ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΜΕ ΕΦΑΡΜΟΓΕΣ ΣΤΗ ΒΙΟΙΑΤΡΙΚΗ

ΑΝΑΠΤΥΞΗ ΑΛΓΟΡΙΘΜΩΝ ΓΙΑ ΤΟΠΟΘΕΤΗΣΗ ΚΑΙ ΜΕΤΑΚΙΝΗΣΗ ΠΡΑΚΤΟΡΩΝ ΣΕ ΚΑΤΑΝΕΜΗΜΕΝΑ ΣΥΣΤΗΜΑΤΑ

Σαρούκος Ελευθέριος

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Υπεύθυνος

Λουκόπουλος Αθανάσιος

Λέκτορας

Λαμία, Φεβρουάριος 2016

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΜΕ ΕΦΑΡΜΟΓΕΣ ΣΤΗ ΒΙΟΙΑΤΡΙΚΗ

ΑΝΑΠΤΥΞΗ ΑΛΓΟΡΙΘΜΩΝ ΓΙΑ ΤΟΠΟΘΕΤΗΣΗ ΚΑΙ ΜΕΤΑΚΙΝΗΣΗ ΠΡΑΚΤΟΡΩΝ ΣΕ ΚΑΤΑΝΕΜΗΜΕΝΑ ΣΥΣΤΗΜΑΤΑ

Σαρούκος Ελευθέριος

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Επιβλέπων/σα

Λουκόπουλος Αθανάσιος

Λέκτορας

Λαμία, Φεβρουάριος 2016

Με ατομική μου ευθύνη και γνωρίζοντας τις κυρώσεις [1], που προβλέπονται από της διατάξεις της παρ. 6 του άρθρου 22 του Ν. 1599/1986, δηλώνω ότι:

1. *Δεν παραθέτω κομμάτια βιβλίων ή άρθρων ή εργασιών άλλων αυτολεξεί* ***χωρίς να τα περικλείω σε εισαγωγικά*** *και χωρίς να αναφέρω το συγγραφέα, τη χρονολογία, τη σελίδα. Η αυτολεξεί παράθεση χωρίς εισαγωγικά χωρίς αναφορά στην πηγή, είναι λογοκλοπή. Πέραν της αυτολεξεί παράθεσης, λογοκλοπή θεωρείται και η παράφραση εδαφίων από έργα άλλων, συμπεριλαμβανομένων και έργων συμφοιτητών μου, καθώς και η παράθεση στοιχείων που άλλοι συνέλεξαν ή επεξεργάσθηκαν, χωρίς αναφορά στην πηγή. Αναφέρω πάντοτε με πληρότητα την πηγή κάτω από τον πίνακα ή σχέδιο, όπως στα παραθέματα.*

2. *Δέχομαι ότι η αυτολεξεί* ***παράθεση χωρίς εισαγωγικά****, ακόμα κι αν συνοδεύεται από αναφορά στην πηγή σε κάποιο άλλο σημείο του κειμένου ή στο τέλος του, είναι αντιγραφή. Η αναφορά στην πηγή στο τέλος π.χ. μιας παραγράφου ή μιας σελίδας, δεν δικαιολογεί συρραφή εδαφίων έργου άλλου συγγραφέα, έστω και παραφρασμένων, και παρουσίασή τους ως δική μου εργασία.*

3. *Δέχομαι ότι υπάρχει επίσης περιορισμός στο μέγεθος και στη συχνότητα των παραθεμάτων που μπορώ να εντάξω στην εργασία μου εντός εισαγωγικών. Κάθε μεγάλο παράθεμα (π.χ. σε πίνακα ή πλαίσιο, κλπ), προϋποθέτει ειδικές ρυθμίσεις, και όταν δημοσιεύεται προϋποθέτει την άδεια του συγγραφέα ή του εκδότη. Το ίδιο και οι πίνακες και τα σχέδια*

4. *Δέχομαι όλες τις συνέπειες σε περίπτωση λογοκλοπής ή αντιγραφής.*

Ημερομηνία: ……/…../20……

Ο – Η Δηλ.

(Υπογραφή)

(1) «Όποιος εν γνώσει του δηλώνει ψευδή γεγονότα ή αρνείται ή αποκρύπτει τα αληθινά με έγγραφη υπεύθυνη δήλωση του άρθρου 8 παρ. 4 Ν. 1599/1986 τιμωρείται με φυλάκιση τουλάχιστον τριών μηνών. Εάν ο υπαίτιος αυτών των πράξεων σκόπευε να προσπορίσει στον εαυτόν του ή σε άλλον περιουσιακό όφελος βλάπτοντας τρίτον ή σκόπευε να βλάψει άλλον, τιμωρείται με κάθειρξη μέχρι 10 ετών.

ΑΝΑΠΤΥΞΗ ΑΛΓΟΡΙΘΜΩΝ ΓΙΑ ΤΟΠΟΘΕΤΗΣΗ ΚΑΙ ΜΕΤΑΚΙΝΗΣΗ ΠΡΑΚΤΟΡΩΝ ΣΕ ΚΑΤΑΝΕΜΗΜΕΝΑ ΣΥΣΤΗΜΑΤΑ

Σαρούκος Ελευθέριος

Τριμελής Επιτροπή:

Λουκόπουλος Αθανάσιος, Λέκτορας (επιβλέπων)

Σανδαλίδης Χαρίλαος, Επίκουρος

Αναγνωστόπουλος Ιωάννης, Επίκουρος

Ευχαριστώ την οικογένεια και τους φίλους μου για την υπομονή τους και τον κ. Θανάση Λουκόπουλο για την άριστη συνεργασία μας.

Οι φιλόσοφοι έχουν ερμηνεύσει τον κόσμο με διάφορους τρόπους.
Το θέμα, όμως, είναι να τον αλλάξουμε.
K. Marx

ΠΕΡΙΕΧΟΜΕΝΑ

ΠΕΡΙΛΗΨΗ

Αυτή η πτυχιακή εργασία σκοπό έχει την παρουσίαση και πρακτική μιας πρότασης, που για πρώτη φορά προτάθηκε από τον κ. Θανάση Λουκόπουλο, για τη βελτίωση του αλγορίθμου εκτόπισης πρακτόρων (agent eviction algorithm[1]). Για τη δοκιμή και σύγκριση της πρότασης με τα αποτελέσματα του υπάρχοντος αλγορίθμου, υλοποιείται πρόγραμμα σε C++ που προσομοιώνει την κατανομή και μετακίνηση των πρακτόρων στο δίκτυο.

Στο 1ο κεφάλαιο γίνεται μια εισαγωγή στα Ασύρματα Δίκτυα Αισθητήρων (ΑΣΑ), δίνεται ο ορισμός τους προβλήματος της βέλτιστης κατανομής πρακτόρων και αναφέρονται έρευνες που ήδη έχουν προηγηθεί για τον ίδιο ή παρεμφερή σκοπό

Στο 2ο κεφάλαιο παρουσιάζεται ο αλγόριθμος Single Agent Migration (SAM) και οι περιορισμοί που εμφανίζονται στην εφαρμογή του σε πραγματικά ΑΣΑ

Στο 3ο κεφάλαιο αναλύεται η μέθοδος των εκτοπίσεων agent

Στο 4ο κεφάλαιο γίνεται παρουσίαση της καινούριας μεθόδου, για Μετακινήσεις με Αρνητικό Κόστος (Negative Evictions)

Στο 5ο κεφάλαιο δίνεται μια εξηγείται η λογική πίσω από τον κώδικα που υλοποιήθηκε, συνοψίζονται τα αποτελέσματα της προσομοίωσης και εξάγονται συμπεράσματα σε σύγκριση με άλλους αλγορίθμους

Στο τέλος της εργασίας παρατίθεται παράρτημα με τον κώδικα της εφαρμογής

1 Nikos Tziritas, Petros Lampsas, Spyros Lalis, Thanasis Loukopoulos, Samee Ullah Khan, Cheng-Zhong Xu, "Introducing Agent Evictions to Improve Application Placement in Wireless Distributed Systems"

ΚΕΦΑΛΑΙΟ 1

ΕΙΣΑΓΩΓΗ

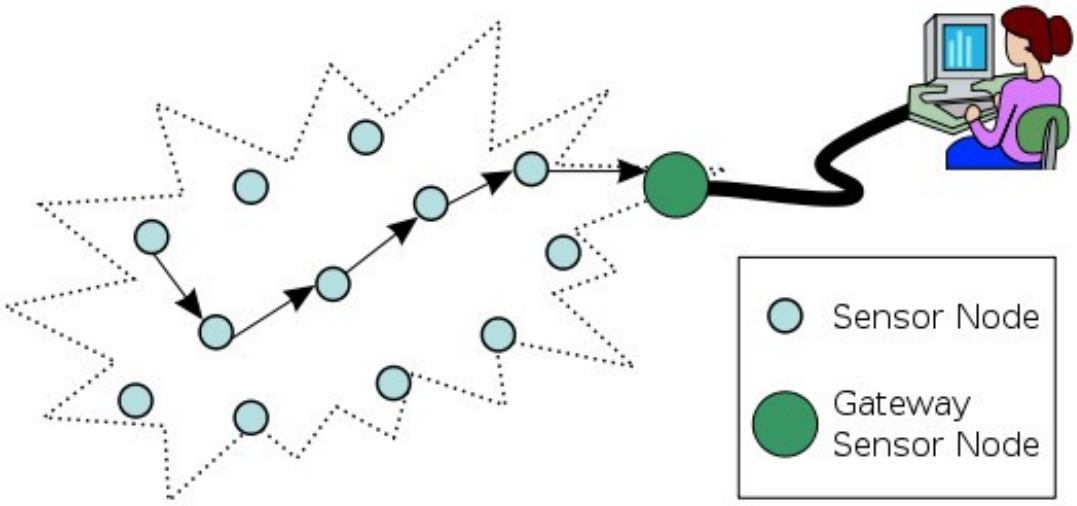
Στο κεφάλαιο αυτό δίνεται ο ορισμός τους προβλήματος που πραγματευόμαστε καθώς και το σύστημα στο οποίο αναφέρεται. Στην τελευταία ενότητα, παρουσιάζονται παρόμοιες μελέτες που έγιναν στο παρελθόν και οι διαφορές τους με την παρούσα.

Κεφάλαιο 1.1. Ασύρματα Δίκτυα Αισθητήρων (ΑΔΑ) και Mobile Agents

Ασύρματα Δίκτυα Αισθητήρων (ΑΔΑ) είναι δίκτυα από αυτόνομους αισθητήρες που συνεργάζονται και φέρουν ειδικά όργανα ικανά να συλλέγουν, επεξεργάζονται και να μεταφέρουν δεδομένα από κάθε γωνιά του δικτύου μέσω συνεργασίας σε μια συγκεκριμένη τοποθεσία. Οι αισθητήρες μαζί με τον εξοπλισμό τους (μικροεπεξεργαστής, μπαταρία, ραδιοπομποδέκτης) συνθέτουν τους *κόμβους* του δικτύου, οργανωμένους με κατανεμημένη αρχιτεκτονική, αφού είναι συχνά επιρρεπείς σε αποτυχίες και μια κεντρική δομή θα κατέρρεε συχνά. Επίσης, ανάλογα με το σκοπό και την τοποθεσία τους, το μέγεθος και το κόστος τους ποικίλει, θέτοντας περιορισμούς σε μνήμη, υπολογιστική ισχύ, ενέργεια και στο εύρος ζώνης των ραδιοεπικοινωνιών.

Τα ΑΔΑ χρησιμοποιούνται για την απομακρυσμένη παρατήρηση φυσικών ή περιβαλλοντικών συνθηκών όπως θερμοκρασία, υγρασία, ένταση ανέμων, ελεγκτές ταχύτητας κτλ. Πολλές φορές μάλιστα, η πρόσβαση σε αυτά δεν είναι εύκολη. Έτσι, η εξασφάλιση μακροχρόνιας βιωσιμότητας των κόμβων είναι καθοριστική για το κόστος και τη συνολική τους λειτουργία. Ο χρόνος ζωής ενός κόμβου βρίσκεται σε άμεση εξάρτηση με την κατανάλωση ενέργειας του (ειδικά σε

συστήματα που η μόνη πηγή ενέργειας τους είναι μπαταρίες) και αυτή με τη σειρά της εξαρτάται σε μεγάλο βαθμό από τον *φόρτο επικοινωνίας των κόμβων*, δηλαδή το πόσο συχνά ανταλλάσσουν μηνύματα μεταξύ τους.



Εικόνα 1.α. Τυπική αρχιτεκτονική ΑΔΑ

Μια από τις τεχνικές που προσαρμόστηκε στη λειτουργία των ΑΔΑ είναι αυτή των κινητών πρακτόρων (mobile agent programming). Γενικά, οι πράκτορες (software agents) είναι συνδυασμός λογισμικών προγραμμάτων και δεδομένων, που λειτουργούν και αλληλεπιδρούν με το περιβάλλον τους χωρίς την συνεχόμενη ανθρώπινη παρέμβαση ή επίβλεψη. Συγκεκριμένες προγραμματιστικές τεχνικές έχουν αναπτυχθεί που επιτρέπουν την δημιουργία agents ικανών να μετακινούνται από μηχάνημα σε μηχάνημα (στην δική μας περίπτωση από κόμβο σε κόμβο) και να παραμένουν πλήρως λειτουργικοί. Την απόφαση για το που και πότε χρειάζεται να μετακινηθούν την παίρνουν μόνοι τους, εκτιμώντας τι θα ήταν πιο αποδοτικό για το σκοπό που έχουν προγραμματιστεί. Όταν ένας agent αποφασίζει να μετακομίσει, αποθηκεύει την παρούσα κατάσταση του, τη μεταφέρει μαζί με όλα τα δεδομένα που χειρίζεται στον νέο κόμβο και συνεχίζει την εκτέλεση του από εκεί που σταμάτησε.

Η εφαρμογή τους στα ΑΔΑ γέννησε καινούριες δυνατότητες. Διάφορες εφαρμογές τοποθετούνται στο δίκτυο χωρισμένες σε αυτοτελή agents που συνεργάζονται και λειτουργούν σαν μια ολότητα. Οι agents ανταλλάσσουν μεταξύ τους δεδομένα αξιοποιώντας τους μηχανισμούς που τους παρέχονται από τον κόμβο που τους φιλοξενεί γι αυτό και η κατανομή τους στο δίκτυο επηρεάζει το φόρτο επικοινωνίας. Μια σωστότερη τοποθέτηση θα μείωνε τον αριθμό των μηνυμάτων που στέλνουν οι agents κατά την επικοινωνία τους άρα και την κατανάλωση ενέργειας. Έτσι, αν η αρχική τοποθέτηση των agent δεν είναι η βέλτιστη, ή αν έχουν προστεθεί νέοι, η μέθοδος που εξετάζουμε σε αυτή τη διατριβή βοηθά στον προσδιορισμό μιας αποδοτικότερης κατανομής τους μέσα στο δίκτυο.

Κεφάλαιο 1.2: Το Μοντέλο των Εφαρμογών και του Συστήματος και Ορισμός του Προβλήματος

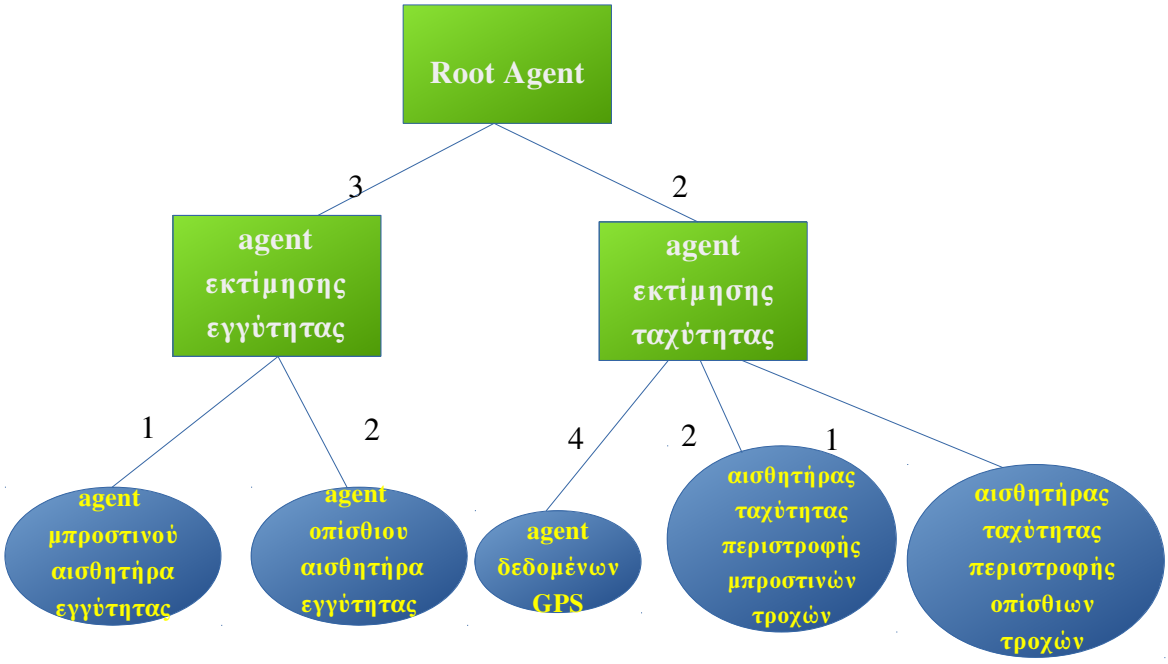
Σαν σύνολο πρακτόρων μέσα σε ΑΔΑ, μια εφαρμογή αναλαμβάνει από τη συλλογή των μετρήσεων των αισθητήρων μέχρι την μεταφορά τους στον τελικό τους προορισμό. Οι agents σκορπίζονται μέσα στο δίκτυο και διακρίνονται σε δύο κατηγορίες ανάλογα με τη λειτουργία που επιτελούν. Η πρώτοι είναι εκείνοι που χειρίζονται τους αισθητήρες. Αυτοί βέβαια, είναι εξαρτημένοι και αμετακίνητοι από τους κόμβους στους οποίους βρίσκεται ο αντίστοιχος αισθητήρας και για αυτό τους ονομάζουμε *συγκεκριμένων κόμβων (node-specific)*. Μετατρέπουν τις μετρήσεις των αισθητήρων σε ψηφιακά δεδομένα. Οι δεύτεροι αξιοποιούνται για την επεξεργασία των δεδομένων που συλλέγονται από τους αισθητήρες και για την λήψη σχετικών αποφάσεων. Είναι υπεύθυνοι για την επικοινωνία με τους άλλους agents και η λειτουργία τους είναι ανεξάρτητη από τη θέση τους στο δίκτυο, γι αυτό και τους καλούμε *ανεξαρτήτων κόμβων (node-neutral)*.

Το σύνολο των agents μιας εφαρμογής είναι δομημένο σαν ένα δέντρο, που ονομάζουμε *δέντρο εφαρμογής* και χαρτογραφεί τη μεταξύ συσχέτιση τους. Μόνο οι συγγενείς (πατέρας, παιδιά)

ανταλλάσσουν μηνύματα μεταξύ τους. Στα φύλλα του βρίσκονται οι *συγκεκριμένων κόμβων agents* ενώ στα ψηλότερα επίπεδα οι άλλοι. Κάθε agent έχει ένα *φόρτο επικοινωνίας*, που καθορίζεται σαν το πλήθος των μονάδων δεδομένων που στέλνει στους συγγενείς του ανά μονάδα χρόνου.

Στην *εικόνα 1.β* φαίνεται ένα τέτοιο παράδειγμα δέντρου agent. Απεικονίζει ένα υποθετικό σύστημα αυτόματου φρεναρίσματος του αυτοκινήτου για την αποφυγή πιθανής πρόσκρουσης. Τα φύλλα του αναπαριστούν συγκεκριμένων κόμβων agents, που χειρίζονται τους αισθητήρες ταχύτητας ή απόστασης. Οι υπόλοιποι agent είναι ανεξαρτήτων κόμβων και μπορούν να μετακινηθούν. Οι αριθμοί μεταξύ των agent είναι τα δεδομένα που ανταλλάσσουν μεταξύ τους ανά μονάδα χρόνου.

Οι αισθητήρες εγγύτητας ενημερώνουν τον εκτιμητή για την απόσταση από πιθανά εμπόδια μπροστά ή πίσω από το αυτοκίνητο. Αν η απόσταση του αυτοκινήτου από ένα αντικείμενο μετρηθεί κάτω από ένα όριο, ο εκτιμητής εγγύτητας ενημερώνει τον root Agent. Αντίστοιχα, οι τιμές που φτάνουν στον εκτιμητή ταχύτητας από το GPS και τους αισθητήρες περιστροφής των τροχών, του επιτρέπουν να εντοπίσει αν το αυτοκίνητο είναι ήδη σταθμευμένο ή προς ποια κατεύθυνση και με τι ταχύτητα κινείται. Έπειτα, ενημερώνει τον root Agent που αποφασίζει με βάση όλα τα δεδομένα που έφτασαν, αν το αυτοκίνητο έχει πορεία σύγκρουσης με άλλο αντικείμενο, οπότε και ενεργοποιεί τα φρένα ή αν κάτι άλλο κατευθύνεται προς το αυτοκίνητο οπότε και αναπαράγει προειδοποιητικό ηχητικό σήμα.



Εικόνα 1.β. Δέντρο εφαρμογής αυτόματου φρεναρίσματος αυτοκινήτου

Προσαρμοσμένοι στους κόμβους ενός ΑΔΑ, οι agents αξιοποιούν τους μηχανισμούς που παρέχονται από τον κόμβο που φιλοξενούνται για την επικοινωνία τους με απομακρυσμένους agent. Οι κόμβοι συνδέονται με τους υπόλοιπους με μοναδικά μονοπάτια. Έτσι, η ανταλλαγή μηνυμάτων μεταξύ συγγενών agent περνά αναγκαστικά από όλους τους κόμβους στο μονοπάτι που συνδέει εκείνους που τους φιλοξενούν. Αυτό (με εξαίρεση την περίπτωση που οι συγγενείς agent φιλοξενούνται σε γείτονες κόμβους) οδηγεί σε σπατάλη ενέργειας, καθώς αναγκάζει ενδιάμεσους, άσχετους με τα μηνύματα που ανταλλάσσονται, κόμβους να λειτουργούν για να μεταβιβάζουν μηνύματα. Δηλαδή, όσο πιο κοντά βρίσκονται οι συγγενείς, τόσο μειώνεται και ο συνολικός φόρτος. Αυτός είναι και ο σκοπός της εργασίας μας!

Υποθέτοντας πως η αρχική τοποθέτηση τους στους κόμβους του συστήματος δεν είναι η βέλτιστη, εκτελώντας μια σειρά από μετατοπίσεις αναζητούμε μια καλύτερη κατανομή τους στο δίκτυο. Με αυτόν τον τρόπο μειώνεται το ενεργειακό κόστος που προκαλεί η επικοινωνία των agent.

Κεφάλαιο 1.3: Παρόμοιες Μελέτες

Στο παρελθόν, έχουν προταθεί και άλλοι μέθοδοι επανατοποθέτησης agent. Όμως, αντίθετα με την παρούσα εργασία που στοχεύει στην μείωση του φόρτου που προκαλεί η μεταξύ των agent επικοινωνία, οι σκοποί ή και η δομή των άλλων ερευνών διαφέρουν. Παράδειγμα το [1] ασχολείται με την καλύτερη τοποθέτηση διεργασιών σε δίκτυο torus για την αποφυγή αποσυμφόρησης. Στο [2] οι συγγραφείς καταπιάνονται με τα προβλήματα αποδοχής των agent για την εξασφάλιση της μακροζωίας ενός ΑΔΑ. Κατανομή διεργασιών αναφέρεται επίσης στα [3], [4], [5] αλλά σε NoC συστήματα, ενώ στο [4] αναφέρεται η καλύτερη τοποθέτηση υπηρεσιών για την μείωση της καθυστέρησης που αντιμετωπίζουν οι clients σε virtual networks.

Σχετικά με τα ΑΔΑ, στο [6] προτείνεται κατανομή των εφαρμογών για μικρότερη κατανάλωση ενέργειας, αλλά το μοντέλο εφαρμογής που χρησιμοποιείται είναι διαφορετικό. Στο [14] προτείνεται η δημιουργία ενός βέλτιστου δέντρου διάδοσης δεδομένων ενώ στο [15] συζητιέται η μετακόμιση agents αλλά με σκοπό την εύρεση βέλτιστων μονοπατιών συλλογής δεδομένων.

Τέλος, να σημειωθεί, ότι υπάρχουν πολλά συστήματα που υποστηρίζουν την λειτουργία και την μετακίνηση αυτόνομων εφαρμογών (βλέπε Agilla[7], SensorWar[8], Mobile-C[9], MagnetOS[10], Pleiades[11], DFuse[12] και Rovers[13]). Παρόλα αυτά, τα περισσότερα αφήνουν το πολύπλοκο έργο της τοποθέτησης των agent στον προγραμματιστή. Φυσικά, έτσι μειονεκτούν σε ευελιξία, καθώς σε περίπτωση αναδιάταξης ή προσθήκης περισσότερων κόμβων στο δίκτυο, το σύστημα δεν χρειάζεται να προγραμματιστεί ξανά. Η παρούσα μελέτη, αφορά περισσότερο συστήματα όπως το POBICOS[14], που αναθέτουν την κατανομή των agents στο middleware.

ΚΕΦΑΛΑΙΟ 2

Ο ΑΛΓΟΡΙΘΜΟΣ SINGLE AGENT MIGRATION

Ο αλγόριθμος που χρησιμοποιείται σαν βάση τον προτάσεων αυτής της εργασίας είναι ο **Single Agent Migration** και η ένταξη σε αυτόν **μεθόδους εκτοπίσεων**. Αυτό το κεφάλαιο ασχολείται με την ανάλυση αυτών καθώς και **μεθόδων εύρεσης των καλύτερων εκτοπίσεων**.

Κεφάλαιο 2.1: Ο Αλγόριθμος Single Agent Migration (SAM)

Σε αυτόν τον αλγόριθμο agents μεταφέρονται διαρκώς στον πιο επωφελή για μετακόμιση γείτονα κόμβο τους με σκοπό πάντα τη μείωση του συνολικού φόρτου του συστήματος. Οι μετακινήσεις των agent γίνονται ένα βήμα τη φορά (1-hop migrations), δηλαδή από γείτονα σε γείτονα κόμβο. Πιο συγκεκριμένα, κόμβοι του δικτύου, που πριν βρίσκονταν σε αδράνεια, ενεργοποιούνται τυχαία και επιλέγουν ποιον από τους agents που φιλοξενούν θα προσπαθήσουν να μεταφέρουν και σε ποιον γείτονα. Για να είναι σε θέση να επιλέξουν, οι κόμβοι γνωρίζουν 1) τον *τοπικό φόρτο* και 2) τον *απομακρυσμένο φόρτο*. Ο πρώτος είναι ο φόρτος επικοινωνίας μεταξύ του κάθε φιλοξενούμενου agent και των συγγενών του (πατέρας ή παιδιά στο δέντρο εφαρμογής) στην παρούσα τοποθεσία τους, ενώ ο *απομακρυσμένος* αναφέρεται στον φόρτο που θα χε αν ο agent μετακόμιζε σε έναν γείτονα κόμβο.

Γενικά, ο φόρτος επικοινωνίας δίνεται από τον τύπο:

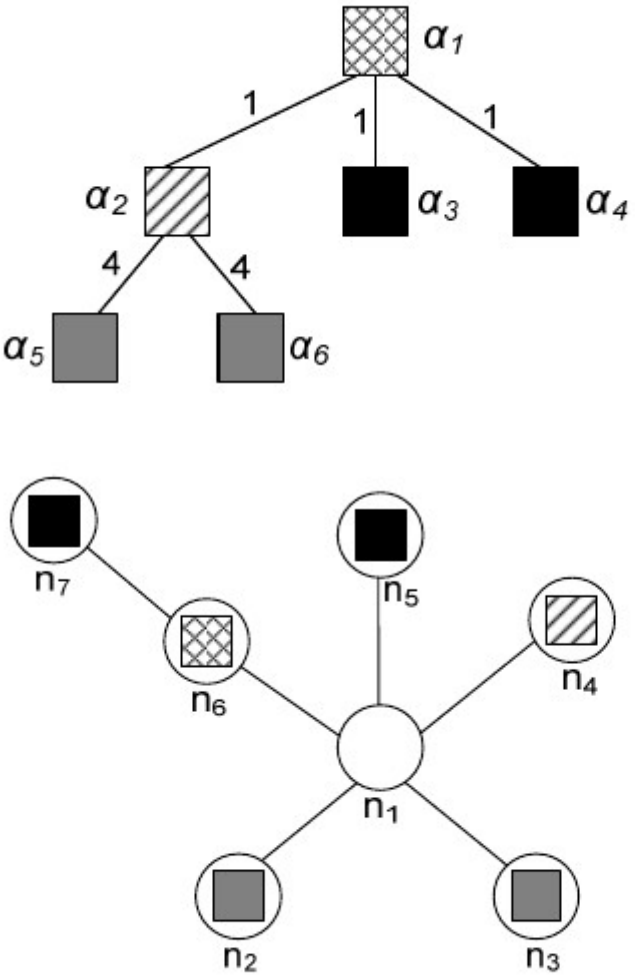
$$localLoad(a_k) = \sum_{r \in Rel(a)} (totalData(a,r) * distance(a,r)) \qquad (1)$$

όπου a_k: agent φιλοξενούμενος στον κόμβο k,

Rel(a): σύνολο που περιέχει όλους τους συγγενείς του a στο δέντρο εφαρμογής,
r: μεταβλητή που παίρνει τιμές από το Rel(a),
totalData(a,r): το άθροισμα των δεδομένων που ανταλλάσσονται ανά μονάδα χρόνου μεταξύ των a και r
και distance(a,r): το πλήθος των βημάτων ώστε να φτάσει ο a από τον κόμβο που τον φιλοξενεί (k) μέχρι τον κόμβο που φιλοξενεί τον r, ακολουθώντας το μοναδικό μονοπάτι που τους ενώνει. Για agents ου φιλοξενούνται στον ίδιο κόμβο distance=0.

Αφαιρώντας τον απομακρυσμένο από τον τοπικό φόρτο[2] προκύπτει το *κέρδος μετακόμισης του κάθε agent*. Ανάλογα αν αυτό είναι θετικό ή όχι οι κόμβοι γνωρίζουν αν αντίστοιχα είναι επωφελής ή μη η μετακόμιση ενός agent και ταυτόχρονα, συγκρίνοντας τα διάφορα κέρδη, ξέρουν ποιος από τους φιλοξενούμενους τους agent έχει προτεραιότητα να μετακινηθεί. Για αυτόν, στέλνουν *αίτηση φιλοξενίας* (host request) στον πιο επωφελή γείτονα κόμβο τους και εκείνος ελέγχοντας την ελεύθερη μνήμη που διαθέτει απαντά αν τελικά μπορεί να αποδεχτεί τον agent (οπότε και αρχίζει η διαδικασία μετακόμισης του) ή όχι. Στα ακόλουθα παραδείγματα θεωρούμε ότι η μνήμη των κόμβων είναι άπειρη και δεν εμποδίζεται καμιά μετακίνηση λόγω τέτοιων περιορισμών.

2 Κέρδος = τοπικός φόρτος – απομακρυσμένος φόρτος (2)



Εικόνα 2.α. πάνω: δέντρο εφαρμογής. Οι αριθμοί είναι το πλήθος των δεδομένων που ανταλλάσσουν οι agents μεταξύ τους. Κάτω: Αρχική τοποθέτηση του δέντρου εφαρμογής σε δίκτυο κόμβων.

Η λειτουργία του αλγόριθμου φαίνεται καθαρά στις εικόνες 2.α και 2.β. Στο παράδειγμα της παραπάνω εικόνας μόνο οι agents α_1 και α_2 είναι ανεξαρτήτων κόμβων και μπορούν να μετακινηθούν. Έστω ότι ενεργοποιείται τυχαία ο κόμβος n_4. Αυτός φιλοξενεί μόνο τον agent a_2, άρα ο φόρτος επικοινωνίας καθορίζεται από το πλήθος των δεδομένων που ανταλλάσσει αυτός με τους συγγενείς του (α_1, α_5, α_6) και δίνεται από τον τύπο (1). Για τον τοπικό φόρτο επικοινωνίας (όπου η θέση του α_2 είναι ο κόμβος 4) ισχύει:

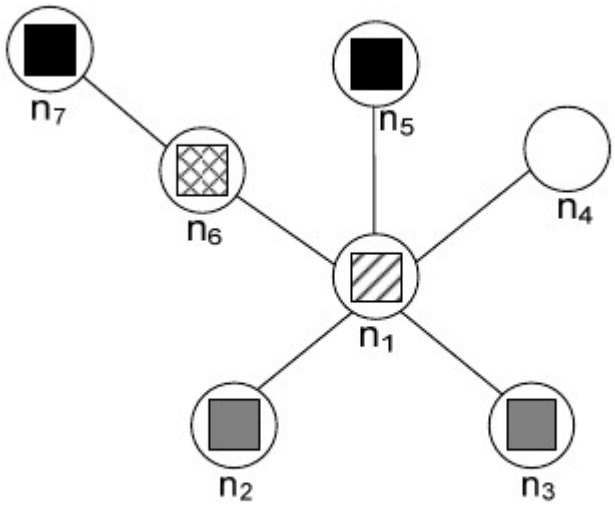
τοπικός φόρτος=totalData(α_2,α_1)*distance(α_2,α_1)+totalData(α_2,α_5)*distance(α_2,α_5)+totalData(α_2,α_6) *distance(α_2,α_6)=1*2+4*2+4*2=18. Για τον απομακρυσμένο, θεωρούμε ότι ο α_2 βρίσκεται στην

ζητούμενη θέση (κόμβος n_1) και ισχύει:

απομακρυσμένος φόρτος=totalData(α_2,α_1)*distance(α_2,α_1)+totalData(α_2,α_5)*distance(α_2,α_5)+totalData(α_2,α_6)*distance(α_2,α_6)=1*1+4*1+4*1=9.

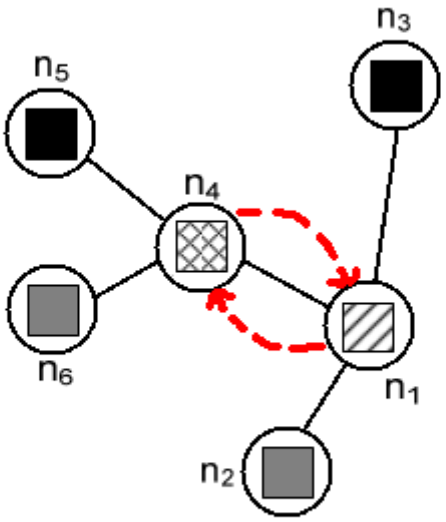
Άρα το κέρδος=τοπικός φόρτος – απομακρυσμένος φόρτος = 18-9 = 9, δηλαδή >0.

Το θετικό κέρδος, συνεπάγεται μείωση του φόρτου επικοινωνίας του συστήματος και δείχνει στον κόμβο ότι μπορεί να προχωρήσει στη μετακόμιση του agent του, καταλήγοντας στην μορφή που φαίνεται στην εικόνα 2.β.



Εικόνα 2.β. τελική τοποθέτηση των agents της εικόνας 2.α μετά από εφαρμογή του Single Agent Migration αλγόριθμου.

Στον SAM οι μετακινήσεις agent γίνονται ταυτόχρονα, δηλαδή την ίδια χρονική στιγμή που ο α_2 μετακομίζει στον κόμβο n_1 θα μπορούσε και ο α_1 να μεταφερθεί στον n_7. Αυτό όμως θα μπορούσε να δημιουργήσει προβλήματα και αδιέξοδα. Ας κοιτάξουμε την 2.γ.



Εικόνα 2.γ. συνεχόμενες ανταλλαγές agent από τους κόμβους n_6 και n_7. Οι n_1 και n_4 έχουν αρκετή μνήμη ώστε να εκτελούνται οι μετακομίσεις ανεμπόδιστα.

Για τη συγκεκριμένη τοποθέτηση των agent στο δίκτυο, αν ενεργοποιηθούν ταυτόχρονα οι κόμβοι 1 και 4 εξετάζουν την πιθανότητα μετακόμισης των agent τους. Για τον α_1 πιο επωφελής προορισμός είναι ο κόμβος n_1, με κέρδος=τοπικός-απομακρυσμένος φόρτος=4 – 3=1. Για τον α_2 κέρδος=13 – 12=1. Γνωρίζοντας αυτά, οι κόμβοι αποφασίζουν την ίδια χρονική στιγμή να μετακομίσουν τους α_1, α_2 και αφού δεν υπάρχουν περιορισμοί μνήμης ολοκληρώνουν τις μεταφορές. Στη νέα διάταξη οι α_1 και α_2 έχουν ανταλλάξει θέσεις. Ο συνολικός φόρτος επικοινωνίας του συστήματος όμως δεν έχει μειωθεί καθόλου, αφού πάλι τοπικός φόρτος α_1 = 4 και τοπικός φόρτος α_2 = 13! Αυτό είναι λογικό γιατί η απόσταση των δυο agents δεν μειώθηκε.

Για την αποφυγή των παραπάνω εφαρμόζεται μηχανισμός αποτροπής ανταλλαγών στον SAM. Ένας κόμβος (στο προηγούμενο παράδειγμα ο n_1) που αποφασίζει να μετακινήσει κάποιον agent του (α_2) σε γείτονα κόμβο (n_4), στέλνει πρώτα *μήνυμα ελέγχου* που εξετάζει αν σε αυτόν τον γείτονα περιέχεται κάποιος συγγενής του agent του (α_1) που έχει επιλεχθεί να μετακομίσει στο μέρος του. Αν ναι, τότε όποιος από τους δύο agents είναι πατέρας (α_1) μετακομίζει κανονικά ενώ ο άλλος περιμένει στην αρχική του θέση. Βέβαια, με αυτόν τον τρόπο οι κόμβοι αναγκάζονται να ανταλλάσσουν δύο μηνύματα για κάθε μετακόμιση, που μπορεί να φαίνονται αμελητέα αλλά σε

ένα μεγάλο δίκτυο με πολλές αιτήσεις για μετακόμιση δεν είναι.

