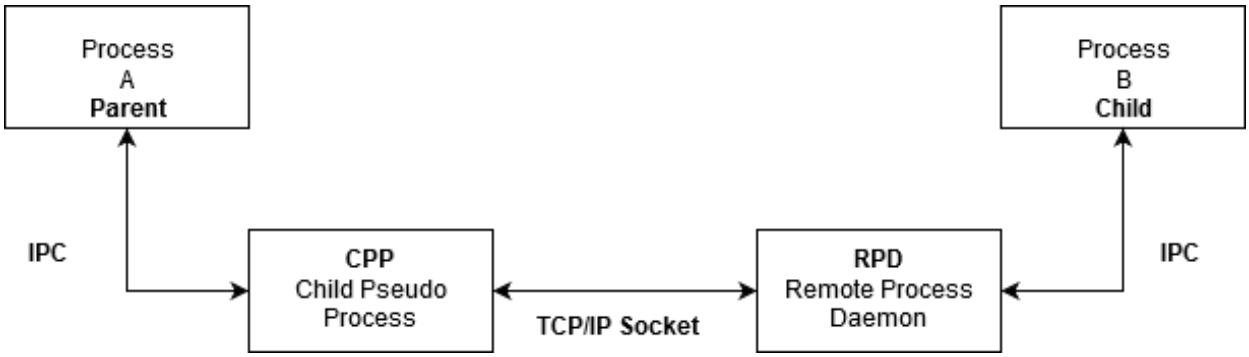
3.1 Επισκόπησή υλοποίησης

Κρίνουμε σκόπιμο να σκιαγραφήσουμε τα κύρια τμήματα της υλοποίησης πρωτού να εμβαθύνουμε στο κάθε επιμέρους. Όταν χρησιμοποιείται η κλήση συστήματος fork(), υπό κανονικές συνθήκες η θυγατρική διεργασία εκτελείται στο ίδιο σύστημα με τη καλούσα διεργασία (Σχήμα 3.1). Ο πυρήνας παρέχει ένα σύνολο από μηχανισμούς διαδιεργασιακής επικοινωνίας, συμπεριλαμβανομένων αυτών που περιγράφηκαν προηγουμένως (Ενότητα 2.4). Όταν οι δύο διεργασίες εκτελούνται σε διαφορετικούς hosts, οι μηχανισμοί αυτοί παύουν να είναι λειτουργικοί, χωρίς τουλάχιστον να τους παραμετροποιήσουμε. Προς αυτή τη κατεύθυνση, δημιουργούμε δύο νέες διεργασίες οι οποίες παρεμβάλλονται των αρχικών, και έχουν ως κύριο στόχο να αναμεταδίδουν οποιαδήποτε IPC πληροφορία μέσω δικτύου στις αρχικές (Σχήμα 3.2). Οι δύο αυτές διεργασίες ονομάζονται Child Pseudo Process (CPP) και Remote Process Daemon (RPD). Προκειμένου να μη θυσιάσουμε σε transparency, η διεργασία CPP αποτελεί παιδί της διεργασίας πατέρα και η διεργασία RPD γονέα της αρχικής διεργασίας παιδί. Η διεργασία CPP δημιουργείται κατά την κλήση της προφορτωμένης fork() ενώ η RPD από την υπηρεσία WDS η οποία εκτελείται σε κάθε κόμβο που είναι ικανός να φιλοξενήσει διεργασίες παιδιά. Η διαδικασία δημιουργίας των ενδιάμεσων διεργασιών αναλύεται στην Ενότητα 3.2.

Οποιαδήποτε πληροφορία διέρχεται από το socket που συνδέει τις διεργασίες CPP και RPD ορίζεται από ένα πρωτόκολλο επικοινωνίας που σχεδιάστηκε ειδικά για τους σκοπούς της παρούσας διπλωματικής εργασίας.



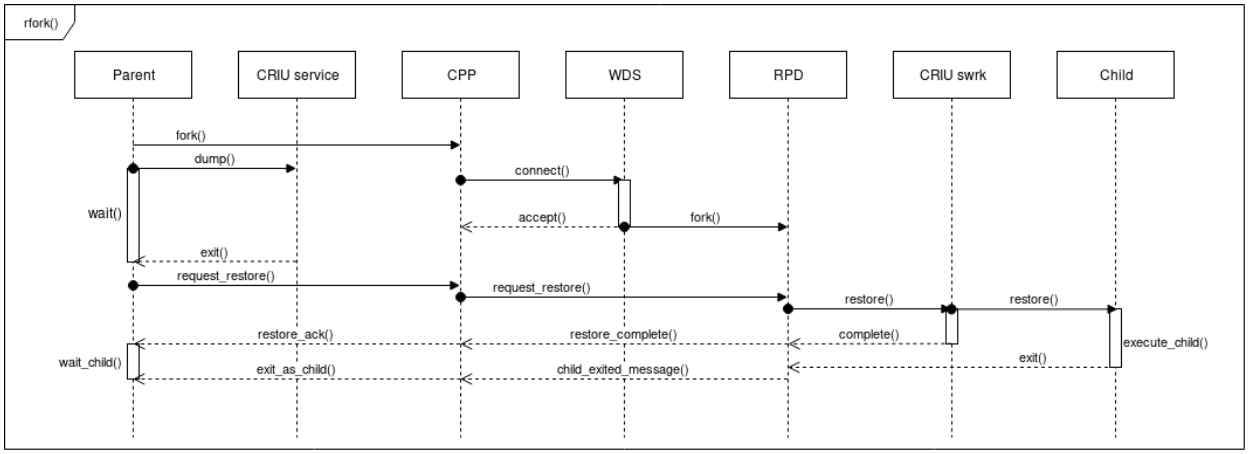
Σχήμα 3.1: fork() με τις δυο διεργασίες στο ίδιο μηχάνημα



Σχήμα 3.2: Εκτέλεση fork() σε απομακρυσμένο υπολογιστικό κόμβο

3.2 Δημιουργία ενδιάμεσων διεργασιών

Προκειμένου να ικανοποιήσουμε την αρχική απαίτηση για διαφάνεια (transparency) της υλοποίησής, υιοθετούμε την προσέγγιση της εισαγωγής ενδιάμεσων διεργασιών ανάμεσα στις διεργασίες πατέρα και παιδί, οι οποίες έχουν ως ρόλο να δίνουν την εντύπωση πως η εκτέλεση γίνεται σε ένα τοπικό σύστημα, προωθώντας κάθε στοιχείο διαδιεργασιακής επικοινωνίας μέσω μίας συγκεκριμένης ζεύξης. Στην υλοποίησή, η ζεύξη αυτή είναι ένα TCP socket και η πληροφορία που διακινείται ακολουθεί το πρωτόκολλο που περιγράφεται στην ενότητα 3.3.



Σχήμα 3.3: Διαφανής δημιουργία ενδιάμεσων διεργασιών

3.3 Πρωτόκολλο επικοινωνίας CPP - RPD

Σκοπός της ζεύξης TCP ανάμεσα σε CPP και RPD είναι η μετάδοση οποιασδήποτε επικοινωνίας ανάμεσα σε διεργασία πατέρα και διεργασία παιδί (δεδομένα διοχετεύσεων, σήματα κ.α.). Η επικοινωνία στη συγκεκριμένη υλοποίηση γίνεται μέσα από τη δομή frame, η οποία και μεταφέρει όλη την πιθανή πληροφορία. Υπό κανονικές συνθήκες, αυτή η πληροφορία, όντας μια δομή, θα μπορούσε να μεταφερθεί αυτούσια ως μια σειρά από bytes μέχρι τον προορισμό. Προκειμένου όμως να μην εισάγουμε επιπλέον απαιτήσεις ή περιορισμούς (για παράδειγμα ως προς το endianness), επιλέξαμε να υλοποιήσουμε μια, προσαρμοσμένη στα μέτρα της συγκεκριμένης εργασίας, βιβλιοθήκη σειριοποίησης (serialization), και βάσει αυτής να κατασκευάζουμε συμβολοσειρές οι οποίες μεταφέρονται μέσω της

ζεύξης TCP/IP. Εναλλακτική προσέγγιση αποτελεί η χρήση μηνυμάτων protobuf όπως αυτά της υλοποίησης Google protobuf **[9]**, όμως υλοποιήσαμε τη δική μας βιβλιοθήκη ώστε να αποφύγουμε επιβραδύνσεις από τον γενικότερο χαρακτήρα και την επαυξημένη λειτουργικότητα μιας έτοιμης λύσης. Οι δύο απομακρυσμένες διεργασίες CPP και RPD επικοινωνούν μέσω ενός TCP socket, το οποίο αρχικά δημιουργείται ανάμεσα στις CPP και WDS και στη συνέχεια κληρονομείται από τη δεύτερη στη διεργασία RPD. Τα μηνύματα του πρωτοκόλλου έχουν την ακόλουθη δομή:

```
<message_type>|<message_payload>
```

Καταχώρηση 3.1: Γενική μορφή μηνύματος του rfork protocol

Κατά τη διάρκεια της μεταφοράς και της επανέναρξης της διεργασίας παιδί, καθώς και κατά τον τερματισμό της, αποστέλλονται συγκεκριμένα μηνύματα ώστε να επιτυγχάνεται ο απαραίτητος συγχρονισμός ανάμεσα στις διεργασίες CPP και RPD. Τα μηνύματα αυτά είναι των ακόλουθων τύπων (<message_type>):

- Type 0: Μήνυμα ανακοίνωσης τερματισμού της θυγατρικής διεργασίας. Πυροδοτεί την αποστολή του σήματος SIGCHLD από τη διεργασία CPP στη γονική διεργασία.

```
*** SE|<sender_pid>|SIGCHLD|<exit_code> ***

```

Καταχώρηση 3.2: Μήνυμα τερματισμού θυγατρικής διεργασίας

- Type 2: Αποστέλλεται από τη διεργασία CPP όταν ολοκληρωθεί η διαδικασία αποθήκευσης της κατάστασης της θυγατρικής διεργασίας (dump) στον κόμβο προορισμού, προκειμένου να πυροδοτήσει τη διαδικασία επαναφοράς (restore) της θυγατρικής στον κόμβο προορισμού. Η υλοποίηση επιβάλλει την αποθήκευση των αρχείων κατάστασης της κάθε διεργασίας σε κατάλογο του διαμοιραζόμενου συστήματος αρχείων NFS και συγκεκριμένα στο directory με όνομα τον προσδιοριστή διεργασίας PID της μόλις αποθηκευμένης διεργασίας. Με τη λήψη ενός μηνύματος αυτού του τύπου, ο κόμβος προορισμού γνωρίζει ποιόν κατάλογο αρχείων οφείλει να χρησιμοποιήσει για την επαναφορά της θυγατρικής διεργασίας.

```
*** R|<dump_pid> ***

```

Καταχώρηση 3.3: Μήνυμα πυροδότησης επαναφοράς θυγατρικής διεργασίας

- Type 5: Αποστέλλεται από τη διεργασία RPD όταν ολοκληρωθεί επιτυχώς η διαδικασία της επαναφοράς της θυγατρικής διεργασίας στον απομακρυσμένο κόμβο, προκειμένου να πυροδοτήσει τη συνέχιση της εξέλιξης της γονικής διεργασίας.

```
*** RA|<null> ***

```

Καταχώρηση 3.4: Μήνυμα επιβεβαίωσης επιτυχούς επαναφοράς θυγατρικής διεργασίας

Εκτός των προαναφερθέντων τύπων μηνυμάτων, υπάρχει ένας επιπρόσθετος αριθμός ο οποίος αφορά την ανταλλαγή πληροφοριών σχετικά με τους μηχανισμούς διαδιεργασιακής επικοινωνίας. Συγκεκριμένα

- Type 1: Μήνυμα αποστολής σήματος από και προς οποιαδήποτε επιθυμητή διεύθυνση. Αποστέλλεται όταν είτε η CPP είτε η RPD ειδοποιηθούν για σήμα από τον signalfd.

```
*** S|<sender_pid>|<signal_no> ***

```

Καταχώρηση 3.5: Μήνυμα αποστολής σήματος

- Type 3: Μήνυμα πληροφόρησης για ανοιχτό pipe. Αποστέλλεται από τη διεργασία CPP πριν πυροδοτηθεί η επαναφορά της θυγατρικης διεργασίας, ώστε να αντικατασταθούν οι περιγραφητές αρχείων <read_end> και <write_end> μέσω της λειτουργικότητας inherit_fd του CRIU από νέους περιγραφητές που δημιουργούνται επιτόπου από τη διεργασία RPD.

```
*** P|<pipe_name>|<read_end>|<write_end> ***

```

Καταχώρηση 3.6: Μήνυμα πληροφόρησης για ανοιχτή διοχέτευση (pipe)

