ΠΟΛΥΤΕΧΝΕΙΟ
ΚΡΗΤΗΣ

# Trantor - Terminus

Κατανεμημένος Υπολογισμός με Αποκεντρωμένη Επικοινωνία

Χρήστος Σμαραγδάκης

ΧΑΝΙΑ 2011

Στην Ευγενία,
Στο Μιχάλη,
Στο Γιάννη,
Στο Βασίλη

# Περιεχόμενα

# Κεφάλαιο 1

# Εισαγωγή

Οι υπολογιστικές ανάγκες στο ερευνητικό πεδίο πολλών επιστημών ολοένα και αυξάνονται. Η χρήση υπολογιστικών μοντέλων εξομοίωσης πειραμάτων σε διάφορες επιστήμες αποτελεί πλέον ένα πανίσχυρο εργαλείο στα χέρια των επιστημόνων τους και σε ορισμένες περιπτώσεις το κατεξοχήν μέσο για την έρευνα, καθώς η εκτέλεση πραγματικών πειραμάτων είναι πολλές φορές ανέφικτη και τα αποτελέσματά τους αμφιλεγόμενης ακρίβειας [5]. Η Φυσική, η Βιολογία, τα Μαθηματικά, οι Πολιτικές Επιστήμες καθώς και υπολογιστικές εφαρμογές κρυπτογράφησης ή επεξεργασίας video και σημάτων είναι μερικοί από τους βασικούς επιστημονικούς τομείς που χρησιμοποιούν τέτοια πειραματικά μοντέλα. Προβλήματα που μέχρι σήμερα φάνταζαν ανεπίλυτα, τώρα μπορούν να λυθούν, ανοίγοντας έτσι νέες προοπτικές στην επιστημονική έρευνα για χρήση ακόμα πιο ακριβών και ρεαλιστικών πειραματικών μοντέλων.

Παρόλαυτα, το πανίσχυρο αυτό εργαλείο απαιτεί εξαιρετικό μέγεθος υπολογιστικής ισχύος για την εκτέλεση τέτοιων εφαρμογών, σε βαθμό που πλέον η χρήση μεγάλων παράλληλων υπολογιστών, υπερ-υπολογιστών που μπορούν να βρεθούν μόνο σε ερευνητικά ιδρύματα να είναι ανεπαρκής, υπερβολικά ακριβή και δύσκολη στην πρόσβαση. Τα τελευταία χρόνια ο προσανατολισμός για την εκτέλεση τέτοιων πειραμάτων έχει αλλάξει και στρέφεται προς τη χρήση πολλών εκατομμυρίων προσωπικών υπολογιστών προερχόμενων από όλον τον κόσμο, συνδεδεμένων στο Internet. Η εθελοντική προσφορά υπολογιστικής ισχύος από το κοινό, με πρώτιστο παράδειγμα το project SETI@home αποδεικνύουν τη σημασία και τη δύναμη που μπορεί να διατεθεί στην υπηρεσία της επιστήμης αλλά και το ενδιαφέρον για συμμετοχή σε τέτοια επιστημονικά πειρμάματα που μπορεί να έχουν οι απλοί χρήστες.

Η προσφορά υπολογιστικής ισχύος από τους απλούς χρήστες με οικονομικό αντάλλαγμα [16] προς αυτούς αποτελεί ένα ακόμα κίνητρο που μπορεί να θέσει ακόμα μεγαλύτερο όγκο χρηστών και συμμετοχή στη διάθεση ακόμα περισσότερων project. Δεδομένης μάλιστα της εξάπλωσης της ευρυζωνικής πρόσβασης στο Internet ολοένα περισσότεροι προσωπικοί υπολογιστές θα συνδεθούν σε αυτό και θα μπορούν να συμμετέχουν με καλύτερες συνθήκες σε τέτοια project. Πολύ σύντομα και άλλες συσκευές, όπως παιχνιδομηχανές, κινητά τηλέφωνα κλπ. θα διαθέτουν τη δυνατότητα να συνδεθούν με τέτοια δίκτυα υπολογισμού και θα έχουν

την απαραίτητη υπολογιστική ισχύ ώστε να συνεισφέρουν σημαντικά σε τέτοιους υπολογισμούς.

Καθίσταται λοιπόν σαφής η ανάγκη ανάπτυξης συστημάτων που εκμεταλλεύονται αυτές τις οικιακές «υπολογιστικές» συσκευές και δίνουν κίνητρα συμμετοχής στους χρήστες. Ταυτόχρονα όμως διακρίνονται και οι απαιτήσεις που χρειάζεται να ικανοποιούν τέτοιου είδους λύσεις. Τα συστήματα πρέπει να παρέχουν ασφάλεια στους χρήστες από κακοπροαίρετα προγράμματα διαμοιρασμού της υπολογιστικής ισχύος. Πρέπει παράλληλα να παρέχουν εγγυήσεις στους διοργανωτές των project (επιστήμονες, εταιρίες κλπ.) για την ορθή διεξαγωγή τους και θωράκιση από κακοπροαίρετους χρήστες. Τέλος πρέπει να παρέχεται μια αξιόπιστη μέθοδος στατιστικών για τη συνεισφορά κάθε χρήστη, είτε για ηθική είτε για οικονομική ικανοποίησή του.

Η εργασία αυτή είναι δομημένη ως εξής: Στο κεφάλαιο 2 θα κάνουμε μία μικρή επισκόπηση της περιοχής που μελετούμε με ανάλυση σε project, πλατφόρμες εφαρμογών και σχετική έρευνα. Στο κεφάλαιο 3 θα εξετάσουμε τη δική μας πρόταση, την αρχιτεκτονική που ακολουθήσαμε, την υλοποίηση καθώς και προτάσεις για περαιτέρω βελτίωση. Στο κεφάλαιο 4 θα δούμε τα πειραματικά αποτελέσματα των επιδόσεων του συστήματός μας. Στο κεφάλαιο 5 παρατίθενται τα συμπεράσματά μας.

# Κεφάλαιο 2

# Προηγούμενη Δουλειά

Τα τελευταία χρόνια έχει γίνει αξιοσημείωτη δουλειά στην περιοχή του κατανεμημένου υπολογισμού, τόσο με project τα οποία εξετάζουν συγκεκριμένου είδους προβλήματα όσο και με πλατφόρμες ανάπτυξης ανάλογων εφαρμογών. Σε αυτό το κεφάλαιο θα επικεντρωθούμε στις πιο αξιόλογες και δημοφιλείς προσπάθειες, με σκοπό να αναδείξουμε τα κύρια χαρακτηριστικά τους. Στην παράγραφο 2.1 θα παρουσιάσουμε τα παρελθόντα και τρέχοντα project στην περιοχή που εξετάζουμε. Στην παράγραφο 2.2 εξετάζουμε τις υπάρχουσες πλατφόρμες κατανεμημένου υπολογισμού, που επιτρέπουν την ανάπτυξη αλλά και την εκτέλεση διαφόρων project. Στην παράγραφο 2.3 θα παρουσιάσουμε την προηγούμενη έρευνα στην περιοχή με ανάλυση επιλεγμένων papers. Στην παράγραφο 2.4 θα δούμε τα ζητήματα που παραμένουν ανοιχτά στην περιοχή του κατανεμημένου υπολογισμού και τα οποία κατά τη γνώμη μας χρειάζεται να ερευνηθούν και να λυθούν.

## 2.1 **Project** Κατανεμημένου Υπολογισμού

Το SETI@home είναι το πιο δημοφιλές και ως εκ τούτου το μεγαλύτερο εν εξελίξει project στην περιοχή. Το project υποστηρίζεται από το Πανεπιστήμιο του Berkeley και ξεκίνησε το 1999. Σκοπός του είναι να αναλύει δεδομένα που έχουν συλλεγεί από το τηλεσκόπιο του SETI στο Arecibo του Puerto Rico, με σκοπό την αναζήτηση προτύπων σε αυτά που να υποδεικνύουν ότι είναι προϊόν εξωγήινης νοημοσύνης.

Η αρχιτεκτονική του συστήματος αναλύεται και περιγράφεται στο [6]. Αποτελείται από clients που επεξεργάζονται τα δεδομένα και διάφορους servers που παρέχουν τα δεδομένα και συλλέγουν τα αποτελέσματα. Οι servers του SETI@home είναι τρεις: ένας κρατά τα δεδομένα των χρηστών και τα στατιστικά των υπολογισμών τους. Ένας άλλος κρατάει επιστημονικά στοιχεία για κάθε μονάδα εργασίας, καθώς και πόσες φορές έχει δοθεί κάθε εργασία και πόσες φορές έχει επιστρέψει τιμή αποτελέσματος. Ένας τρίτος κρατάει τα δεδομένα των μονάδων εργασίας και τα αποτελέσματά τους. Στο εξής θα χρησιμοποιούμε τον όρο server εννοώντας και τους τρεις αυτούς servers σαν μια ενιαία ομοούσια τριαδική οντότητα, έχοντας

όμως πάντα υπόψιν ποιά δουλειά κάνει ο καθένας, όπως περιγράψαμε παραπάνω. Η διαδικασία του υπολογισμού έχει ως εξής: Το λογισμικό του client συνδέεται στο server του SETI@home για να δεχτεί τα προς επεξεργασία δεδομένα. Τα δεδομένα διαιρούνται από το server σε μικρές μονάδες εργασίας, καθεμιά από τις οποίες δίνεται σε μερικούς clients για ανάλυση. Η φάση της ανάλυσης των δεδομένων είναι ανεξάρτητη για κάθε μονάδα εργασίας. Έτσι, το λογισμικό του client δε συνδέεται με κανένα άλλον client για ανταλλαγή δεδομένων, αλλά μόνο με τον server, μόλις η διαδικασία της επεξεργασίας των δεδομένων ολοκληρωθεί, για να επιστρέψει τα αποτελέσματά της. Το SETI@home δεν κατεβάζει ή εγκαθιστά επιπλέον κώδικα πέρα από τον αρχικό του, παρέχοντας έτσι ασφάλεια στα συστήματα που το εκτελούν. Επίσης, αρκετές μέθοδοι έχουν χρησιμοποιηθεί για να προστατεύσουν το project από κακοπροαίρετους χρήστες ή απρόβλεπτα σφάλματα. Η πιο ουσιαστική και εύκολη μέθοδος ελέγχου των αποτελεσμάτων που χρησιμοποιεί το SETI@home είναι η εξέταση των τιμών των αποτελεσμάτων ώστε να εμπίπτουν στο φάσμα των αποδεκτών τιμών. Παρόλαυτα, αυτή η μέθοδος δεν εξαλείφει τα λανθασμένα αποτελέσματα που προκαλούνται από κακοπροαίρετους χρήστες ή από τροποποιημένο λογισμικό του client. Τέτοιες περιπτώσεις έχουν συμβεί επανειλημμένα στην ιστορία του project του SETI@home. Χρήστες που ήθελαν να αυξήσουν την επεξεργασία των μονάδων εργασίας του συστήματός τους, τροποποίησαν το λογισμικό του client τους με απρόβλεπτες επιπτώσεις στην ορθότητα των αποτελεσμάτων [28]. Ως αποτέλεσμα, μια διαφορετική προσέγγιση έπρεπε να γίνει: Από το ξεκίνημα του project, πάνω από 5 εκατομμύρια χρήστες έχουν λάβει μέρος στο SETI@home, παράγοντας κατά μέσο όρο 72.3 TeraFLOPs το δευτερόλεπτο. Με μια τέτοια υπολογιστική ισχύ, πολλά προβλήματα μπορούν να επιλυθούν, χρησιμοποιώντας πλεονάζοντα (redundant) υπολογισμό. Ο πλεονασμός στον υπολογισμό δίνει τη δυνατότητα για εύκολη ανίχνευση και απόρριψη λανθασμένων αποτελεσμάτων που προκλήθηκαν από προβληματικούς υπολογισμούς ή κακοπροαίρετη παρεμβολή του χρήστη. Όπως προαναφέρθηκε, ο server του SETI@home δίνει κάθε μονάδα εργασίας σε διάφορους clients. Η μέθοδος του πλεονασμού που χρησιμοποιείται στο SETI@home στέλνει κάθε μονάδα εργασίας σε έναν αριθμό $M$, χρηστών και αφαιρεί τη μονάδα εργασίας από το server[7]. Παρόλο που αυτή η μέθοδος μπορεί να έχει ως αποτέλεσμα μερικές μονάδες εργασίας να μην αναλυθούν ποτέ (αν όλοι οι χρήστες που τις πήραν αποτύχουν να επιστρέψουν αποτέλεσμα), είναι προτιμότερη από το να περιμένει ο server να φτάσουν $N$ αποτελέσματα για αυτή τη μονάδα εργασίας, πριν να την αφαιρέσει. Ο λόγος αυτής της απόφασης είναι ότι η δέυτερη μέθοδος προκαλούσε bottleneck από το πιο συχνό γέμισμα του χώρου αποθήκευσης μονάδων εργασίας του server. Είναι επίσης προφανές ότι η ίδια η φύση του προβλήματος επιτρέπει μια μικρή απώλεια αποτελεσμάτων, η οποία μπορεί πρακτικά να εξαλειφθεί αυξάνοντας το $M$. Τα αποτελέσματα συλλέγονται από ένα πρόγραμμα ελέγχου των πλεοναζόντων αποτελεσμάτων, το οποίο τα εξετάζει και παράγει ένα κανονικοποιημένο αποτέλεσμα, βασισμένο στην πλειοψηφία των αποτελεσμάτων.

Η ευρεία δημόσια ανταπόκριση στο project προσέφερε μια ακόμα, πιο σημαντική, επιστημονικά, ευκολία. Η διαθέσιμη υπολογιστική ισχύς κατέστησε δυνατή μια πιο πλατιά, βαθύτερη και πιο ενδελεχή δειγματοληψία των σημάτων του τηλεσκοπίου, που είχε σαν αποτέλεσμα πιο ακριβείς υπολογισμούς.

Το project του SETI@home είναι ένα βολικό πρόβλημα για κατανεμημένο υπολογισμό. Η εργασία του, όπως ισχύει και στην πλειοψηφία των υπαρχόντων project, είναι ένα embarrassingly parallel υπολογιστικό πρόβλημα, που επιτρέπει δηλαδή τη διαίρεση των δεδομένων και την ανεμπόδιστη επεξεργασία τους από τον client χωρίς εξαρτήσεις των δεδομένων και περαιτέρω επικοινωνία. Επιπλέον έχει μια υψηλή αναλογία υπολογισμού προς δεδομένα επικοινωνίας. Τα δεδομένα που ανταλλάσσονται είναι περίπου 350 kb και ο χρόνος επεξεργασίας περίπου 6.5 ώρες για κάθε μονάδα εργασίας. Τέλος, η εργασία παρέχει ανοχή στα σφάλματα. Η απώλεια η ο λανθασμένος υπολογισμός του αποτελέσματος μιας μονάδας εργασίας δεν επηρεάζει τη συνολική δουλειά και μπορεί να αγνοηθεί.

Παρά την ευκολία που παρέχει το πρόβλημά του, το SETI@home εξακολουθεί να είναι το κατεξοχήν επιτυχημένο παράδειγμα εφαρμογής του κατανεμημένου υπολογισμού. Αποτελεί ορόσημο και σημείο αναφοράς ως προς τον όγκο της υπολογιστικής ισχύος που έχει εξασφαλίσει, της επιτυχίας του αλλά και ως προς την αντιμετώπιση σε τόσο μεγάλη κλίμακα πολλών επιστημονικών αλλά και κοινωνικών προβλημάτων που ανέκυψαν κατά τη λειτουργία του.

Την ίδια πορεία και λογική με το SETI@home ακολούθησαν και τα project Folding@Home και Genome@Home του Stanford University[23]. Πρόκειται για προγράμματα που σαν σκοπό τους έχουν τη μελέτη υπολογιστικών εξομοιώσεων που μοντελοποιούν προβλήματα στην επιστημονική περιοχή της βιολογίας, ο υπολογισμός των οποίων ως τώρα φάνταζε ανεπίλυτος λόγω της αλγοριθμικής πολυπλοκότητάς τους. Αυτές οι πειραματικές προσεγγίσεις μέσω υπολογιστικών μοντέλων αντιμετωπίζουν συχνά δύο ειδών εμπόδια: περιορισμούς στην περιγραφή τους, δηλαδή πόσο αφαιρετική είναι η αναπαράσταση του φυσικού αντικειμένου για χάρη του υπολογισμού, και περιορισμούς στη δειγματοληψία, δηλαδή πόσοι συνδυασμοί παραμέτρων μπορούν να δοκιμαστούν και πόσο χρόνο μπορεί το project να διαρκέσει. Οι περιορισμοί της δειγματοληψίας αφορούν τους υπολογιστικούς πόρους που έχουμε οι οποίοι στη σημερινή βιολογία είναι 1000 με 100.000 φορές αργότεροι από ότι απαιτείται. Τους περιορισμούς αυτούς έρχεται να καλύψει ο κατανεμημένος υπολογισμός που παρέχει την ισχύ που η επιστήμη χρειάζεται.

Το Folding@Home μελετά τον τρόπο «αναδίπλωσης» των πρωτεϊνών, ο οποίος είναι υπεύθυνος για πολλές ασθένειες. Το Genome@Home μελετά το ίδιο πρόβλημα από την αντίθετη όψη: τον τρόπο της δημιουργίας μιας πρωτεΐνης με προκαθορισμένη δομή, κάτι που είναι πολύ χρήσιμο για τη δημιουργία φαρμάκων και την ανάπτυξη πρωτεϊνών. Αυτά τα δύο προβλήματα, αλλά και πολλά άλλα, σχετικά με φαινόμενα του φυσικού κόσμου, έχουν ορισμένα χαρακτηριστικά που τα καθιστούν πιο εύκολα στον υπολογισμό. Τα φυσικά φαινόμενα που εξομοιώνουν, η μετάβαση δηλαδή του αντικειμένου παρατήρησης από μία κατάσταση σε μία άλλη, δεν εξελίσσονται σταθερά αλλά «περιμένουν» τις κατάλληλες συνθήκες για να μεταπηδήσουν ένα φράγμα και να περάσουν στο επόμενο επίπεδο της κατάστασής τους [31]. Έτσι, καθίσταται δυνατή η εξομοίωση των φαινομένων αυτών χρησιμοποιώντας πολλές διαδικασίες μικρότερες από το μέγεθος ολόκληρου του φαινομένου. Σε κάθε μία από αυτές πρέπει να γίνει η εξομοίωση για διαφορετικές παραμέτρους ώστε να εξεταστεί με ποιό σετ παραμέτρων το αντικείμενο ξεπερνά πρώτα το φράγμα. Ο παραλληλισμός εφαρμόζεται σε αυτό το σημείο δίνοντας ένα σετ παραμέτρων σε κάθε επεξεργαστή και μοιράζοντας έτσι το χρόνο αναμονής

για το πέρασμα του φράγματος παράλληλα, αντί να συμβεί αυτό σειριακά. Όταν ξεπεραστεί το φράγμα μίας διαδικασίας, όλοι οι επεξεργαστές παίρνουν νέο σετ παραμέτρων και αρχικοποιούνται στις ρυθμίσεις του αντικειμένου που πέρασε πρώτο το προηγούμενο φράγμα. Έτσι ξεκινούν τον υπολογισμό για την επόμενη φάση, δηλαδή το επόμενο φράγμα που πρέπει να ξεπεράσει το αντικείμενο. Πρακτικά, λοιπόν, σε κάθε διαδικασία επιτυγχάνεται επιτάχυνση ίση με το πλήθος των σετ παραμέτρων, ή με το πλήθος των επεξεργαστών (ότι είναι μικρότερο από τα δύο). Επομένως, επειδή οι διαδοχικές διαδικασίες είναι σειριακές, επιτυγχάνεται η ίδια επιτάχυνση και σε ολόκληρο τον υπολογισμό. Η κλιμάκωση του αλγορίθμου έχει λοιπόν τον παραπάνω περιορισμό αλλά και ένα ακόμη: τον χρόνο κατά τη μεταπήδηση του φράγματος, ο οποίος δεν μπορεί να παραλληλιστεί. Άρα αν έχουμε περισσότερους επεξεργαστές από το λόγο του συνολικού χρόνου αναδίπλωσης/συνολικό χρόνο μετάβασης φραγμάτων δεν θα κερδίσουμε σε επιτάχυνση.

Ασχολούμενοι με το ίδιο πρόβλημα, αυτό της αναδίπλωσης των πρωτεϊνών, μια ερευνητική ομάδα από την Ελβετία [38] προσπάθησαν να επιταχύνουν την επίδοση του συστήματος Grid, όπου εφάρμοσαν τους χημικούς αλγορίθμους τους παραλληλίζοντας τις διεργασίες αλλά και τα δεδομένα. Για να επιτύχουν τον παραλληλισμό των δεδομένων στο σύστημά τους εισήγαγαν την έννοια των clustered worker, δηλαδή ενός συνόλου από workers οι οποίοι είναι ομαδοποιημένοι σύμφωνα με τις επεξεργαστικές και δικτυακές τους επιδόσεις. Στο εσωτερικό του κάθε clustered worker, τα μέλη μοιράζονται τα δεδομένα του υπολογισμού τους. Για να επιτευχθεί αυτό, τα μέλη του κάθε cluster worker επιλέγονται ώστε να έχουν γρήγορη δικτυακή σύνδεση μεταξύ τους. Η επιλογή των μελών κάθε clustered worker γίνεται βάσει στατικών network maps. Γενικά το σύστημα κατά τη λειτουργία των clustered workers, για λόγους ευκολίας στον προγραμματισμό του πρωτοτύπου δεν επιτρέπει πολλές μεταβολές και δεν παρέχει μεγάλη ευελιξία και ανοχή στα σφάλματα, κάτι που απαιτεί βελτίωση. Η λειτουργία των workers ακολουθεί τις βασικές αρχές και διαδικασίες που έχουμε περιγράψει στα παραπάνω projects. Τα αποτελέσματα των πειραμάτων έδειξαν μια σαφή βελτίωση στην ποιότητα των αποτελεσμάτων των εξομοιώσεων όταν χρησιμοποιήθηκε παραλληλισμός διεργασιών και δεδομένων έναντι της χρήσης μόνο παραλληλισμού δεδομένων.

## 2.2 Πλατφόρμες Κατανεμημένων Εφαρμογών

Το JET που παρουσιάστηκε από τους H. Pedroso, L. M. Silva και J. G. Silva είναι μία παράλληλη βιβλιοθήκη, για παράλληλο υπολογισμό μέσω Java applets ενσωματωμένα σε μία σελίδα του παγκοσμίου ιστού[33]. Ο χρήστης απλά συνδέεται μέσω του web browser του στη σελίδα του JET, φορτώνει αυτόματα ένα applet το οποίο επικοινωνεί με τον server του JET, κατεβάζει μία εργασία και συνεχίζει με την επεξεργασία της. Η επικοινωνία μεταξύ worker και server γίνεται με χρήση UDP sockets. Παρότι το πρωτόκολλο UDP δεν παρέχει εγγυήσεις για την παράδοση των μηνυμάτων, η επικοινωνία στο JET είναι υλοποιημένη ούτως ώστε να το εγγυάται αυτό χωρίς σφάλματα.

Η χρήση Java applets για την εκμετάλλευση της υπολογιστικής ισχύος των εθελοντών σε παράλληλο υπολογισμό είναι μια ευρέως διαδεδομένη ιδέα που συγ-

κεντρώνει μερικά πολύ σημαντικά πλεονεκτήματα αλλά και ορισμένους εξίσου σοβαρούς περιορισμούς. Στα πλεονεκτήματα της προσέγγισης αυτή περιλαμβάνονται τα εξής:

- Απλότητα στη χρήση: Ο χρήστης συμμετέχει απλά επισκεπτόμενος μία σελίδα με τον web browser του, χωρις να κατεβάζει πρόσθετο λογισμικό στον υπολογιστή του.

- Ασφάλεια: το πλαίσιο εκτέλεσης ενός applet είναι πολύ στενό και δεν επιτρέπει σε ανεπιθύμητες αλληλεπιδράσεις με άλλα προγράμματα, το σκληρό δίσκο ή άλλους πόρους του υπολογιστή. Έτσι ο χρήστης μπορεί να εμπιστευτεί εύκολα την εκτέλεση τέτοιων εφαρμογών.

- Φορητότητα της εφαρμογής: Java applets μπορούν να εκτελεστούν σε όλους τους καινούριους browsers.

Από την άλλη, σοβαρά μειονεκτήματα υπάρχουν στη χρήση Java applets:

- Ο worker δεν μπορεί να χρησιμοποιήσει το σκληρό δίσκο για ενδιάμεση αποθήκευση αποτελεσμάτων, κώδικα και δεδομένων εισόδου. Αυτό περιορίζει κατά πολύ τη λειτουργία πολλών εφαρμογών. Επίσης αναγκάζει τον worker κάθε φορά που θέλει να συμμετέχει να κατεβάζει τα δεδομένα εισόδου και τον κώδικα εκ νέου, επιβαρύνοντας έτσι την επικοινωνία και επομένως την απόδοση του συστήματος.

- Η επικοινωνία μπορεί να γίνει μόνο με τον κεντρικό server από όπου το applet προήλθε, πράγμα που θέτει προβλήματα bottleneck για την επικοινωνία του server και υπονομεύει την δυνατότητα κλιμάκωσης του συστήματος.

- Περιορισμό στα μέρη του JDK που υποστηρίζει ο κάθε browser. Αυτό υπονομεύει την ευελιξία του προγραμματιστή αλλά και τη φορητότητα που είναι χαρακτηριστικό της γλώσσας Java.

Τα Java applets, αν και όπως δείχνουν τα προαναφερθέντα πλεονεκτήματα βοηθούν στην απλότητα του σχεδιασμού και της υλοποίησης των συστημάτων, έχουν τέτοια μειονεκτήματα απόδοσης και φέρνουν τόσους περιορισμούς στο εύρος των εφαρμογών που μπορούν να υλοποιηθούν με αυτά, που κρίνεται πλέον άστοχη η ανάπτυξη μιας ανοιχτής πλατφόρμας κατανεμημένου υπολογισμού για γενικές εφαρμογές βασισμένης σε αυτά.

Οι άνθρωποι του Πανεπιστημίου της Καλιφόρνια, Berkeley, αναγνωρίζοντας την ανάγκη ύπαρξης μίας σταθερής και ανοιχτής πλατφόρμας για Public Resource Computing, που θα φιλοξενεί εφαρμογές όπως το άκρως επιτυχημένο τους SETI@home, ανέπτυξαν το BOINC, του οποίου τα αρχικά σημαίνουν Berkeley Open Infrastructure for Network Computing [8]. Η επιτυχία του SETI@home έφερε πολύ παραπάνω υπολογιστική ισχή από όσο χρειαζόταν. Έτσι, δημιουργήθηκε η πλατφόρμα του BOINC όπου μπορούν να διεξάγονται πολλά πειράματα ταυτόχρονα, εκμεταλλευόμενα αυτή την περίσσεια της προσφοράς.

Ο χρήστης αρχικά κατεβάζει το λογισμικό του BOINC και κατόπιν επισκέπτεται το website ενός από τα project για να εγγραφεί. Ενημερώνεται για το id του μέσω email και με αυτό και το link του project συνδέεται στους servers του project αυτού. Η παραπάνω διαδικασία μπορεί να γίνει για πολλά project και ο διαμοιρασμός της υπολογιστικής ισχύος σε καθένα από αυτά καθορίζεται από το χρήστη. Ο χρήστης ορίζει ένα ελάχιστο και ένα μέγιστο πλήθος εργασιών που θα είναι αποθηκευμένο τοπικά στο μηχάνημά του, για να μην τελειώνουν γρήγορα οι εργασίες. Όταν ο αριθμός των εργασιών πέσει χαμηλότερα από το ελάχιστο, το λογισμικό συνδέεται με το server για να πάρει αρκετές καινούριες εργασίες ώσπου να φτάσει το μέγιστο πλήθος. Την ίδια στιγμή ενημερώνει το server για τις εργασίες που έχει ολοκληρώσει από την προηγούμενη φορά που συνδέθηκε. Μετά το πέρας της επεξεργασίας ο client επικοινωνεί με τον server μέσω HTTP για να επιστρέψει τα αποτελέσματα και αφού αυτά ληφθούν, ενημερώνει τον scheduler για να δρομολογήσει άλλες εργασίες. Αυτό σημαίνει ότι το αποτέλεσμα μπορεί να μείνει για μέρες είτε στον client είτε στον server χωρίς ο scheduler να το γνωρίζει, σε περίπτωση που η σύνδεση δεν είναι εφικτή.

Η ασφάλεια από την πλευρά του χρήστη επιτυγχάνεται με την ψηφιακή υπογραφή της τιμής hashing του κώδικα και των δεδομένων που λαμβάνει. Επιβεβαιώνεται έτσι η αυθεντικότητα του κώδικα ως προς την πηγή του, χωρίς όμως να διασφαλίζεται ότι η πηγή δεν είναι κακοπροαίρετη και ο κώδικας επιβλαβής. Επίσης τίθενται όρια από το ίδιο το project για τη χρήση του χρόνου του επεξεργαστή και από τον χρήστη για τη χρήση του δίσκου. Αν ξεπεραστούν αυτά τα όρια, το project τερματίζεται και τα αρχεία του σβήνονται.

Η ασφάλεια του project είναι ένα σημαντικό ζήτημα που προσπαθεί να αντιμετωπίσει το BOINC. Το ζήτημα της επιστροφής λανθασμένων τιμών αντιμετωπίζεται όπως και στο SETI@home μέσω redundancy, δηλαδή δίνοντας την ίδια εργασία σε παραπάνω του ενός (συνήθως τρεις) workers. Το redundancy είναι σημαντικό φυσικά και για την περίπτωση που κάποιος worker δεν επιστρέψει καθόλου αποτέλεσμα. Το redundancy του BOINC είναι διαφορετικό από αυτό του SETI@home ως προς το ότι εδώ, λόγω του εύρους των εφαρμογών, η κάθε εργασία πρέπει οπωσδήποτε να δώσει ένα ελάχιστο πλήθος redundant αποτελεσμάτων και δεν είναι ανεκτή η απώλεια του υπολογισμού κάποιου αποτελέσματος.

Υπάρχει επίσης το ζήτημα της επιστροφής υπερβολικά μεγάλων τιμών από το χρήστη με σκοπό την επιβάρυνση του server. Αυτό το ενδεχόμενο αντιμετωπίζεται περιορίζοντας το μέγιστο μέγεθος των αρχείων των αποτελεσμάτων που μπορεί να δεχτεί ο server. Τέλος τίθεται το ζήτημα ενός χρήστη να προσπαθήσει να πιστωθεί περισσότερη διεκπεραιωθείσα εργασία από ό,τι αξίζει. Αυτό αντιμετωπίζεται με τη μέθοδο του redundancy και πάλι, μην πιστώνοντας ο server το ματαιόδοξο χρήστη και χρεώνοντας τις μονάδες για την εργασία αυτή σαν πρόσθετες στους υπόλοιπους συμμετέχοντες στο συγκεκριμένο υπολογισμό.

Κατά τον υπολογισμό, μία εφαρμογή του BOINC επιτρέπεται να προβάλλει στοιχεία και γραφικά που αφορούν τον υπολογισμό που γίνεται και την εξέλιξή του, προς τέρψιν και ηθική αποζημίωση του εθελοντή που θα θελήσει να παρακολουθεί την οθόνη του υπολογιστή του.

Το checkpointing του worker γίνεται κατόπιν ελέγχου των ρυθμίσεων του χρήστη για το αν επιτρέπεται εκείνη τη στιγμή να διεξαχθεί (το πρόγραμμα δίνει τη

δυνατότητα στο χρήστη να περιορίσει τη χρήση του δίσκου κατά την εκτέλεσή του για λόγους οικονομίας π.χ. laptop).

Βασισμένοι στην επιτυχία του BOINC, οι Jakob Gregor Pedersen και Christian Ulrik Søttrup παρουσίασαν το Master Thesis τους [32]. Με σκοπό να πετύχουν καλύτερη ποιότητα υπηρεσιών στα συστήματα κατανεμημένου υπολογισμού θέλησαν να δημιουργήσουν ένα bridge του BOINC με τα Grids[32].

Δοκίμασαν το σύστημά τους με ένα project του CERN, το LHC@home, το οποίο προσομοιώνει τη συμπεριφορά ενός επιταχυντή σωματιδίων LHC (Large Hadron Collider) που κατασκευάζεται στο CERN. Ο σκοπός του project ήταν να κατευθύνει το σχεδιασμό του επιταχυντή με τέτοιο τρόπο ώστε να αποφευχθεί η κακή του λειτουργία. Οι υπολογισμοί αφορούσαν εξομοιώσεις που εξέταζαν τη σταθερότητα των τροχιών των σωματιδίων στο χρόνο, κάτω από διαφορετικές συνθήκες.

Στο LHC επιταχύνονται σωματίδια σε ταχύτητα κοντά σε αυτή του φωτός και μετά συγκρούονται σε ελεγχόμενο απο αισθητήρες χώρο. Τα πολλά διαφορετικά σήματα των αισθητήρων χρησιμοποιούνται για να αναπαραστήσουν το φαινόμενο εξετάζοντας έτσι νόμους της φυσικής. Για τη ρύθμιση του μηχανήματος απαιτούνται πολλαπλές εξομοιώσεις με διαφορετικές ρυθμίσεις παραμέτρων. Ταυτόχρονα πρέπει να γίνουν εξομοιώσεις για τις συγκρούσεις των σωματιδίων αφού μόνο ένα μικρό μέρος από αυτές ενδιαφέρουν τους επιστήμονες και δεν είναι εφικτή η πλήρης ανάλυση κάθε σύγκρουσης. Η ανάλυση τέτοιων φαινομένων απαιτεί πολύ παραπάνω από 10000 φορές την ισχύ ενός μέσου επεξεργαστή. Ενδιαφέρον χαρακτηριστικό αυτών των πειραμάτων είναι ότι είναι ανεξάρτητα για κάθε σύγκρουση και επομένως εύκολα παραλληλίσιμα.

