Πανεπιστήμιο Ιωαννίνων
Τμήμα Πληροφορικής

Μεταπτυχιακη εργασια εξειδικευσης

Κινητικότητα και ενημερώσεις σε δίκτυα peer to peer

*Η*λίας Λεοντιάδης

Επιβλέπων Καθηγητής
Βασίλειος Δημακόπουλος

Ιωάννινα
Οκτώβριος 2004

Περιεχόμενα

Περίληψη

Σε ένα σύστημα πολλαπλών πρακτόρων, υπάρχουν πράκτορες που χρειάζονται υπηρεσίες που τις παρέχουν άλλοι, αλλά δεν γνωρίζουν ποιος πράκτορας παρέχει ποια υπηρεσία. Υποθέτουμε ότι έχουμε μια αδόμητη προσέγγιση ομοτίμων (peer-to-peer) όπου ο κάθε πράκτορας διατηρεί μια τοπική cache που περιέχει εγγραφές για k υπηρεσίες, δηλαδή για κάθε υπηρεσία αποθηκεύει το που θα βρει τον πράκτορα που την παρέχει. Όμως, όταν ένας πράκτορας ή μια υπηρεσία μετακινείται οι εγγραφές αυτές μένουν ανενημέρωτες. Έτσι, προτείνουμε έναν αριθμό από πολιτικές ενημέρωσης των εγγραφών αυτών, που βασίζονται στους επιδημικούς αλγορίθμους push και pull. Επιπλέον, θα επεκταθούμε στον τομέα της δημιουργίας και συντήρησης αντιγράφων σε ένα δίκτυο ομοτίμων. Θα εξετάσουμε πως οι αλγόριθμοι push/pull μπορούν να χρησιμοποιηθούν ώστε να προσεγγίσουμε τον βέλτιστο αριθμό αντιγράφων κάθε αντικειμένου, αλλά και το πως μπορούν να χρησιμοποιηθούν ώστε να κρατήσουμε τα αντίγραφα αυτά συνεπή μεταξύ τους όταν υπάρχουν αλλαγές.

Κεφάλαιο 1

Εισαγωγή

Η αρχιτεκτονική που επικρατούσε στο Internet πριν την εμφάνιση των δικτύων ομοτίμων[1] ήταν βασισμένη στο μοντέλο πελάτη-εξυπηρέτη (client-server). Σήμερα, με την εμφάνιση ολοένα γρηγορότερων και συνεχώς ενεργών συνδέσεων στο internet (always on) έχει αναδειχθεί ένα νέο είδος διασύνδεσης. Η σύνδεση πλέον γίνεται ατομικά μεταξύ των client. Αντί να καλούμε αυτή την επικοινωνία client σε client την καλούμε peer-to-peer. Τα δίκτυα P2Pείναι μια νέα διασύνδεση που επικαλύπτει τις απλές συνδέσεις του internet (overlay network). Γενικά, στα δίκτυα peer to peer *κάθε συμμετέχων* κόμβος δρα και ως πελάτης αλλά και ως εξυπηρέτης (servent) παρέχοντας κάποιους πόρους ή υπηρεσίες. Ένα βασικό χαρακτηριστικό των δικτύων P2Pείναι το δυναμικό περιβάλλον λειτουργίας, ότι δηλαδή οι peers μπορούν να συνδεθούν ή να αποσυνδεθούν ανά πάσα στιγμή. Επιπλέον, δεν υπάρχει κάποια καθολική γνώση για όλο το σύστημα όπως για παράδειγμα το πόσοι peers υπάρχουν ή τι υπηρεσίες προσφέρει ο καθένας.

Τα συστήματα P2Pέχουν αναπτυχθεί πολύ τα τελευταία χρόνια και έγιναν γνωστά κυρίως μετά την εμφάνιση των πρώτων συστημάτων ανταλλαγής μουσικών αρχείων (π.χ. Napster [1]). Σήμερα υπάρχει μια πληθώρα εφαρμογών που εκμεταλλεύονται την αρχιτεκτονική P2Pόπως για παράδειγμα συστήματα κατανεμημένων υπολογισμών, ασφάλειας, υπηρεσίες ανταλλαγής μηνυμάτων, ανταλλαγής αρχείων κτλ.