Τελικά, η μείωση του φόρτου κάποια στιγμή σταθεροποιείται όταν δεν υπάρχουν πια επωφελής μετακινήσεις στο δίκτυο, ή με άλλα λόγια όταν οι agents τοποθετούνται στην βέλτιστη θέση τους στο δίκτυο. Ο αριθμός των μετακινήσεων είναι λογικό να είναι μετρήσιμος εφόσον αφενός έχει εξαλειφθεί η πιθανότητα άπειρων ανταλλαγών και αφετέρου όταν η μετακίνηση κάποιου agent προς μια κατεύθυνση έχει όφελος x σημαίνει ότι μόλις ολοκληρώσει όλα τα βήματα που θα μειώσουν το φόρτο κατά x μονάδες οποιαδήποτε παραπέρα μετακίνηση θα έχει αρνητικό κέρδος (σημειώνεται ότι για x μονάδες ο μέγιστος αριθμός βημάτων που μπορεί να ακολουθήσει είναι x βήματα με κέρδος=1 για το κάθε ένα). Όταν και ο τελευταίος κόμβος διαπιστώσει ότι δεν υπάρχει επωφελής προορισμός για τους agent που φιλοξενεί δεν στέλνονται άλλα μηνύματα για ανακατανομή των agent και το δίκτυο ηρεμεί.

Κεφάλαιο 2.2: Περιορισμοί Μνήμης στη Λειτουργία του SAM

Τα ασύρματα δίκτυα αισθητήρων δεν διαθέτουν άπειρη μνήμη όπως απλουστευτικά θεωρήσαμε στην προηγούμενη υποενότητα. Στην πραγματικότητα είναι ζήτημα κατασκευαστικού κόστους η μνήμη και είναι περιορισμένη. Αυτό συνεπάγεται ότι δεν μπορούν όλες οι μετακινήσεις να εκτελεστούν. Ακόμα και αν είναι επωφελής η μετακίνηση κάποιου agent πρέπει ο κόμβος προορισμός να διαθέτει αρκετή ελεύθερη μνήμη ώστε να τον φιλοξενήσει. Για να μπορεί λοιπόν ο SAM να προσαρμοστεί σε αυτές τις καταστάσεις αναπτύχθηκαν δύο μέθοδοι που εφαρμόζονται από τους κόμβους του δικτύου ώστε να έχουν επίγνωση της διαθέσιμης μνήμης των γειτόνων τους.

Η πρώτη είναι η *Inquire-Lock Before (ILB)*. Κατά αυτήν, οι κόμβοι προτού αποφασίσουν που θα μεταφέρουν τον agent που επέλεξαν, στέλνουν σε όλους τους γείτονες αίτηση δέσμευσης μνήμης ίσης με το μέγεθος του agent. Αν δεν διαθέτουν αρκετή μνήμη απαντούν με σχετικό μήνυμα, αλλιώς ενημερώνουν για το πόσο ελεύθερο χώρο έχουν και δεσμεύουν όσο χρειάζεται για να

εκτελεστεί η μετακίνηση. Ύστερα από αυτό, οι κόμβοι έχουν μια λίστα των γειτόνων στους οποίους θα είναι εξασφαλισμένη η μετακόμιση των agent που επέλεξαν. Όποιος μέσα από τη λίστα είναι ο πιο επωφελής προορισμός τελικά προτιμάται ενώ στους υπόλοιπους στέλνεται σχετικό μήνυμα για να απελευθερώσουν το χώρο που δέσμευσαν και η λίστα διαγράφεται.

Στη δεύτερη μέθοδο, την *Inquire-Lock After (ILA)*, οι κόμβοι διατηρούν μονίμως μια λίστα, με πιθανότατα απαρχαιωμένα, μη έγκυρα στοιχεία για τον ελεύθερο χώρο όλων των γειτόνων τους. Αρχικά, η λίστα θεωρεί ότι όλοι έχουν άπειρη ελεύθερη μνήμη. Βάση αυτής, στέλνονται αιτήσεις δέσμευσης χώρου σε όσους γείτονες εμφανίζονται να έχουν αρκετή ελεύθερη μνήμη για να φιλοξενήσουν τους επιλεγμένους για μετακόμιση agents. Οι γείτονες απαντούν όπως πριν με τον διαθέσιμο τους χώρο και έτσι ενημερώνεται η λίστα με τα καινούρια δεδομένα. Λόγω των απαρχαιωμένων στοιχείων στη λίστα, είναι πιθανόν ορισμένοι κόμβοι της να μην επιλέγονται για μετακόμιση, ενώ στην πραγματικότητα να διαθέτουν όσο χώρο χρειάζεται. Γι αυτό το λόγο, με κάποιο συντελεστή πιθανότητας, κάθε φορά ορισμένοι από τους κόμβους θεωρείται ότι διαθέτουν όλη τους τη μνήμη ελεύθερη και συγκαταλέγονται στους πιθανούς προορισμούς αποστολής του agent προς μετακόμιση. Αν, τελικά, επιλεχθεί ένας από αυτούς σαν πιο επωφελής προορισμός απορρίπτει την αίτηση για φιλοξενία του agent και απαντά με τον διαθέσιμο του χώρο ώστε να ανανεωθεί η λίστα του κόμβου που φιλοξενεί αυτόν τον agent.

Και για στις δύο μεθόδους (ILA, ILB) η μείωση του φόρτου επικοινωνίας σταματά κάποια στιγμή μαζί με τις μετακινήσεις. Σε αντίθεση όμως με πριν, που δεν υπήρχαν περιορισμοί μνήμης και οι agents ταξίδευαν ελεύθερα όσο βρισκόταν κάποιος επωφελής προορισμός, εδώ συναντούν εμπόδια στην πορεία τους. Ο πιο επωφελής προορισμός τους μπορεί να μην έχει χώρο να τους φιλοξενήσει και να μην μπορούν να τον φτάσουν. Συνεπώς, ενώ υπάρχουν προοπτικές παραπέρα μείωσης του συνολικού φόρτου δεν αξιοποιούνται λόγω αυτών των περιορισμών στη μνήμη. Στο επόμενο κεφάλαιο εισάγονται τρόποι να προσπεραστούν αυτές οι δυσκολίες μέσα από *εκτοπίσεις*.

ΚΕΦΑΛΑΙΟ 3

ΕΚΤΟΠΙΣΕΙΣ

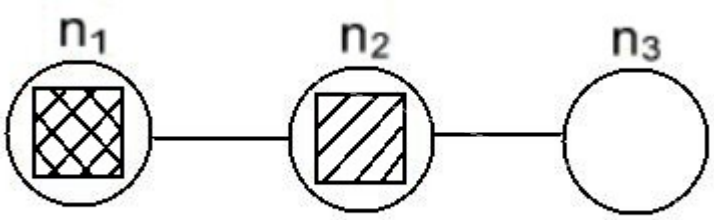
Σε αυτό το κεφάλαιο αναλύεται η εφαρμογή μεθόδων εκτοπίσεων στον αλγόριθμο Single Agent Migration, ώστε να ξεπεραστούν εμπόδια που εισάγονται λόγω περιορισμών μνήμης και να επιτευχθεί μεγαλύτερη μείωση του φόρτου επικοινωνίας. Στο τέλος, γίνεται μια σύγκριση των αποτελεσμάτων με αυτά των μεθόδων που αναπτύχθηκαν στα προηγούμενα κεφάλαια (ILA, ILB).

Κεφάλαιο 3.1: Η Μέθοδος των Εκτοπίσεων (evictions)

Σε ένα θεωρητικό δίκτυο κόμβων με άπειρη μνήμη οι agents μπορούν να ταξιδέψουν μέχρι τον πιο επωφελή τους προορισμό ανεμπόδιστα. Στην πραγματικότητα όμως, οι κόμβοι διαθέτουν συγκεκριμένο αποθηκευτικό χώρο και κάθε agent που φιλοξενούν καταλαμβάνει ένα μέρος του. Σε κάθε χρονική στιγμή, αφαιρώντας τα μεγέθη όλων των agent που φιλοξενούν βρίσκουν τον ελεύθερο χώρος τους.

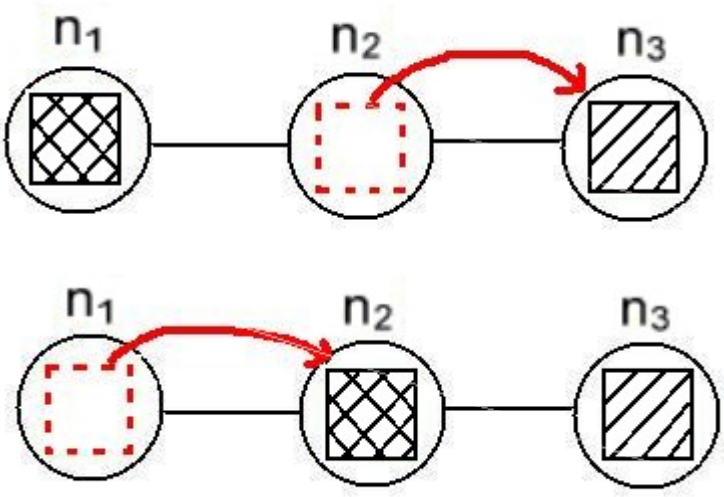
Στον αλγόριθμό μας (SAM), για να μπορεί να εκτελεστεί η μετακόμιση κάποιου agent απαιτείται ο κόμβος που θα τον παραλάβει να διαθέτει αρκετή ελεύθερη μνήμη για να τον υποδεχτεί και πρέπει να είναι τουλάχιστον όσο το μέγεθος του agent. Αυτό όμως δεν είναι πάντοτε εφικτό. Πολλές φορές ενώ είναι ωφέλιμο (δηλαδή κέρδος>0) να μετακινηθεί ένας agent δεν υπάρχει διαθέσιμος ελεύθερος χώρος να φιλοξενηθεί! Δεδομένων μάλιστα, τόσο της αρχικής μη βέλτιστης τοποθέτησης, όσο και των πολυάριθμων αλλαγών που προκαλούνται κατά την εκτέλεση του

αλγορίθμου, το πρόβλημα αυτό παρουσιάζεται πολύ συχνά. Για την επίλυση του καταφεύγουμε στη χρήση *εκτοπίσεων*. Οι κόμβοι διώχνουν δηλαδή agents που τους δεσμεύουν μνήμη, σε άλλους, γείτονες ώστε να έχουν αρκετό χώρο να υποδεχτούν κάποιον agent για τον οποίο έλαβαν αίτηση φιλοξενίας.



Εικόνα 3.α. Ο agent α_1 ενώ είναι ωφέλιμο να μετακινηθεί στον n_2. Το πρόβλημα είναι ότι ο κόμβος n_2 δεν έχει αρκετή μνήμη για να φιλοξενήσει ταυτόχρονα και τον α_1 και τον α_2.

Η ανάγκη για εκτοπίσεις φαίνεται καθαρά στην εικόνα 3.α. Σε τέτοιες περιπτώσεις, ο μόνος τρόπος για να πραγματοποιηθεί η μετακίνηση του α_1 στον n_2 είναι ο α_2 να εκτοπιστεί στον κόμβο n_3. Αυτό βέβαια συνεπάγεται δύο περιπτώσεις. Είτε η μεταφορά του α_2 στον n_3 θα είναι ωφέλιμη είτε όχι. Γενικά, το κέρδος της εκτόπισης ενός agent το ονομάζουμε *ποινή (penalty)*, άσχετα αν είναι θετικό ή αρνητικό. Αν η ποινή είναι αρνητική υποδηλώνεται ότι η μετακίνηση του α_2 (λόγω εκτόπισης) δεν είναι ωφέλιμη. Παρόλα αυτά, η συνολική μείωση του φόρτου επικοινωνίας του συστήματος που θα επέλθει από τη μετακίνηση του α_1 μπορεί να είναι μεγαλύτερη από την ποινή που εισάγει η εκτόπιση του α_2. Έτσι, στη μέθοδο που προτείνουμε, επιλέγουμε να προχωρήσουμε σε εκτοπίσεις σε καταστάσεις που το *συνολικό κέρδος* που προκύπτει από το άθροισμα του κέρδους μετατόπισης + το άθροισμα των ποινών των εκτοπίσεων που προκάλεσε αυτή είναι >0. Αντίθετα, αν το συνολικό κέρδος είναι <=0 μια μετακόμιση δεν θεωρείται ωφέλιμη και ακυρώνεται.

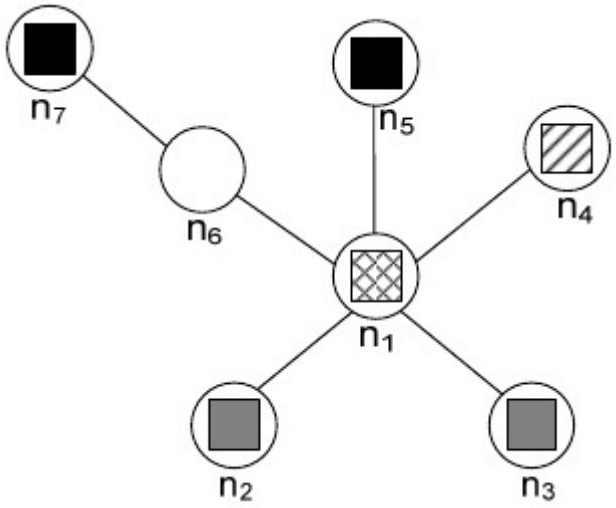


.

Εικόνα 3.β. Διαδικασία εκτόπισης. Ο α_2 "διώχνεται" στον κόμβο n_3 αφήνοντας αρκετό χώρο στον α_1 να ολοκληρώσει τη μετακόμιση του στον κόμβο n_2.

Ας δούμε μέσα από ένα πιο αναλυτικό παράδειγμα πως λειτουργεί ο αλγόριθμος με τις εκτοπίσεις.

Θεωρούμε το δέντρο εφαρμογής της εικόνας 2.α. τοποθετημένο στο δίκτυο όπως φαίνεται στην 3.γ.

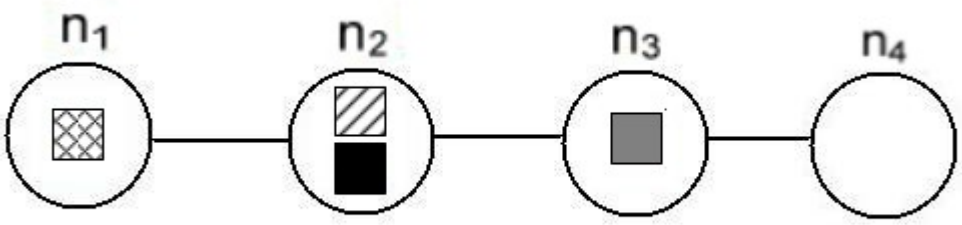


Εικόνα 3.γ. Μη βέλτιστη αρχική τοποθέτηση δέντρου εφαρμογής σε δίκτυο 7 κόμβων.

Αρχικά, ο μόνος άλλος ανεξαρτήτου κόμβου agent, ο α_1, είναι στη βέλτιστη του θέση αφού το κέρδος μετακίνησης τους σε οποιονδήποτε γείτονα δεν είναι θετικό. Για τον α_2 το κέρδος μετακόμισης του στον n_1 είναι κέρδος=τοπικός φόρτος – απομακρυσμένος = 17 – 8 = 9, όμως δεν υπάρχει ελεύθερος χώρος για να φιλοξενηθεί. Αφού λοιπόν, εξεταστεί αν η εκτόπιση του α_1 θα αποδεσμεύσει όση μνήμη χρειάζεται, αναζητείται εκείνος ο γείτονας του n_1 στον οποίο η μεταφορά του θα δώσει τη μικρότερη ποινή. Η μικρότερη ποινή δίνεται από την εκτόπιση του α_1 στον n_6 και είναι 0. Άρα, βγαίνει το συμπέρασμα ότι η εκτόπιση του α_1, ακολουθούμενη από την μετακόμιση του α_2 δίνει στο σύστημα συνολικό κέρδος (δηλαδή μείωση του φόρτου επικοινωνίας) 9 - 0 = 9 και έτσι εφαρμόζεται η διαδικασία όπως απεικονίζεται στην εικόνα 3.β.

Κεφάλαιο 3.2: Μέθοδοι Επιλογής των πιο Ωφέλιμων Εκτοπίσεων

Όταν εμφανιστεί η ανάγκη εκτόπισης για την ολοκλήρωση της μεταφοράς κάποιου agent είναι πιθανό να υπάρχουν περισσότεροι από ένας agent υποψήφιοι για εκτόπιση. Επιπλέον, ένας κόμβος ίσως να είναι ανήμπορος να εκτοπίσει κάποιον agent του λόγω περιορισμένης ελεύθερης μνήμης στον γείτονα προορισμό. Τότε χρειάζεται να γίνει μια ακόμα, διαδοχική εκτόπιση από αυτόν τον γείτονα προορισμό για να υποδεχτεί τον εκτοπισμένο agent. Η πρώτη εκτόπιση, που προξένησε η αίτηση μετακόμισης κάποιου agent, ονομάζεται *κύρια*, ενώ οι υπόλοιπες που προκύπτουν παρακάτω από την ανάγκη διεκπεραίωσης της κύριας εκτόπισης λέγονται *δευτερεύουσες* (εικόνα 3.δ).



Εικόνα 3.δ. Για την μετακίνηση του α_1 στον n_2 υπάρχουν πάνω από ένας agent διαθέσιμοι για εκτόπιση (α_2, α_3). Παράλληλα, για να ολοκληρωθεί η εκτόπιση του α_2 ή του α_3, χρειάζεται να εκτοπίσουν πρώτα τον α_5 στον κόμβο n_4. Η εκτόπιση του α_2 ή (του α_3) που απαιτείται για τη μετακίνηση του α_1 την ονομάζουμε **κύρια** ενώ την εκτόπιση του α_5 που απαιτείται για να ολοκληρωθεί η εκτόπιση του α_2 ή του α_3, την ονομάζουμε **δευτερεύουσα**.

Για την επιλογή του καταλληλότερου (ή καταλληλότερων) agent προς εκτόπιση ώστε η ποινή να είναι η ελάχιστη, προτείνονται 2 μέθοδοι ο *Single Path (SP)* και ο *Network Flooding (NF)*. Η βασική διαφορά τους εντοπίζεται στον τρόπο που υπολογίζουν την ποινή κάθε εκτόπισης και στο χειρισμό των αιτήσεων φιλοξενίας. Μία αίτηση φιλοξενίας περιέχει τα στοιχεία του agent που επιδιώκεται να μετακομίσει (μέγεθος κ.α.) καθώς και το εκτιμώμενο όφελος μεταφοράς του, όπως προκύπτει από τον τύπο (2).

Στον SP ο κόμβος που αναγκάζεται να εκτοπίσει εκτιμά για κάθε agent του τον προορισμό που θα δώσει την ελάχιστη ποινή, σύμφωνα με τον γνωστό μας τύπο τοπικός – απομακρυσμένος φόρτος (2). Αυτή, βέβαια, είναι μονάχα μια υπόθεση της αξίας της ποινής, αφού η ίδια η εκτόπιση μπορεί είτε να μην είναι εφικτή είτε να προκαλέσει και άλλες δευτερεύουσες. Έπειτα, με βάση αυτήν την τιμή κατατάσσει τους agents του σε αύξουσα σειρά και με τέτοια προτεραιότητα αθροίζει τα μεγέθη τους με τον υπάρχοντα ελεύθερο χώρο και τις ποινές μεταξύ τους μέχρι το πρώτο σύνολο να είναι τουλάχιστον ίσο με το μέγεθος ή το δεύτερο να ξεπερνάει το όφελος υποδοχής του agent που αιτήθηκε φιλοξενία στον κόμβο. Αν οι ποινές είναι περισσότερες από το όφελος ακυρώνονται οι εκτοπίσεις, διαφορετικά στέλνονται αιτήσεις φιλοξενίας για τους υποψήφιους προς εκτόπιση agents και προσωρινά "κλειδώνεται" όσος ελεύθερος χώρος υπάρχει και όσος δημιουργείται από τις πιθανές εκτοπίσεις. Σε περίπτωση που έστω και μια εκτόπιση αποτύχει,τα πάντα ακυρώνονται και ο κόμβος στέλνει μήνυμα αποτυχίας υποδοχής. Στον SP δηλαδή οι εκτοπίσεις επιλέγονται όχι με βάση την πραγματική τους ποινή αλλά μιας εκτίμησής της. Στο παράδειγμα της εικόνας 3.δ ο κόμβος θα καθόριζε σαν ποινή του α_2 το κέρδος ή τη ζημιά που θα προκαλούσε η εκτόπιση του από τον n_2 στον n_3, χωρίς να συνυπολογίζει την επιπρόσθετη ποινή από την εκτόπιση του α_5 στον n_4. Αντιθέτως, αυτό ακριβώς καταφέρνει να ξεχωρίσει στον NF!

Ο NF εκτελεί εικονικά όλες τις *δευτερεύουσες εκτοπίσεις* (αυτές που απαιτούνται για να μπορεί να ολοκληρωθεί μια άλλη εκτόπιση). Έτσι μπορεί να κρίνει αφενός αν ένας υποψήφιος για εκτόπιση agent (α_2, α_3) μπορεί να ολοκληρώσει την εκτόπιση του και αφετέρου να κατατάξει τους agent του

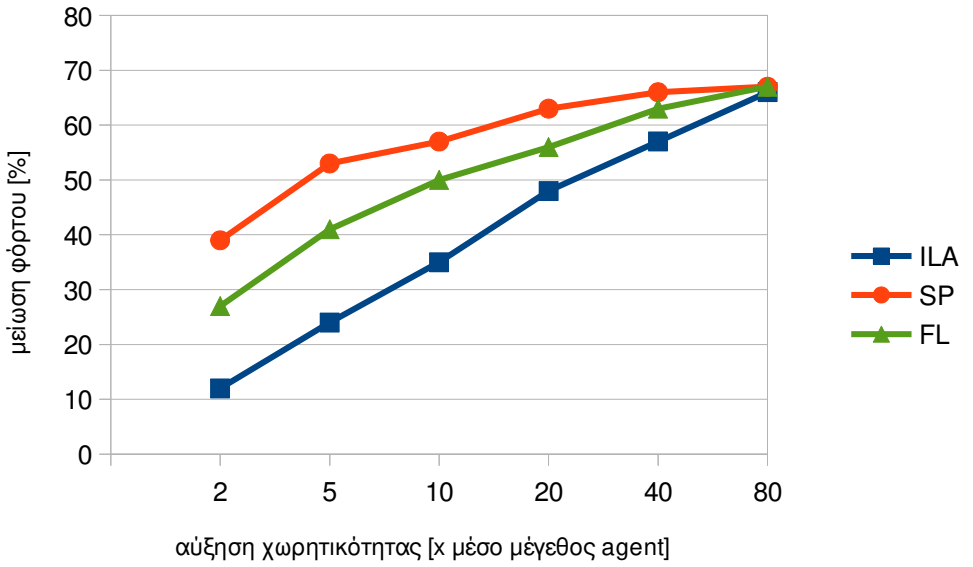
με βάση την πραγματική (και όχι υποθετική όπως ο SP) τους ποινή. Μέχρι εκεί εξαντλούνται και οι διαφορές των δύο μεθόδων.

Από την άλλη, κοινά και στις δύο μεθόδους οι κόμβοι μεταφέρουν από γείτονα σε γείτονα που εμπλέκεται σε εκτοπίσεις το αρχικό κέρδος + την ποινή, όπως διαμορφώθηκε μέχρι αυτούς. Λ.χ. έστω η μετακόμιση του α_1 στον n_2 απέφερε κέρδος 10 χωρίς τις ποινές για τις εκτοπίσεις που χρειάστηκαν. Έστω ότι η εκτόπιση του α_2 στον κόμβο n_3 εισάγει μια ποινή 6, του α_3 7 και του α_5 στον n_4 ποινή 2. Αν τελικά εκτοπιζόταν ο α_2 τότε στην προσπάθεια του να εκτοπίσει τον α_5, θα τον ενημέρωνε ότι μέχρι τώρα η διαφορά κέρδος – ποινή = 10 - 6 =4. Τώρα, ο α_5 γνωρίζει πως αν η εκτόπιση του προκαλέσει ποινή πάνω από >=4 δεν είναι ωφέλιμο να εκτοπιστεί ο α_2.

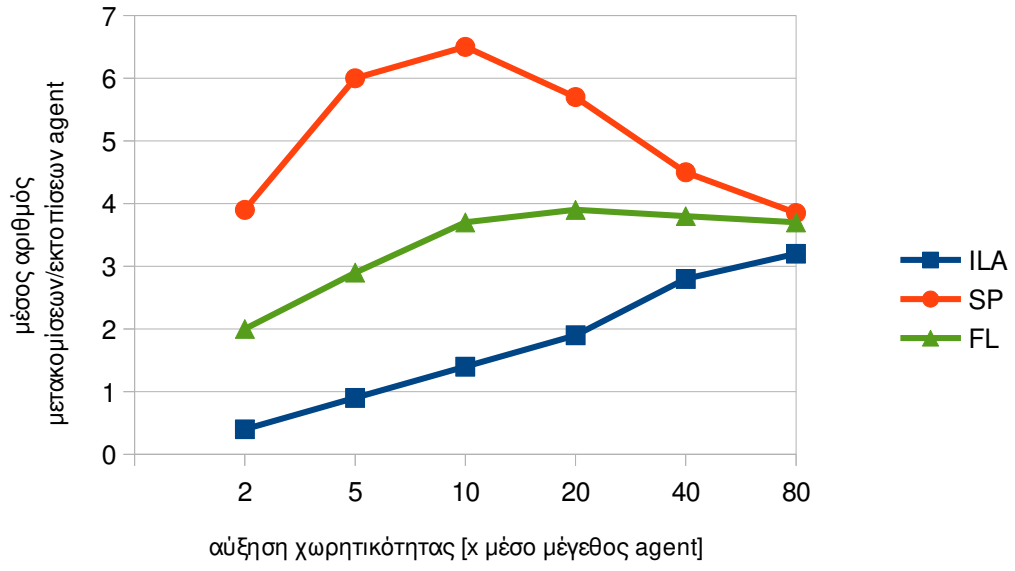
Κεφάλαιο 3.3: Σύγκριση Αποτελεσμάτων Εκτοπίσεων με ILA

Όπως και στον ILA, τα αποτελέσματα των NF και SP σταθεροποιούνται μετά από πολλούς κύκλους εφαρμογής τους. Δεν σημαίνει απαραίτητα αυτό όμως, ότι οι agents έφτασαν στον πιο επωφελή προορισμό τους, στο ίδιο δηλαδή που θα κατέληγαν αν δεν υπήρχαν περιορισμοί μνήμης. Για κάποιον agent μπορεί ακόμα να διαφαίνεται όφελος από μια πιθανή μετακόμιση του αλλά οι εκτοπίσεις που απαιτούνται να την καθιστούν μη κερδοφόρα για το σύστημα. Παρόλα αυτά, παρατηρείται μια σημαντική μείωση του φόρτου επικοινωνίας από την εφαρμογή των NF και SP η οποία παρουσιάζεται στα παρακάτω γραφήματα.

Τα πειράματα που παρήγαγαν τα αποτελέσματα των 3.ε και 3.στ έγιναν σε δίκτυα 50 κόμβων, που περιείχαν ταυτόχρονα 15 διαφορετικές εφαρμογές (5 των 50 agent, 5 των 20 agent και 5 των 10 agent). Κάθε φορά επιχειρούνταν αύξηση της χωρητικότητας του κάθε κόμβου (ανά 2, 5, 10, 20, 40, 80 φορές το μέσο μέγεθος των agent) και καταγράφονταν η διαφορά.



Εικόνα 3.ε. μείωση φόρτου ανά ποσό αύξησης της χωρητικότητας (50 κόμβοι, εφαρμογές διαφόρων πληθών agent).



Εικόνα 3.στ. Μεταβολή του μέσου αριθμού μετακομίσεων/εκτοπίσεων ανά ποσό αύξησης της χωρητικότητας (50 κόμβοι,διαφόρων πληθών agent).

Στο γράφημα της 3.ε είναι φανερό ότι οι μέθοδοι εκτοπίσεων φέρνουν μεγαλύτερη μείωση του φόρτου επικοινωνίας του συστήματος από τον ILA όσο η αποθηκευτική μνήμη των κόμβων περιορίζεται. Σε ορισμένα σημεία είναι κατά 330% μεγαλύτερη και είναι λογικό γιατί για μικρούς χώρους εκτελούνται περισσότερες μετακομίσεις-εκτοπίσεις όπως δείχνει και η 3.στ. Όταν η μνήμη είναι μεγαλύτερη συνεπάγεται ότι οι μετακινήσεις γίνονται πιο ελεύθερα. Λιγότερες εκτοπίσεις χρειάζονται και γι αυτό τα αποτελέσματα και των 3 μεθόδων συγκλίνουν στο τέλος.

Ακόμα ένα στοιχείο που πρέπει να προσέξουμε είναι η απόκλιση των τιμών παράγονται από τον NF και από τον SP. Είναι ξεκάθαρη η κατωτερότητα του Network Flooding και πηγάζει από την ίδια τη φύση του αλγορίθμου. Σε μεγάλης έκτασης συστήματα πολλοί agents στέλνουν ταυτόχρονα αιτήσεις. Στον NF πολλές από αυτές απορρίπτονται γιατί οι agents είναι περισσότερη ώρα από ότι στον SP απασχολημένοι (λόγω αναζήτησης των πραγματικών ποινών) και άρα λιγότερη ώρα διαθέσιμοι για εκτοπίσεις. Έτσι άλλωστε, εξηγείται πως ο SP καταφέρνει περισσότερες μετακομίσεις/εκτοπίσεις (βλ. 3.στ).

ΚΕΦΑΛΑΙΟ 4

ΕΚΤΟΠΙΣΕΙΣ ΜΕ ΑΡΝΗΤΙΚΟ ΚΕΡΔΟΣ

Σε αυτό το κεφάλαιο γίνεται παρουσίαση της καινούριας μεθόδου που προτείνουμε και του σκοπού της. Αφορά τον τρόπο που ένας agent αποφασίζεται αν θα μετακινηθεί ή όχι και συγκεκριμένα εισάγει τη δυνατότητα ένας agent να μετακινείται ακόμα και αν το κέρδος του είναι αρνητικό.

Κεφάλαιο 4.1: Μετακόμιση Agent με Αρνητικό Κέρδος

Είδαμε στα προηγούμενα κεφάλαια πως ο συνολικός φόρτος επικοινωνίας του συστήματος λόγω της επικοινωνίας μεταξύ των agent, μπορεί να μειωθεί φέρνοντας τους όσο πιο κοντά γίνεται. Είδαμε ακόμα, πως μέσα από εκτοπίσεις μπορεί να επιτευχθεί καλύτερη τοποθέτηση τους στο δίκτυο. Στο ταξίδι τους μέχρι το πιο ωφέλιμο προορισμό τους, οι agents αν συναντήσουν στη διαδρομή κάποιον κόμβο που δεν μπορεί να τους φιλοξενήσει (έστω προσωρινά) εξαιτίας περιορισμών μνήμης, ούτε μπορεί να προβεί σε εκτοπίσεις λόγω μεγάλης ποινής, σταματάνε την πορεία τους εκεί. Η μέθοδος που προτείνεται σε αυτή την εργασία στοχεύει ακριβώς στη δημιουργία προϋποθέσεων ώστε να προσπεραστεί ο κόμβος-εμπόδιο, χωρίς να παραβιάζεται η 1-hop φιλοσοφία του αλγορίθμου SAM.

Η καινούρια μέθοδος λοιπόν, η **Negative Eviction (NE)**, αρχίζει εκεί που σταματούν οι NF και SP. Πιο αναλυτικά, αφού η μετακόμιση ενός agent αποτύχει, δοκιμάζονται εκτοπίσεις. Όταν η ποινή

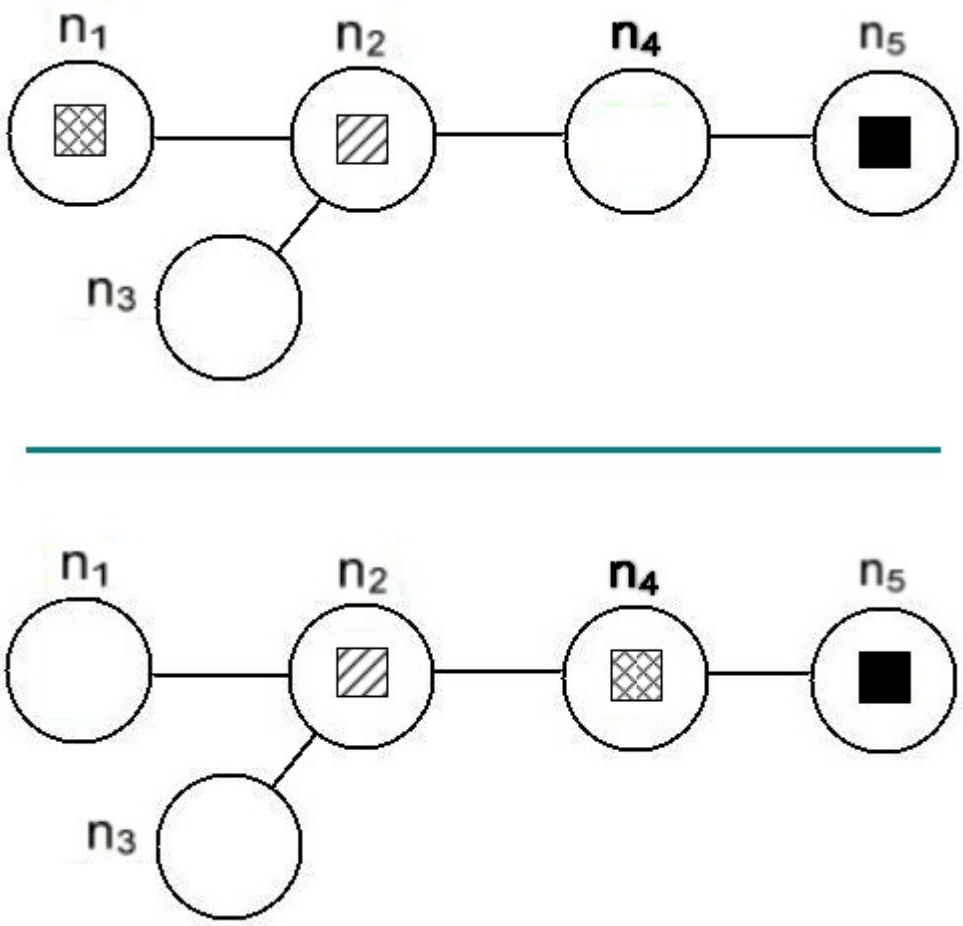
τους είναι μεγαλύτερη από το όφελος της μετακόμισης, για να μην είναι αρνητικό το κέρδος από τη διαδικασία, ο ΝΕ θα δοκιμάσει να εκτοπίσει μερικούς agent σε άλλους κόμβους αλλά μόνο προσωρινά! Στις καινούριες τους τοποθεσίες, οι εκτοπισμένοι agents θα παραμείνουν αμετακίνητοι μέχρις ότου να φτάσει σχετικό μήνυμα για την επαναφορά τους στον αρχικό τους κόμβο. Ο αρχικός τους κόμβος είναι τώρα ελεύθερος να υποδεχτεί τον agent που του αιτήθηκε φιλοξενία. Πρώτα, στην παρούσα του θέση δεσμεύεται χώρος ίσος με το μέγεθος του. Μετά που φεύγει, το επόμενο βήμα είναι να εξεταστεί αν είναι εφικτό και ωφέλιμο να μετακομίσει ξανά σε άλλο γείτονα παρακάτω και αν ναι, ακολουθείται ξανά η διαδικασία μεταφοράς του σε αυτόν (στέλνεται αίτηση φιλοξενίας, ελέγχεται αν υπάρχει αρκετή μνήμη ή χρειάζονται εκτοπίσεις κ.λπ). Σε αυτό το σημείο υπάρχουν μόνο δύο επιλογές για τον agent. Είτε θα ολοκληρωθεί η μεταφορά στο νέο προορισμό, είτε θα αποτύχει και τελικά θα αναγκαστεί να επιστρέψει στον αρχική του θέση από την οποία και στις δύο περιπτώσεις απελευθερώνεται ο χώρος που είχε δεσμευτεί περιμένοντας την απάντηση του. Ο ενδιάμεσος κόμβος έχει πάλι διαθέσιμο χώρο να υποδεχτεί όσους εκτόπισε και γι αυτό ενημερώνει τους προσωρινούς τους κόμβους σχετικά. Στο ψευδοκώδικα που ακολουθεί περιγράφεται η λειτουργία του ΝΕ.