Αρχικά το project δοκιμάστηκε πάνω από το κλασικό υπόβαθρο που παρέχει το BOINC. Στη συνέχεια επεξέτειναν τη λειτουργικότητά του BOINC μεταφέροντάς το σε πόρους από ένα σύστημα Grid αποκτώντας έτσι ένα πιο αξιόπιστο σύστημα. Για να γίνει αυτό χρειάστηκε η προσαρμογή του BOINC έτσι ώστε να εκμεταλλεύεται τα χαρακτηριστικά και τους πόρους του Grid. Επίσης απαιτήθηκε η ανάπτυξη ενός metric για Quality of Service και ένας τρόπος να αποδίδεται και να χρησιμοποιείται σωστά.

Η επόμενη προσπάθειά τους ήταν να χρησιμοποιηθούν μαζί πόροι από Grid και πόροι από χρήστες του Internet κάτι που παρουσίασε πολλές δυσκολίες λόγω των περιορισμών των χρηστών για το είδος των εφαρμογών που μπορούν να εκτελέσουν.

Τέλος σχεδίασαν μία αντίστροφη μεταφορά, από εφαρμογές Grid στο BOINC. Κάτι τέτοιο απαιτεί την πρόβλεψη για την εκτέλεση πολύ generic εργασιών του Grid στους non-generic πόρους που διαθέτει η εκάστοτε πλατφόρμα PRC. Ο σχεδιασμός αυτός περιλαμβάνει την επίλυση του προβλήματος της εκτέλεσης εργασιών που προορίζονται για έμπιστους κόμβους (αυτούς του Grid) σε μη έμπιστους. Ο σχεδιασμός ολοκληρώθηκε επιτυχώς, όμως οι περιορισμοί του συστήματος του BOINC δεν επιτρέπουν μια αποδεκτή τέτοια μεταφορά. Κάτι τέτοιο θα απαιτούσε είτε ευρεία γνώση του συστήματος του BOINC, δηλαδή ουσιαστικά τη δημιουργία ενός δεύτερου συστήματος υποβολής εργασιών για το BOINC ή την αφαίρεση στοιχείων ασφάλειας από τον client. Και οι δύο απαιτήσεις κρίθηκαν ανεπιθύμητες, παρολαυτά η ιδέα της δημιουργίας bridge ανάμεσα σε Public Resource Computing συστήματα και σε Grid είναι εφικτή. Η αρχιτεκτονική λογική του συστήματος και η

επικοινωνία μεταξύ client και server είναι πολύ παρόμοια με αυτή του SETI@home.

Ο Luis Sarmenta στη διδακτορική του εργασία στο M.I.T. το 2001 παρουσίασε το Bayanihan[36]. Το Bayanihan είναι μία πλατφόρμα προσφοράς εθελοντικής υπολογιστικής ισχύος, βασισμένο για τη λειτουργία του στη χρήση Java applets. Το Bayanihan, όπως και σχεδόν όλες οι πλατφόρμες κατανεμημένου υπολογισμού, είναι ιδανικό για εφαρμογές τύπου master-worker που απαιτούν μεγάλο φορτίο υπολογισμού και είναι embarrassingly parallel. Μερικές τέτοιες εφαρμογές είναι οι brute-force αναζητήσεις, το rendering εικόνων, οι γενετικοί αλγόριθμοι, η παραμετρική ανάλυση και οι προσομοιώσεις Monte Carlo.

Πέρα από αυτές τις εφαρμογές, το Bayanihan επιχειρεί να ανοίξει την υποστήριξή του σε ένα πιο ευρύ φάσμα εφαρμογών, σε αυτές που χρησιμοποιούν message-passing. Το μοντέλο που προτείνεται για να αντιμετωπίσει αυτή την πρόκληση ονομάζεται BSP (Bulk Synchronous Parallel) και υποστηρίζει την επικοινωνία μεταξύ των workers αλλά σε συγκεκριμένα χρονικά σημεία συγχρονισμού και για όσα μηνύματα έχουν συλλεγεί, μαζικά. Κατά το BSP, οι workers εκτελούν τα εξής κυκλικά επαναλαμβανόμενα βήματα:

- Εκτελείται ο τοπικός υπολογισμός: Για την καλή απόδοση του συστήματος, ο υπολογισμός πρέπει να είναι αρκετά απαιτητικός σε υπολογιστική ισχύ.
- Μεταδίδονται όλα τα μηνύματα μεταξύ των κόμβων: Ανάλογα με τις απαιτήσεις που προκύπτουν από τον τοπικό υπολογισμό σε κάθε κόμβο, οι κόμβοι ανταλλάσσουν δεδομένα.
- Γενικός συγχρονισμός: Τίθεται ένα φράγμα (barrier) στο οποίο περιμένουν όλοι οι κόμβοι μέχρι να ολοκληρωθούν όλες οι επικοινωνίες του βήματος 2 ώστε να ξεκινήσει η επανάληψη της διαδικασίας από το βήμα 1.

Η επικοινωνία στο τέλος του υπολογισμού και ο συγχρονισμός μέσω barrier εγγυάται την ανεξαρτησία μεταξύ των εργασιών κατά τη διάρκεια μίας επανάληψης από τα παραπάνω 3 βήματα. Αυτό παρέχει ευκολία προγραμματισμού αλλά και ευκολία στην εφαρμογή σε ήδη υπάρχοντα συστήματα master-worker.

Τέλος, στο Bayanihan αντιμετωπίζεται το ζήτημα της ορθότητας των αποτελεσμάτων και της προστασίας απέναντι σε κακοπροαίρετους χρήστες. Αυτό επιτυγχάνεται με μηχανισμούς voting, με δειγματοληπτικούς ελέγχους των workers για εργασίες με γνωστά αποτελέσματα και με τη χρήση ανοχής σφαλμάτων βασισμένη στην αξιοπιστία. Οι μηχανισμοί που προτείνονται, αναλύονται καλύτερα στη σχετική παράγραφο 2.3.

Στο [3] παρουσιάζεται το σύστημα ParCop, ένα αποκεντρωμένο σύστημα υπολογισμού peer-to-peer. Στο ParCop δεν υπάρχει κεντρικός server που να ρυθμίζει και να δρομολογεί τις εργασίες και τα δεδομένα. Το σύστημα επιτρέπει σε κάθε peer να χρησιμοποιεί και να προσφέρει υπολογιστικούς πόρους. Είναι ανεπτυγμένο σαν ξεχωριστή εφαρμογή Java και μπορεί να εκτελεστεί σε κάθε πλατφόρμα που υποστηρίζει Java. Έτσι, το ParCop αντιμετωπίζει προβλήματα bottleneck και αστοχίας του εξυπηρετητή, υποστηρίζοντας εφαρμογές τύπου master/worker που μπορούν να χωριστούν σε ανεξάρτητα, μη επικοινωνούντα tasks. Ένας peer μπορεί

να είναι είτε master (δηλαδή να διανέμει εργασίες και να συλλέγει τα αποτελέσματα) είτε worker (δηλαδή να εκτελεί υπολογισμούς και να επιστρέφει τα αποτελέσματά τους) κάθε φορα. Οι peers διατηρούν μόνιμη επικοινωνία με τους γείτονές τους για να γνωρίζουν ποιοί είναι ζωντανοί, καθώς και προσωρινή επικοινωνία κατά τη διαδικασία υπολογισμού ενός task. Προκειμένου ο master να βρει έναν idle worker στέλνει μηνύματα αιτήσεων στους γείτονές του και αυτοί τα προωθούν ώσπου να βρεθεί κάποιος idle ο οποίος απαντά μέσω του ίδιου μονοπατιού. Όταν βρει όσους workers χρειάζεται, στέλνει τα tasks και ξεκινά ο υπολογισμός. Η επικοινωνία γίνεται μόνο μεταξύ των workers και του master, μέσω της προσωρινής σύνδεσης που αποκαθιστούν και αφορά την αρχή και το τέλος, δηλαδή την είσοδο και την έξοδο για κάθε task. Διατηρώντας μία λίστα με τους αναξιόπιστους workers διευκολύνεται η σωστή λειτουργία του συστήματος. Το σύστημα δείχνει ικανοποιητική κλιμάκωση με τον αριθμό των peers που χρησιμοποιούνται και ανοχή στα σφάλματα. Γενικά οι ιδανικές εφαρμογές για το ParCop είναι αυτές με υψηλό λόγο υπολογισμού προς επικοινωνία.

Το Cosm [25] είναι ένα σύστημα που αναπτύχθηκε από την εταιρία Mithral και είναι γραμμένο σε C. Η αρχιτεκτονική του Cosm χωρίζεται σε τρία επίπεδα:

- Υπάρχουν οι servers που λειτουργούν σε clusters του ενός ή παραπάνω υπολογιστών. Η δουλειά τους είναι να μοιράζουν εργασίες, να κρατούν τα απαραίτητα για το κάθε project δεδομένα και να συλλέγουν τα αποτελέσματα. Οι servers επικοινωνούν με τους γειτονικούς proxies και τους servers αλλά όχι απευθείας με τους clients.

- Οι proxies είναι οι ενδιάμεσοι του δικτύου και επιτελούν το έργο της επικοινωνίας, αποθηκεύοντας προσωρινά και μεταφέροντας δεδομένα και εργασίες από τους servers. Οι proxies επίσης επικοινωνούν μεταξύ τους για να προωθούν μηνύματα, να εξισορροπούν το φορτίο και να επιλύουν προβλήματα στο δίκτυο.

- Οι clients παίρνουν εργασίες από τους proxies, τις εκτελούν και επιστρέφουν τα αποτελέσματα σε αυτούς.

Οι proxies και οι servers λειτουργούν σε δακτυλίους, όπου κάθε κόμβος σε έναν δακτύλιο είναι ικανός να λειτουργήσει όπως οποιοσδήποτε άλλος. Κάθε δακτύλιος δηλαδή λειτουργεί σαν ένας server ή σαν ένας proxy. Ένας proxy μπορεί να ανήκει μόνο σε ένα δακτύλιο και ένας client συνδέεται μόνο με ένα δακτύλιο από proxies. Ένας δακτύλιος μπορεί να είναι συνδεδεμένος ιεραρχικά προς τα κάτω με έναν ή περισσότερους δακτυλίους - τέκνα και προς τα πάνω με έναν μόνο δακτύλιο-γονέα. Ένας κόμβος proxy μπορεί να επικοινωνεί μόνο με τους proxies του δακτυλίου του ή με εκείνους γειτονικού δακτυλίου και τα παιδιά του. Ως προς την ασφάλεια του συστήματος, κάθε κόμβος προτού χρησιμοποιήσει δεδομένα ή δεχτεί εντολές από άλλο κόμβο πρέπει να επιβεβαιώσει ότι προέρχονται από έμπιστη πηγή, διαφορετικά τα αγνοεί. Στην αρχιτεκτονική του Cosm όλοι οι κόμβοι μπορούν να επικοινωνούν μεταξύ τους για τα δεδομένα που χρειάζεται να ανταλλάξουν, οποιαδήποτε στιγμή. Η σχεδίαση παρέχει μεγάλη ανοχή, ώστε αν ένας κόμβος χαθεί, κανένας άλλος δεν

θα επηρεαστεί. Οι χρήστες, έχοντας τα κατάλληλα κλειδιά, μπορούν να ελέγξουν για τους υπολογισμούς τους τους clients που χρειάζονται.

Η κλιμάκωση του συστήματος φαίνεται να είναι ικανοποιητική. Με μετριοπαθείς υπολογισμούς, έχοντας 32 κύριους proxies και 64 κεντρικούς servers το σύστημα αντέχει 176 εκατομμύρια clients.

Παρόλα τα πλεονεκτήματά της, η ανάπτυξη της πλατφόρμας του Cosm μοιάζει να έχει σταματήσει τα τελευταία χρόνια.

Στο Πανεπιστήμιο του Paris Sud της Γαλλίας ο Franck Cappello, ο Gilles Fedak και άλλοι ανέπτυξαν μία πλατφόρμα παράλληλου υπολογισμού βασισμένη στην αρχιτεκτονική του Global Computing, το XtremWeb[20]. Η βασική ιδέα του Global Computing είναι η χρήση του ανεκμετάλλευτου χρόνου πολλών υπολογιστικών πόρων που είναι συνδεδεμένοι με ευρεία διασπορά στο Διαδίκτυο και παρέχουν εθελοντικά αυτή την υπολογιστική ισχύ. Η αρχιτεκτονική του συστήματος υποστηρίζει πολύ μεγάλο αριθμό κόμβων, μεγάλη κινητικότητα αυτών και χαμηλή ποιότητα επικοινωνίας μεταξύ τους. Το XtremWeb υποστηρίζει δύο τρόπους χρήσης: Ως user, όπου παρέχεται η υπολογιστική ισχύς του όσο είναι αδρανής, και ως collaborator όπου λειτουργεί σαν server που μοιράζει τις εργασίες στους συνδεδεμένους με αυτόν υπολογιστές και επιτρέπει στο υπόλοιπο δίκτυο να τους χρησιμοποιήσει όταν ο ίδιος δεν τους χρειάζεται. Η ασφάλεια του συστήματος προς τον χρήστη περιλαμβάνει έμπιστους παροχείς κώδικα, δοκιμαστική εκτέλεση του σε workers, κρυπτογράφησή του και ασφαλής μετάδοσή του στον worker και αποστολή checksum από το worker στον server για επιβεβαίωση. Η επικοινωνία ξεκινά από την πλευρά του χρήστη και περιλαμβάνει τα εξής βήματα:

- διαπίστευση από τον server και λήψη λίστας των διαθέσιμων servers
- αίτηση εργασίας προς τον server και επιστροφή μιας διαθέσιμης από αυτόν
- ενημέρωση του server για τη συνέχιση της επεξεργασίας (isAlive)
- επιστροφή των αποτελεσμάτων και ενημέρωση του server.

Όπως είναι φανερό, στο XtremWeb δεν διατίθεται η δυνατότητα επικοινωνίας μεταξύ των κόμβων. Η βασική ανάγκη για τη δημιουργία του ήταν η παροχή μιας ενιαίας πλατφόρμας υποστήριξης πολλαπλών εφαρμογών του τύπου του SETI@home για πολλούς workers και με μεγάλο scalability. Αρχικά προοριζόταν για χρήση από το Παρατηρητήριο κοσμικής ακτινοβολίας Pierre Auger [24] που απαιτούσε ετησίως ένα μεγάλο αριθμό εξομοιώσεων και πειραμάτων για την ανάπτυξη και δοκιμή θεωρητικών επιστημονικών μοντέλων. Έτσι, η κλιμάκωση της πλατφόρμας φτάνει τις αρκετές εκατοντάδες χιλιάδες κόμβους και επιτυγχάνεται χάρη στη χρήση ενός cluster από servers που δρομολογούν τις εργασίες και ενός meta-server που αναθέτει τους workers σε κάποιον server. Επίσης εφαρμόζονται τεχνικές load balancing μεταξύ των servers και χρήση εξειδικευμένων servers για συγκομιδή των αποτελεσμάτων.

Ενώ η λογική πάνω στην οποία στηρίχθηκε του XtremWeb είναι ο παράλληλος υπολογισμός πάνω από ένα δίκτυο που παρουσιάζει τα χαρακτηριστικά των peer-to-peer δικτύων, η ίδια ομάδα αργότερα εξέτασε το ενδεχόμενο συνένωσης του

XtremWeb με τα Grid και σύγκρινε τα χαρακτηριστικά των δύο αρχιτεκτονικών [17]. Η γνώση των σημαντικών πληροφοριών για τους συμμετέχοντες στο σύστημα που παρέχει το Grid (χαρακτηριστικά μηχανήματος, συμμετοχή στον υπολογισμό κλπ.) καθώς, η ασφάλεια του συστήματος και ο έλεγχός των συμμετεχόντων που παρέχεται στον διοργανωτή του υπολογισμού αποτελούν πλεονεκτήματα που αλλάζουν και την ίδια την αρχιτεκτονική των συστημάτων που λειτουργούν πάνω από το Grid. Έτσι, μια συνένωση του XtremWeb με το Grid θα καθιστούσε το πρώτο πιο ευέλικτο και ασφαλές.

Η εταιρία Parabon Computation παρουσίασε τη δική της πλατφόρμα κατανεμημένου υπολογισμού, το Frontier[27]. Το Frontier είναι μια εμπορική εφαρμογή ως προς την παροχή της πρόσβασης στους υπολογιστικούς πόρους και στα εργαλεία για την ανάπτυξη και την εκτέλεση εφαρμογών πάνω από την πλατφόρμα. Οι workers παρέχουν εθελοντικά την υπολογιστική ισχύ των μηχανημάτων τους όταν αυτά είναι αδρανή. Τα συστατικά μέρη της πλατφόρμας χρησιμοποιούν τη γλώσσα Java, όπως και οι εφαρμογές που εκτελούνται στην πλευρά του worker.

Ο client παρέχει τον κώδικα και ρυθμίζει τις παραμέτρους εκτέλεσής του. Μπορεί να επιλέξει κάθε εργασία να αποδίδεται σε παραπάνω του ενός workers για λόγους redundancy και αξιοπιστίας. Επίσης καθορίζει ποιά στοιχεία από τους πόρους του worker θα χρειαστεί η εκτέλεση του προγράμματός του. Χωρίζει την εφαρμογή σε tasks και τα στέλνει στον server.

Ο server συντονίζει τη λειτουργία των παράλληλων υπολογισμών. Ελέγχει την κατάσταση κάθε worker (αν είναι «ζωντανός») και την πορεία των εργασιών που έχει μοιράσει. Αν ένας worker έχει καθυστερήσει να δώσει αποτελέσματα και δεν αποκρίνεται, δίνει τις εργασίες του σε άλλον worker. Η αρχιτεκτονική του παρέχει αξιοπιστία, διαθεσιμότητα και σταθερότητα μέσω του redundancy που χρησιμοποιεί.

Ο worker κατεβάζει από τον server μία ή παραπάνω εργασίες προς επεξεργασία και τις αποθηκεύει στο δίσκο.

Για λόγους ασφάλειας, χρησιμοποιείται η τεχνική εκτέλεσης σε κλειστό, ελεγχόμενο περιβάλλον που παρέχει η γλώσσα Java (sandboxing στα ελληνικά) και που εφαρμόζεται με ανάλογο τρόπο στα applets. Η μέθοδος αυτή δεν επιτρέπει στον κώδικα να λειτουργήσει έξω από τα περιορισμένα πλαίσια λειτουργίας του (π.χ. να διαβάσει χωρίς έγκριση πληροφορίες από το δίσκο, να αλληλεπιδράσει με άλλα προγράμματα ή να χρησιμοποιήσει ανεξέλεγκτα το δίκτυο). Σε κάθε επικοινωνία χρησιμοποιείται το πρωτόκολλο SSL για τη διαπίστευση του worker στον server, για τη μεταφορά του κώδικα, των δεδομένων εισόδου και των αποτελεσμάτων. Επίσης, ο worker δεν μπορεί να γνωρίζει τι είδους εφαρμογές εκτελούνται ανά πάσα στιγμή στον υπολογιστή του, απομακρύνοντας έτσι το ενδεχόμενο να κάνει reverse engineering στον κώδικα, να υποκλέψει τα αποτελέσματα ή να προσπαθήσει να τα παραποιήσει, αφού δεν γνωρίζει τη φύση τους. Τέλος, για να εντοπίσει τυχόν κακοπροαίρετους workers ο server δίνει τυχαία εργασίες που γνωρίζει το αποτέλεσμα. Σε περίπτωση που το αποτέλεσμα του worker δεν είναι το σωστό, τον διαγράφει από το σύστημα.

Η επικοινωνία, όπως παρουσιάστηκε παραπάνω, γίνεται μόνο μεταξύ client-server και worker-server [26]. Οι workers δεν επικοινωνούν μεταξύ τους κατά τη διάρκεια του υπολογισμού. Οι εφαρμογές που είναι ιδανικές για το Frontier,

λοιπόν, εμπίπτουν κυρίως στην κατηγορία των embarrassingly parallel εφαρμογών (π.χ. αλγόριθμοι divide and conquer, συνδυαστικά προβλήματα με πολλές παραμέτρους, Monte Carlo μοντέλα, νευρωνικά δίκτυα). Οι εργασίες πρέπει να μπορούν να χωριστούν σε μικρές εργασίες που είναι ανεξάρτητες μεταξύ τους και να περιλαμβάνουν εντατικό υπολογισμό (η αναλογία υπολογισμού/δεδομένων να είναι μεγάλη).

Το distributed.net είναι ένα επίσης σημαντικό project της περιοχής που έχει μεγάλη επιτυχία [1]. Επικεντρώνεται σε προβλήματα κρυπτογράφησης χρησιμοποιώντας κατανεμημένο υπολογισμό και προσπαθεί να σπάσει αλγορίθμους κωδικοποίησης. Αυτή η προσπάθεια βασίζεται στην εξαντλητική δοκιμή κλειδιών, κάτι που προφανώς είναι εύκολα παραλληλίσιμο αφού περιλαμβάνει επαναληπτική δοκιμή της ίδιας εφαρμογής για διαφορετικές εισόδους. Σε αντίθεση με τον client του SETI@home, που υπολογίζει μία μόνο εφαρμογή, ο client του distributed.net περικλείει διάφορους αλγορίθμους έτοιμους για εκτέλεση. Ο υπολογισμός αυτών των αλγορίθμων απαιτεί πολύ μικρό μέγεθος δεδομένων εισόδου και εξόδου, μηδενική επικοινωνία κατά τον υπολογισμό και μεγάλο φόρτο υπολογισμού. Στην υπόλοιπη λειτουργία και αρχιτεκτονική του, το σύστημα ακολουθεί την προσέγγιση client-server με διαδοχικές λήψεις εργασιών, επεξεργασία τους και επιστροφή των αποτελεσμάτων. Για την ορθότητα των αποτελεσμάτων χρησιμοποιείται τυχαίο και δειγματοληπτικό redundancy. Ως προς την ασφάλεια, το distributed.net χρησιμοποιεί κρυπτογράφηση των δεδομένων και των αποτελεσμάτων, πράγμα που είναι διασκεδαστικά ειρωνικό, όταν την ίδια στιγμή προσπαθεί να σπάσει παρόμοιους αλγορίθμους κρυπτογράφησης.

Το ParaWeb [15] είναι ένα σύστημα που χρησιμοποιεί ετερογενείς υπολογιστικούς πόρους πάνω από το Internet ή από intra-nets για τον υπολογισμό σειριακών προγραμμάτων σε ισχυρούς υπολογιστές του δικτύου, ή τον παράλληλο υπολογισμό προγραμμάτων σε πολλούς υπολογιστές. Χρησιμοποιεί υπάρχοντες μηχανισμούς που παρέχει η γλώσσα Java για λόγους ευκολίας και προσιτότητας προς τον προγραμματιστή των παράλληλων εφαρμογών και προς αυτόν που επιθυμεί να τις εκτελέσει. Το ParaWeb παρέχει δυνατότητες απομακρυσμένης δημιουργίας και εκτέλεσης νημάτων σε έναν worker και διευκολύνει την επικοινωνία μεταξύ τους. Το διαμοιρασμό των εργασιών και τη διατήρηση των πληροφοριών για τους workers τον εκτελέι ο scheduling server. Ο παροχέας του κώδικα συμβουλεύεται τον scheduling server όμως αναλαμβάνει ο ίδιος την αποστολή του κώδικα και τη συλλογή των αποτελεσμάτων.

Στο [18] παρουσιάζεται το YML, ένα framework που προσφέρει δυνατότητες σχεδιασμού και εκτέλεσης παράλληλων εφαρμογών πάνω από δίκτυα peer-to-peer και grids. Το YML αναλαμβάνει να επιλύσει κεντρικά και αυτόματα τα ζητήματα της ετερογένειας των πόρων, της ανεκτικότητας σε σφάλματα, της διαχείρισης πόρων και της χρονοδρομολόγησης, προβλήματα που αφορούν το middleware και που μέχρι τώρα επιλύονταν από το χρήστη και κατά περίπτωση. Όμως, στα περιβάλλοντα που λειτουργούν αυτές οι εφαρμογές, οι πόροι παρουσιάζουν έντονη κινητικότητα με αποτέλεσμα η διαχείρισή τους από το χρήστη να είναι πρακτικά αδύνατη. Αυτό σημαίνει ότι η ίδια η εφαρμογή πρέπει να προσαρμόζεται αυτόματα στις συνθήκες. Το YML παρέχει εργαλεία ανάπτυξης παραλληλων εφαρμογών που παρέχουν abstraction ώς προς την ετερογένεια των πόρων κάθε δικτύου ώ-

στε οι εφαρμογές να συμπεριφέρονται ενιαία σε αυτούς. Αυτό δίνει τη δυνατότητα μεταφερσιμότητας των εφαρμογών ανάμεσα στα middleware. Το abstraction που δίνει το YML παρέχει ταυτόχρονα και ένα σύνολο λειτουργιών στην εφαρμογή του χρήστη, ανεξαρτήτως middleware. Το YML χρησιμοποιεί μια γλώσσα περιγραφής παράλληλων εφαρμογών που περιλαμβάνει περιγραφή του ακυκλικού γράφου εξάρτησης των στοιχείων της εφαρμογής. Ο γράφος αποτελεί την αναπαράσταση του παράλληλου αλγόριθμου. Τα στοιχεία του γράφου μπορεί να αποτελούνται από εργασίες σε διαφορετικούς workers που έχουν τη δυνατότητα να επικοινωνούν μοιραζόμενοι τα δεδομένα τους μέσω ενός κεντρικού μηχανισμού κοινής μνήμης. Η κοινή μνήμη μπορεί να θεωρηθεί ως ένα ενδιάμεσο checkpoint στη λειτουργία του υπολογισμού. Αυτή η δόμηση της κεντρικής κοινής μνήμης επιφέρει ένα αναμενόμενο bottleneck στη λειτουργία του server που φιλοξενεί το YML framework. Αν και υποστηρίζεται λοιπόν επικοινωνία ανάμεσα στα στενά συνδεδεμένα tasks (που ορίζονται ως ενιαία στοιχεία του γράφου), αυτή η επικοινωνία δεν είναι άμεση αλλά με μεσάζοντα. Το Framework κατά τη διάρκεια του υπολογισμού εξετάζει και ελέγχει την κατάσταση του middleware ώστε να εξυπηρετήσει τις αιτήσεις για εργασίες διαχειριζόμενο τις εξαρτήσεις μεταξύ τους, που περιγράφονται στον γράφο. Τα πειραματικά αποτελέσματα δεν είναι ιδιαίτερα ικανοποιητικά κυρίως λόγω του προαναφερθέντος bottleneck στον server που παρέχει την κοινή μνήμη στις διεργασίες, το οποίο αποτελεί ένα σχεδιαστικό πρόβλημα, αλλά και λόγω του προβλήματος που επελέγη για τις μετρήσεις που περιλαμβάνει μεγάλη επικοινωνία και μικρό υπολογισμό. Παρολαυτά η ιδέα της επικοινωνίας μεταξύ των workers εξακολουθεί να είναι ένα ζητούμενο που περιμένει λύση.

Το Javelin++ είναι η πλατφόρμα του Πανεπιστημίου της Καλιφόρνια στη Santa Barbara προορισμένη για Global Computing και γραμμένη σε Java [29]. Στο σύστημα αυτό υπάρχουν τρεις οντότητες:

- οι clients, οι οποίοι προσφέρουν εργασίες και αναζητούν υπολογιστικούς πόρους για να τις εκτελέσουν
- οι hosts, που φιλοξενούν τέτοιες εργασίες, παρέχοντας την υπολογιστική τους ισχύ
- οι brokers, οι οποίοι συντονίζουν τον εντοπισμό των υπολογιστικών πόρων

Η επικοινωνία στο σύστημα χρησιμοποιεί το Java RMI και οι εργασίες είναι εφαρμογές Java (και όχι applets). Κατά τη λειτουργία του συστήματος, οι clients προσφέρουν εργασίες και οι hosts τις ζητούν από τους brokers. Εκείνοι τις αναθέτουν στους hosts, οι οποίοι τις εκτελούν και επιστρέφουν τα αποτελέσματα πίσω στους brokers. Ένα μηχάνημα μπορεί να είναι ταυτόχρονα και host και client. Οι brokers αποτελούνται από ένα κατανεμημένο δίκτυο υπολογιστών και μπορούν να επικοινωνήσουν με σταθερό αριθμό (διαφορετικό ανάλογα με τον broker) από άλλους brokers ανά πάσα στιγμή. Ομοίως, κάθε broker είναι υπεύθυνος για σταθερό άριθμό από hosts. Οι σταθερές αυτές καθορίζονται από την ισχύ και την ταχύτητα δικτύου του broker. Οι brokers είναι ουσιαστικά άλλη μια εφαρμογή που τρέχει πάνω στο δίκτυο του Javelin++, με τη διαφορά ότι υπάρχουν απαιτήσεις αξιοπιστίας και ισχύος για αυτούς.

Η ασφάλεια του συστήματος εξασφαλίζεται μέσω της υλοποίησης ενός Security Manager που διασφαλίζει την επικοινωνία μόνο μεταξύ εφαρμογών του Javelin++ και περιορίζει την πρόσβαση σε τοπικούς πόρους.

Στο Javelin++ χρησιμοποιούνται δύο προσεγγίσεις για το διαμοιρασμό των εργασιών και τη δομή του δικτύου, μία probabilistic και μία deterministic. Στο probabilistic model ο κάθε host διατηρεί τοπικά ένα hash table με διευθύνσεις άλλων hosts και μία double-ended λίστα με εργασίες. Από το ένα άκρο της λίστας εργασιών παίρνει ο ίδιος και από το άλλο άκρο εξυπηρετούνται οι απομακρυσμένες αιτήσεις για εργασία. Όταν τελειώσουν οι τοπικές εργασίες ο host κάνει αίτηση για μία νέα από ένα τυχαία επιλεγμένο γείτονα από το hash table. Ο γείτονας αποθηκεύει τη διεύθυνση του αιτούντα στο δικό του hash table και ακόμα και αν δε διαθέτει εργασία του επιστρέφει έναν αριθμό δικών του γειτόνων. Έτσι εμπλουτίζεται διαρκώς το hash table των hosts του καθενός. Σε αυτή την προσέγγιση δεν υπάρχει κεντρικός κόμβος. Στο deterministic model χρησιμοποιείται μία δομή ισορροπημένου δέντρου. Κάθε host παίρνει κάθε φορά ένα κομμάτι της δουλειάς από τον πατέρα του και όταν τελειώσει ζητά δουλειά από τα παιδιά του, διαφορετικά και πάλι από τον πατέρα του. Αυτό διασφαλίζει την ολοκλήρωση της δουλειάς σε κάθε υποδέντρο πρώτα. Η ρίζα του δέντρου είναι ο client και με τη δομή αυτή εξασφαλίζεται ότι η δουλειά είναι ισόρροπα μοιρασμένη από άποψη φορτίου.

Οι δύο αυτές προσεγγίσεις εξασφαλίζουν καλή κλιμάκωση και παρουσιάζουν παρόμοιο speedup (52 για 60 επεξεργαστές η πρώτη, 46 για 52 επεξεργαστές η δεύτερη).

Το 2006 οι Dayi Zhou και Virginia Lo από το πανεπιστήμιο του Oregon παρουσίασαν το WaveGrid [40]. Το Wavegrid είναι ένα peer-to-peer σύστημα αυτοοργανωνόμενων ετερογενών κόμβων. Η οργάνωση των κόμβων σε διάταξή ανάλογη προς τα peer-to-peer συστήματα διαμοιρασμού αρχείων εξασφαλίζει κλιμάκωση και αποκλείει το ενδεχόμενο bottleneck. Οι κόμβοι μπορούν να είναι ταυτόχρονα workers που επεξεργάζονται εργασίες αλλά και clients που προσφέρουν εργασίες. Το σύστημα υποστηρίζει γρήγορη ανεύρεση διαθέσιμων πόρων, με αποστολή λίγων μηνυμάτων, κάτι που είναι δύσκολο (και για αυτό το λόγο σημαντικό) σε τέτοια συστήματα αφού η σύνθεσή τους αλλάζει ταχύτατα και δυναμικά, με πολλούς κόμβους να εισέρχονται και να εξέρχονται, σε και από αυτά, διαρκώς. Επίσης το σύστημα υποστηρίζει χρονοδρομολόγηση προσαρμοζόμενη στην ετερογένεια κάθε κόμβου και μεταφορά του φόρτου υπολογισμού σε κόμβους που βρίσκονται σε νυχτερινή ζώνη ώρας, κατά την οποία η διαθεσιμότητα είναι πολύ μεγαλύτερη. Η κάθε εργασία «μεταναστεύει» σε επόμενη ζώνη ώρας όταν το μηχάνημα δεν είναι πλέον διαθέσιμο. Τέλος, το σύστημα παίρνει ελάχιστες πληροφορίες για το μηχάνημα κάθε χρήστη (γεωγραφική θέση, CPU clock) ώστε να μην τον αναγκάζει να μοιραστεί πληροφορίες που δεν θα ήθελε να μοιραστεί.

Τα βήματα που ακολουθεί η λειτουργία του συστήματος είναι τα εξής:

- Το δίκτυο χωρίζεται σε ομάδες από ζώνες ώρας και οι κόμβοι τους ομαδοποιούνται σε αυτές.