---

[1] θα τα αποκαλούμε με τον αγγλικό όρο peer-to-peer ή P2P

Ένα σύστημα πολλαπλών πρακτόρων (MAS-multi agent system) είναι ένα δίκτυο από πράκτορες που συνεργάζονται για να προσφέρουν κάποια κατανεμημένη υπηρεσία ή να επιλύσουν ένα πρόβλημα που δεν θα μπορούσε να το λύσει ένας μόνο κόμβος. Για να επιτύχουν αυτή την συνεργασία, κάθε πράκτορας παρέχει κάποιες υπηρεσίες (resources) τις οποίες μπορούν να χρησιμοποιήσουν και οι υπόλοιποι μέσω κάποιου δικτύου που τους διασυνδέει. Για να χρησιμοποιήσουν μια υπηρεσία, πρέπει να επικοινωνήσουν με τον πράκτορα που την προσφέρει. Στα *κλειστά MAS (closed MAS)* κάθε πράκτορας γνωρίζει όλους τους άλλους πράκτορες που υπάρχουν στο σύστημα. Στα *ανοιχτά MAS (open MAS)* όμως, οι πράκτορες δεν γνωρίζουν ποιοι πράκτορες συμμετέχουν στο σύστημα, πόσο μάλλον το ποιοι πράκτορες προσφέρουν ποια υπηρεσία.

Στο μοντέλο που θα εξετάσουμε κάθε πράκτορας διατηρεί μια προσωπική cache περιορισμένου μεγέθους, που περιέχει πληροφορίες για τον εντοπισμό K διαφορετικών υπηρεσιών. Για κάθε μία από τις K υπηρεσίες κρατά πληροφορίες για το πού θα βρει τον πράκτορα που την παρέχει. Αυτό έχει σαν αποτέλεσμα να διατηρείται ένας κατανεμημένος κατάλογος που περιέχει πληροφορίες που αντιστοιχίζουν υπηρεσίες με τους πράκτορες που τα προσφέρουν.

Για να αναζητήσει κάποιος έναν πράκτορα η γενική πολιτική είναι ότι η μέθοδος της πλημμύρας (flooding): ο πράκτορας που ενδιαφέρεται ρωτάει ένα υποσύνολο από τους K πράκτορες που διατηρεί στην cache του. Αν και αυτοί δεν γνωρίζουν, προωθούν το μήνυμα στους δικούς τους γνωστούς κ.ο.κ. Μόλις βρεθεί ο πράκτορας που περιέχει την υπηρεσία, επιστρέφεται πίσω στον αρχικό κόμβο η πληροφορία.

1.1 Το πρόβλημα της κινητικότητας

Το πρώτο πρόβλημα που εξετάζουμε είναι το τι συμβαίνει στο παραπάνω δίκτυο όταν υπάρχει κινητικότητα. Με τον όρο *κινητικότητα* (*mobility*) εννοούμε ότι ένας πράκτορας ή μια υπηρεσία μπορεί να αλλάξει θέση στο δίκτυο. *Θέση* (*location*) ενός κόμβου, μπορεί να θεωρηθεί *η πληροφορία που χρειαζόμαστε για να επικοινωνήσουμε με τον συγκεκριμένο κόμβο*. Για παράδειγμα, θέση μπορεί να είναι η IP διεύθυνση σε ένα φυσικό TCP/IP δίκτυο. Ένα σύστημα open MAS είναι δυναμικό και συνεχώς μεταβαλλόμενο και άρα οι μετακινήσεις και οι αποσυνδέσεις κόμβων θα πρέπει να θεωρούνται συχνό φαινόμενο.

Η μετακίνηση ενός πράκτορα ή μιας υπηρεσίας μπορεί να προκαλέσει *ασυνέπεια στην cache των υπολοίπων πρακτόρων (cache inconsistency).* Τα πρόβλημα που δημιουργείται από την μετακίνηση ενός κόμβου είναι αρκετά σημαντικό. Κάθε μετακίνηση δημιουργεί έναν αριθμό μη έγκυρων εγγραφών στην cache και κάθε μη έγκυρη εγγραφή cache ουσιαστικά αφαιρεί μια σύνδεση από το δίκτυο. ΄Ομως, το σημαντικότερο πρόβλημα είναι ότι η νέα θέση του πράκτορα δεν είναι γνωστή σε κανέναν αφού όλοι γνωρίζουν μόνο την παλιά.

΄Ετσι, προτείνουμε κάποιες μεθόδους για την αντιμετώπιση των προβλημάτων που προκύπτουν από την κινητικότητα. Οι αλγόριθμοι αυτοί ονομάζονται *αλγόριθμοι ενημέρωσης* αφού φροντίζουν να ενημερώσουν τους πράκτορες του δικτύου για τις μετακινήσεις που έγιναν, ώστε και αυτοί με την σειρά τους να ενημερώσουν με τις νέες θέσεις τις cache τους. Ουσιαστικά, με το να ενημερώνουμε τις cache των πρακτόρων, ενημερώνουμε τις συνδέσεις τους. Κατα συνέπεια ενημερώνουμε την ίδια την τοπολογία του δικτύου.