Το πρωτόκολλο που εκτελείται στον αρχικό κόμβο

```
ΓΙΑ κάθε agent με agent.state!=TENTATIVE{
        ΓΙΑ κάθε γείτονα κόμβο{
                έλεγχος οφέλους μετακόμισης του agent στον κόμβο;
                ενημέρωση της πιο επωφελής μετακόμισης;
        }
        ΑΝ agent.state=NE ΤΟΤΕ{ //state=ASLEEP, TENTATIVE=όταν έχει
εκτοπιστεί προσωρινά, NE= όταν έχει μετακομίσει προσωρινά
                ορισμός ως πιο επωφελής για μετακόμιση agent;
                break;
        }
        ενημέρωση του πιο επωφελή για μετακόμιση agent;
}
ΑΝ υπάρχει μετακόμιση με κέρδος>0 ΤΟΤΕ{
        αποστολή αίτησης φιλοξενίας (agent, όφελος);
        αναμονή απάντησης;  //απάντηση=θετική, αρνητική, θετική_με_NE
        ΑΝ απάντηση=θετική ΤΟΤΕ
                ξεκίνα μετακόμιση;
                ΑΝ agent.state=NE ΤΟΤΕ
                        ξεκίνα επιστροφή των TENTATIVE agents;
        ΑΛΛΙΩΣ_ΑΝ απάντηση=θετική_με_NE ΤΟΤΕ{
                agent.state=NE;
                ξεκίνα μετακόμιση;
                δέσμευσε μνήμη ίση με agent.size;
        }
}ΑΛΛΙΩΣ ΑΝ agent.state=NE ΤΟΤΕ
        ξεκίνα επιστροφή στον αρχικό του κόμβο;
        ξεκίνα επιστροφή των TENTATIVE agents;
```

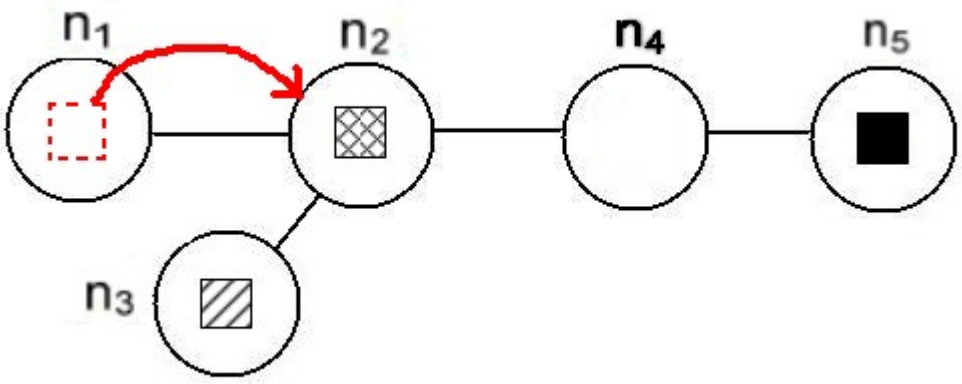
Το πρωτόκολλο που εκτελείται στον κόμβο που λαμβάνει αίτηση φιλοξενίας(agent, όφελος), αν όφελος >0

```
ΑΝ ελεύθερος_χώρος >= agent.size ΤΟΤΕ
        ΕΠΕΣΤΡΕΨΕ απάντηση (θετική);
ΑΛΛΙΩΣ_ΑΝ συνολική_μνήμη_κόμβου>=agent.size ΤΟΤΕ{
        κατάταξε τους τοπικούς agents σε λίστα με φθήνουσα σειρά ποινής;  /*Με NF ή SP. Η λίστα περιέχει (agent, ποινή, πιο_επωφελής_κόμβος_μετακόμισης) */
        συνολικό_μέγεθος=0;  συνολική_ποινή=0;
        ΕΠΑΝΑΛΑΒΕ{
                επέλεξε επόμενο τοπικό agent από τηλίστα;
                συνολικό_μέγεθος+=τοπικος_agent.size;
                συνολική_ποινή+=τοπικός_agent.ποινή;
        }ΜΕΧΡΙ(συνολικό_μέγεθος+ελεύθερος_χώρος >= agent.size Ή
                λίστα δεν έχει άλλα στοιχεία);
        ΑΝ συνολικό_μέγεθος+ελεύθερος_χώρος < agent.size ΤΟΤΕ
                ΕΠΕΣΤΡΕΨΕ απάντηση (αρνητική);
        ΑΛΛΙΩΣ{
                λίστα=λίστα-{όσοι τοπικοί agents δεν επιλέχθηκαν};
                ΓΙΑ κάθε agent στην λίστα{
                        στείλε αίτηση φιλοξενίας (agent, όφελος-ποινή) στον πιο_επωφελή_κόμβο_μετακόμισης; //εκτοπίσεις με ποινή<όφελος
                        αναμονή απάντησης;
                }
                ΑΝ έστω μία απάντηση = αρνητική ΤΟΤΕ ΕΠΕΣΤΡΕΨΕ απάντηση (αρνητική);
                ΑΛΛΙΩΣ{
                        ΑΝ συνολική_ποινή < όφελος ΤΟΤΕ{
                                ξεκίνα εκτοπίσεις;
                                ΕΠΕΣΤΡΕΨΕ απάντηση (θετική);
                        }ΑΛΛΙΩΣ{
                                ξεκίνα εκτοπίσεις και θέσε για κάθε μια agent.state=TENTATIVE;
                                ΕΠΕΣΤΡΕΨΕ απάντηση (θετική με ΝΕ);
                        }
        }
}ΑΛΛΙΩΣ
        επέστρεψε απάντηση (αρνητική);
```



Εικόνα 4.α. Επάνω: η αρχική τοποθέτηση των agents στο δίκτυο. Ο n_2 περιέχει ελεύθερο χώρο μόνο για να φιλοξενήσει τον α_2. Το κέρδος μετακίνησης του α_1 στον n_2 είναι αρνητικό λόγω εκτόπισης του α_2. Κάτω: βέλτιστη τοποθέτηση• ο α_1 ήρθε πιο κοντά στον α_3.

Η εικόνα 4.α είναι χαρακτηριστική του προβλήματος μας. Ο συνολικός φόρτος του συστήματος θα μειωθεί αν ο α_1 πλησιάσει τον α_3. Για να τον προσεγγίσει, ο κόμβος n_2 στέκεται εμπόδιο, αφού δεν διαθέτει αρκετό ελεύθερο χώρο να φιλοξενήσει προσωρινά τον α_1. Ακόμα χειρότερα, η ποινή από την εκτόπιση του α_2 (που δεν έχει συγγένεια με τον α_1 ή α_3) είναι μεγαλύτερη από το όφελος της μετακόμισης του α_1 στον n_2. Σε αυτό το σημείο οι NF και SP θα ακύρωναν τη μετακόμιση. Με την μέθοδο των εκτοπίσεων όμως, ο α_2 παρά το αρνητικό κέρδος εκτοπίζεται στον n_3 (εξετάζουμε στην επόμενη ενότητα την περίπτωση να εκτοπιστεί στον n_4) και ο α_1 κάνει ένα πρώτο βήμα φτάνοντας στον κόμβο n_2 (εικόνα 4.β).



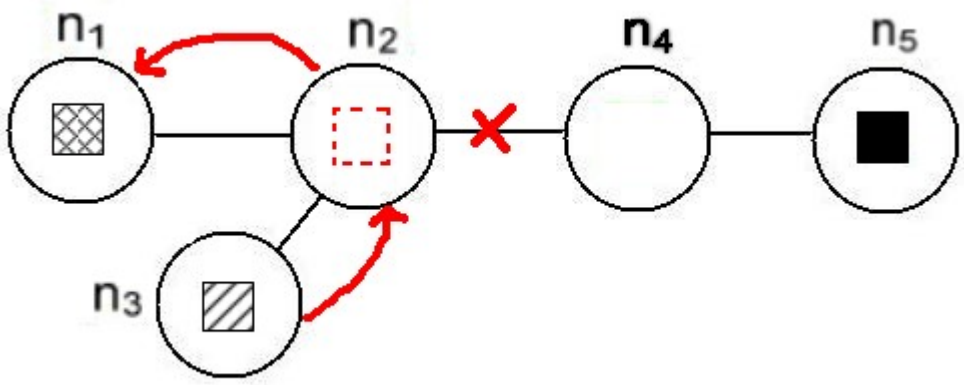
εικόνα 4.β. Ο α_2 εκτοπίστηκε προσωρινά στον n_3 για να "ανοίξει δρόμο" για τον α_1.

Ταυτόχρονα, ο agent α_2 μπαίνει σε μια ειδική κατάσταση αδράνειας (state=TENTATIVE στον ψευδοκώδικα) αποτρέποντας τον από το να μετακινηθεί ή να εκτοπιστεί ενώ οι κόμβοι n_1 και n_2 δεσμεύουν όσο από τον ελεύθερο τους χώρο χρειάζεται για την ομαλή επιστροφή των α_1 και α_2[3] αντίστοιχα. Όταν σε κάποια χρονική στιγμή αργότερα ο κόμβος n_2 αφυπνιστεί, δίνει προτεραιότητα στη μετακίνηση του α_1. Αν ο α_1 μετακινηθεί σε οποιονδήποτε γείτονα κόμβο[4] τότε ο n_2 στέλνει ειδικό μήνυμα στον n_3 για την επιστροφή του α_2 και ο n_1 αποδεσμεύει το χώρο που κράταγε για τον α_1. Αν όμως εντοπιστεί πως η μετακίνηση του α_1 είναι αδύνατη, τότε ο κόμβος n_1 τον "επιστρέφει" στον κόμβο n_1 και τότε εκτελείται η επιστροφή του agent α_2 όπως περιγράφεται παρακάτω (εικόνα 4.γ).

Έτσι, αξιοποιώντας αυτή τη μέθοδο, φαίνεται εφικτή μια ακόμα καλύτερη τοποθέτηση agent στο δίκτυο, επιτρέποντας στον α_1 να προσπεράσει το εμπόδιο του κόμβου n_2 και την ίδια στιγμή να μην δημιουργήσει πρόβλημα με τους κόμβους που περιέχει.

3 Αν το μέγεθος του b είναι μεγαλύτερο από το μέγεθος του a ο 2 δεσμεύει size(b)-size(a) από τον ελεύθερό του χώρο. Αλλιώς, αν size(b)<=size(a) δεν χρειάζεται να δεσμεύσει χώρο, αφού ο απαραίτητος χώρος για τη φιλοξενία του b θα απελευθερωθεί μόλις μετακινηθεί ο a.

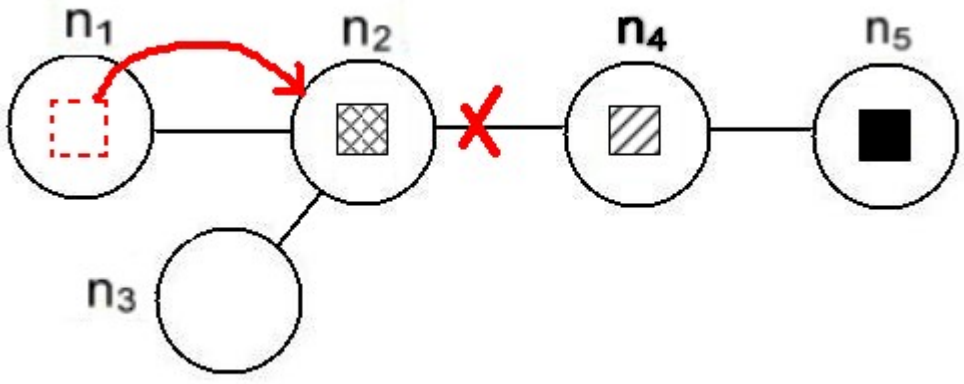
4 Προφανώς ο a δεν θα επιλεγεί να μετακινηθεί πίσω στον 1 από όπου ήρθε, αφού η μετακίνηση του μακρυά από αυτόν είχε θετικό όφελος. Άρα, η επιστροφή του θα έχει αρνητικό.



εικόνα 4.γ. Ο α_1 δεν ήταν δυνατό να μετακομίσει κάπου αλλού πέρα από τον n_2 οπότε επιστρέφει στον n_1 και ενημερώνεται ο α_2 να επιστρέψει πάλι στην αρχική του θέση.

Στο προηγούμενο παράδειγμα οι καταστάσεις είναι ιδανικές για την επίτευξη του βέλτιστου αποτελέσματος. Τι θα γινόταν όμως αν ο α_2 τύχαινε να εκτοπιστεί στον κόμβο n_4 (εικόνα 4.δ);

Ο κόμβος n_2 δεν μπορεί εκ των προτέρων να γνωρίζει την "κατεύθυνση" του agent α_1. Δηλαδή δεν μπορεί πριν να παραλάβει τον α_1 να γνωρίζει ποιος θα ναι ο επόμενος προορισμός του (στο παράδειγμα της εικόνα ο n_4). Έτσι, είναι πολύ πιθανό, να εκτοπίσει σε αυτόν τον α_2 εμποδίζοντας ουσιαστικά τον α_1 από την προσέγγιση του α_3 και αναγκάζοντας πάλι αυτόν να επιστρέψει στον κόμβο n_1 και τον α_2 στον n_2. Όλο αυτό, μπορεί να εισάγει στο σύστημα ανεπιθύμητες μετακινήσεις και καθυστερήσεις που μάλιστα, δεν αποκλείεται να επαναλαμβάνονται ξανά και ξανά.



εικόνα 4.δ. Ο α_2 εκτοπίστηκε στον κόμβο που ήθελε ο α_1 να φτάσει και τον εμποδίζει.

ΚΕΦΑΛΑΙΟ 5

ΠΡΟΣΟΜΟΙΩΣΗ ΚΑΙ ΑΠΟΤΕΛΕΣΜΑΤΑ

Σε αυτό το κεφάλαιο παρουσιάζεται συνοπτικά η εφαρμογή σε C++ που αναπτύχθηκε για τη προσομοίωση του δικτύου και της λειτουργίας του Negative Eviction αλγορίθμου που προτείνεται σε αυτήν την εργασία. Ακόμα, μέσα από γραφήματα εκτίθενται τα αποτελέσματα από της προσομοιώσει και τα συμπεράσματα που προκύπτουν από αυτές,

Κεφάλαιο 5.1: Ανάπτυξη Εφαρμογής για την Πρακτική Δοκιμή

Ιδεατά, για τη δοκιμή της καινούριας μεθόδου που προτείνει αυτή η εργασία, θα πρεπε να είχαμε πρόσβαση σε ένα πραγματικό δίκτυο αισθητήρων, τέτοιο που να πληρεί τα χαρακτηριστικά που περιγράφηκαν στα πρώτα κεφάλαια. Επειδή, όμως, κάτι τέτοιο δεν ήταν εφικτό η εξέταση των αποτελεσμάτων έγινε μέσα από προσομοιώσεις. Τόσο η εικονική ανάπτυξη του δίκτυο όσο και των αποτελούμενων από agents, εφαρμογών, υλοποιήθηκε σε γλώσσα προγραμματισμού C++. Στο παράρτημα στο τέλος της εργασίας παρατίθεται ολόκληρος ο κώδικας.

Για την αναπαράσταση της αλληλοσυσχέτισης τους αναπτύχθηκε εφαρμογή που δημιουργεί και αποτυπώνει σε ένα αρχείο ".txt" το δέντρο των agents. Όμοια σε άλλο αρχείο, δεύτερη εφαρμογή κατασκευάζει το γράφο του δικτύου, ακολουθώντας τον αλγόριθμο του Kruskal[14] για την αποφυγή δημιουργίας κύκλων. Κάθε φορά που τρέχουν αυτές οι εφαρμογές παράγουν καινούρια, ψευδο-τυχαία αποτελέσματα. Τέλος, η κύρια εφαρμογή, σαρώνει τις πληροφορίες από τα δύο αυτά αρχεία και τις αξιοποιεί για την κατασκευή της προσομοίωσης.

Στην κύρια εφαρμογή, οι agents και οι κόμβοι παίρνουν "σάρκα και οστά". Κατασκευάζονται agent και node structs με τα βασικά γνωρίσματα τους (εικόνες 5.α, 5.β). Έπειτα, οι agents τοποθετούνται μέσα στους κόμβους σύμφωνα με τις υποδείξεις των .txt αρχείων και αρχικοποιούνται με τυχαίες τιμές μεγεθών.

```
struct agent{
    int ID;
    int size;
    int state;
    int parentAgentID;
    list childrenAgents;
        ...
}
```

Εικόνα 5.α. Βασικές μεταβλητές για τον ορισμό του agent struct.

```
struct node{
    int ID;
    int size;
    int mode;
    int freeSpace;
    list agentsContained;
    list neighbourNodes;
        ...
}
```

Εικόνα 5.β. Βασικές μεταβλητές για τον ορισμό του node struct.

Όσον αφορά τη λειτουργία τους, οι κόμβοι είναι ικανοί να στέλνουν και να λαμβάνουν αιτήσεις μετακίνησης agent και μάλιστα ταυτόχρονα. Με ανταλλαγές κατάλληλων μηνυμάτων

συνεργάζονται αρμονικά και ενημερώνονται για το αν μια μετακίνηση είναι εφικτή ή όχι. Ένα μήνυμα μετακίνησης (**sendHostRequest**(agent, destinationNode, benefit)), φέρει τη διεύθυνση του agent προς μετακίνηση, τον κόμβο προορισμό και το κέρδος μετακίνησης. Αν το μήνυμα στέλνεται για την εκτόπιση κάποιου agent b, ώστε να χωρέσει στον κόμβο του κάποιος a, τότε benefit=κέρδος_μετακόμισης_a – ποινή_εκτόπισης_b. Έτσι, σε κάθε hostRequest για εκτόπιση, το benefit στέλνεται μειούμενο κατά την ποινή εκτόπισης του agent.

Επιπλέον, ένα σημαντικό κομμάτι της λειτουργίας του εικονικού δικτύου είναι η έννοια του χρόνου. Χωρίς αυτό ούτε οι χρονικές καθυστερήσεις που προκαλούν οι μετακινήσεις των agent θα ήταν μετρήσιμες, ούτε και θα μπορούσαμε να προσομοιώσουμε την ταυτόχρονη δράση των κόμβων. Για αυτό το σκοπό, το πρόγραμμα εκτελείται σε κύκλους εκτέλεσης (time), που αντιπροσωπεύουν μονάδες χρόνου και δημιουργείται μια δομή ουράς που αποτελείται από timeHeap structs (εικόνα 4.γ).

```
struct timeHeap{
        int time;
        int commandID;
        list commandArguments;
        struct timeHeap* next;
}
```

Εικόνα 5.γ. Ορισμός του timeHip struct.

Κάθε timeHeap struct περιέχει την ώρα που αντιπροσωπεύει (time), μια εντολή (commandID) προς εκτέλεση για εκείνη την ώρα και τα ορίσματα εισόδου της εντολής (commandArguments). Στην ουρά μας, μπορεί να υπάρχουν πολλά timeHeaps με ίδια ώρα εκτέλεσης. Χάριν απλότητας, η ώρα είναι απλοί ακέραιοι αριθμοί.

Το σύστημα υιοθετεί event-driven φιλοσοφία. Δηλαδή, παραμένει αδρανοποιημένο και η ροή του συνεχίζεται ή, καλύτερα, προωθείται από μηνύματα (π.χ. αιτήσεις φιλοξενίας) που στέλνουν οι κόμβοι στους γείτονες τους. Πιο συγκεκριμένα, το πρόγραμμα ξεκινά με όλους τους κόμβους σε κατάσταση (state) ASLEEP. Σε κάθε κύκλο εκτέλεσης (time) η εφαρμογή "ξυπνά" 4 τυχαίους κόμβους και εκείνοι εφαρμόζουν τον αλγόριθμο NE στέλνουν αιτήσεις σε γείτονες τους για να μετακομίσουν κάποιον από τους agents που φιλοξενούν. Αυτό αλυσιδωτά, οδηγεί στην αποστολή μιας σειράς νέων αιτήσεων (π.χ. για εκτοπίσεις) που απευθύνονται και ενεργοποιούν και άλλους αδρανοποιημένους κόμβους. Τα μηνύματα, αποθηκεύονται σαν εντολές κάποιου timeHeap struct και σε κάθε κύκλο, το πρόγραμμα ανατρέχει στην ουρά από timeHeaps, εκτελεί όλες τις εντολές που βρίσκονται εκεί με time ίσο με τη συγκεκριμένη χρονική στιγμή και τις αφαιρεί από της ουρά. Η λειτουργία του τελειώνει ύστερα από RUNTIME κύκλους εκτέλεσης, που ορίζονται από τον χρήστη. Στις δικές μας προσομοιώσεις φροντίσαμε να είναι αρκετές ώστε να προλάβουν να εκτελεστούν όλες οι εφικτές μετακομίσεις.

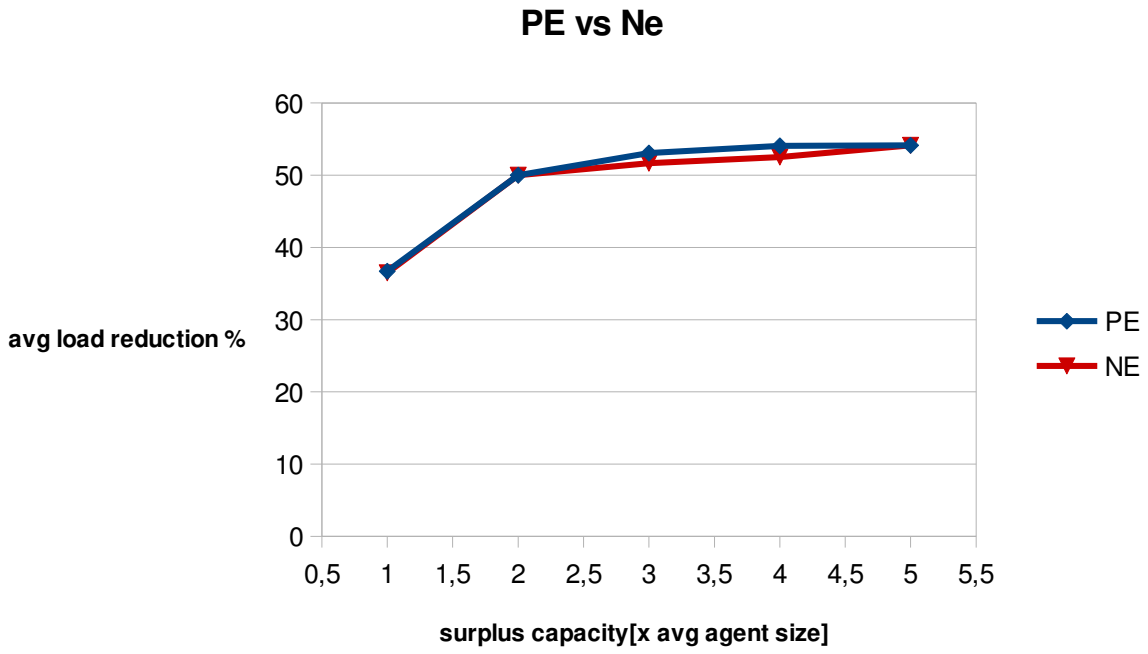
Τέλος, πρέπει να αναφερθεί, ότι η εφαρμογή υλοποιεί το Negative Eviction αλγόριθμο, με αξιοποίηση της Network Flooding μεθόδου για τον προσδιορισμό της ποινής που προκαλεί η κάθε εκτόπιση.

Κεφάλαιο 5.2: Σύγκριση Μείωσης Φόρτου Επικοινωνίας

Στην ενότητα αυτή μέσα από γραφήματα, αναπτύσσονται οι εκτιμήσεις από τα αποτελέσματα της εκτέλεσης της προσομοίωσης. Η σύγκριση γίνεται ανάμεσα στα αποτελέσματα της προσομοίωσης του Single Agent Migration αλγόριθμου με Network Flooding, που θα αναφέρεται στο εξής σαν *Possitive Eviction (PE)* και του ίδιο αλγόριθμο με εφαρμογή της δικιάς μας πρότασης για μετακόμιση με αρνητικό κέρδος NE. Τριών ειδών πειράματα διεξήχθησαν. Για κάθε ένα

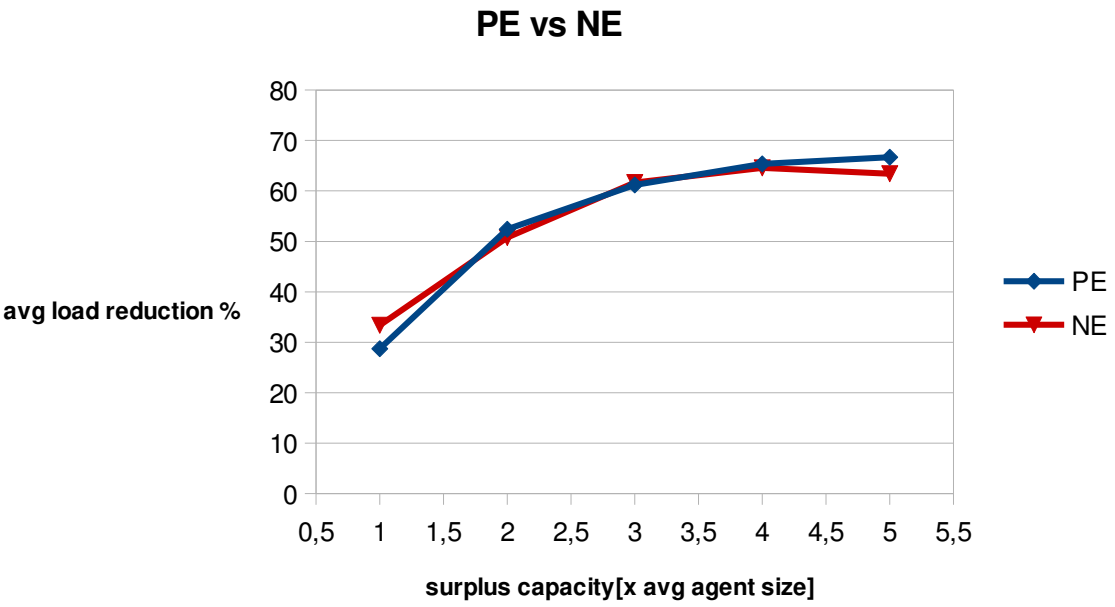
δημιουργούνται καινούρια τυχαία δέντρα εφαρμογών και δίκτυα κόμβων. Ο μέσος όρος από όλες τις τυχαίες περιπτώσεις διαμόρφωσε τα δεδομένα που καταγράφονται στις παρακάτω εικόνες.

Για κάθε δίκτυο και δέντρο κόμβων που δημιουργήθηκαν, η μέθοδος δοκιμάστηκε με τις ίδιες παραμέτρους, αυξάνοντας μόνο την χωρητικότητα των κόμβων 1,2,3,4 και 5 φορές το μέσο μέγεθος των agents του δέντρου εφαρμογών. Αυτή η προσέγγιση, παρέχει μεν στους agents μεγαλύτερη ευχέρεια μετακίνησης στο δίκτυο, αλλά στην πραγματική ζωή συνεπάγεται κατασκευή κόμβων με μεγαλύτερο αποθηκευτική ικανότητα και αύξηση του κόστους κατασκευής τους. Είναι αναμενόμενο, πως προχωρώντας σε πολύ μεγάλη αύξηση της μνήμης τα αποτελέσματα των δύο αλγορίθμων θα ταυτίζονται, αφού οι agent θα μπορούν να μετακινούνται ελεύθερα, χωρίς την ανάγκη εκτοπίσεων.

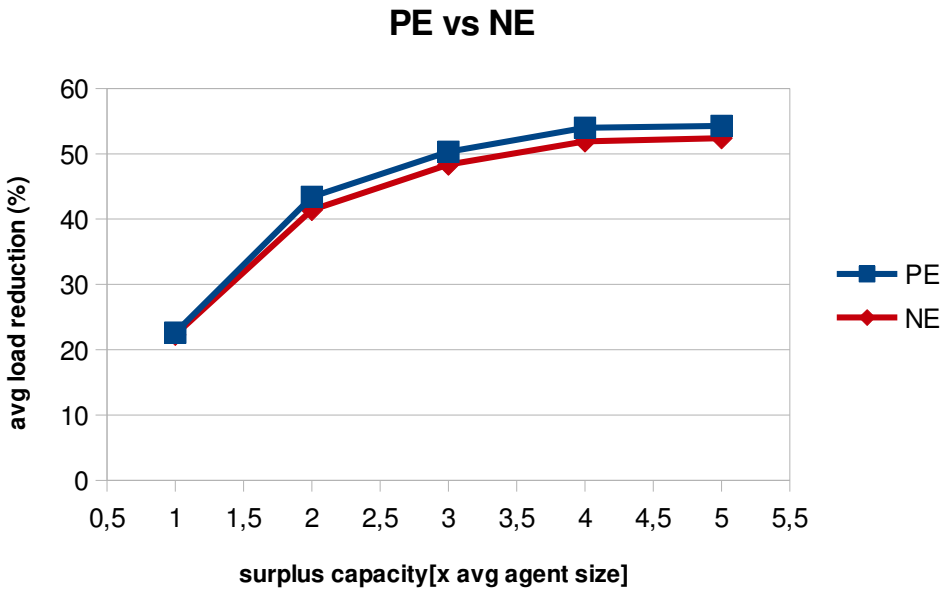


Εικόνα 5.δ. 10 agents σε δίκτυο 20 κόμβων. Μείωση φόρτου επικοινωνίας συστήματος σε σχέση με τη χωρητικότητα των κόμβων

.



Εικόνα 5.ε. 25 agents σε δίκτυο 50 κόμβων. Μείωση φόρτου επικοινωνίας συστήματος σε σχέση με τη χωρητικότητα των κόμβων.

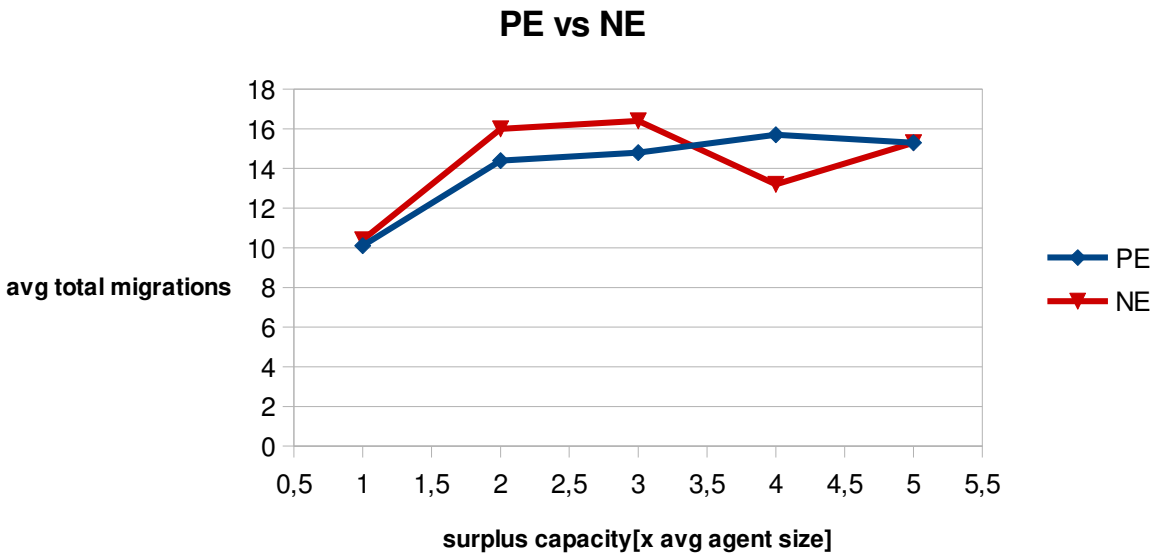


Εικόνα 5.στ. Διάφορα πλήθη agent (10, 25, 50) σε δίκτυο 50 κόμβων. Μείωση φόρτου επικοινωνίας συστήματος σε σχέση με τη χωρητικότητα των κόμβων.

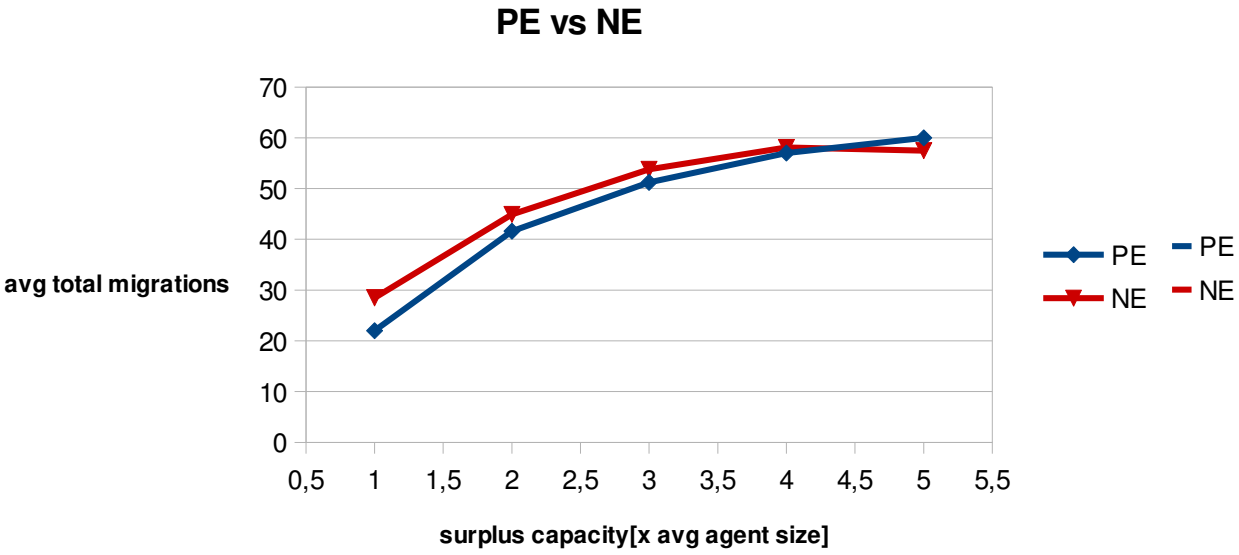
Από τα τρία πειράματα (εικόνες 5.δ, 5.ε, 5.στ), φανερώνεται συνολικά μικρότερη μείωση του φόρτου επικοινωνίας εφαρμόζοντας την καινούρια μέθοδο. Αν και υπάρχουν εξαιρέσεις σε ορισμένες περιπτώσεις όπου η καινούρια μέθοδος παρουσιάζει να μειώνει τον φόρτο, τα πράγματα είναι πιο ξεκάθαρα από τα αποτελέσματα του τρίτου πειράματος (5.στ) που συνδυάζει διαφορετικά πλήθη και δοκιμάζει μεγαλύτερο αριθμό agents.

Κεφάλαιο 5.3: Σύγκριση Αύξησης Μετακινήσεων των Agent

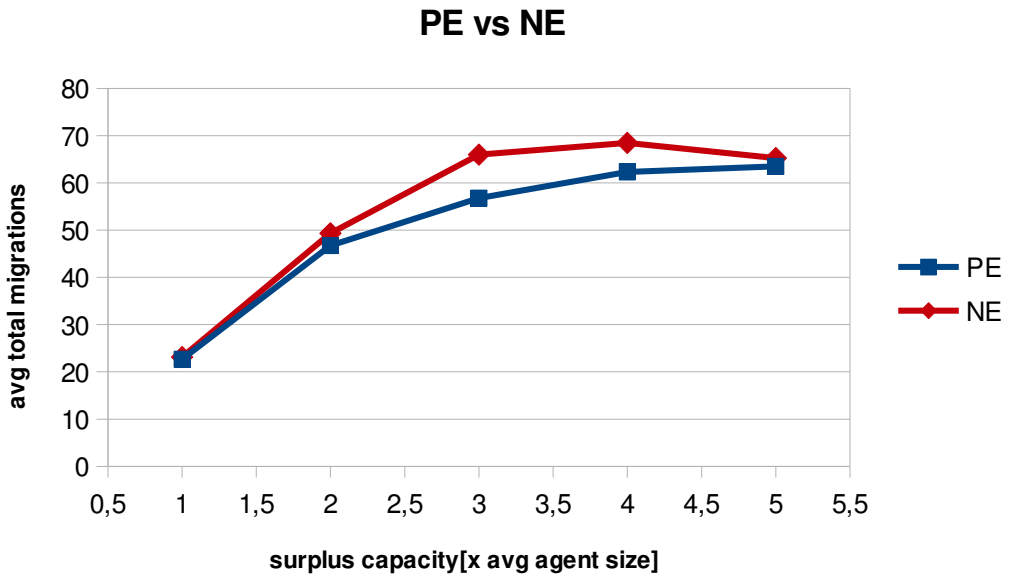
Με την αύξηση του μεγέθους μνήμης των κόμβων στα πειράματα, δημιουργούνται μεταβολές στον αριθμό των μετακινήσεων των agent. Οι επόμενες εικόνες (5.ζ, 5.η, 5.θ) αποτυπώνουν αυτές τις αλλαγές όπως προέκυψαν από τα προηγούμενα πειράματα.



Εικόνα 5.ζ. 10 agents σε δίκτυο 20 κόμβων. Μεταβολή του πλήθους μετακινήσεων σε σχέση με την αύξηση χωρητικότητας των κόμβων.



Εικόνα 5.η. 25 agents σε δίκτυο 50 κόμβων. Μεταβολή του πλήθους μετακινήσεων σε σχέση με την αύξηση χωρητικότητας των κόμβων.



Εικόνα 5.θ. Διάφορα πλήθη agent (10, 25, 50) σε δίκτυο 50 κόμβων. Μεταβολή του πλήθους μετακινήσεων σε σχέση με την αύξηση χωρητικότητας των κόμβων.

Γενικά τα αποτελέσματα παρουσιάζουν αύξηση των μετακινήσεων όσο μεγαλώνει η χωρητικότητα των κόμβων και μείωση του ρυθμού αύξησης για μεγάλα μεγέθη μνήμης, που είναι επόμενο αφού με περισσότερη μνήμη λιγότερες μετακινήσεις εμποδίζονται. Ταυτόχρονα, η καινούρια μέθοδος φαίνεται να προκαλεί περισσότερες μετακινήσεις μέσα στο δίκτυο που συνεπάγεται περισσότερο χρόνο για την ανατοποθέτηση των agent στο δίκτυο, αλλά και κατανάλωση ενέργειας από τους κόμβους του συστήματος για την διεκπεραίωση των επιπρόσθετων μετακινήσεων. Εξετάζοντας και τα αποτελέσματα από τη σύγκριση των αλγορίθμων SP και NF (βλ. κεφάλαιο 3.3), οι περισσότερες μετακομίσεις στον NE είναι λογικές, αφού ο καινούριος μας αλγόριθμος προχωρά σε περισσότερες και πιο σύνθετες μετακινήσεις από τον NF προκαλώντας έτσι περισσότερα conflicts.