- Ο κόμβος συνδέεται με το δίκτυο δηλώνοντας τη ζώνη ώρας του και μπαίνοντας στην αντίστοιχη ομάδα

- Όποιος θέλει να προσφέρει εργασίες προς επεξεργασία επιλέγει την ομάδα από ζώνες ώρας (και επομένως από κόμβους) από όπου επιθυμεί να ξεκινήσει ο υπολογισμός του.
- Ο scheduler επιλέγει τυχαία έναν αριθμό από hosts και τους στέλνει αίτηση παροχής πόρων. Εκείνοι αναζητούν περισσότερους υποψήφιους. Κατόπιν επιλέγει σύμφωνα με κριτήρια διαθεσιμότητας και ισχύος τον καλύτερο και του δρομολογεί την εργασία.

Για να αποφευχθεί η δρομολόγηση εργασιών του WaveGrid εις βάρος των τοπικών εργασιών του χρήστη, οι εργασίες δρομολογούνται μόνο όταν η χρήση του επεξεργαστή είναι μικρή.

Τα αποτελέσματα δείχνουν για το WaveGrid βελτιωμένες επιδόσεις ως προς τη σταθερότητα, τη μετανάστευση των εργασιών και το πλήθος των αναδρομολογήσεων. Αποδεικνύουν επίσης τη σημασία της εκμετάλλευσης του νυχτερινού χρόνου στα μηχανήματα που προσφέρουν υπολογιστική ισχύ.

## 2.3 Προηγούμενη έρευνα

Στο [2] οι Agrawal και Casanova παρουσιάζουν μία μέθοδο για την αναγνώριση των κοντινών clients. Καθώς οι ως τώρα προταθείσες λύσεις για το πρόβλημα ήταν αλγοριθμικά ασύμφορες, οι συγγραφείς προτείνουν μία λύση βασισμένη σε δύο ευριστικές μεθόδους για τη δημιουργία clusters από γειτονικούς clients και του αντίστοιχου γράφου που τα περιγράφει.

Σύμφωνα με την πρώτη ευριστική μέθοδο που προτείνεται, για κάθε client μετριέται η απόσταση από γνωστούς κόμβους - ορόσημα και κατασκευάζεται ένας χάρτης με τις αποστάσεις των clients από αυτούς. Κατόπιν, οι clients με απόσταση μεταξύ τους μικρότερη μιας μεταβλητής $D$, οριζόμενης από το χρήστη, ομαδοποιούνται. Για την ομαδοποίηση των clients χρησιμοποιούνται κάποιοι από αυτούς, οι λεγόμενοι markers, σαν τα κέντρα των διαφορετικών clusters. Οι markers απέχουν μεταξύ τους απόσταση μεγαλύτερη από $D$. Αφού επιλεγούν αρχικά οι markers, συγκεντρώνονται στα cluster τους όλοι οι clients. Κατόπιν ελέγχεται η διάμετρος του κάθε cluster και σε περίπτωση που υπερβαίνει σε κάποιο από αυτά το μέγεθος $D$ εφαρμόζεται για εκείνο το cluster ξανά ο αλγόριθμος ώσπου να το χωρίσει σε αποδεκτής διαμέτρου cluster. Στη δεύτερη ευριστική μέθοδο που χρησιμοποιείται ακολουθούνται οι ίδιες διαδικασίες μέχρι και την επιλογή των markers. Κατόπιν όλοι οι clients συγκεντρώνονται στον κοντινότερό τους marker. Στη συνέχεια υπολογίζεται το «κέντρο βάρους» κάθε cluster και αναδιανέμονται οι clients βάσει των αποστάσεών τους από αυτά. Η διαδικασία επαναλαμβάνεται μέχρι να συγκλίνουν τα clusters. Στο τέλος εξετάζεται για κάθε cluster αν η διάμετρος είναι μεγαλύτερη από $D$, περίπτωση κατά την οποία ο αλγόριθμος καλείται για να χωρίσει το συγκεκριμένο cluster σε μικρότερα.

Αυτή η λύση μπορεί να εφαρμοστεί μεταξύ άλλων σε peer-to-peer δίκτυα για την αποδοτική προώθηση αιτήσεων πάνω από αυτά. Επίσης μπορεί να χρησιμοποιηθεί σε εφαρμογές κατανεμημένου υπολογισμού αντιμετωπίζοντας ζητήματα του locality

των δεδομένων και καθιστώντας εφικτή την εκτέλεση μέρους του παράλληλου υπολογισμού μέσα σε μία ομάδα «κοντινών» clients, εκμεταλλευόμενες τον χαμηλό χρόνο απόκρισης μεταξύ τους, πράγμα που δεν θα ήταν εφικτό σε μία τυχαία ομάδα clients.

Στο [37], ο Luis Sarmenta εξέτασε μηχανισμούς ανοχής της εσκεμμένης επιστροφής λανθασμένων αποτελεσμάτων από κακοπροαίρετους εθελοντές σε ένα κατανεμημένο υπολογιστικό σύστημα. Σε αυτού του είδους τα ειθελημένα σφάλματα δεν μπορούν να ακολουθηθούν προσεγγίσεις βασισμένες σε ελέγχους parity και checksum, όπως εφαρμόζονται στα σφάλματα αστοχίας των κόμβων σε ένα παράλληλο υπολογισμό, αφού είναι δυνατή η ανακάλυψη του μηχανισμού checksum και η παραγωγή σωστών bit ελέγχου για λανθασμένα αποτελέσματα. Η κλασική μέθοδος της ψηφοφορίας, δηλαδή της ανάθεσης της ίδιας εργασίας σε πολλούς και η λήψη του πιο δημοφιλούς αποτελέσματος, μειώνει εκθετικά τα λάθη με τον αριθμό των ψηφοφόρων (redundancy). Αυτή η μέθοδος όμως απαιτεί τουλάχιστον διπλή εκτέλεση των εργασιών και δε λειτουργεί αποδοτικά όταν οι κακοπροαίρετοι χρήστες είναι πολλοί.

Ο Sarmenta παρουσιάζει μία νέα τεχνική, ονομαζόμενη spot-checking που μειώνει τα σφάλματα γραμμικά με τον όγκο της δουλειάς, επιφέροντας μόνο μια μικρή επιβάρυνση στον αρχικό χρόνο. Σύμφωνα με την τεχνική αυτή, ο master δίνει δειγματοληπτικά στον worker μία εργασία που είναι ήδη γνωστό το σωστό αποτέλεσμα ή είναι εύκολα ελέγξιμο μετά τον υπολογισμο. Σε περίπτωση επιστροφής λανθασμένου αποτελέσματος, ο master ψάχνει προς τα πίσω όλα τα αποτελέσματα που έχει δώσει ο συγκεκριμένος worker και τα ακυρώνει.

Τέλος, εισάγεται ένας μηχανισμός ανοχής λαθών βασισμένος στην αξιοπιστία των κόμβων, που χρησιμοποιεί εκτίμηση πιθανότητας για τον περιορισμό του redundancy και τη σωστή δειγματοληπτική χρήση του. Συνδυαζόμενοι οι παραπάνω δύο μηχανισμοί με την τεχνική της δειγματοληψίας βάσει της αξιοπιστίας και κατόπιν μεταξύ τους μας δίνουν τη δυνατότητα να μειώσουμε εκθετικά ένα ήδη γραμμικά μειωμένο ποσοστό σφαλμάτων, με τρόπο αποδοτικό και με χαμηλό κόστος.

Τα πειράματα που ακολούθησαν [35] έγιναν χρησιμοποιώντας εφαρμογές παραμετρικής ανάλυσης για προσομοιώσεις Monte Carlo. Οι εφαρμογές παραμετρικής ανάλυσης απαιτούν επαναλαμβανόμενες ανεξάρτητες εκτελέσεις του ίδιου σειριακού αλγορίθμου για πολλά διαφορετικά σετ παραμέτρων εισόδου. Οι προσομοιώσεις Monte Carlo είναι τέτοιες εφαρμογές, με τα σετ των παραμέτρων εισόδου να είναι τυχαία παραγόμενα και όχι προκαθορισμένα. Τα αποτελέσματα δείχνουν ότι με διπλάσια ή τριπλάσια περίπου επιβάρυνση του συνολικού χρόνου εκτέλεσης μπορεί να επιτευχθεί η μείωση του ποσοστού λαθών κατά αρκετές τάξεις μεγέθους.

Στη Γαλλία, στο Πανεπιστήμιο του Paris Sud οι Aurelien Bouteiller, Franck Cappello και άλλοι ανέπτυξαν το 2002 την πλατφόρμα MPICH-V [9], ένα περιβάλλον που χρησιμοποιεί Message Passing (MPI) πάνω σε μεγάλα clusters υπολογιστών. Ο σκοπός της ανάπτυξης αυτής της πλατφόρμας ήταν η αντιμετώπιση των αστοχιών των κόμβων σε αυτά τα μεγάλης κλίμακας παράλληλα και κατανεμημένα συστήματα, οι οποίες δεν είναι καθόλου αμελητέες και επηρεάζουν αρνητικά την απόδοση ολόκληρου του συστήματος. Μια ακόμα ανάγκη που προσπάθησε να καλύψει το περιβάλλον αυτό ήταν η έλλειψη υποστήριξης επικοινωνίας μεταξύ των κόμβων σε μεγάλης κλίμακας κατανεμημένα συστήματα.

Τα βασικά χαρακτηριστικά του MPICH-V είναι τα εξής: Χρησιμοποιεί το standard του MPI ώστε να μπορούν να εκτελεστούν έτοιμες παράλληλες εφαρμογές που το χρησιμοποιούν, χωρίς μετατροπές. Επίσης, ανέχεται πολλά ταυτόχρονα σφάλματα κόμβων, μέχρι και $n$ (όπου $n$ το πλήθος των διεργασιών MPI), χρησιμοποιώντας γνωστές μεθόδους redundancy και checkpoint/restart. Το πρωτόκολλο checkpoint/restart είναι υλοποιημένο κατανεμημένα και ασύγχρονα για να παρέχει ανοχή στην αστοχία των κόμβων κατά τη διάρκεια του συγχρονισμού και για να μην επιφέρει επιβάρυνση στην απόδοση του συστήματος. Ως προς την επικοινωνία, το MPICH-V έχει τη δυνατότητα να παρακάμψει τα firewalls. Τέλος, η αρχιτεκτονική του παρέχει κλιμάκωση και περιλαμβάνει μόνο βιβλιοθήκες στο επίπεδο του χρήστη.

Το σύστημα αποτελείται από τις εξής οντότητες: τον dispatcher, τα Channel memories, τους Checkpoint servers και τους κόμβους υπολογισμού και επικοινωνίας.

Τα Channel memories είναι κόμβοι που διοχετεύουν και αποθηκέυουν τα μηνύματα. Χρησιμοποιούνται για να εξασφαλιστεί η παράκαμψη του εμποδίου των firewalls και η ανοχή στα σφάλματα του συστήματος. Κάθε μήνυμα μεταξύ των κόμβων υπολογισμού περνάει μέσα από αυτούς. Η ανοχή στα σφάλματα εξασφαλίζεται με αποθήκευση των context του υπολογισμού (από τους Checkpoint servers) και της επικοινωνίας (από τα Channel memories) κάθε κόμβου ανά τακτά χρονικά διαστήματα με αποκεντρωμένο και ασύγχρονο τρόπο. Όταν ένας κόμβος πάρει μία εργασία που είχε διακοπεί, ξεκινά από το context του υπολογισμού που παίρνει από τον Checkpoint server και αρχίζει να δέχεται το τελευταίο μήνυμα που στάλθηκε στον κόμβο που διακόπηκε καθώς και τον αριθμό των μηνυμάτων που είχε στείλει ο κόμβος που διακόπηκε.

Ο Dispatcher αρχικά μοιράζει ένα σύνολο εργασιών στους συμμετέχοντες κόμβους. Στη διάρκεια του υπολογισμού συντονίζει όλους τους πόρους για την εκτέλεση ενός παράλληλου προγράμματος, αναθέτει σε κάθε νέο κόμβο έναν Channel memory server και έναν Checkpoint server, επιβεβαιώνει ανά τακτά διαστήματα αν ο κάθε κόμβος είναι ζωντανός και σε περίπτωση σφάλματος αναθέτει και πάλι την εργασία που διακόπηκε.

Η ανάπτυξη αυτών των πρωτοκόλλων βασίστηκε και δοκιμάστηκε στην υλοποίηση του XtremWeb, της ίδιας ομάδας, που περιγράφθηκε παραπάνω. Η απόδοση του συστήματος δείχνει ότι με την ύπαρξη κόμβων που αποτυγχάνουν ένας ανα 110 sec και με τη χρήση checkpoint/restartμηχανισμών μειώνεται λιγότερο από 2 φορές κάτι που θεωρείται ικανοποιητικό.

Το 2003 η ίδια ομάδα παρουσίασε το MPICH-V2, τη δεύτερη έκδοση του πρωτοκόλλου MPICH-V[11]. Σε αυτό το πρωτόκολλο οι προσπάθειες επικεντρώθηκαν στην ανοχή σε ακόμη περισσότερα σφάλματα χρησιμοποιώντας καταγραφή μηνυμάτων (message logging) από την πλευρά του αποστολέα. Τα κεντρικά χαρακτηριστικά του MPICH-V παραμένουν και εδώ αναλλοίωτα: μη συντονισμένα checkpoints, κατανεμημένη καταγραφή μηνυμάτων και χρήση αξιόπιστων server για συντονισμό και διατήρηση των checkpoints. Επιπλέον χρησιμοποιήθηκαν pessimistic πρωτόκολλα για την καταγραφή των μηνυμάτων. Τα pessimistic πρωτόκολλα εγγυώνται ότι κάθε μήνυμα που δέχεται μια διεργασία καταγράφεται σε αξιόπιστο μέσο πριν ακόμα η διεργασία αποστείλει δεδομένα στο σύστημα. Αυτό το πρωτόκολλο, σε αντίθεση με τα optimistic πρωτόκολλα που που φροντίζουν απλά και μόνο για την

καταγραφή των μηνυμάτων σε κάποιο όχι απαραίτητα αξιόπιστο μέσον, διασφαλίζει μεγάλη ανοχή σε σφάλματα.

Επιδίωξη της δεύτερης έκδοσης του MPICH-V είναι να αυξήσει την αποδοτικότητά του αφαιρώντας την αναγκαιότητα των Channel memories που επιφέρουν μια μείωση του bandwidth στο μισό. Αυτό επιτυγχάνεται με τη διατήρηση των μηνυμάτων τοπικά σε κάθε αποστολέα: Όταν ο αποστολέας του μηνύματος στέλνει το μήνυμα στον παραλήπτη, το κρατάει και τοπικά. Ο παραλήπτης δεν μπορεί να κάνει τίποτα προτού καταγράψει το id του μηνύματος σε ένα αξιόπιστο μέσον. Αν ο παραλήπτης παρουσιάσει πρόβλημα ξαναξεκινά την εργασία ζητώντας από τους υπόλοιπους να του ξαναστείλουν τα μηνύματά τους που έχουν ήδη αποθηκεύσει, βάσει των id των μηνυμάτων του που έχουν σωθεί. Σε περίπτωση που και ο παραλήπτης και ο αποστολέας παρουσιάσουν πρόβλημα, θα πρέπει να επανέλθουν στο τελευταίο checkpoint. Επειδή ενδεχομένως να χρειάζονται ο ένας μηνύματα του άλλου, πράγμα που μπορεί να οδηγήσει σε διαδοχικά rollbacks και των δύο, σε κάθε checkpoint σώζεται και ένα αντίγραφο με τα μηνύματα του κάθε αποστολέα. Έτσι δεν είναι πλέον απαραίτητη η χρήση ενδιάμεσου Channel memory server, η οποία επέφερε σημαντική επιβάρυνση στην απόδοση του συστήματος, αλλά και στις απαιτήσεις για αξιόπιστους servers που πλέον μειώνονται σημαντικά.

Η επιλογή του μη συντονισμένου checkpoint στις δύο εκδόσεις του MPICH-V σε αντιδιαστολή με το συντονισμένο, υπαγορεύεται από τις αρχές που φαίνονται και στα πειραματικά αποτελέσματα στο [14], όπου συγκρίνονται το MPICH-V2 με μία έκδοση του MPICH όπου εφαρμόζεται μια global στρατηγική για checkpoint βασισμένη στον αλγόριθμο Chandy-Lamport. Οι δύο τεχνικές φάνηκε να έχουν παρόμοιες επιδόσεις εκτός από την περίπτωση της ανάνηψης (recovery) μετά από σφάλμα. Εκεί η τεχνική του μη συντονισμένου checkpoint μέσω message logging παρέχει καλύτερα αποτελέσματα και επομένως καλύτερο επίπεδο ανοχής στα σφάλματα από το συντονισμένο checkpoint.

Παρολαυτά στο [13] επανέρχονται με βελτιωμένες εκδόσεις και για τις δύο τεχνικές, ξανασυγκρίνοντάς τες και βρίσκοντας σαφώς βελτιωμένες επιδόσεις της μεθόδου του συντονισμένου checkpoint σε σχέση με το message logging και σε σύγκριση με τα προηγούμενα πειράματα, αλλά και πάλι κατώτερες. Η βελτίωση στην τεχνική του message logging έγκειται στην εξής ιδέα: Μέχρι τώρα, για να διατηρηθεί η ιεραρχική σχέση μεταξύ των μηνυμάτων μεταξύ δύο κόμβων ώστε να αναπαραχθεί σωστά όταν χρειαστεί σε περίπτωση σφάλματος, από τη λήψη κάθε μηνύματος και πριν την καταχώρησή του σε αξιόπιστο μέσο αποθήκευσης, ο παραλήπτης δεν μπορούσε να επηρεάσει το σύστημα. Αυτό επέφερε καθυστέρηση, η οποία αναιρέθηκε αφαιρώντας αυτόν τον περιορισμό του παραλήπτη και προσθέτοντας επιπλέον πληροφορία μέσα στα μηνύματα. Αυτό όμως επιβάρυνε τη ενημέρωση και διατήρηση του αρχείου των ανταλλαγών μηνυμάτων (message event logging) αφού κάθε νέα καταχώρηση «έσερνε» ολοένα και περισσότερη πληροφορία και επιβάρυνε τη μνήμη του server. Ένας τρόπος επίλυσης αυτού του προβλήματος εξετάζεται στο [12] μέσω της αποθήκευσης των καταχωρήσεων σε σταθερό αποθηκευτικό χώρο που ονομάζεται Event Logger. Η χρήση Event Logger στα πειράματα απέδωσε σημαντικά, τέθηκαν όμως σοβαρά ζητήματα scalability όσο οι διεργασίες αυξάνονται και το φάσμα του bottleneck στον Event Logger είναι πιο ορατό. Η χρήση πολλών Event Loggers και η διατήρηση των δεδομένων κατα-

νεμημένα σε αυτούς εγείρει αρκετά ζητήματα σχεδιασμού και αρχιτεκτονικής που πρέπει να επιλυθούν.

Η ίδια ομάδα εξέτασε στο [10] τη χρονοδρομολόγηση εφαρμογών MPI στα Grids. Προσπάθησαν να επιτύχουν χρονοδρομολόγηση των παράλληλων εφαρμογών πάνω από ένα Grid ούτως ώστε να είναι δυνατή η ταυτόχρονη εκτέλεση πολλών διαφορετικών εφαρμογών. Για το σκοπό αυτό χρησιμοποίησαν ένα υβρίδιο από τις δύο πιο γνωστές τεχνικές για διαμοιρασμό των πόρων ενός cluster: Το co-scheduling, που βασίζεται στο λειτουργικό σύστημα κάθε κόμβου για να δρομολογηθούν αποδοτικά οι διεργασίες και το gang scheduling, που εξασφαλίζει την ταυτόχρονη δρομολόγηση μίας εφαρμογής σε όλους τους κόμβους. Καθώς το co-scheduling είναι πιο αποδοτικό από το gang scheduling όσο η φυσική μνήμη του κόμβου δεν έχει εξαντληθεί, χρησιμοποιήθηκε για αυτές τις περιπτώσεις το co-scheduling και για τις υπόλοιπες το gang scheduling, ώστε να επωφεληθεί το σύστημα από τα πλεονεκτήματα και των δύο τεχνικών. Η εφαρμογή των τριών τεχνικών έγινε σε συστήματα που υποστήριζαν το MPICH-V που περιγράψαμε παραπάνω και χρησιμοποιούσαν τεχνικές checkpoint/restart για τη διαχείριση της μνήμης. Τα αποτελέσματα δείχνουν ότι η τεχνική είναι εξίσου αποδοτική με το co-scheduling όσο η μνήμη του κόμβου δεν έχει εξαντληθεί και καλύτερη και από τις δύο προαναφερθείσες μεθόδους όταν εξαντληθεί.

Το P2P-RPC είναι ένα απλό Application Programming Interface για Remote Procedure Calls για προγραμματισμό εφαρμογών πάνω από συστήματα peer-to-peer και παρουσιάστηκε το 2003 από τον Samir Djilali[19]. Βασικό χαρακτηριστικό του είναι ο μηχανισμός μετανάστευσης (migration) εφαρμογών που χρησιμοποιούν Remote Procedure Calls σε peer-to-peer συστήματα. Το πλεονέκτημα της χρήσης του RPC είναι η δυνατότητα επικοινωνίας μεταξύ κατανεμημένων αντικειμένων. Το σύστημα υλοποιήθηκε πάνω από την πλατφόρμα XtremWeb και εξομοιώνει την κλήση μιας Remote Procedure Call πάνω από την πλατφόρμα αυτή. Οι παροχείς υποβάλλουν στον server την εργασία και τα inputs, αυτός τη δρομολογεί και τη μοιράζει στους workers και αυτοί την εκτελούν επιστρέφοντας τα αποτελέσματα στο server.

Στο [22] οι Konstantin Kreyman, David Lorge Parnas και Sanzheng Qiao εξετάζουν μεθόδους επιβεβαίωσης της ακρίβειας και αξιοπιστίας προγραμμάτων που υπολογίζουν φυσικά φαινόμενα. Σε ορισμένα προγράμματα εξομοίωσης, η ακρίβεια και η ορθότητα των υπολογισμών είναι κρίσιμη (σε αντίθεση για παράδειγμα με το SETI@home). Προγράμματα ελέγχου πυρηνικών, χημικών ή άλλων μεγάλων εργοστασίων αναλαμβάνουν πολλές φορές να προβλέψουν κατά τον σχεδιασμό ενδεχόμενη αστοχία υλικών, ή προβλήματα ασφάλειας που μπορεί να στοιχίσουν ανθρώπινες ζωές. Η μέθοδος ελέγχου των αποτελεσμάτων που προτείνεται περιλαμβάνει μεταξύ άλλων:

- τον χωρισμό των μοντέλων σε φάσεις και την εξέταση των τιμών που παράγονται σε καθεμία από αυτές ούτως ώστε να εμπίπτει σε αποδεκτό φάσμα τιμών.
- εξέταση των ακραίων περιπτώσεων (αρχικοποίηση, όρια κλπ.)

Οι μέθοδοι που προτείνονται μπορούν να συμπεριληφθούν σε αντίστοιχα project κατανεμημένου υπολογισμού για διασφάλιση της ορθότητας των αποτελεσμάτων

που επιστρέφει ένας worker είτε λόγω σφάλματος του λογισμικού είτε λόγω κακής πρόθεσης του χρήστη.

Ο Noam Nisan εξετάζει στο [30] μηχανισμούς αποτροπής της μη συμμόρφωσης με τον προβλεπόμενο τρόπο λειτουργίας των συμμετεχόντων σε ένα κατανεμημένο υπολογισμό. Η ανάγκη, όπως υπογραμμίζει, έγκειται στο ότι οι χρήστες που προσφέρουν τους υπολογιστές τους πάνω από μία μη ελεγχόμενη πλατφόρμα όπως είναι το Internet δεν μπορούν να θεωρηθούν αξιόπιστοι και ενδεχομένως να δρουν με προσωπικά τους κίνητρα, διαφορετικά από αυτά του προγράμματος στο οποίο συμμετέχουν. Για αυτό, οι εφαρμογές που χρησιμοποιούν πόρους που προσφέρονται εθελοντικά πρέπει να ενσωματώσουν μηχανισμούς προσφοράς κινήτρων για συμμετοχή στους εθελοντές. Ο συγγραφέας δίνει τρία παραδείγματα τέτοιων εφαρμογών: resource allocation πάνω από το Internet, routing που απαιτεί μεγάλο bandwidth και διαμοιρασμό πληροφοριών για ηλεκτρονικό εμπόριο. Στο paper προτείνονται μαθηματικοί αλγόριθμοι για την δίκαιη και όσο το δυνατόν προς το κοινό συμφέρον αίτηση χρήσης κοινών πόρων από πολλούς clients, τη δίκαιη και συμφέρουσα και για τις δύο πλευρές, client και server, πλειοδοσία για τη χρήση υπηρεσιών ενός server από έναν client (ανάμεσα σε πολλούς), την εύρεση φθηνότερων μονοπατιών πληροφοριών από κόμβους που διαθέτουν ιδιωτικές πληροφορίες που θέλουμε να μοιραστούν, τη βέλτιστη ανάθεση διαφορετικού βάρους tasks σε διαφορετικής ισχύος servers και τη σωστή διεξαγωγή μυστικών δημοπρασιών ανάμεσα σε αγνώστους συμμετέχοντες. Πολλοί από τους παραπάνω μηχανισμούς έχουν άμεση εφαρμογή και αποτελούν σημαντικό θεωρητικό υπόβαθρο για το σχεδιασμό κατανεμημένων υπολογιστικών συστημάτων που βασίζονται σε εθελοντική προσφορά συμμετεχόντων με ενδεχομένως εγωιστικά κίνητρα.

Το POPCORN [34], που δημιούργησαν οι Ori Regev και Noam Nisan του Πανεπιστημίου της Ιερουσαλήμ, είναι ένα κατανεμημένο σύστημα διαμοιρασμού υπολογιστικής ισχύος που περιλαμβάνει την αντικειμενική μέτρηση της υπολογιστικής ισχύος που προσφέρει κάθε χρήστης, το διαμοιρασμό της σε άλλους χρήστες αλλά και ένα μηχανισμό δημοπράτησης που παρεμβάλλεται ανάμεσα στους προσφέροντες και τους αιτούντες για την οικονομικά συμφέρουσα συναλλαγή υπολογιστικών πόρων και για τις δύο πλευρές. Η εθελοντική διάθεση για προσφορά δε μοιάζει να πείθει σαν ικανό κίνητρο τους δημιουργούς του POPCORN οι οποίοι προτείνουν την οικονομική αποζημίωση σαν πιο αξιόπιστη βάση συναλλαγής υπολογιστικής ισχύος. Το POPCORN είναι ένα σύστημα που υποστηρίζει αυτού του είδους τη συναλλαγή.

Η αρχιτεκτονική του στηρίζεται στις αρχές των περισσότερων προαναφερθέντων project. Οι παράλληλες εφαρμογές που εκτελούνται στο σύστημα αυτό είναι Java applets και λειτουργούν σαν «αγοραστές» χρόνου επεξεργασίας. Ο «πωλητής» χρόνου δεν έχει παρά να επισκεφθεί μία σελίδα με τον web-browser του για να ξεκινήσει η διαδικασία. Τέλος, η οντότητα «market» έχει ως αποστολή να φέρει σε επαφή αυτόματα και δυναμικά τις δύο πλευρές, σύμφωνα με οικονομικούς μηχανισμούς, χρεώνοντας τον αγοραστή για το χρόνο που πήρε. Το μέγεθος που έχει υιοθετηθεί από το σύστημα σαν αντικειμενικά μετρούμενο και αποδεκτό ανταλλάξιμο μέγεθος είναι τα JOPs (Java Operations, κατά το FLOPS). Η αξία του υπολογισμού κάθε εργασίας είναι ανάλογη των JOPs που χρειάστηκε για την εκτέλεσή της. Ο τρόπος υπολογισμού (για την ακρίβεια: εκτίμησης) του αριθμού των

JOPs είναι μέσω ενός απλού προγράμματος μέτρησης που συμπεριλαμβάνεται σε κάθε εργασία που αποστέλλεται. Αυτή η μέθοδος μέτρησης είναι αμφισβητίσημης ακρίβειας αφού όχι μόνο φέρνει υπολογιστική επιβάρυνση στον ίδιο τον «πωλητή», αλλά είναι και διαβλητή ως προς την αξιοπιστία της, αφού τρέχει στο μηχάνημα εκείνου που έχει συμφέρον από αυτήν. Για αυτό προτείνεται μια δεύτερη μέθοδος μέτρησης, πιο χοντρική, μέσω του πλήθους εργασιών που διεκπεραιώνει ο «πωλητής». Σε κάθε περίπτωση το παζάρι (sic) διεξάγεται ως εξής: Ο αγοραστής προτείνει μία τιμή προσφοράς ανά JOP ή ανά εργασία και ο πωλητής επιλέγει την εφαρμογή που επιθυμεί. Από εκείνη τη στιγμή ξεκινά η συμμετοχή του πωλητή στην εφαρμογή. Από την πλευρά του market, έρχονται σε επαφή πωλητές και αγοραστές, μεταφέρονται οι εργασίες και τα αποτελέσματά τους, διατηρούνται τα στατιστικά και διεκπεραιώνονται οι πληρωμές. Αυτό φέρνει στον server αξιόλογη επιβάρυνση, πράγμα που θεωρείται όμως αμελητέο από τους συγγραφείς σε σχέση με τον σημαντικά μεγαλύτερο χρόνο υπολογισμού των εργασιών που εκτελείται κατανεμημένα. Το market αναλαμβάνει με οικονομικά κριτήρια να εκτελέσει αυτόματα τη συναλλαγή. Αυτό επιτυγχάνεται βάσει δύο αρχών:

- Οι συναλλαγές γίνονται με οικονομικά αποδεκτό τρόπο, δηλαδή μεγιστοποιούν τη σφαιρική αξία των συναλλαγών δίνοντας υπολογιστικό χρόνο σε αυτόν που πραγματικά τον χρειάζεται περισσότερο. Αυτό εξασφαλίζεται με το παρακάτω:

- Ο μηχανισμός της αγοράς πρέπει να δίνει κίνητρο σε καθένα να αποκαλύψει πόση αξία πραγματικά έχει για αυτόν ο χρόνος που αγοράζει ή πουλάει (ούτε μικρότερη ούτε μεγαλύτερη).

Η διασφάλιση των αρχών αυτών επιτυγχάνεται με τους μηχανισμούς που περιέγραψε ο Nisan στην προαναφερθείσα του μελέτη.

Στο [21] οι Chris Kenyon και Giorgos Cheliotis εξετάζουν τη δημιουργία υπηρεσιών με εγγυήσεις για τα αποτελέσματά τους, σε συστήματα προσφοράς υπολογιστικής ισχύος που εφαρμόζονται από αρκετές εταιρίες πάνω σε Grid αρχιτεκτονικές ώστε να είναι δυνατή η δημιουργία «συμβολαίων» που θα είναι αποδεκτά αμφίπλευρα για την παροχή αυτών των υπηρεσιών. Οι μέθοδοι που εξετάζονται δεν βασίζονται στο reputation του παροχέα των υπηρεσιών αλλά στην ίδια την ποιότητα της υπηρεσίας ώστε κάθε συμβόλαιο να αντικατοπτρίζει πραγματικά συναλλάξιμη αξία. Έτσι υιοθετήθηκαν στατιστικά metrics που εγγυώνται αυστηρά (hard guarantee) την ποιότητα των υπηρεσιών ξεχωριστά. Αυτή η προσέγγιση μπορεί να εφαρμοστεί άμεσα για να εξασφαλίσει την ποιότητα παροχής υπολογιστικής ισχύος από τους χρήστες και τις εταιρίες και ενδεχομένως την προσφορά αυτής για οικονομικό όφελος των χρηστών, αφού πλέον μπορεί να διασφαλιστεί και προς τις δύο πλευρές, αντικειμενικά και μετρήσιμα η συνεισφορά.

Το ζήτημα του ελέγχου της ορθότητας των αποτελεσμάτων για την επίτευξη αξιοπιστίας των εφαρμογών εξετάζουν οι Hal Wasserman και Manuel Blum στο [39]. Προτείνουν μηχανισμούς υλοποίησης τέτοιων ελέγχων κατά το χρόνο εκτέλεσης των προγραμμάτων, κάτι που επιτρέπει την ενσωμάτωσή τους σε αυτά προσφέροντας αξιοπιστία αλλά και ένα βοήθημα για αποδοτική αποσφαλμάτωση. Οι ως τώρα

μηχανισμοί αποσφαλμάτωσης των προγραμμάτων κρίνονται ανεπαρκείς αφού δοκιμάζουν ένα πολύ στενό σε εύρος και ενδεχομένως μη αντιπροσωπευτικό δείγμα των δυνατών παραμέτρων εισόδου. Επίσης, η ορθότητα των αποτελεσμάτων πολύ συχνά συγκρίνεται με αποτελέσματα παλαιότερων προγραμμάτων που αυθαίρετα αποδεχόμαστε ως αξιόπιστα. Δύο προσεγγίσεις μπορούν να ακολουθηθούν: είτε απαιτούμε αυστηρά ορθή λειτουργία του προγράμματος, κάτι για το οποίο όμως είναι πολύ δύσκολο να παρέχουμε μηχανισμούς απόδειξης, είτε δημιουργούμε περισσότερες της μίας υλοποιήσεις και κάθε φορά τις εκτελούμε συγκρίνοντας τα αποτελέσματά τους, πράγμα που είναι υπερβολικά απαιτητικό σε ανθρώπινους και υπολογιστικούς πόρους. Μια τρίτη προσέγγιση είναι περισσότερο υποσχόμενη, αυτή του ελέγχου των λαθών και της αυτόματης διόρθωσής τους, υποστηριζόμενη από ισχυρές μαθηματικές εγγυήσεις, που έχει εφαρμοστεί επιτυχώς στις επικοινωνίες. Αυτή η προσέγγιση απαιτεί την ανάπτυξη συναρτήσεων ελέγχου, οι οποίες είναι πολύ πιο εύκολο να διαπιστώσουν την ορθότητα του υπολογισμού, από ότι είναι η ίδια η διεκπεραίωση του υπολογισμού. Ο μηχανισμός που προτείνεται βασίζεται στη χρήση συναρτήσεων ελέγχου αποτελεσμάτων που μπορούν να παράγουν, επεξεργάζονται και αποθηκεύουν τυχαία bits πριν την εκτέλεση του προγράμματος και κατόπιν να χρησιμοποιούν αυτή την πληροφορία σε μία σειρά από ελέγχους κατά την εκτέλεσή τους.