Οι αλγόριθμοι ενημέρωσης που θα προτείνουμε, συνδυάζουν την τεχνική *pull*, η οποία ξεκίνα από τους πράκτορες που χρειάζονται την ενημέρωση, και την τεχνική *push*, που ξεκινά από τους πράκτορες που μετακινούνται. Μελετήσαμε την χρήση περιοδικών pull όπου οι πράκτορες περιοδικά ενημερώνουν ολόκληρη την cache τους και την καθ'απαίτηση (on-demand) χρήση pull όπου οι πράκτορες ενημερώνουν κάποια εγγραφή στην cache μόνο όταν ανακαλύψουν ότι δεν είναι έγκυρη. Κατα το push, όταν ο πράκτορας μετακινηθεί ενημερώνει τους υπολοίπους για την νέα θέση του. Μελετήσαμε αρκετές παραλλαγές αλγορίθμων flooding-push με στόχο να ενημερώσουμε όσο περισσότερους πράκορες γίνεται χρησιμοποιώντας όσο το δυνατό λιγότερα μηνύματα.

Προτείνουμε επίσης μία νέα παραλλαγή flooding στην οποία οι πράκτορες που λαμβάνουν πληροφορία σχετικά με την μετακίνηση άλλων, την κρατούν για λίγο σε κάποιον κατάλογο που ονομάζουμε *κατάλογο παρακολούθησης (snooping directory).* Συγκρίνουμε αυτές τις μεθόδους (που βασίζονται στο flooding) με την χρήση ενός «ενημερωμένου» push όπου οι πράκτορες κρατούν επιπρόσθετα και μία ανεστραμμένη cache (inverted cache). Στην inverted cache ένας πράκτορας A φυλάει πληροφορίες σχετικά με το *ποιοι πράκτορες τον έχουν στην cache τους*, δηλαδή ο A γνωρίζει ποιοι πράκτορες τον έχουν γείτονα και άρα ξέρει ποιους να ενημερώσει όταν μετακινηθεί. Με στόχο να μειώσουμε το μέγεθος του καταλόγου inverted cache θα συνδυάσουμε τον αλγόριθμο αυτό με την τεχνική μίσθωσης (leasing) όπου κάθε πράκτορας μισθώνει τον χρόνο του στην inverted cache κάποιου άλλου.

1.2 Το πρόβλημα των αντιγράφων

Το δεύτερο ζήτημα με το οποίο θα ασχοληθούμε αφορά τα αντίγραφα πληροφορίας (replicas) σε δίκτυα P2P. Πιο συγκεκριμένα, μελετήσαμε δυο σχετικά θέματα: την δημιουργία και των αντιγράφων και την συντήρησή τους. Στην περίπτωση της δημιουργίας αντιγράφων, βασικό ζήτημα αποτελεί ο αριθμός των αντιγράφων που δημιουργείται για κάθε αντικείμενο. Όπως θα δούμε ο βέλτιστος αριθμός αντιγράφων ενός αντικειμένου είναι ανάλογος της τετραγωνικής ρίζας αριθμού αναζητήσεων που γίνονται για αυτό. Μια καλή προσέγγιση για την επίτευξη αυτής της κατανομής είναι σε κάθε αναζήτηση να δημιουργούμε αντίγραφα που είναι ανάλογα με τον αριθμό των κόμβων που ερωτήθηκαν. Θα δούμε πως μπορούμε να εφαρμόσουμε τις μεθόδους push/pull για να επιτύχουμε αυτή την κατανομή.

Στην περίπτωση της συντήρησης/ενημέρωσης αντιγράφων, εξετάσαμε πως μπορούν να εφαρμοστούν οι αλγόριθμοι push/pull. Όταν ο ιδιοκτήτης ενός αντικειμένου προβεί σε αλλαγές κάνει push στο δίκτυο, ενώ οι κόμβοι που έχουν αντίγραφα εξετάζουν περιοδικά αν υπάρχει νέα έκδοση κάνοντας ένα on-demand pull. Παρουσιάσαμε επίσης έναν απλό τρόπο για να προσαρμόζεται χρόνος μεταξύ δυο περιοδικών pull. Θα δούμε επίσης πως μπορούμε να εκμεταλλευτούμε τον μηχανισμό παραγωγής των αντιγράφων ώστε να επιτύχουμε αποτελεσματικότερη ενημέρωση.