ΕΠΙΛΟΓΟΣ

Συμπερασματικά, η πρακτική εξέταση απέρριψε τις θεωρητικές εκτιμήσεις. Η καινούρια μέθοδος που αναπτύχθηκε έδωσε μεν την δυνατότητα σε agents να προσεγγίσουν ακόμα περισσότερο τις βέλτιστες θέσεις τους, όπως αυτές υποδεικνύονται από την εφαρμογή του αλγορίθμου Single Agent Migration άνευ περιορισμών μνήμης, αλλά οι επιπλέον πόροι (agents, κόμβοι) που δεσμεύονται κατά τη λειτουργία της οδηγούν σε συγκρούσεις και αυτές με τη σειρά τους σε μικρότερη μείωση του φόρτου επικοινωνίας από αυτή που θα πετύχαινε ο PE.

ΒΙΒΛΙΟΓΡΑΦΙΑ ΚΑΙ ΑΛΛΕΣ ΑΝΑΦΟΡΕΣ

[1] Z. Abrams and J. Liu, “Greedy is Good: On Service Tree Placement for in-network Stream Processing,” in Proc. ICDCS, 2006.

[2] N. Tziritas, T. Loukopoulos, S. Lalis, P. Lampsas, “Agent placement in wireless embedded systems: memory space and energy optimizations,” in Proc. Int. Workshop on Performance Modeling, Evaluation and Optimization of Ubiquitous Computing and Networked Systems (PMEO-UCNS), 2010.

[3] S. Chai, Y. Li, J. Wang, and C. Wu, “An Energy-efficient Scheduling Algorithm for Computation-Intensive Tasks on NoC-based MPSoCs,” Journal of Computational Information Systems, vol. 9, no. 5, pp. 1817-1826, 2013.

[4] M. Mandelli, L. Ost, E. Carara, G. Guindani, T. Gouvea, G. Medeiros, F. G. Moraes, “Energy-aware Dynamic Task Mapping for NoC-based MPSoCS,” In Proc. ISCAS, 2011.

[5] O. Sinnen, A. To, and M. Kaur, “Contention-Aware Scheduling with Task Duplication,” in JPDC, vol. 71, no. 1, pp. 77-86, 2011.

[6] Y. Tian, J. Boangoat, E. Ekici and F. zg ner, “Real-time Task Mapping and Scheduling for Collaborative In-network Processing in DVS-enabled Wireless Sensor Networks,” in Proc. IPDPS, 2006.

[7] L. Fok, G. Roman, and C. Lu, “Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications,” in Proc. ICDCS 2005.

[8] A. Boulis, C.-C. Han, R. Shea, and M.B. Srivastava, “SensorWare: Programming sensor networks beyond code update and querying,” Pervasive and Mobile Computing Journal, 3(4), 2007, Elsevier.

[9] B. Chen, H.H. Cheng, J. Palen, “Mobile-C: A Mobile Agent Platform for Mobile C-C++ agents,”in Journal Software Practice and Experience, vol. 36(15), pp. 1711-1733, 2006.

[10] H. Liu, T. Roeder, K. Walsh, R. Barr, E.G. Sirer, “Design and Implementation of a Single System Image Operating System for Ad Hoc Networks”, in ProcMOBISYS, 2005.

[11] N. Kothari, R. Gummadi, T. Millstein, R. Govindan, “Reliable and Efficient Programming Abstractions for Wireless Sensor Networks”, ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007.

[12] U. Ramachandran, R. Kumar, M. Wolenetz, B. Cooper, B. Agarwalla, J. Shin, P. Hutto, A. Paul, “Dynamic Data Fusion for Future Sensor Networks,” ACM Transactions on Sensor Networks, vol. 2 (3), pp. 404-443, 2006.

[13] J. Domaszewicz, M. Roj, A. Pruszkowski, M. Golanski and K. Kacperski, “ROVERS: Pervasive Computing Platform for Heterogeneous Sensor-ActuatoNetworks,” in Proc. WoWMoM, 2006.

[14] H.S. Kim, T.F. Abdelzaher and W.H. Kwon, “Dynamic Delay-Constrained Minimum-Energy Dissemination in Wireless Sensor Networks,” in ACM Trans. on Embedded Computing Systems, vol. 4 (3), pp. 679-706, 2005.

[15] Q. Wu, N. S. V. Rao, J. Barhen, et al., “On Computing Mobile Agent Routes for Data Fusion in

Distributed Sensor Networks," in IEEE Transactions on Knowledge and Data Engineering, vol. 16 (6), pp. 740–753, 2004.

[16] Kruskal's Algorithm, https://en.wikipedia.org/wiki/Kruskal's_algorithm

[17] Software Agent, https://en.wikipedia.org/wiki/Software_agent

[18] What Is Software Agent?, http://whatis.techtarget.com/definition/software-agent

[19] Mobile agent, https://en.wikipedia.org/wiki/Mobile_agent

[20] Wireless sensor network, https://en.wikipedia.org/wiki/Wireless_sensor_network

ΠΑΡΑΡΤΗΜΑ

Application Tree Generator

File: appTreeGeneratot.cpp

```
#include<time.h>        //definition of time
#include<iostream>
#include<stdlib.h>      //definition of rand, srand, malloc,free,atoi
#include<stddef.h>      //definition of NULL
#include<fstream>       //file handling functions
#include<string>
using namespace std;

int MAX_AGENTS;

typedef struct tNode *treeNode; //forward decleration
treeNode* addressTable;

//*****DYNAMIC ARRAY CLASS*************************
typedef struct lNode{
  int ID;  // treeNode address;
  struct lNode * next;
} * listNode;

class list{

    listNode head;
    int length;
  public:
    list();     //constructor
    int getLength(){return length;};
    void addListNode(int tNodeAddress);
    int getNodeID(int position);
    int removeNode(int position);
    void printList();
    string listToString();
    void init();
};

list::list(){
  head=NULL;
  length=0;
}
void list::init(){
  head=NULL;
  length=0;
}

void list::printList(){
  listNode temp=head;
  if(temp==NULL)
    cout<<"empty list!\n";
  while(temp!=NULL){
    cout<<temp->ID<<"|";
    temp=temp->next;
  }
  cout<<"\n";
}

string list::listToString(){
  listNode temp=head;
  string returnString="";
  char tempString[5];
```

```
  if(temp!=NULL){
    while(temp!=NULL){
      sprintf(tempString,"%d",temp->ID);
      returnString.append(tempString);
      returnString+="|";
      temp=temp->next;
    }
  }
 return(returnString);
}

void list::addListNode(int tNodeID){
/* insert node at the top of the list */
  listNode temp;
  //-----create new node to insert--------
  temp=(listNode)malloc(sizeof(lNode));
  temp->ID=tNodeID;
  //-----insert----------
  temp->next=head;
  head=temp;
  length++;
}

int list::getNodeID(int position){
  if(position>=length)
    return -1;
  else{
    listNode temp=head;
    for(int i=0;i<position;i++){
      temp=temp->next;
    }
    return temp->ID;
  }
}

int list::removeNode(int position){
/* removes from list node in position 'position' and decreases by 1 the value of length
   returns the node->ID if removal was succesfull. else returns -1
 */
  if(position>=length)
    return -1;
  else{
    int nodeID;
    if(length==1){
      nodeID=head->ID;
      free(head);
      head=NULL;
    }else{
      listNode current=head;
      listNode previous=head;
      for(int i=0;i<position-1;i++){
        previous=previous->next;
      }
      if(position==0){
        current=previous;
        head=current->next;
      }
      else
        current=previous->next;
      nodeID=current->ID;
      previous->next=current->next;
      free(current);
    }
    length--;
    return nodeID;
  }
}

//********TREE CLASS***********
typedef struct tNode{
  int ID;
```

```
  int size;
  list childrenList;
} * treeNode;

class tree{
  treeNode root;
  public:
    tree();     //constructor
    void printTree();
    void printTreeToFile();
};

tree::tree(){
 list buffer;           //list of nodes to be grouped as children
 int i,random,randomTimes,offset;
 treeNode node;
 int idCounter=1;
 int limit;

//========initialize hash table=================
    offset=2*MAX_AGENTS-1;              //==worst case scenario
    addressTable=new treeNode[offset];  //initialize addressTable
    for(i=0;i<offset;i++)               //empty addressTable
      addressTable[i]=NULL;
//==============================================

 buffer.init();
  for(i=0;i<MAX_AGENTS;i++){                    //create first MAX_AGENTS children
    node=(treeNode)malloc(sizeof(tNode));
    node->size=rand()%1001+100;
    node->ID=idCounter;
    node->childrenList.init();
    addressTable[idCounter-1]=node;
    buffer.addListNode(node->ID);
    idCounter++;
  }

  do{                                           //division in groups of 5 until one root parent is
created
    offset=0;
    while(offset<buffer.getLength()){
     if(buffer.getLength()-offset<=2){  //if there are only two or one nodes left,
        node=(treeNode)malloc(sizeof(tNode));   //create a parent node
        node->size=rand()%1001+100;
        node->ID=idCounter;
        node->childrenList.init();
        addressTable[idCounter-1]=node;
        idCounter++;
        while(offset<buffer.getLength()){               //and add them (in both cases) to parent's
children list
          node->childrenList.addListNode(buffer.removeNode(offset));
        }
        buffer.addListNode(node->ID);   //add newly created parent node to buffer list
        offset++;                       //exit from outer while loop
      }else{
        if((buffer.getLength()-offset)<5)       //if there are 3 or 4 nodes left
          limit=buffer.getLength()-offset;              //form a group of childer with maximum
(limit) 3 or 4 members
        else
          limit=5;
        node=(treeNode)malloc(sizeof(tNode));   //create a parent node
        node->size=rand()%1001+100;                     //assign random value size of agent
        node->childrenList.init();
        addressTable[idCounter-1]=node;
        node->ID=idCounter;                             //assign an incrementing ID number;
        idCounter++;

        randomTimes=rand()%(limit-1)+2; //random value in [2,limit-1] = number of nodes to add as
children
        for(i=0;i<randomTimes;i++){
          random=rand()%(limit-i)+offset;               //random value in [offset,offset+(5-i)]
          node->childrenList.addListNode(buffer.removeNode(random));    //get info of buffer node
at position 'random' and then remove it from buffer
        }
```

```
        offset=offset+(limit-randomTimes);      //since we removed 'randomTimes' nodes (out of the
first group of 5) from buffer, offset must point to the beggining of the next group of 5 nodes in
buffer
        buffer.addListNode(node->ID);   //add newly created parent node to buffer list
        offset++;
       }
     }
   }while(buffer.getLength()>1);
   root=node;
   root->size=rand()%1001+100;
   root->ID=idCounter-1;

   MAX_AGENTS=idCounter;
}


void tree::printTree(){
 int i=0;
 cout<<"root ID="<<root->ID<<"\n";
  while(addressTable[i]!=NULL){
    cout<<i+1<<"="<<addressTable[i]<<":";
    addressTable[i]->childrenList.printList();
    i++;
  }
}

void tree::printTreeToFile(){
  int i=0;
  ofstream myfile;
  myfile.open("appTreeFile.txt");

  myfile<<root->ID<<"\n";
  while(addressTable[i]!=NULL){
    myfile<<i+1<<":";
    myfile<<addressTable[i]->childrenList.listToString()<<"\n";
    i++;
  }
  myfile<<"$\n" ;
  myfile.close();
}


int main(int argc,char* args[]){
//args[1]==number of agents
  if(argc<=1){
    cout<<"not enough parameters!!\n";
  }else{
    srand(time(NULL));
    MAX_AGENTS=atoi(args[1]);
    tree t;
    t.printTree();
    t.printTreeToFile();
  }

//system("Pause");

  return 0;
}
```

Network Generator

File: networkGenerator..cpp

```
#include<time.h>      //definition of time
#include<stdlib.h>      //definition of rand, srand, malloc,free,atoi
#include<stddef.h>      //definition of NULL
#include<math.h>        //sqrt function
#include<iostream>      //definition of cout
```

```
#include<iomanip>       //setw()
#include<fstream>       //file handling functions
#include<string>
using namespace std;

#define endLine
cout<<"------------------------------------------------------------------------------------------ \n"

//**********decleration of arguments*****************
int PLANE_MAX;          //=80
int MAX_NODES;          // =20
int e_distance;         //euclidean distance    =30
string filename="";

//******dynamic Array definition**********************
typedef struct dynamicArrayNode{//dynamic list containing connecting nodes
  int ID;
  int distance;
  struct dynamicArrayNode* next;
} *daNode;

class dynamicArray{     //list
    daNode head;          //head of list
    int length;           //size of list
  public:
    dynamicArray();
    void addEntry(int nodeID,int d);
    void rmEntry(int nodeID);
    void printArray();
    int getNodeID(int pos);             //returns ID of node in 'pos's position of the list
    int getNodeDistance(int pos);       //returns distance of node in 'pos's position of the list
    int getLength(){return length;};
    void init();
};

dynamicArray::dynamicArray(){
    head=NULL;
    length=0;
}

void dynamicArray::init(){
  head=NULL;
  length=0;
}

void dynamicArray::addEntry(int nodeID,int d){
  daNode temp=head;
  daNode newNode;
 //-----initialize new node---------------
  newNode=(daNode)malloc(sizeof(struct dynamicArrayNode));
  newNode->ID=nodeID;
  newNode->distance=d;
  newNode->next=NULL;
 //-----add new node to the list-------
  if(temp==NULL){
    head=newNode;
  }else{
    while(temp->next!=NULL)     //move to the end of the list
      temp=temp->next;
    temp->next=newNode;         //ad ne node
  }
  length++;
}

void dynamicArray::rmEntry(int nodeID){
  daNode previous;
  daNode current;
  int flag=0;
  if(head!=NULL){
    if(head->ID==nodeID){
      head=head->next;
      flag=1;
    }else{
      current=head->next;
      previous=head;
```

```
      while(current!=NULL && flag==0){
        if(current->ID==nodeID){
          previous->next=current->next;
          free(current);
          current=NULL;
          flag=1;
        }else{
          current=current->next;
          previous=previous->next;
        }
      }
    }
    if(flag==1)
      length--;
  }
}

void dynamicArray::printArray(){
  daNode temp=head;
  while(temp!=NULL){
    cout << temp->ID<< " distance= "<<temp->distance<<" | ";
    temp=temp->next;
  }
  cout << '\n';
}

int dynamicArray::getNodeID(int pos){
  daNode temp=head;
  int i=0;
  while(i<pos && temp!=NULL){
    temp=temp->next;
    i++;
  }
  if(temp==NULL)
    return -1;
  else
    return temp->ID;
}


int dynamicArray::getNodeDistance(int pos){
  daNode temp=head;
  int i=0;
  while(i<pos && temp!=NULL){
    temp=temp->next;
    i++;
  }
  if(temp==NULL)
    return -1;
  else
    return temp->distance;
}

//************node definition*************************
typedef struct NODE{
  //---agent space-----
  int agent;
  int nodeID;
  int x;
  int y;
  //--pointers to other nodes (maximum MAX_NODES)--
  dynamicArray connectingNodesList;
}* node;

//********node network definition**********************
class node_network{
    node *nodeAdresses;
  public:
    node_network();     //constructs a random node network
    void printNodeDetails();
    void createMinimumSpanningTree();
```

```
};

node_network::node_network(){
 int plane[PLANE_MAX][PLANE_MAX];
 int i,j,m,n,k,limit,result;
 int distance;
 node temp;
 nodeAdresses=(node *)malloc(MAX_NODES*sizeof(node));
//------fills plane with zeros---------------------------
  for(i=0;i<PLANE_MAX;i++)
    for(j=0;j<PLANE_MAX;j++)
      plane[i][j]=0;

 //-----fills random positions of the aray with numbers from 1-MAX_NODES -----
  srand(time(NULL));                            //initialize rand function
//    srand(2);
  for(i=1;i<=MAX_NODES;i++){
    do{
      m=(rand()%100)%(PLANE_MAX-1);                     //get randmom position on the table
      n=(rand()%100)%(PLANE_MAX-1);
    }while(plane[m][n]!=0);                     //if random position is already occupied, get new
random position
    plane[m][n]=i;                                //fill random position with numbers from 1 to
MAX_NODES
  }
/*/--------------prints graph----------------------------
  for(i=0;i<PLANE_MAX;i++){
    for(j=0;j<PLANE_MAX;j++)
      cout<<plane[i][j]<<" ";
    cout<<'\n';
  }
//----------------------------------------  */
 //create node network
  for(m=0;m<PLANE_MAX;m++){                     //for every row
    for(n=0;n<PLANE_MAX;n++){                   //and every column of plane
      if(plane[m][n]!=0){                         //if it is currently hosting a node
        temp=(node)malloc(sizeof(struct NODE));     //create a new node
        nodeAdresses[(plane[m][n]-1)]=temp;         //save it's adress in the addresses Array
        temp->nodeID=plane[m][n];                   //define its ID. e.g. node 2
        temp->x=m;
        temp->y=n;
        temp->connectingNodesList.init();

      //---find connecting nodes-----------
        //search only the nodes that are located not further from the euclidean distance
        i=m;
        k=0;
        distance=0;
        do{
          result=(int)sqrt((e_distance-1)*(e_distance-1)-(k*k));
          limit=(n+result);
          if(limit>=PLANE_MAX){         //ensures that limit won't exceed PLANE_MAX (end of array)
            j=PLANE_MAX-1;
          }else
            j=limit;
          while(j>=(n-result) && j>=0){
            if(plane[i][j]!=0 && ((i==m && j==n)==false)){                      //if there exists
nodes in distance lower than euclidean distance [exept from itself (i,j)==(m,n)]
              distance=(int)sqrt((j-n)*(j-n)+(i-m)*(i-m));
              temp->connectingNodesList.addEntry(plane[i][j],distance); //add to connecting nodes
list
            }
            j--;
          }
          i--;
          k++;
        }while(i>=0 && k<e_distance);
        i=m+1;
        if(i<PLANE_MAX){
          k=1;
          do{
            result=(int)sqrt((e_distance-1)*(e_distance-1)-(k*k));
            limit=(n+result);
            if(limit>=PLANE_MAX)                //ensures that limit won't exceed PLANE_MAX (end of
array)
```

```
            j=PLANE_MAX-1;
          else
            j=limit;
          while(j>=(n-result) && j>=0){
            if(plane[i][j]!=0){                        //if there exists nodes in distance lower
than euclidean distance
              distance=(int)sqrt((j-n)*(j-n)+(i-m)*(i-m));
              temp->connectingNodesList.addEntry(plane[i][j],distance);       //add to connecting
nodes list
            }
            j--;
          }
          i++;
          k++;
        }while(i<PLANE_MAX && k<=e_distance);
      }
    }
  }
}

void node_network::printNodeDetails(){
int i=0;
  for(i=0;i<MAX_NODES;i++){
    cout << "node "<<i+1<<" ("<< nodeAdresses[i]->x<<","<<nodeAdresses[i]->y<<")"<<":\t";
    nodeAdresses[i]->connectingNodesList.printArray();
    endLine;
  }
}

void node_network::createMinimumSpanningTree(){
int i,j,k,l,tree,temp,temp2,counter=0;
  for(i=0;i<MAX_NODES;i++){
    for(j=0;j<nodeAdresses[i]->connectingNodesList.getLength();j++){
      temp=nodeAdresses[i]->connectingNodesList.getNodeID(j);   //find connected vertexes. E.g.
vertex ID 20
      nodeAdresses[temp-1]->connectingNodesList.rmEntry(i+1);   //remove dublicates
     }
    counter=counter+nodeAdresses[i]->connectingNodesList.getLength();
  }

  int connectionTable[4][counter];

  counter=0;
  for(i=0;i<MAX_NODES;i++){
    if(nodeAdresses[i]->connectingNodesList.getLength()>0)
    for(j=0;j<nodeAdresses[i]->connectingNodesList.getLength();j++){
      connectionTable[0][counter]=i+1;
      connectionTable[1][counter]=nodeAdresses[i]->connectingNodesList.getNodeID(j);
      connectionTable[2][counter]=nodeAdresses[i]->connectingNodesList.getNodeDistance(j);
      connectionTable[3][counter]=0;
      counter++;
    }
  }


//------------bubblesort------------------
  for(i=1;i<counter;i++)
    for(j=(counter-1);j>=i;j--){
      if(connectionTable[2][j]<connectionTable[2][j-1]){
        for(k=0;k<3;k++){
          temp=connectionTable[k][j];
          connectionTable[k][j]=connectionTable[k][j-1];
          connectionTable[k][j-1]=temp;
        }
      }
    }


    tree=1;
     for(i=0;i<counter;i++){
```

```
      //----check if a circle is formed------

        temp=0;
        temp2=0;
        j=0;
        while(j<i && (temp==0 || temp2==0)){    //(temp==0  temp2==0) while no circle is formed
      if(temp==0&&(connectionTable[0][i]==connectionTable[0][j] || connectionTable[0]
[i]==connectionTable[1][j]))
                temp=connectionTable[3][j];
      if(temp2==0&&(connectionTable[1][i]==connectionTable[0][j] || connectionTable[1]
[i]==connectionTable[1][j]))
                temp2=connectionTable[3][j];
          j++;
        }
    if(temp==0 && temp2==0){    //if this pair is not connected with any other tree
          connectionTable[3][i]=tree; //create new tree and add it
          tree++;
   }else if((temp==0 || temp2==0)&&(temp!=0 || temp2!=0)){      //(one==0, one!=0) else if one node
of this pair is part of an existing tree
         if(temp!=0)    //if temp is the one !=0. temp shows the tree the connected pair belongs to
                        connectionTable[3][i]=temp;     //add this pair in 'temp' tree
                    else
//else if temp2 is the one !=0
                        connectionTable[3][i]=temp2;    //add this pair in 'temp2' tree
   }else if(temp!=temp2 && (temp!=0 && temp2!=0)){              //temp!=temp2 && temp!=0 && temp2!
=0. if both nodes of this pair are parts of different existing trees
                    if(temp<temp2){ //if temp is the smallest value
                        connectionTable[3][i]=temp;     //add this pair to the smallest
tree
                        for(k=0;k<i;k++){               //for each pair up to j
                            if(connectionTable[3][k]==temp2)        //with value
==temp2
                                connectionTable[3][k]=temp;             //add to
the smallest tree
                        }
                    }else   if(temp>temp2){ //if temp2 is the smallest value
                        connectionTable[3][i]=temp2;    //add this pair to the smallest
tree
                        for(k=0;k<i;k++){               //for each pair up to j
                            if(connectionTable[3][k]==temp) //with value ==temp
                                connectionTable[3][k]=temp2;            //add to
the smallest tree
                        }
                    }
        }
/*     //-----------print connection table----------------
  if(connectionTable[0][i]==4 && connectionTable[1][i]==37){
//    if(i==83){
    for(j=0;j<=i;j++){
      for(l=0;l<=3;l++)
         cout<<setw(2)<<connectionTable[l][j]<<"|";
      cout<<'\n';
    }
    cout<<i<<"!!!\n";
  }*/
 }


//------------bubblesort-----------------
  for(i=1;i<counter;i++)
    for(j=(counter-1);j>=i;j--){
      if(connectionTable[0][j]<connectionTable[0][j-1]){
        for(k=0;k<=3;k++){
          temp=connectionTable[k][j];
          connectionTable[k][j]=connectionTable[k][j-1];
          connectionTable[k][j-1]=temp;
        }
      }
    }

    ofstream myfile;
    myfile.open(filename.data());
    tree=1;
    k=1;
    while(k!=0){
```

```
      k=0;
      for(i=0;i<counter;i++)
        if(connectionTable[3][i]==tree){
          myfile << connectionTable[0][i]<<"-"<<connectionTable[1][i]<<":"<<connectionTable[2]
[i]<<"\n";
          k++;
        }
      if(k!=0)
        myfile<<"#"<<"\n";
      tree++;
    }
        myfile<<"$"<<"\n";
    myfile.close();

 /* /-----------print connection table----------------
   for(i=0;i<4;i++){
    for(j=0;j<counter;j++)
      cout<<setw(2)<<connectionTable[i][j]<<"|";
    cout<<'\n';
  }
  */

}

int main(int argc,char* args[]){
if(argc<4){
    cout <<"not enough arguments!\n";
}else{
    filename.assign(args[1]);
    cout<<filename<<"\n";
    PLANE_MAX=atoi(args[2]);
    MAX_NODES=atoi(args[3]);
    e_distance=atoi(args[4]);
    cout<<PLANE_MAX<<" "<<MAX_NODES<<" "<<e_distance<<"\n";     //print arguments
    node_network m;
    m.createMinimumSpanningTree();

return 0;
}
//  system("Pause");
}
```

Main Simulation.

Dynamic List Definition. File: dynamicListHeader.h

```
#ifndef WHATEVERA_H_INCLUDED
#define WHATEVERA_H_INCLUDED

#include<iostream>
#include<time.h>        //definition of time
#include<stdlib.h>      //definition of rand, srand, malloc,free,atoi
#include<stddef.h>      //definition of NULL
#include<math.h>        //sqrt function
#include<iostream>      //definition of cout
#include<iomanip>       //setw()
#include<string>

using namespace std;

//*******DYNAMIC ARRAY DEFINITION***************
```

```
typedef struct dynamicListNode{//dynamic list containing connecting nodes
  int ID;
  int distance;
  struct dynamicListNode* next;
} *dlNode;

class dynamicList{      //list
    dlNode head;          //head of list
    int length;           //size of list
  public:
    dynamicList();
    void init();
    void addEntry(int nodeID,int d);
    int addEntryInOrder(int nodeID,int d);      //in increasing order of distance values.return
position:starting position==0
    void addEntryAt(int nodeID,int d,int pos);
    void rmEntry(int nodeID);
    void rmEntryFrom(int pos);
    void printArray(int mode);
    int getNodeID(int pos);             //returns ID of node in 'pos' position of the list
    int getLength(){return length;};
    int getNodeDistance(int pos);       //returns distance of node in 'pos' position of the list
    int getNodeDistance2(int nodeID);   //returns distance of node with ID =nodeID
    void setDistance(int pos,int d);
    int contains(int ID);
    int contains2(int ID,int d);
    int contains3(int d);
};
dynamicList::dynamicList(){
    head=NULL;
    length=0;
}

void dynamicList::init(){
    head=NULL;
    length=0;
}

void dynamicList::addEntryAt(int nodeID,int d,int pos){
//strarting position 0
int i=0;
dlNode current=head,previous=head;
dlNode temp;
  if(head!=NULL && pos>=0){
    while(i<pos && current->next!=NULL){
      if(current!=head)
                                previous=previous->next;
      current=current->next;
      i++;
    }
    if(pos==0){
      temp=(dlNode)malloc(sizeof(struct dynamicListNode));
      temp->ID=nodeID;
      temp->distance=d;
      temp->next=current;
      head=temp;
      length++;
    }else if(i==pos || pos==length-1){
      temp=(dlNode)malloc(sizeof(struct dynamicListNode));
      temp->ID=nodeID;
      temp->distance=d;
      temp->next=current;
      previous->next=temp;
      length++;
    }else if(length==pos){      //add it at the end of the list//20/11/2015
      temp=(dlNode)malloc(sizeof(struct dynamicListNode));
      temp->ID=nodeID;
      temp->distance=d;
                        temp->next=NULL;
                        while(current->next!=NULL)
                                current=current->next;
                        current->next=temp;
                        length++;
                }
  }else if(head==NULL && pos==0){
```

```
    head=(dlNode)malloc(sizeof(struct dynamicListNode));
    head->ID=nodeID;
    head->distance=d;
    head->next=NULL;
    length++;
  }
}

void dynamicList::addEntry(int nodeID,int d){
  dlNode temp=head;
  dlNode newNode;
 //-----initialize new node---------------
  newNode=(dlNode)malloc(sizeof(struct dynamicListNode));
  newNode->ID=nodeID;
  newNode->distance=d;
  newNode->next=NULL;
 //-----add new node to the list-------
  if(temp==NULL){
    head=newNode;
  }else{
    while(temp->next!=NULL)     //move to the end of the list
      temp=temp->next;
    temp->next=newNode;         //add new node
  }
  length++;                     //increment length
}

int dynamicList::addEntryInOrder(int nodeID,int d){
  dlNode previous=head,current=head;
  dlNode newNode;
  int i=0;
 //-----initialize new node---------------
  newNode=(dlNode)malloc(sizeof(struct dynamicListNode));
  newNode->ID=nodeID;
  newNode->distance=d;
  newNode->next=NULL;
 //-----add new node to the list-------
  if(head==NULL){
    head=newNode;
    length++;                   //increment length
    return(i);
  }else{
    while(current->next!=NULL && current->distance<=d){
      if(current!=head)
        previous=previous->next;
      current=current->next;
      i++;
    }
    if(current==head){  //if newValue is the smallest
      newNode->next=head;       //add it at the beginning of the list
      head=newNode;
    }else if(current->distance>=d){     //else if a position somewhere in between the list was found
      newNode->next=current;            //place it between current and previous
      previous->next=newNode;
    }else{                                                  //else if newValue is the greatest
      current->next=newNode;            //add it at the end of the list (case current->next==NULL)
      i++;
    }
    length++;                   //increment length
    return(i);
  }
}

void dynamicList::rmEntry(int nodeID){
  dlNode previous;
  dlNode current;
  int flag=0;
  if(head!=NULL){
    if(head->ID==nodeID){
      head=head->next;
      flag=1;
```

```
    }else{
      current=head->next;
      previous=head;
      while(current!=NULL && flag==0){
        if(current->ID==nodeID){
          previous->next=current->next;
          free(current);
          current=NULL;
          flag=1;
        }else{
          current=current->next;
          previous=previous->next;
        }
      }
    }
    if(flag==1)
      length--;
  }
}

void dynamicList::rmEntryFrom(int pos){
//starting pos=0
  int i;
  dlNode current,previous;
  if(head!=NULL){
    current=head;
    previous=head;
    i=0;
    if(pos==0){
      head=head->next;
      length--;
    }else{
      while(current!=NULL && i<pos){
        if(current!=head)
          previous=previous->next;
        current=current->next;
        i++;
      }
      if(i==pos){       //if found
        previous->next=current->next;
        length--;
      }
    }
  }
}

void dynamicList::printArray(int mode){
//if mode ==1 print ID and distance
//if mode ==0 only print ID
  dlNode temp=head;
  while(temp!=NULL){
    if(mode==1)
      cout << temp->ID<< " distance= "<<temp->distance<<" | ";
    else
      cout <<temp->ID<<", ";
    temp=temp->next;
  }
  cout << '\n';
}

int dynamicList::getNodeID(int pos){
//from pos 0 to length-1
  dlNode temp=head;
  int i=0;
  while(i<pos && temp!=NULL){
    temp=temp->next;
    i++;
  }
  if(temp==NULL)
    return -1;
  else
    return temp->ID;
}

int dynamicList::getNodeDistance(int pos){
```

```
  dlNode temp=head;
  int i=0;
  while(i<pos && temp!=NULL){
    temp=temp->next;
    i++;
  }
  if(temp==NULL)
    return -1;
  else
    return temp->distance;
}

int dynamicList::getNodeDistance2(int nodeID){
//returns 0 if nodeID not in list, else returns distance
  dlNode temp=head;
  int result=0;
  if(temp==NULL){
    return(0);
  }else{
    result=temp->ID;
    while(temp!=NULL && result!=nodeID){
      temp=temp->next;
      if(temp!=NULL)
        result=temp->ID;
    }
    if(result==nodeID)
      return(temp->distance);
    else
      return(0);
  }
}

void dynamicList::setDistance(int pos,int d){
  dlNode temp=head;
  int i=0;
  while(i<pos && temp!=NULL){
    temp=temp->next;
    i++;
  }
  if(temp==NULL)
    cout<<"limits exceeded!\n";
  else
    temp->distance=d;
}

int dynamicList::contains(int ID){
//returns position[starting position=position 1] in list if ID is contained, else returns -1
  dlNode temp=head;
  int result=-1;
  int counter=0;


  while(temp!=NULL && result!=1 && counter<length){
    if(temp->ID==ID)
      result=1;
    counter++;
    temp=temp->next;
  }
  if(result==1)
    return(counter);
  else
    return(-1);
}

int dynamicList::contains2(int ID,int d){
//returns position[starting position=position 0] in list if ID is contained, else returns -1
  dlNode temp=head;
  int result=-1;
  int counter=0;
  while(temp!=NULL && result!=1 && counter<length){
    if(temp->ID==ID && temp->distance==d)
```

```
      result=1;
    else{
      counter++;
      temp=temp->next;
    }
  }
  if(result==1 && counter<=length-1)    //21/11/2015
    return(counter);
  else
    return(-1);

}

int dynamicList::contains3(int d){
//returns position[starting position=position 0] in list if distance is contained, else returns -1
  dlNode temp=head;
  int result=-1;
  int counter=0;
  while(temp!=NULL && result!=1 && counter<length){
    if(temp->distance==d)
      result=1;
                else
                        counter++;
    temp=temp->next;
  }
  if(result==1)
    return(counter);
  else
    return(-1);
}

#endif
```

Application Tree Class and Relative Functions. File: appTree.h

```
#include<iostream>
#include</dynamicListHeader.h>
#include<time.h>        //definition of time
#include<stdlib.h>      //definition of rand, srand, malloc,free,atoi
#include<stddef.h>      //definition of NULL
#include<math.h>        //sqrt function
#include<iostream>      //definition of cout
#include<iomanip>       //setw()
#include<string>
#include<fstream>       //file handling functions
#ifndef WHATEVER_H_INCLUDED


//==========DEFINITIONS====================
#define appTreeFREE 0
#define appTreeBUSY 1
#define appTreeEVICTION 2

//=========================================

#define WHATEVER_H_INCLUDED


using namespace std;

//*********APP TREE CLASS***************
```

```
typedef struct APPLICATION_TREE_NODE{
  int ID;
  int size;
  int parentID;
  dynamicList childrenList;
  int state; //FREE, BUSY, EVICTION
  int nodesChecked;
  int maxBenefit;
  int mostBeneficialNode;
  int agentBeingChecked;
  int sum;
  int totalBenefit;
  int benefit;
  int migrations;
        int isRootAgent;        //0 no, 1 yes
        int restorationFlag;
        int parentAgent;
  dynamicList evictionsChecked;
  dynamicList tempEvictions;
  dynamicList benefitsArray;
  dynamicList evictionPath;
  dynamicList previousNodes;
  dynamicList agentsOccupied;
        dynamicList migrationsAttempted;
        dynamicList attemptCounter;
        dynamicList nodesToRestore;
}* appTreeNode;


class appTree{
    int rootID;
    appTreeNode* appTreeAddressTable;
    int numberOfElements;
  public:
    appTree();
    void printAddressTableTree();
    int setApplicationTraffic(appTreeNode root);
    int getNumberOfElements(){return numberOfElements;};
    appTreeNode getAgentAddress(int ID);
    int getParentID(int agentID){return(appTreeAddressTable[agentID-1]->parentID);};
                void printAgentValues(int agentID);
};

void appTree::printAgentValues(int agentID){
        agentID--;
        cout<<"=============================================\n";
        cout<<"For agent "<<appTreeAddressTable[agentID]->ID<<":\n";
        cout<<"size="<<appTreeAddressTable[agentID]->size<<"\n";
        if(appTreeAddressTable[agentID]->state==appTreeFREE)
                cout<<"state=FREE\n";
        if(appTreeAddressTable[agentID]->state==appTreeBUSY)
                cout<<"state=BUSY\n";
        if(appTreeAddressTable[agentID]->state==appTreeEVICTION)
                cout<<"state=EVITION";
        cout<<"nodesChecked="<<appTreeAddressTable[agentID]->nodesChecked<<"\n";
  cout<<"maxBenefit="<<appTreeAddressTable[agentID]->maxBenefit<<"\n";
  cout<<"mostBeneficialNode="<<appTreeAddressTable[agentID]->mostBeneficialNode<<"\n";
  cout<<"agentBeingChecked="<<appTreeAddressTable[agentID]->agentBeingChecked<<"\n";
  cout<<"sum="<<appTreeAddressTable[agentID]->sum<<"\n";
  cout<<"totalBenefit="<<appTreeAddressTable[agentID]->totalBenefit<<"\n";
  cout<<"benefit="<<appTreeAddressTable[agentID]->benefit<<"\n";
//  int migrations;
  cout<<"evictionsChecked length="<<appTreeAddressTable[agentID]-
>evictionsChecked.getLength()<<"\n";
  cout<<"tempEvictions length="<<appTreeAddressTable[agentID]->tempEvictions.getLength()<<"\n";
  cout<<"benefitsArray length="<<appTreeAddressTable[agentID]->benefitsArray.getLength()<<"\n";
  cout<<"evictionPath length="<<appTreeAddressTable[agentID]->evictionPath.getLength()<<"\n";
  cout<<"previousNodes length="<<appTreeAddressTable[agentID]->previousNodes.getLength()<<"\n";
  cout<<"agentsOccupied length="<<appTreeAddressTable[agentID]->agentsOccupied.getLength()<<"\n";
        cout<<"migrationsAttempted length="<<appTreeAddressTable[agentID]-
>migrationsAttempted.getLength()<<"\n";
```

```
        cout<<"attemptCounter length="<<appTreeAddressTable[agentID]-
>attemptCounter.getLength()<<"\n";
        cout<<"================================================\n";
}

//======define application traffic========
int appTree::setApplicationTraffic(appTreeNode root){
  int i,length,sum,traffic,CH;
  sum=0;
  length=root->childrenList.getLength();
  if(length==0){
    return(rand()%5+1);
  }else{
    for(i=0;i<length;i++){      //for each child
      CH=root->childrenList.getNodeID(i);               //get child ID
      traffic=setApplicationTraffic(appTreeAddressTable[CH-1]);
      root->childrenList.setDistance(i,traffic);//define traffic between current node and ith child
      sum+=traffic;
    }
    return(sum);
  }
}

//=============constructor APP TREE CLASS================
appTree::appTree(){
 ifstream myfile("appTreeFile.txt");
 string line,temp;
 int i,j,pos,X,CH;
srand(1);
  getline(myfile,line);
  rootID=atoi(line.c_str());
  numberOfElements=rootID;
  appTreeAddressTable=new appTreeNode[numberOfElements];


  while(getline(myfile,line) && line!="$"){
  //====line consists of X:CH1|CH2...=======

    //get X
    pos=(int)line.find(":");
    temp=line.substr(0,pos);
    X=atoi(temp.c_str());

    //initialize appTreeNode
    appTreeAddressTable[X-1]=(appTreeNode)malloc(sizeof(struct APPLICATION_TREE_NODE));
    appTreeAddressTable[X-1]->ID=X;
    appTreeAddressTable[X-1]->parentID=0;
    appTreeAddressTable[X-1]->migrations=0;
    appTreeAddressTable[X-1]->size=rand()%901+100;      //set random size from 100-1000
    appTreeAddressTable[X-1]->childrenList.init();

    pos=(int)line.find(":");
    line=line.substr(pos+1,line.length());
    pos=(int)line.find("|");    //retreive data until |
    while(pos!=-1){     //extract data from children list ch1|ch2|.....
      temp=line.substr(0,pos);
      CH=atoi(temp.c_str());    //convert it to int
      if(temp!=""){             //if children exist
        appTreeAddressTable[X-1]->childrenList.addEntry(CH,0);  //add them to children list
      }
      line=line.substr(pos+1,line.length());    //get line from | and later
      pos=(int)line.find("|");  //retreiva data until |
    }
 }

  //======intialize parentIDs===============
  for(i=0;i<rootID;i++){//for each agent
    appTreeAddressTable[i]->state=appTreeFREE;  //also initialize it's state
    for(j=0;j<appTreeAddressTable[i]->childrenList.getLength();j++){    //for each of his children
      appTreeAddressTable[appTreeAddressTable[i]->childrenList.getNodeID(j)-1]->parentID=i+1;
    }
  }
  //======define application traffic========
  setApplicationTraffic(appTreeAddressTable[rootID-1]);
 myfile.close();
```

```
}

//=============print app tree function===============
void appTree::printAddressTableTree(){
 int i=0;
 cout<<"root ID="<<numberOfElements<<"\n";
  while(i<numberOfElements){
    cout<<i+1<<"-"<<appTreeAddressTable[i]->parentID<<". size="<<appTreeAddressTable[i]->size<<":";
    appTreeAddressTable[i]->childrenList.printArray(1);
                cout<<"\n";
    i++;
  }
}

appTreeNode appTree::getAgentAddress(int pos){
  return(appTreeAddressTable[pos]);
};

#endif
```