Η σημασία αυτών των μηχανισμών για τον έλεγχο της ορθότητας των αποτελεσμάτων σε έναν κατανεμημένο υπολογισμό είναι προφανής: Οι εθελοντές παροχείς υπολογιστικής ισχύος είναι αναξιόπιστοι και ενδεχομένως, για λόγους προσωπικού συμφέροντος, γοήτρου ή κακής προαίρεσης να παράγουν λανθασμένα αποτελέσματα στο μηχάνημά τους. Ένας τέτοιος μηχανισμός ελέγχου των αποτελεσμάτων είναι πολύτιμος για τη διασφάλιση της ακεραιότητας και της ορθότητας ολόκληρου του υπολογισμού.

## 2.4 Ανοιχτά ζητήματα

Βλέποντας συνολικά την περιοχή του κατανεμημένου υπολογισμού, παρατηρούμε ότι έχει γίνει μια αξιόλογη έρευνα, κινητοποιούμενη αρχικά συνήθως από κάποια προϋπάρχουσα επιστημονική ανάγκη. Είτε είναι η ανάλυση ενός σήματος από τηλεσκόπιο, είτε είναι κάποιο πείραμα στη μοριακή βιολογία, είτε η εξομοίωση κάποιων μοντέλων, η κινητήρια ανάγκη των εξελίξεων στην περιοχή ήταν σχεδόν πάντα η μεταφορά ενός ήδη υπάρχοντος ή εύκολα αναπτυσσόμενου παράλληλου αλγόριθμου για εκτέλεση σε πολλούς επεξεργαστές. Αυτό είχε ως αποτέλεσμα τα προβλήματα που ζητούν επίλυση από τα συστήματα κατανεμημένου υπολογισμού να έχουν ήδη αυτολογοκριθεί πριν ζητήσουν τη βοήθεια του κατανεμημένου υπολογισμού, να εμπίπτουν από μόνα τους δηλαδή στην κατηγορία των εύκολα παραλληλίσιμων προβλημάτων. Τα περισσότερα συστήματα που αναπτύχθηκαν, αρχικά υποστήριζαν μόνο μία εφαρμογή και ήταν αναπόσπαστα συνδεδεμένα με αυτήν. Σιγά σιγά εμφανίστηκαν πλατφόρμες υπολογισμού που είχαν τη δυνατότητα να εκτελούν διαφορετικές εφαρμογές. Παρόλαυτα το είδος των εφαρμογών δεν άλλαξε και αφορούσε εφαρμογές χώρις πολλή επικοινωνία, με πολύ υπολογιστικό χρόνο και δομή master-worker όπου ο κάθε worker επικοινωνεί αποκλειστικά με

τον master για να λάβει εργασίες και δεδομένα και να επιστρέψει αποτελέσματα.

Πέρα από αυτές τις εξαιρετικά βολικές για παραλληλισμό εφαρμογές, υπάρχουν πολλές άλλες που δεν είναι τόσο «τυχερές». Ο μεγάλος όγκος επικοινωνίας σε σχέση με τον υπολογισμό είναι ένα χαρακτηριστικό αυτών των εφαρμογών που τις καθιστούν ασύμφορες να εκτελεστούν σε ένα σύστημα της μορφής master-worker. Είναι λοιπόν εμφανής η ανάγκη για ανάπτυξη συστημάτων διαφορετικής νοοτροπίας που θα αποκεντρώνουν και θα κατανείμουν την επικοινωνία, επιτρέποντας στους workers να επικοινωνήσουν μεταξύ τους.

Ένα άλλο σημαντικό ζήτημα είναι η συμπεριφορά των δικτύων υπολογισμού κάτω από αντίξοες συνθήκες συμμετοχής των εθελοντών, όταν δηλαδή η πιθανότητα απότομης αποσύνδεσής τους είναι μεγάλη.

Τέλος, υπάρχουν πολλά ακόμη ανοιχτά ζητήματα στην περιοχή, τα οποία παρουσιάζουν ενδιαφέρον, όπως για παράδειγμα ο έλεγχος των λαθών και της κακοπροαίρετης συμπεριφοράς των worker. Η δουλειά που γίνεται σε αυτό τον τομέα είναι πολύ σημαντική ειδικά όταν ασχολούμαστε με κατανεμημένο υπολογισμό πάνω από ανοιχτά δίκτυα όπως το Internet, όπου οι συμμετέχοντες δεν είναι απαραίτητα αξιόπιστοι.

# Κεφάλαιο 3

# Το Πρότυπο **Trantor-Terminus**

Στο κεφάλαιο αυτό θα παρουσιάσουμε τη δική μας προσέγγιση για το ζήτημα του κατανεμημένου υπολογισμού με τη χρήση εθελοντικής προσφοράς υπολογιστικής ισχύος. Στην §3.1 θα παρουσιάσουμε τις ανάγκες που μας ώθησαν να ακολουθήσουμε μια διαφορετική προσέγγιση. Στην §3.2 θα εξεταστούν ζητήματα σχεδίασης και αποφάσεις που λήφθησαν για να εξυπηρετηθούν οι παραπάνω ανάγκες. Στην §3.3 θα περιγραφούν οι τεχνολογίες και τα εργαλεία που αποφασίσαμε να χρησιμοποιήσουμε. Στην §3.4 θα περιγραφεί η λειτουργία του συστήματός μας. Στην §3.5 θα κάνουμε έναν σχολιασμό σε σημαντικά σημεία του συστήματος και θα παραθέσουμε προτάσεις για περαιτέρω βελτιώσεις.

## 3.1 Η Ανάγκη

Όπως εξετάσαμε στο κεφάλαιο 2 όλες σχεδόν οι εφαρμογές και οι πλατφόρμες που έχουν αναπτυχθεί στην περιοχή επικεντρώνονται στον υπολογισμό μιας ειδικής κατηγορίας παράλληλων προβλημάτων. Πρόκειται για embarrassingly parallel προβλήματα, που εκτελούνται πάνω από συστήματα τύπου master-worker, όπου η επικοινωνία είναι πολύ μικρή και γίνεται μόνο μεταξύ του worker και του server ο οποίος συγκεντρώνει και διανέμει τα δεδομένα εισόδου και εξόδου.

Το κίνητρο μας λοιπόν, όπως γίνεται συχνά στην έρευνα, ήταν η περιέργεια, η οποία εκφραζόταν από την εξής απορία: Τι θα γινόταν αν οι workers μιλούσαν άμεσα μεταξύ τους; Θα είχαμε καλύτερα αποτελέσματα και πόσο καλύτερα; Θα μπορούσαμε να επιλύσουμε αποδοτικά διαφορετικής κατηγορίας παράλληλα προβλήματα με αυτό τον τρόπο; Θα βελτίωνε τα bottleneck του server δίνοντας μεγαλύτερη δυνατότητα κλιμάκωσης στο σύστημα μια τέτοια αρχιτεκτονική;

Αυτά τα ερωτήματα μας ώθησαν να σχεδιάσουμε το σύστημα Trantor-Terminus και να υλοποιήσουμε ένα πρωτότυπο για να προσπαθήσουμε να ικανοποιήσουμε την περιέργειά μας. Παράλληλα το σύστημά μας θέλαμε να συγκεντρώνει και ορισμένα άλλα χαρακτηριστικά που θα το έκαναν πιο ελκυστικό:

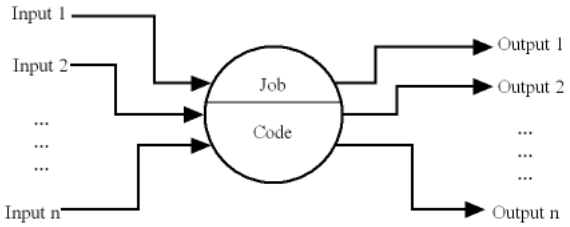
- Το σύστημα θα πρέπει να αποτελεί μια πλατφόρμα για επίλυση πολλών εφαρμογών και όχι ένα σύστημα υπολογισμού ενός συγκεκριμένου προβλήματος.

- Θέλουμε ένα σύστημα εύχρηστο που να χρησιμοποιεί γνωστές τεχνολογίες και να παρέχει ασφάλεια στο χρήστη.
- Η επικοινωνία worker-server θα πρέπει να είναι η ελάχιστη αναγκαία.

## 3.2 Σχεδίαση - τεχνολογίες

Με όλα τα παραπάνω υπόψη, σχεδιάστηκε το σύστημά μας. Σαν γλώσσα υλοποίησης επελέγη η Java για την υποστήριξη που παρέχει σε απομακρυσμένες κλήσεις, τον οντοκεντρικό της χαρακτήρα, την ευκολία και την τεκμηρίωση που διαθέτει καθώς και την ευρεία της αποδοχή. Για λόγους φορητότητας των εφαρμογών κάναμε την επιλογή οι εφαρμογές που θα υποστηρίζονται από το σύστημα να είναι κλάσεις Java και πάλι. Επίσης η Java παρέχει τους μηχανισμούς ασφάλειας που χρειαζόμαστε (sandboxing, Security Manager, policy files) ώστε να υπάρχει έλεγχος στον τρόπο με τον οποίο εκτελούνται οι εφαρμογές στην πλευρά του χρήστη. Αποφασίστηκε επίσης η χρήση HTTP για μεταφορά κώδικα και δεδομένων σαν εύκολος και προσιτός τρόπος μεταφοράς, ο οποίος επίσης παρακάμπτει τυχόν δικτυακά προβλήματα των workers σε περίπτωση ύπαρξης firewall.

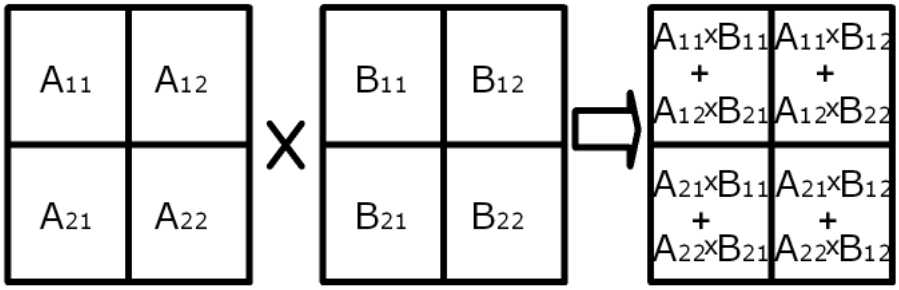
### 3.2.1 Η δομή των εφαρμογών - οι εργασίες




Σχήμα 3.1: Δομή εργασιών.

Μια παράλληλη εφαρμογή που μπορεί να εκτελέσει ο server αποτελείται από πολλές εργασίες (jobs). Κάθε εργασία έχει τη δομή του σχήματος 3.1. Οι εργασίες αποτελούνται από τον κώδικα που εκτελούν, τα δεδομένα εισόδου που μπορεί να είναι από 0 ως απεριόριστα και τα δεδομένα εξόδου τους που μπορεί να είναι από 1 ως απεριόριστα. Η κάθε εργασία σαν οντότητα πρέπει να είναι εσωτερικά ανεξάρτητη και αυτόνομη. Δεν επιτρέπουμε την επικοινωνία των εργασιών κατά τη διάρκεια διεκπεραίωσής τους, παρά μόνο στην αρχή και στο τέλος τους. Αν λοιπόν χρειάζεται για την εκτέλεση μιας εφαρμογής μία εργασία να ανταλλάξει πληροφορίες κατά τη διάρκεια της επεξεργασίας της, τότε θα πρέπει να χωρίσουμε αυτή την εργασία σε επιμέρους εργασίες ώστε το στάδιο της ανταλλαγής πληροφοριών να βρίσκεται στην αρχή ή στο τέλος μιας επιμέρους εργασίας. Έτσι ο υπολογισμός απλουστεύεται στα στοιχειώδη ανεξάρτητα στάδιά του και η επικοινωνία των βημάτων μοντελοποιείται εύκολα στις εισόδους και τις εξόδους.

Ορισμένες από τις εργασίες μπορούν να εκτελεστούν παράλληλα. Κάποιες άλλες ενδεχομένως να χρειάζονται σαν είσοδο αποτελέσματα που έχουν προκύψει από την έξοδο άλλων εργασιών. Στη γενική περίπτωση που συναντάμε πολύ συχνά, μια εφαρμογή αποτελείται από βήματα υπολογισμού στο εσωτερικό των οποίων οι εργασίες είναι ανεξάρτητες μεταξύ τους και επομένως παραλληλίσιμες. Οι εργασίες επόμενων βημάτων συχνά χρειάζονται αποτελέσματα εργασιών προηγούμενων βημάτων.

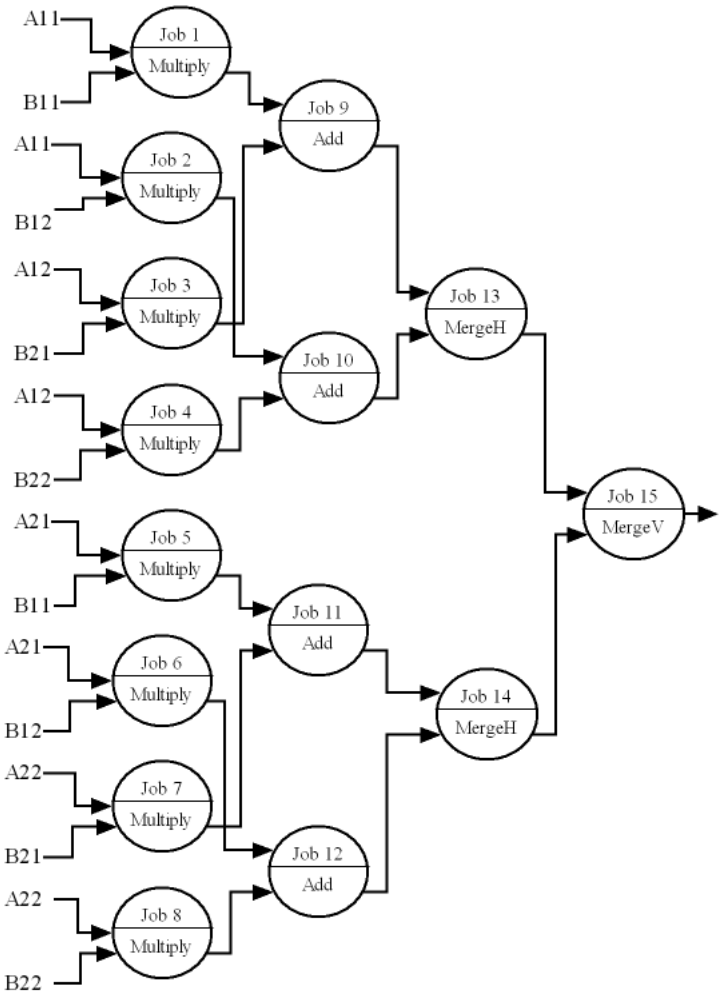



Σχήμα 3.2: Πολλαπλασιασμός πινάκων με 2x2 υποπίνακες.

Ένα παράδειγμα, αυτό του πολλαπλασιασμού πινάκων, παρατίθεται στα σχήματα 3.2 και 3.3. Πρόκειται για πολλαπλασιασμό πινάκων χωρισμένων σε 2x2 υποπίνακες ο καθένας. Στο σχήμα 3.2 βλέπουμε τους πίνακες A και B που θέλουμε να πολλαπλασιάσουμε, την κατάτμησή τους σε υποπίνακες καθώς και το αποτέλεσμα του πολλαπλασιασμού των πινάκων και πώς αυτό προκύπτει από τη σύνθεση και το άθροισμα των πολλαπλασιασμών των επιμέρους υποπινάκων. Το σχήμα 3.3 δείχνει τον γράφο που περιγράφει ολόκληρο τον υπολογισμό, όπου ο κώδικας Multiply κάνει τον πολλαπλασιασμό των υποπινάκων, ο Add την πρόσθεση και οι MergeH και MergeV την οριζόντια και κάθετη ένωση των υποπινάκων που προκύπτουν για να προκύψει ολόκληρος ο πίνακας.

### 3.2.2 Τρόπος διαμοιρασμού εργασιών

Ο server, έχοντας τον γράφο του υπολογισμού μπορεί να αρχίσει να μοιράζει τις εργασίες. Η μέθοδος δρομολόγησης των εργασιών που ακολουθείται κατά τον υπολογισμό είναι μία μορφή της μεθόδου που συναντάται στη βιβλιογραφία με τον όρο eager scheduling. Σύμφωνα με αυτό τον τρόπο δρομολόγησης των εργασιών, οι διαθέσιμες εργασίες συγκεντρώνονται στο work pool και ανατίθενται άμεσα στους workers που τις ζητούν. Οι γρήγοροι workers λαμβάνουν πιο συχνά εργασίες, οπότε σε γενικές γραμμές και μεγαλύτερο φορτίο δουλειάς. Επίσης, αν δεν υπάρχουν διαθέσιμες εργασίες που δεν έχουν δοθεί, επιλέγουμε να αναθέσουμε μία ήδη δοσμένη εργασία, αυτή που δόθηκε για τελευταία φορά πιο παλιά. Έτσι οι αργοί workers δεν καθυστερούν υπερβολικά την εξέλιξη του υπολογισμού και η μέθοδος αυτή διασφαλίζει μια απλή μορφή load balancing.




Σχήμα 3.3: Διάγραμμα πολλαπλασιασμού πινάκων με 2x2 υποπίνακες.

### 3.2.3 Ο Υπολογισμός

Το σύστημά μας υποστηρίζει δύο λειτουργίες, ριζικά διαφορετικές στη νοοτροπία κατανομής των δεδομένων. Πρόκειται για τις λειτουργίες που θέλουμε να συγκρίνουμε ως προς την απόδοσή τους και ονομάσαμε Trantor mode και Terminus mode και θα αναλύσουμε παρακάτω. Η διαφορά των δύο αυτών λειτουργιών έγκειται στον τρόπο επικοινωνίας μεταξύ worker-server και στον τρόπο λήψης των δεδομένων εισόδου και επιστροφής των δεδομένων εξόδου. Πέρα από αυτές τις διαφορές, τις οποίες και θα αναλύσουμε αργότερα, σε αυτή τη φάση θα περιγράψουμε τη διαδικασία του υπολογισμού και τη γενικότερη σχεδίαση του συστήματος.

Αναλυτικά η διαδικασία του υπολογισμού ακολουθεί τα εξής βήματα:

- Ο server ξεκινά και διαβάζει ένα αρχείο που περιγράφει όλες τις εργασίες που πρέπει να δρομολογήσει στους workers. Το αρχείο αυτό αποτελεί την περιγραφή του κατευθυνόμενου ακυκλικού γράφου (DAG) που αναπαριστά τα βήματα του υπολογισμού. Το αρχείο είναι ένα απλό αρχείο κειμένου με δομή που θα αναλύσουμε παρακάτω.

- Ο server χρησιμοποιεί τα δεδομένα του αρχείου που περιγράφει το DAG για να τα εισαγάγει σε μία βάση δεδομένων την οποία διατηρεί και ενημερώνει σε κάθε περίσταση και στην οποία κρατά τα δεδομένα που αφορούν τις εργασίες, τις εισόδους, τις εξόδους τους καθώς και ένα μητρώο των χρηστών που έχουν συνδεθεί με το σύστημα. Για τη βάση δεδομένων χρησιμοποιήσαμε μια σχετικά ελαφριά, αποδοτική και πολύ γνωστή βάση δεδομένων σε Java την HSQLDB. Το σχήμα των πινάκων που διατηρούμε σε αυτή θα περιγραφεί αργότερα.

- Ο worker καλεί τον server για μία εργασία: Αποφασίσαμε να χρησιμοποιήσουμε τη μέθοδο RMI για αυτό το σκοπό. Ο worker μέσω μιας κλήσης RMI καλεί μια κλάση στην πλευρά του server για να διεκπεραιωθεί η επικοινωνία μεταξύ τους.

- Ο server επιστρέφει τα στοιχεία της νέας εργασίας: Η κλήση RMI στην πλευρά του server αναζητά μια νέα εργασία που να είναι διαθέσιμη και να μην έχει δοθεί σε άλλο worker. Κατασκευάζει ένα αρχείο με μορφή xml που περιγράφει την εργασία (κώδικας, δεδομένα εισόδου-εξόδου) ώστε να το επιστρέψει στον worker. Η μορφή αυτού του αρχείου (job.xml) περιγράφεται παρακάτω.

- Ο worker διαβάζει και αναλύει το αρχείο: Για αυτό το σκοπό αποφασίστηκε η χρήση του Castor, ενός εργαλείου που κάνει αυτόματο parsing σε δεδομένα που περιγράφονται με μορφή xml. Έτσι ο worker έχει όλες τις λεπτομέρειες της εργασίας που πρέπει να εκτελέσει.

- Ο worker κατεβάζει τον κώδικα και τα δεδομένα εισόδου (αν χρειάζεται) μέσω HTTP τοπικά και εκτελεί το πρόγραμμα επίσης τοπικά, φορτώνοντας την κλάση μέσω του ClassLoader.

- Ο πίνακας που περιγράφει τα αποτελέσματα, είτε είναι inline είτε URL που δείχνει την τοποθεσία που είναι διαθέσιμα επιστρέφεται στον server μέσω της επόμενης κλήσης RMI σαν (προαιρετική) παράμετρος εισόδου της.

- Αν το αποτέλεσμα της εργασίας είναι σε μορφή αρχείου ή αρχείων, ο worker το/τα διαθέτει μέσω HTTP. Για το σκοπό αυτό αποφασίστηκε η χρήση ενός ελαφρύ WebServer, του NanoHTTP. Ανάλογα με τη λειτουργία του συστήματος, ο worker θα τα διαθέσει είτε στον server είτε σε άλλους workers απευθείας.

- Τέλος, ο worker διατηρεί μια βάση δεδομένων (πάλι χρησιμοποιήθηκε η H-SQLDB) για να κρατάει ένα μητρώο των αρχείων που έχει δημιουργήσει και τα οποία πρέπει να σβήνει όταν λήξει το timeout τους.

- Το σβήσιμο των αρχείων των οποίων λήγει το timeout, στην πλευρά του worker, αναλαμβάνει μια μέθοδος που καλείται περιοδικά ανά τακτά διαστήματα, ελέγχει τη βάση δεδομένων που κρατά το μητρώο των αρχείων και σε περίπτωση που έχει λήξει το timeout κάποιου το διαγράφει.

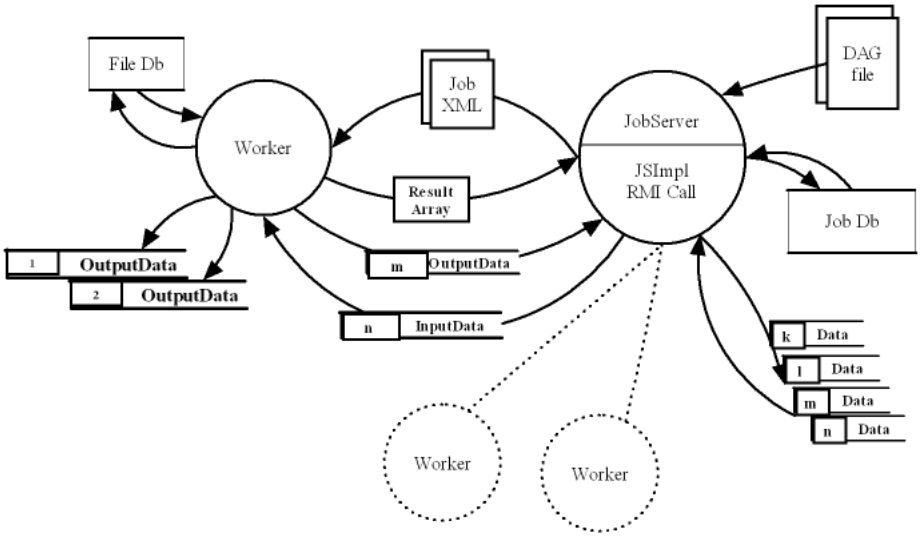
- Με το σβήσιμο των αρχείων εξόδου από την πλευρά του χρήστη δημιουργείται η ανάγκη ενημέρωσης της βάσης δεδομένων του server. Μια μέθοδος στην πλευρά του server αναλαμβάνει περιοδικά να ελέγχει τα μητρώα των αποτελεσμάτων και να βλέπει αν το timeout τους έχει λήξει. Αν έχει λήξει, τότε ο server πρέπει να θεωρήσει ότι ο worker τα έχει διαγράψει, επομένως να ελέγξει αν τα αποτελέσματα αυτά χρειάζονται πλέον στην εκτέλεση της εφαρμογής και αν χρειάζονται να αναθέσει εκ νέου την εργασία εκείνη. Έτσι η κλήση ανά τακτά διαστήματα της μεθόδου αυτής διατηρεί και από την πλευρά του server συνεπή τα μητρώα των αποτελεσμάτων των εργασιών.

Όπως βλέπουμε, όλη η επικοινωνία μεταξύ worker-server πραγματοποιείται σε ένα βήμα για κάθε εργασία, αφού ο πίνακας αποτελεσμάτων του υπολογισμού περνάει σαν δεδομένο εισόδου στην επόμενη κλήση RMI για αίτηση εργασίας. Έτσι μειώνονται οι κλήσεις προς και από τον server με αποτέλεσμα να έχουμε μικρότερη επιβάρυνσή του.

Αφού εξετάσαμε τα γενικά χαρακτηριστικά της δομής του συστήματος και των εφαρμογών που εκτελούνται σε αυτό, της πολιτικής ανάθεσης εργασιών και τα βήματα του υπολογισμού, ήρθε η στιγμή να αναλύσουμε τις δύο διαφορετικές προσεγγίσεις στην επικοινωνία των κόμβων, που αλλάζουν ουσιαστικά τη λειτουργία του συστήματος.

### 3.2.4 **Trantor mode** - Το κέντρο του Γαλαξία

Το 1951 ο Isaac Asimov εξέδωσε το πρώτο βιβλίο από το σημαντικότερο, ίσως, έπος επιστημονικής φαντασίας που έχει γραφτεί, τη σειρά βιβλίων Foundation (ελληνικός τίτλος: Γαλαξιακή Αυτοκρατορία). Ο πλανήτης Trantor, στον εσωτερικό υποδακτύλιο των σπειροειδών του Γαλαξία, ήταν η πρωτεύουσα, το οικονομικό και πολιτικό κέντρο της Γαλαξιακής Αυτοκρατορίας, σύμβολο της ακμής και της λάμψης της σε ολόκληρο το Γαλαξία. Στην τεράστια μητρόπολή του, 45 δισεκατομμύρια άνθρωποι ασχολούνταν με τη διοίκηση όλης της επικράτειας. Αποκομμένος από τις παραγωγικές διαδικασίες των αναγκαίων αγαθών για την επιβίωση, καθώς όλος ο πληθυσμός ασχολούταν με διοικητικό έργο, ο πλανήτης ήταν εξαρτημένος ως προς την τροφοδότησή του από 20 διαφορετικούς κόσμους. Αυτή η εξάρτηση και ο υδροκεφαλισμός τον έκανε ευάλωττο στις πολιορκίες. Στη λάμψη και τη δύναμη που ακτινοβολούσε όμως ο πλανήτης, κανείς σχεδόν δεν μπορούσε να δει τα σημάδια της παρακμής και κανείς δε φανταζόταν πόσο κοντά είναι η πτώση. Κανείς, εκτός από τον Hari Seldon, τον μεγάλο κοινωνικό επιστήμονα και μαθηματικό.




Σχήμα 3.4: Το σύστημα Trantor. Δομή της επικοινωνίας.

Η επικοινωνία του συστήματός μας, κατά τη λειτουργία Trantor είναι απλή, βασίζεται στο μοντέλο master-worker και μπορεί να συνοψιστεί στα παρακάτω βήματα:

- Ο worker ανταλλάσσει με τον server τα δεδομένα της εργασίας (xml αρχείο) μέσω της κλήσης RMI.
- Ο worker κατεβάζει τον κώδικα και τα δεδομένα εισόδου της εργασίας του από το server.
- Αφού εκτελεστεί η εργασία, ο worker διαθέτει τα δεδομένα εξόδου της.
- Ο server αμέσως τα μεταφέρει τοπικά ώστε να μπορεί είτε να τα επεξεργαστεί τοπικά, είτε να τα κρατήσει σε περίπτωση που είναι τελικά αποτελέσματα, είτε να τα διαθέσει ο ίδιος πάλι στον worker που ενδεχομένως θα τα χρειαστεί σαν δεδομένα εισόδου.

Έτσι, μία ολοκληρωμένη απεικόνιση του συστήματός μας, ως προς τη δομή και την επικοινωνία του, βλέπουμε στο σχήμα 3.4. Αυτό το μοντέλο επικοινωνίας είναι ανάλογο με αυτό που χρησιμοποιούν τα περισσότερα συστήματα κατανεμημένου υπολογισμού και το οποίο αναλύσαμε στο κεφάλαιο 2.

Όπως και ο πλανήτης Trantor, η διαχείριση του κατανεμημένου υπολογισμού κατά τον τρόπο λειτουργίας Trantor mode, παρουσιάζει παρόμοια χαρακτηριστικά:

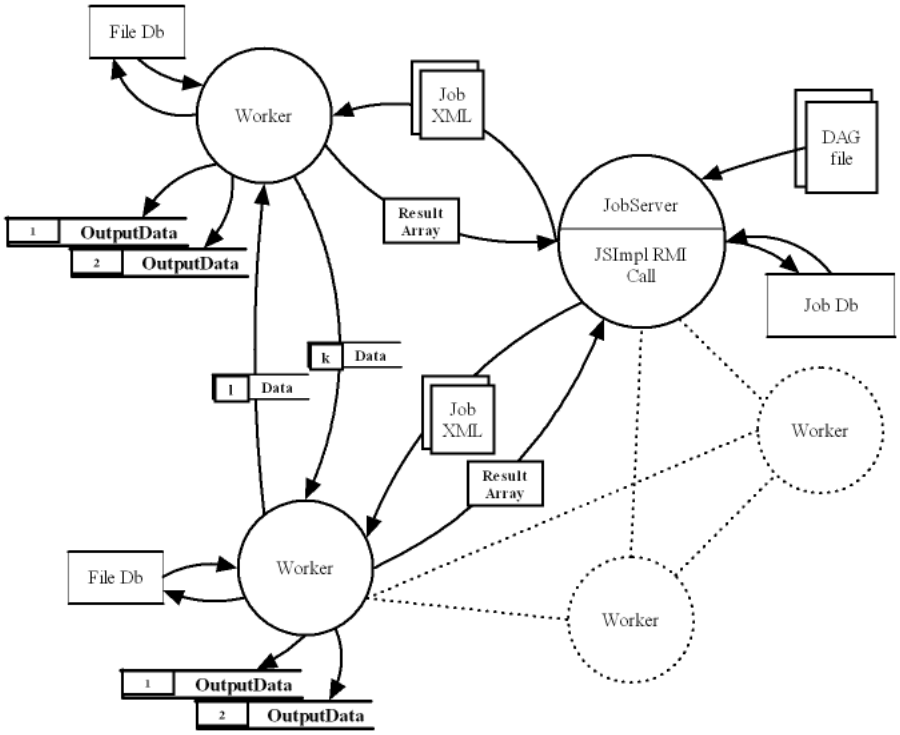
- Όλη η διαχείριση, λειτουργικότητα και επικοινωνία περνάει από τον server που είναι το κέντρο του υπολογιστικού σύμπαντος. Ο όγκος της επικοινωνίας είναι πολύ μεγάλος και απαιτούνται ισχυρές υποδομές δικτύου και υπολογιστικής ισχύος για να αντεπεξέλθει ο server.

- Ο server, ως κομβικό σημείο του συστήματος αποτελεί ευαίσθητο κομμάτι αυτού. Θεωρείται αξιόπιστος αλλά αν αποτύχει, διακυβέβεται η ακεραιότητα ολόκληρου του υπολογισμού.

- Η κλιμάκωση του συστήματος απαιτεί περαιτέρω συγκέντρωση ισχύος στον server. Η αύξηση της κίνησης δημιουργεί βοττλενεχς στην επικοινωνία τα οποία πρέπει να αντιμετωπιστούν ώστε να μην επηρεάζουν την απόδοση του συστήματος.

### 3.2.5 **Terminus mode** - Η Ελπίδα από την Περιφέρεια

Ο Terminus, ένας φτωχός, ξεκομμένος πλανήτης στην περιφέρεια του Γαλαξία παρέμενε ακατοίκητος πέντε αιώνες μετά την ανακάλυψή του. Μέχρι που ο Hari Seldon τον επέλεξε για να εγκαταστήσει εκεί το επιστημονικό του κέντρο πλαισιωμένο απο 100 χιλιάδες επιστήμονες, ώστε απερίσπαστος να διεξαγάγει την έρευνά του και να σώσει το Γαλαξία από το χάος που θα προξενούσε η πτώση της Γαλαξιακής Αυτοκρατορίας.




Σχήμα 3.5: Το σύστημα Terminus. Δομή της επικοινωνίας.