- Type 4: Μήνυμα εγγραφής δεδομένων σε διοχέτευση από οποιονδήποτε κόμβο. Κάθε πλευρά διατηρεί την αντιστοίχηση των διοχετεύσεων, εστω κι αν η εγγραφή γίνεται εν τέλει σε άκρα με διαφορετικούς περιγραφητές.

```
*** PW|<pipe_id>|<pipe_data> ***

```

Καταχώρηση 3.7: Μήνυμα εγγραφής δεδομένων σε διοχέτευση (pipe)

3.4 Αποφυγή PID collision μέσω namespaces

Η βιβλιοθήκη CRIU, με στόχο τη διατήρηση της διαφάνειας κατά τις ενέργειες dump και restore, επαναφέρει τις αποθηκευμένες διεργασίες με ακριβώς το ίδιο αναγνωριστικό διεργασίας (Process Identifier - PID). Αυτό εισάγει ένα προφανές πρόβλημα, στην περίπτωση που στο σύστημα προορισμού υπάρχει διεργασία με ορισμένο το ίδιο PID. Το πρόβλημα αυτό ονομάζεται σύγκρουση αναγνωριστικών διεργασίας (PID collision). Αντιμετωπίζεται ορίζοντας έναν νέο ονοματοχώρο αναγνωριστικών διεργασίας (PID namespace), και εκκινώντας κάθε φορά τη διεργασία RPD εντός αυτού. Επιπλέον, το criu χρησιμοποιεί σε μεγάλο βαθμό το σύστημα αρχείων proc/, οπότε στον νέο ονοματοχώρο οφείλουμε να προσαρτήσουμε, ακριβώς στο ίδιο σημείο προσάρτησης (mountpoint), το σύστημα αρχείων proc/ του πατέρα. Τότε, η διεργασία RPD θα έχει πάντοτε το PID 1 και η θυγατρική διεργασία που επαναφέρεται θα έχει το ίδιο PID που είχε στο σύστημα προορισμού. Τα παραπάνω καθίστανται δυνατά με τη χρήση της κλήσης συστήματος clone, όπως φαίνεται στις Καταχωρήσεις 3.8 και 3.9.

```
static int              /* Start function for cloned child */
childFunc(void *arg)
{
    signal(SIGTTOU, SIG_IGN);
    if (setpgid(0, 0) == -1)
        error("setpgid");
    if (tcsetpgrp(STDIN_FILENO, getpgrp()) == -1)
```

```
8           error("tcsetpgrp-child");
9
10      char **argv = arg;
11      execve("client", arg, NULL);
12  }
13
14  int main(){
15  ....
16  childpid = clone(childFunc,
17                           child_stack + STACK_SIZE,
18                           CLONE_NEWNS | CLONE_NEWPID | SIGCHLD, exec_args);
19          if (childpid < 0)
20          {
21              error("CLONE on WDS");
22          }
23  ....
24  }
```

Καταχώρηση 3.8: Ορισμός νέου PID namespace κατά την εκκίνηση της διεργασίας RPD από τη WDS

```
1  ....
2  mount("none", "/proc", NULL, MS_SLAVE, NULL);
3  mount("proc", "/proc", "proc", 0, NULL);
4  ....
```

Καταχώρηση 3.9: Προσάρτηση proc filesystem για αποφυγή PID collision

Κατά την προσάρτηση του proc/ filesystem, οφείλουμε να δώσουμε ιδιαίτερη προσοχή στο mount propagation. Σύμφωνα με αυτό, αν γίνουν αλλαγές σε ένα προσαρτημένο μονοπάτι εντός ενός νέου mount namespace, αυτές διαδίδονται και στο αρχικό namespace. Στην περίπτωσή μας, αυτό σημαίνει πως το νέο namespace θα εξακολουθούσε να είναι εκτεθειμένο στο πρόβλημα του PID collision. Για αυτόν το λόγο, αρχικά καλούμε την κλήση συστήματος mount με τη σημαία MS_SLAVE. Όταν ένα mount point γίνεται slave, επιτρέπει τη διάδοση των αλλαγών από το κύριο group στο οποίο ήταν μέλος πρότερα, ενώ δεν διαδίδει το ίδιο τις αλλαγές του [**17**]. Στη συνέχεια προσαρτούμε ξανά με την κλήση συστήματος mount() το proc/ σύστημα αρχείων.

3.5 Διοχετεύσεις

Η υποστήριξη διαδιεργασιακής επικοινωνίας μέσω διοχετεύσεων(pipes) βασίζεται στη λειτουργικότητα inherit-fd που παρέχει το ίδιο το CRIU. Υπάρχουν σενάρια χρήσης του CRIU κατά τα οποία συγκεκριμένα αρχεία τα οποία ήταν ανοιχτά κατά τη χρονική στιγμή του dump δεν είναι διαθέσιμα κατά τη χρονική στιγμή του restore, όπως αρχεία καταγραφής της διεργασίας και γενικότερα συμβατικά αρχεία. Σε αυτή την περίπτωση, ο περιγραφητής δεν αντιστοιχεί στο επιθυμητό αρχείο και το σφάλμα εκτέλεσης της restore διεργασίας είναι επικείμενο. Για την επίλυση αυτού το προβλήματος, το CRIU παρέχει τη λειτουργία inherit-fd, ώστε η διεργασία που πυροδοτεί την επαναφορά να παρέχει έναν νέο περιγραφητή αρχείου, ώστε να αντικαταστήσει τον προηγούμενο μη προσβάσιμο. Η συγκεκριμένη λειτουργικότητα, αν και ιδιαίτερα χρήσιμη, πρέπει να εφαρμόζεται με φειδώ, αφού bugs και race conditions

είναι αναμενόμενα σε αρκετές εφαρμογές, όπως όταν η εκτέλεση εξαρτάται από την κατάσταση του αρχείου (π.χ. seek offset) [**18**].

Οι διοχετεύσεις, για το UNIX, είναι προσβάσιμες με διεπαφή αρχείου, οπότε χαρακτηρίζονται από περιγραφήτές αρχείων, όπως περιγράφηκε στην Ενότητα 2.4.1. Η λειτουργικότητα inherit-fd παρέχει τη δυνατότητα αντικατάστασης των δύο άκρων της διοχέτευσης από νέους περιγραφητές αρχείου προτού εκτελεστεί το restore. Δημιουργούμε επομένως ένα νέο pipe ανάμεσα στη διεργασία RPD και τη διεργασία παιδί, για κάθε pipe το οποίο κληρονομεί από τον πατέρα. Προϋποτίθεται συνεπώς η γνώση για τις κληρονομούμενες διοχετεύσεις κατά την επαναφορά της διεργασίας στο απομακρυσμένο σύστημα. Για αυτόν τον σκοπό, προφορτώνουμε τη συνάρτηση pipe() ώστε σε κάθε δημιουργία διοχέτευσης από τη γονική διεργασία να αποθηκεύουμε τους αντίστοιχους περιγραφητές αρχείων. Η διαδικασία αυτή αποτυπώνεται σε κώδικα στην Καταχώρηση 3.10.

```
/**
*   This is the Linked List that holds pipe file descriptors.
*   List elements are of type rfork_pipe_list as defined in rfork_lib/pipes.h
*/
rfork_pipe_list * rfork_pipe_list_head;

/**
*   Global variable that is used as an increment value for different pipes created
    using the pipe() call.
*   This is the way that we discern which pipe ends belong to each pipe in order to
    pass the pipe argument to the server executable.
*   pipe_id is incremented after each pipe system call.
*/
int pipe_id=0;

int pipe(int fds[2]) {
    //first get the original pipe sys call
    pid_t (*original_pipe)();
    original_pipe = dlsym(RTLD_NEXT, "pipe");

    int ret;
    ret = original_pipe(fds);
    pipe_id++;
    push(rfork_pipe_list_head, fds[0],0, pipe_id);
    push(rfork_pipe_list_head, fds[1],1, pipe_id);

    return ret;
}
```

Καταχώρηση 3.10: Προφορτωμένη pipe() για αποθήκευση κληρονομούμενων περιγραφητών διοχετεύσεων

Επιπροσθέτως, προφορτώνουμε και την κλήση συστήματος call, οπότε στη δομή rfork_pipe_list διατηρούμε μόνο τους ανοιχτούς περιγραφητές διοχετεύσεων, όπως αποτυπώνεται στην Καταχώρηση 3.11.

```
int close(int fd){
    pid_t (*original_close)();
    original_close = dlsym(RTLD_NEXT, "close");
    int ret;
    ret = original_close(fd);
    remove_by_value(&rfork_pipe_list_head, fd);

    return ret;
}
```

Καταχώρηση 3.11: Προφορτωμένη close() για ενημέρωση των διαθέσιμων περιγραφητών διοχετεύσεων

3.6 Σήματα

Αν θέλουμε να υποστηρίξουμε τη διαδιεργασιακή επικοινωνία με χρήση σημάτων, δεν έχουμε παρά να τα προωθήσουμε μέσω της ζεύξης TCP. Προκειμένου οι διαδικασίες CPP και RPD να προωθούν τα σήματα στη ζεύξη, αρχικοποιούμε ένα αρχείο τύπου signalfd. Τα signalfd αρχεία αποτελούν μια εναλλακτική στους παραδοσιακούς χειριστές σημάτων (signal handlers), ενώ, παρέχοντας τη διεπαφή αρχείου, επιτρέπουν τον χειρισμό των σημάτων με χρήση της epoll(κατ' επέκταση και των poll και select). Όποτε λάβει κάποιο σήμα μια διεργασία που επιβλέπει έναν signalfd μέσω της epoll, ειδοποιείται και διαβάζει απο τον περιγραφητή συγκεκριμένα bytes (μεγέθους sizeof(signalfd_siginfo)), οπότε και λαμβάνει όλες τις απαραίτητες πληροφορίες για το ληφθέν σήμα [**19**].

Αφού αποκτήσει η διεργασία επίγνωση για το ληφθέν σήμα, δημιουργεί ένα νέο μήνυμα για μετάδοση μέσω της TCP ζεύξης και στη συνέχεια το αποστέλλει.

Προς το παρόν, η παραπάνω μεθοδολογία δεν δύναται να εφαρμοστεί για δύο συγκεκριμένα σήματα, το SIGKILL και το SIGSTOP. Τα δύο αυτά σήματα δεν πιάνονται από κανέναν handler, οπότε και δεν συμπεριλαμβάνονται στη λίστα με τα σήματα που ειδοποιούν τον signalfd.

3.7 Μηχανισμός κοινής μνήμης μέσω δικτύου

Ένας από τους προαναφερθέντες μηχανισμούς διαδιεργασιακής επικοινωνίας, ο οποίος και υλοποιήθηκε στα πλαίσια αυτής της διπλωματικής, είναι ο μηχανισμός κοινής μνήμης. Συγκεκριμένα, υλοποιήθηκε, μέσω προφόρτωσης των χρησιμοποιούμενων συναρτήσεων, ο μηχανισμός τόσο για το πρότυπο System V [**16**] όσο και για το POSIX, με σκοπό τη διάφανη και απρόσκοπτη υποστήριξη όσο το δυνατό μεγαλύτερου αριθμού προγραμμάτων. Στη συνέχεια περιγράφεται τόσο η κοινή μεθοδολογία προσομοίωσης κοινής μνήμης μέσω δικτύου, όσο και οι επιμέρους διαφοροποιήσεις για τις δύο διεπαφές.

3.7.1 Κοινή μνήμη μέσω αρχείου στον διαμοιραζόμενο αποθηκευτικό χώρο

Ο μηχανισμός που προτείνουμε αποφεύγει το δύσκολο έργο της υλοποίησης memory disaggregation, ορίζοντας ως κοινή μνήμη ένα αρχείο αποθηκευμένο στο διαχειριζόμενο σύστημα αρχείων μεταξύ των κόμβων που αποτελούν πιθανούς προορισμούς μίας διεργασίας που κλωνοποιείται μέσω της fork(). Η ιδέα αυτή, αν και εισάγει στο πρόβλημα ζητήματα συγχρονισμού και αποδοτικότητας, έχει χρησιμοποιηθεί στο παρελθόν σε προγραμματιστικές διεπαφές και λειτουργικά συστήματα, όπως στο Windows API

[20]. Στα πλαίσια της παρούσας περιγραφής, θα αναφερόμαστε, για λόγους ευκολίας, στη διεργασία που δημιουργεί το τμήμα διαμοιραζόμενης μνήμης (shared memory segment) και εισάγει δεδομένα σε αυτό με την ονομασία Writer και στη διεργασία που προσπελαύνει αυτή την κοινή μνήμη ως Reader. Το κοινό αρχείο δημιουργείται από τον Writer, ο οποίος ορίζει την ονομασία, τα δικαιώματα πρόσβασης καθώς και το μέγεθος του αρχείου. Έπειτα, χαρτογραφεί στη μνήμη του το αρχείο (memory mapping), λαμβάνοντας μια αρχική διεύθυνση μνήμης, μέσω της οποίας δύναται πλέον να προσπελάσει το αρχείο χρησιμοποιώντας το υποσύστημα μνήμης. Αφού εκτελέσει τις επιθυμητές ενέργειες, ο Writer διαγράφει την αντιστοίχηση στον εικονικό χώρο διευθύνσεών του και κλείνει το αρχείο. Ο Reader με τη σειρά του, αποκτά πρόσβαση στον κοινό χώρο μνήμης, και χαρτογραφεί στον εικονικό χώρο διευθύνσεων το ίδιο αρχείο που δημιουργήθηκε από τον Writer, αποκτώντας πρόσβαση στα δεδομένα που επεξεργάστηκε ο τελευταίος.