Network Node Class and Relative Functions for Network Flooding, SAM algorithm. File: networkNodeHeader.h

```
#include</appTree.h>
#include<time.h>        //definition of time
#include<stdlib.h>      //definition of rand, srand, malloc,free,atoi
#include<stddef.h>      //definition of NULL
#include<math.h>        //sqrt function
#include<iostream>      //definition of cout
#include<iomanip>       //setw()
#include<string>
#include<fstream>       //file handling functions


#ifndef WHAT_H_INCLUDED

//===========DEFINITIONS====================
#define MAX_NODES 20
#define ENDLINE "================================================\n"
#define TESTLINE "test============================================\n"
#define INVALID -1
#define RUNTIME 100000

#define AVG_SIZE_MULTIPLE 3

//------agent state definitions-------
#define FREE 0
#define BUSY 1
#define EVICTION 2

//----node mode definitions-----------
#define ASLEEP -1
#define AWAKE 0
#define BUSY 1
#define MIGRATING 2
#define EVICTING 3 //node is available for those agents in the node hosting agentBeingServed
#define PRE_EVICTING 4
#define PRE_ASLEEP 5

//----Hip commands definitions-----------
#define SEND_HOST_REQUEST 0
#define MIGRATE 1
#define MOVE_AGENT 2
#define ANSWER 3
#define EVICT_MIGRATE 4
#define SETMODE 5
```

```
#define SET_STATE 6
#define NEXT_MODE 7
#define RESTORATION 8
#define SET_AB_SERVED 9
//=========================================

#define WHAT_H_INCLUDED

using namespace std;

string fileName="testFile";

appTree a;

//*******NETWORK NODE DEFINITION*****************
typedef struct NETWORK_NODE{
  int ID;
  int size;
  int remainingSpace;
  int mode;
  int agentsChecked;
  int agentBeingServed;
        int nodeBeingServed;
        int isMostBeneficialFor;
        dynamicList reservations;
  dynamicList benefitsArray;
  dynamicList agentList;
  dynamicList childrenList;
        dynamicList agentsYetToCheck;
        dynamicList nodesOccupied;
}* netNode;

class network{
    netNode addressTable[MAX_NODES];    //hash array of nodes addresses
    int outputValue;
  public:
    network();
                void initializeAgent(int agentID);
                void initializeNode(int nodeID);
    void setupNetwork();
    void printNetwork();
    void simulation();
    int getNumberOfHops(int startID,int parentID,int destinationID,int result);
    int getLoad(int agentID,int nodeID);
    int calculateRemainingSpace(int nodeID);
    void sendHostRequestNFL(int agentID,int nodeID,int benefit,int timeHip);
    void moveAgentToNode(int agentID,int nodeID,int timeHip);
    void notifyNode(int agentID,int nodeID,int benefit,int timeHip);
    void migrationCheck(int agentID,int nodeID,int benefit,int timeHip);
    void checkForEvictions(int agentID,int nodeID,int timeHip);
    int getTotalLoad();
    int getTotalMigrations();
                void printNodeValues(int nodeID);
                void setOutputValue(int value){outputValue=value;};
                void printOutputValue(){cout<<outputValue<<ENDLINE;};
};

network n;

//*********HIP CLASS DEFINITION*****************
typedef struct HIP_NODE{
  int timeHip;  //timeHip
  int command;  //command to execute. SEND_HOST_REQUEST
  dynamicList args;     //
  struct HIP_NODE* next;
}* hipNode;

class hipClass{
    hipNode hip;
  public:
    void init();
    void addToHip(int timeHipValue,int commandValue,dynamicList arguments);
    void executeHipCommands(int time);
    void printHipQueue(int time);
};
```

```
hipClass h;

void hipClass::init(){
  hip=NULL;
}

void hipClass::addToHip(int timeHipValue,int commandValue,dynamicList arguments){
//add command to hip queue in order of timeHip
  hipNode temp,current,previous;
  //=====initialize new hipNode=================
  temp=(hipNode)malloc(sizeof(struct HIP_NODE));
  temp->timeHip=timeHipValue;
  temp->command=commandValue;
  temp->args=arguments;
  temp->next=NULL;
  //============================================

  if(hip==NULL){
    hip=temp;
  }else{
    current=hip;
    previous=hip;
    while(current!=NULL && current->timeHip<timeHipValue ){
      if(current!=hip)
        previous=current;
      current=current->next;
    }
    if(current==hip){ //if it is to be inserted at the beggining of the list
      temp->next=hip;
      hip=temp;
    }else{              //else isert between previous and current
      previous->next=temp;
      temp->next=current;
    }
  }
}

void hipClass::executeHipCommands(int time){
//execute all comands in hip queue that are to be executed in timeHip==time
 hipNode current,previous;
  current=hip;
  previous=hip;
if(current!=NULL){
  //---------reach timehip commands----------------
  while(current->next!=NULL && (current->timeHip < time) ){
    if(current!=hip)
      previous=current;
                current=current->next;
  }
if(current->timeHip>=time){
  //--------execute them--------------------------
  while(current!=NULL && current->timeHip==time){
    if(current!=hip)
      previous=current;

    //==============execute command and remove it from hip queue==============================
    if(current->command==SEND_HOST_REQUEST){ //case 1: command for SEND_HOST_REQUEST
      n.sendHostRequestNFL(current->args.getNodeID(0),current->args.getNodeID(1),current-
>args.getNodeID(2),time);
    }else if(current->command==MIGRATE){        //case 2: command for migration
      n.migrationCheck(current->args.getNodeID(0),current->args.getNodeID(1),current-
>args.getNodeID(2),time);
    }else if(current->command==MOVE_AGENT){     //case 3: command move agent to node
      n.moveAgentToNode(current->args.getNodeID(0),current->args.getNodeID(1),time);
    }else if(current->command==ANSWER){
      n.notifyNode(current->args.getNodeID(0),current->args.getNodeID(1),current-
>args.getNodeID(2),time);
                }

    //-----------------remove it from hip queue---------------------------------
```

```
    if(current==hip){
      hip=hip->next;
      free(current);
      previous=hip;
      current=hip;
    }else{
      current=current->next;
      previous->next=current;
    }
  }
}
}
}


void hipClass::printHipQueue(int time){
  hipNode current;
  if(hip==NULL){
    cout<<"no commands in hip at time: "<<time<<"\n";
  }else{
    current=hip;
    while(current->next!=NULL && current->timeHip < time)
      current=current->next;
    cout<<"time: "<<time<<",commands:\n";
    while(current->next!=NULL && current->timeHip==time){
      cout<<current->command<<" | ";
      current->args.printArray(0);
      current=current->next;
    }
    if(current->timeHip==time){
      cout<<current->command<<" | ";
      current->args.printArray(0);
    }
  }
}

network::network(){
 int i,pos,offset,X,Y,Z,lastEntryID=-1;
 string line,temp;
 ifstream myfile(fileName.c_str());

 for(i=0;i<MAX_NODES;i++)       //initialize addressTable
   addressTable[i]=NULL;
 do{
   while(getline(myfile,line) && line!="#" && line!="$"){       //read results of netork generation
algorith from file 'myfile'
    //====line consists of X-Y:Z=======
    //get X
    pos=(int)line.find("-");
    temp=line.substr(0,pos);
    X=atoi(temp.c_str());
    if(addressTable[X-1]==NULL){                //if a node with this ID was not created
      addressTable[X-1]=(netNode)malloc(sizeof(struct NETWORK_NODE));   //create
      addressTable[X-1]->ID=X;                                          //and initialize it
      addressTable[X-1]->size=0;
      addressTable[X-1]->agentList.init();
      addressTable[X-1]->childrenList.init();
    }

    if(lastEntryID!=-1){                //if there were more than one minimum spanning tree formed
(see Kruskal's algorith)
      addressTable[lastEntryID-1]->childrenList.addEntry(X,10); //add the current spanning tree to
a node of the previous one
      lastEntryID=-1;
    }


    //get Y
    offset=pos+1;
    pos=(int)line.find(":");
    temp=line.substr(offset,pos);
    Y=atoi(temp.c_str());
    if(addressTable[Y-1]==NULL){                //if a node with this ID was not created
      addressTable[Y-1]=(netNode)malloc(sizeof(struct NETWORK_NODE));   //create
      addressTable[Y-1]->ID=Y;                                          //and initialize it
```

```
      addressTable[Y-1]->size=0;
      addressTable[Y-1]->agentList.init();
      addressTable[Y-1]->childrenList.init();
    }

    //get Z
    offset=pos+1;
    pos=(int)line.length();
    temp=line.substr(offset,pos);
    Z=atoi(temp.c_str());

    addressTable[X-1]->childrenList.addEntry(Y,Z);
    addressTable[Y-1]->childrenList.addEntry(X,Z);
    }

    lastEntryID=Y;
 }while(line!="$");

 for(i=0;i<MAX_NODES;i++){
    if(addressTable[i]==NULL){//if there are nodes that are not part of the tree
      addressTable[i]=(netNode)malloc(sizeof(struct NETWORK_NODE));     //create
      addressTable[i]->ID=i+1;                                    //and initialize it
      addressTable[i]->size=0;
      addressTable[i]->agentList.init();
      addressTable[i]->childrenList.init();
      addressTable[lastEntryID-1]->childrenList.addEntry((i+1),10);     //add it to the tree
      addressTable[i]->childrenList.addEntry(lastEntryID,10);
  }
  //addressTable[i]->state=SLEEP;       //put all nodes to sleep
 }

 myfile.close();


/*
 //=========print addressTable===========
 for(i=0;i<MAX_NODES;i++){
   cout<<"ID="<<i+1<<": ";
   if(addressTable[i]!=NULL){ //=======put all nodes to sleep=================
    addressTable[i]->childrenList.printArray(1);
   }
   else{
     cout<<"NO!\n";
   }
 }*/
}

int network::calculateRemainingSpace(int nodeID){
//calculates the remaining space
 int i,totalSize=0,length,chID;
  length=addressTable[nodeID-1]->agentList.getLength();
  for(i=0;i<length;i++){  //for each agent in network nodeID
    chID=addressTable[nodeID-1]->agentList.getNodeID(i);
    totalSize+=a.getAgentAddress(chID-1)->size;
  }
  return(addressTable[nodeID-1]->size-totalSize);
}

void network::printNodeValues(int nodeID){
 int reservedSpace,i;
        nodeID--;
        cout<<"================================================\n";
  cout<<"for node "<<nodeID+1<<":\n";
  cout<<"size="<<addressTable[nodeID]->size<<"\n";
  cout<<"remaining space="<<addressTable[nodeID]->remainingSpace<<"\n";
        reservedSpace=0;
        for(i=0;i<addressTable[nodeID-1]->reservations.getLength();i++)
                reservedSpace+=addressTable[nodeID-1]->reservations.getNodeDistance(i);
        cout<<"reservedSpace="<<reservedSpace<<"\n";
  if(addressTable[nodeID]->mode==ASLEEP)
                cout<<"mode=ASLEEP\n";
```

```
        if(addressTable[nodeID]->mode==AWAKE)
                cout<<"mode=AWAKE\n";
        if(addressTable[nodeID]->mode==BUSY)
                cout<<"mode=BUSY\n";
        if(addressTable[nodeID]->mode==MIGRATING)
                cout<<"mode=MIGRATING\n";
        if(addressTable[nodeID]->mode==EVICTING)
                cout<<"mode=EVICTING\n";
        if(addressTable[nodeID]->mode==PRE_EVICTING)
                cout<<"mode=PRE_EVICTING\n";
  cout<<"agentsChecked="<<addressTable[nodeID]->agentsChecked<<"\n";
        cout<<"nodeBeingServed="<<addressTable[nodeID]->nodeBeingServed<<"\n";
  cout<<"agentBeingServed="<<addressTable[nodeID]->agentBeingServed<<"\n";
  cout<<"benefitsArray length="<<addressTable[nodeID]->benefitsArray.getLength()<<"\n";
        cout<<"agentsYetToChecklength="<<addressTable[nodeID]->agentsYetToCheck.getLength()<<"\n";
        cout<<"================================================\n";
}

void network::setupNetwork(){
 int i,random,j,totalSize=0;
 int currentAgent;
 srand(1);
    a.getAgentAddress(i)->totalBenefit=0;
                n.setOutputValue(INVALID);
  for(i=0;i<a.getNumberOfElements();i++){
    random=rand()%MAX_NODES;
    addressTable[random]->agentList.addEntry(i+1,0);
    addressTable[random]->size+=a.getAgentAddress(i)->size;
    totalSize+=a.getAgentAddress(i)->size;
    a.getAgentAddress(i)->agentsOccupied.init();
  }

  for(i=0;i<MAX_NODES;i++){
      addressTable[i]->size+=AVG_SIZE_MULTIPLE*totalSize/a.getNumberOfElements();       //set
network nodes size=size+average agent size
      addressTable[i]->remainingSpace=calculateRemainingSpace(i+1);
      addressTable[i]->mode=ASLEEP;
      addressTable[i]->agentsChecked=0;
                        addressTable[i]->benefitsArray.init();
                        addressTable[i]->agentBeingServed=INVALID;
                        addressTable[i]->agentsYetToCheck.init();
                        addressTable[i]->nodesOccupied.init();
                        addressTable[i]->reservations.init();
      //---------initialize agent parameters----------------------
      for(j=0;j<addressTable[i]->agentList.getLength();j++){    //for each agent in currentNode
                                currentAgent=addressTable[i]->agentList.getNodeID(j);
                                a.getAgentAddress(currentAgent-1)->attemptCounter.init();
                                a.getAgentAddress(currentAgent-1)->nodesChecked=0;
                                a.getAgentAddress(currentAgent-1)->isRootAgent=0;
                                a.getAgentAddress(currentAgent-1)->restorationFlag=0;
                                a.getAgentAddress(currentAgent-1)->parentAgent=INVALID;
                                a.getAgentAddress(currentAgent-1)->mostBeneficialNode=INVALID;
                                a.getAgentAddress(currentAgent-1)->maxBenefit=INVALID;
                                a.getAgentAddress(currentAgent-1)->agentBeingChecked=INVALID;
                                a.getAgentAddress(currentAgent-1)->totalBenefit=INVALID;
                                a.getAgentAddress(currentAgent-1)->agentsOccupied.init();
                                a.getAgentAddress(currentAgent-1)->migrationsAttempted.init();
                                a.getAgentAddress(currentAgent-1)->tempEvictions.init();
                                a.getAgentAddress(currentAgent-1)->benefitsArray.init();
                                a.getAgentAddress(currentAgent-1)->evictionsChecked.init();
                                a.getAgentAddress(currentAgent-1)->nodesToRestore.init();
                        }
  }
}

void network::printNetwork(){
  int i;
  for(i=0;i<MAX_NODES;i++){
    if(addressTable[i]->agentList.getLength()!=0){
      cout<<"node "<<i+1<<" has size:"<<addressTable[i]->size<<" and includes agents: ";
      addressTable[i]->agentList.printArray(0);
    }else
      cout<<"node "<<i+1<<" has size:"<<addressTable[i]->size<<" and includes no agents";
    cout<<" and connects with the nodes: ";
    addressTable[i]->childrenList.printArray(1);
```

```
    cout<<ENDLINE;

  }
}

int network::getNumberOfHops(int startID,int parentID,int destinationID,int result){
//returns the number of hops needed to reach destinationID starting from startID
  int pos,i,tempResult=0;
 if(startID==destinationID){
    return(0);
  }else if((addressTable[startID-1]->childrenList.getLength()==1)&&(addressTable[startID-1]-
>childrenList.getNodeID(0)==parentID)){//if we reached a node only connected to it's parent
    return(0);//this node does not connect with the one we are looking for, so return 0
        }else{                  //else
    pos=addressTable[startID-1]->childrenList.contains(destinationID);
    if(pos!=-1){        //if destinationID node is startID's children
      return(result+1);
    }else{              //else
      for(i=0;i<addressTable[startID-1]->childrenList.getLength();i++){ //keep searching in this
node's childrenList
                                if(addressTable[startID-1]->childrenList.getNodeID(i)!=parentID){
//for all nodes except parent
                                        tempResult+=getNumberOfHops(addressTable[startID-1]-
>childrenList.getNodeID(i),startID,destinationID,result+1);
                                }
      }
      if(tempResult==0) //if destinationID was not found in childrenList
                                return(0);
      else
                                return(tempResult);
    }
  }
}

int network::getLoad(int agentID,int nodeID){
//returns load if agentID is placed in nodeID
 int i,pAgentID,k,hops,load=0,traffic,length,chID;

 //========find load associated with parent agent===========
  pAgentID=a.getParentID(agentID);      //find parentAgent ID
  if(pAgentID==0)       //if it is the root agent
    traffic=0;
  else{                         //if it is not the root agent
    traffic=a.getAgentAddress(pAgentID-1)->childrenList.getNodeDistance2(agentID);//get traffic
between current agent and it's parent

    //-------find network node that contains pAgent--------
    k=0;
    while(k<MAX_NODES && addressTable[k]->agentList.contains(pAgentID)==-1)//scann network nodes
untill pAgentID is found
      k++;
    k++;        //k shows the ID of the network node that contains pAgentID
    //-----------------------------------------------------
    hops=getNumberOfHops(nodeID,-1,k,0);
    load+=traffic*hops;
  }
  //============================================================


  //=======find load associated with children agents========
  length=a.getAgentAddress(agentID-1)->childrenList.getLength();
  for(i=0;i<length;i++){ //for each child of agentID in appTree
    chID=a.getAgentAddress(agentID-1)->childrenList.getNodeID(i);

    //-------find network node that contains child agent--------
    k=0;
    while(k<MAX_NODES && addressTable[k]->agentList.contains(chID)==-1)//scann network nodes untill
pAgentID is found
      k++;
    k++;        //k shows the ID of the network node that contains chID
    //-----------------------------------------------------
```

```
    hops=getNumberOfHops(nodeID,-1,k,0);
    traffic=a.getAgentAddress(agentID-1)->childrenList.getNodeDistance(i);
    load+=traffic*hops;
  }
  //=============================================================
  return(load);
}

void network::initializeAgent(int agentID){
        agentID=agentID-1;
  a.getAgentAddress(agentID)->state=FREE; //FREE, BUSY, EVICTION
  a.getAgentAddress(agentID)->nodesChecked=0;
  a.getAgentAddress(agentID)->restorationFlag=0;
  a.getAgentAddress(agentID)->maxBenefit=INVALID;
  a.getAgentAddress(agentID)->mostBeneficialNode=INVALID;
  a.getAgentAddress(agentID)->agentBeingChecked=INVALID;
  a.getAgentAddress(agentID)->sum=0;
  a.getAgentAddress(agentID)->totalBenefit=0;
  a.getAgentAddress(agentID)->benefit=0;
  a.getAgentAddress(agentID)->parentAgent=INVALID;
  //int migrations;
        a.getAgentAddress(agentID)->isRootAgent=0;
  a.getAgentAddress(agentID)->evictionsChecked.init();
  a.getAgentAddress(agentID)->tempEvictions.init();
  a.getAgentAddress(agentID)->benefitsArray.init();
  a.getAgentAddress(agentID)->evictionPath.init();
  a.getAgentAddress(agentID)->previousNodes.init();
  a.getAgentAddress(agentID)->agentsOccupied.init();
        a.getAgentAddress(agentID)->migrationsAttempted.init();
        a.getAgentAddress(agentID)->attemptCounter.init();
        a.getAgentAddress(agentID)->nodesToRestore.init();

}

void network::initializeNode(int nodeID){
        nodeID=nodeID-1;
        addressTable[nodeID]->mode=ASLEEP;
        addressTable[nodeID]->agentsChecked=0;
  addressTable[nodeID]->agentBeingServed=INVALID;
  addressTable[nodeID]->benefitsArray.init();
        addressTable[nodeID]->agentsYetToCheck.init();
        addressTable[nodeID]->nodesOccupied.init();
        addressTable[nodeID]->nodeBeingServed=INVALID;
        addressTable[nodeID]->reservations.init();
        addressTable[nodeID]->isMostBeneficialFor=INVALID;
}

void network::moveAgentToNode(int agentID,int nodeID,int timeHip){
//moves agent to node and removes from nodes remainingSpace agentID's size
 int parentNode=0,i;
 dynamicList tempList;
 tempList.init();
   a.getAgentAddress(agentID-1)->migrations++;
    //-----find current network node hosting agentID---------
    while(addressTable[parentNode]->agentList.contains(agentID)==-1)
      parentNode++;
    //-----------------------------------------------------
                if(parentNode==nodeID-1)
                        a.getAgentAddress(agentID-1)->migrations--;
    if(addressTable[parentNode]->nodeBeingServed==INVALID && a.getAgentAddress(agentID-1)-
>isRootAgent==1){     //if this is the node who send the initial host request. Λύση στο conflict 9.

                        for(i=0;i<addressTable[parentNode]->nodesOccupied.getLength();i++){
                                n.initializeNode(addressTable[parentNode]-
>nodesOccupied.getNodeID(i));
                        }
                        if((addressTable[nodeID-1]->mode==ASLEEP
                        ||((addressTable[nodeID-1]->mode==EVICTING  || addressTable[nodeID-1]-
>mode==MIGRATING) && addressTable[nodeID-1]->nodeBeingServed==parentNode+1))
                        && parentNode!=nodeID-1){       //αν ο nodeID δεν περιέχει κάποιον
rootAgent.
                                n.initializeNode(nodeID);
                        }
                        //----------------------------------
```

```
                        n.initializeNode(parentNode+1);
                }else{
                        if(a.getAgentAddress(agentID-1)->previousNodes.contains(nodeID)!=-1){   //if
it is moving to a path back to it's initial position
                                a.getAgentAddress(agentID-1)->previousNodes.rmEntry(nodeID);//remove
it from previousNodes list
                        }else{
                                a.getAgentAddress(agentID-1)-
>previousNodes.addEntry(parentNode+1,0);
                        }
                }
        //=========delete agentID from current network node=============
    addressTable[parentNode]->agentList.rmEntry(agentID);       //remove agentID from this network
node
    addressTable[parentNode]->remainingSpace+=a.getAgentAddress(agentID-1)->size; //free the space
it was occupying in network node
  //============================================================

  //===========add agentID to network node=======================
    addressTable[nodeID-1]->agentList.addEntry(agentID,0); //add agentID to nodeID's agent list
    addressTable[nodeID-1]->remainingSpace-=a.getAgentAddress(agentID-1)->size;//reduce nodeID's
remaing space by agentID's size
    addressTable[nodeID-1]->reservations.rmEntry(agentID);
  //============================================================
        if(a.getAgentAddress(agentID-1)->previousNodes.getLength()>0){
                i=a.getAgentAddress(agentID-1)->parentAgent;
                tempList=a.getAgentAddress(agentID-1)->previousNodes;
                n.initializeAgent(agentID);
                a.getAgentAddress(agentID-1)->parentAgent=i;
                a.getAgentAddress(agentID-1)->previousNodes=tempList;
        }else{
                n.initializeAgent(agentID);
        }
}

void network::checkForEvictions(int agentID,int nodeID,int timeHip){
  //have to initialize sum=0 and agentBeingChecked=-1,tempEvictions
  int
currentAgent,i,j,parentNode=0,currentNode,position,flag,aBCheckedBenefit,penalty,tempParentNode;
  int timer=timeHip+1;
        int tempCurrentNode,reservedSpace;
  int aBChecked=a.getAgentAddress(agentID-1)->agentBeingChecked;
  dynamicList arguments;
//********--------------********
  while(addressTable[parentNode]->agentList.contains(agentID)==-1)
    parentNode++;
        if(addressTable[parentNode]->mode!=BUSY)
                addressTable[parentNode]->mode=EVICTING;
        if(addressTable[nodeID-1]->mode!=BUSY)//********--------------********
                addressTable[nodeID-1]->mode=EVICTING;
  position=a.getAgentAddress(agentID-1)->tempEvictions.contains2(aBChecked,nodeID);

        if(position!=-1){       //if agentBeingChecked is in tempEvictions
                a.getAgentAddress(agentID-1)->sum+=a.getAgentAddress(aBChecked-1)->size;
                if(a.getAgentAddress(aBChecked-1)->mostBeneficialNode==parentNode+1){   //αν
εκτοπίζεται κόμβος στη δικιά μου θέση.
                        if(addressTable[parentNode]->agentBeingServed!=INVALID)
//πρέπει να ενημερώσω τον agentBeingServed μου για αυτό
                                a.getAgentAddress(addressTable[parentNode]->agentBeingServed-1)-
>sum-=a.getAgentAddress(aBChecked-1)->size;
                }
                        if(addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]-
>mode==EVICTING
                        && addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]-
>agentBeingServed==aBChecked && a.getAgentAddress(agentID-1)->restorationFlag==1){
                                addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]-
>agentBeingServed=INVALID;
                        }

                        if(addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]-
>reservations.contains(aBChecked)==-1)
```

```
                                addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]-
>reservations.addEntry(aBChecked,a.getAgentAddress(aBChecked-1)->size);
                        tempParentNode=0;
                        while(addressTable[tempParentNode]->agentList.contains(aBChecked)==-1)
                                tempParentNode++;

    a.getAgentAddress(agentID-1)->tempEvictions.rmEntryFrom(position);  //remove from
tempEvictionsList
    aBCheckedBenefit=a.getAgentAddress(agentID-1)->benefitsArray.getNodeDistance(position);
    a.getAgentAddress(agentID-1)->totalBenefit+=aBCheckedBenefit;
    a.getAgentAddress(agentID-1)->benefitsArray.rmEntryFrom(position);  //also remove from benefits
array

                for(i=0;i<a.getAgentAddress(aBChecked-1)->evictionsChecked.getLength();i++)
{//προσθέτω στο evictionsChecked του agentID το evictionsChecked του aBChecked
                        if(a.getAgentAddress(agentID-1)-
>evictionsChecked.contains2(a.getAgentAddress(aBChecked-1)-
>evictionsChecked.getNodeID(i),a.getAgentAddress(aBChecked-1)-
>evictionsChecked.getNodeDistance(i))==INVALID){
                                a.getAgentAddress(agentID-1)-
>evictionsChecked.addEntry(a.getAgentAddress(aBChecked-1)-
>evictionsChecked.getNodeID(i),a.getAgentAddress(aBChecked-1)-
>evictionsChecked.getNodeDistance(i));
                        }
                }
                for(i=0;i<a.getAgentAddress(aBChecked-1)->nodesToRestore.getLength();i++){
                        if(a.getAgentAddress(agentID-1)-
>nodesToRestore.contains(a.getAgentAddress(aBChecked-1)->nodesToRestore.getNodeID(i))==INVALID){
                                a.getAgentAddress(agentID-1)-
>nodesToRestore.addEntry(a.getAgentAddress(aBChecked-1)->nodesToRestore.getNodeID(i),nodeID);
                        }
                }
                if(a.getAgentAddress(agentID-1)-
>evictionsChecked.contains2(aBChecked,nodeID)==INVALID){
                        a.getAgentAddress(agentID-1)-
>evictionsChecked.addEntry(aBChecked,nodeID);//add it to the evictionsChecked list
                }
                if(a.getAgentAddress(agentID-1)->nodesToRestore.contains(nodeID)==INVALID)
                        a.getAgentAddress(agentID-1)->nodesToRestore.addEntry(nodeID,nodeID);
                if(a.getAgentAddress(agentID-1)-
>nodesToRestore.contains(a.getAgentAddress(aBChecked-1)->mostBeneficialNode)==INVALID){
                        if((addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]-
>mode!=BUSY && addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]->mode!=EVICTING
                                && addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-
1]->mode!=PRE_EVICTING)
                        ||(addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]-
>mode==EVICTING
                        && addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]-
>nodeBeingServed==tempParentNode+1))
                                a.getAgentAddress(agentID-1)-
>nodesToRestore.addEntry(a.getAgentAddress(aBChecked-1)->mostBeneficialNode,nodeID);
                }
                reservedSpace=0;
                for(i=0;i<addressTable[nodeID-1]->reservations.getLength();i++)
                        if(addressTable[nodeID-1]->reservations.getNodeID(i)!=agentID)
                                reservedSpace+=addressTable[nodeID-1]-
>reservations.getNodeDistance(i);
            //-----also move aBChecked to most beneficial destination----------//moved it here
from above.
                        tempCurrentNode=a.getAgentAddress(aBChecked-1)->mostBeneficialNode;
                        n.moveAgentToNode(aBChecked,tempCurrentNode,timer);
                        a.getAgentAddress(aBChecked-1)->mostBeneficialNode=tempCurrentNode;


    if((/*a.getAgentAddress(agentID-1)->sum+*/addressTable[nodeID-1]->remainingSpace-
reservedSpace)<a.getAgentAddress(agentID-1)->size){        //if still haven't freed enough space

                a.getAgentAddress(agentID-1)->agentBeingChecked=-1; //get new agentBeingChecked
                        checkForEvictions(agentID,nodeID,timer);
          }else{        //if eventually, enough space could be freed
      //-------SEND_HOST_REQUEST of agentID to nodeID---------------  //********-------------
********
                        //-------and initialize all agents from tempEvictions of agentID--------
                        //--------add nodeID's nodesOccupied list to parentNode's list.-----------
                        for(i=0;i<addressTable[nodeID-1]->nodesOccupied.getLength();i++){
```

```
                                if(addressTable[parentNode]-
>nodesOccupied.contains(addressTable[nodeID-1]->nodesOccupied.getNodeID(i))==-1)
                                        addressTable[parentNode]-
>nodesOccupied.addEntry(addressTable[nodeID-1]->nodesOccupied.getNodeID(i),nodeID);
                        }
                        //-------------------------------------------------------------------------
-------
                        timer++;
                        if(addressTable[parentNode]->nodeBeingServed==INVALID  &&
a.getAgentAddress(agentID-1)->isRootAgent==1){ //if this is the rootAgen.Λύση στο conflict 9.
                                arguments.init();
                                arguments.addEntry(agentID,0);
                                arguments.addEntry(nodeID,0);
                                h.addToHip(timer++,MOVE_AGENT,arguments);
                                a.getAgentAddress(agentID-1)->totalBenefit=0;
                                addressTable[parentNode]->mode=MIGRATING;
                                addressTable[nodeID-1]->mode=MIGRATING;
                                if(addressTable[nodeID-1]->reservations.contains(agentID)==-1)
                                        addressTable[nodeID-1]-
>reservations.addEntry(agentID,a.getAgentAddress(agentID-1)->size);
                        }else{  //if this is not the rootAgent//********-------------********
                                if(a.getAgentAddress(agentID-1)->restorationFlag==0){
                                        for(i=a.getAgentAddress(agentID-1)-
>evictionsChecked.getLength()-1;i>=0;i--){   //for every eviction done.
                                                position=a.getAgentAddress(agentID-1)-
>nodesToRestore.contains2(a.getAgentAddress(agentID-1)->evictionsChecked.getNodeDistance(i),nodeID);
                                                if(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i)==nodeID
                                                || position!=-1){
                                                        currentAgent=a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeID(i);
                                                        //-------- restore migration of
currentAgent----------------
                                                        arguments.init();
//********-------------********
                                                        arguments.addEntry(currentAgent,0);

arguments.addEntry(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i),0);//********-------------********
                                                        h.addToHip(timer,MOVE_AGENT,arguments);
                                                        timer++;
                                                        addressTable[a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i)-1]->mode=MIGRATING;
                                                        a.getAgentAddress(currentAgent-1)-
>migrations=a.getAgentAddress(currentAgent-1)->migrations-2;

                                                        if(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i)==nodeID)
                                                                a.getAgentAddress(agentID-1)->sum-
=a.getAgentAddress(currentAgent-1)->size;
                                                        tempParentNode=0;
                                                        while(addressTable[tempParentNode]-
>agentList.contains(currentAgent)==-1)
                                                                tempParentNode++;
                                                        if((addressTable[tempParentNode]->mode!=BUSY
&& addressTable[tempParentNode]->mode!=EVICTING && addressTable[tempParentNode]->mode!=PRE_EVICTING)
                                                        ||(addressTable[tempParentNode]-
>mode==EVICTING && addressTable[tempParentNode]->nodeBeingServed==nodeID)){
                                                                addressTable[tempParentNode]-
>mode=MIGRATING;
                                                        }
                                                }
                                        }
                                        if((addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->mode!=BUSY && addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->mode!=EVICTING
                                                && addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->mode!=PRE_EVICTING)
                                        ||(addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->mode==EVICTING && addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->nodeBeingServed==nodeID)){
```

```
                                        addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->mode=MIGRATING;
                                }
                        }

                        arguments.init();
                        arguments.addEntry(agentID,0);
                        arguments.addEntry(nodeID,0);
                        arguments.addEntry(a.getAgentAddress(agentID-1)->totalBenefit,0);
                        h.addToHip(timer++,ANSWER,arguments);   //********-------------
********
                        a.getAgentAddress(agentID-1)->totalBenefit=0;
                }
                        for(i=0;i<a.getAgentAddress(agentID-1)-
>tempEvictions.getLength();i++){
                                position=addressTable[parentNode]-
>nodesOccupied.contains(a.getAgentAddress(agentID-1)->tempEvictions.getNodeID(i));
                                if(position!=-1)
                                        addressTable[parentNode]-
>nodesOccupied.rmEntryFrom(position);
                                n.initializeAgent(a.getAgentAddress(agentID-1)-
>tempEvictions.getNodeID(i));
                        }
                        a.getAgentAddress(agentID-1)->tempEvictions.init();
                        a.getAgentAddress(agentID-1)->benefitsArray.init();
    }
//=============================================================================================
===
  }else{  //=========================if agentBeingChecked not in tempEvictions
list============================

//=============================================================================================
===
    position=0;
    flag=0;
    while(position<a.getAgentAddress(agentID-1)->tempEvictions.getLength() && flag==0){ //check in
tempEvictions of agentBeingServed
                if(a.getAgentAddress(agentID-1)-
>tempEvictions.getNodeDistance(position)==nodeID){      //if there are evictions refered to this
node
                        flag=1;
//raise flag
      }else
                        position++;
    }
    if(flag==1){                //if there are still agents in tempEvictions of agentBeingServed
not yet checked
      a.getAgentAddress(agentID-1)->agentBeingChecked=a.getAgentAddress(agentID-1)-
>tempEvictions.getNodeID(position);  //get the next agentBeingChecked from benefitsArray
      aBChecked=a.getAgentAddress(agentID-1)->agentBeingChecked;
      a.getAgentAddress(agentID-1)->tempEvictions.rmEntryFrom(position);
//and remove it from tempEvictions
      a.getAgentAddress(agentID-1)->benefitsArray.rmEntryFrom(position);
//and from benefitsArray

      penalty=getLoad(aBChecked,nodeID)-getLoad(aBChecked,a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode);  //estimate penalty of such migration
      aBCheckedBenefit=a.getAgentAddress(aBChecked-1)->benefit+penalty;
      if(aBCheckedBenefit>0){   //if it is a beneficial migration
                        arguments.init();                       //send host request
                        arguments.addEntry(aBChecked,0);
                        arguments.addEntry(a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode,0);
                        arguments.addEntry(aBCheckedBenefit,0);
                        if(addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-
1]->mode!=BUSY)      //********-------------********
                                addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->mode=EVICTING;

                        tempParentNode=0;
                        while(addressTable[tempParentNode]-
>agentList.contains(aBChecked)==-1)
                                tempParentNode++;

                        if(a.getAgentAddress(aBChecked-1)->mostBeneficialNode!
```

```
=addressTable[tempParentNode]->nodeBeingServed){  //      αν ο mostBeneficialNode δεν περιέχει τον
agentBeingServed του tempParentNode
                                //δηλαδή αν ο aBChecked δεν στέλνει αίτηση σε προηγούμενο node
                                        if(addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->mode!=BUSY &&
                                                addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->nodeBeingServed==nodeID){
                                                addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->agentBeingServed=aBChecked;
                                        }
                                }
                                a.getAgentAddress(aBChecked-1)->benefitsArray.init();
                                a.getAgentAddress(aBChecked-1)->tempEvictions.init();
                                a.getAgentAddress(aBChecked-1)->restorationFlag=1;
                                h.addToHip(timer,SEND_HOST_REQUEST,arguments);  //get it's benefit

      }else{                                                    //if it is not a beneficial
migration
                                arguments.init();       //send invalid reply
                                arguments.addEntry(aBChecked,0);
                                arguments.addEntry(a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode,0);
                                arguments.addEntry(INVALID,0);
                                h.addToHip(timer,ANSWER,arguments);
      }
    }else{// if((a.getAgentAddress(agentID-1)->sum+addressTable[nodeID-1]-
>remainingSpace)<a.getAgentAddress(agentID-1)->size){         //if enough space couldn't eventually
be freed
                        i=a.getAgentAddress(agentID-1)->evictionsChecked.getLength();
                        while(i>=0){    //for every eviction done.
                                position=a.getAgentAddress(agentID-1)-
>nodesToRestore.contains2(a.getAgentAddress(agentID-1)->evictionsChecked.getNodeDistance(i),nodeID);
                                if(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i)==nodeID
                                || position!=-1){
                                        currentAgent=a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeID(i);
                                        //-------- restore migration of currentAgent---------------
                                        arguments.init();       //********-------------********
                                        arguments.addEntry(currentAgent,0);
                                        arguments.addEntry(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i),0);//********-------------********
                                        h.addToHip(timer,MOVE_AGENT,arguments);
                                        timer++;
a.getAgentAddress(currentAgent-1)->migrations=a.getAgentAddress(currentAgent-1)->migrations-2;
//25/1/2016
                                        addressTable[a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i)-1]->mode=MIGRATING;

                                        if(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i)==nodeID)
                                                a.getAgentAddress(agentID-1)->sum-
=a.getAgentAddress(currentAgent-1)->size;
                                        tempParentNode=0;
                                        while(addressTable[tempParentNode]-
>agentList.contains(currentAgent)==-1)
                                                tempParentNode++;
                                        if((addressTable[tempParentNode]->mode!=BUSY    &&
addressTable[tempParentNode]->mode!=EVICTING && addressTable[tempParentNode]->mode!=PRE_EVICTING)
                                        ||(addressTable[tempParentNode]->mode==EVICTING &&
addressTable[tempParentNode]->nodeBeingServed==nodeID)){
                                                addressTable[tempParentNode]->mode=MIGRATING;
                                        }
                                        a.getAgentAddress(agentID-1)-
>evictionsChecked.rmEntryFrom(i);
                                }else{
                                        i--;
                                }
                        }

                        a.getAgentAddress(agentID-1)->totalBenefit=0;
```

```
      //---------notify agentID migration was not possible------------
      arguments.init();
      arguments.addEntry(agentID,0);
      arguments.addEntry(nodeID,0);
      arguments.addEntry(INVALID,0);    //INVALID benefit
      h.addToHip(timer,ANSWER,arguments);
      //-----------------------------------------------------------------
                        for(i=0;i<a.getAgentAddress(agentID-1)->tempEvictions.getLength();i++){
//30/10/2015
                                position=addressTable[parentNode]-
>nodesOccupied.contains(a.getAgentAddress(agentID-1)->tempEvictions.getNodeID(i));
                                if(position!=-1)
                                        addressTable[parentNode]-
>nodesOccupied.rmEntryFrom(position);
                                n.initializeAgent(a.getAgentAddress(agentID-1)-
>tempEvictions.getNodeID(i));
                        }
                        a.getAgentAddress(agentID-1)->tempEvictions.init();
                        a.getAgentAddress(agentID-1)->benefitsArray.init();
                }
  }
}

void network::notifyNode(int agentID,int nodeID,int benefit,int timeHip){
//notify node currently hosting agentID (parentNode)
  dynamicList arguments,tempList;
  int sum,parentNode=0,i,j,currentAgent,temp,tempBenefit,length,currentNode;
        int flag=0,position,localBenefit,maxBenefit,maxNode,aBServed,tempParentNode;
  a.getAgentAddress(agentID-1)->nodesChecked--;  //inform agentID that a node (to which he
previously sent a HOST_REQUEST), has returned an answer
  while(addressTable[parentNode]->agentList.contains(agentID)==-1)
    parentNode++;
        if(addressTable[parentNode]->nodeBeingServed!=INVALID){
                tempParentNode=addressTable[parentNode]->nodeBeingServed-1;
        }else
                tempParentNode=INVALID;
        if(addressTable[parentNode]->agentList.contains(addressTable[nodeID-1]-
>isMostBeneficialFor)!=-1)
                temp=1;
        else
                temp=0;

        if(a.getAgentAddress(agentID-1)->nodesChecked<0){       //if function call was triggered by
checkForEvictions
                if(benefit!=INVALID){  //if eviction is beneficial
                        a.getAgentAddress(addressTable[parentNode]->agentBeingServed-1)-
>agentBeingChecked=agentID;     //********-------------********
                        temp=a.getAgentAddress(addressTable[parentNode]->agentBeingServed-1)-
>benefitsArray.addEntryInOrder(agentID,benefit);  //add inOrder and get position
      a.getAgentAddress(addressTable[parentNode]->agentBeingServed-1)-
>tempEvictions.addEntryAt(agentID,parentNode+1,temp);     //add it to tempEvictions of
agentBeingServed
                        addressTable[nodeID-1]->isMostBeneficialFor=agentID;
                        checkForEvictions(addressTable[parentNode]-
>agentBeingServed,parentNode+1,timeHip);
    }else{      //else if eviction was invalid
                        addressTable[nodeID-1]->agentBeingServed=INVALID;
//********-------------********
                        cout<<parentNode+1<<ENDLINE;
                        position=a.getAgentAddress(addressTable[parentNode]->agentBeingServed-1)-
>tempEvictions.contains(agentID);
                //-------restore all nodes to EVICTION or ASLEEP mode.31/11/2015-------------
      i=0;
                        while(i<a.getAgentAddress(agentID-1)->nodesToRestore.getLength()){
//for every node that needs to be restored.
                                if(a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeDistance(i)==nodeID){
                                        if(addressTable[a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i)-1]->mode!=BUSY)
                                                if(temp!=1){
                                                        n.initializeNode(a.getAgentAddress(agentID-
1)->nodesToRestore.getNodeID(i));
                                                }else{
                                                        addressTable[a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i)-1]->mode=EVICTING;
```

```
                                                addressTable[a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i)-1]->agentBeingServed=INVALID;
                                        }
                                        currentNode=a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i);
                                        for(j=0;j<addressTable[currentNode-1]-
>agentList.getLength();j++){      //also restore their agents
                                                n.initializeAgent(addressTable[currentNode-1]-
>agentList.getNodeID(j));
                                        }
                                        a.getAgentAddress(agentID-1)->nodesToRestore.rmEntryFrom(i);
                                }else{
                                        i++;
                                }
                        }
                        //-------------------------------------------------------------------------
----
                        if(a.getAgentAddress(agentID-1)->nodesToRestore.getLength()<=0)
                                n.initializeAgent(agentID);
                        a.getAgentAddress(addressTable[parentNode]->agentBeingServed-1)-
>agentBeingChecked=INVALID;     //new agent has to become agentBeingChecked
      checkForEvictions(addressTable[parentNode]->agentBeingServed,parentNode+1,timeHip);     //call
check for evictions
                }

//=========================================================================================================
=====================================================
        }else{
//=========================================================================================================
=====================================================
                if(a.getAgentAddress(agentID-1)->maxBenefit<benefit){           //if
benefit/agentSize of this migration/eviction is greater than currently maxBenefit
      a.getAgentAddress(agentID-1)->maxBenefit=benefit;
                        if(a.getAgentAddress(agentID-1)->mostBeneficialNode!=INVALID){  //if there
is a currently most Beneficial Node
                                if(a.getAgentAddress(agentID-1)->mostBeneficialNode!
=tempParentNode+1 && temp!=1//if currently most beneficial node δεν χρησιμοποιείται από κάποιον
συγκάτοικο agent
                                && ((addressTable[a.getAgentAddress(agentID-1)->mostBeneficialNode-
1]->mode!=BUSY        && addressTable[a.getAgentAddress(agentID-1)->mostBeneficialNode-1]->mode!
=EVICTING
                                        && addressTable[a.getAgentAddress(agentID-1)-
>mostBeneficialNode-1]->mode!=MIGRATING)   //και δεν είναι BUSY
                                || ((addressTable[a.getAgentAddress(agentID-1)->mostBeneficialNode-
1]->mode==EVICTING
                                                || addressTable[a.getAgentAddress(agentID-1)-
>mostBeneficialNode-1]->mode==MIGRATING) //ή αν είναι EVICTING ή MIGRATING τότε
                                                && addressTable[a.getAgentAddress(agentID-1)-
>mostBeneficialNode-1]->nodeBeingServed==parentNode+1
                                                && addressTable[a.getAgentAddress(agentID-1)-
>mostBeneficialNode-1]->agentBeingServed==agentID))){      //αν ο nodeBeingServed του είναι ο
parentNode
                                        i=0;
                                        while(i<addressTable[parentNode]->nodesOccupied.getLength())
{
                                                if(addressTable[parentNode]-
>nodesOccupied.getNodeDistance(i)==a.getAgentAddress(agentID-1)->mostBeneficialNode){
//20/11/2015
                                                        n.initializeNode(addressTable[parentNode]-
>nodesOccupied.getNodeID(i));
                                                        addressTable[parentNode]-
>nodesOccupied.rmEntryFrom(i);
                                                }else
                                                        i++;
                                        }
                                        n.initializeNode(a.getAgentAddress(agentID-1)-
>mostBeneficialNode);     //free and put currently most beneficial node to SLEEP
                                }
                                i=0;
                                while(i<a.getAgentAddress(agentID-1)->evictionsChecked.getLength()){
                                        position=a.getAgentAddress(agentID-1)-
```

```
>nodesToRestore.contains2(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i),a.getAgentAddress(agentID-1)->mostBeneficialNode);
                                        if(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i)==a.getAgentAddress(agentID-1)->mostBeneficialNode
                                        || position!=-1){
                                                a.getAgentAddress(agentID-1)-
>evictionsChecked.rmEntryFrom(i);
                                        }else
                                                i++;
                                }
                        }
      a.getAgentAddress(agentID-1)->mostBeneficialNode=nodeID;//and set this node as the currently
most beneficial for the agent to MIGRATE
    }else if(nodeID!=tempParentNode+1 && ((addressTable[nodeID-1]->mode==ASLEEP)
                || ((addressTable[nodeID-1]->mode==EVICTING || addressTable[nodeID-1]-
>mode==MIGRATING) && addressTable[nodeID-1]->nodeBeingServed==parentNode+1))){
                        if(temp!=1){
                                i=0;
                                while(i<addressTable[parentNode]->nodesOccupied.getLength()){
                                        if(addressTable[parentNode]-
>nodesOccupied.getNodeDistance(i)==nodeID){
                                                n.initializeNode(addressTable[parentNode]-
>nodesOccupied.getNodeID(i));
                                                addressTable[parentNode]-
>nodesOccupied.rmEntryFrom(i);
                                        }else
                                                i++;
                                }
                                if(addressTable[nodeID-1]->mode!=BUSY)
                                        n.initializeNode(nodeID);       //free this node
                                i=0;
                                while(i<a.getAgentAddress(agentID-1)->evictionsChecked.getLength())
{
                                        position=a.getAgentAddress(agentID-1)-
>nodesToRestore.contains2(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i),nodeID);
                                        if(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i)==nodeID
                                        || position!=-1){
                                                a.getAgentAddress(agentID-1)-
>evictionsChecked.rmEntryFrom(i);
                                        }else
                                                i++;
                                }
                        }
                }


                //-------restore all nodes to EVICTION or ASLEEP mode--------------
      i=0;
                        while(i<a.getAgentAddress(agentID-1)->nodesToRestore.getLength()){
//for every node that needs to be restored.
                                if(a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeDistance(i)==nodeID){
                                        if(a.getAgentAddress(agentID-1)-
>mostBeneficialNode==nodeID){
                                                if(addressTable[a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i)-1]->mode!=BUSY)
                                                        addressTable[a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i)-1]->mode=EVICTING;
                                                        addressTable[a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i)-1]->agentBeingServed=INVALID;
                                        }else{
                                                if(addressTable[a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i)-1]->mode!=BUSY)
                                                        if(temp!=1){

n.initializeNode(a.getAgentAddress(agentID-1)->nodesToRestore.getNodeID(i));
                                                        }else{

addressTable[a.getAgentAddress(agentID-1)->nodesToRestore.getNodeID(i)-1]->mode=EVICTING;

addressTable[a.getAgentAddress(agentID-1)->nodesToRestore.getNodeID(i)-1]-
>agentBeingServed=INVALID;
                                                        }
```

```
                                        }
                                        currentNode=a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i);
                                        for(j=0;j<addressTable[currentNode-1]-
>agentList.getLength();j++){      //also restore their agents
                                                n.initializeAgent(addressTable[currentNode-1]-
>agentList.getNodeID(j));
                                        }
                                        a.getAgentAddress(agentID-1)->nodesToRestore.rmEntryFrom(i);
                                }else
                                        i++;
                        }
                        //-------------------------------------------------------------

                if(a.getAgentAddress(agentID-1)->nodesChecked==0){      //if agent got response from
all nodes to which he send hostRequests.
                        addressTable[parentNode]->agentsChecked--;                      //notify
node that max benefit of agentID has been found
                        tempBenefit=a.getAgentAddress(agentID-1)-
>maxBenefit*1000/a.getAgentAddress(agentID-1)->size;   //(*1000 so the outcome is an integer)
                        if(a.getAgentAddress(agentID-1)->maxBenefit!=INVALID){  //if there was a
possible eviction
                                addressTable[parentNode]-
>benefitsArray.addEntryInOrder(agentID,tempBenefit); //add it to the benefitsArray of parent node in
increasing order
                                addressTable[a.getAgentAddress(agentID-1)-
>mostBeneficialNode-1]->isMostBeneficialFor=agentID;
                        }else{
                                if(a.getAgentAddress(agentID-1)->state==BUSY){
                                        n.initializeAgent(agentID);
                                        a.getAgentAddress(agentID-1)->state=BUSY;
                                        a.getAgentAddress(agentID-1)->isRootAgent=1;
                                }else
                                        n.initializeAgent(agentID);
                        }
    }
    if(addressTable[parentNode]->nodeBeingServed!=nodeID && addressTable[nodeID-1]-
>agentBeingServed==agentID){ //αν δεν στέλνει αίτημα στον προηγούμενο κομβο και ο nodeID εξυπηρετεί
τον agentID
                        addressTable[nodeID-1]->agentBeingServed=INVALID;
                }

    if(addressTable[parentNode]->agentsChecked==0){     //if all agents of parentNode have returned
their maxBenefit
      if(addressTable[parentNode]->benefitsArray.getLength()>0){  //if there was at least one
possible eviction
                                if(addressTable[parentNode]->agentBeingServed!=INVALID){
//if agentID is not the rootAgent
                                        tempParentNode=0;
                                        while(addressTable[tempParentNode]-
>agentList.contains(addressTable[parentNode]->agentBeingServed)==-1) //get parentNode of
agentBeingServed.
                                                tempParentNode++;
                                        for(i=0;i<addressTable[parentNode]-
>benefitsArray.getLength();i++){     //for every possible eviction
                                                currentAgent=addressTable[parentNode]-
>benefitsArray.getNodeID(i);
                                                a.getAgentAddress(addressTable[parentNode]-
>agentBeingServed-1)->tempEvictions.addEntry(currentAgent,parentNode+1);     //add it to
tempEvictions of agentBeingServed
                                                a.getAgentAddress(addressTable[parentNode]-
>agentBeingServed-1)->benefitsArray.addEntry(currentAgent,a.getAgentAddress(currentAgent-1)-
>maxBenefit);
                                                if(addressTable[tempParentNode]-
>nodesOccupied.contains(a.getAgentAddress(currentAgent-1)->mostBeneficialNode)!=-2      //fake!
4/12/2015
                                                        &&
((addressTable[a.getAgentAddress(currentAgent-1)->mostBeneficialNode-1]->mode!=BUSY
                                                                &&
addressTable[a.getAgentAddress(currentAgent-1)->mostBeneficialNode-1]->mode!=EVICTING)
                                                        ||
```

```
(addressTable[a.getAgentAddress(currentAgent-1)->mostBeneficialNode-1]->mode==EVICTING
                                                                    &&
addressTable[a.getAgentAddress(currentAgent-1)->mostBeneficialNode-1]-
>nodeBeingServed==parentNode+1))){
                                                addressTable[tempParentNode]-
>nodesOccupied.addEntry(a.getAgentAddress(currentAgent-1)->mostBeneficialNode,parentNode+1);
                                            }
                                    }
                                    a.getAgentAddress(addressTable[parentNode]-
>agentBeingServed-1)->agentBeingChecked=INVALID;
                                    checkForEvictions(addressTable[parentNode]-
>agentBeingServed,parentNode+1,timeHip);
                                }else{                          //if agentID is the one who sent
the initial host request       //********-------------********
                                    cout<<"That never happens!\n";
                                }
      }else{ //else if there weren't any possible evictions
                                addressTable[parentNode]->benefitsArray.init();
                                if(addressTable[parentNode]->nodeBeingServed!=INVALID ||
a.getAgentAddress(agentID-1)->isRootAgent==0){ // if this is not the rootAgent
                                    arguments.init();
//return benefit INVALID to agentBeingServed
                                    arguments.addEntry(addressTable[parentNode]-
>agentBeingServed,0);
                                    arguments.addEntry(parentNode+1,0);
                                    arguments.addEntry(INVALID,0);
                                    h.addToHip(timeHip+1,ANSWER,arguments);
                                    aBServed=addressTable[parentNode]->agentBeingServed;
//********-------------********
                                    n.initializeAgent(agentID);
                                }else{  //if this is the rootAgent
                                    //----find parent of agentID-------initialize--
                                parentNode=0;
                                    while(addressTable[parentNode]-
>agentList.contains(agentID)==-1)
                                            parentNode++;

                                    addressTable[parentNode]-
>agentsYetToCheck.rmEntry(agentID);    //remove this agent from agentsYetToBeChecked list (as it
failed to return a valid result)

                                    for(i=0;i<addressTable[parentNode]-
>nodesOccupied.getLength();i++){
                                            n.initializeNode(addressTable[parentNode]-
>nodesOccupied.getNodeID(i));
                                        }
                                        addressTable[parentNode]->nodesOccupied.init();

                                        if((addressTable[nodeID-1]->mode==ASLEEP)
                                        ||((addressTable[nodeID-1]->mode==EVICTING ||
addressTable[nodeID-1]->mode==MIGRATING) && addressTable[nodeID-1]->nodeBeingServed==parentNode+1))
                                            n.initializeNode(nodeID);

                                        arguments.init();
                                        arguments.addEntry(agentID,0);
                                        arguments.addEntry(parentNode+1,0);
                                        h.addToHip(timeHip+1,MOVE_AGENT,arguments);
                                        //-----------------------------------
                                }
      }
    }
  }
}

void network::migrationCheck(int agentID,int nodeID,int benefit,int timeHip){
int parentNode=0;
  dynamicList arguments;
  arguments.init();

  while(addressTable[parentNode]->agentList.contains(agentID)==-1)
    parentNode++;

  if(a.getAgentAddress(agentID-1)->state!=EVICTION){
    arguments.addEntry(agentID,0);
    arguments.addEntry(nodeID,0);
```

```
    h.addToHip(timeHip+1,MOVE_AGENT,arguments);
                timeHip+=2;     //********-------------********

                if(addressTable[parentNode]->mode==MIGRATING){  //δηλαδή όταν ο agentID χωρά χωρίς
eviction στο nodeID
                        addressTable[parentNode]->mode==BUSY;
                }
  }else{        //if agentID state==EVICTION
                if(addressTable[nodeID-1]->mode==ASLEEP && addressTable[nodeID-1]-
>agentBeingServed==INVALID){  // Για να αποφευγχθεί η περίπτωση όπου ο nodeID
                        addressTable[nodeID-1]->agentBeingServed=agentID;
//ήταν BUSY και έχει μόλις αποφασιστεί τι θα γινόταν και έγινε πάλι ASLEEP
                }
                if(addressTable[nodeID-1]->mode==MIGRATING){    //if nodeID was ASLEEP before
agentID's hostRequest
                        addressTable[nodeID-1]->mode=EVICTING;//********-------------********
                }
                arguments.init();
    arguments.addEntry(agentID,0);
    arguments.addEntry(nodeID,0);
    arguments.addEntry(benefit,0);
    h.addToHip(timeHip+1,ANSWER,arguments);
  }
}

void network::sendHostRequestNFL(int agentID,int nodeID,int benefit,int timeHip){
 int previousTimer=0,currentTimer=1,parentNode=0,numberOfAgents,currentAgent,position,reservedSpace;
  int
i,j,numberOfNodes,penalty,temp,currentNode,agentServed,flag=0,flag2=0,flag3=0,PEflag=0,invalidFlag=0
;     //PEflag==PRE_EVICTING_FLAG
  dynamicList arguments;
  arguments.init();
  numberOfAgents=addressTable[nodeID-1]->agentList.getLength(); //get number of agents hosted in
nodeID
  //-----------find node hosting agentID-----------------
  while(addressTable[parentNode]->agentList.contains(agentID)==-1)
    parentNode++;
  //----------------------------------------------------
        agentServed=addressTable[nodeID-1]->agentBeingServed;   //set agentServed as the agent Being
Served by nodeID
// cout<<parentNode+1<<" mode="<<a.getAgentAddress(agentID-1)->state<<ENDLINE;
        if(addressTable[nodeID-1]->mode==AWAKE) //********-------------********
                addressTable[nodeID-1]->mode=ASLEEP;
//----raise flag in case agentID is sending host request to the node hosting the agent currently
trying to evict agentID(previous in path node)----
        if(nodeID==addressTable[parentNode]->nodeBeingServed)  //********-------------********
                flag=1;

        if(addressTable[nodeID-1]->nodeBeingServed==parentNode+1){ //********-------------********
                if(agentServed!=agentID && agentServed!=INVALID)                //if nodeID is busy
serving another agent inside parentNode
                        flag2=1;
//raise flag2
                else
//else if agentBeingServed by nodeID is agentID, or no agent is currently beingServed by nodeID(έχει
κληθεί από την checkForEvictions)
                        flag3=1;
//raise flag3
        }

        if(addressTable[parentNode]->mode==PRE_EVICTING){
                PEflag=1;
                addressTable[parentNode]->mode=MIGRATING;
        }

        if(a.getAgentAddress(agentID-1)->attemptCounter.getNodeDistance2(nodeID)>140){
//********-------------********
                a.getAgentAddress(agentID-1)->attemptCounter.rmEntry(nodeID);
                invalidFlag=0;
        }
                if(addressTable[nodeID-1]->mode==ASLEEP){
```

```
                        for(i=0;i<addressTable[nodeID-1]->agentList.getLength();i++){
                                if(a.getAgentAddress(addressTable[nodeID-1]-
>agentList.getNodeID(i)-1)->state!=FREE){
                                        flag2=1;
                                }
                        }
                }
        reservedSpace=0;
        for(i=0;i<addressTable[nodeID-1]->reservations.getLength();i++)
                if(addressTable[nodeID-1]->reservations.getNodeID(i)!=agentID)
                        reservedSpace+=addressTable[nodeID-1]->reservations.getNodeDistance(i);


        if((addressTable[nodeID-1]->mode==MIGRATING ||flag2==1
        || addressTable[nodeID-1]->mode==PRE_EVICTING) && invalidFlag==0){      //if node is busy
migrating, or is currently doing eviction of another agent of parentNode
                if(PEflag==1){  //in case parentNode has sent host requests to other nodes as well,
apart from nodeID, so it needs to be in mode=EVICTING//28/11/2015
                        addressTable[parentNode]->mode=PRE_EVICTING;    //19/12/2015
                }
                temp=a.getAgentAddress(agentID-1)->attemptCounter.getNodeDistance2(nodeID)+1;
//********-------------********
                a.getAgentAddress(agentID-1)->attemptCounter.rmEntry(nodeID);
//********-------------********
                a.getAgentAddress(agentID-1)->attemptCounter.addEntry(nodeID,temp);
//********-------------********
    arguments.addEntry(agentID,0);      //wait and try again in next timeHip
    arguments.addEntry(nodeID,0);
    arguments.addEntry(benefit,0);
    h.addToHip(timeHip+1,SEND_HOST_REQUEST,arguments);
  }else{        //if nodeID mode =ASLEEP, BUSY,(EVICTING && flag==1)
                if(((addressTable[nodeID-1]->remainingSpace-
reservedSpace>=a.getAgentAddress(agentID-1)->size) && invalidFlag==0) &&
                        (addressTable[nodeID-1]->mode==ASLEEP || addressTable[nodeID-1]->mode==BUSY
|| (addressTable[nodeID-1]->mode==EVICTING))){//case 1: migration is possible
                        a.getAgentAddress(agentID-1)->attemptCounter.rmEntry(nodeID);
//********-------------********
                        if(addressTable[parentNode]->mode!=EVICTING){
                                addressTable[parentNode]->mode=MIGRATING;               //set
parent to migrating so if some agent send hostRequest to this node he will have to resend the
request
                        }

                        if(flag3==1 && addressTable[nodeID-1]->agentBeingServed==INVALID)
//αν ο nodeID έχει nodeBeingServed τον parentNode και δεν τον απασχολεί κάποιος συγκάτοικος
18/11/2015
                                addressTable[nodeID-1]->agentBeingServed=agentID;
                        if(addressTable[nodeID-1]->mode==ASLEEP){
                                addressTable[nodeID-1]->agentBeingServed=agentID;
                                addressTable[nodeID-1]->nodeBeingServed=parentNode+1;
                                addressTable[nodeID-1]->mode=MIGRATING;
                        }
                        if(addressTable[nodeID-1]->reservations.contains(agentID)==-1)  //αν ο
nodeID δεν έχει ήδη δεσμεύσει χώρο για τον agentID. 8/11/2015
                                addressTable[nodeID-1]-
>reservations.addEntry(agentID,a.getAgentAddress(agentID-1)->size);
                        arguments.init();
      arguments.addEntry(agentID,0);
      arguments.addEntry(nodeID,0);
      arguments.addEntry(benefit,0);
      h.addToHip(timeHip+1,MIGRATE,arguments);
    }else if(addressTable[nodeID-1]->size>=a.getAgentAddress(agentID-1)->size  && flag!=1 &&
invalidFlag==0
                        && (addressTable[nodeID-1]->mode==ASLEEP || (addressTable[nodeID-1]-
>mode==EVICTING && flag3!=0))){     //case 2: migrations might be possible after evictions
                        a.getAgentAddress(agentID-1)->attemptCounter.rmEntry(nodeID);
//********-------------********
                        addressTable[nodeID-1]->agentsChecked=0;        //initialize agentsChecked
      addressTable[nodeID-1]->benefitsArray.init();       //initialize array with the maximum
benefit of each agent
      a.getAgentAddress(agentID-1)->agentsOccupied.init();
      for(i=0;i<numberOfAgents;i++){ //for each agent in nodeID
                                currentAgent=addressTable[nodeID-1]->agentList.getNodeID(i);
                                if(a.getAgentAddress(currentAgent-1)->state==FREE &&
a.getAgentAddress(currentAgent-1)->childrenList.getLength()>0
```

```
                                        && a.getAgentAddress(currentAgent-1)->parentAgent==INVALID){
//avoid trying to MIGRATE BUSY && node specific agents
                                                n.initializeAgent(currentAgent);
                                                numberOfNodes=addressTable[nodeID-1]-
>childrenList.getLength();
                                                for(j=0;j<numberOfNodes;j++){   //for every node
that connects with the current
                                                        currentNode=addressTable[nodeID-1]-
>childrenList.getNodeID(j);
                                                penalty=getLoad(currentAgent,nodeID)-
getLoad(currentAgent,currentNode);  //estimate penalty of such migration
                                                flag=0;
                                                if(addressTable[currentNode-1]-
>nodeBeingServed==nodeID){       //if currentNode already serves an agent of nodeID
                                                                flag=1;
                                                }
                                                //-----------------------------------------
-------------------------------------------------------------

                                                if(benefit+penalty>0 &&
addressTable[currentNode-1]->mode!=PRE_EVICTING){       //and if it is Beneficial to keep doing
evictions
                                                        a.getAgentAddress(currentAgent-1)-
>nodesChecked++;      //increment number of nodes that have been checked and they could lead to a
possible eviction
                                                        a.getAgentAddress(currentAgent-1)-
>state=EVICTION;      //make agent unavailable to MIGRATE
                                                        a.getAgentAddress(currentAgent-1)-
>benefit=benefit;
                                                        a.getAgentAddress(currentAgent-1)-
>parentAgent=agentID;
                                                        arguments.init();
                                                        arguments.addEntry(currentAgent,0);
                                                        arguments.addEntry(currentNode,0);

arguments.addEntry(benefit+penalty,0);
                                                        arguments.addEntry(agentID,0);

h.addToHip(timeHip+currentTimer,SEND_HOST_REQUEST,arguments);   //send host request at the following
timeHip
                                                        previousTimer=currentTimer;
                                                }
                                                }
                                        }
                                        if(previousTimer==currentTimer){        //if at least one
hostRequest was sent for this agent
                                                a.getAgentAddress(agentID-1)-
>agentsOccupied.addEntry(currentAgent,0);  //add current agent to the list of agents that agentID
keeps occupied
                                                if(addressTable[parentNode]->nodeBeingServed!
=INVALID)  //if agentID is not the rootAgent
                                                        addressTable[parentNode]->mode=EVICTING;
                                                addressTable[nodeID-1]->agentsChecked++;
                                                currentTimer++;         //send next host request to
the next timeHip
                                        }
                                }
        if(previousTimer==0){     //if size of node was not enough and evictions couldn't be made
//ATTENTION: add condition to check if total size of possible to evict agents is greater or equal to
agentID->size.
//if not no reason to proceed to evictions
                                if(addressTable[nodeID-1]->mode==ASLEEP)
                                        addressTable[nodeID-1]->nodeBeingServed=parentNode+1;
                                addressTable[nodeID-1]->agentBeingServed=agentID;
                                if(PEflag==1){  //in case parentNode has sent host requests to other
nodes as well, apart from nodeID, so it needs to be in mode=EVICTING//28/11/2015
                                        addressTable[parentNode]->mode=EVICTING;
                                }
                                arguments.init();
                                arguments.addEntry(agentID,0);
                                arguments.addEntry(nodeID,0);
```

```
                                arguments.addEntry(INVALID,0);
                                h.addToHip(timeHip+currentTimer,ANSWER,arguments);
      }else{    //if at least one eviction was possible
                                addressTable[nodeID-1]->nodeBeingServed=parentNode+1;
                                addressTable[nodeID-1]->agentBeingServed=agentID;
//set this node to serve agentID
                                if(addressTable[nodeID-1]->mode==ASLEEP){  //********--------------
********
                                        addressTable[nodeID-1]->mode=PRE_EVICTING;
                                        //cout<<nodeID<<ENDLINE;
                                }
                        }
    }else{  //case 3: if agentID can't fit in nodeID's space even after evictions
                        if(addressTable[parentNode]->nodeBeingServed==INVALID &&
a.getAgentAddress(agentID-1)->isRootAgent==1){ //δηλαδή ο agentID είναι rootAgent. Λύση conflict 9.
18/11/2015
                                addressTable[parentNode]->mode=MIGRATING;
                        }
                        if(addressTable[nodeID-1]->mode==ASLEEP){
                                addressTable[nodeID-1]->mode=MIGRATING;
                                addressTable[nodeID-1]->nodeBeingServed=parentNode+1;
                        }
                        if(PEflag==1){  //in case parentNode has sent host requests to other nodes
as well, apart from nodeID, so it needs to be in mode=EVICTING
                                addressTable[parentNode]->mode=EVICTING;
                        }
                        arguments.init();
      arguments.addEntry(agentID,0);
      arguments.addEntry(nodeID,0);
      arguments.addEntry(INVALID,0);                    //send INVALID ANSWER
      h.addToHip(timeHip+currentTimer,ANSWER,arguments);
    }
  }
}

void network::simulation(){
 dynamicList arguments;
 arguments.init();
 int iterations,random, i,j,k,l,numberOfAgents,agentID,pAgentID;//parentAgent ID
 int currentBenefit,movingLoad,traffic,hops,length,flag;
 int nodeID,mostBeneficialAgent,mostBeneficialNetworkNode;
 int timer=0,benefit,currentNode,maxBenefit,localBenefit,parentNode;
 int nodesToWake=4;
 int used[nodesToWake*2];
srand(1);
  while(timer<=RUNTIME){        //number of times to repeat scanning of network
    //===========find all possible benefits==========
                h.printHipQueue(timer);
    //====excute all comands in hip queue that are to be executed at this timeHip=====
     h.executeHipCommands(timer);
                //===========then=============
                for(i=0;i<nodesToWake*2;i++)
                        used[i]=-1;
   //=============wake up and nodes and send hostRequests==============
    for(i=0;i<nodesToWake;i++){ //number nodes to wake
      random=rand()%MAX_NODES;  //random+1=current network node ID
      maxBenefit=INVALID; currentNode=random; mostBeneficialAgent=-1;
numberOfAgents=addressTable[currentNode]->agentList.getLength();
                        flag=0;
                        for(j=0;j<nodesToWake;j++){     //23/11/2015
                                if(used[j]==random+1)
                                        flag=1;
                        }
                        if(addressTable[currentNode]->mode==ASLEEP && flag==0){
                                addressTable[currentNode]->agentsYetToCheck.init();
                                for(j=0;j<numberOfAgents;j++){  //scann all agents contained in
node with nodeID=currentNode
                                        agentID=addressTable[currentNode]->agentList.getNodeID(j);
                                        if(a.getAgentAddress(agentID-1)->childrenList.getLength()>0
&& a.getAgentAddress(agentID-1)->state==FREE
                                        && a.getAgentAddress(agentID-1)-
>previousNodes.getLength()==0){//if it is a node neutral agent
                                                localBenefit=INVALID;
                                                for(k=0;k<addressTable[currentNode]-
>childrenList.getLength();k++){//for every connecting network node
```

```
                                                        nodeID=addressTable[currentNode]-
>childrenList.getNodeID(k);    //connecting network node ID
                                                        benefit=getLoad(agentID,currentNode+1)-
getLoad(agentID,nodeID); //find benefit of migration
                                flag=0;
                        for(l=0;l<nodesToWake*2;l++){
                                if(used[l]==nodeID)
                                        flag=1;
                        }
                                                        if(benefit>localBenefit && flag==0)
//find maximum benefit among agentID's possible migrations to connecting network nodes
                                                                localBenefit=benefit;
                                                        //--------decide which migration is the most
Beneficial---------
                                                        if(benefit>maxBenefit   && flag==0){
//find maximum benefit among all agentIDs' migrations
                                                                maxBenefit=benefit;
                                                                mostBeneficialAgent=agentID;
                                                                mostBeneficialNetworkNode=nodeID;
                                                        }
                                                        //-----------------------------------------
--------------------
                                                }
                                                addressTable[currentNode]-
>agentsYetToCheck.addEntryInOrder(agentID,localBenefit);              //?????????
                                        }
                                }
                                if(maxBenefit>0){               //if currentNode contains a
beneficial to MIGRATE agent

                                        //-------------initialize that agent's
variables-----------------
                                        n.initializeAgent(mostBeneficialAgent);
                                        a.getAgentAddress(mostBeneficialAgent-1)->state=BUSY;
                                        a.getAgentAddress(mostBeneficialAgent-1)->nodesChecked=1;
                                        a.getAgentAddress(mostBeneficialAgent-1)-
>mostBeneficialNode=currentNode;
                                        a.getAgentAddress(mostBeneficialAgent-1)->isRootAgent=1;
                                        //-----------------------------------------------------------
-------

                                        if(addressTable[mostBeneficialNetworkNode-1]->mode==ASLEEP)
//********-------------********
                                                addressTable[mostBeneficialNetworkNode-1]-
>mode=AWAKE;
                                        //----send Host request from
mostBeneficialAgent--------------
                                        arguments.init();
                                        arguments.addEntry(mostBeneficialAgent,0);
                                        arguments.addEntry(mostBeneficialNetworkNode,0);
                                        arguments.addEntry(maxBenefit,0);
                                        arguments.addEntry(40,0);
                                        h.addToHip(timer+1,SEND_HOST_REQUEST,arguments); //send
command to hip queue
                                        //-----------------------------------------------------------
--
                                        used[i]=mostBeneficialNetworkNode;
                                        used[i+nodesToWake]=currentNode+1;
                                        n.initializeNode(currentNode+1);
                                        addressTable[currentNode]->mode=BUSY;
                                        addressTable[currentNode]->agentsChecked=1;
                                }
                        }
                }
    //=================================================================

    timer++;
  }
}

int network::getTotalLoad(){
```

```
 int i,j,sum=0;
 for(i=0;i<MAX_NODES;i++){                                          //for each node
    for(j=0;j<addressTable[i]->agentList.getLength();j++){          //for each agent in the
node
      sum+=getLoad(addressTable[i]->agentList.getNodeID(j),i+1);   //add it's load to sum
    }
  }
  return(sum);
}

int network::getTotalMigrations(){
  int i,result=0;
  for(i=0;i<a.getNumberOfElements();i++){
    result+=a.getAgentAddress(i)->migrations;
     //cout<<i+1<<" MIGRATEd "<<a.getAgentAddress(i)->migrations<<" times!\n";
  }
  result=result;
  return(result);
}

void testNetwork(){
  int i;
    for(i=0;i<a.getNumberOfElements();i++){
      cout<<i<<" size="<<a.getAgentAddress(i)->size<<"\n";
    }
}

#endif
```