Το σύστημά μας, όταν βρίσκεται σε λειτουργία Terminus, λειτουργεί σαν ένα

peer-to-peer δίκτυο κατανεμημένου υπολογισμού. Σε αυτή τη λειτουργία κεντρικό ρόλο στη διαδικασία παίζουν οι workers. Αυτοί, αν και απομακρυσμένοι είναι το κέντρο του υπολογισμού αλλά και των δεδομένων. Αν ένας worker χρειάζεται σαν δεδομένα εισόδου τα αποτελέσματα της εργασίας ενός άλλου worker τότε απευθύνεται σε αυτόν, από όπου τα λαμβάνει, και όχι σε έναν κεντρικό server. Η επικοινωνία του συστήματος, αποκεντρωμένη ως προς τον όγκο της από τον κεντρικό server, επιτελείται από τον κάθε worker χωριστά, για τις εργασίες που έχει ο ίδιος επεξεργαστεί.

Στο σχήμα 3.5 παρατηρούμε τη διαφορά στη δομή του συστήματος κατά τη λειτουργία Terminus σε σχέση με τη λειτουργία Trantor. Η επικοινωνία μεταξύ των κόμβων ακολουθεί πλέον διαφορετικά μονοπάτια:

- Ο worker ανταλλάσσει με τον server όπως και στη λειτουργία Trantor τα δεδομένα της εργασίας (xml αρχείο) μέση της κλήσης RMI.
- Ο worker κατεβάζει τα δεδομένα εισόδου της εργασίας του από το URL που αναφέρεται στο job.xml. Το URL αυτό, αν πρόκειται για δεδομένα εισόδου παραγόμενα από τον υπολογισμό άλλης εργασίας, αναφέρεται σε αρχείο που διαθέτει κάποιος άλλος worker.
- Αφού εκτελεστεί η εργασία, ο worker διαθέτει τα δεδομένα εξόδου της για όποιον τα χρειαστεί. Ενημερώνει μέσω της επόμενης κλήσης RMI του server για τα URLs όπου βρίσκονται αυτά.
- Ο server καταχωρεί τις τοποθεσίες των αποτελεσμάτων για να μπορεί να τις δώσει σε όποιον worker ενδεχομένως τα χρειαστεί σαν δεδομένα εισόδου ώστε εκείνος να τα κατεβάσει άμεσα.

Όπως βλέπουμε, ο server επιβαρύνεται μονάχα με την επικοινωνία για ανταλλαγή των στοιχείων των εργασιών και δεν ασχολείται με τη μεταφορά των δεδομένων ούτε ο ίδιος ούτε κάποιος άλλος κεντρικός server. Η επικοινωνία συνολικά στο σύστημά μας μειώνεται στο μισό αφού πλέον δε χρησιμοποιείται μεσάζοντας για την ανταλλαγή αρχείων. Όμως το κέρδος δεν προκύπτει τόσο από τη συνολική μείωση της επικοινωνίας όσο από την αποκέντρωσή της από τον server. Ακόμα και αν υιοθετήσουμε για λόγους αξιοπιστίας, όπως περιγράφουμε παρακάτω, κάποιες τεχνικές που αυξάνουν πάλι τη συνολική επικοινωνία του συστήματος σε αντίστοιχα επίπεδα με πριν, κρίνουμε ότι δε θα επηρεαστεί σημαντικά η απόδοσή του, λόγω αυτής της αποκέντρωσης και της κατανομής των δεδομένων και επομένως της επικοινωνίας.

## 3.3 Αποφάσεις - Υλοποίηση

### 3.3.1 Αρχείο Γράφου **DAG** εργασιών

Το αρχείο που περιγράφει τον κατευθυνόμενο ακυκλικό γράφο εκτέλεσης των βημάτων υπολογισμού είναι το θεμελιωδες αρχείο του υπολογισμού που καθορίζει όλη του την πορεία. Κάθε γραμμή του αρχείου αντικατοπτρίζει μία στοιχειώδη

εργασία με τις παραμέτρους της. Στο αρχείο μπορούν να υπάρχουν στοιχεία πολλών διαφορετικών εφαρμογών. Τα βασικά στοιχεία που αποτελούν μια γραμμή του αρχείου χωρίζονται με τον κενό χαρακτήρα και είναι τα εξής:

```
<job_id> <code-version> <code-url> <inputs> <outputs>
```

Το `job_id` είναι ένας φυσικός αριθμός που πρέπει να είναι μοναδικός για κάθε εργασία και τη χαρακτηρίζει.

Το `code-version` είναι η έκδοση του κώδικα. Σε περίπτωση που χρειαστεί να αλλάξουμε τον εκτελέσιμο κώδικα για τη συγκεκριμένη εργασία (π.χ. βελτιώσεις, διόρθωση σφαλμάτων), μεταβάλλουμε την καταχώρηση αυτή.

Το `code-url` είναι η διεύθυνση στο Internet όπου ο κώδικας είναι διαθέσιμος. Είναι φυσικά ένα string της γνωστής μορφής των URLs. Ο κώδικας πρέπει να είναι της μορφής jar ή class.

Τα `inputs` είναι μία σειρά από ορίσματα εισόδου που χωρίζονται μεταξύ τους με το θαυμαστικό (!). Έτσι υποστηρίζεται μη σταθερός αριθμός ορισμάτων εισόδου.

Κάθε όρισμα εισόδου αποτελείται από παραμέτρους που το περιγράφουν και που μεταξύ τους χωρίζονται με το σύμβολο @. Οι παράμετροι αυτές είναι:

`arg-id`: Αύξων αριθμός που ορίζει μονοσήμαντα μέσα σε μια εργασία το όρισμα εισόδου.

`arg-method`: Ο τρόπος λήψης του ορίσματος εισόδου. Παίρνει τις τιμές 0, για inline πέρασμα του ορίσματος, 1 για στατικό URL και 2 για λήψη από αποτέλεσμα άλλης εργασίας.

`arg-value`: Αν το πέρασμα του ορίσματος γίνεται inline αυτή είναι η τιμή του. Αν γίνεται μέσω URL αυτή η παράμετρος είναι το URL. Αν γίνεται από άλλη εργασία, αυτή η παράμετρος είναι το `job_id` της εργασίας εκείνης

`arg-out-id`: Σε περίπτωση που το συγκεκριμένο όρισμα εισόδου πρέπει να ληφθεί από άλλη εργασία, αυτή η παράμετρος είναι το `id` του αποτελέσματος εξόδου. Διαφορετικά είναι -1.

Τα `outputs` είναι μία σειρά από ορίσματα εξόδου που, όπως και τα ορίσματα εισόδου, χωρίζονται μεταξύ τους με τον χαρακτήρα "!". Έτσι υποστηρίζεται και εδώ μη σταθερός αριθμός ορισμάτων εξόδου.

Τα ορίσματα εξόδου αποτελούνται επίσης από παραμέτρους που τα περιγράφουν και μεταξύ τους χωρίζονται με το σύμβολο @. Οι παράμετροι αυτές είναι:

`arg-out-id`: Αύξων αριθμός που ορίζει μονοσήμαντα μέσα σε μια εργασία το όρισμα εξόδου.

`arg-method`: Ο τρόπος επιστροφής του ορίσματος εξόδου. Παίρνει τις τιμές 0, για inline πέρασμα του ορίσματος και 1 για αποθήκευση σε αρχείο.

`arg-filename`: Αν το πέρασμα του ορίσματος γίνεται inline αυτή η τιμή είναι 0. Αν γίνεται μέσω αρχείου αυτή η παράμετρος είναι το filename του αρχείου εξόδου.

`arg-timeout`: Σε περίπτωση που το συγκεκριμένο όρισμα εξόδου είναι αρχείο, η τιμή αυτή είναι ο χρόνος σε λεπτά που το αρχείο πρέπει να διατηρηθεί στον worker για λήψη από τον server ή από άλλους workers. Διαφορετικά είναι -1.

Μια τυπική γραμμή του αρχείου φαίνεται παρακάτω:

```
21 4 http://147.27.1.191:8181/Multiply.jar
1@1@http://147.27.1.191:8181/A11.txt@-1
!2@1@http://147.27.1.191:8181/B11.txt@-1
1@1@ArrC.txt@120
```

Η εργασία αυτή έχει `job_id` 21, ο κώδικας έχει έκδοση 4 και URL http://147.27.1.191:8181/Multiply.jar. Η εργασία έχει 2 ορίσματα εισόδου. Το πρώτο, με `arg_id` 1 λαμβάνεται μέσω URL το οποίο είναι το http://147.27.1.191:8181/A11.txt. Το δεύτερο με `arg_id` 2 λαμβάνεται πάλι μέσω URL το οποίο είναι το http://147.27.1.191:8181/B11.txt Το μοναδικό όρισμα εξόδου της έχει `arg_out_id` 1, είναι αρχείο, έχει filename ArrC.txt και timeout 120 λεπτά.

### 3.3.2 Βάση Δεδομένων **Server**

Ο server χρησιμοποιεί μια βάση δεδομένων για να κρατάει το μητρώο των εργασιών, τις εισόδους και τις εξόδους τους, καθώς και ένα αρχείο με τους workers. Η βάση δεδομένων περιλαμβάνει 4 πίνακες οι οποίοι αποθηκεύονται για λόγους αξιοπιστίας στο σκληρό δίσκο και μετά από κάθε μεταβολή γίνονται commit οι αλλαγές σε αυτούς:

- Τον `job_table`, που κρατά τα γενικά δεδομένα μιας εργασίας, την κατάστασή της κλπ.
- Τον `arg_in_table`, που περιέχει τα στοιχεία για τα ορίσματα εισόδου των εργασιών.
- Τον `arg_out_table`, που διατηρεί τα αποτελέσματα των εργασιών και στοιχεία για τα πολλαπλά αντίγραφά τους.
- Τον `client_table` με το μητρώο των workers.

Αναλυτικά εξηγούμε τη δομή του κάθε πίνακα και τα πεδία του.

#### `job_table`

Κάθε καταχώρηση στο `job_table` αποτελεί ένα αντίγραφο μιας εργασίας που πρέπει να γίνει, γίνεται ή έχει ολοκληρωθεί. Έτσι, αν η εργασία έχει δοθεί σε παραπάνω από έναν workers στο `job_table` θα υπάρχουν τόσες καταχωρήσεις για την εργασία αυτή όσα και τα αντίγραφά της. Το `job_table` αποτελείται από τα πεδία που φαίνονται παρακάτω:

| id | dag_id | code_id | code_uri | timestamp | state | client_id |
|---|---|---|---|---|---|---|

Πίνακας 3.1: `job_table`.

Το id είναι το κλειδί του πίνακα, αφορά το συγκεκριμένο αντίγραφο της εργασίας και το αποδίδει αυτόματα η βάση δεδομένων σε κάθε καταχώρηση. Το dag_id λαμβάνεται από το αρχείο του γράφου και χαρακτηρίζει την εργασία. Το code_id είναι ουσιαστικα η έκδοση του κώδικα και το code_uri η τοποθεσία του. Και τα δύο λαμβάνονται από το αρχείο του γράφου. Το timestamp αρχικά είναι κενό και όταν η εργασία είναι δοσμένη προς υπολογισμό δείχνει την ώρα εκκίνησης του υπολογισμού, ενώ αν έχει ολοκληρωθεί δείχνει την ώρα τερματισμού. Το state δείχνει την κατάσταση της εργασίας στο σύνολό της και αφορά όλα τα αντίγραφά της και όχι το συγκεκριμένο. Είναι 0 αν η εργασία δεν έχει δοθεί, αρνητικό αν όλα τα αντίγραφά της είναι δοσμένα και δεν έχουν ακόμα ολοκληρώσει και θετικό αν έστω και ένα έχει ολοκληρώσει τον υπολογισμό. Αυτό επελέγη για να βρίσκουμε γρήγορα αν η εργασία γενικά στο σύνολό της έχει ολοκληρωθεί κάπου, έστω και σε έναν worker. Επίσης η απόλυτη τιμη του state δείχνει το πλήθος των workers που έχουν αναλάβει τη συγκεκριμένη εργασία. Το client_id δείχνει τον worker που έχει αναλάβει το συγκεκριμένο αντίγραφο.

Αρχικά για κάθε εργασία του γράφου υπολογισμού δημιουργείται μία καταχώρηση με κενό το timestamp, το state 0 και το client_id -1 αφού δεν έχει δοθεί ακόμη. Κατά τη διάρκεια της εκτέλεσης, οι καταχωρήσεις στο job_table αλλάζουν δυναμικά ως προς τα πεδία timestamp, state και client_id. Επίσης για κάθε νέο στιγμιότυπο ήδη υπάρχουσας εργασίας δημιουργείται και ένα αντίγραφο της καταχώρησής της στο job_table με διαφορετικά τα πεδία timestamp και client_id και αναβαθμισμένο το state.

**arg_in_table**

Ο πίνακας των ορισμάτων εισόδου περιέχει τις πληροφορίες που χρειάζονται για να λάβουμε τα δεδομένα εισόδου μιας εργασίας. Κάθε καταχώρηση σε αυτόν αφορά ένα μόνο όρισμα εισόδου κάποιας εργασίας. Έτσι για μία εργασία με 3 ορίσματα εισόδου θα έχουμε 3 καταχωρήσεις στον πίνακα αυτόν. Οι καταχωρήσεις αφορούν όλα τα αντίγραφα της εργασίας που περιγράφουν, αφού είναι ανεξάρτητες από το στιγμιότυπο της εργασίας. Μία καταχώρηση στον arg_in_table αποτελείται από τα εξής πεδία

| id | dag_id | arg_id | arg_method | arg_in | arg_out_id |
|---|---|---|---|---|---|

Πίνακας 3.2: arg_in_table .

Το id είναι το κλειδί και πάλι του πίνακα και αφορά το συγκεκριμένο όρισμα της συγκεκριμένης εργασίας. Το dag_id δείχνει την εργασία που αφορά η καταχώρηση και το arg_id το συγκεκριμένο όρισμά της που περιγράφουμε. Το arg_method δείχνει τον τρόπο που περνιέται το όρισμα. Παίρνει τις τιμές 0 για inline πέρασμα, 1 για URL και 2 για λήψη από άλλη εργασία. Το arg_in δείχνει το URL σε περίπτωση που το arg_method είναι 1 και το dag_id της εργασίας που θα δώσει το αποτέλεσμα, αν το arg_method είναι 2. Τέλος, το arg_out_id δείχνει το id του ορίσματος εξόδου της εργασίας που θα δώσει το αποτέλεσμα,

σε περίπτωση που το arg_method είναι 2.

Τα πεδία του arg_in_table λαμβάνονται από τα αντίστοιχα πεδία στο αρχείο του γράφου στην εκκίνηση του συστήματος και δε μεταβάλλονται κατά τη διάρκεια της εκτέλεσης της εφαρμογής, εκτός από την περίπτωση που διαγραφεί όλη η εργασία την οποία αφορούν, οπότε και διαγράφονται όλες οι καταχωρήσεις που αναφέρονται σε αυτήν.

**arg_out_table**

Ο πίνακας καταχωρήσεων των αποτελεσμάτων είναι ο μεγαλύτερος πίνακας του συστήματος. Εδώ καταχωρούνται τα διαφορετικά αποτελέσματα κάθε εργασίας και για κάθε διαφορετικό αντίγραφό της. Έτσι, μία καταχώρηση σε αυτόν τον πίνακα αφορά μια συγκεκριμένη εκτέλεση μιας συγκεκριμένης εργασίας. Τα πεδία του πίνακα είναι τα εξής:

| id | dag_id | arg_id | arg_method | arg_out_filename | timeout | client_id | arg_out |
|---|---|---|---|---|---|---|---|

Πίνακας 3.3: arg_out_table .

Το id είναι πάλι το κλειδί του πίνακα, αφορά τη συγκεκριμένη καταχώρηση στον πίνακα και δίνεται αυτόματα από τη βάση. Το dag_id δείχνει την εργασία που αφορά η καταχώρηση και το arg_id το συγκεκριμένο όρισμά της που περιγράφουμε. Το arg_method δείχνει τον τρόπο που περνιέται το όρισμα. Παίρνει τις τιμές 0 για inline πέρασμα αν είναι απλός τύπος δεδομένων και 1 για URL σε περίπτωση που το αποτέλεσμα είναι αρχείο. Το arg_out_filename δείχνει το όνομα του αρχείου σε περίπτωση που το arg_method είναι 1. Το timeout δείχνει την τιμή σε λεπτά του διαστήματος που πρέπει να διατηρηθεί το αποτέλεσμα στον worker σε περίπτωση που αυτό είναι αρχείο (δηλαδή το arg_method ισούται με 1). Το client_id είναι ο χαρακτηριστικός αριθμός του worker που έχει αναλάβει το συγκεκριμένο αντίγραφο της εργασίας που θα δώσει το εν λόγω αποτέλεσμα. Το arg_out είναι η τιμή του αποτελέσματος. Αν πρόκειται για απλό τύπο ισούται με το string που αναπαριστά το αποτέλεσμα. Αν το αποτέλεσμα είναι ένα αρχείο, το arg_out είναι το URL στο οποίο το αρχείο αυτό είναι διαθέσιμο.

Οι καταχωρήσεις στον πίνακα των αποτελεσμάτων αλλάζουν δυναμικά. Αρχικά για κάθε εργασία δημιουργείται ένα σετ από τις καταχωρήσεις των αποτελεσμάτων της, πριν αυτή δοθεί σε κάποιον worker και επομένως χωρίς client_id και arg_out. Για κάθε νέο αντίγραφο κάποιας εργασίας, αντιγράφονται και οι καταχωρήσεις των αποτελεσμάτων στον πίνακα αυτόν. Επίσης, τα πεδία client_id και arg_out αλλάζουν κατά τη διάρκεια της εκτέλεσης. Τα υπόλοιπα πεδία αφορούν γενικά το συγκεκριμένο αποτέλεσμα της εργασίας, είναι ανεξάρτητα του στιγμιότυπού της και λαμβάνονται από το γράφο του υπολογισμού στην εκκίνηση του server.

| id | client_id | timestamp |
|---|---|---|

Πίνακας 3.4: `client_table` .

**`client_table`**

Ο πίνακας `client_table` δημιουργείται δυναμικά κατά τη λειτουργία του server και δεν παίρνει καμία τιμή από το γράφο υπολογισμού. Αποτελεί ένα μητρώο χρηστών του συστήματος, όπου, όπως φαίνεται και στα πεδία του κρατάμε το χαρακτηριστικό αριθμό του worker (ώστε σε κάθε νέα εισαγωγή worker στο σύστημα να μπορούμε να βρούμε ένα μοναδικό νέο αριθμό να του αποδώσουμε) και το χρόνο που ολοκλήρωσε την τελευταία του εργασία (για στατιστικούς λόγους και για την διαχείριση της περίπτωσης να έχει αποσυνδεθεί από το σύστημά μας προ πολλού)

### 3.3.3 Βάση Δεδομένων **Worker**

Εκτός από τον server και ο worker διαθέτει μία βάση δεδομένων, για τη διατήρηση του μητρώου των αρχείων που έχει δημιουργήσει, με σκοπό τη διαγραφή τους μετά το πέρας του timeout. Οι καταχωρήσεις εισάγονται δυναμικά, κάθε στιγμή που παράγεται κάποιο αρχείο από τον κώδικα που εκτελεί ο worker (για την ακρίβεια, μετά το πέρας της εκτέλεσης του κώδικα που παρήγαγε το αρχείο). Ο μοναδικός πίνακας που την αποτελεί, ο `file_table` διαθέτει τα εξής πεδία:

| id | filename | deadline |
|---|---|---|

Πίνακας 3.5: `file_table`.

Το `id` είναι και πάλι το κλειδί του πίνακα και δίνεται αυτόματα από τη βάση. Το `filename` είναι το όνομα του αρχείου που παρήγαγε ο κώδικας που εκτέλεσε ο worker και είναι ίδιο με το αντίστοιχο αρχείο του `arg_out_table` του server το οποίο λαμβάνεται από το γράφο υπολογισμού. Το `deadline` είναι το άθροισμα του χρόνου δημιουργίας της καταχώρησης (δηλαδή κατά προσέγγιση του χρόνου δημιουργίας του αρχείου) με το timeout του αρχείου και δείχνει την ώρα που το αρχείο πρέπει να διαγραφεί.

### 3.3.4 **Castor xsd - job.xml**

Όπως προαναφέραμε, ο τρόπος με τον οποίο ανταλλάσσεται η πληροφορία για μία νέα εργασία μεταξύ worker και server είναι ένα αρχείο xml που επιστρέφει η κλήση RMI του server. Η ανάγνωση και εξαγωγή των δεδομένων από αυτό το μορφοποιημένο αρχείο κειμένου, γίνεται από έναν εξειδικευμένο parser τον Castor. Αρχικά κατασκευάζεται το πρότυπο του xml αρχείου που θα περαστεί, το αρχείο xsd, που προδιαγράφει τι είδους μεταβλητές και οντότητες θα περιέχει το xml αρχείο. Το αρχείο xsd που κατασκευάσαμε και χρησιμοποιήσαμε για να περιγράψουμε τις εργασίες του συστήματός μας φαίνεται στο Παράρτημα. Παρακάτω

περιγράφουμε όλα τα στοιχεία που συνθέτουν ένα αρχείο job.xml που περιγράφει μια εργασία. Σε παρενθέσεις είναι οι πληθικότητες (min,max) των στοιχείων αυτών μέσα σε ένα τέτοιο αρχείο. Τα στοιχεία αυτά συνήθως προκύπτουν άμεσα ή έμμεσα από το αρχείο DAG που έχει δώσει τις αντίστοιχες τιμές στα πεδία της βάσης δεδομένων, εκτός αν αναφέρουμε διαφορετική πηγή τους.

Μια εργασία (Job) αποτελείται από τα εξής στοιχεία:

- Το `job-id` (1,1), τύπου long, που είναι μοναδικό και χαρακτηρίζει την εργασία.
- Το `client-id` (1,1) που είναι επίσης μοναδικό για κάθε client. Είναι επίσης τύπου long και δίνεται στο νέο worker μέσω τυχαίου αλγορίθμου που ψάχνει για κάποιο αριθμό από το 0 ως το 9999 που να μην είναι καταχωρημένος στον πίνακα `client_table` ως `client_id`.
- Τον κώδικα (`code`) (1,1) που πρέπει να εκτελεστεί και αναλύεται σε επιμέρους στοιχεία.
- Τα δεδομένα εισόδου (`data-in`) (1,1) για την εργασία που αναλύονται σε επιμέρους στοιχεία.
- Τα δεδομένα εξόδου (`data-out`) (1,1) για την εργασία που επίσης αναλύονται σε επιμέρους στοιχεία.

Ο κώδικας (`code`) αποτελείται από τα εξής:

- Το πεδίο `code-uri` (1,1), τύπου URI, που δείχνει την αρχική τοποθεσία από όπου ο worker μπορεί να κατεβάσει τον κώδικα.
- Το πεδίο `code-version` (1,1), τύπου long που υποδεικνύει την έκδοση του κώδικα και μας διευκολύνει να αναβαθμίσουμε τον κώδικα που τρέχει ο worker απλά αλλάζοντας το πεδίο αυτό (και αντικαθιστώντας φυσικά στο URL την παλιά έκδοση με μια νέα).

Τα δεδομένα εισόδου (`data-in`) αποτελούνται από ορίσματα εισόδου (`arg-in`) (0, απεριόριστα), τα οποία με τη σειρά τους περιέχουν:

- Το `arg-in-id` (1,1) τύπου long, που χαρακτηρίζει το όρισμα και
- είτε το πεδίο `arg-in-inline`, τύπου string για άμεσο πέρασμα των δεδομένων εισόδου
- είτε τα εξής: ο Το `arg-in-uri` (1,1), τύπου URI, που είναι η θέση του ορίσματος στο WWW. ο Το `arg-in-credential` (1,1), τύπου string, που είναι το password για να δοθούν ασφαλώς τα δεδομένα αυτά. Το πεδίο αυτό δεν χρησιμοποιείται λειτουργικά στην παρούσα υλοποίηση.

Η επιλογή για τον τύπο του ορίσματος (inline ή με URI) γίνεται από το αντίστοιχο πεδίο του αρχείου DAG.

Τα δεδομένα εξόδου (`data-out`) αποτελούνται από ορίσματα εξόδου (`arg-out`) (1,απεριόριστα), τα οποία με τη σειρά τους περιέχουν:

- Το `arg-out-id` (1,1) τύπου long, που χαρακτηρίζει το όρισμα
- Το `arg-method` (1,1) τύπου long, που ορίζει τον τρόπο που θα επιστραφεί το όρισμα εξόδου
- Το `arg-out-filename` (0,1), τύπου string, που είναι το όνομα του αρχείου του αποτελέσματος
- Το `arg-out-credential` (0,1), τύπου string, που είναι το password για να παρέχονται ασφαλώς τα δεδομένα αυτά. Το πεδίο αυτό δεν χρησιμοποιείται λειτουργικά στην παρούσα υλοποίηση.
- Το `arg-out-timeout` (0,1), τύπου string, που είναι ο χρόνος σε λεπτά που θα κρατήσει το αρχείο ο worker μέχρι να το διαγράψει.

Στη συνέχεια, το Castor δημιουργεί, βάσει του ξσδ αρχείου, τον κώδικα σε Java ο οποίος μπορεί να διαβάσει, να κάνει parse και να αποθηκεύσει σε δομές τα δεδομένα ενός xml αρχείου που ακολουθεί τη δομή xsd που μόλις περιγράψαμε. Ο παραγόμενος κώδικας του Castor συμπεριλαμβάνεται με τον κώδικά μας και οι συναρτήσεις του για τη διαχείριση των στοιχείων αυτών είναι διαθέσιμες στο πρόγραμμά μας. Έτσι, ο server παράγει κάθε φορά ένα αρχείο xml της μορφής που περιγράφει το xsd. Στο σχήμα βλέπουμε ένα από τα παραγόμενα xml αρχεία. Αυτό το αρχείο γίνεται parse μέσω των μεθόδων του Castor από την πλευρά του worker, ώστε να εξαχθούν τα δεδομένα προς χρήση.

```
<?xml version="1.0" encoding="UTF-8"?>
<Job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="Job">
  <job-id>118</job-id>
  <client-id>99</client-id>
  <code>
    <code-uri>http://147.27.1.191:8181/MergeH.jar</code-uri>
    <code-version>1</code-version>
  </code>
  <data-in>
    <arg-in>
      <arg-in-id>1</arg-in-id>
      <arg-in-credential>password</arg-in-credential>
      <arg-in-uri>http://147.27.1.191:8181/Output107/ArrC.txt</arg-in-uri>
    </arg-in>
    <arg-in>
      <arg-in-id>2</arg-in-id>
      <arg-in-credential>password</arg-in-credential>
      <arg-in-uri>http://147.27.1.191:8181/Output108/ArrC.txt</arg-in-uri>
    </arg-in>
  </data-in>
  <data-out>
    <arg-out>
      <arg-out-id>1</arg-out-id>
      <arg-method>1</arg-method>
      <arg-out-credential>password</arg-out-credential>
```

```
      <arg-out-filename>ArrC.txt</arg-out-filename>
      <arg-out-timeout>133</arg-out-timeout>
    </arg-out>
  </data-out>
</Job>
```

## 3.4 Το σύστημα - Λειτουργία

Το σύστημα ξεκινά με την εκκίνηση του server. Από εκείνη τη στιγμή ο worker μπορεί να κάνει μια κλήση RMI στον server η οποία διεκπεραιώνει τις αναγκαίες διαδικασίες για την εύρεση και την επιστροφή των στοιχείων της διεργασίας. Με την επιστροφή των δεδομένων, ο worker εκτελεί τον κώδικα και επιστρέφει με την επόμενη κλήση RMI τα αποτελέσματα στον server.

Το σύστημα αποτελείται από τρία συστατικά κομμάτια που σηκώνουν τη λειτουργικότητα του προγράμματος:

- Το κομμάτι του JobServer, που κάνει όλες τις αρχικοποιήσεις, ξεκινά τους δαίμονες που χρειάζεται ο server και τον προετοιμάζει να δεχτεί οποιαδήποτε αίτηση εργασίας
- Τον worker ο οποίος κάνει μια κλήση RMI στον server, παίρνει τα στοιχεία της νέας εργασίας και τα αναλύει. Φορτώνει τη νέα εργασία, την εκτελεί και επιστρέφει τα αποτελέσματα.
- Την κλήση RMI στην πλευρά του server για την επιστροφή νέας εργασίας, η οποία υλοποιεί ένα Remote Interface, γνωστό στον worker. Η κλήση αναλαμβάνει να πάρει τα αποτελέσματα του υπολογισμού της προηγούμενης εργασίας, να βρει και να επιστρέψει στον worker μια νέα εργασία καθώς και να δώσει στον worker ένα καινούριο `id` σε περίπτωση που δεν έχει.

Θα εξετάσουμε αναλυτικά τα τρία αυτά μέρη του συστήματος.

### 3.4.1 Ο **JobServer**

Ο JobServer είναι το πρόγραμμα που καλείται αρχικά στην πλευρά του server για να ξεκινήσει όλες τις απαραίτητες λειτουργίες που χρειάζεται ο server.

#### Αρχικοποιήσεις

Στην αρχή, γίνεται εκκίνηση της βάσης δεδομένων που είναι απαραίτητη για τη διατήρηση των δεδομένων των εργασιών και των χρηστών. Γίνεται έλεγχος για το αν έχει ήδη δημιουργηθεί το απαραίτητο σχήμα πινάκων στη βάση και σε περίπτωση που δεν έχει δημιουργηθεί (δηλαδή δεν έχει ξαναλειτουργήσει ο server ή έχουν διαγραφεί τα δεδομένα της βάσης) κατασκευάζονται, οι εξής πίνακες:

- `job_table`: για τη διατήρηση των βασικών δεδομένων των εργασιών

- `arg_in_table`: για τη διατήρηση των δεδομένων εισόδου κάθε διεργασίας
- `arg_out_table`: για τη διατήρηση των δεδομένων εξόδου κάθε διεργασίας
- `client_table`: για τη διατήρηση του αρχείου χρηστών του συστήματος

Η δομή των παραπάνω πινάκων έχει αναφερθεί στην παράγραφο που περιγράψαμε τη σχεδίαση του συστήματος.

Στη συνέχεια εισάγονται τα στοιχεία των νέων εργασιών που πρέπει να δρομολογηθούν από το server. Αυτά βρίσκονται σε ένα αρχείο dagger.txt με τη μορφή που έχουμε ήδη περιγράψει και αποτελούν την αναπαράσταση του ακυκλικού γράφου που περιγράφει τα βήματα του υπολογισμού της εφαρμογής ή των εφαρμογών που θέλουμε να διεκπεραιώσουμε. Μέσα στο αρχείο είναι δυνατόν να βρίσκονται στοιχεία για παραπάνω από μία εφαρμογές. Το αρχείο διαβάζεται από το πρόγραμμα και εισάγει για κάθε εργασία τα δεδομένα της στον πίνακα `job_table` σημαδεύοντάς την παράλληλα με την ένδειξη idle και δίνοντάς της `client_id` -1 ώστε να είναι διαθέσιμη προς ανάθεση. Με τον ίδιο τρόπο συμπληρώνονται οι πίνακες `arg_in_table` και `arg_out_table` για τα δεδομένα εισόδου και εξόδου της κάθε εργασίας αντίστοιχα.

**checkTimeout**

Κατόπιν, τίθεται σε λειτουργία η συνάρτηση `checkTimeOut` ανά τακτά χρονικά διαστήματα. Η συνάρτηση αυτή αναλαμβάνει να ελέγχει περιοδικά την εγκυρότητα των δεδομένων εξόδου κάθε εργασίας σε σχέση με το αν έχει λήξει το timeout που τους είχε δοθεί από το dagger.txt σαν διάρκεια ζωής. Η συνάρτηση συνδέεται με τη βάση δεδομένων και αναζητά τις εργασίες που έχουν `client_id` μεγαλύτερο του 0 (δηλαδή έχουν δοθεί). Από αυτές επιλέγονται όσες έχουν `arg_out` δηλαδή όσες έχουν πραγματικά τελειώσει. Αυτό δεν θα μπορούσε να γίνει εξετάζοντας το `state` τους μόνο, γιατί το `state` αφορά όλα τα αντίγραφα των εργασιών και όχι μόνο το συγκεκριμένο που έχει ολοκληρώσει τον υπολογισμό. Έτσι, αν ολοκληρωθεί ένα αντίγραφο μιας εργασίας, τότε όλα τα αντίγραφα θα πάρουν την ένδειξη finished στο πεδίο της κατάστασής τους.

Από τις ολοκληρωμένες πλέον εργασίες, για καθεμιά παίρνουμε το `timestamp`, το οποίο δηλώνει το χρόνο λήξης της εργασίας πλέον, το κάνουμε parse σε μία μεταβλητή τύπου Calendar, του προσθέτουμε το timeout σε λεπτά και το συγκρίνουμε με την τρέχουσα ώρα. Αν το άθροισμα της ώρας ολοκλήρωσης με το timeout είναι νωρίτερα από την τρέχουσα ώρα, σημαίνει ότι το timeout έχει περάσει και πρέπει να θεωρήσουμε ότι το αποτέλεσμα δεν είναι πλέον διαθέσιμο. Έτσι προσπαθούμε να δούμε αν μας χρειάζεται ακόμα αυτό το αποτέλεσμα. Αναζητούμε τις εργασίες από τον πίνακα των δεδομένων εισόδου που χρειάζονται το αποτέλεσμα αυτό σαν είσοδο. Αν δεν υπάρχουν, τότε το αποτέλεσμα δεν είναι απαραίτητο και σβήνουμε την καταχώρησή του από το `arg_out_table`. Αν υπάρχουν, ελέγχουμε αν έχουν ολοκληρωθεί. Αν έστω και μία δεν έχει ολοκληρωθεί, σημαίνει ότι το αποτέλεσμα μας είναι αναγκαίο, διαφορετικά το σβήνουμε όπως παραπάνω από το `arg_out_table`. Αν λοιπόν είναι αναγκαίο το αποτέλεσμά μας ψάχνουμε αν

υπάρχει άλλο αντίγραφο της ίδιας εργασίας που μπορεί να το παραγάγει. Αν υπάρχει, τότε μπορούμε να το σβήσουμε από το arg_out_table, διαφορετικά θέτουμε την εργασία idle το timeout κενό και το client_id -1 για να ξαναδοθεί προς υπολογισμό. Έτσι, αφαιρώντας τα περιττά arg_out κρατάμε ενημερωμένο τον πίνακα arg_out_table.