3.7.2 Κοινή μνήμη κατά System V

Προκειμένου να υποστηρίξουμε με πλήρη διαφάνεια τη διεπαφή κοινής μνήμης που παρείχε το System V **[16]**, οφείλουμε να προφορτώσουμε τις ακόλουθες κλήσεις συστήματος:

- ftok
- shmget
- shmat
- shmdt
- shmctl

Για κάθε μια περιγράφουμε τόσο την υλοποίηση όσο και πιθανές παραδοχές:

3.7.2.1 ftok

Η συνάρτηση ftok() επιστρέφει ένα κλειδί το οποίο χρησιμεύει ως αναγνωριστικό για όλες τις διαδικασίες IPC του System V. Χρησιμοποιεί τη διεύθυνση ενός αρχείου σε ένα σύστημα αρχείων προκειμένου να εξάγει ένα μοναδικό τέτοιο κλειδί. Το κλειδί αυτό αποτελεί συνάρτηση δύο παραμέτρων, του αριθμού inode και του αριθμού συσκευής ενός αρχείου **[21]**. Τις δύο αυτές τιμές μπορούμε να τις λάβουμε εύκολα μέσω του Bash εργαλείου stat. Εκτελώντας το σε δύο διαφορετικούς κόμβους με όρισμα το ίδιο αρχείο, αποθηκευμένο στον κοινό αποθηκευτικό χώρο, διαπιστώνουμε πως, αν και ο αριθμός inode είναι ίσος (αφού είναι χαρακτηριστικό του συστήματος αρχείων το οποίο είναι κοινό), δεν ισχύει το ίδιο απαραίτητα για τον αριθμό συσκευής. Προκειμένου να προσομοιώσουμε την εκτέλεση της ftok στο ίδιο μηχάνημα, την προφορτώνουμε με μία εναλλακτική υλοποίηση η οποία αγνοεί τον αριθμό συσκευής και χρησιμοποιεί μια σταθερά, που την ορίζουμε ως χαρακτηριστική για την κάθε συστάδα κόμβων που εκτελούν την υλοποίησή μας.

```
1 #define CLUSTER_ID 7111917
2 key_t ftok (const char *pathname, int proj_id)
3 {
```

```
  struct stat64 st;
  key_t key;
  if (__xstat64 (_STAT_VER, pathname, &st) < 0)
    return (key_t) -1;
  key = ((st.st_ino & 0xffff) | ((CLUSTER_ID & 0xff) << 16)
         | ((proj_id & 0xff) << 24));
  return key;
}
```

Καταχώρηση 3.12: Υλοποίηση προφορτωμένης ftok()

3.7.2.2 shmget

Όπως αναφέρθηκε, η έννοια του τμήματος κοινής μνήμης δεν υφίσταται στη δικτυακή υλοποίησή μας, επομένως αποτελεί στόχο να αντικαταστήσουμε τα τμήματα αυτά με διαμοιραζόμενα αρχεία. Η shmget() επιστρέφει ένα αναγνωριστικό για το κάθε τμήμα, οπότε επιλέγουμε να επιστρέψουμε τον περιγραφητή του αρχείου που δημιουργούμε (ή ανοίγουμε). Επιπλέον, μέσω της ftruncate() ορίζουμε το μέγεθος αυτού του αρχείου σύμφωνα με την παράμετρο size της αρχικής κλήσης, ενώ εφαρμόζουμε και τις αντίστοιχες παραμέτρους της shmflg κατά τη δημιουργία του αρχείου.

```
size_t global_size;
int shmget(key_t key, size_t size, int shmflg){
    int shmid;

    //stub for now
    global_size = size;
    char key_str[100] = {'\0'};
    sprintf(key_str,"%d",key);

    int open_flag = 0;
    //construct flag from shmflg

    // IPC_CREAT
    if (shmflg & IPC_CREAT){
        open_flag = open_flag | O_CREAT;
    }

    if (shmflg & IPC_EXCL){
        open_flag = open_flag | O_EXCL;
    }

    if (shmflg & SHM_NORESERVE){
        open_flag = open_flag | MAP_NORESERVE;
    }

    // construct mode
    //get octal representation of shmflg
    int open_mode = shmflg & 0777;

    //permissions in octal transfered to open third argument
```

```
31
32      shmid = open (key_str, O_RDWR | open_flag, open_mode);
33      ftruncate(shmid,size);
34      lseek(shmid, 0, SEEK_SET);
35
36      return shmid;
37  }
```

Καταχώρηση 3.13: Υλοποίηση προφορτωμένης shmget()

Σημειώνουμε πως η παραμετροποιημένη, για τους σκοπούς της υλοποίησης, shmget() δεν υποστηρίζει ακόμη Huge Pages **[22]** οπότε τα αντίστοιχα bits της παραμέτρου shmflg αγνοούνται. Επιπλέον, δε δημιουργείται η συσχετιζόμενη δομή κοινής μνήμης shmid_ds **[23]**, γεγονός που περιορίζει τα σενάρια χρήσης σε εφαρμογές που δεν χρησιμοποιούν αυτή την πληροφορία.

3.7.2.3 shmat

Μέχρι τώρα έχουμε δημιουργήσει (ή ανοίξει) ένα αρχείο το οποίο αντιμετωπίζεται ως τμήμα κοινής μνήμης. Προκειμένου να το χαρτογραφήσουμε (map) στον εικονικό χώρο διευθύνσεων της καλούσας διεργασίας χρησιμοποιούμε την shmat(). Χρησιμοποιώντας το αναγνωριστικό που επιστρέφεται από την shmget() αντιστοιχούμε το ανοιχτό αρχείο με τον χώρο διευθύνσεων χρησιμοποιώντας την mmap(). Προς το παρόν υποστηρίζουμε κλήσεις της mmap() οι οποίες δεν ορίζουν συγκεκριμένα την αρχική διεύθυνση που επιλέγεται για τη χαρτογράφηση. Διατηρώντας την αρχική διεπαφή, επιστρέφουμε τη διεύθυνση στην οποία συνδέεται το αρχείο κοινής μνήμης.

```
1  void * shmat(int shmid, const void * shmaddr, int shmflg){
2      void* file_addr;
3      int mmap_prot;
4
5      if (shmflg & SHM_RDONLY){
6          mmap_prot = mmap_prot | PROT_WRITE;
7      }
8      else{
9          mmap_prot = mmap_prot | PROT_WRITE | PROT_READ;
10     }
11
12     file_addr = mmap(0, global_size, mmap_prot, MAP_SHARED, shmid, 0);
13     return file_addr;
14 }
```

Καταχώρηση 3.14: Υλοποίηση προφορτωμένης shmat()

Σημειώνεται πως προς το παρόν δεν υποστηρίζεται η λειτουργικότητα που παρέχεται από την παράμετρο SHM_REMAP.

3.7.2.4 shmdt

```
1  int shmdt(const void * shmaddr){
2          munmap((void *)shmaddr, global_size);
3          return 0; //check munmap for errors later
```

```
}
```

Καταχώρηση 3.15: Υλοποίηση προφορτωμένης shmdt()

3.7.2.5 shmctl

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf){
    int (*original_shmctl)();
    original_shmctl = dlsym(RTLD_NEXT, "shmctl");

    if (cmd == IPC_RMID)
    {
        //get path of shared memory mapped file to be removed
        ssize_t bytes_written;
        char buf_full_path[200] = {'\0'};
        char pathname[100] = {'\0'};
        sprintf(pathname,"/proc/self/fd/%d", shmid);

        bytes_written = readlink(pathname, buf_full_path, sizeof(buf));
        if (bytes_written<0){
            //should also set errno in future versions
            return -1;
        }
        int unlink_ret;
        unlink_ret = unlink(buf_full_path);
        return unlink_ret;
    }
    else{
        //dummy operation for trasnparency in local usage
        return original_shmctl(shmid,cmd,buf);
    }
}
```

Καταχώρηση 3.16: Υλοποίηση προφορτωμένης shmctl()

3.7.3 Κοινή μνήμη κατά POSIX

Σχετικά με το POSIX, η υλοποίηση είναι ακόμη πιο ευθεία, δεδομένου ότι ούτως η άλλως το πρότυπο ακολουθεί τις αρχές διαχείρισης κοινού χαρτογραφημένου στη μνήμη αρχείου για την παροχή αυτής της λειτουργικότητας. Συγκεκριμένα, η κοινή μνήμη κατά POSIX υλοποιείται μέσω αρχείων χαρτογραφημένων στη μνήμη τα οποία αποθηκεύονται σε ένα ειδικό σύστημα αρχείων που βρίσκεται στο /dev/shm και είναι τύπου tmpfs, ώστε να παρέχεται η μέγιστη απόδοση, αφού βρίσκεται εξ' ολοκλήρου στη μνήμη και όχι σε κάποιο αποθηκευτικό μέσο. Εμείς, λόγω της απαίτησης για απομακρυσμένη πρόσβαση θα υλοποιήσουμε την ίδια λειτουργικότητα μέσω αρχείου αποθηκευμένου σε κοινό, δικτυακό αποθηκευτικό χώρο, χάνοντας αυτό το κέρδος απόδοσης που προσφέρει το tmpfs. Καλούμαστε να προφορτώσουμε τις ακόλουθες συναρτήσεις:

- shm_open

- shm_unlink

Για κάθε μια περιγράφουμε τόσο την υλοποίηση όσο και πιθανές παραδοχές:

3.7.3.1 shm_open

```
int shm_open(const char *name, int oflag, mode_t mode){
    int shmid;
    shmid = open (name, oflag, mode);
    return shmid;
}
```

Καταχώρηση 3.17: Υλοποίηση προφορτωμένης shm_open()

3.7.3.2 shm_unlink