Network Node Class and Relative Functions for Network Flooding, SAM WITH NEGATIVE EVICTIONS algorithm. File: networkNodeHeaderV2.h

```
#include</appTree.h>
#include<time.h>        //definition of time
#include<stdlib.h>      //definition of rand, srand, malloc,free,atoi
#include<stddef.h>      //definition of NULL
#include<math.h>        //sqrt function
#include<iostream>      //definition of cout
#include<iomanip>       //setw()
#include<string>
#include<fstream>       //file handling functions


#ifndef WHAT_H_INCLUDED

//===========DEFINITIONS====================
#define MAX_NODES 20
#define ENDLINE "================================================\n"
#define TESTLINE "test============================================\n"
#define INVALID -1
#define RUNTIME 100000

#define AVG_SIZE_MULTIPLE 3
```

```
//------agent state definitions-------
#define FREE 0
#define BUSY 1
#define EVICTION 2

//----node mode definitions-----------
#define ASLEEP -1
#define AWAKE 0
#define BUSY 1
#define MIGRATING 2
#define EVICTING 3 //node is available for those agents in the node hosting agentBeingServed
#define PRE_EVICTING 4
#define PRE_ASLEEP 5

//----Hip commands definitions-----------
#define SEND_HOST_REQUEST 0
#define MIGRATE 1
#define MOVE_AGENT 2
#define ANSWER 3
#define EVICT_MIGRATE 4
#define SETMODE 5
#define SET_STATE 6
#define NEXT_MODE 7
#define RESTORATION 8
#define SET_AB_SERVED 9
//=========================================

#define WHAT_H_INCLUDED

using namespace std;

string fileName="testFile";

appTree a;

//*******NETWORK NODE DEFINITION*****************
typedef struct NETWORK_NODE{
  int ID;
  int size;
  int remainingSpace;
  int mode;
  int agentsChecked;
  int agentBeingServed;
        int nodeBeingServed;
        int isMostBeneficialFor;
        dynamicList reservations;
  dynamicList benefitsArray;
  dynamicList agentList;
  dynamicList childrenList;
        dynamicList agentsYetToCheck;
        dynamicList nodesOccupied;
}* netNode;

class network{
    netNode addressTable[MAX_NODES];    //hash array of nodes addresses
    int outputValue;
  public:
    network();
                void initializeAgent(int agentID);
                void initializeNode(int nodeID);
    void setupNetwork();
    void printNetwork();
    void simulation();
    int getNumberOfHops(int startID,int parentID,int destinationID,int result);
    int getLoad(int agentID,int nodeID);
    int calculateRemainingSpace(int nodeID);
    void sendHostRequestNFL(int agentID,int nodeID,int benefit,int timeHip);
    void moveAgentToNode(int agentID,int nodeID,int timeHip);
    void notifyNode(int agentID,int nodeID,int benefit,int timeHip);
    void migrationCheck(int agentID,int nodeID,int benefit,int timeHip);
    void checkForEvictions(int agentID,int nodeID,int timeHip);
```

```
    int getTotalLoad();
    int getTotalMigrations();
                void printNodeValues(int nodeID);
                void setOutputValue(int value){outputValue=value;};
                void printOutputValue(){cout<<outputValue<<ENDLINE;};
};

network n;

//*********HIP CLASS DEFINITION*****************
typedef struct HIP_NODE{
  int timeHip;  //timeHip
  int command;  //command to execute. SEND_HOST_REQUEST
  dynamicList args;     //
  struct HIP_NODE* next;
}* hipNode;

class hipClass{
    hipNode hip;
  public:
    void init();
    void addToHip(int timeHipValue,int commandValue,dynamicList arguments);
    void executeHipCommands(int time);
    void printHipQueue(int time);
};

hipClass h;

void hipClass::init(){
  hip=NULL;
}

void hipClass::addToHip(int timeHipValue,int commandValue,dynamicList arguments){
//add command to hip queue in order of timeHip
  hipNode temp,current,previous;
  //=====initialize new hipNode=================
  temp=(hipNode)malloc(sizeof(struct HIP_NODE));
  temp->timeHip=timeHipValue;
  temp->command=commandValue;
  temp->args=arguments;
  temp->next=NULL;
  //============================================

  if(hip==NULL){
    hip=temp;
  }else{
    current=hip;
    previous=hip;
    while(current!=NULL && current->timeHip<timeHipValue ){
      if(current!=hip)
        previous=current;
      current=current->next;
    }
    if(current==hip){ //if it is to be inserted at the beggining of the list
      temp->next=hip;
      hip=temp;
    }else{              //else isert between previous and current
      previous->next=temp;
      temp->next=current;
    }
  }
}

void hipClass::executeHipCommands(int time){
//execute all comands in hip queue that are to be executed in timeHip==time
 hipNode current,previous;
  current=hip;
  previous=hip;
if(current!=NULL){
  //---------reach timehip commands---------------
  while(current->next!=NULL && (current->timeHip < time) ){
    if(current!=hip)
      previous=current;
                current=current->next;
  }
```

```
if(current->timeHip>=time){
  //--------execute them-------------------------
  while(current!=NULL && current->timeHip==time){
    if(current!=hip)
      previous=current;

    //==============execute command and remove it from hip queue============================
    if(current->command==SEND_HOST_REQUEST){ //case 1: command for SEND_HOST_REQUEST
      n.sendHostRequestNFL(current->args.getNodeID(0),current->args.getNodeID(1),current-
>args.getNodeID(2),time);
    }else if(current->command==MIGRATE){        //case 2: command for migration
      n.migrationCheck(current->args.getNodeID(0),current->args.getNodeID(1),current-
>args.getNodeID(2),time);
    }else if(current->command==MOVE_AGENT){     //case 3: command move agent to node
      n.moveAgentToNode(current->args.getNodeID(0),current->args.getNodeID(1),time);
    }else if(current->command==ANSWER){
      n.notifyNode(current->args.getNodeID(0),current->args.getNodeID(1),current-
>args.getNodeID(2),time);
                }

    //-----------------remove it from hip queue---------------------------------
    if(current==hip){
      hip=hip->next;
      free(current);
      previous=hip;
      current=hip;
    }else{
      current=current->next;
      previous->next=current;
    }
  }
}
}
}


void hipClass::printHipQueue(int time){
  hipNode current;
  if(hip==NULL){
    cout<<"no commands in hip at time: "<<time<<"\n";
  }else{
    current=hip;
    while(current->next!=NULL && current->timeHip < time)
      current=current->next;
    cout<<"time: "<<time<<",commands:\n";
    while(current->next!=NULL && current->timeHip==time){
      cout<<current->command<<" | ";
      current->args.printArray(0);
      current=current->next;
    }
    if(current->timeHip==time){
      cout<<current->command<<" | ";
      current->args.printArray(0);
    }
  }
}

network::network(){
 int i,pos,offset,X,Y,Z,lastEntryID=-1;
 string line,temp;
 ifstream myfile(fileName.c_str());

 for(i=0;i<MAX_NODES;i++)       //initialize addressTable
   addressTable[i]=NULL;
 do{
   while(getline(myfile,line) && line!="#" && line!="$"){       //read results of netork generation
algorith from file 'myfile'
    //====line consists of X-Y:Z=======
    //get X
    pos=(int)line.find("-");
    temp=line.substr(0,pos);
```

```
    X=atoi(temp.c_str());
    if(addressTable[X-1]==NULL){                //if a node with this ID was not created
      addressTable[X-1]=(netNode)malloc(sizeof(struct NETWORK_NODE));   //create
      addressTable[X-1]->ID=X;                                          //and initialize it
      addressTable[X-1]->size=0;
      addressTable[X-1]->agentList.init();
      addressTable[X-1]->childrenList.init();
    }

    if(lastEntryID!=-1){                //if there were more than one minimum spanning tree formed
(see Kruskal's algorith)
      addressTable[lastEntryID-1]->childrenList.addEntry(X,10); //add the current spanning tree to
a node of the previous one
      lastEntryID=-1;
    }


    //get Y
    offset=pos+1;
    pos=(int)line.find(":");
    temp=line.substr(offset,pos);
    Y=atoi(temp.c_str());
    if(addressTable[Y-1]==NULL){                //if a node with this ID was not created
      addressTable[Y-1]=(netNode)malloc(sizeof(struct NETWORK_NODE));   //create
      addressTable[Y-1]->ID=Y;                                          //and initialize it
      addressTable[Y-1]->size=0;
      addressTable[Y-1]->agentList.init();
      addressTable[Y-1]->childrenList.init();
    }

    //get Z
    offset=pos+1;
    pos=(int)line.length();
    temp=line.substr(offset,pos);
    Z=atoi(temp.c_str());

    addressTable[X-1]->childrenList.addEntry(Y,Z);
    addressTable[Y-1]->childrenList.addEntry(X,Z);
    }

    lastEntryID=Y;
 }while(line!="$");

 for(i=0;i<MAX_NODES;i++){
    if(addressTable[i]==NULL){//if there are nodes that are not part of the tree
      addressTable[i]=(netNode)malloc(sizeof(struct NETWORK_NODE));     //create
      addressTable[i]->ID=i+1;                                          //and initialize it
      addressTable[i]->size=0;
      addressTable[i]->agentList.init();
      addressTable[i]->childrenList.init();
      addressTable[lastEntryID-1]->childrenList.addEntry((i+1),10);     //add it to the tree
      addressTable[i]->childrenList.addEntry(lastEntryID,10);
  }
  //addressTable[i]->state=SLEEP;       //put all nodes to sleep
 }

 myfile.close();


/*
 //=========print addressTable===========
 for(i=0;i<MAX_NODES;i++){
   cout<<"ID="<<i+1<<": ";
   if(addressTable[i]!=NULL){ //=======put all nodes to sleep=================
    addressTable[i]->childrenList.printArray(1);
   }
   else{
     cout<<"NO!\n";
   }
 }*/
}

int network::calculateRemainingSpace(int nodeID){
//calculates the remaining space
 int i,totalSize=0,length,chID;
```

```
  length=addressTable[nodeID-1]->agentList.getLength();
  for(i=0;i<length;i++){  //for each agent in network nodeID
    chID=addressTable[nodeID-1]->agentList.getNodeID(i);
    totalSize+=a.getAgentAddress(chID-1)->size;
  }
  return(addressTable[nodeID-1]->size-totalSize);
}

void network::printNodeValues(int nodeID){
 int reservedSpace,i;
        nodeID--;
        cout<<"================================================\n";
  cout<<"for node "<<nodeID+1<<":\n";
  cout<<"size="<<addressTable[nodeID]->size<<"\n";
  cout<<"remaining space="<<addressTable[nodeID]->remainingSpace<<"\n";
        reservedSpace=0;
        for(i=0;i<addressTable[nodeID-1]->reservations.getLength();i++)
                reservedSpace+=addressTable[nodeID-1]->reservations.getNodeDistance(i);
        cout<<"reservedSpace="<<reservedSpace<<"\n";
  if(addressTable[nodeID]->mode==ASLEEP)
                cout<<"mode=ASLEEP\n";
        if(addressTable[nodeID]->mode==AWAKE)
                cout<<"mode=AWAKE\n";
        if(addressTable[nodeID]->mode==BUSY)
                cout<<"mode=BUSY\n";
        if(addressTable[nodeID]->mode==MIGRATING)
                cout<<"mode=MIGRATING\n";
        if(addressTable[nodeID]->mode==EVICTING)
                cout<<"mode=EVICTING\n";
        if(addressTable[nodeID]->mode==PRE_EVICTING)
                cout<<"mode=PRE_EVICTING\n";
  cout<<"agentsChecked="<<addressTable[nodeID]->agentsChecked<<"\n";
        cout<<"nodeBeingServed="<<addressTable[nodeID]->nodeBeingServed<<"\n";
  cout<<"agentBeingServed="<<addressTable[nodeID]->agentBeingServed<<"\n";
  cout<<"benefitsArray length="<<addressTable[nodeID]->benefitsArray.getLength()<<"\n";
        cout<<"agentsYetToChecklength="<<addressTable[nodeID]->agentsYetToCheck.getLength()<<"\n";
        cout<<"================================================\n";
}

void network::setupNetwork(){
 int i,random,j,totalSize=0;
 int currentAgent;
 srand(1);
    a.getAgentAddress(i)->totalBenefit=0;
                n.setOutputValue(INVALID);
  for(i=0;i<a.getNumberOfElements();i++){
    random=rand()%MAX_NODES;
    addressTable[random]->agentList.addEntry(i+1,0);
    addressTable[random]->size+=a.getAgentAddress(i)->size;
    totalSize+=a.getAgentAddress(i)->size;
    a.getAgentAddress(i)->agentsOccupied.init();
  }

  for(i=0;i<MAX_NODES;i++){
      addressTable[i]->size+=AVG_SIZE_MULTIPLE*totalSize/a.getNumberOfElements();       //set
network nodes size=size+average agent size
      addressTable[i]->remainingSpace=calculateRemainingSpace(i+1);
      addressTable[i]->mode=ASLEEP;
      addressTable[i]->agentsChecked=0;
                        addressTable[i]->benefitsArray.init();
                        addressTable[i]->agentBeingServed=INVALID;
                        addressTable[i]->agentsYetToCheck.init();
                        addressTable[i]->nodesOccupied.init();
                        addressTable[i]->reservations.init();
      //---------initialize agent parameters----------------------
      for(j=0;j<addressTable[i]->agentList.getLength();j++){    //for each agent in currentNode
                                currentAgent=addressTable[i]->agentList.getNodeID(j);
                                a.getAgentAddress(currentAgent-1)->attemptCounter.init();
                                a.getAgentAddress(currentAgent-1)->nodesChecked=0;
                                a.getAgentAddress(currentAgent-1)->isRootAgent=0;
                                a.getAgentAddress(currentAgent-1)->restorationFlag=0;
```

```
                                a.getAgentAddress(currentAgent-1)->parentAgent=INVALID;
                                a.getAgentAddress(currentAgent-1)->mostBeneficialNode=INVALID;
                                a.getAgentAddress(currentAgent-1)->maxBenefit=INVALID;
                                a.getAgentAddress(currentAgent-1)->agentBeingChecked=INVALID;
                                a.getAgentAddress(currentAgent-1)->totalBenefit=INVALID;
                                a.getAgentAddress(currentAgent-1)->agentsOccupied.init();
                                a.getAgentAddress(currentAgent-1)->migrationsAttempted.init();
                                a.getAgentAddress(currentAgent-1)->tempEvictions.init();
                                a.getAgentAddress(currentAgent-1)->benefitsArray.init();
                                a.getAgentAddress(currentAgent-1)->evictionsChecked.init();
                                a.getAgentAddress(currentAgent-1)->nodesToRestore.init();
                        }
  }
}

void network::printNetwork(){
  int i;
  for(i=0;i<MAX_NODES;i++){
    if(addressTable[i]->agentList.getLength()!=0){
      cout<<"node "<<i+1<<" has size:"<<addressTable[i]->size<<" and includes agents: ";
      addressTable[i]->agentList.printArray(0);
    }else
      cout<<"node "<<i+1<<" has size:"<<addressTable[i]->size<<" and includes no agents";
    cout<<" and connects with the nodes: ";
    addressTable[i]->childrenList.printArray(1);
    cout<<ENDLINE;

  }
}

int network::getNumberOfHops(int startID,int parentID,int destinationID,int result){
//returns the number of hops needed to reach destinationID starting from startID
  int pos,i,tempResult=0;
 if(startID==destinationID){
    return(0);
  }else if((addressTable[startID-1]->childrenList.getLength()==1)&&(addressTable[startID-1]-
>childrenList.getNodeID(0)==parentID)){//if we reached a node only connected to it's parent
    return(0);//this node does not connect with the one we are looking for, so return 0
        }else{                      //else
    pos=addressTable[startID-1]->childrenList.contains(destinationID);
    if(pos!=-1){        //if destinationID node is startID's children
      return(result+1);
    }else{              //else
      for(i=0;i<addressTable[startID-1]->childrenList.getLength();i++){ //keep searching in this
node's childrenList
                                if(addressTable[startID-1]->childrenList.getNodeID(i)!=parentID){
//for all nodes except parent
                                        tempResult+=getNumberOfHops(addressTable[startID-1]-
>childrenList.getNodeID(i),startID,destinationID,result+1);
                                }
      }
      if(tempResult==0) //if destinationID was not found in childrenList
                                return(0);
      else
                                return(tempResult);
    }
  }
}

int network::getLoad(int agentID,int nodeID){
//returns load if agentID is placed in nodeID
 int i,pAgentID,k,hops,load=0,traffic,length,chID;

 //========find load associated with parent agent===========
  pAgentID=a.getParentID(agentID);      //find parentAgent ID
  if(pAgentID==0)       //if it is the root agent
    traffic=0;
  else{                         //if it is not the root agent
    traffic=a.getAgentAddress(pAgentID-1)->childrenList.getNodeDistance2(agentID);//get traffic
between current agent and it's parent

    //-------find network node that contains pAgent--------
    k=0;
    while(k<MAX_NODES && addressTable[k]->agentList.contains(pAgentID)==-1)//scann network nodes
untill pAgentID is found
```

```
      k++;
    k++;        //k shows the ID of the network node that contains pAgentID
    //-----------------------------------------------------
    hops=getNumberOfHops(nodeID,-1,k,0);
    load+=traffic*hops;
  }
  //============================================================


  //=======find load associated with children agents========
  length=a.getAgentAddress(agentID-1)->childrenList.getLength();
  for(i=0;i<length;i++){ //for each child of agentID in appTree
    chID=a.getAgentAddress(agentID-1)->childrenList.getNodeID(i);

    //-------find network node that contains child agent--------
    k=0;
    while(k<MAX_NODES && addressTable[k]->agentList.contains(chID)==-1)//scann network nodes untill
pAgentID is found
      k++;
    k++;        //k shows the ID of the network node that contains chID
    //-----------------------------------------------------

    hops=getNumberOfHops(nodeID,-1,k,0);
    traffic=a.getAgentAddress(agentID-1)->childrenList.getNodeDistance(i);
    load+=traffic*hops;
  }
  //============================================================
  return(load);
}

void network::initializeAgent(int agentID){
        agentID=agentID-1;
  a.getAgentAddress(agentID)->state=FREE; //FREE, BUSY, EVICTION
  a.getAgentAddress(agentID)->nodesChecked=0;
  a.getAgentAddress(agentID)->restorationFlag=0;
  a.getAgentAddress(agentID)->maxBenefit=INVALID;
  a.getAgentAddress(agentID)->mostBeneficialNode=INVALID;
  a.getAgentAddress(agentID)->agentBeingChecked=INVALID;
  a.getAgentAddress(agentID)->sum=0;
  a.getAgentAddress(agentID)->totalBenefit=0;
  a.getAgentAddress(agentID)->benefit=0;
  a.getAgentAddress(agentID)->parentAgent=INVALID;
  //int migrations;
        a.getAgentAddress(agentID)->isRootAgent=0;
  a.getAgentAddress(agentID)->evictionsChecked.init();
  a.getAgentAddress(agentID)->tempEvictions.init();
  a.getAgentAddress(agentID)->benefitsArray.init();
  a.getAgentAddress(agentID)->evictionPath.init();
  a.getAgentAddress(agentID)->previousNodes.init();
  a.getAgentAddress(agentID)->agentsOccupied.init();
        a.getAgentAddress(agentID)->migrationsAttempted.init();
        a.getAgentAddress(agentID)->attemptCounter.init();
        a.getAgentAddress(agentID)->nodesToRestore.init();

}

void network::initializeNode(int nodeID){
        nodeID=nodeID-1;
        addressTable[nodeID]->mode=ASLEEP;
        addressTable[nodeID]->agentsChecked=0;
  addressTable[nodeID]->agentBeingServed=INVALID;
  addressTable[nodeID]->benefitsArray.init();
        addressTable[nodeID]->agentsYetToCheck.init();
        addressTable[nodeID]->nodesOccupied.init();
        addressTable[nodeID]->nodeBeingServed=INVALID;
        addressTable[nodeID]->reservations.init();
        addressTable[nodeID]->isMostBeneficialFor=INVALID;
}

void network::moveAgentToNode(int agentID,int nodeID,int timeHip){
//moves agent to node and removes from nodes remainingSpace agentID's size
```

```
 int parentNode=0,i;
 dynamicList tempList;
 tempList.init();
   a.getAgentAddress(agentID-1)->migrations++;
    //-----find current network node hosting agentID---------
    while(addressTable[parentNode]->agentList.contains(agentID)==-1)
      parentNode++;
    //-----------------------------------------------------
                if(parentNode==nodeID-1)
                        a.getAgentAddress(agentID-1)->migrations--;
    if(addressTable[parentNode]->nodeBeingServed==INVALID && a.getAgentAddress(agentID-1)-
>isRootAgent==1){     //if this is the node who send the initial host request.

                        for(i=0;i<addressTable[parentNode]->nodesOccupied.getLength();i++){
                                n.initializeNode(addressTable[parentNode]-
>nodesOccupied.getNodeID(i));
                        }
                        if((addressTable[nodeID-1]->mode==ASLEEP
                        ||((addressTable[nodeID-1]->mode==EVICTING  || addressTable[nodeID-1]-
>mode==MIGRATING) && addressTable[nodeID-1]->nodeBeingServed==parentNode+1))
                        && parentNode!=nodeID-1){       //αν ο nodeID δεν περιέχει κάποιον
rootAgent.
                                n.initializeNode(nodeID);
                        }
                        //---------------------------------
                        n.initializeNode(parentNode+1);/
                }else{
                        if(a.getAgentAddress(agentID-1)->previousNodes.contains(nodeID)!=-1){
//if it is moving to a path back to it's initial position
                                a.getAgentAddress(agentID-1)-
>previousNodes.rmEntry(nodeID);//remove it from previousNodes list
                        }else{
                                a.getAgentAddress(agentID-1)-
>previousNodes.addEntry(parentNode+1,0);
                        }
                }
        //=========delete agentID from current network node=============
    addressTable[parentNode]->agentList.rmEntry(agentID);       //remove agentID from this network
node
    addressTable[parentNode]->remainingSpace+=a.getAgentAddress(agentID-1)->size; //free the space
it was occupying in network node
  //================================================================

  //===========add agentID to network node=======================
    addressTable[nodeID-1]->agentList.addEntry(agentID,0); //add agentID to nodeID's agent list
    addressTable[nodeID-1]->remainingSpace-=a.getAgentAddress(agentID-1)->size;//reduce nodeID's
remaing space by agentID's size
    addressTable[nodeID-1]->reservations.rmEntry(agentID);
  //============================================================
        if(a.getAgentAddress(agentID-1)->previousNodes.getLength()>0){
                i=a.getAgentAddress(agentID-1)->parentAgent;
                tempList=a.getAgentAddress(agentID-1)->previousNodes;
                n.initializeAgent(agentID);
                a.getAgentAddress(agentID-1)->parentAgent=i;
                a.getAgentAddress(agentID-1)->previousNodes=tempList;
        }else{
                n.initializeAgent(agentID);
        }
}

void network::checkForEvictions(int agentID,int nodeID,int timeHip){
  //have to initialize sum=0 and agentBeingChecked=-1,tempEvictions
  int
currentAgent,i,j,parentNode=0,currentNode,position,flag,aBCheckedBenefit,penalty,tempParentNode;
  int timer=timeHip+1;
        int tempCurrentNode,reservedSpace;
  int aBChecked=a.getAgentAddress(agentID-1)->agentBeingChecked;
  dynamicList arguments;
//********------------********
  while(addressTable[parentNode]->agentList.contains(agentID)==-1)
    parentNode++;
        if(addressTable[parentNode]->mode!=BUSY)
                addressTable[parentNode]->mode=EVICTING;
        if(addressTable[nodeID-1]->mode!=BUSY)//********------------********
                addressTable[nodeID-1]->mode=EVICTING;
```

```
  position=a.getAgentAddress(agentID-1)->tempEvictions.contains2(aBChecked,nodeID);

        if(position!=-1){       //if agentBeingChecked is in tempEvictions
                a.getAgentAddress(agentID-1)->sum+=a.getAgentAddress(aBChecked-1)->size;
                if(a.getAgentAddress(aBChecked-1)->mostBeneficialNode==parentNode+1){   //αν
εκτοπίζεται κόμβος στη δικιά μου θέση.
                        if(addressTable[parentNode]->agentBeingServed!=INVALID)
//πρέπει να ενημερώσω τον agentBeingServed μου για αυτό
                                a.getAgentAddress(addressTable[parentNode]->agentBeingServed-1)-
>sum-=a.getAgentAddress(aBChecked-1)->size;
                }
                        if(addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]-
>mode==EVICTING
                        && addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]-
>agentBeingServed==aBChecked && a.getAgentAddress(agentID-1)->restorationFlag==1){
                                addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]-
>agentBeingServed=INVALID;
                        }

                        if(addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]-
>reservations.contains(aBChecked)==-1)
                                addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]-
>reservations.addEntry(aBChecked,a.getAgentAddress(aBChecked-1)->size);

                        tempParentNode=0;
                        while(addressTable[tempParentNode]->agentList.contains(aBChecked)==-1)
                                tempParentNode++;

    a.getAgentAddress(agentID-1)->tempEvictions.rmEntryFrom(position);  //remove from
tempEvictionsList
    aBCheckedBenefit=a.getAgentAddress(agentID-1)->benefitsArray.getNodeDistance(position);
    a.getAgentAddress(agentID-1)->totalBenefit+=aBCheckedBenefit;
    a.getAgentAddress(agentID-1)->benefitsArray.rmEntryFrom(position);  //also remove from benefits
array

                for(i=0;i<a.getAgentAddress(aBChecked-1)->evictionsChecked.getLength();i++)
{//προσθέτω στο evictionsChecked του agentID το evictionsChecked του aBChecked
                        if(a.getAgentAddress(agentID-1)-
>evictionsChecked.contains2(a.getAgentAddress(aBChecked-1)-
>evictionsChecked.getNodeID(i),a.getAgentAddress(aBChecked-1)-
>evictionsChecked.getNodeDistance(i))==INVALID){
                                a.getAgentAddress(agentID-1)-
>evictionsChecked.addEntry(a.getAgentAddress(aBChecked-1)-
>evictionsChecked.getNodeID(i),a.getAgentAddress(aBChecked-1)->evictionsChecked.getNodeDistance(i));
                        }
                }
                for(i=0;i<a.getAgentAddress(aBChecked-1)->nodesToRestore.getLength();i++){
                        if(a.getAgentAddress(agentID-1)-
>nodesToRestore.contains(a.getAgentAddress(aBChecked-1)->nodesToRestore.getNodeID(i))==INVALID){
                                a.getAgentAddress(agentID-1)-
>nodesToRestore.addEntry(a.getAgentAddress(aBChecked-1)->nodesToRestore.getNodeID(i),nodeID);
                        }
                }
                if(a.getAgentAddress(agentID-1)-
>evictionsChecked.contains2(aBChecked,nodeID)==INVALID){
                        a.getAgentAddress(agentID-1)-
>evictionsChecked.addEntry(aBChecked,nodeID);//add it to the evictionsChecked list
                }
                if(a.getAgentAddress(agentID-1)->nodesToRestore.contains(nodeID)==INVALID)
                        a.getAgentAddress(agentID-1)->nodesToRestore.addEntry(nodeID,nodeID);
                if(a.getAgentAddress(agentID-1)-
>nodesToRestore.contains(a.getAgentAddress(aBChecked-1)->mostBeneficialNode)==INVALID){
                        if((addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]-
>mode!=BUSY && addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]->mode!=EVICTING
                                && addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-
1]->mode!=PRE_EVICTING)
                        ||(addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]-
>mode==EVICTING
                        && addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-1]-
>nodeBeingServed==tempParentNode+1))
                                a.getAgentAddress(agentID-1)-
```

```
>nodesToRestore.addEntry(a.getAgentAddress(aBChecked-1)->mostBeneficialNode,nodeID);
                }
                reservedSpace=0;
                for(i=0;i<addressTable[nodeID-1]->reservations.getLength();i++)
                        if(addressTable[nodeID-1]->reservations.getNodeID(i)!=agentID)
                                reservedSpace+=addressTable[nodeID-1]-
>reservations.getNodeDistance(i);
            //-----also move aBChecked to most beneficial destination-----------//moved it here
from above. 27/11/2015
                        tempCurrentNode=a.getAgentAddress(aBChecked-1)->mostBeneficialNode;
                        n.moveAgentToNode(aBChecked,tempCurrentNode,timer);
                        a.getAgentAddress(aBChecked-1)->mostBeneficialNode=tempCurrentNode;


    if((/*a.getAgentAddress(agentID-1)->sum+*/addressTable[nodeID-1]->remainingSpace-
reservedSpace)<a.getAgentAddress(agentID-1)->size){        //if still haven't freed enough space

                a.getAgentAddress(agentID-1)->agentBeingChecked=-1; //get new agentBeingChecked
                        checkForEvictions(agentID,nodeID,timer);
          }else{        //if eventually, enough space could be freed
      //-------SEND_HOST_REQUEST of agentID to nodeID--------------  //********-------------
********
                        //-------and initialize all agents from tempEvictions of agentID--------
                        //--------add nodeID's nodesOccupied list to parentNode's list------------
                        for(i=0;i<addressTable[nodeID-1]->nodesOccupied.getLength();i++){
                                if(addressTable[parentNode]-
>nodesOccupied.contains(addressTable[nodeID-1]->nodesOccupied.getNodeID(i))==-1)
                                        addressTable[parentNode]-
>nodesOccupied.addEntry(addressTable[nodeID-1]->nodesOccupied.getNodeID(i),nodeID);
                        }
                        //-------------------------------------------------------------------------
--------
                        timer++;
                        if(addressTable[parentNode]->nodeBeingServed==INVALID  &&
a.getAgentAddress(agentID-1)->isRootAgent==1){ //if this is the rootAgen.Λύση στο conflict 9.
                                arguments.init();
                                arguments.addEntry(agentID,0);
                                arguments.addEntry(nodeID,0);
                                h.addToHip(timer++,MOVE_AGENT,arguments);
//********-------------********
                                a.getAgentAddress(agentID-1)->totalBenefit=0;
                                addressTable[parentNode]->mode=MIGRATING;
                                addressTable[nodeID-1]->mode=MIGRATING;
                                if(addressTable[nodeID-1]->reservations.contains(agentID)==-1)
                                        addressTable[nodeID-1]-
>reservations.addEntry(agentID,a.getAgentAddress(agentID-1)->size);
                        }else{  //if this is not the rootAgent//********-------------********
                                if(a.getAgentAddress(agentID-1)->restorationFlag==0){
                                        for(i=a.getAgentAddress(agentID-1)-
>evictionsChecked.getLength()-1;i>=0;i--){   //for every eviction done.
                                                position=a.getAgentAddress(agentID-1)-
>nodesToRestore.contains2(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i),nodeID);
                                                if(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i)==nodeID
                                                || position!=-1){
                                                        currentAgent=a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeID(i);
                                                        //-------- restore migration of
currentAgent----------------
                                                        arguments.init();
//********-------------********
                                                        arguments.addEntry(currentAgent,0);

arguments.addEntry(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i),0);//********-------------********
                                                        h.addToHip(timer,MOVE_AGENT,arguments);
                                                        timer++;
                                                        addressTable[a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i)-1]->mode=MIGRATING;
                                                        a.getAgentAddress(currentAgent-1)-
>migrations=a.getAgentAddress(currentAgent-1)->migrations-2;

                                                        if(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i)==nodeID)
```

```
                                                                a.getAgentAddress(agentID-1)->sum-
=a.getAgentAddress(currentAgent-1)->size;
                                                        tempParentNode=0;
                                                        while(addressTable[tempParentNode]-
>agentList.contains(currentAgent)==-1)
                                                                tempParentNode++;
                                                        if((addressTable[tempParentNode]->mode!=BUSY
&& addressTable[tempParentNode]->mode!=EVICTING && addressTable[tempParentNode]->mode!=PRE_EVICTING)
                                                        ||(addressTable[tempParentNode]-
>mode==EVICTING && addressTable[tempParentNode]->nodeBeingServed==nodeID)){
                                                                addressTable[tempParentNode]-
>mode=MIGRATING;
                                                        }
                                                }
                                        }
                                        if((addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->mode!=BUSY && addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->mode!=EVICTING
                                                && addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->mode!=PRE_EVICTING)
                                        ||(addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->mode==EVICTING && addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->nodeBeingServed==nodeID)){
                                                addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->mode=MIGRATING;
                                        }
                                }

                                arguments.init();
                                arguments.addEntry(agentID,0);
                                arguments.addEntry(nodeID,0);
                                arguments.addEntry(a.getAgentAddress(agentID-1)->totalBenefit,0);
                                h.addToHip(timer++,ANSWER,arguments);   //********--------------
********
                                a.getAgentAddress(agentID-1)->totalBenefit=0;
                        }
                                for(i=0;i<a.getAgentAddress(agentID-1)->tempEvictions.getLength();i+
+){
                                        position=addressTable[parentNode]-
>nodesOccupied.contains(a.getAgentAddress(agentID-1)->tempEvictions.getNodeID(i));
                                        if(position!=-1)
                                                addressTable[parentNode]-
>nodesOccupied.rmEntryFrom(position);
                                        n.initializeAgent(a.getAgentAddress(agentID-1)-
>tempEvictions.getNodeID(i));
                                }
                                a.getAgentAddress(agentID-1)->tempEvictions.init();
                                a.getAgentAddress(agentID-1)->benefitsArray.init();
    }
//=============================================================================================================
==
  }else{  //=========================if agentBeingChecked not in tempEvictions
list==============================

//=============================================================================================================
==
    position=0;
    flag=0;
    while(position<a.getAgentAddress(agentID-1)->tempEvictions.getLength() && flag==0){ //check in
tempEvictions of agentBeingServed
                        if(a.getAgentAddress(agentID-1)-
>tempEvictions.getNodeDistance(position)==nodeID){      //if there are evictions refered to this
node
                                flag=1;
//raise flag
      }else
                                position++;
    }
    if(flag==1){                //if there are still agents in tempEvictions of agentBeingServed not
yet checked
      a.getAgentAddress(agentID-1)->agentBeingChecked=a.getAgentAddress(agentID-1)-
```

```
>tempEvictions.getNodeID(position);  //get the next agentBeingChecked from benefitsArray
      aBChecked=a.getAgentAddress(agentID-1)->agentBeingChecked;
      a.getAgentAddress(agentID-1)->tempEvictions.rmEntryFrom(position);
//and remove it from tempEvictions
      a.getAgentAddress(agentID-1)->benefitsArray.rmEntryFrom(position);
//and from benefitsArray

      penalty=getLoad(aBChecked,nodeID)-getLoad(aBChecked,a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode);  //estimate penalty of such migration
      aBCheckedBenefit=a.getAgentAddress(aBChecked-1)->benefit+penalty;
      if(aBCheckedBenefit>0){   //if it is a beneficial migration
                                arguments.init();                       //send host request
                                arguments.addEntry(aBChecked,0);
                                arguments.addEntry(a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode,0);
                                arguments.addEntry(aBCheckedBenefit,0);
                                if(addressTable[a.getAgentAddress(aBChecked-1)->mostBeneficialNode-
1]->mode!=BUSY)      //********-------------********
                                        addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->mode=EVICTING;

                                tempParentNode=0;
                                while(addressTable[tempParentNode]-
>agentList.contains(aBChecked)==-1)
                                        tempParentNode++;

                                if(a.getAgentAddress(aBChecked-1)->mostBeneficialNode!
=addressTable[tempParentNode]->nodeBeingServed){  //      αν ο mostBeneficialNode δεν περιέχει τον
agentBeingServed του tempParentNode
                                //δηλαδή αν ο aBChecked δεν στέλνει αίτηση σε προηγούμενο node
                                        if(addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->mode!=BUSY &&
                                                addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->nodeBeingServed==nodeID){
                                                addressTable[a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode-1]->agentBeingServed=aBChecked;
                                        }
                                }
                                a.getAgentAddress(aBChecked-1)->benefitsArray.init();
                                a.getAgentAddress(aBChecked-1)->tempEvictions.init();
                                a.getAgentAddress(aBChecked-1)->restorationFlag=1;
                                h.addToHip(timer,SEND_HOST_REQUEST,arguments);  //get it's benefit

      }else{                                                    //if it is not a beneficial
migration
                                arguments.init();       //send invalid reply
                                arguments.addEntry(aBChecked,0);
                                arguments.addEntry(a.getAgentAddress(aBChecked-1)-
>mostBeneficialNode,0);
                                arguments.addEntry(INVALID,0);
                                h.addToHip(timer,ANSWER,arguments);
      }
    }else{// if((a.getAgentAddress(agentID-1)->sum+addressTable[nodeID-1]-
>remainingSpace)<a.getAgentAddress(agentID-1)->size){         //if enough space couldn't eventually
be freed
                        i=a.getAgentAddress(agentID-1)->evictionsChecked.getLength();
                        while(i>=0){    //for every eviction done.
                                position=a.getAgentAddress(agentID-1)-
>nodesToRestore.contains2(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i),nodeID);
                                if(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i)==nodeID
                                || position!=-1){
                                        currentAgent=a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeID(i);
                                        //-------- restore migration of
currentAgent----------------
                                        arguments.init();       //********-------------********
                                        arguments.addEntry(currentAgent,0);
                                        arguments.addEntry(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i),0);//********-------------********
                                        h.addToHip(timer,MOVE_AGENT,arguments);
                                        timer++;
a.getAgentAddress(currentAgent-1)->migrations=a.getAgentAddress(currentAgent-1)->migrations-2;
//25/1/2016
```

```
                                        addressTable[a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i)-1]->mode=MIGRATING;

                                        if(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i)==nodeID)
                                                a.getAgentAddress(agentID-1)->sum-
=a.getAgentAddress(currentAgent-1)->size;
                                        tempParentNode=0;
                                        while(addressTable[tempParentNode]-
>agentList.contains(currentAgent)==-1)
                                                tempParentNode++;
                                        if((addressTable[tempParentNode]->mode!=BUSY    &&
addressTable[tempParentNode]->mode!=EVICTING && addressTable[tempParentNode]->mode!=PRE_EVICTING)
                                        ||(addressTable[tempParentNode]->mode==EVICTING &&
addressTable[tempParentNode]->nodeBeingServed==nodeID)){
                                                addressTable[tempParentNode]->mode=MIGRATING;
                                        }
                                        a.getAgentAddress(agentID-1)-
>evictionsChecked.rmEntryFrom(i);
                                }else{
                                        i--;
                                }
                        }

                        a.getAgentAddress(agentID-1)->totalBenefit=0;
      //---------notify agentID migration was not possible------------
      arguments.init();
      arguments.addEntry(agentID,0);
      arguments.addEntry(nodeID,0);
      arguments.addEntry(INVALID,0);    //INVALID benefit
      h.addToHip(timer,ANSWER,arguments);