Αφού ελέγξουμε όλα τα αποτελέσματα εφαρμόζουμε εκκαθάριση στους πίνακες job_table και arg_in_table. Κοιτάμε όλες τις εργασίες του job_table και τις συγκρίνουμε με αυτές του arg_out_table. Οι εργασίες του job_table που δεν υπάρχουν στο arg_out_table (το οποίο μόλις πριν καθαρίσαμε από τα ανεπιθύμητα arg_out) πρέπει να εκκαθαριστούν. Αναζητούμε αν υπάρχουν άλλα αντίγραφα της εργασίας που έχουν τελειώσει. Αν υπάρχουν αφήνουμε το state ως έχει (finished) αλλιώς το κάνουμε given (γιατί, για να υπάρχει αντίγραφο, σημαίνει ότι έχει δοθεί η εργασία αναγκαστικά και ότι δεν είναι idle). Την ίδια στιγμή μειώνουμε την απόλυτη τιμή του state κατά ένα για να δείξουμε ότι έχουμε ένα αντίγραφο λιγότερο. Σε περίπτωση που η εργασία προς αφαίρεση ήταν μοναδική σημαίνει ότι και τα inputs της δεν χρειάζονται οπότε αφαιρούμε από το arg_in_table τις καταχωρήσεις που την αφορούν. Στο τέλος λοιπόν της checkTimeOut έχουμε και τους τρεις πίνακες που αφορούν τις εργασίες ενημερωμένους.

#### WebServer

Αφού θέσουμε σε λειτουργία την checkTimeOut περιοδικά ώστε να ελέγχει ανά τακτά διαστήματα για τα αποτελέσματα που έχει λήξει η εγκυρότητά τους και να κάνει τις απαραίτητες εκκαθαρίσεις, θέτουμε σε λειτουργία και τον webserver που θα μας χρειαστεί για την παροχή των αρχείων εισόδου του server στους workers. Η θύρα που θα χρησιμοποιηθεί για τη λειτουργία του webserver διαβάζεται από ένα αρχείο (serversettings.txt) ώστε να μπορεί να αλλάξει ανάλογα με τις ρυθμίσεις του server ή του firewall. Η προκαθορισμένη τιμή της θύρας του NanoHTTPD(σε περίπτωση που δε βρεθεί το αρχείο serversettings.txt) ορίσαμε να είναι η 8181.

### 3.4.2 Ο **Worker**

Ο worker είναι το πρόγραμμα που τρέχει ο εθελοντής παροχέας υπολογιστικής ισχύος για να προσφέρει τους κύκλους του επεξεργαστή του στην υπηρεσία των εφαρμογών μας.

#### Initialization

Αρχικά το πρόγραμμα καλείται με παράμετρο την διεύθυνση IP του server που παρέχει την κλήση RMI για τη νέα εργασία. Το πρόγραμμα αναζητά το αρχείο όπου είναι αποθηκευμένο το client id του, σε περίπτωση που έχει ξανακληθεί στο παρελθόν και έχει λάβει ένα τέτοιο. Κατόπιν ξεκινά τη βάση δεδομένων, που είναι απαραίτητη για τη διατήρηση ενός ιστορικού των αρχείων των αποτελεσμάτων που παράγει, ώστε να μπορεί να τα σβήνει περιοδικά. Ελέγχει αν υπάρχει

έτοιμος ο πίνακας των αρχείων και αν δεν υπάρχει τον δημιουργεί με το σχήμα που έχουμε ήδη περιγράψει. Κατόπιν θέτει σε επαναλαμβανόμενη λειτουργία τη συνάρτηση `cleanupFiles` η οποία έχει σαν σκοπό την εκκαθάριση των αρχείων των οποίων το timeout έχει λήξει. Η `cleanupFiles` ελέγχει αν η τρέχουσα ώρα είναι μεγαλύτερη από την ώρα του `deadline`, του χρόνου λήξης, κάποιου αρχείου. Αν ισχύει αυτό, διαγράφει το αρχείο καθώς και την καταχώρησή του στον πίνακα αρχείων.

Κατόπιν ξεκινά ο web server στην πλευρά του worker, ο οποίος θα παρέχει τα αρχεία των αποτελεσμάτων. Η θύρα λειτουργίας του διαβάζεται από το εξωτερικό αρχείο clientport.txt ώστε να μπορεί εύκολα να αλλάξει. Αν το αρχείο δε βρεθεί, η προεπιλεγμένη θύρα λειτουργίας του web server του worker είναι η θύρα 8282.

Στον worker είναι γνωστό μόνο το interface του αντικειμένου που υλοποιεί την κλήση RMI. Αρχικά αναζητεί το αντικείμενο στον απομακρυσμένο server και στη συνέχεια το πρόγραμμα μπαίνει σε μία επαναλαμβανόμενη εκτέλεση των λειτουργιών, που θα ονομάζουμε computation loop και θα περιγράψουμε από εδώ και στο εξής, οι οποίες σε γενικές γραμμές περιλαμβάνουν την επικοινωνία με τον server, τη λήψη μιας εργασίας, την εκτέλεσή της και τη δημιουργία του πίνακα των αποτελεσμάτων που πρέπει να επιστραφούν.

### Computation Loop

Ξεκινώντας το computation loop, καλείται η απομακρυσμένη κλήση (RMI) στον server με παραμέτρους τον πίνακα αποτελεσμάτων, το `client_id`, το `job_id` που εκτελέστηκε στο προηγούμενο loop και μια παράμετρο που υποδεικνύει την επιθυμία για λήψη καινούριας εργασίας. Φυσικά αν μιλάμε για την πρώτη λήψη εργασίας του προγράμματος ο πίνακας αποτελεσμάτων είναι κενός και το `job_id` -1. Αν μάλιστα δεν έχει ξανακληθεί ποτέ ο worker το `client_id` θα είναι και αυτό -1 κατά την κλήση. Ο server λοιπόν, επιστρέφει ένα string που είναι το xml αρχείο που περιλαμβάνει τα στοιχεία της νέας εργασίας και τη δομή του οποίου έχουμε περιγράψει νωρίτερα. Το string αυτό αποθηκεύεται τοπικά σαν ένα αρχείο με το όνομα job.xml για ασφάλεια σε περίπτωση διακοπής λειτουργίας του worker. Κατόπιν διαβάζουμε τις παραμέτρους του αρχείου της εργασίας μέσω των συναρτήσεων του Castor που παρέχουν ευκολία στο parsing xml αρχείων. Αν δεν είχαμε `client_id` το εξάγουμε και το αποθηκεύουμε σε ένα αρχείο με όνομα client.id. Αρχικοποιούμε το μέγεθος του πίνακα αποτελεσμάτων αναλόγως του αριθμού τους για την τρέχουσα εργασία και παίρνουμε το `job_id`.

Στη συνέχεια εξετάζουμε την ύπαρξη ή όχι του κώδικα προς εκτέλεση. Όταν ο worker κατεβάζει νέο κώδικα τον αποθηκεύει τοπικά με όνομα αρχείου το οποίο είναι της μορφής:

<αρχικό όνομα κώδικα>-<έκδοση κώδικα>.<κατάληξη>

Έτσι, για παράδειγμα, ο κώδικας με filename code.jar και έκδοση 42 θα αποθηκευτεί τοπικά ώς code-42.jar Αυτό μας παρέχει έναν εύκολο τρόπο να ελέγχουμε την έκδοση του κώδικα ώστε σε περίπτωση που υπάρχει ήδη τοπικά να μη χρειαστεί

να τον ξανακατεβάσουμε και σε περίπτωση που δεν υπάρχει ή που υπάρχει αλλά η έκδοση είναι διαφορετική να τον κατεβάσουμε άμεσα χωρίς άλλους ελέγχους.

Στη συνέχεια, όπως αναφέραμε, ο κώδικας κατεβαίνει από το URL που έχουμε ήδη εξαγάγει, σε περίπτωση που δεν υπάρχει τοπικά. Ετοιμάζουμε τον πίνακα των δεδομένων εισόδου ώστε να έχει μέγεθος ίσο προς πλήθος τους και για κάθε ένα αρχίζουμε να ετοιμάζουμε τον πίνακα. Αν ένα όρισμα εισάγεται inline, τότε αποθηκεύουμε την τιμή του κατευθείαν σαν string στον πίνακα. Διαφορετικά ελέγχουμε εάν το αρχείο εισόδου υπάρχει τοπικά. Αν δεν υπάρχει το κατεβάζουμε σαν βιναρψ αρχείο τοπικά από το URL που μας έχει δοθεί μέσω του αρχείου xml. Στον πίνακα των δεδομένων εισόδου αποθηκεύουμε για αυτό το όρισμα το όνομα του αρχείου σαν string (όπως δηλαδή θα καλούσαμε και από command line το πρόγραμμά μας).

Σε περίπτωση που υπάρξει πρόβλημα με κάποιο όρισμα, ο worker κάνει μια κλήση RMI στον server ενημερώνοντάς τον για το πρόβλημα που παρουσιάστηκε καθώς επίσης και για τον αριθμό του ορίσματος και το `job_id` στα οποία παρουσιάστηκε το πρόβλημα και κάνει continue στο computation loop ώστε να ξαναζητήσει νέα εργασία από την αρχή.

Στη συνέχεια έρχεται η στιγμή της εκτέλεσης του κώδικα που έχουμε ήδη κατεβάσει με τα ορίσματα εισόδου που έχουμε ήδη ετοιμάσει. Φορτώνουμε την κλάση και την καλούμε με παράμετρο τον πίνακα των δεδομένων εισόδου. Τα αποτελέσματα που επιστρέφει ο κώδικας αποθηκεύονται σαν string σε έναν πίνακα.

Το string αυτό περιέχει τις τιμές των ορισμάτων που επιστρέφει inline ο κώδικας. Όμως, μπορεί να υπάρχουν και άλλα αποτελέσματα σε μορφή αρχείων. Κατασκευάζουμε, λοιπόν, τον ολοκληρωμένο πίνακα των αποτελεσμάτων συμπληρώνοντάς τον για ένα-ένα τα στοιχεία του, είτε από τον πίνακα των inline αποτελεσμάτων, είτε με τα URLs των αρχείων που παρήγαγε ο κώδικας. Έτσι, για όλα τα δεδομένα εξόδου (το πλήθος των οποίων γνωρίζουμε από το job.xml) εξετάζουμε αν είναι inline ή παραγόμενα αρχεία. Αν είναι inline παίρνουμε το επόμενο στοιχείο του πίνακα των inline επιστρεφόμενων τιμών και το αποθηκεύουμε στον πίνακα. Διαφορετικά κατασκευάζουμε ένα φάκελο με τίτλο Output και τον αριθμό του `job_id`. Δηλαδή για παράδειγμα, για `job_id` $= 42$ ο φάκελος θα είχε όνομα Output42. Εκεί μεταφέρουμε το αρχείο που προέκυψε από την εκτέλεση του κώδικα, ώστε να μην μπερδευτεί με άλλα αρχεία επόμενων εκτελέσεων και για να έχουμε ξεχωριστά τα αρχεία αποτελεσμάτων για κάθε εργασία. Κατόπιν παίρνουμε την IP διεύθυνση του υπολογιστή του worker και κατασκευάζουμε το URL όπου θα είναι διαθέσιμο το αποτέλεσμα, βάσει της διεύθυνσης, του φακέλου των αποτελεσμάτων και του filename του.

Στη συνέχεια παίρνουμε την τρέχουσα ώρα και της προσθέτουμε το timeout για το συγκεκριμένο αποτέλεσμα. Υποστηρίζουμε διαφορετικά timeouts για κάθε αρχείο αποτελέσματος μιας εργασίας για λόγους ευελιξίας. Παράγουμε σε μορφή string την ημερομηνία με τη βοήθεια της συνάρτησης `createTimestamp` που έχουμε κατασκευάσει. Η `createTimestamp` μετατρέπει μία ημερομηνία που της δίνεται σαν είσοδος σε string της μορφής YYYYMMDDHHmm, κάτι που δεν μας παρείχε κάποια έτοιμη συνάρτηση. Έτσι προκύπτει ο χρόνος λήξης του αρχείου. Εισάγουμε στον πίνακα αρχείων το path του αρχείου (δηλαδή το όνομα του φακέλου και το όνομα του αρχείου) και το χρόνο λήξης του αρχείου ώστε να μπορεί η συνάρτηση `cleanupFiles` αργότερα να αφαιρέσει τα αρχεία που έχουν λήξει.

### 3.4.3 Τα **JSImpl** και **JSInterface**

Η JSImpl είναι η κλάση που χρησιμοποιείται για την κλήση RMI από τον worker στον server για την επιστροφή μιας νέας εργασίας, την υποβολή των αποτελεσμάτων της προηγούμενης και άλλες λειτουργίες διαχείρισης του server. Η JSImpl υλοποιεί το interface JSInterface. Το JSInterface είναι το interface που περιγράφει τη λειτουργικότητα της κλάσης RMI, JSImpl και είναι γνωστό στον worker ώστε να μπορεί να χρησιμοποιήσει την κλάση αυτή. Σύμφωνα με το JSInterface η κλάση JSImpl περιλαμβάνει μία μέθοδο, την getJob με παραμέτρους τον πίνακα των τιμών αποτελεσμάτων της προηγούμενης εργασίας, το client_id, το job_id της προηγούμενης εργασίας και ένα flag που δείχνει την επιθυμία ή όχι του worker για λήψη νέας εργασίας.

#### Διαχείριση αποτελεσμάτων

Η getJob στην υλοποίηση της JSImpl ξεκινά με τον ορισμό της τιμής μιας boolean μεταβλητής με όνομα mode, η οποία όταν είναι false δείχνει ότι η λειτουργία του συστήματος γίνεται σε Trantor mode δηλαδή ο server είναι αυτός που μοιράζει όλα τα δεδομένα και κατεβάζει τοπικά τα αποτελέσματα από όλους τους workers, ενώ όταν είναι true ο server λειτουργεί σε Terminus mode με κατανεμημένα τα δεδομένα των αποτελεσμάτων για να τα ανταλλάσσουν μεταξύ τους ο κάθε worker και όχι μέσω του server. Η τιμή αυτή, για λόγους ευκολίας, διαβάζεται από τη δεύτερη γραμμή του εξωτερικού αρχείου serversettings.txt (η πρώτη γραμμή περιλαμβάνει τη θύρα λειτουργίας του NanoHTTPD του server, όπως προαναφέραμε).

Στη συνέχεια διαχειριζόμαστε τυχόν αποτελέσματα από προηγούμενη εκτέλεση εργασίας του worker. Αν ο πίνακας αποτελεσμάτων δεν είναι κενός και τα job_id και client_id με τα οποία κλήθηκε η getJob δεν είναι -1, τότε καταλαβαίνουμε ότι ο worker θέλει να μας επιστρέψει αποτελέσματα. Αναζητούμε στον πίνακα arg_out_table τα αποτελέσματα της συγκεκριμένης εργασίας και του συγκεκριμένου χρήστη για να τα ενημερώσουμε. Αν έχει δοθεί όντως η εργασία αυτή στο συγκεκριμένο χρήστη και ο αριθμός των αποτελεσμάτων είναι σωστός, τότε επιτρέπεται η ανάθεση των αποτελεσμάτων. Αυτό είναι ένα είδος authentication ώστε να μην προσπαθήσει κάποιος χρήστης να επιστρέψει αποτελέσματα εργασίας που δεν έχει αναλάβει.

Για κάθε αποτέλεσμα που επιστράφηκε εκτελούμε την παρακάτω διαδικασία: Αν λειτουργούμε σε Trantor mode (δηλαδή όλα τα δεδομένα κεντρικοποιημένα στο server) και ο τύπος του αποτελέσματος είναι αρχείο (δηλαδή δεν περνιέται inline το αποτέλεσμα) τότε πρέπει να το κατεβάσουμε τοπικά. Κατεβάζουμε το αρχείο και κατασκευάζουμε ένα φάκελο με τίτλο Output και το job_id της εργασίας, όπως ακριβώς στην πλευρά του worker, για να βάλουμε μέσα το αρχείο. Δηλαδή για job_id=126 ο φάκελος θα ονομάζεται Output126. Αφού κατεβάσουμε το αρχείο, δημιουργήσουμε το φάκελο και βάλουμε μέσα το αρχείο, ετοιμάζουμε και το URL στο οποίο θα εξυπηρετείται το αποτέλεσμα, που αποτελείται από το IP του server, τη θύρα (8181) το όνομα του φακέλου και το filename.

π.χ. http://147.27.1.191:8181/Output136/fordprefect.txt

Αυτό το URL το θέτουμε και στην τιμή arg_out του συγκεκριμένου αποτελέ-

σματος στη βάση δεδομένων μας, ώστε να μπορεί να το πάρει όποιος το χρειάζεται.

Αν δε λειτουργούμε σε Trantor mode ή αν το αποτέλεσμα είναι περασμένο inline τότε ενημερώνουμε τη βάση δεδομένων μας στην τιμή `arg_out` του συγκεκριμένου αποτελέσματος με την τιμή που μας πέρασε ο worker σαν όρισμα, είτε αυτή είναι URL είτε είναι τιμή αποτελέσματος.

Αφού ενημερώσουμε τον πίνακα `arg_out_table` για τις νέες τιμές, ενημερώνουμε και τον πίνακα `job_table` για το χρόνο λήξης της εργασίας (δηλαδή το πεδίο `timestamp`) και για την κατάστασή της που πλέον είναι finished (δηλαδή μεγαλύτερη του 0). Επίσης ενημερώνουμε τον πίνακα client_table για το ότι ο συγκεκριμένος worker ολοκλήρωσε αυτή τη στιγμή την τελευταία του εργασία (για να γνωρίζουμε πόσο ενεργός είναι ο κάθε client).

### Διαχείριση χρηστών

Έτσι έχουμε ολοκληρώσει τη διαδικασία διαχείρισης των αποτελεσμάτων. Στη συνέχεια εξετάζουμε αν ο worker μας κάλεσε με κάποιο `client_id`. Αν αυτό δεν ισχύει, τότε αναζητούμε ένα `client_id` με τυχαίο αλγόριθμο, το οποίο να είναι διαθέσιμο, το καταχωρούμε στη βάση και το σώζουμε για να του το επιστρέψουμε μαζί με το string της εργασίας.

### Διαχείριση σφαλμάτων

Κατόπιν, εξετάζουμε αν ο worker καλεί την `getJob` δηλώνοντας ότι υπήρξε κάποιο πρόβλημα. Σε περίπτωση που ο worker καλέσει με `client_id` έγκυρο αλλά με `job_id`=-1 τότε την τελευταία φορά είχε διακοπεί η λειτουργία του, επομένως η εργασία που του είχε δοθεί θα πρέπει να μη θεωρείται δοσμένη σε αυτόν πλέον. Βρίσκουμε τη μη ολοκληρωμένη εργασία του συγκεκριμένου worker (λογικά θα είναι μία) και εξετάζουμε αν τα αποτελέσματά της χρειάζονται σε κάποια άλλη εργασία σαν δεδομένα εισόδου. Αν δεν το χρειάζεται κανείς τότε τη σβήνουμε από το `job_table`, διαγράφουμε και τις καταχωρήσεις της στο `arg_out_table` και εξετάζουμε αν ήταν το μοναδικό αντίγραφο της συγκεκριμένης εργασίας. Αν δεν ήταν, προσαρμόζουμε ανάλογα το `state` στα υπόλοιπα αντίγραφα, μειώνοντας την απόλυτη τιμή του (δηλαδή τον αριθμό των αντιγράφων που έχουν δοθεί). Διαφορετικά σβήνουμε την καταχώρηση των δεδομένων εισόδου της εργασίας (που πλέον δε μας είναι απαραίτητα αφού και η ίδια η εργασία έχει διαγραφεί) από τον πίνακα `arg_in_table`. Αν τα αποτελέσματα της εργασίας είναι απαραίτητα σαν είσοδος σε άλλη εργασία ελέγχουμε αν υπάρχει κι άλλο αντίγραφό της. Αν υπάρχει μπορούμε να διαγράψουμε τις καταχωρήσεις της συγκεκριμένης εργασίας από το `job_table` και των αποτελεσμάτων της από το `arg_out_table` και να διορθώσουμε και την απόλυτη τιμή του `state` ώστε να αντικατοπτρίζει το σωστό πλήθος αντιγράφων. Αν δεν υπάρχει άλλο αντίγραφο τότε κάνουμε την εργασία idle, σβήνοντας παράλληλα το `timestamp` της και θέτοντας το `client_id` της -1 στο `job_table` αλλά και στο `arg_out_table` για τα αποτελέσματά της. Έτσι η εργασία είναι ξανά έτοιμη να δοθεί για υπολογισμό.

Ανάθεση νέας εργασίας

Μετά από όλες αυτές τις περιπέτειες, είμαστε πλέον έτοιμοι να αναθέσουμε μια νέα εργασία στον worker. Αυτό βέβαια, μόνο στην περίπτωση που η παράμετρος κλήσης της `getJob` είναι μεγαλύτερη του 0. Επιλέγουμε όλες τις εργασίες του `job_table` που δεν έχουν δοθεί ακόμη (δηλαδή είναι idle, το `state` τους είναι 0) και ψάχνουμε κάποια από αυτές να έχει διαθέσιμα τα δεδομένα εισόδου της, ώστε να είναι έτοιμη να δοθεί προς υπολογισμό. Αυτό το κάνουμε μέσω της συνάρτησης `checkArgAvailable` που καλείται με παράμετρο το id της εργασίας που θέλουμε να ελέγξει.

Η `checkArgAvailable` αναζητά τα στοιχεία της εργασίας στο `arg_in_table` ώστε ταυτόχρονα να ελέγξει ότι η εργασία υπάρχει. Για κάθε δεδομένο εισόδου ελέγχει τον τρόπο (method) περάσματός του. Αν το δεδομένο προέρχεται από αποτέλεσμα άλλης εργασίας (είτε URL είτε inline) τότε αναζητούμε στον `arg_out_table` το συγκεκριμένο αποτέλεσμα της συγκεκριμένης εργασίας που να μην είναι κενό (δηλαδή να είναι διαθέσιμο). Αν αυτό δεν υπάρχει τότε διακόπτουμε την έρευνα των δεδομένων εισόδου και η `checkArgAvailable` επιστρέφει το string "noavail". Αν το δεδομένο υπάρχει, τότε κατασκευάζουμε τον xml κώδικα που το περιγράφει από το `id` του και το URL του πρώτου διαθέσιμου αποτελέσματος που βρήκαμε. Αυτό βέβαια σημαίνει ότι αν το συγκεκριμένο αποτέλεσμα μιας εργασίας ζητηθεί πολλές φορές, κάθε φορά θα δίνουμε το πρώτο διαθέσιμο που βρίσκουμε. Για λόγους load balancing αυτό μπορεί να αλλάξει εύκολα, δίνοντας ένα τυχαίο από τα διαθέσιμα ίδια αποτελέσματα. Αν το δεδομένο δεν προερχόταν από αποτέλεσμα άλλης εργασίας αλλά από σταθερό URL ή από πέρασμα inline τιμής τότε ανάλογα πάλι κατασκευάζουμε τον xml κώδικα που το περιγράφει. Αυτό το string το προσθέτουμε στα προηγούμενα strings των ορισμάτων εισόδου και έτσι, στο τέλος της `checkArgAvailable` έχουμε όλο τον κώδικα xml που περιγράφει τα δεδομένα εισόδου, ένα-ένα με τη σειρά, σε ένα string. Το string αυτό το επιστρέφουμε για να είναι έτοιμη η `getJob` να το ενσωματώσει στο συνολικό xml κώδικα που θα στείλει στον worker.

Επιστρέφοντας στην `getJob` και αφού ελέγξαμε μέσω της `checkArgAvailable` αν η idle εργασία που βρήκαμε έχει διαθέσιμα δεδομένα εισόδου, σε περίπτωση που δεν έχει προχωράμε μέσω του loop μας στην επόμενη idle διεργασία. Αν η εργασία που ελέγξαμε έχει διαθέσιμα δεδομένα εισόδου τότε καλούμε την `createArgOut` η οποία παράγει τον xml κώδικα για τα δεδομένα εξόδου.

Η `createArgOut` παίρνει σαν είσοδο το `id` της εργασίας μας και το `id` του client. Επιλέγει τα δεδομένα εξόδου της εργασίας από το `arg_out_table`. Αν δεν τα βρει σημαίνει ότι έχει γίνει κάποιο λάθος και επιστρέφει το string "noavail". Αν τα βρει αρχίζει να συμπληρώνει το xml string με τα δεδομένα εξόδου του πρώτου αντιγράφου της εργασίας (γιατί μπορεί η εργασία να έχει ξαναδοθεί, οπότε στο `arg_out_table` να υπάρχουν πολλαπλές φορές τα στοιχεία για το ίδιο αποτέλεσμα της ίδιας εργασίας που είναι δοσμένη όμως σε άλλον worker). Σχηματίζουμε λοιπόν το xml string βάσει των στοιχείων του κάθε αποτελέσματος που είναι σταθερά και ανεξάρτητα από τον worker στον οποίο έχει δοθεί η εργασία. Αυτά είναι το `id` του αποτελέσματος, η μέθοδος περάσματός του, το όνομα του αρχείου και το timeout του. Σε περίπτωση που έχει δοθεί ήδη η εργασία σε κάποιον, τότε

για κάθε αποτέλεσμα εισάγουμε μία νέα καταχώρηση στο `arg_out_table` που δείχνει ότι το αποτέλεσμα θα το παράγει και ο νέος μας worker. Διαφορετικά, αν η εργασία δεν έχει δοθεί ακόμα, ενημερώνουμε την καταχώρηση των αποτελεσμάτων της εργασίας αυτής με το `client_id` του worker μας. Στο τέλος έχουμε ένα xml string, που περιλαμβάνει τις πληροφορίες για όλα τα δεδομένα εξόδου της εργασίας μας, και το οποίο επιστρέφουμε στην `getJob`.

Επιστρέφοντας λοιπόν και πάλι στην `getJob` έχουμε πλέον και το xml string για τα αποτελέσματα. Συμπληρώνουμε το συνολικό xml με τα δύο strings που έχουμε πάρει καθώς επίσης με πληροφορίες για το `job_id`, το `client_id`, το URL του κώδικα και την έκδοση του κώδικα. Έτσι καταλήγουμε σε ένα xml string που ακολουθεί τη μορφή που έχουμε προδιαγράψει παραπάνω και μπορεί να το διαβάσει ο worker με τις έτοιμες συναρτήσεις του Castor. Ενημερώνουμε ανάλογα το `job_table` με τη νέα τιμή του `state` (που από 0 γίνεται -1), το τρέχον timestamp μέσω της `createTimeStamp` που περιγράψαμε παραπάνω και το `client_id` στο οποίο δώσαμε την εργασία.

Στην περίπτωση που δεν υπάρχει ανενεργή διαθέσιμη εργασία, ή τα δεδομένα εισόδου όλων των ανενεργών εργασιών δεν είναι διαθέσιμα, αναζητούμε να δώσουμε την πιο παλιά δοσμένη εργασία. Παίρνουμε όλες τις εργασίες που έχουν δοθεί και αναζητούμε για κάθε διαφορετική εργασία το πιο πρόσφατό της αντίγραφο. Από τα πιο πρόσφατα αντίγραφα κάθε εργασίας (για τα οποία έχουμε φτιάξει μία λίστα) επιλέγουμε το πιο παλιό ώστε να ξαναδώσουμε την εργασία που έχει περάσει περισσότερος χρόνος από την τελευταία φορά που δώσαμε. Έτσι φροντίζουμε να δώσουμε την εργασία που έχει περισσότερες πιθανότητες να έχει παρουσιάσει σφάλμα ο worker της, ή που απαιτεί τόσο μεγάλο φόρτο (αφού την επεξεργάζεται τόση ώρα ο worker) που το κόστος ενδεχόμενης απώλειάς της είναι το μεγαλύτερο και επομένως συμφέρει να την αναθέσουμε και πάλι. Ελέγχουμε αν τα δεδομένα εισόδου της είναι διαθέσιμα μέσω της `checkArgAvailable` και σε περίπτωση που δεν είναι την αφαιρούμε από τη λίστα και δοκιμάζουμε με την επόμενη. Αν τα δεδομένα εισόδου είναι διαθέσιμα, αλλάζουμε το `state` της ώστε η απόλυτη τιμή του να δείχνει ένα αντίγραφο παραπάνω και εισάγουμε στο `job_table` τη νέα καταχώρηση ανάθεσης της εργασίας στον worker μας. Κατασκευάζουμε και το xml string για τα αποτελέσματα και τα συνθέτουμε όπως παραπάνω για να προκύψει το xml string ολόκληρης της εργασίας.

Στο τέλος της `getJob` επιστρέφουμε το xml string που έχει προκύψει είτε από ανάθεση νέας idle εργασίας είτε από εκ νέου ανάθεση παλιάς, ήδη δοσμένης εργασίας.

## 3.5 Σχολιασμός των δύο συστημάτων

Στην παράγραφο αυτή θα ασχοληθούμε με το σχολιασμό της δομής και της σχεδίασης του συστήματός μας. Έχουμε αναφερθεί και προηγουμένως στα χαρακτηριστικά των δύο τρόπων λειτουργίας του, εδώ θα επικεντρωθούμε σε δύο σημαντικά ζητήματα απόδοσης τα οποία θα αναλύσουμε και τελικά θα προτείνουμε δύο βελτιώσεις που κατά τη γνώμη μας συνεισφέρουν στη βελτίωσή του.

### 3.5.1 Επικοινωνία

Στο σύστημα Trantor όπως είδαμε υποστηρίζει μία κεντρικοποιημένη δομή της επικοινωνίας, όπου όλα τα δεδομένα περνούν από τον server, καταλήγουν και προέρχονται από αυτόν. Ο worker λαμβάνει όλα τα δεδομένα εισόδου του από τον server. Όλα εκτός από εκείνα που ήδη έχει τοπικά επειδή έτυχε ο ίδιος να υπολογίσει. Στο σύστημα Terminus ο worker λαμβάνει όλα τα δεδομένα εισόδου των εργασιών του από άλλους workers. Στην περίπτωση όμως πάλι που τύχει να διαθέτει κάποια από αυτά τοπικά, δεν χρειάζεται να τα λάβει. Είναι λοιπόν εύλογη η απορία του τι θα γινόταν αν η τοπικότητα των δεδομένων εισόδου ενός worker δεν αφηνόταν στην τύχη. Αν δρομολογούσαμε τις εργασίες με τέτοιο τρόπο που να μη χρειάζεται τόση πολλή επικοινωνία μεταξύ worker-server στην περίπτωση Trantor και worker μεταξύ τους στην περίπτωση Terminus.

#### Προτάσεις

Είναι σαφές ότι η τοπικότητα των δεδομένων στον worker θα ωφελήσει και τις δύο λειτουργίες του συστήματός μας, αφού οι workers μπορούν να εκμεταλλευτούν και στις δύο περιπτώσεις την ύπαρξη των δεδομένων τοπικά για να μην τα κατεβάσουν από άλλη πηγή. Είναι επίσης φανερό ότι κάτι τέτοιο απαιτεί μεταβολές στον τρόπο διαμοιρασμού των εργασιών από το server ώστε οι εργασίες να ανατίθενται σε workers που έχουν όσο το δυνατό μεγαλύτερο μέρος από τα δεδομένα εισόδου τους τοπικά. Ταυτόχρονα, επιθυμούμε να γίνει μια τέτοια αλλαγή στον τρόπο διαμοιρασμού των εργασιών χωρίς να αλλάξουμε τον χαρακτήρα του eager scheduling, της άμεσης ανάθεσης των εργασιών, η οποία μας προσφέρει load balancing στον υπολογισμό και δεν επιτρέπει την καθυστέρηση των γρήγορων workers από τους πιο αργούς. Προτείνουμε λοιπόν την εξής διαδικασία:

- Στο ξεκίνημα του server και σχηματίζοντας το γράφο της εφαρμογής προσπαθούμε να ομαδοποιήσουμε τις εργασίες σε σύνολα παραγωγών δεδομένων και καταναλωτών δεδομένων.
- Ξεκινάμε από μία εργασία που δεν απαιτεί είσοδο από αποτέλεσμα άλλης. Την βάζουμε σε μια καινούρια ομάδα, ως παραγωγό.
- Βρίσκουμε τις εργασίες στις οποίες δίνει άμεσα δεδομένα και τις αποθηκεύουμε στην ίδια ομάδα ως καταναλωτές.
- Για καθένα καταναλωτή βρίσκουμε τους παραγωγούς που τους παρέχουν δεδομένα και αν δεν είναι στην ομάδα τους προσθέτουμε.
- Επαναλαμβάνουμε τη διαδικασία για τους καταναλωτές των παραγωγών που προσθέσαμε. Γενικότερα επαναλαμβάνουμε τη διαδικασία μέχρι να μη γίνονται άλλες προσθήκες παραγωγών και καταναλωτών. Έτσι έχουμε μία ομάδα εργασιών που παράγουν για τους καταναλωτές.
- Επαναλαμβάνουμε για όλες τις εργασίες που δεν απαιτούν είσοδο από άλλες μέχρι να προστεθούν όλες σε μια ομάδα.