```
int shm_unlink(const char *name){
    int unlink_ret;
    unlink_ret = unlink(name);
    return unlink_ret;
}
```

Καταχώρηση 3.18: Υλοποίηση προφορτωμένης shm_unlink()

Κεφάλαιο 4

Αξιολόγηση

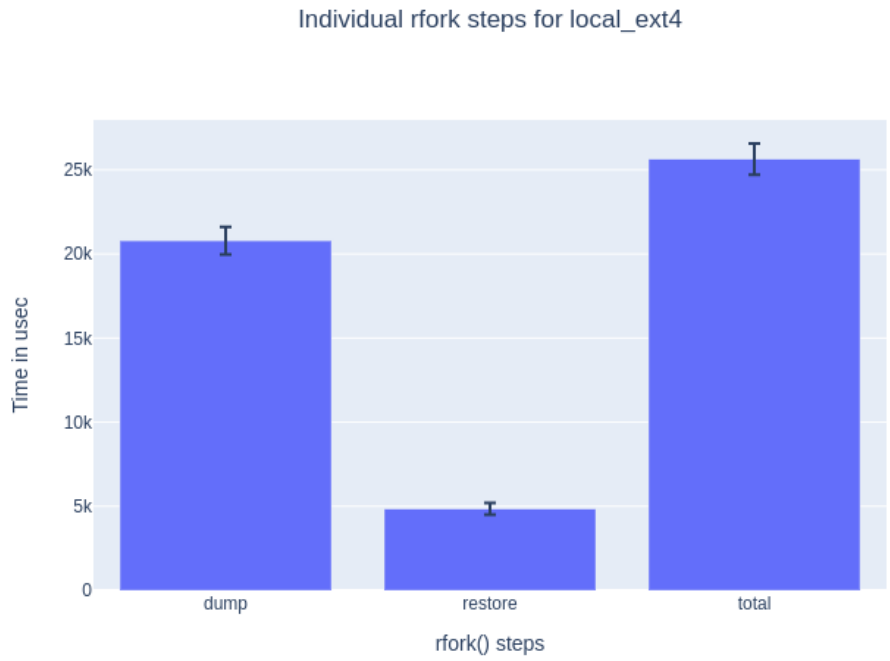
Σε αυτήν την ενότητα, αξιολογούμε την υλοποίηση μεταφοράς και απομακρυσμένης εκτέλεσης διεργασιών, λαμβάνοντας υπόψιν τρεις παράγοντες, το σύστημα αρχείων (file-system) επί του οποίου λειτουργεί το πρωτόκολλο κατανεμημένου συστήματος αρχείων NFS ανάμεσα στους δύο κόμβους, την επιλογή κόμβου φιλοξενίας (NFS Server) αυτού του διαμοιραζόμενου συστήματος αρχείων, καθώς και τη μέθοδο διασύνδεσης των δύο κόμβων. Συγκεκριμένα, ελέγχουμε την απόδοση κάθε σταδίου της μεταφοράς για συστήματα αρχείων tmpfs και ext4, με τον NFS server να εγκαθίσταται είτε στον κόμβο πηγή, είτε στον κόμβο προορισμό της διεργασίας, με διασύνδεση είτε Gigabit Ethernet είτε InfiniBand.

4.1 Σύστημα αρχείων

4.1.1 Τοπική εκτέλεση (χωρίς διαμοιραζόμενο σύστημα αρχείων)

Αρχικά ελέγχουμε την επίδοση του συστήματος που αναπτύσσουμε για εκτέλεση σε έναν host, αφαιρώντας τον παράγοντα του διαμοιραζόμενου συστήματος αρχείων (NFS). Προς αυτήν την κατεύθυνση, εκτελούμε την απομακρυσμένη κλήση συστήματος fork για δύο τοπικά συστήματα αρχείων, το ext4 και το tmpfs.

4.1.1.1 ext4 σύστημα αρχείων

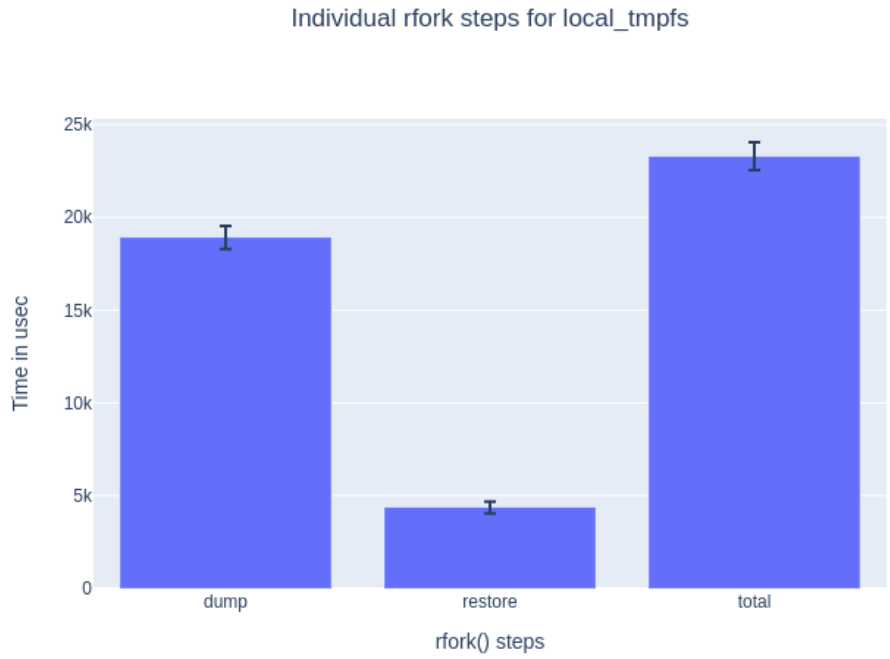


Σχήμα 4.1: Χρόνος εκτέλεσης για τα επιμέρους βήματα της rfork() σε Local/ext4

Παρατηρούμε πως για αυτές τις συνθήκες εκτέλεσης, το στάδιο αποθήκευσης της κατάστασης της διεργασίας παιδί (dump) είναι σχετικά χρονοβόρο, συγκριτικά με τη διαδικασία επαναφοράς της (restore), γεγονός που υποδεικνύει το μεγαλύτερο πλήθος αιτημάτων εγγραφής/ανάγνωσης του σταδίου αποθήκευσης της κατάστασης.

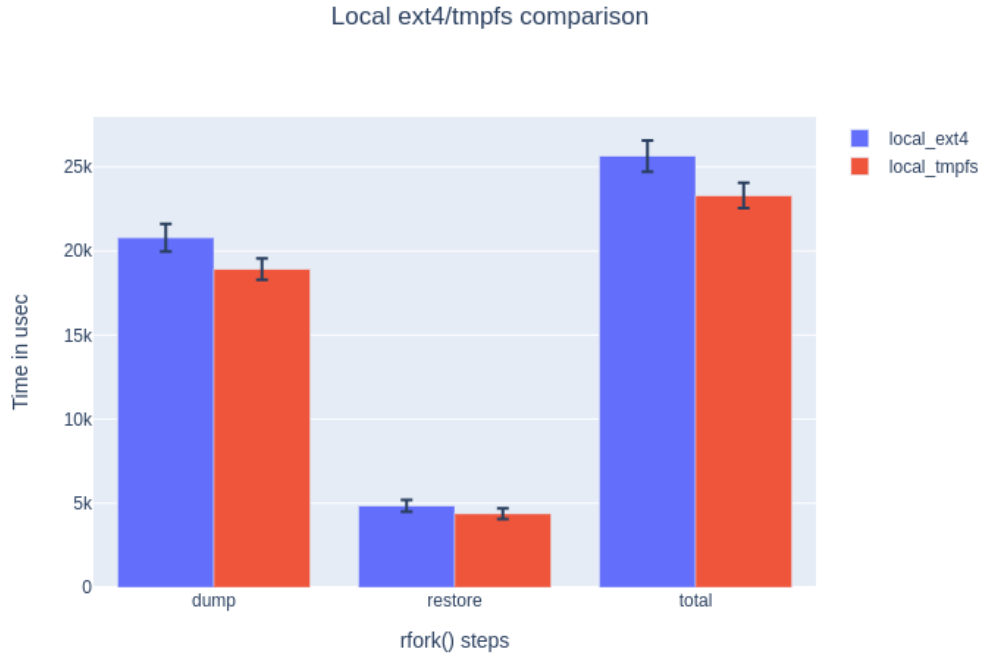
4.1.1.2 tmpfs σύστημα αρχείων

Στη συνέχεια, προσαρτούμε ένα σημείο του συστήματος αρχείων ως tmpfs. Ένα σύστημα αρχείων της μορφής tmpfs, παρέχει την ίδια διεπαφή με οποιοδήποτε προσαρτημένο σύστημα αρχείων, όμως αντί να τοποθετείται σε ένα μόνιμο σύστημα αρχείων, τοποθετείται στη μνήμη. Αυτή η μεθοδολογία εισάγει μεγαλύτερες ταχύτητες ανάγνωσης και εγγραφής.



Σχήμα 4.2: Χρόνος εκτέλεσης για τα επιμέρους βήματα της rfork() σε Local/tmpfs

Η σχέση μεταξύ των σταδίων dump και restore παραμένει ίδια και στο tmpfs σύστημα αρχείων, το οποίο, όπως διαφαίνεται στο Σχήμα 4.3, είναι αποδοτικότερο ως προς τον χρόνο εκτέλεσης.

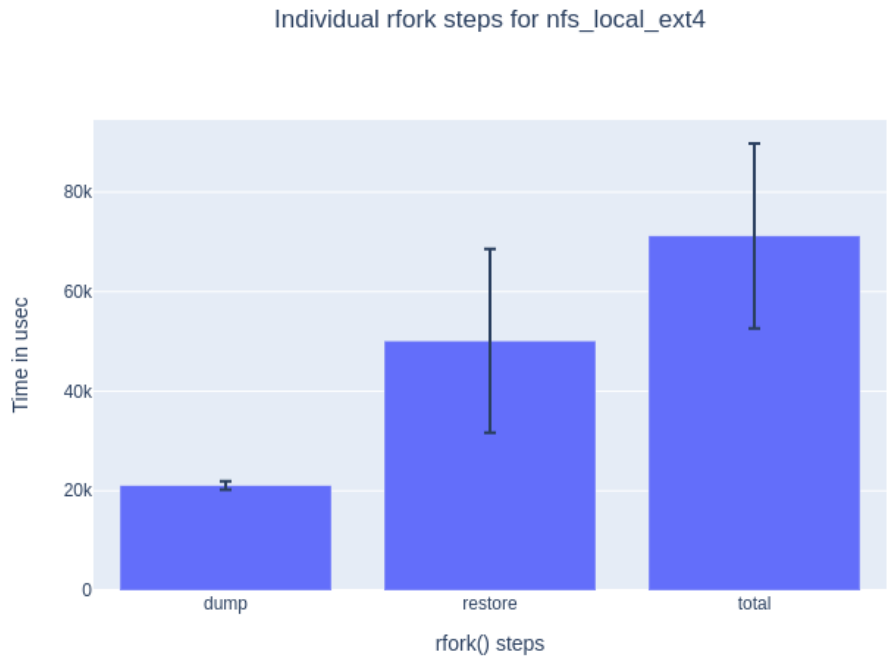


Σχήμα 4.3: Σύγκριση χρόνου εκτέλεσης σε local για ext4-tmpfs

4.1.2 Τοπικό σύστημα διαμοιραζόμενων αρχείων (NFS)

Σε αυτήν την υποενότητα, εισάγουμε τον παράγοντα του διαμοιραζόμενου συστήματος αρχείων NFS, εκτελώντας όμως την υλοποίηση στον ίδιο host, προκειμένου να μελετήσουμε την επίδραση του NFS στην υλοποίηση ανεξάρτητα από το δίκτυο. Προς την ίδια κατεύθυνση με τα προηγούμενα, εκτελούμε την υλοποίηση για NFS πάνω από ext4 και tmpfs σύστημα αρχείων. Επιπλέον, εγκαθιστούμε το διαμοιραζόμενο directory με τέτοια μεθοδολογία ώστε να χρησιμοποιείται το nfs μόνο κατά την επαναφορά της διεργασίας, οπότε ως NFS server λογίζεται το directory της διεργασίας πατέρα. Στην υποενότητα 4.2 μελετάται η επίδραση της θέσης του NFS directory στην απόδοση του συνολικού συστήματος.

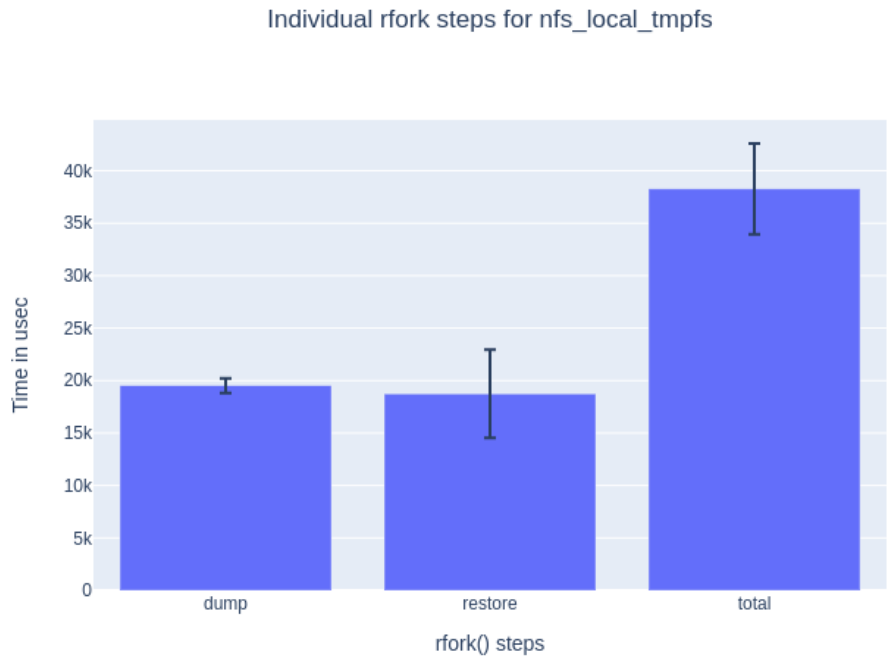
4.1.2.1 ext4 σύστημα αρχείων



Σχήμα 4.4: Χρόνος εκτέλεσης για τα επιμέρους βήματα της rfork() σε Local NFS/ext4

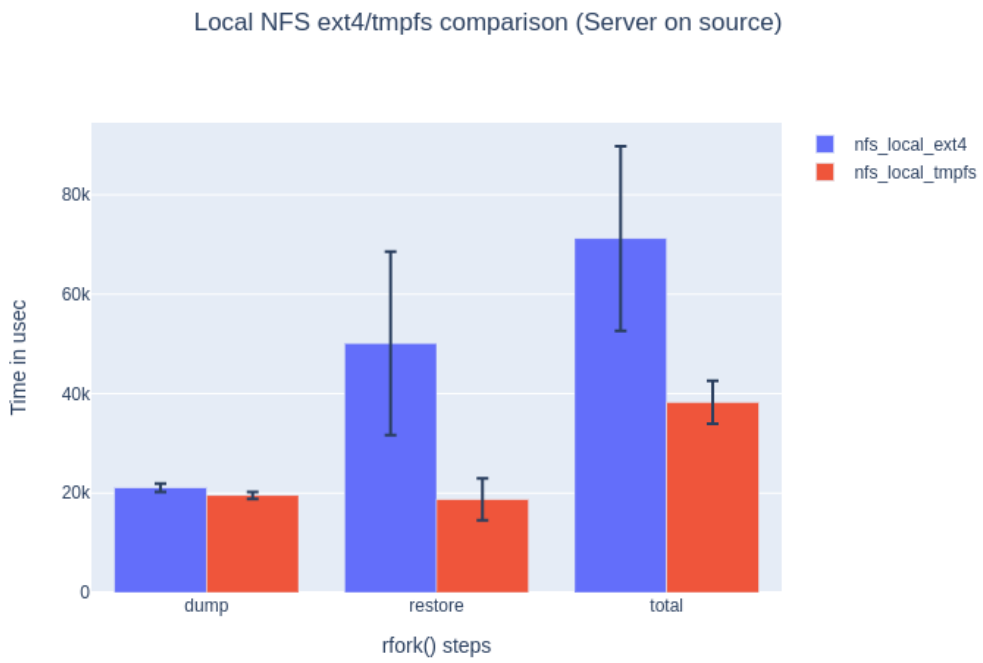
Στη δοκιμή αυτή ορίζουμε ως NFS server το directory από το οποίο πηγάζει η διεργασία παιδί, ενώ ως NFS client το directory προορισμού της διεργασίας αυτής. Παρατηρούμε επομένως παρόμοιες επιδόσεις με την απλή ext4 περίπτωση για τη διαδικασία αποθήκευσης της διεργασίας. Παρατηρούμε επίσης μια ιδιαίτερη αύξηση στον χρόνο εκτέλεσης της διαδικασίας επαναφοράς της διεργασίας, αφού αυτή απαιτεί ένα σύνολο RPC κλήσεων από τον NFS client στον NFS server καθώς και την εξυπηρέτηση αυτών από τον δεύτερο.

4.1.2.2 tmpfs σύστημα αρχείων



Σχήμα 4.5: Χρόνος εκτέλεσης για τα επιμέρους βήματα της rfork() σε Local NFS/tmpfs