      //-----------------------------------------------------------------
                        for(i=0;i<a.getAgentAddress(agentID-1)->tempEvictions.getLength();i++){
                                position=addressTable[parentNode]-
>nodesOccupied.contains(a.getAgentAddress(agentID-1)->tempEvictions.getNodeID(i));
                                if(position!=-1)
                                        addressTable[parentNode]-
>nodesOccupied.rmEntryFrom(position);
                                n.initializeAgent(a.getAgentAddress(agentID-1)-
>tempEvictions.getNodeID(i));
                        }
                        a.getAgentAddress(agentID-1)->tempEvictions.init();
                        a.getAgentAddress(agentID-1)->benefitsArray.init();
                }
  }
}

void network::notifyNode(int agentID,int nodeID,int benefit,int timeHip){
//notify node currently hosting agentID (parentNode)
  dynamicList arguments,tempList;
  int sum,parentNode=0,i,j,currentAgent,temp,tempBenefit,length,currentNode;
        int flag=0,position,localBenefit,maxBenefit,maxNode,aBServed,tempParentNode;
  a.getAgentAddress(agentID-1)->nodesChecked--;  //inform agentID that a node (to which he
previously sent a HOST_REQUEST), has returned an answer
  while(addressTable[parentNode]->agentList.contains(agentID)==-1)
    parentNode++;
        if(addressTable[parentNode]->nodeBeingServed!=INVALID){
                tempParentNode=addressTable[parentNode]->nodeBeingServed-1;
        }else
                tempParentNode=INVALID;
        if(addressTable[parentNode]->agentList.contains(addressTable[nodeID-1]-
>isMostBeneficialFor)!=-1)
                temp=1;
        else
                temp=0;

        if(a.getAgentAddress(agentID-1)->nodesChecked<0){       //if function call was triggered by
checkForEvictions
                if(benefit!=INVALID){  //if eviction is beneficial
                        a.getAgentAddress(addressTable[parentNode]->agentBeingServed-1)-
>agentBeingChecked=agentID;     //********-------------********
```

```
                        temp=a.getAgentAddress(addressTable[parentNode]->agentBeingServed-1)-
>benefitsArray.addEntryInOrder(agentID,benefit);  //add inOrder and get position
      a.getAgentAddress(addressTable[parentNode]->agentBeingServed-1)-
>tempEvictions.addEntryAt(agentID,parentNode+1,temp);     //add it to tempEvictions of
agentBeingServed
                        addressTable[nodeID-1]->isMostBeneficialFor=agentID;
                        checkForEvictions(addressTable[parentNode]-
>agentBeingServed,parentNode+1,timeHip);
    }else{      //else if eviction was invalid
                        addressTable[nodeID-1]->agentBeingServed=INVALID;
//********-------------********
                        cout<<parentNode+1<<ENDLINE;
                        position=a.getAgentAddress(addressTable[parentNode]->agentBeingServed-1)-
>tempEvictions.contains(agentID);
                //-------restore all nodes to EVICTION or ASLEEP mode--------------
      i=0;
                        while(i<a.getAgentAddress(agentID-1)->nodesToRestore.getLength()){
//for every node that needs to be restored
                                if(a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeDistance(i)==nodeID){
                                        if(addressTable[a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i)-1]->mode!=BUSY)
                                                if(temp!=1){
                                                        n.initializeNode(a.getAgentAddress(agentID-
1)->nodesToRestore.getNodeID(i));
                                                }else{
                                                        addressTable[a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i)-1]->mode=EVICTING;
                                                        addressTable[a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i)-1]->agentBeingServed=INVALID;
                                                }
                                        currentNode=a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i);
                                        for(j=0;j<addressTable[currentNode-1]-
>agentList.getLength();j++){      //also restore their agents
                                                n.initializeAgent(addressTable[currentNode-1]-
>agentList.getNodeID(j));
                                        }
                                        a.getAgentAddress(agentID-1)-
>nodesToRestore.rmEntryFrom(i);
                                }else{
                                        i++;
                                }
                        }
                        //-------------------------------------------------------------------------
-----
                        if(a.getAgentAddress(agentID-1)->nodesToRestore.getLength()<=0)
                                n.initializeAgent(agentID);
                        a.getAgentAddress(addressTable[parentNode]->agentBeingServed-1)-
>agentBeingChecked=INVALID;     //new agent has to become agentBeingChecked
      checkForEvictions(addressTable[parentNode]->agentBeingServed,parentNode+1,timeHip);
//call check for evictions
                }

//=============================================================================================================
=========================================================
        }else{
//=============================================================================================================
=========================================================
                if(a.getAgentAddress(agentID-1)->maxBenefit<benefit){           //if
benefit/agentSize of this migration/eviction is greater than currently maxBenefit
      a.getAgentAddress(agentID-1)->maxBenefit=benefit;
                        if(a.getAgentAddress(agentID-1)->mostBeneficialNode!=INVALID){  //if there
is a currently most Beneficial Node
                                if(a.getAgentAddress(agentID-1)->mostBeneficialNode!
=tempParentNode+1 && temp!=1//7/11/2015.if currently most beneficial node δεν χρησιμοποιείται από
κάποιον συγκάτοικο agent
                                && ((addressTable[a.getAgentAddress(agentID-1)->mostBeneficialNode-
1]->mode!=BUSY        && addressTable[a.getAgentAddress(agentID-1)->mostBeneficialNode-1]->mode!
=EVICTING
                                        && addressTable[a.getAgentAddress(agentID-1)-
>mostBeneficialNode-1]->mode!=MIGRATING)   //και δεν είναι BUSY
                                || ((addressTable[a.getAgentAddress(agentID-1)->mostBeneficialNode-
1]->mode==EVICTING
                                                || addressTable[a.getAgentAddress(agentID-1)-
```

```
>mostBeneficialNode-1]->mode==MIGRATING) //ή αν είναι EVICTING ή MIGRATING τότε
                                                && addressTable[a.getAgentAddress(agentID-1)-
>mostBeneficialNode-1]->nodeBeingServed==parentNode+1
                                                && addressTable[a.getAgentAddress(agentID-1)-
>mostBeneficialNode-1]->agentBeingServed==agentID))){      //αν ο nodeBeingServed του είναι ο
parentNode   //16/11/2015
                                        i=0;
                                        while(i<addressTable[parentNode]->nodesOccupied.getLength())
{
                                                if(addressTable[parentNode]-
>nodesOccupied.getNodeDistance(i)==a.getAgentAddress(agentID-1)->mostBeneficialNode){
                                                        n.initializeNode(addressTable[parentNode]-
>nodesOccupied.getNodeID(i));
                                                        addressTable[parentNode]-
>nodesOccupied.rmEntryFrom(i);
                                                }else
                                                        i++;
                                        }
                                        n.initializeNode(a.getAgentAddress(agentID-1)-
>mostBeneficialNode);     //free and put currently most beneficial node to SLEEP
                                }
                                i=0;
                                while(i<a.getAgentAddress(agentID-1)->evictionsChecked.getLength()){
                                        position=a.getAgentAddress(agentID-1)-
>nodesToRestore.contains2(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i),a.getAgentAddress(agentID-1)->mostBeneficialNode);
                                        if(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i)==a.getAgentAddress(agentID-1)->mostBeneficialNode
                                        || position!=-1){
                                                a.getAgentAddress(agentID-1)-
>evictionsChecked.rmEntryFrom(i);
                                        }else
                                                i++;
                                }
                        }
      a.getAgentAddress(agentID-1)->mostBeneficialNode=nodeID;//and set this node as the currently
most beneficial for the agent to MIGRATE
    }else if(nodeID!=tempParentNode+1 && ((addressTable[nodeID-1]->mode==ASLEEP)
                || ((addressTable[nodeID-1]->mode==EVICTING || addressTable[nodeID-1]-
>mode==MIGRATING) && addressTable[nodeID-1]->nodeBeingServed==parentNode+1))){
                        if(temp!=1){
                                i=0;
                                while(i<addressTable[parentNode]->nodesOccupied.getLength()){
                                        if(addressTable[parentNode]-
>nodesOccupied.getNodeDistance(i)==nodeID){
                                                n.initializeNode(addressTable[parentNode]-
>nodesOccupied.getNodeID(i));
                                                addressTable[parentNode]-
>nodesOccupied.rmEntryFrom(i);
                                        }else
                                                i++;
                                }
                                if(addressTable[nodeID-1]->mode!=BUSY)
                                        n.initializeNode(nodeID);       //free this node
                                i=0;
                                while(i<a.getAgentAddress(agentID-1)->evictionsChecked.getLength()){
                                        position=a.getAgentAddress(agentID-1)-
>nodesToRestore.contains2(a.getAgentAddress(agentID-1)->evictionsChecked.getNodeDistance(i),nodeID);
                                        if(a.getAgentAddress(agentID-1)-
>evictionsChecked.getNodeDistance(i)==nodeID
                                        || position!=-1){
                                                a.getAgentAddress(agentID-1)-
>evictionsChecked.rmEntryFrom(i);
                                        }else
                                                i++;
                                }
                        }
                }


                //-------restore all nodes to EVICTION or ASLEEP mode--------------
```

```
      i=0;
                        while(i<a.getAgentAddress(agentID-1)->nodesToRestore.getLength()){
//for every node that needs to be restored
                                if(a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeDistance(i)==nodeID){
                                        if(a.getAgentAddress(agentID-1)-
>mostBeneficialNode==nodeID){
                                                if(addressTable[a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i)-1]->mode!=BUSY)
                                                        addressTable[a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i)-1]->mode=EVICTING;
                                                        addressTable[a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i)-1]->agentBeingServed=INVALID;
                                        }else{
                                                if(addressTable[a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i)-1]->mode!=BUSY)
                                                        if(/*a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i)!=nodeID
                                                        || (a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i)==nodeID && */temp!=1){

n.initializeNode(a.getAgentAddress(agentID-1)->nodesToRestore.getNodeID(i));
                                                        }else{

addressTable[a.getAgentAddress(agentID-1)->nodesToRestore.getNodeID(i)-1]->mode=EVICTING;

addressTable[a.getAgentAddress(agentID-1)->nodesToRestore.getNodeID(i)-1]-
>agentBeingServed=INVALID;
                                                        }
                                        }
                                        currentNode=a.getAgentAddress(agentID-1)-
>nodesToRestore.getNodeID(i);
                                        for(j=0;j<addressTable[currentNode-1]-
>agentList.getLength();j++){      //also restore their agents
                                                n.initializeAgent(addressTable[currentNode-1]-
>agentList.getNodeID(j));
                                        }
                                        a.getAgentAddress(agentID-1)-
>nodesToRestore.rmEntryFrom(i);
                                }else
                                        i++;
                        }
                        //-------------------------------------------------------------

                if(a.getAgentAddress(agentID-1)->nodesChecked==0){      //if agent got response
from all nodes to which he send hostRequests.
                        addressTable[parentNode]->agentsChecked--;                      //notify
node that max benefit of agentID has been found
                        tempBenefit=a.getAgentAddress(agentID-1)-
>maxBenefit*1000/a.getAgentAddress(agentID-1)->size;   //(*1000 so the outcome is an integer)
                        if(a.getAgentAddress(agentID-1)->maxBenefit!=INVALID){  //if there was a
possible eviction
                                addressTable[parentNode]-
>benefitsArray.addEntryInOrder(agentID,tempBenefit); //add it to the benefitsArray of parent node
in increasing order
                                addressTable[a.getAgentAddress(agentID-1)-
>mostBeneficialNode-1]->isMostBeneficialFor=agentID;
                        }else{
                                if(a.getAgentAddress(agentID-1)->state==BUSY){
                                n.initializeAgent(agentID);
                                a.getAgentAddress(agentID-1)->state=BUSY;
                                a.getAgentAddress(agentID-1)->isRootAgent=1;
                                }else
                                n.initializeAgent(agentID);
                        }
    }
    if(addressTable[parentNode]->nodeBeingServed!=nodeID && addressTable[nodeID-1]-
>agentBeingServed==agentID){ //αν δεν στέλνει αίτημα στον προηγούμενο κομβο και ο nodeID εξυπηρετεί
τον agentID
                        addressTable[nodeID-1]->agentBeingServed=INVALID;
                }

    if(addressTable[parentNode]->agentsChecked==0){     //if all agents of parentNode have returned
their maxBenefit
      if(addressTable[parentNode]->benefitsArray.getLength()>0){  //if there was at least one
```

```
possible eviction
                                if(addressTable[parentNode]->agentBeingServed!=INVALID){
//if agentID is not the rootAgent
                                        tempParentNode=0;
                                        while(addressTable[tempParentNode]-
>agentList.contains(addressTable[parentNode]->agentBeingServed)==-1) //get parentNode of
agentBeingServed
                                                tempParentNode++;
                                        for(i=0;i<addressTable[parentNode]-
>benefitsArray.getLength();i++){     //for every possible eviction
                                                currentAgent=addressTable[parentNode]-
>benefitsArray.getNodeID(i);
                                                a.getAgentAddress(addressTable[parentNode]-
>agentBeingServed-1)->tempEvictions.addEntry(currentAgent,parentNode+1);     //add it to
tempEvictions of agentBeingServed
                                                a.getAgentAddress(addressTable[parentNode]-
>agentBeingServed-1)->benefitsArray.addEntry(currentAgent,a.getAgentAddress(currentAgent-1)-
>maxBenefit);
                                                if(addressTable[tempParentNode]-
>nodesOccupied.contains(a.getAgentAddress(currentAgent-1)->mostBeneficialNode)!=-2      //fake!
                                                        &&
((addressTable[a.getAgentAddress(currentAgent-1)->mostBeneficialNode-1]->mode!=BUSY
                                                                &&
addressTable[a.getAgentAddress(currentAgent-1)->mostBeneficialNode-1]->mode!=EVICTING)
                                                        ||
(addressTable[a.getAgentAddress(currentAgent-1)->mostBeneficialNode-1]->mode==EVICTING
                                                                &&
addressTable[a.getAgentAddress(currentAgent-1)->mostBeneficialNode-1]-
>nodeBeingServed==parentNode+1))){
                                                        addressTable[tempParentNode]-
>nodesOccupied.addEntry(a.getAgentAddress(currentAgent-1)->mostBeneficialNode,parentNode+1);
                                                }
                                        }
                                        a.getAgentAddress(addressTable[parentNode]-
>agentBeingServed-1)->agentBeingChecked=INVALID;
                                        checkForEvictions(addressTable[parentNode]-
>agentBeingServed,parentNode+1,timeHip);
                                }else{                                  //if agentID is the one who sent the
initial host request       //********-------------********
                                        cout<<"That never happens!\n";
                                }
        }else{ //else if there weren't any possible evictions
                                addressTable[parentNode]->benefitsArray.init();
                                if(addressTable[parentNode]->nodeBeingServed!=INVALID ||
a.getAgentAddress(agentID-1)->isRootAgent==0){ //conflict 9 solution 18/11/2015. if this is not the
rootAgent
                                        arguments.init();
//return benefit INVALID to agentBeingServed
                                        arguments.addEntry(addressTable[parentNode]-
>agentBeingServed,0);
                                        arguments.addEntry(parentNode+1,0);
                                        arguments.addEntry(INVALID,0);
                                        h.addToHip(timeHip+1,ANSWER,arguments);
                                        aBServed=addressTable[parentNode]->agentBeingServed;
//********-------------********
                                        n.initializeAgent(agentID);
                                }else{  //if this is the rootAgent
                                        //----find parent of agentID-------initialize--
                                parentNode=0;
                                        while(addressTable[parentNode]-
>agentList.contains(agentID)==-1)
                                                parentNode++;

                                        addressTable[parentNode]->agentsYetToCheck.rmEntry(agentID);
//remove this agent from agentsYetToBeChecked list (as it failed to return a valid result)

                                        for(i=0;i<addressTable[parentNode]-
>nodesOccupied.getLength();i++){
                                                n.initializeNode(addressTable[parentNode]-
>nodesOccupied.getNodeID(i));
                                        }
```

```
                                                addressTable[parentNode]->nodesOccupied.init();

                                                if((addressTable[nodeID-1]->mode==ASLEEP)
                                                ||((addressTable[nodeID-1]->mode==EVICTING ||
addressTable[nodeID-1]->mode==MIGRATING) && addressTable[nodeID-1]->nodeBeingServed==parentNode+1))
                                                        n.initializeNode(nodeID);
//                                                      addressTable[parentNode]-
>nodesOccupied.addEntry(nodeID,-1);
                                                arguments.init();
                                                arguments.addEntry(agentID,0);
                                                arguments.addEntry(parentNode+1,0);
                                                h.addToHip(timeHip+1,MOVE_AGENT,arguments);
                                                //-----------------------------------
                                        }
        }
      }
    }
}

void network::migrationCheck(int agentID,int nodeID,int benefit,int timeHip){
int parentNode=0;
  dynamicList arguments;
  arguments.init();

  while(addressTable[parentNode]->agentList.contains(agentID)==-1)
    parentNode++;

  if(a.getAgentAddress(agentID-1)->state!=EVICTION){
    arguments.addEntry(agentID,0);
    arguments.addEntry(nodeID,0);
    h.addToHip(timeHip+1,MOVE_AGENT,arguments);
                timeHip+=2;     //********-------------********

                if(addressTable[parentNode]->mode==MIGRATING){  //δηλαδή όταν ο agentID χωρά χωρίς
eviction στο nodeID
                        addressTable[parentNode]->mode==BUSY;
                }
  }else{        //if agentID state==EVICTION
                if(addressTable[nodeID-1]->mode==ASLEEP && addressTable[nodeID-1]-
>agentBeingServed==INVALID){  //24/12/2015. Για να αποφευγχθεί η περίπτωση όπου ο nodeID
                        addressTable[nodeID-1]->agentBeingServed=agentID;
//ήταν BUSY και έχει μόλις αποφασιστεί τι θα γινόταν και έγινε πάλι ASLEEP
                }
                if(addressTable[nodeID-1]->mode==MIGRATING){    //if nodeID was ASLEEP before
agentID's hostRequest
                        addressTable[nodeID-1]->mode=EVICTING;//********-------------********
                }
                arguments.init();
    arguments.addEntry(agentID,0);
    arguments.addEntry(nodeID,0);
    arguments.addEntry(benefit,0);
    h.addToHip(timeHip+1,ANSWER,arguments);
  }
}

void network::sendHostRequestNFL(int agentID,int nodeID,int benefit,int timeHip){
 int
previousTimer=0,currentTimer=1,parentNode=0,numberOfAgents,currentAgent,position,reservedSpace;
  int
i,j,numberOfNodes,penalty,temp,currentNode,agentServed,flag=0,flag2=0,flag3=0,PEflag=0,invalidFlag=
0;     //PEflag==PRE_EVICTING_FLAG
  dynamicList arguments;
  arguments.init();
  numberOfAgents=addressTable[nodeID-1]->agentList.getLength(); //get number of agents hosted in
nodeID
  //-----------find node hosting agentID-----------------
  while(addressTable[parentNode]->agentList.contains(agentID)==-1)
    parentNode++;
  //-------------------------------------------------------
        agentServed=addressTable[nodeID-1]->agentBeingServed;   //set agentServed as the agent
Being Served by nodeID
// cout<<parentNode+1<<" mode="<<a.getAgentAddress(agentID-1)->state<<ENDLINE;
        if(addressTable[nodeID-1]->mode==AWAKE) //********-------------********
                addressTable[nodeID-1]->mode=ASLEEP;
//----raise flag in case agentID is sending host request to the node hosting the agent currently
```

```
trying to evict agentID(previous in path node)----
        if(nodeID==addressTable[parentNode]->nodeBeingServed)  //********-------------********
                flag=1;

        if(addressTable[nodeID-1]->nodeBeingServed==parentNode+1){ //********-------------********
                if(agentServed!=agentID && agentServed!=INVALID)                //if nodeID is busy
serving another agent inside parentNode
                        flag2=1;
//raise flag2
                else
//else if agentBeingServed by nodeID is agentID, or no agent is currently beingServed by nodeID(έχει
κληθεί από την checkForEvictions)
                        flag3=1;
//raise flag3
        }

        if(addressTable[parentNode]->mode==PRE_EVICTING){
                PEflag=1;
                addressTable[parentNode]->mode=MIGRATING;
        }

        if(a.getAgentAddress(agentID-1)->attemptCounter.getNodeDistance2(nodeID)>140){
//********-------------********
                a.getAgentAddress(agentID-1)->attemptCounter.rmEntry(nodeID);
                invalidFlag=0;
        }
        //=============conflict 7 solution==========
                if(addressTable[nodeID-1]->mode==ASLEEP){
                        for(i=0;i<addressTable[nodeID-1]->agentList.getLength();i++){
                                if(a.getAgentAddress(addressTable[nodeID-1]->agentList.getNodeID(i)-
1)->state!=FREE){
                                        flag2=1;
                                }
                        }
                }
        //=================================================
        reservedSpace=0;
        for(i=0;i<addressTable[nodeID-1]->reservations.getLength();i++)
                if(addressTable[nodeID-1]->reservations.getNodeID(i)!=agentID)
                        reservedSpace+=addressTable[nodeID-1]->reservations.getNodeDistance(i);


        if((addressTable[nodeID-1]->mode==MIGRATING ||flag2==1
        || addressTable[nodeID-1]->mode==PRE_EVICTING) && invalidFlag==0){      //if node is busy
migrating, or is currently doing eviction of another agent of parentNode
                if(PEflag==1){  //in case parentNode has sent host requests to other nodes as well,
apart from nodeID, so it needs to be in mode=EVICTING
                        addressTable[parentNode]->mode=PRE_EVICTING;
                }
                temp=a.getAgentAddress(agentID-1)->attemptCounter.getNodeDistance2(nodeID)+1;
//********-------------********
                a.getAgentAddress(agentID-1)->attemptCounter.rmEntry(nodeID);
//********-------------********
                a.getAgentAddress(agentID-1)->attemptCounter.addEntry(nodeID,temp);
//********-------------********
    arguments.addEntry(agentID,0);      //wait and try again in next timeHip
    arguments.addEntry(nodeID,0);
    arguments.addEntry(benefit,0);
    h.addToHip(timeHip+1,SEND_HOST_REQUEST,arguments);
  }else{        //if nodeID mode =ASLEEP, BUSY,(EVICTING && flag==1)
                if(((addressTable[nodeID-1]->remainingSpace-
reservedSpace>=a.getAgentAddress(agentID-1)->size) && invalidFlag==0) &&
                        (addressTable[nodeID-1]->mode==ASLEEP || addressTable[nodeID-1]->mode==BUSY
|| (addressTable[nodeID-1]->mode==EVICTING))){//case 1: migration is possible
                        a.getAgentAddress(agentID-1)->attemptCounter.rmEntry(nodeID);
//********-------------********
                        if(addressTable[parentNode]->mode!=EVICTING){
                                addressTable[parentNode]->mode=MIGRATING;               //set parent
to migrating so if some agent send hostRequest to this node he will have to resend the request
                        }
```

```
                        if(flag3==1 && addressTable[nodeID-1]->agentBeingServed==INVALID)
//αν ο nodeID έχει nodeBeingServed τον parentNode και δεν τον απασχολεί κάποιος συγκάτοικος
                                addressTable[nodeID-1]->agentBeingServed=agentID;
                        if(addressTable[nodeID-1]->mode==ASLEEP){
                                addressTable[nodeID-1]->agentBeingServed=agentID;
                                addressTable[nodeID-1]->nodeBeingServed=parentNode+1;
                                addressTable[nodeID-1]->mode=MIGRATING;
                        }
                        if(addressTable[nodeID-1]->reservations.contains(agentID)==-1)  //αν ο
nodeID δεν έχει ήδη δεσμεύσει χώρο για τον agentID.
                                addressTable[nodeID-1]-
>reservations.addEntry(agentID,a.getAgentAddress(agentID-1)->size);
                        arguments.init();
      arguments.addEntry(agentID,0);
      arguments.addEntry(nodeID,0);
      arguments.addEntry(benefit,0);
      h.addToHip(timeHip+1,MIGRATE,arguments);
    }else if(addressTable[nodeID-1]->size>=a.getAgentAddress(agentID-1)->size  && flag!=1 &&
invalidFlag==0
                        && (addressTable[nodeID-1]->mode==ASLEEP || (addressTable[nodeID-1]-
>mode==EVICTING && flag3!=0))){     //case 2: migrations might be possible after evictions
                        a.getAgentAddress(agentID-1)->attemptCounter.rmEntry(nodeID);
//********-------------********
                        addressTable[nodeID-1]->agentsChecked=0;        //initialize agentsChecked
      addressTable[nodeID-1]->benefitsArray.init();       //initialize array with the maximum
benefit of each agent
      a.getAgentAddress(agentID-1)->agentsOccupied.init();
      for(i=0;i<numberOfAgents;i++){ //for each agent in nodeID
                                currentAgent=addressTable[nodeID-1]->agentList.getNodeID(i);
                                if(a.getAgentAddress(currentAgent-1)->state==FREE &&
a.getAgentAddress(currentAgent-1)->childrenList.getLength()>0
                                        && a.getAgentAddress(currentAgent-1)->parentAgent==INVALID)
{    //avoid trying to MIGRATE BUSY && node specific agents
                                                n.initializeAgent(currentAgent);
                                                numberOfNodes=addressTable[nodeID-1]-
>childrenList.getLength();
                                                for(j=0;j<numberOfNodes;j++){   //for every node
that connects with the current
                                                        currentNode=addressTable[nodeID-1]-
>childrenList.getNodeID(j);
                                                        penalty=getLoad(currentAgent,nodeID)-
getLoad(currentAgent,currentNode);  //estimate penalty of such migration
                                                        flag=0;
                                                        if(addressTable[currentNode-1]-
>nodeBeingServed==nodeID){       //if currentNode already serves an agent of nodeID.
                                                                        flag=1;
                                                        }
                                                        //-----------------------------------------
-------------------------------------------------------------

                                                        if(benefit+penalty>0 &&
addressTable[currentNode-1]->mode!=PRE_EVICTING){       //and if it is Beneficial to keep doing
evictions
                                                                a.getAgentAddress(currentAgent-1)-
>nodesChecked++;      //increment number of nodes that have been checked and they could lead to a
possible eviction
                                                                a.getAgentAddress(currentAgent-1)-
>state=EVICTION;      //make agent unavailable to MIGRATE
                                                                a.getAgentAddress(currentAgent-1)-
>benefit=benefit;
                                                                a.getAgentAddress(currentAgent-1)-
>parentAgent=agentID;//20/12/2015
                                                                arguments.init();
                                                                arguments.addEntry(currentAgent,0);
                                                                arguments.addEntry(currentNode,0);

arguments.addEntry(benefit+penalty,0);
                                                                arguments.addEntry(agentID,0);

h.addToHip(timeHip+currentTimer,SEND_HOST_REQUEST,arguments);   //send host request at the
following timeHip
                                                                previousTimer=currentTimer;
                                                        }
                                                }
                                        }
```

```
                                        if(previousTimer==currentTimer){        //if at least one
hostRequest was sent for this agent
                                                a.getAgentAddress(agentID-1)-
>agentsOccupied.addEntry(currentAgent,0);  //add current agent to the list of agents that agentID
keeps occupied
                                                if(addressTable[parentNode]->nodeBeingServed!
=INVALID)  //if agentID is not the rootAgent
                                                        addressTable[parentNode]->mode=EVICTING;
                                                addressTable[nodeID-1]->agentsChecked++;
                                                currentTimer++;         //send next host request to
the next timeHip
                                        }
                                }
        if(previousTimer==0){     //if size of node was not enough and evictions couldn't be made
//ATTENTION: add condition to check if total size of possible to evict agents is greater or equal to
agentID->size.
//if not no reason to proceed to evictions
                                if(addressTable[nodeID-1]->mode==ASLEEP)
                                        addressTable[nodeID-1]->nodeBeingServed=parentNode+1;
                                addressTable[nodeID-1]->agentBeingServed=agentID;
                                if(PEflag==1){  //in case parentNode has sent host requests to other
nodes as well, apart from nodeID, so it needs to be in mode=EVICTING
                                        addressTable[parentNode]->mode=EVICTING;
                                }
                                arguments.init();
                                arguments.addEntry(agentID,0);
                                arguments.addEntry(nodeID,0);
                                arguments.addEntry(INVALID,0);
                                h.addToHip(timeHip+currentTimer,ANSWER,arguments);
        }else{    //if at least one eviction was possible
                                addressTable[nodeID-1]->nodeBeingServed=parentNode+1;
                                addressTable[nodeID-1]->agentBeingServed=agentID;
//set this node to serve agentID
                                if(addressTable[nodeID-1]->mode==ASLEEP){  //********-------------
********
                                        addressTable[nodeID-1]->mode=PRE_EVICTING;
                                        //cout<<nodeID<<ENDLINE;
                                }
                        }
      }else{  //case 3: if agentID can't fit in nodeID's space even after evictions
                        if(addressTable[parentNode]->nodeBeingServed==INVALID &&
a.getAgentAddress(agentID-1)->isRootAgent==1){ //δηλαδή ο agentID είναι rootAgent.
                                addressTable[parentNode]->mode=MIGRATING;
                        }
                        if(addressTable[nodeID-1]->mode==ASLEEP){
                                addressTable[nodeID-1]->mode=MIGRATING;
                                addressTable[nodeID-1]->nodeBeingServed=parentNode+1;
                        }
                        if(PEflag==1){  //in case parentNode has sent host requests to other nodes
as well, apart from nodeID, so it needs to be in mode=EVICTING
                                addressTable[parentNode]->mode=EVICTING;
                        }
                        arguments.init();
        arguments.addEntry(agentID,0);
        arguments.addEntry(nodeID,0);
        arguments.addEntry(INVALID,0);                  //send INVALID ANSWER
        h.addToHip(timeHip+currentTimer,ANSWER,arguments);
      }
    }
}

void network::simulation(){
 dynamicList arguments;
 arguments.init();
 int iterations,random, i,j,k,l,numberOfAgents,agentID,pAgentID;//parentAgent ID
 int currentBenefit,movingLoad,traffic,hops,length,flag;
 int nodeID,mostBeneficialAgent,mostBeneficialNetworkNode;
 int timer=0,benefit,currentNode,maxBenefit,localBenefit,parentNode;
 int nodesToWake=4;
 int used[nodesToWake*2];
srand(1);
```

```
  while(timer<=RUNTIME){        //number of times to repeat scanning of network
    //===========find all possible benefits==========
                h.printHipQueue(timer);
    //====excute all comands in hip queue that are to be executed at this timeHip=====
     h.executeHipCommands(timer);
                //===========then=============
                for(i=0;i<nodesToWake*2;i++)
                        used[i]=-1;
   //=============wake up and nodes and send hostRequests==============
    for(i=0;i<nodesToWake;i++){ //number nodes to wake
      random=rand()%MAX_NODES;  //random+1=current network node ID
      maxBenefit=INVALID; currentNode=random; mostBeneficialAgent=-1;
numberOfAgents=addressTable[currentNode]->agentList.getLength();
                        flag=0;
                        for(j=0;j<nodesToWake;j++){
                                if(used[j]==random+1)
                                        flag=1;
                        }
                        if(addressTable[currentNode]->mode==ASLEEP && flag==0){
                                addressTable[currentNode]->agentsYetToCheck.init();
                                for(j=0;j<numberOfAgents;j++){  //scann all agents contained in
node with nodeID=currentNode
                                        agentID=addressTable[currentNode]->agentList.getNodeID(j);
                                        if(a.getAgentAddress(agentID-1)->childrenList.getLength()>0
&& a.getAgentAddress(agentID-1)->state==FREE
                                        && a.getAgentAddress(agentID-1)-
>previousNodes.getLength()==0){//if it is a node neutral agent
                                                localBenefit=INVALID;
                                                for(k=0;k<addressTable[currentNode]-
>childrenList.getLength();k++){//for every connecting network node
                                                nodeID=addressTable[currentNode]-
>childrenList.getNodeID(k);    //connecting network node ID
                                                benefit=getLoad(agentID,currentNode+1)-
getLoad(agentID,nodeID); //find benefit of migration
                                        flag=0;
                                for(l=0;l<nodesToWake*2;l++){
                                        if(used[l]==nodeID)
                                                flag=1;
                                }
                                                if(benefit>localBenefit && flag==0)
//find maximum benefit among agentID's possible migrations to connecting network nodes
                                                        localBenefit=benefit;
                                                //--------decide which migration is the
most Beneficial---------
                                                if(benefit>maxBenefit   && flag==0){
//find maximum benefit among all agentIDs' migrations
                                                        maxBenefit=benefit;
                                                        mostBeneficialAgent=agentID;
                                                        mostBeneficialNetworkNode=nodeID;
                                                }
                                                //----------------------------------------
---------------------
                                                }
                                                addressTable[currentNode]-
>agentsYetToCheck.addEntryInOrder(agentID,localBenefit);              //?????????
                                        }
                                }
                                if(maxBenefit>0){               //if currentNode contains a
beneficial to MIGRATE agent

                                        //-------------initialize that agent's
variables-----------------
                                        n.initializeAgent(mostBeneficialAgent);
                                        a.getAgentAddress(mostBeneficialAgent-1)->state=BUSY;
                                        a.getAgentAddress(mostBeneficialAgent-1)->nodesChecked=1;
                                        a.getAgentAddress(mostBeneficialAgent-1)-
>mostBeneficialNode=currentNode;
                                        a.getAgentAddress(mostBeneficialAgent-1)->isRootAgent=1;
//conflict 9 solution
                                        //----------------------------------------------------
--------

                                        if(addressTable[mostBeneficialNetworkNode-1]->mode==ASLEEP)
//********--------------********
                                                addressTable[mostBeneficialNetworkNode-1]-
```

```
>mode=AWAKE;
                                        //----send Host request from
mostBeneficialAgent-------------
                                        arguments.init();
                                        arguments.addEntry(mostBeneficialAgent,0);
                                        arguments.addEntry(mostBeneficialNetworkNode,0);
                                        arguments.addEntry(maxBenefit,0);
                                        arguments.addEntry(40,0);
                                        h.addToHip(timer+1,SEND_HOST_REQUEST,arguments); //send
command to hip queue
                                        //-------------------------------------------------------
--
                                        used[i]=mostBeneficialNetworkNode;
                                        used[i+nodesToWake]=currentNode+1;
                                        n.initializeNode(currentNode+1);
                                        addressTable[currentNode]->mode=BUSY;
                                        addressTable[currentNode]->agentsChecked=1;
                                    }
                                }
                            }
    //=====================================================================

    timer++;
  }
}

int network::getTotalLoad(){
 int i,j,sum=0;
 for(i=0;i<MAX_NODES;i++){                                          //for each node
    for(j=0;j<addressTable[i]->agentList.getLength();j++){          //for each agent in the node
      sum+=getLoad(addressTable[i]->agentList.getNodeID(j),i+1);    //add it's load to sum
    }
  }
  return(sum);
}

int network::getTotalMigrations(){
  int i,result=0;
  for(i=0;i<a.getNumberOfElements();i++){
    result+=a.getAgentAddress(i)->migrations;
     //cout<<i+1<<" MIGRATEd "<<a.getAgentAddress(i)->migrations<<" times!\n";
  }
  result=result;
  return(result);
}

void testNetwork(){
  int i;
    for(i=0;i<a.getNumberOfElements();i++){
      cout<<i<<" size="<<a.getAgentAddress(i)->size<<"\n";
    }
}

#endif
```