- Επαναλαμβάνουμε τα παραπάνω βήματα χρησιμοποιώντας τους καταναλωτές σαν παραγωγούς πλέον και δημιουργώντας έτσι νέες ομάδες.
- Αυτό συνεχίζεται μέχρι να έχουν μπει σε ομάδες όλες οι εργασίες. Πλέον έχουμε ομαδοποιημένες όλες μας τις εργασίες.
- Σε κάθε αίτηση για νέα εργασία από έναν worker, ο server ελέγχει τα δεδομένα που διαθέτει ο worker. Εξετάζει τις ομάδες στις οποίες ανήκουν σαν παραγωγοί τα αποτελέσματα των εργασιών που διαθέτει.
- Ο server επιλέγει μία διαθέσιμη εργασία από την ομάδα στην οποία ο worker έχει τη μεγαλύτερη συμμετοχή με παραγωγούς.
- Αν δεν υπάρχει διαθέσιμη εργασία επιλέγει την επόμενη ομάδα εργασιών και βρίσκει μια διαθέσιμη από εκεί.
- Αν δε βρει προσπαθεί να επιλέξει εργασία από την επόμενη ομάδα από αυτή της προηγούμενης εργασίας που είχε ανατεθεί. Έτσι οι workers παίρνουν εργασίες από άλλες ομάδες ο καθένας.
- Αν δε βρεθεί καμία εργασία, διατρέχουμε κυκλικά όλες τις ομάδες αναζητώντας μία διαθέσιμη.

Ας δούμε όμως τι κερδίζουμε με τη μέθοδο αυτή στην περίπτωση του πολλαπλασιασμού πινάκων με 2x2 υποπίνακες του σχήματος 3.3. Έστω ότι έχουμε 4 workers. Ας δούμε τα βήματα του υπολογισμού:

- Αρχικά οι εργασίες χωρίζονται σε ομάδες Παραγωγών και Καταναλωτών:
  Ομάδα 1: Π:1,3 Κ:9
  Ομάδα 2: Π:2, 4 Κ:10
  Ομάδα 3: Π:5, 7 Κ:11
  Ομάδα 4: Π:6, 8 Κ:12
  Ομάδα 5: Π:9, 10 Κ:13
  Ομάδα 6: Π:11, 12 Κ:14
  Ομάδα 7: Π:13, 14 Κ:15
- Οι 4 workers παίρνουν τις εργασίες 1, 2, 5, 6 λαμβάνοντας τα 8 αρχικά δεδομένα εισόδου από τον server.
- Αφού ολοκληρώσουν τους ανατίθενται οι εργασίες 3, 4, 7, 8 αντίστοιχα, λαμβάνοντας τα άλλα 8 αρχικά δεδομένα εισόδου από το server.
- Στη συνέχεια παίρνουν τις εργασίες 9, 10, 11, 12 αντίστοιχα χωρίς να δεχτούν καθόλου δεδομένα εισόδου από αλλού αφού όλοι τα έχουν.
- Έπειτα θα ανατεθεί η εργασία 13 σε έναν εκ των workers 1, 2 και η 14 σε έναν εκ των workers 3, 4 ανταλλάσσοντας ένα αρχείο εισόδου ο καθένας.

- Τέλος, κάποιος από αυτούς θα πάρει την εργασία 15 ανταλλάσσοντας πάλι ένα αρχείο εισόδου.

Βλέπουμε λοιπόν ότι σε σύνολο 14 αρχείων εισόδου (πέραν των αρχικών που αναγκαστικά θα ληφθούν από τον server) οι workers χρειάστηκε να κατεβάσουν μόνο 3 για να ολοκληρώσουν τον υπολογισμό. Πέρα από την αρχική επιβάρυνση του server για την κατασκευή των ομάδων η οποία γίνεται μόνο μία φορά, η επιβάρυνση στη μνήμη και στον υπολογισμό στη συνέχεια, κατά την αίτηση του worker εξαρτάται από την υλοποίηση. Το κέρδος σε επικοινωνία στο σύστημα, από τη χρήση της μεθόδου αυτής είναι σημαντικό και γίνεται ακόμη μεγαλύτερο αν οι εφαρμογές ανταλλάσσουν μεγάλο όγκο δεδομένων σαν είσοδο και έξοδο.

### 3.5.2 Ανοχή σε σφάλματα

Στο σύστημά μας, σε περίπτωση που δεν υπάρχουν άλλες μη δοσμένες διαθέσιμες εργασίες, ο server δρομολογεί τις ήδη δοσμένες, ξεκινώντας από εκείνη που έχει δοθεί παλαιότερα. Έτσι, αν ένας worker δεν έχει επιστρέψει αποτέλεσμα, η εργασία του δρομολογείται τελικά εκ νέου. Με αυτό τον τρόπο του redundancy στον υπολογισμό είμαστε σίγουροι για τη μη απώλεια των εργασιών που δεν ολοκληρώνονται ή καθυστερεί υπερβολικά η ολοκλήρωσή τους στο συγκεκριμένο worker. Έτσι, ενισχύουμε την αποδοτική λειτουργία του συστήματος.

Στη λειτουργία Trantor, ο server μόλις λάβει την αίτηση για νέα εργασία από τον worker παίρνει και τα URLs των αποτελεσμάτων της προηγούμενης εργασίας που ενδεχομένως ολοκλήρωσε ο ίδιος worker. Αμέσως, ο server τα κατεβάζει τοπικά. Έτσι με αυτή την προσέγγιση δεν προκύπτουν προβλήματα σε περίπτωση που ο χρήστης που παρέχει τα αποτελέσματα αποσυνδεθεί ξαφνικά ή τα αποτελέσματα διαγραφούν λόγω timeout. Αυτά τα ζητήματα όμως υπάρχουν στη λειτουργία Terminus.

Η διαγραφή των αποτελεσμάτων κατά τη λήξη του timeout αντιμετωπίζεται από το σύστημα με περιοδικό έλεγχο των αποτελεσμάτων και των timeouts τους. Μια μέθοδος που καλείται αυτόματα ανά τακτά διαστήματα στην πλευρά του server ελέγχει αν κάποιο αποτέλεσμα από το μητρώο αποτελεσμάτων της βάσης δεδομένων έχει λήξει σύμφωνα με το timeout του. Αν ισχύει κάτι τέτοιο, ελέγχει αν το αποτέλεσμα είναι πλέον απαραίτητο για κάποια εργασία και στην περίπτωση αυτή η εργασία που το παρήγαγε ανατίθεται και πάλι. Έτσι διατηρείται η ακεραιότητα του υπολογισμού σε περίπτωση λήξης της εγκυρότητας ενός αποτελέσματος με εκ νέου ανάθεση της εργασίας. Η περίπτωση αυτή είναι αρκετά σπάνια ώστε να μην επηρεάζει την απόδοση του συστήματος. Τα timeouts εφαρμόζονται για να εξασφαλίζεται η διατήρηση των αποτελεσμάτων στον worker για ένα επαρκές διάστημα, επομένως με αυτό το γνώμονα επιλέγονται και επομένως η ανάγκη παροχής ενός αποτελέσματος πέραν του timeout είναι μάλλον απίθανη.

Το πρόβλημα που δημιουργείται με την απροειδοποίητη αποσύνδεση ενός worker είναι μάλλον μεγαλύτερο και αποτελεί και πεδίο πολλών προβληματισμών γενικότερα στα συστήματα κατανεμημένου υπολογισμού. Στην περίπτωσή μας η αποσύνδεση του worker έχει δύο επιπτώσεις: τη διακοπή του υπολογισμού της τρέχουσας εργασίας και τη διακοπή παροχής προηγούμενων αποτελεσμάτων του

worker. Η διακοπή του υπολογισμού της τρέχουσας εργασίας αντιμετωπίζεται μέσω του redundancy του υπολογισμού που αναφέρθηκε παραπάνω. Η διακοπή παροχής των αποτελεσμάτων αντιμετωπίζεται ως εξής: Σε περίπτωση που μια εργασία ζητήσει δεδομένα εισόδου από κάποιον άλλο worker που δεν είναι διαθέσιμα, ενημερώνει τον server για το σφάλμα. Ο server βρίσκει την εργασία που παρήγαγε το αποτέλεσμα, θεωρεί τα αποτελέσματά της άκυρα (αφού ο worker της δεν τα παρέχει) και αναθέτει στον worker που αντιμετώπισε το σφάλμα, την εργασία που παράγει το αποτέλεσμα αυτό. Έτσι έχουμε ένα backtracking μέχρι να βρούμε μια εργασία με διαθέσιμα δεδομένα εισόδου. Αυτή η αντιμετώπιση εξασφαλίζει την ακεραιότητα του υπολογισμού της εφαρμογής σε περίπτωση σφάλματος. Ας εξετάσουμε όμως την απόδοση του συστήματος.

Στην ακραία περίπτωση που το σύστημα λειτουργεί χωρίς σφάλματα των workers, ή με ελάχιστα και σποραδικά, τότε όπως είναι προφανές θα γίνουν από 0 μέχρι ελάχιστες αναθέσεις εκ νέου της εργασίας. Ας πάρουμε για παράδειγμα τον πολλαπλασιασμό πινάκων που αναφέραμε νωρίτερα. Ας δούμε και πάλι το σχήμα 3.3, που αναπαριστά τον πολλαπλασιασμό πινάκων με 2x2 υποπίνακες και ας υποθέσουμε ότι έχουμε 4 CPUs και ότι ο worker που έχει πάρει το job3 το ολοκληρώνει κανονικά, παίρνει το job6 και αποσυνδέεται. Οι υπόλοιποι παράλληλα έχουν εκτελέσει ο καθένας τις πρώτες τους εργασίες και παίρνουν τις επόμενες. Ολοκληρώνουν και αυτές ενώ ο server περιμένει το αποτέλεσμα του job6 πιστεύοντας παράλληλα ότι το αποτέλεσμα του job3 είναι διαθέσιμο. Αναθέτει λοιπόν το job9 σε κάποιο υγιή worker ο οποίος τον ενημερώνει για το πρόβλημα του job3 και επομένως ο ίδιος worker αμέσως μετά αναλαμβάνει το job3. Μέχρι στιγμής έχουμε μία επιπλέον ανάθεση εργασίας. Οι άλλοι δύο workers παίρνουν τα job10 και job11. Στη συνέχεια και αφού τα job9, και job12 (καθώς και τα επόμενα που εξαρτώνται από αυτά) δεν είναι διαθέσιμα, ο server αναθέτει το job6 που δεν έχει επιστρέψει αποτέλεσμα στον επόμενο worker. Από εκεί και πέρα υπολογίζονται με τη σειρά τα job 9, 12, 13, 14 και 15 και ολοκληρώνεται ο υπολογισμός. Έχουμε λοιπόν 2 αναθέσεις νέων εργασιών στη χειρότερη περίπτωση. Αν το job9 είχε λάβει το αποτέλεσμα του job3 τότε δεν θα χρειαζόταν ο εκ νέου υπολογισμός του και θα είχαμε μόνο μία εκ νέου ανάθεση.

Στην άλλη ακραία περίπτωση, όπου οι workers που αποσυνδέονται είναι όλοι, ή πάρα πολλοί, το πρόβλημα μεγαλώνει σε τέτοιο βαθμό που μπορεί να οδηγήσει στον επαναϋπολογισμό της εφαρμογής από την αρχή, δηλαδή στην ανάθεση όλων των εργασιών εκ νέου.

### Προτάσεις

Όπως βλέπουμε αυτή η σειρά εκ νέου αναθέσεων των εργασιών επιφέρει σημαντική επιβάρυνση στην απόδοση του συστήματος. Φυσικά στην περίπτωση που έχουμε πολλούς workers (όπως π.χ. στο SETI@home) οι εργασίες δίνονται σε πολλούς χρήστες η καθεμία και η αποσύνδεση όλων των παροχέων ενός αποτελέσματος ώστε αυτό να χαθεί είναι πιο σπάνια.

Η ανοχή των σφαλμάτων μέσω του redundancy είναι όπως είδαμε μια ευρέως αποδεκτή μέθοδος. Προτείνουμε λοιπόν μια μέθοδο κατανεμημένης αντιγραφής των αποτελεσμάτων των workers για τη μείωση των επιπτώσεων στα σφάλματα

που προκαλούνται από την ξαφνική αποσύνδεση των workers από το δίκτυό μας. Στο εξής για να μπορέσουμε να εξηγήσουμε καλύτερα τη διαδικασία της αντιγραφής θα συμβολίζουμε σαν worker-c τον worker που αντιγράφει τα αποτελέσματα κάποιου άλλου, και worker-s τον worker που τα δίνει. Η μέθοδος αυτή μπορεί να ενσωματωθεί στο ήδη υπάρχον πρότυπο του Terminus και αναλύτικά περιλαμβάνει τις εξής διαδικασίες:

- Κάθε worker αναλαμβάνει να κρατάει τοπικά ένα αντίγραφο του αποτελέσματος μιας εργασίας ενός άλλου worker. Έτσι ένας worker είναι ταυτόχρονα και worker-c των άλλων, αλλά και worker-s σε άλλους.

- Για το σκοπό αυτό στη βάση δεδομένων του server, στο μητρώο των workers, προστίθεται ένα πεδίο που δείχνει τον τελευταίο worker-s του κάθε worker-c. Στο μητρώο των αποτελεσμάτων προστίθεται ένα πεδίο που δείχνει αν έχει αντιγραφεί το αποτέλεσμα ή όχι.

- Κατά την ολοκλήρωση της εργασίας ενός worker και την υποβολή των URL των αποτελεσμάτων με την επόμενη κλήση RMI στον server, αυτός γίνεται worker-c λαμβάνοντας από τον server μία εικονική εργασία η οποία απλά κατεβάζει και αποθηκεύει το αποτέλεσμα κάποιου worker-s.

- Η επιλογή του αποτελέσματος που θα αντιγραφεί γίνεται με κυκλική επιλογή των worker-s από το μητρώο του server. Γνωρίζουμε για κάθε worker-c τον τελευτάιο worker-s οπότε αναζητούμε από το μητρώο τον επόμενο (με κυκλική διάταξη) ο οποίος έχει παράξει κάποιο αποτέλεσμα το οποίο δεν έχει ακόμη αντιγραφεί. Αφού τον βρούμε ενημερώνουμε το μητρώο του worker-c με τον νέο τελευταίο worker-s.

- Αφού επιλεγεί ο worker-s η εργασία δίνεται στον worker-c και ολοκληρώνεται επιστρέφοντας την επιβεβαίωση στο server και την αίτηση νέας εργασίας.

- Ο server αυτή τη φορά αναθέτει στον worker μια κανονική εργασία, ενημερώνοντας ταυτόχρονα τη βάση δεδομένων στον πίνακα των αποτελεσμάτων ότι τα συγκεκριμένα αποτελέσματα της εργασίας του worker-s τα διαθέτει και ο worker-c (αντιγράφοντας ουσιαστικά τις καταχωρήσεις των αποτελεσμάτων της συγκεκριμένης εργασίας και αντικαθιστώντας τα URLs με αυτά του worker-c).

- Σε περίπτωση αποσύνδεσης ενός worker απροειδοποίητα, ο server ενημερώνεται από την εργασία που χρειάζεται τα δεδομένα που διαθέτει. Ο server τότε δίνει στην εργασία το URL του αντιγράφου των δεδομένων. Επίσης αναλαμβάνει να βρει όλα τα δεδομένα και τα αντίγραφα δεδομένων που διέθετε ο worker που αποσυνδέθηκε και να τα δρομολογήσει για αντιγραφή.

Ουσιαστικά, στο γράφο του υπολογισμού της εφαρμογής προσθέτουμε μετά από την εκτέλεση μίας εργασίας ένα ακόμη βήμα, αυτό του κατεβάσματος των αποτελεσμάτων κάποιας άλλη εργασίας. Παρεμβάινουμε λοιπόν, στον αρχικά δοσμένο γράφο με την εξωτερική προσθήκη βημάτων σε αυτόν. Η δομή φυσικά του

γράφου και η ροή που ακολουθείται δεν αλλάζει ουσιαστικά. Έτσι κάθε worker διατηρεί αποτελέσματα από εργασίες άλλων και ανά πάσα στιγμή (σχεδόν) κάθε αποτέλεσμα υπάρχει σε δύο (τουλάχιστον) αντίγραφα στο σύστημά μας. Ο αριθμός φυσικά των αντιγράφων που δημιουργούνται μπορεί να αυξηθεί αν χρειάζεται για μεγαλύτερη αξιοπιστία, όμως εδώ θα εξετάσουμε την απλή περίπτωση της διπλής υπόστασης των αποτελεσμάτων (δηλαδή ενός αντιγράφου πέρα από το αυθεντικό αποτέλεσμα).

Ο αριθμός των αποτελεσμάτων άλλων που διατηρεί ο κάθε worker είναι όσα και τα αποτελέσματα που έχει ο ίδιος παράξει, αφού αντιγράφει ένα αποτέλεσμα για κάθε επιστροφή δικού του αποτελέσματος. Η επιβάρυνση στην επικοινωνία και στον τοπικό αποθηκευτικό χώρο συνολικά του συστήματος είναι διπλάσια. Επίσης η επιβάρυνση αυτή αφορά τους workers και επομένως μιλάμε για μια κατανομή της συνολικής επιβάρυνσης σε όλους τους workers. Για τον server η επιβάρυνση είναι πολύ μικρή και αφορά μερικές προσθήκες στη βάση δεδομένων και κάποιους ελέγχους και αναβαθμίσεις αυτής επιπλέον. Στο server βέβαια το πλήθος των εργασιών που δρομολογούνται διπλασιάζεται αφού για κάθε εργασία προστίθεται και μία πλασματική εργασία αντιγραφής. Η επιβάρυνση όμως είναι μικρότερη από τη διπλάσια καθώς οι πλασματικές εργασίες δεν καταχωρούνται στη βάση δεδομένων. Τέλος, οι εργασίες αυτές δεν έχουν παρά ελάχιστο πραγματικά κόστος στους workers αφού πέρα από την αντιγραφή δεν κάνουν καμία επεξεργασία.

Από την άλλη, τα πλεονεκτήματα αυτής της μεθόδου είναι πολλαπλάσια. Ας θεωρήσουμε ότι έχουμε $n$ workers στο σύστημά μας και ο καθένας από αυτούς έχει διεκπεραιώσει (για λόγους ευκολίας τον ίδιο αριθμό) $m$ εργασίες. Επίσης ας θεωρήσουμε την πιθανότητα αποσύνδεσης ενός worker $e$. Σε περίπτωση που αποτύχει κάποιος (με πιθανότητα $e$), τα αποτελέσματά του αναπαράγονται με πολύ χαμηλό κόστος για όλες τις εργασίες του. Χωρίς το σύστημά μας το αναμενόμενο κόστος για εκ νέου αναθέσεις παλιών εργασιών θα ήταν περίπου $e{\cdot}m$. Το κέρδος στην περίπτωση αυτή είναι προφανές. Τι γίνεται όμως όταν αποτύχουν 2 workers, όπου χάνονται εντελώς τα αποτελέσματα που έχουν κοινά;

Με τη μέθοδο κυκλικής αντιγραφής αποτελεσμάτων, κάθε worker έχει μόνο μία μερίδα των αποτελεσμάτων ενός άλλου. Η μέθοδος αυτή εξασφαλίζει την κατανομή των αποτελεσμάτων ενός worker σε όλους τους άλλους (αφού ο καθένας εξυπηρετεί αποτελέσματα κυκλικά και ισορροπημένα όλων των άλλων). Αν λοιπόν κάθε worker έχει κάνει $m$ εργασίες τότε καθένας άλλος έχει $m/(n\text{-}1)$ από αυτά τα αποτελέσματα. Αν αποσυνδεθούν 2 χρήστες σε μικρό διάστημα τα αποτελέσματα που θα χαθούν θα είναι $2{\cdot}m/(n\text{-}1)$. Η πιθανότητα όμως να συμβεί αυτό είναι $e^2$ Έτσι το αναμενόμενο κόστος για εκ νέου αναθέσεις παλιών εργασιών είναι $e^2{\cdot}2m/(n\text{-}1)$.

Οι δύο μέθοδοι που παρουσιάστηκαν, αυτή της «έξυπνης» ανάθεσης των εργασιών σε workers και η αντιγραφή των αποτελεσμάτων, αποκτούν ακόμη μεγαλύτερο ενδιαφέρον αν συνδυαστούν. Θα μπορούσαμε να εφαρμόσουμε μια «έξυπνη» αντιγραφή των αποτελεσμάτων ώστε ταυτόχρονα να εξασφαλίσουμε μεγαλύτερη αξιοπιστία του συστήματος αλλά και να εκμεταλλευτούμε μελλοντικά το locality των δεδομένων στον worker που θα τα χρειαστεί για να γλιτώσουμε την περαιτέρω επικοινωνία των workers. Έτσι, ένα μέρος της επιπλέον επικοινωνίας και του επιπλέον χώρου που θα κατανάλωνε ο χρήστης για τη μεγαλύτερη αξιοπιστία του

συστήματος θα αξιοποιηθεί αφού τελικά ούτως η άλλως θα τα κατανάλωνε για να χρησιμοποιήσει τα δεδομένα που κάνει backup για είσοδο στη δική του εργασία.

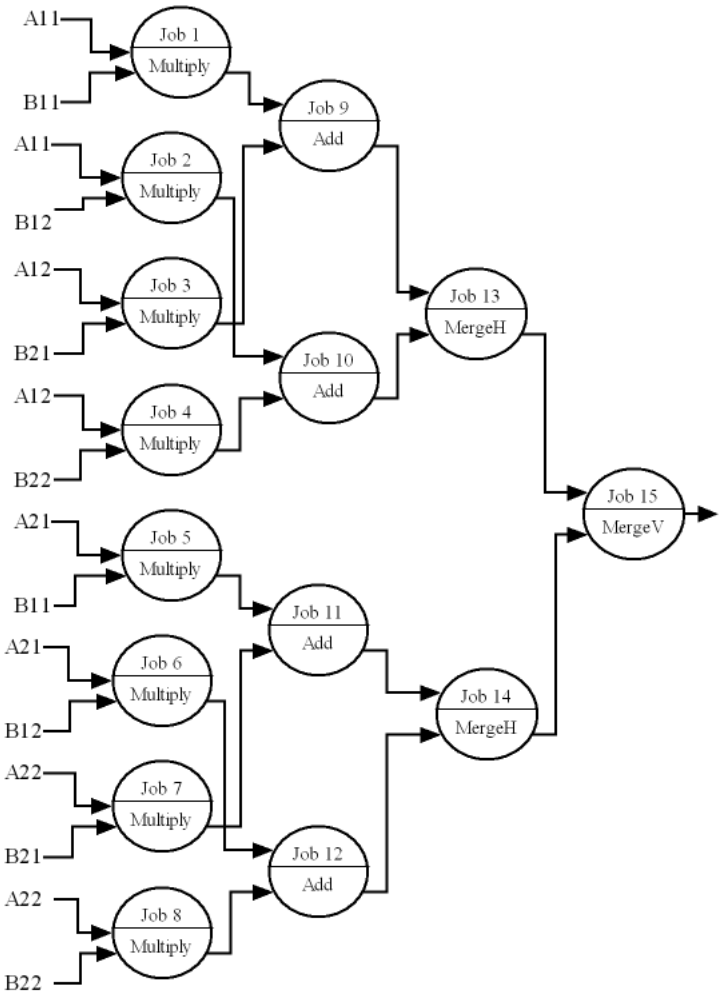
# Κεφάλαιο 4

# Πειράματα

Για τη διεξαγωγή των πειραμάτων χρησιμοποιήθηκε ο υπολογιστής πλέγματος του Πολυτεχνείου Κρήτης που αποτελείται από 44 HP Proliant BL465c server blade, εξοπλισμένους ο καθένας με 2 επεξεργαστές AMD Opteron στα 2.6GHz, με 2MB Cache, 4 GB μνήμη, δικτύωση Gigabit Ethernet και Λειτουργικό Σύστημα Scientific Linux 5.4. Ο ένας κόμβος χρησιμοποιήθηκε σαν Server και οι υπόλοιποι σαν workers.

Η αρχιτεκτονική του grid επιτρέπει τη δρομολόγηση μέχρι τεσσάρων εργασιών ανά κόμβο (μια εργασία για κάθε πυρήνα από τους 2 πυρήνες, του καθενός από τους 2 επεξεργαστές του κόμβου). Μια τέτοια δρομολόγηση εργασιών θα περιέπλεκε τις μετρήσεις καθώς θα προέκυπταν πολύ διαφορετικοί χρόνοι για την εκτέλεση 40 εργασιών σε 10 κόμβους (με 4 εργασίες ανά κόμβο) από ό,τι σε 40 κόμβους (με μία εργασία ανά κόμβο). Κατά τις δοκιμές λοιπόν δόθηκε μεγάλη προσοχή ώστε οι εργασίες να δίνονται σε διαφορετικούς κόμβους, αλλά και οι υπόλοιποι κόμβοι να μην απασχολούνται από άλλες εργασίες που ενδεχομένως θα καθυστερούσαν την εκτέλεση των εργασιών. Πριν από κάθε πείραμα, σε κάθε κόμβο που θα συμμετείχε σε αυτό, αντιγράφονταν τοπικά τα απαραίτητα αρχεία για την εκτέλεση του worker, ώστε να μη χρησιμοποιείται κανένας άλλος κοινόχρηστος αποθηκευτικός ή δικτυακός πόρος κατά τη διάρκεια της δοκιμής, πέρα από την ανταλλαγή των δεδομένων που απαιτούσε το πρόγραμμα.

Το πρόβλημα που χρησιμοποιήθηκε για τις δοκιμές ήταν ο πολλαπλασιασμός μεγάλων πινάκων χωρισμένων σε υποπίνακες. Τα στοιχεία των πινάκων ήταν αριθμοί κινητής υποδιαστολής. Τον πολλαπλασιασμό πινάκων με 2x2 υποπίνακες τον εξετάσαμε παραπάνω και παραθέτουμε και πάλι το σχήμα 4.1 για καλύτερη κατανόηση της δομής της εφαρμογής. Στις δοκιμές μας χρησιμοποιήθηκαν πίνακες με 8x8 υποπίνακες των 1000x1000 στοιχείων ο κάθε υποπίνακας, η διαδικασία όμως είναι πολύ παρόμοια (αν και αρκετά πιο πολύπλοκη) με αυτή του πολλαπλασιασμού πινάκων με 2x2 υποπίνακες. Πλέον έχουμε να εκτελέσουμε 512 πολλαπλασιασμούς υποπινάκων, 448 προσθέσεις (256, κατόπιν 128 και τελικά 64) , 56 οριζόντιες ενώσεις (αρχικά 32 μετά 16 και έπειτα 8) και 7 κατακόρυφες ενώσεις υποπινάκων (4 αρχικές, 2 στη συνέχεια και 1 για την συγκεντρωτική ένωση ολόκληρου του πίνακα) .




Σχήμα 4.1: Διάγραμμα πολλαπλασιασμού πινάκων με 2x2 υποπίνακες.

Η φύση του προβλήματος δείχνει ότι δεν εμπίπτει στην κατηγορία embarrassingly parallel. Υπάρχει μεγάλος όγκος επικοινωνίας για την είσοδο και έξοδο των εργασιών, καθώς σε κάθε βήμα υπολογισμού η κάθε εργασία απαιτεί τα αποτελέσματα από δύο εργασίες του προηγούμενου βήματος. Υπάρχει επίσης μία γραμμικότητα στην εκτέλεση του υπολογισμού όσο πλησιάζουμε προς το τέλος του και οι πίνακες συγκεντρώνονται ολοένα και σε λιγότερους workers για να ενωθούν (αρχικά σε 4, μετά σε 2 και τελικά σε 1 worker). Επομένως, υπάρχουν κομμάτια του υπολογισμού (προς το τέλος αυτού) τα οποία δεν είναι επαρκώς παραλληλίσιμα και ουσιαστικά ο υπολογισμός τείνει να εκτελεστεί σειριακά.

## 4.1 Χρόνοι εκτέλεσης

Έχοντας τα παραπάνω υπόψιν θα δούμε τα αποτελέσματα των μετρήσεων για 1, 10, 20, 30 και 40 workers στις δύο λειτουργίες του συστήματός μας, τη λειτουργία Trantor με κεντρικοποιημένη επικοινωνία που περνά όλη μέσω του server και τη λειτουργία Terminus με αποκεντρωμένη επικοινωνία, μεταξύ των worker.

| workers | Trantor Mode | Terminus Mode |
|---|---|---|
| 1 | 70323 | 69092 |
| 10 | 17916 | 16891 |
| 20 | 14807 | 14290 |
| 30 | 14230 | 13406 |
| 40 | 14045 | 12971 |

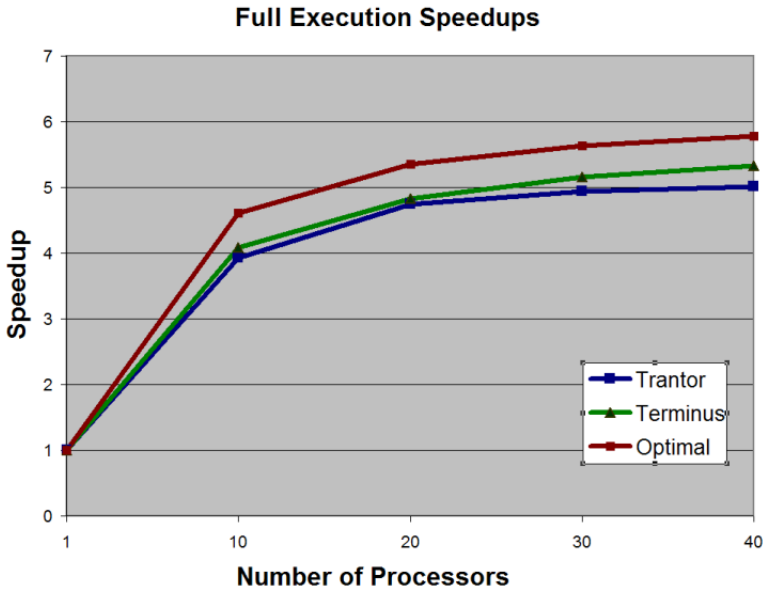
Πίνακας 4.1: Συνολικοί Χρόνοι Εκτέλεσης (sec)

Παρατηρούμε σαφώς μικρότερο χρόνο εκτέλεσης για τη λειτουργία Terminus σε σχέση με τη λειτουργία Trantor σε κάθε πείραμα, από πολύ νωρίς, με λίγους workers, μέχρι και στις τελευταίες μετρήσεις με 30 ή 40 workers. Μάλιστα, η διαφορά είναι εμφανής και στη σειριακή εκτέλεση του πειράματος, με έναν μόνο worker. Αυτό ίσως μοιάζει παράδοξο: Στην περίπτωση αυτή ο worker δε ζητά ποτέ από τον server δεδομένα εισόδου, με εξαίρεση τους αρχικούς πίνακες, καθώς ό,τι θα χρειαστεί προέρχεται από προηγούμενη εργασία του ίδιου. Ωστόσο, κάθε φορά που ολοκληρώνεται μια εργασία, ο Trantor ούτως ή άλλως κατεβάζει τα αποτελέσματά της για να τα έχει στη διάθεση όποιου τα ζητήσει. Επομένως, και σε αυτή την περίπτωση ο φόρτος του Trantor είναι μεγαλύτερος από τον Terminus, ο οποίος δεν κατεβάζει τα αποτελέσματα αλλά μόνο κρατάει το URI τους.

| workers | Trantor Mode | Terminus Mode | Optimal Speedup |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 10 | 3,9252 | 4,0905 | 4,6045 |
| 20 | 4,7493 | 4,8350 | 5,3571 |
| 30 | 4,9419 | 5,1538 | 5,6385 |
| 40 | 5,0070 | 5,3267 | 5,7837 |

Πίνακας 4.2: Speedups συνολικής εκτέλεσης

Οι διαφορές ίσως μοιάζουν μικρές βλέποντας το συνολικό χρόνο. Αυτό, όμως, είναι αναμενόμενο, καθώς έχουμε ένα πολύ μεγάλο τμήμα της δοκιμής που δεν είναι πλήρως ή και καθόλου παραλληλίσιμο. Για την ακρίβεια τα πρώτα τμήματα του υπολογισμού, τα οποία είναι πλήρως έως ικανοποιητικά παραλληλίσιμα (512 πολλαπλασιασμοί, 256 + 128 + 64 προσθέσεις και 32 ενώσεις) κατά τη σειριακή εκτέλεση των δοκιμών διαπιστώσαμε ότι ολοκληρώθηκαν σε χρόνο που αντιστοιχεί μόνο στο 56% περίπου του συνολικού χρόνου εκτέλεσης. Επομένως 46% της



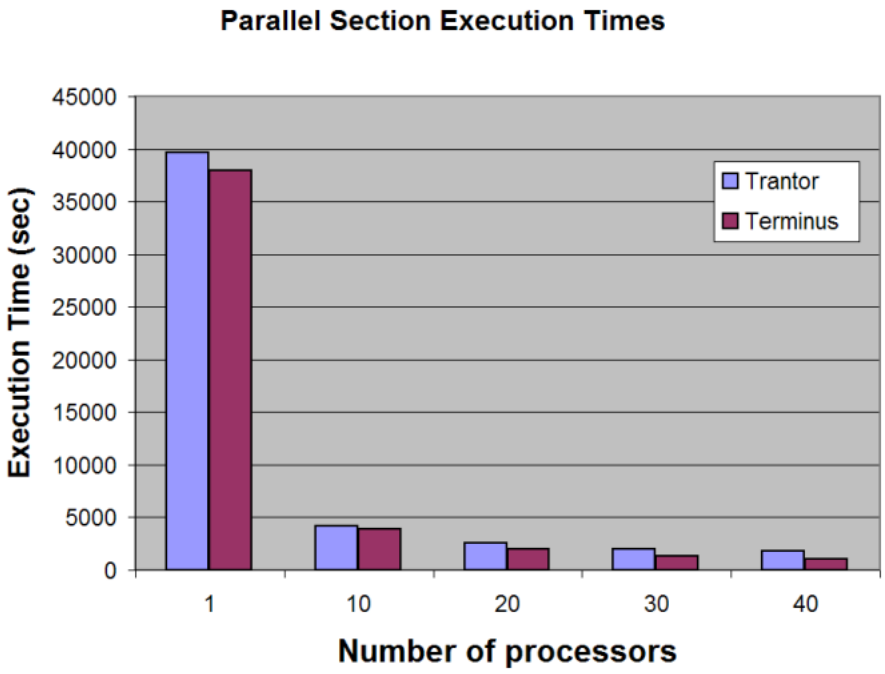

Σχήμα 4.2: Speedups συνολικής εκτέλεσης.