Παρατηρούμε πως η εκτέλεση για Local NFS με tmpfs είναι αποδοτικότερη της αντίστοιχης για ext4 (Σχήμα 4.6). Το tmpfs σύστημα αρχείων παρέχει αποδοτικότερους χρόνους εγγραφής ανάγνωσης, οπότε κατά τις κλήσεις RPC για λήψη των αρχείων, αυτά εγγράφονται ταχύτερα στο τοπικό directory, συνεπώς το στάδιο restore για τη δεύτερη περίπτωση είναι εξαιρετικά αποδοτικότερο.



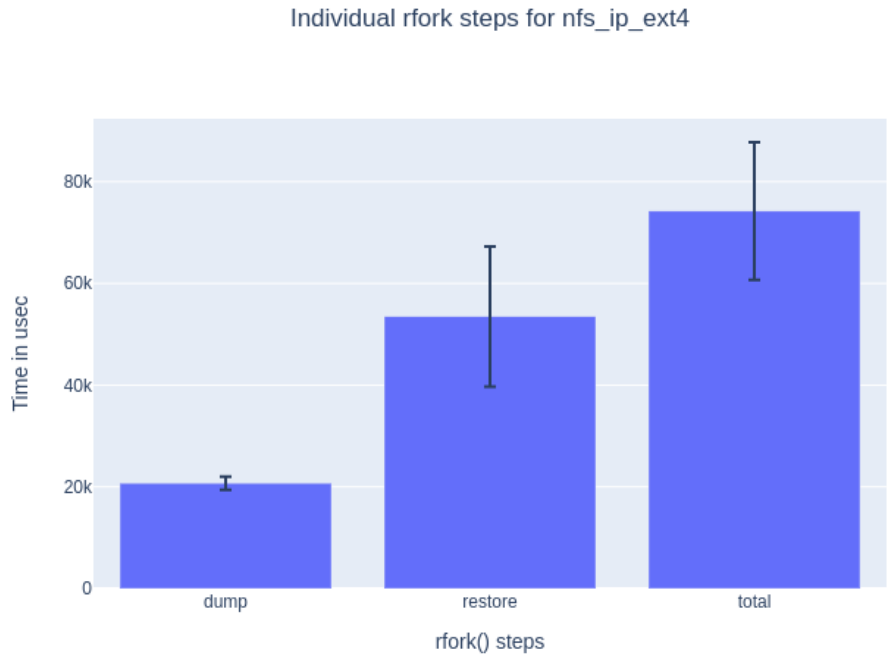
Σχήμα 4.6: Σύγκριση χρόνου εκτέλεσης σε NFS local για ext4-tmpfs (Server on source)

4.1.3 Διαμοιραζόμενο σύστημα αρχείων (NFS) με Gigabit Ethernet

Στην υποενότητα αυτή, εισάγουμε τον παράγοντα του δικτύου στο σύστημα αρχείων NFS. Ακόμη μια φορά, εκτελούμε την υλοποίηση για NFS πάνω από ext4 και tmpfs σύστημα αρχείων. Διατηρούμε την ίδια τοπολογία όπως προηγουμένως, με τον NFS server να βρίσκεται στην πλευρά του κόμβου αποστολέα, ενώ χρησιμοποιούμε Gigabit Ethernet για τη διασύνδεση των κόμβων.

4.1.3.1 ext4 σύστημα αρχείων

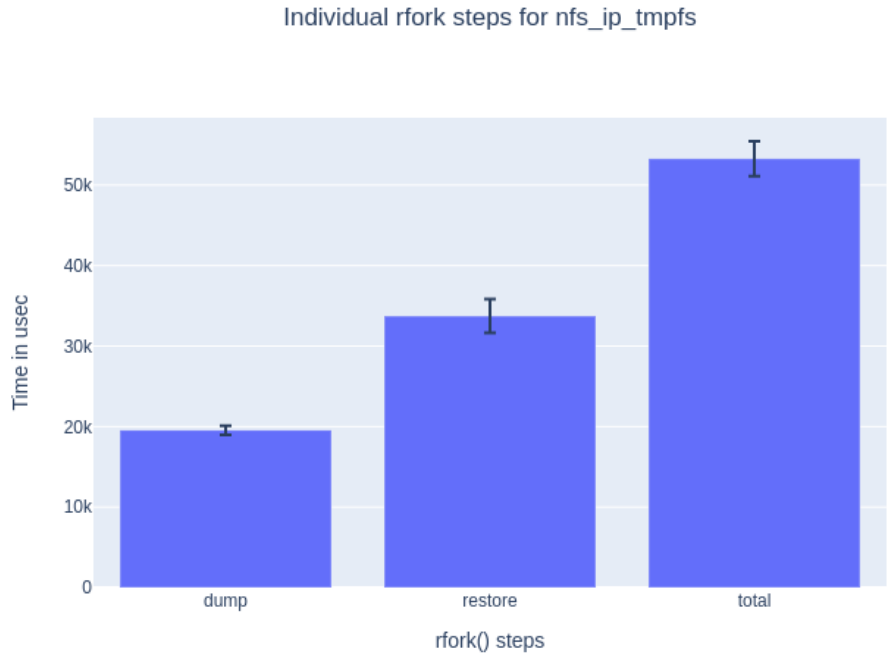
Αρχικά, ελέγχουμε την απόδοση της υλοποίησης για ext4 σύστημα αρχείων.



Σχήμα 4.7: Χρόνος εκτέλεσης για τα επιμέρους βήματα της rfork() σε NFS IP ext4

4.1.3.2 tmpfs σύστημα αρχείων

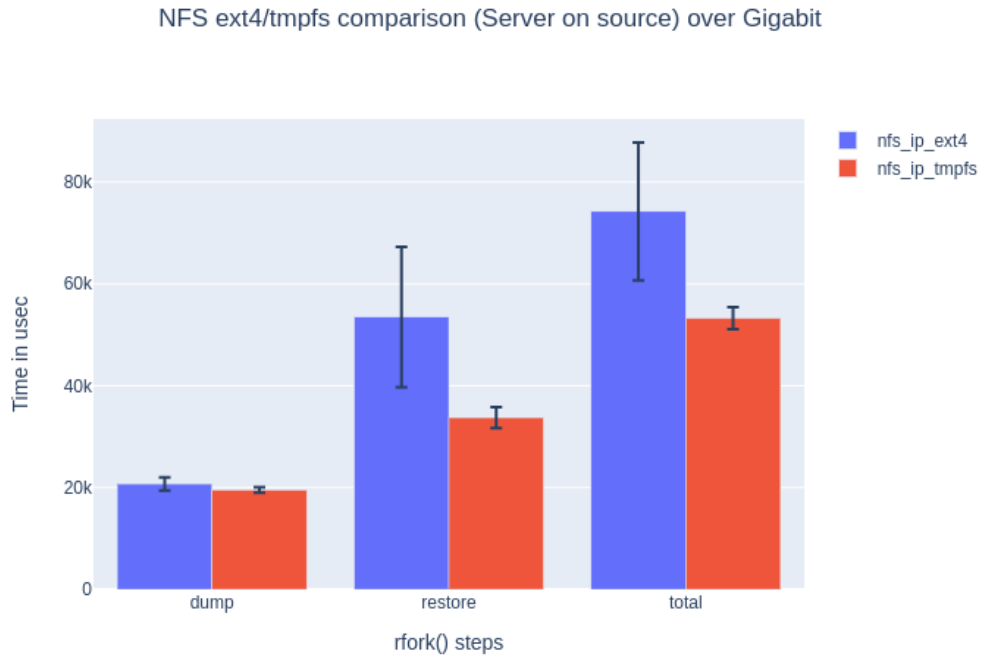
Στη συνέχεια, διατηρούμε την ίδια τοπολογία και αλλάζουμε το σύστημα αρχείων σε tmpfs.



Σχήμα 4.8: Χρόνος εκτέλεσης για τα επιμέρους βήματα της rfork() σε NFS IP tmpfs

4.1.4 Συμπέρασμα

Παρατηρούμε πως, για μια ακόμη φορά και όπως ήταν αναμενόμενο, το tmpfs σύστημα αρχείων μειώνει αισθητά τον χρόνο εκτέλεσης της μεταφοράς και επαναφοράς της διεργασίας, όπως φαίνεται και στο Σχήμα 4.9. Επιπλέον, παρατηρούμε τη μείωση της τυπικής απόκλισης σε κάθε στάδιο της διαδικασίας μεταφοράς και επαναφοράς, ως αποτέλεσμα της χρήσης tmpfs συστήματος αρχείων.



Σχήμα 4.9: Σύγκριση χρόνου εκτέλεσης σε NFS για ext4-tmpfs (Server on source) με Gigabit Ethernet

Το σύστημα αρχείων, όπως παρουσιάστηκε προηγουμένως, διαδραματίζει καίριο ρόλο στον χρόνο απόδοσης του συστήματος μεταφοράς και απομακρυσμένης εκτέλεσης που υλοποιείται στην παρούσα διπλωματική. Η εισαγωγή του διαμοιραζόμενου συστήματος αρχείων (NFS) αυξάνει σημαντικά τον χρόνο εκτέλεσης, γεγονός αναπόφευκτο για την ολοκληρωμένη λειτουργικότητα της υλοποίησής μας. Επιπλέον, το σύστημα αρχείων το οποίο αποτελεί βάση για το NFS διαδραματίζει εξίσου σημαντικό ρόλο, με το tmpfs, λόγω του ότι χρησιμοποιεί τη μνήμη για αποθήκευση δεδομένων, να εμφανίζει μικρότερους χρόνους εκτέλεσης συγκριτικά με το ext4.

4.2 Κόμβος φιλοξενίας του διαμοιραζόμενου συστήματος αρχείων (NFS)

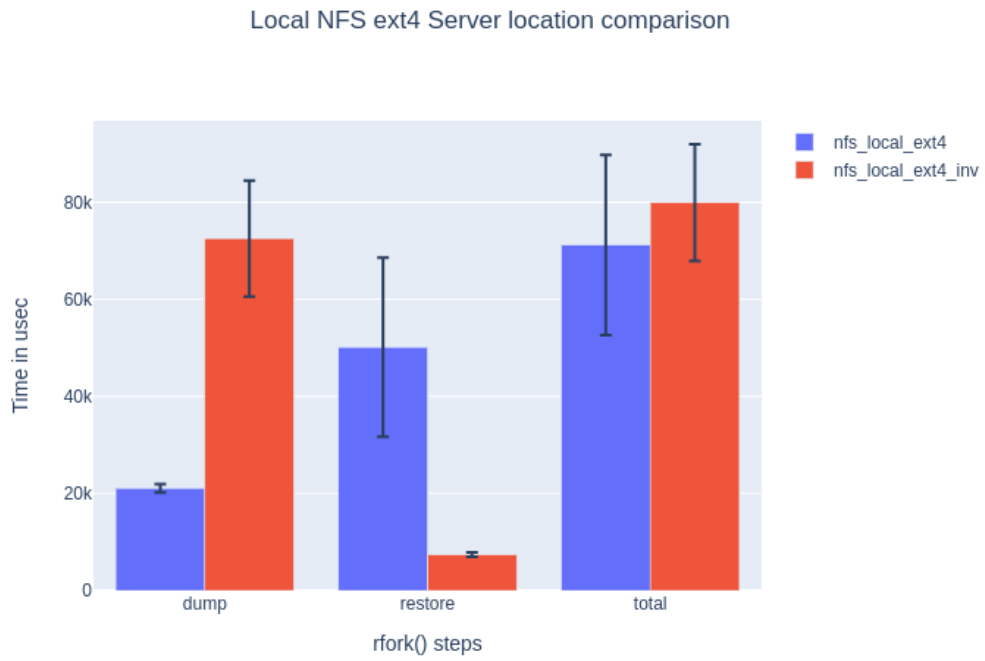
Σε αυτή την ενότητα, μελετάμε την επίδραση της θέσης του NFS στον χρόνο εκτέλεσης του συστήματος μεταφοράς και απομακρυσμένης εκτέλεσης διεργασιών. Αν αυτό φιλοξενείται στον κόμβο πηγής της διεργασίας, τότε η αποθήκευση της κατάστασης της διεργασίας (dump) γίνεται τοπικά, χωρίς να εμπλέκεται το NFS, ενώ η διαδικασία επαναφοράς της διεργασίας (restore) εκτελείται απομακρυσμένα και κάθε αίτημα για ανάγνωση εξυπηρετείται από το NFS. Αν αυτό φιλοξενείται στον κόμβο

προορισμού, η αποθήκευση της διεργασίας γίνεται απομακρυσμένα ενώ η επαναφορά της τοπικά.

4.2.1 Τοπικό διαμοιραζόμενο σύστημα αρχείων (NFS)

Στους ελέγχους για το τοπικό NFS, εμπλέκεται μόνο ένας κόμβος οπότε ασχολούμαστε με το directory αποθήκευσης και επαναφοράς της διεργασίας, ώστε να χρησιμοποιείται το NFS. Για λόγους πληρότητας, η σύγκριση περιλαμβάνει αποτελέσματα τόσο για το ext4, όσο και για το tmpfs σύστημα αρχείων.

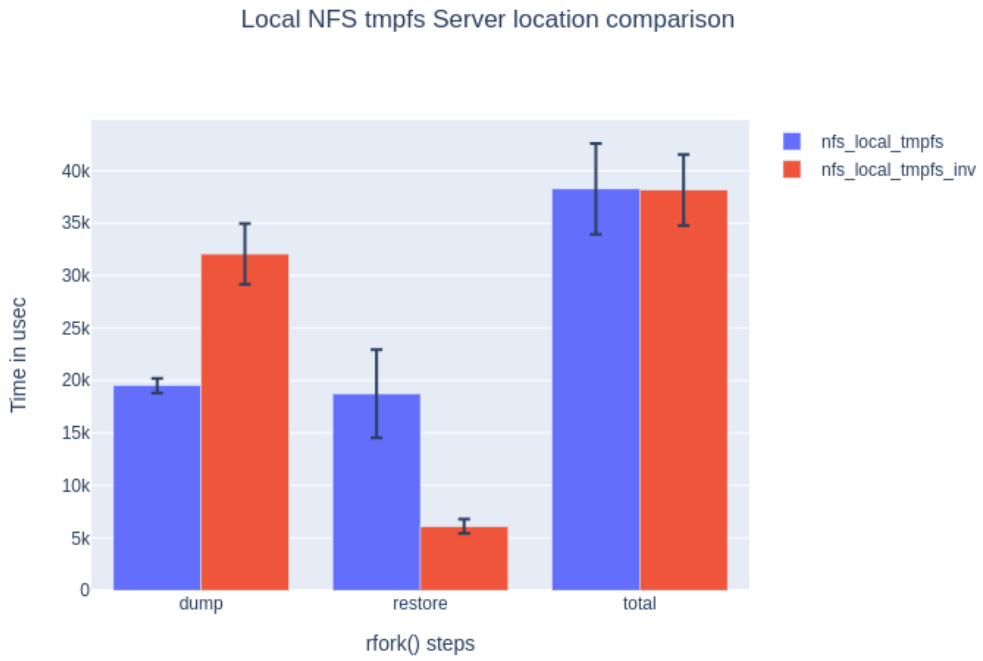
4.2.1.1 ext4 Σύστημα αρχείων



Σχήμα 4.10: Σύγκριση χρόνου εκτέλεσης σε Local NFS/ext4 με NFS server στον κόμβο πηγής και στον κόμβο φιλοξενίας

Όπως παρατηρούμε στο άνωθεν διάγραμμα, για το σύστημα αρχείων ext4, σε τοπικό NFS(το οποίο δεν εμπλέκει τον παράγοντα της ταχύτητας δικτύου) η φιλοξενία του NFS Server στον κόμβο πηγής είναι αποδοτικότερη, με το στάδιο αποθήκευσης της διεργασίας να εκτελείται αποδοτικότερα, λόγω τοπικής εγγραφής, ενώ το στάδιο επαναφοράς να εκτελείται με καθυστερήσεις, οι οποίες όμως δεν επηρεάζουν το τελικό αποτέλεσμα.