Main function of simulation. File: simulation.cpp

```
#include</appTree.h>
#include</dynamicListHeader.h>
#include</networkNodeHeader.h>
#include<time.h>        //definition of time
#include<stdlib.h>      //definition of rand, srand, malloc,free,atoi
#include<stddef.h>      //definition of NULL
#include<math.h>        //sqrt function
#include<iostream>      //definition of cout
#include<iomanip>       //setw()
```

```
#include<string>
#include<fstream>       //file handling functions
using namespace std;

#define MAX_NODES 20
#define ENDLINE "==================================================\n"
#define TESTLINE "test==================================================\n"
#define INVALID -1

int main(int argc,char* args[]){
int result,first;
double percent;
h.init();

n.setupNetwork();

  first=n.getTotalLoad();
  cout<<"no simulation load="<<first<<"\n";
        n.printNetwork();

        n.simulation();
        n.printOutputValue();

  result=n.getTotalLoad();
  percent=((double)result*100/first);
  cout<<"total Load="<<result<<", %="<<percent<<", reduction="<<100-percent<<"\n";
  result=n.getTotalMigrations();
  cout<<"total Migrations Per Agent="<<result<<"\n";

  cout<<ENDLINE;

//    system("Pause");
/*

//testNetwork();
  */
  return 0;
}
```

Sample file of application Tree. File: appTreeFile.txt

```
13  //total number of agents
1:
2:
3:
4:
5:
6:
7:
8:
9:
10:
11:9|10|7|6|8|
12:2|3|5|
13:11|12|1|4|
$
```

Sample file of network Tree. testFile

```
//nodeID -(connected with) nodeID: distance

1-19:8

1-11:12

2-19:8

2-5:8

2-12:16

3-6:2

3-9:13

3-14:13

3-10:18

4-7:3

5-14:6

7-13:13

8-18:7

9-15:4

12-17:13

13-15:23

16-20:5

17-20:15

18-20:5

#

$
```