εκτέλεσης είναι από λίγο έως καθόλου παραλληλίσιμη, καθώς αποτελούνται από 16 + 8 οριζόντιες ενώσεις πινάκων και 4 + 2 + 1 κατακόρυφες ενώσεις, που μάλιστα η καθεμιά έχει πολλαπλάσιο όγκο επεξεργασίας από την προηγούμενη. Σύμφωνα με το νόμο του Amdahl [4] η μέγιστη αναμενόμενη επιτάχυνση (speedup) περιορίζεται από το ποσοστό του μη παραλληλίσιμου χρόνου του προγράμματος. Με μια πιο λεπτομερή ανάλυση των χρόνων των δοκιμών, υπολογίσαμε τα βέλτιστα speedups για καθένα πλήθος κόμβων που χρησιμοποιήσαμε και παρατίθενται στον Πίνακα 4.2 μαζί με τα πραγματικά στοιχεία για τις δύο λειτουργίες του server.

Παρατηρούμε ότι η επιτάχυνση της εκτέλεσης πλησιάζει ικανοποιητικά τη βέλτιστη θεωρητική επιτάχυνση. Μάλιστα αυτό είναι περισσότερο εμφανές στη λειτουργία Terminus, όπου η καμπύλη των επιταχύνσεων ακολουθεί την καμπύλη των βέλτιστων επιταχύνσεων σε αντίθεση με τη λειτουργία Trantor, όπου παρατηρούμε μια κάμψη από τους 30 workers και μετά (Σχήμα 4.2).

Για να δούμε πιο καθαρά τις διαφορές στην επίδοση των δύο συστημάτων, στο εξής θα επικεντρωθούμε στο παράλληλο τμήμα της εκτέλεσης των δοκιμών.

Στο παράλληλο τμήμα της εκτέλεσης, πλέον, οι πίνακες των χρόνων εκτέλεσης και των επιταχύνσεων δείχνουν ξεκάθαρα τις επιδόσεις των δύο συστημάτων. Πτωτικές πορείες για τους χρόνους εκτέλεσης και στις δύο λειτουργίες, με τον Terminus να βρίσκεται σταθερά χαμηλότερα από τον Trantor και μάλιστα με τη διαφορά να μεγαλώνει καθώς αυξάνονται οι κόμβοι επεξεργασίας. Στο speedup, όπως φαίνεται στον Πίνακα 4.4 τα πράγματα φαίνονται ακόμη πιο καθαρά: Ο Terminus ακολουθεί με ικανοποιητική σταθερότητα το βέλτιστο speedup, όπως το υπολο-

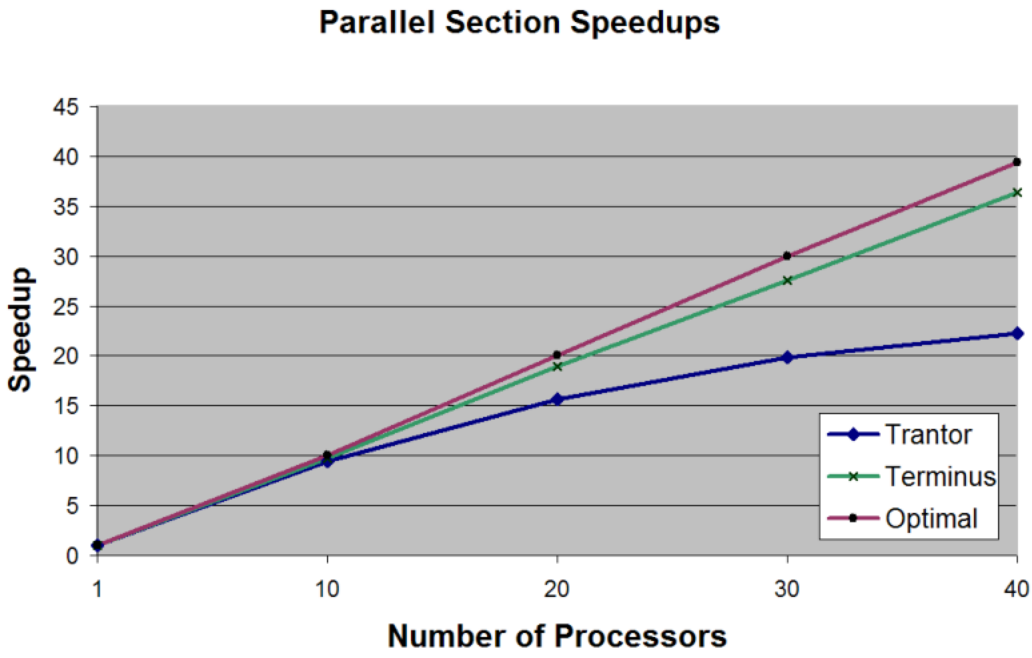



Σχήμα 4.3: Χρόνοι εκτέλεσης παράλληλου τμήματος

| workers | Trantor Mode | Terminus Mode |
|---|---|---|
| 1 | 39647 | 37931 |
| 10 | 4217 | 3914 |
| 20 | 2532 | 2006 |
| 30 | 2000 | 1377 |
| 40 | 1783 | 1044 |

Πίνακας 4.3: Χρόνοι Εκτέλεσης παράλληλου τμήματος (sec)

γίσαμε για το παράλληλο τμήμα, σύμφωνα με το νόμο του Amdahl) σε όλες τις μετρήσεις. Ακόμα και στη μέτρηση με 40 υπολογιστικούς κόμβους ο Terminus μας δίνει speedup μεγαλύτερο από 36, με το θεωρητικό βέλτιστο λίγο πάνω από το 39. Αντίθετα, ο Trantor δείχνει τις πρώτες ενδείξεις κάμψης από τους 20 υπολογιστικούς κόμβους κιόλας, ενώ στους 30 και τους 40 φαίνεται ότι φτάνει στα όριά του και ότι ακόμα κι αν υπήρχαν κι άλλοι κόμβοι δε θα ήταν σε θέση να τους εξυπηρετήσει και να εκμεταλλευτεί την ισχύ τους για να ολοκληρώσει γρηγορότερα τη δοκιμή. Το αντίστοιχο διάγραμμα (Σχήμα 4.4) δείχνει ακόμη πιο παραστατικά αυτή την τάση του Terminus να ακολουθεί τη βέλτιστη καμπύλη, με τον Trantor να




Σχήμα 4.4: Speedups παράλληλου τμήματος.

κινείται σε πολύ χαμηλότερα επίπεδα επιταχύνσεων.

Αυτή η διαφαινόμενη διαφορά μένει να αποτυπωθεί σε ένα συγκριτικό πίνακα. Ο Πίνακας 4.5 δείχνει τα ποσοστά της βελτίωσης του χρόνου εκτέλεσης του παράλληλου τμήματος ανάμεσα στον Terminus και τον Trantor. Παρατηρούμε μια διαρκώς αυξανόμενη διαφορά, ξεκινώντας από 4,3 % για τη γραμμική εκτέλεση της δοκιμής και διαρκώς αυξανόμενη για να καταλήξει τελικά στο 41,4 % για την εκτέλεση με 40 υπολογιστικούς κόμβους. Αυτό είναι ακριβώς ό,τι περιμέναμε, καθώς το κέρδος από τη χρήση της κατανεμημένης επικοινωνίας γίνεται εμφανές όσο πιο πολλοί είναι οι workers και επομένως όσο πιο μεγάλος είναι ο φόρτος που πρέπει να αντιμετωπίσει ο server.

## 4.2 Επικοινωνία

Ας δούμε όμως τι γίνεται με την επικοινωνία του server, για να καταλάβουμε καλύτερα για ποιό λόγο εμφανίζεται η βελτίωση αυτή με τον κατανεμημένο τρόπο λειτουργίας.

Στη λειτουργία Terminus ο server, αρχικά παρέχει όλα τα αρχεία που αφορούν

| workers | Trantor Mode | Terminus Mode | Optimal Speedup |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 10 | 9,4017 | 9,6911 | 10 |
| 20 | 15,6584 | 18,9088 | 20 |
| 30 | 19,8235 | 27,5461 | 30 |
| 40 | 22,2361 | 36,3324 | 39,4221 |

Πίνακας 4.4: Parallel section Speedup

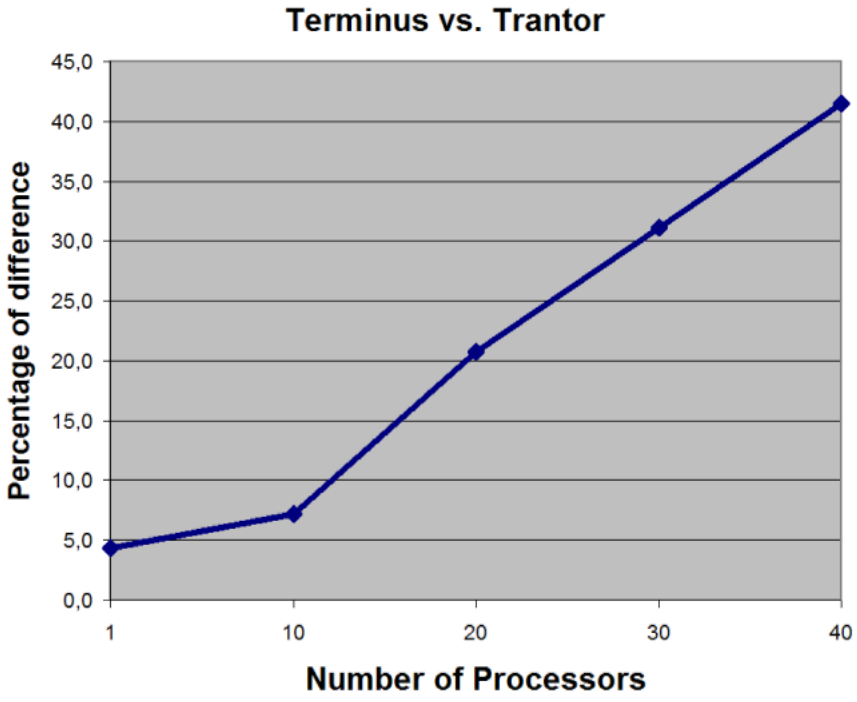
| workers | % of improvement |
|---|---|
| 1 | 4,328 |
| 10 | 7,185 |
| 20 | 20,774 |
| 30 | 31,15 |
| 40 | 41,447 |

Πίνακας 4.5: Terminus vs. Trantor - ποσοστά βελτίωσης

τις πρώτες εργασίες και τα οποία δεν προκύπτουν από προηγούμενους υπολογισμούς. Οι εργασίες αυτές είναι 512, το μέγεθος των 2 αρχείων που ο server δίνει για κάθε εργασία είναι 4.5 Mb αθροιστικά και θεωρούμε ότι οι workers ζητούν και τα 1024 αρχεία (δηλαδή δεν τυχαίνει να έχουν ήδη κάποιο από αυτά από προηγούμενη εκτέλεση). Συνολικά λοιπόν ο server δίνει 512x4,5 = 2304 Mb. Για τις επόμενες εργασίες ο server δεν λαμβάνει αρχεία, αλλά ούτε και παρέχει. Οι workers ανταλλάσσουν τα αποτελέσματά τους μεταξύ τους και έτσι η επικοινωνία του server περιορίζεται στην ανταλλαγή του job.xml, το οποίο είναι της τάξης του 1 Kb, δηλαδή αμελητέο. Στη λειτουργία Trantor ο server παρέχει με τον ίδιο τρόπο τα πρώτα 512 αρχεία. Παράλληλα όμως, για κάθε εργασία που εκτελείται, παίρνει από τους workers τα αποτελέσματά της. Συνολικά δηλαδή τα εισερχόμενα δεδομένα για τον server είναι:

7,5 `Mb` από καθεμιά από τις 512 εργασίες `Multiply`
7,5 `Mb` από καθεμιά από τις 448 (256 + 128 +64) εργασίες `Add`
15 `Mb` από καθεμιά από τις 32 πρώτες εργασίες `MergeH` (οριζόντια ένωση)
30 `Mb` από καθεμιά από τις 16 επόμενες εργασίες `MergeH`
60 `Mb` από καθεμιά από τις 8 επόμενες εργασίες `MergeH`
120 `Mb` από καθεμιά από τις 4 πρώτες εργασίες `MergeV` (κατακόρυφη ένωση)
240 `Mb` από καθεμιά από τις 2 εργασίες `MergeV`
480 `Mb` από την τελευτάια εργασία `MergeV`
Σύνολο: 10080 `Mb`

Επίσης ο server δίνει όλα τα αρχεία εισόδου στους workers που τα χρειάζονται. Κάποια αρχεία τα έχουν ήδη αυτοί, από δική τους προηγούμενη εκτέλεση εργασίας, επομένως δε χρειάζεται πάντα να τα κατεβάσουν από τον server. Έτσι η εξερχόμενη




Σχήμα 4.5: Terminus vs. Trantor - Ποσοστιαίες διαφορές χρόνων εκτέλεσης στο παράλληλο τμήμα.

επικοινωνία του server είναι:

`512x4,5 Mb` στις 512 εργασίες `Multiply`
2 αρχεία των 7,5 `Mb` στις 448 (256 + 128 +64) εργασίες `Add`
2 αρχεία των 7,5 `Mb` στις 32 εργασίες `MergeH`
2 αρχεία των 15 `Mb` στις 16 εργασίες `MergeH`
2 αρχεία των 30 `Mb` στις 8 εργασίες `MergeH`
2 αρχεία των 60 `Mb` στις 4 εργασίες `MergeV`
2 αρχεία των 120 `Mb` στις 2 εργασίες `MergeV`
2 αρχεία των 240 `Mb` στην τελευτάια εργασία `MergeV`

Στις 448 εργασίες `Add`, κατά προσέγγιση 1 στους 2 workers έχει ένα από τα δύο αρχεία ήδη. Δηλαδή χρειάζεται η μεταφορά των 3/4 των αρχείων. Στις επόμενες εργασίες όλοι οι workers έχουν κατά προσέγγιση ένα αρχείο, επομένως γίνονται οι μισές μεταφορές. Με αυτούς τους υπολογισμούς, ο server του Trantor παίρνει 10080 Mb και δίνει 8784 Mb δηλαδή ανταλλάσσει συνολικα 18864 Mb. Ο Terminus έχει συνολική επικοινωνία 2784 Mb και μάλιστα μόνο στην αρχή και στο τέλος της επεξεργασίας, χωρίς, επομένως, να επηρεάζεται ολόκληρη η λειτουργία

του συστήματος. Είναι κατανοητό, λοιπόν, το κέρδος στο κόστος της επικοινωνίας από τη χρήση της λειτουργίας Terminus και οι διαφορές ανάμεσα στις δύο λειτουργίες ήταν αναμενόμενες. Οι διαφορές αυτές, όπως αποτυπώνονται και στο Σχήμα 4.5 δείχνουν ότι ένα σύστημα κατανεμημένου υπολογισμού με αποκεντρωμένη επικοινωνία έχει σημαντικά μεγαλύτερη ικανότητα κλιμάκωσης σε σχέση με τα παραδοσιακά συστήματα που στηρίζονται στην συγκεντρωτική κεντρική επικοινωνία μέσω του server. Ακόμη και αντιμετωπίζοντας ένα πρόβλημα που δεν ήταν embarrassingly parallel φάνηκε ότι μπορούμε να πετύχουμε σαφή βελτίωση στην επιτάχυνση, επιβεβαιώνοντας έτσι την αρχική μας υπόθεση.

# Κεφάλαιο 5

# Συμπεράσματα

Σε αυτή την εργασία εξετάσαμε την περιοχή του κατανεμημένου υπολογισμού, μια περιοχή διαρκώς αναπτυσσόμενη, με έρευνα που βρίσκει συνεχώς μεγαλύτερη εφαρμογή, όσο τα δίκτυα των υπολογιστών εξελίσσονται και η διεισδυση αξιόλογης υπολογιστικής ισχύος φτάνει σε ολοένα και περισσότερες και πιο ετερογενείς συσκευές, που είναι έτοιμες να συνδεθούν στο Internet και να μοιραστούν την υπολογιστική τους ισχύ. Η ανάγκη της κατανομής του υπολογιστικού φόρτου αφορά μια πληθώρα εφαρμογών, που δεν περιορίζονται ως προς την κατηγορία τους στις embarrasingly parallel εφαρμογές με τις οποίες κυρίως ασχολούνται οι επιτυχημένες πλατφόρμες κατανεμημένου υπολογισμού ως τώρα. Προτείναμε ένα σύστημα κατανεμημένου υπολογισμού γενικών εφαρμογών, γραμμένο σε Java, αποκεντρωμένου, βασισμένου στις αρχές των peer-to-peer δικτύων, με ελάχιστη υπολογιστική και επικοινωνιακή επιβάρυνση και παρέμβαση του server, ο οποίος απλά φέρνει σε επικοινωνία τους peers. Προσπαθήσαμε να προσδώσουμε ανοχή στα σφάλματα και ασφάλεια στο σύστημα και χρησιμοποιήσαμε μεθόδους redundancy για τον υπολογισμό των εργασιών. Προτείναμε δύο μεθόδους που θα βελτιώσουν την απόδοση του συστήματος κάτω από αντίξοες συνθήκες και θα εκμεταλλευτούν την τοπικότητα των δεδομένων κάθε worker. Τέλος, διαπιστώσαμε ότι οι διαφορές στην απόδοση του συστήματος που προτείνεται, σε σχέση με την αντίστοιχη παραδοσιακή υλοποίηση είναι ορατές, από μικρό φόρτο του server ακόμη, και μεγαλώνουν καθώς ο φόρτος αυτός αυξάνει, κάτι που δείχνει ότι αυτή η εναλλακτική πρόταση μας δίνει μια καινούρια μέθοδο πιο αποδοτικής αντιμετώπισης εκείνων των προβλήμάτων που δεν είναι embarrasingly parallel.

Η έρευνα στην περιοχή εξακολουθεί να εξελίσσεται ραγδαία και ως εκ τούτου είναι συναρπαστική. Εξίσου συναρπαστικό όμως είναι και το αντικείμενό της, η δημιουργία κοινοτήτων εθελοντικής προσφοράς υπολογιστικής ισχύος, η παροχή των εχεγγύων για την ασφαλή συμμετοχή οποιουδήποτε θέλει να προσφέρει, όσο αδύναμος και αν είναι και η ανάγκη ώστε αυτή η συμμετοχή να πιάσει τόπο, να είναι αποδοτική, να μην πάει χαμένη.

Ο κόσμος στον οποίο ζούμε χαρακτηρίζεται από την ιδιώτευση και την αποχή από οτιδήποτε συλλογικό. Όσοι λοιπόν ασχολούμαστε με αυτή την περιοχή, προχωρούμε συνειδητά ή ασυνείδητα ενάντια στο ρεύμα της εποχής μας, προσπα-

θούμε να ενώσουμε τους ανθρώπους πίσω από τους υπολογιστές για έναν κοινό σκοπό, προσπαθούμε να αλλάξουμε τον κόσμο. Ίσως να κυνηγάμε μια ουτοπία. Όμως κάποιος είχε πει: «Τι είναι ουτοπία; Είναι αυτό, που όταν πλησιάζεις ένα βήμα, απομακρύνεται δυο. Όταν πλησιάζεις δυο, απομακρύνεται τέσσερα. Όταν πλησιάζεις τρία βήματα, φεύγει έξι μακριά. Και τότε σε τι χρειάζεται η ουτοπία; Σε βοηθά να προχωράς»

Παράρτημα Α΄

# Τα αρχεία περιγραφής των εργασιών

## Α΄.1 Το αρχείο του **DAG** για πολλαπλασιασμό πινάκων με **2x2** υποπίνακες

```
1 1 http://147.27.1.191:8081/Multiply.jar 1@1@http://147.27.1.191:8081/A11.txt@-1!2@1@http://147.27.1.191:8081/B11.txt@-1 1@1@ArrC.txt@120
2 1 http://147.27.1.191:8081/Multiply.jar 1@1@http://147.27.1.191:8081/A11.txt@-1!2@1@http://147.27.1.191:8081/B12.txt@-1 1@1@ArrC.txt@121
3 1 http://147.27.1.191:8081/Multiply.jar 1@1@http://147.27.1.191:8081/A12.txt@-1!2@1@http://147.27.1.191:8081/B21.txt@-1 1@1@ArrC.txt@122
4 1 http://147.27.1.191:8081/Multiply.jar 1@1@http://147.27.1.191:8081/A12.txt@-1!2@1@http://147.27.1.191:8081/B22.txt@-1 1@1@ArrC.txt@123
5 1 http://147.27.1.191:8081/Multiply.jar 1@1@http://147.27.1.191:8081/A21.txt@-1!2@1@http://147.27.1.191:8081/B11.txt@-1 1@1@ArrC.txt@124
6 1 http://147.27.1.191:8081/Multiply.jar 1@1@http://147.27.1.191:8081/A21.txt@-1!2@1@http://147.27.1.191:8081/B12.txt@-1 1@1@ArrC.txt@125
7 1 http://147.27.1.191:8081/Multiply.jar 1@1@http://147.27.1.191:8081/A22.txt@-1!2@1@http://147.27.1.191:8081/B21.txt@-1 1@1@ArrC.txt@126
8 1 http://147.27.1.191:8081/Multiply.jar 1@1@http://147.27.1.191:8081/A22.txt@-1!2@1@http://147.27.1.191:8081/B22.txt@-1 1@1@ArrC.txt@127
9 1 http://147.27.1.191:8081/Add.jar 1@2@1@1!2@2@3@1 1@1@ArrC.txt@128
10 1 http://147.27.1.191:8081/Add.jar 1@2@2@1!2@2@4@1 1@1@ArrC.txt@129
11 1 http://147.27.1.191:8081/Add.jar 1@2@5@1!2@2@7@1 1@1@ArrC.txt@130
12 1 http://147.27.1.191:8081/Add.jar 1@2@6@1!2@2@8@1 1@1@ArrC.txt@131
13 1 http://147.27.1.191:8081/MergeH.jar 1@2@9@1!2@2@10@1 1@1@ArrC.txt@132
14 1 http://147.27.1.191:8081/MergeH.jar 1@2@11@1!2@2@12@1 1@1@ArrC.txt@133
15 1 http://147.27.1.191:8081/MergeV.jar 1@2@13@1!2@2@14@1 1@1@ArrC.txt@134
```

## Α΄.2 Το αρχείο **job.xsd**

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<!--
       JOB description
     -->
<xsd:element name="Job">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="job-id" type="xsd:long" minOccurs="1" maxOccurs="1"/>
                        <xsd:element name="client-id" type="xsd:long" minOccurs="1" maxOccurs="1"/>
                        <xsd:element ref="code" minOccurs="1" maxOccurs="1"/>
                        <xsd:element ref="data-in" minOccurs="1" maxOccurs="1"/>
                        <xsd:element ref="data-out" minOccurs="1" maxOccurs="1"/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<!--
       CODE description
     -->
<xsd:element name="code">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="code-uri" type="xsd:anyURI" minOccurs="1" maxOccurs="1"/>
                        <xsd:element name="code-version" type="xsd:string" minOccurs="1" maxOccurs="1"/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<!--
       DATA IN description
     -->
<xsd:element name="data-in">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="arg-in" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>

<!--
       ARG-IN description
     -->
<xsd:element name="arg-in">
        <xsd:complexType>
                <xsd:sequence>

                        <xsd:element name="arg-in-id" type="xsd:long" minOccurs="1" maxOccurs="1"/>

                        <xsd:choice minOccurs="1" maxOccurs="1">
                                <xsd:element name="arg-in-inline" type="xsd:string"/>
                                <xsd:sequence>
                                        <xsd:element name="arg-in-credential" type="xsd:string" minOccurs="1" maxOccurs="1"/>
                                        <xsd:element name="arg-in-uri" type="xsd:anyURI" minOccurs="1" maxOccurs="1"/>
                                </xsd:sequence>
                        </xsd:choice>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<!--
       DATA OUT description
```

```
    -->
<xsd:element name="data-out">
       <xsd:complexType>
              <xsd:sequence>
                     <xsd:element ref="arg-out" minOccurs="1" maxOccurs="unbounded"/>
              </xsd:sequence>
       </xsd:complexType>
</xsd:element>
<!--
       ARG-OUT description
    -->
<xsd:element name="arg-out">
       <xsd:complexType>
              <xsd:sequence>
                     <xsd:element name="arg-out-id" type="xsd:long" minOccurs="1" maxOccurs="1"/>
                     <xsd:element name="arg-method" type="xsd:long" minOccurs="1" maxOccurs="1"/>
                            <xsd:sequence>
                                   <xsd:element name="arg-out-credential" type="xsd:string" minOccurs="1" maxOccurs="1"/>
                                   <xsd:element name="arg-out-filename" type="xsd:string" minOccurs="1" maxOccurs="1"/>
                                   <xsd:element name="arg-out-timeout" type="xsd:string" minOccurs="1" maxOccurs="1"/>
                            </xsd:sequence>
              </xsd:sequence>
       </xsd:complexType>
</xsd:element>

</xsd:schema>
```

# Βιβλιογραφία


[1] distributed.net project. http://www.distributed.net/, 2000.

[2] Abhishek Agrawal και Henri Casanova. Clustering hosts in P2P and global computing platforms. Στο *3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, σελίδες 367–373, 2003.

[3] Nidal A. Al-Dmour και William John Teahan. ParCop: A decentralized peer-to-peer computing system. Στο *3rd International Symposium on Parallel and Distributed Computing (ISPDC 2004), 3rd International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogenous Networks (HeteroPar 2004)*, σελίδες 162–168, 2004.

[4] Gene Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. Στο *AFIPS Conference Proceedings*, σελίδες 483–485, 1967.

[5] D. P. Anderson. Public computing: Reconnecting people to science. Στο *Conference on Shared Knowledge and the Web*, 2003.

[6] D. P. Anderson, J.Cobb, E. Korpela, M. Lebofsky και D. Werthimer. SETI@home massively distributed computing for SETI. *Computing in Science & Engineering*, 3:78 – 83, 2001.

[7] D. P. Anderson, J.Cobb, E. Korpela, M. Lebofsky και D. Werthimer. SETI@home: An Experiment in Public-Resource Computing. *Communications of the ACM*, 45:55–61, 2002.

[8] David P. Anderson. BOINC: A system for public-resource computing and storage. Στο *GRID*, επιμελητής: Rajkumar Buyya, σελίδες 4–10. IEEE Computer Society, 2004.

[9] George Bosilca, Aurélien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fédak, Cécile Germain, Thomas Hérault, Pierre Lemarinier, Oleg Lodygensky, Frédéric Magniette, Vincent Néri και Anton Selikhov. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. Στο *The IEEE/ACM SC2002 Conference*, 2002.

[10] Aurélien Bouteiller, Hinde Lilia Bouziane, Pierre Lemarinier, Thomas Herault και Franck Cappello. Hybrid preemptive scheduling for MPI applications on the Grids. Στο *5th IEEE/ACM International Workshop on Grid Computing*, 2004.

[11] Aurélien Bouteiller, Franck Cappello, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier και Frédéric Magniette. MPICH-V2: a fault tolerant MPI for volatile nodes based on the pessimistic sender based message logging. Στο *The IEEE/ACM SC2003 Conference*, 2003.

[12] Aurélien Bouteiller, Boris Collin, Thomas Herault, Pierre Lemarinier και Franck Cappello. Impact of event logger on causal message logging protocols for fault tolerant MPI. Στο *19th IEEE/ACM International Parallel and Distributed Processing Symposium*, 2005.

[13] Aurélien Bouteiller, Pierre Lemarinier, Thomas Hérault, Géraud Krawezik και Franck Cappello. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. Στο *The 2004 IEEE International Conference on Cluster Computing*, 2004.

[14] Aurélien Bouteiller, Pierre Lemarinier, Géraud Krawezik και Franck Cappello. Coordinated checkpoint versus message log for fault tolerant MPI. Στο *The 2003 IEEE International Conference on Cluster Computing*, 2003.

[15] T. Brecht, H. Sandhu, M. Shan και J. Talbot. ParaWeb: Towards world-wide supercomputing, 1996.

[16] R. Buyya, D. Abramson και J. Giddy. Economy driven resource management architecture for computational power grids. Στο *PDPTA2000, International Conference on Parallel and Distributed Processing Techniques and Applications*, 2000.

[17] Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Herault, Frédéric Magniette, Vincent Néri και Oleg Lodygensky. Computing on large scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with Grid. *Future Generation Computer Systems*, 21(3):417–437, 2004.

[18] Olivier Delannoy και Serge G. Petiton. A peer to peer computing framework: Design and performance evaluation of YML. Στο *3rd International Symposium on Parallel and Distributed Computing (ISPDC 2004), 3rd International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogenous Networks (HeteroPar 2004)*, σελίδες 362–369, 2004.

[19] Samir Djilali. P2P-RPC: Programming scientific applications on peer-to-peer systems with remote procedure call. Στο *3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, σελίδες 406–413, 2003.

[20] G. Fedak, C. Germain, V. Néri και F. Cappello. XtremWeb: A generic global computing system, 2001.

[21] Chris Kenyon και Giorgos Cheliotis. Creating services with hard guarantees from cycle-harvesting systems. Στο *3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, σελίδες 224–231, 2003.

[22] K. Kreyman, D. L. Parnas και S. Qiao. Inspection procedures for critical programs that model physical phenomena. Τεχνική Αναφορά υπ. αρίθμ. 368, McMaster University, Software Engineering Research Group, 1999.

[23] S. M. Larson, C. D. Snow, M. Shirts και V. S. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. Στο *Computational Genomics*, επιμελητής: R. Grant. Horizon Press, 2002.

[24] Oleg Lodygensky, Gilles Fedak, Vincent Néri, Alain Cordier και Franck Cappello. Auger & XtremWeb: Monte carlo computation on a global computing platform. Στο *Computing in High Energy and Nuclear Physics*, 2003.

[25] A. L. Beberg. The Cosm Project. Design and goals available at `http://www.mithral.com/projects/cosm/`, χ.χ.

[26] Parabon Computation. The Frontier Application Programming Interface, Version 1.5.2. Platform API white paper, χ.χ.

[27] Parabon Computation. Frontier: The Premier Internet Computing Platform. Platform design white paper, 1999.

[28] D. Molnar. The SETI@home Problem. *ACM Crossroads*, 2000.

[29] Michael O. Neary, Sean P. Brydon, Paul Kmiec, Sami Rollins και Peter Cappello. Javelin++: scalability issues in global computing. *Concurrency: Practice and Experience*, 12(8):727–753, 2000.

[30] Noam Nisan. Algorithms for selfish agents. Στο *STACS 99, 16th Annual Symposium on Theoretical Aspects of Computer Science*, σελίδες 1–15, 1999.

[31] V. S. Pande, I. Baker, J. Chapman, S. P. Elmer, S. Khaliq, S. M. Larson, Y. M. Rhee, M. R. Shirts, C. D. Snow, E. J. Sorin και B. Zagrovic. Atomistic protein folding simulations on the submillisecond time scale using worldwide distributed computing. *Biopolymers*, 68:91–109, 2003.

[32] Jakob Gregor Pedersen και Christian Ulrik Søttrup. Developing distributed computing solutions combining Grid computing and public computing. Διπλωματική εργασία Master, University of Copenhagen, Μάρτιος 2005. http://www.fatbat.dk/thesis/boincThesis.pdf.

[33] Hernâni Pedroso, Luís Moura Silva και João Gabriel Silva. JET: Massively parallel computing with Java. Στο *Third International Conference on Massively Parallel Computing Systems*, 1998.

[34] Ori Regev και Noam Nisan. The POPCORN market - an online market for computational resources. Στο *ICE '98: Proceedings of the first international conference on Information and computation economies*, σελίδες 148–157, New York, NY, USA, 1998. ACM Press.

[35] Luis F. G. Sarmenta. Studying sabotage-tolerance mechanisms through web-based parallel parametric analysis and monte carlo simulation, χ.χ.

[36] Luis F. G. Sarmenta. *Volunteer Computing*. Διδακτορική Διατριβή , Massachusetts Institute of Technology, Ιούνιος 2001. http://www.cag.lcs.mit.edu/bayanihan/papers/phd/sarmenta-phd-mit2001.pdf.

[37] Luis F. G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, 18(4):561–572, 2002.

[38] B. Uk, Michela Taufer, Thomas Stricker, Gianni Settanni, Andrea Cavalli και Amedeo Caflisch. Combining task- and data parallelism to speed up protein folding on a desktop Grid platform. Στο *3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, σελίδες 240–, 2003.

[39] H. Wasserman και M. Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, 1997.

[40] Dayi Zhou και Virginia Lo. WaveGrid: a scalable fast-turnaround heterogeneous peer-based desktop Grid system. Στο *20th IEEE/ACM International Parallel and Distributed Processing Symposium*, 2006.