4.2.1.2 tmpfs Σύστημα αρχείων



Σχήμα 4.11: Σύγκριση χρόνου εκτέλεσης σε Local NFS/tmpfs με NFS server στον κόμβο πηγής και στον κόμβο φιλοξενίας

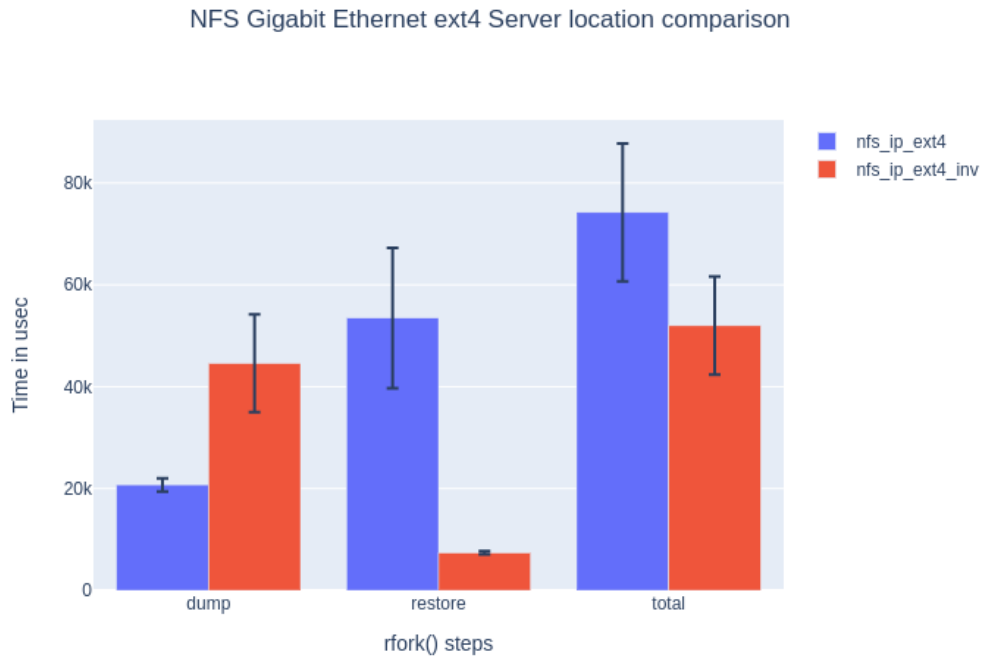
Αναμένουμε το tmpfs σύστημα αρχείων να βελτιστοποιήσει τον χρόνο εκτέλεσης των διαδικασιών αποθήκευσης και επαναφοράς, γεγονός που επιβεβαιώνεται απο τις μετρήσεις μας, ακριβώς όπως προηγουμένως, στην υποενότητα 4.1.2. Η διαφορά στην εκτέλεση μεταξύ των δύο περιπτώσεων φιλοξενίας NFS Server είναι αμελητέα πλέον, ενώ παρατηρούμε σχεδόν υποδιπλασιασμό του χρόνου εκτέλεσης για tmpfs σε σχέση με το ext4.

Καθίσταται σαφές το ότι η διαδικασία αποθήκευσης της διεργασίας απαιτεί περισσότερα RPC calls στον NFS Server, γεγονός που οφείλεται στο προηγούμενο πόρισμα της υποενότητας 4.1.1.1, το οποίο αναφέρει πως η διαδικασία αποθήκευσης απαιτεί περισσότερες κινήσεις εγγραφής/αποθήκευσης σε σχέση με τη διαδικασία επαναφοράς.

4.2.2 NFS μέσω Gigabit Ethernet

Στη συνέχεια, εισάγουμε τον παράγοντα του δικτύου και ελέγχουμε ξανά την απόδοση του υλοποιημένου συστήματος για φιλοξενία του NFS Server, τόσο στον κόμβο πηγή όσο και στον κόμβο προορισμού. Ακόμη μία φορά, η σύγκριση περιλαμβάνει αποτελέσματα τόσο για το ext4, όσο και για το tmpfs σύστημα αρχείων.

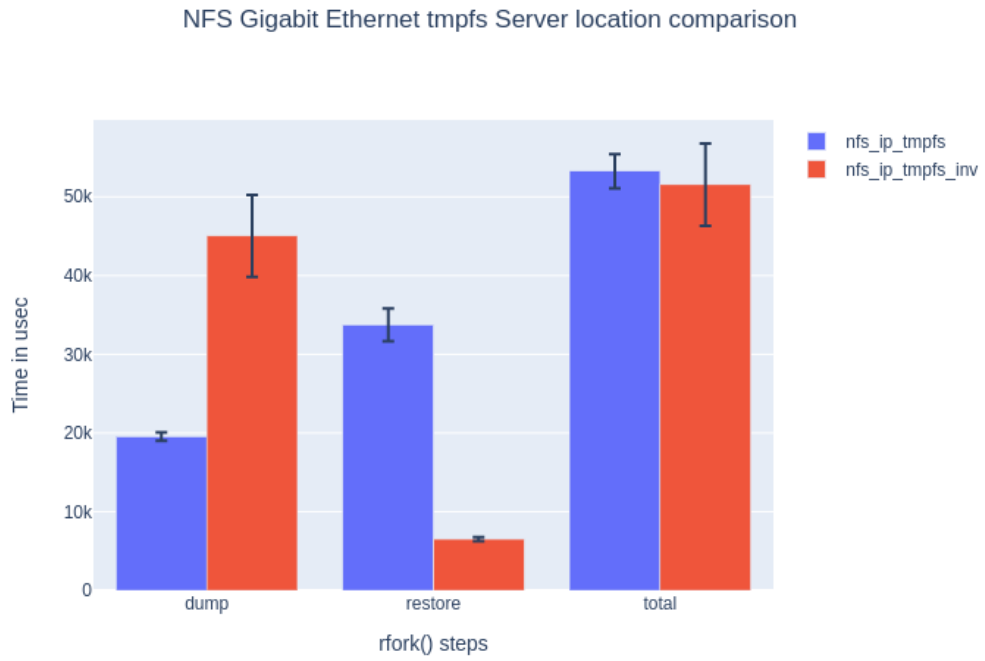
4.2.2.1 ext4 Σύστημα αρχείων



Σχήμα 4.12: Σύγκριση χρόνου εκτέλεσης σε NFS/ext4 Gigabit Ethernet με NFS server στον κόμβο πηγής και στον κόμβο φιλοξενίας

Παρατηρούμε πως την υλοποίηση είναι αξιοσημείωτα ταχύτερη για εκτέλεση με τον NFS Server στον κόμβο προορισμού, γεγονός που αποδίδεται στο γεγονός πως η διαδικασία του dump απαιτεί σημαντικά περισσότερες εγγραφές/αναγνώσεις από εκείνη του restore, οπότε αυτές εκτελούνται ταχύτερα αν δεν εμπλέκεται το NFS.

4.2.2.2 tmpfs Σύστημα αρχείων



Σχήμα 4.13: Σύγκριση χρόνου εκτέλεσης σε NFS/tmpfs Gigabit Ethernet με NFS server στον κόμβο πηγής και στον κόμβο φιλοξενίας

Στο συγκεκριμένο σενάριο εκτέλεσης, όπως προηγουμένως, η εγκατάσταση του NFS Server στον κόμβο προορισμού καθιστά, με μικρότερη διαφορά, ταχύτερη την υλοποίηση.

4.2.3 Συμπέρασμα

Απο τα παραπάνω, εξάγουμε το συμπέρασμα πως η επιλογή κόμβου εγκατάστασης του NFS Server η οποία καθιστά βέλτιστη την υλοποίηση είναι αυτή του κόμβου προορισμού.

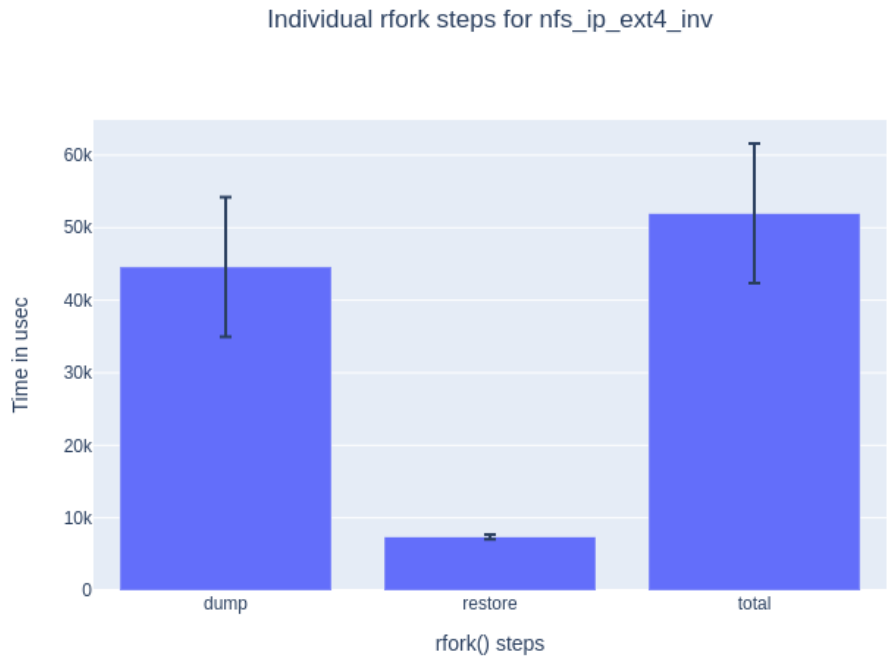
4.3 Διασύνδεση κόμβων

Ο τελευταίος παράγοντας που θα μας απασχολήσει κατά την αξιολόγηση της απόδοσης της υλοποίησης είναι αυτός της τεχνολογίας διασύνδεσης των κόμβων του συστήματος. Συγκεκριμένα, πρόκειται να μελετήσουμε την επίδραση της χρήσης του Gigabit Ethernet και του InfiniBand στον χρόνο εκτέλεσης της τροποποιημένη fork(). Επιλέγουμε να παρουσιάσουμε μετρήσεις μόνο για το σενάριο φιλοξενίας του διακομιστή NFS στον κόμβο προορισμού, αφού αυτό επιλέχθηκε ως αποδοτικότερο τόσο για το ext4 όσο και για το tmpfs σύστημα αρχείων.

4.3.1 Gigabit Ethernet

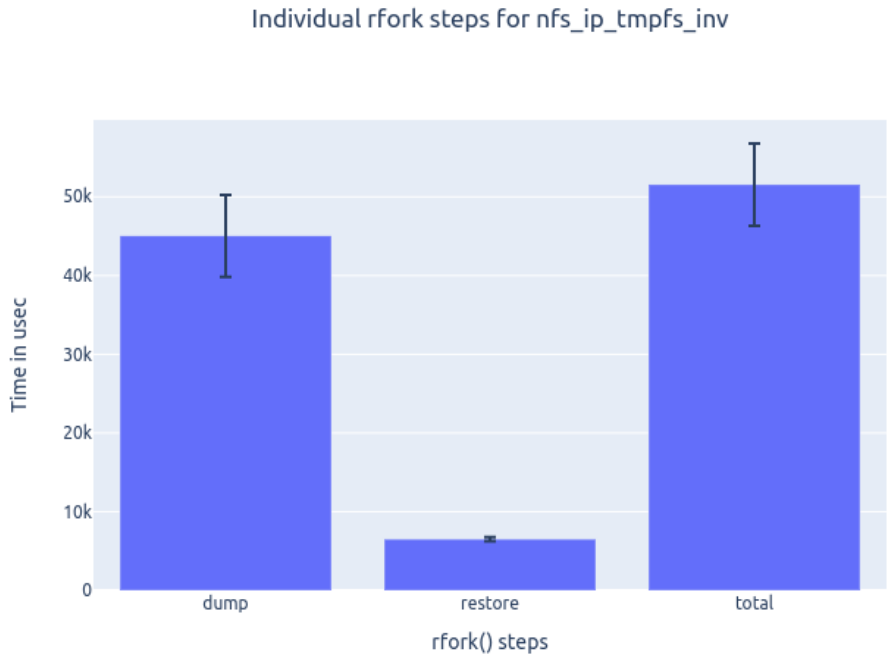
Αρχικά, διεξάγουμε μετρήσεις της απόδοσης υλοποίησης για διασύνδεση Gigabit Ethernet, δοκιμάζοντας δύο συστήματα αρχείων, το ext4 και το tmpfs.

4.3.1.1 ext4 Σύστημα αρχείων



Σχήμα 4.14: Χρόνος εκτέλεσης των επιμέρους βημάτων της rfork() για NFS ext4 μέσω Gigabit Ethernet

4.3.1.2 tmpfs Σύστημα αρχείων



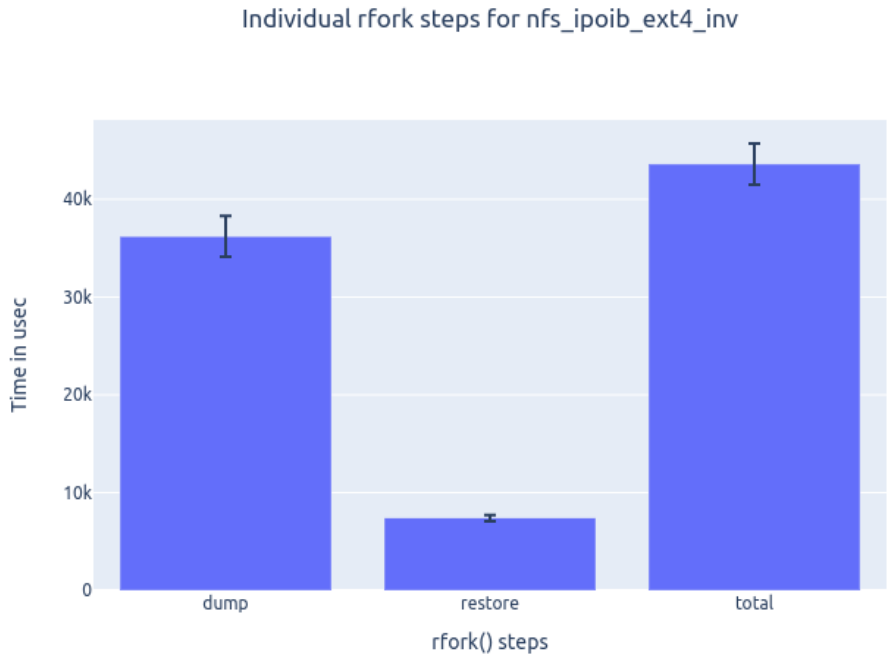
Σχήμα 4.15: Χρόνος εκτέλεσης των επιμέρους βημάτων της rfork() για NFS tmpfs μέσω Gigabit Ethernet

Παρατηρούμε την ελάχιστα ταχύτερη φύση του tmpfs σχετικά με την υλοποίησή μας και τη μειωμένη τυπική απόκλιση του χρόνου εκτέλεσης κάθε σταδίου της μεταφοράς, χαρακτηριστικά τα οποία αποτελούν πορίσματα προηγούμενων μετρήσεων της παρούσας διπλωματικής εργασίας.

4.3.2 InfiniBand

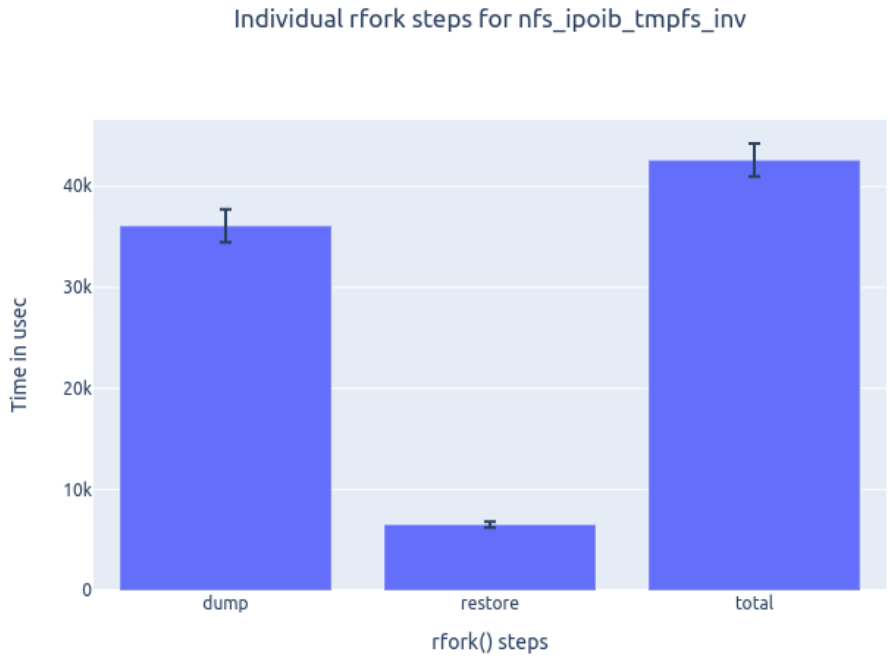
Έπειτα, διεξάγουμε μετρήσεις της απόδοσης της υλοποίησης για διασύνδεση InfiniBand. Για λόγους πληρότητας, παραθέτουμε ξανά τα μετρώμενα αποτελέσματα για τους δύο τύπους συστήματος αρχείων.

4.3.2.1 ext4 Σύστημα αρχείων



Σχήμα 4.16: Χρόνος εκτέλεσης των επιμέρους βημάτων της rfork() για NFS ext4 μέσω InfiniBand

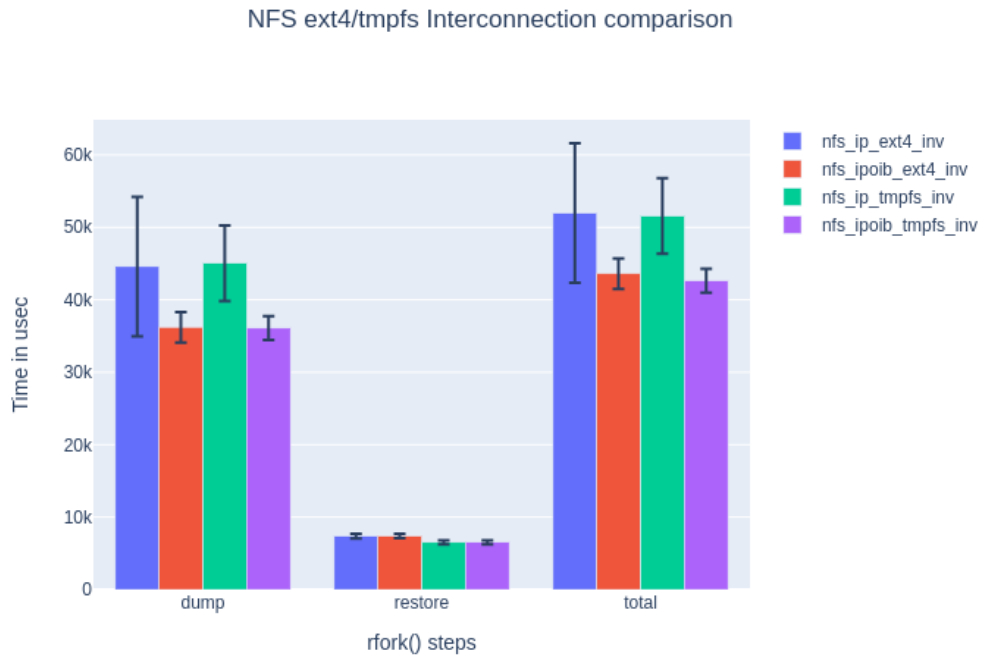
4.3.2.2 tmpfs Σύστημα αρχείων



Σχήμα 4.17: Χρόνος εκτέλεσης των επιμέρους βημάτων της rfork() για NFS tmpfs μέσω InfiniBand

Για άλλη μια φορά, παρατηρούμε ελάχιστη διαφορά στον χρόνο εκτέλεσης της υλοποίησης επί των δύο συστημάτων αρχείων. Παράλληλα παρατηρούμε ελαχιστοποιημένη τυπική απόκλιση για κάθε στάδιο και σύστημα αρχείων, χαρακτηριστικό που αποδίδεται στο αποδοτικότερο του Gigabit Ethernet InfiniBand.

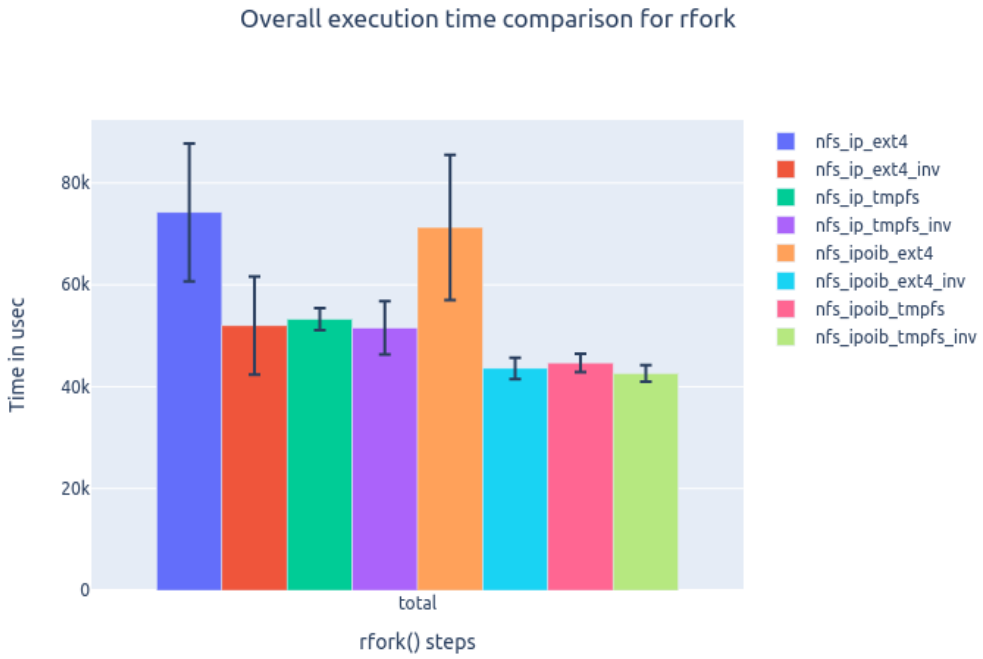
4.3.3 Συμπέρασμα



Σχήμα 4.18: Σύγκριση χρόνου εκτέλεσης σε NFS ext4/tmpfs μέσω Gigabit Ethernet και InfiniBand

Όπως διαφαίνεται από τις μετρήσεις απόδοσης του συστήματος μεταφοράς και απομακρυσμένης εκτέλεσης, που παρατίθενται στο Σχήμα 4.18, η διασύνδεση των κόμβων με InfiniBand αποτελεί τη βέλτιστη επιλογή, για κάθε τύπο συστήματος αρχείου. Παρατηρούμε ελάχιστες διαφοροποιήσεις ως προς τον χρόνο εκτέλεσης του σταδίου επαναφοράς της διεργασίας για κάθε τύπο συστήματος αρχείου, γεγονός που οφείλεται στην επιλογή μας να φιλοξενούμε τον NFS server στον κόμβο προορισμού, δηλαδή αυτόν που εκτελεί τοπικά τη διαδικασία restore. Η διαφοροποίηση εντοπίζεται στη διαδικασία dump, η οποία και επηρεάζεται από τη μέθοδο διασύνδεσης, αναδεικνύοντας την υπεροχή του InfiniBand έναντι του Gigabit Ethernet.

4.4 Σχολιασμός αποτελεσμάτων



Σχήμα 4.19: Συνολική σύγκριση χρόνου εκτέλεσης του υλοποιημένου συστήματος rfork

Οι μετρήσεις της παρούσας ενότητας, οι οποίες συνοψίζονται στο Σχήμα 4.19 , υποδεικνύουν τον βέλτιστο συνδυασμό παραμέτρων, ως προς τον χρόνο εκτέλεσης του συστήματος rfork. Η βέλτιστη υλοποίηση της τοπολογίας, σύμφωνα πάντα με την αξιολόγηση της συγκεκριμένης διπλωματικής εργασίας, απαιτεί τη χρήση tmpfs συστήματος αρχείων, με τον NFS διακομιστή εγκατεστημένο στον κόμβο προορισμού της απομακρυσμένης μεταφοράς και διασύνδεση με τεχνολογία InfiniBand. Η εναλλακτική φύση της συγκεκριμένης υλοποίησης δεν επιτρέπει οποιαδήποτε σύγκριση με την εκτέλεση της υλοποιημένης και standard fork κλήσης συστήματος, με τις δύο υλοποιήσεις να εμφανίζουν αξιοσημείωτες διαφορές ως προς τον χρόνο εκτέλεσης.

Κεφάλαιο 5

Σύνοψη

5.1 Συμπεράσματα

Οι υπάρχουσες υλοποιήσεις για απομακρυσμένη εκτέλεση διεργασιών εισάγουν αυστηρούς περιορισμούς τόσο στην έκδοση πυρήνα όσο και στα patches τα οποία πρέπει να εφαρμόζονται σε αυτόν, προκειμένου να είναι λειτουργικές. Πολλές από αυτές, επιπροσθέτως, δεν προσφέρουν διαφάνεια στις υλοποιήσεις των συμβατών εφαρμογών, απαιτώντας τροποποιήσεις και εκ νέου compilation του κώδικα, ενώ οι περισσότερες είτε είναι κλειστού κώδικα, είτε δεν αναπτύσσονται ενεργά πλέον. Η υλοποίηση της παρούσας διπλωματικής εργασίας εισάγει στην αρχιτεκτονική το εργαλείο CRIU, το οποίο παρέχει λειτουργικότητα checkpoint/restore και αναπτύσσεται ενεργά, σε ανοιχτό κώδικα. Επιπλέον, υλοποιώντας τον μηχανισμό των ενδιάμεσων διεργασιών (Ghost Processes), δεν απαιτείται εφαρμογή οποιουδήποτε patch στον πυρήνα του λειτουργικού συστήματος προκειμένου να υπάρχει διαδιεργασιακή επικοινωνία.

5.2 Μελλοντικές κατευθύνσεις

Αναγνωρίζουμε πως, αν και η υλοποίηση πλέον είναι σε θέση να υποστηρίξει την παράλληλη και απομακρυσμένη εκτέλεση για εφαρμογές που εκτελούν πλήθος υπολογισμών, υπάρχει αρκετός χώρος για επεκτάσεις και βελτιστοποίηση του κώδικά της. Η παρούσα υλοποίηση στηρίζεται σε ένα ειδικά κατασκευασμένο για τις ανάγκες της διπλωματικής εργασίας πρωτόκολλο επικοινωνίας, το οποίο δεν είναι τυποποιημένο και ενδέχεται να περιορίζει τη συμβατότητα με ετερογενή υπολογιστικά συστήματα. Επιπλέον, η υλοποίηση αυτή τη στιγμή δεν έχει αναπτυχθεί ώστε να επιλέγει τον κόμβο προορισμού της απομακρυσμένης εκτέλεσης με συγκεκριμένα κριτήρια, ώστε να βελτιστοποιεί την κατανομή του υπολογιστικού φόρτου του συνολικού συστήματος. Τέλος, η ανάγκη για την ύπαρξη κατανεμημένου συστήματος αρχείων αποτελεί ένα περιοριστικό παράγοντα, ο οποίος ενδέχεται να εξαλειφθεί, όταν υλοποιηθεί πλήρως η λειτουργικότητα post copy μετανάστευσης διεργασιών με χρήση του CRIU page server, ο οποίος καθιστά τα αρχεία κατάστασης της διεργασίας διαθέσιμα απευθείας στο δίκτυο, αποφεύγοντας την ανάγκη για περιττές εγγραφές στον δίσκο. Με βάση τα παραπάνω, ορισμένες ιδέες για μελλοντική στοχοθεσία του έργου της παρούσας υλοποίησης είναι οι εξής:

- Μετάβαση από το ειδικά κατασκευασμένο πρωτόκολλο επικοινωνίας των κόμβων σε εκείνο των protocol buffers μέσω συνδέσεων sockets, προκειμένου να υποστηρίζεται πλήρης ετερογένεια, τόσο στη γλώσσα υλοποίησης, όσο και στο υλικό των κόμβων.

- Χειρισμός των σημάτων SIGKILL και SIGSTOP ώστε να μπορούν να μεταδίδονται με βέλτιστο τρόπο μέσω του διαύλου socket.
- Υλοποίηση συστήματος κατανομής φόρτου και βέλτιστης επιλογής κόμβων προορισμού, ώστε κατά την κλήση fork() να επιλέγεται ο βέλτιστος κόμβος από μια λίστα διαθέσιμων για μετανάστευση.
- Εισαγωγή του CRIU page server στη δομή της υλοποίησής ώστε να παρέχεται η δυνατότητα post copy μεταφοράς διεργασιών, καθώς και στη libcriu βιβλιοθήκη του CRIU για τη C.
- Υλοποίηση, με γνώμονα τη διαφάνεια, επιπλέον μηχανισμών διαδιεργασιακής επικοινωνίας και εφαρμογή του συστήματος απομακρυσμένης εκτέλεσης σε web servers οι οποίοι υποστηρίζουν μοντέλο διεργασιών, όπως ο nginx.

Ακρονύμια

BLCR Berkeley Lab Checkpoint/Restart. 21

CPP Child Pseudo Process. 27–30, 33

CRIU Checkpoint Restore In Userspace. 20, 21, 30, 31, 57

DMTCP Distributed MultiThreaded CheckPointing. 21

FIFO First In First Out. 24

IPC Inter-Process Communication. 24

PID Process Identifier. 16, 21, 23, 29–31

RPC Remote Procedure Call. 20, 21

RPD Remote Process Daemon. 27–30, 32, 33

SCR Scalable Checkpoint Restart. 21

SSI Single System Image. 16

WDS Worker Dispatch Service. 27, 29

Βιβλιογραφία

[1] *MOSIX*, http://www.mosix.org/ σελ. 16

[2] *OpenMOSIX*, http://openmosix.sourceforge.net/ σελ. 16

[3] *Kerrighed*, http://www.kerrighed.org/wiki/index.php/Main_Page σελ. 16

[4] *Popcorn Linux*, http://www.popcornlinux.org/index.php/overview σελ. 16

[5] M. B. Thomas Sterling Matthew Anderson, *High Performance Computing - Modern Systems and Practices*. Morgan Kaufmann, 2018, pp. 591–603. DOI: 10.1016/B978-0-12-420158-3.00020-4 σελ. 19–21

[6] *CRIU Upstream kernel commits*, https://criu.org/Upstream_kernel_commits σελ. 20

[7] *C API for CRIU*, https://criu.org/C_API σελ. 20, 21

[8] *Python API for CRIU*, https://criu.org/Py_API σελ. 20

[9] *Google Protocol Buffers*, https://developers.google.com/protocol-buffers σελ. 20, 29

[10] Adrian Reber, "CRIU and the PID dance", *Linux Plumbers Conference*, 2019 σελ. 21

[11] *Checkpoint/Restore Libraries Comparison*, https://criu.org/Comparison_to_other_CR_projects σελ. 21

[12] Abraham Silberschatz, Peter B. Galvin, Greg Gagne, *Operating System Concepts*, 9th ed. Wiley Publishing, 2012 σελ. 22–26

[13] *The Everything is a File principle*, https://yarchive.net/comp/linux/everything_is_file.html σελ. 24

[14] *shm_ overview - overview of POSIX shared memory, Linux Programmer's Manual*, http://man7.org/linux/man-pages/man7/shm_overview.7.html σελ. 26

[15] Kay A. Robbins, Steven Robbins, *Unix™ Systems Programming: Communication, Concurrency, and Threads*. Prentice Hall, 2003 σελ. 26

[16] AT&T, *UNIX System V User's Manual*. AT&T, 1986, vol. 1 σελ. 26, 33, 34

[17] *mount, Linux Programmer's Manual*, http://man7.org/linux/man-pages/man2/mount.2.html σελ. 31

[18] *Inheriting FDs on restore - CRIU documentation*, https://criu.org/Inheriting_FDs_on_restore σελ. 32

[19] *signalfd, Linux Programmer's Manual*, http://man7.org/linux/man-pages/man2/signalfd.2.html σελ. 33

[**20**] Pavel Yosifovich, Mark E. Russinovich, David A. Solomon, Alex Ionescu, *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more*, 7th ed. Microsoft Press, 2017, vol. 1 σελ. 34

[**21**] *Implementation of System V ftok()*, `https://code.woboq.org/userspace/glibc/sysvipc/ftok.c.html` σελ. 34

[**22**] *HugeTLBPage support in the Linux kernel*, `https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt` σελ. 36

[**23**] *SHMGET(2) - Linux Programmer's Manual*, `http://man7.org/linux/man-pages/man2/shmget.2.html` σελ. 36