1.3 Πειραματικές μελέτες

Εκτός από την θεωρητική ανάπτυξη μεθόδων και αλγορίθμων, υλοποιήσαμε έναν εξομοιωτή με στόχο να αξιολογήσουμε την συμπεριφορά τους. Όσον αφορά τους αλγορίθμους ενημέρωσης στο δίκτυο των caches, κύριος στόχος μας είναι να εξετάσουμε την ποιότητα του δικτύου που προκύπτει με την χρήση του κάθε αλγορίθμου ενημέρωσης καθώς και τον φόρτο μηνυμάτων που προκαλεί κάθε αλγόριθμος. Τα πειραματικά αποτελέσματα υποδεικνύουν ότι με την σωστή ρύθμιση των σχετικών παραμέτρων, ο αλγόριθμος με καταλόγους snooping μπορεί να οδηγήσει στην ίδια συνέπεια των cache χρησιμοποιώντας πολύ λιγότερα μηνύματα σε σχέση με τον αλγόριθμο plain push/pull. Η μέθοδος με τους καταλόγους inverted cache είναι πολύ αποδοτική όσον αφορά το κόστος μηνυμάτων αλλά δεν μπορεί να χρησιμοποιηθεί όταν έχουμε πολλές αλλαγές στις cache και όταν έχουμε αναξιόπιστα δίκτυα.

Όσον αφορά την δημιουργία αντιγράφων, τα πειραματικά αποτελέσματα υποδεικνύουν ότι η μέθοδος που προτείναμε προσεγγίζει πολύ καλά την βέλτιστη κατανομή τετραγωνικής ρίζας. Επιπλέον, όπως θα δούμε οι αλγόριθμοι push/pull μπορούν να εφαρμοστούν για την ενημέρωση των αντιγράφων με μεγάλη επιτυχία.

1.4 Δομή κειμένου

Η δομή του κειμένου που ακολουθεί είναι η εξής. Στο κεφάλαιο 2 κρίθηκε απαραίτητο να γίνει μία σύντομη επισκόπηση των δικτύων P2Pτης πληθώρας ειδών και αλγορίθμων που μπορεί να συναντήσει κανείς. Θα ορίσουμε το τι είναι δίκτυα P2P, ποιες είναι οι χρήσεις και οι βασικές αρχές των δικτύων αυτών. Έπειτα θα αναλύσουμε την βασική αρχιτεκτονική για κάθε τύπο και θα αναφέρουμε περιληπτικά πληροφορίες για κάποια βασικά συστήματα . Τέλος θα περιγράψουμε τους βασικούς αλγορίθμους αναζήτησης και δρομολόγησης. Υπάρχει εκτενέστερη περιγραφή των δικτύων αυτών στο παράρτημα A.

Στο κεφάλαιο 3 θα περιγράψουμε το περιβάλλον της εργασίας μας. Θα δούμε τι είναι πράκτορας και multi agent systems. Θα περιγράψουμε αναλυτικά το δίκτυο των cache καθώς και τις βασικές μεθόδους αναζήτησης στο περιβάλλον αυτό.

Στο κεφάλαιο 4 θα κάνουμε επισκότιση της βιβλιογραφίας σε θέματα που αφορούν τις ενημερώσεις σε δίκτυα P2P.

Στο κεφάλαιο 5 θα περιγράψουμε τούς προτεινόμενους αλγορίθμους ενημέρωσης. Θα αναλύσουμε τους αλγορίθμους push και pull, καθώς και τις μεθόδους που προκύπτουν από τον συνδυασμό τους. Θα περιγράψουμε την μέθοδο plain push/pull, την μέθοδο push με καταλόγους snooping και την μέθοδο inverted cache push/pull. Θα αναλύσουμε την ιδέα του leasing και θα την συνδυάσουμε με τη μέθοδο inverted cache push/pull. Τέλος, θα περιγράψουμε τον εξομοιωτή και θα σχολιάσουμε τα πειραματικά αποτελέσματα.

Το κεφάλαιο 6 ασχολείται με την εφαρμογή των προτεινόμενων μεθόδων στην δημιουργία και την συντήρηση αντιγράφων (replicas) σε ένα δίκτυο P2P. Θα δούμε πως οι αλγόριθμοι push/pull μπορούν να εφαρμοστούν για να επιτύχουμε τον βέλτιστο αριθμό αντιγράφων αλλά και πως μπορούν να εφαρμοστούν για την συντήρηση των αντιγράφων αυτών.

Τέλος, στο παράρτημα Α παραθέτουμε μια αναλυτική επισκόπηση των δικτύων P2Pκαι ad-hoc ενώ στα επόμενα παραρτήματα ακολουθεί η τεκμηρίωση της υλοποίησης του εξομοιωτή.

Κεφάλαιο 2

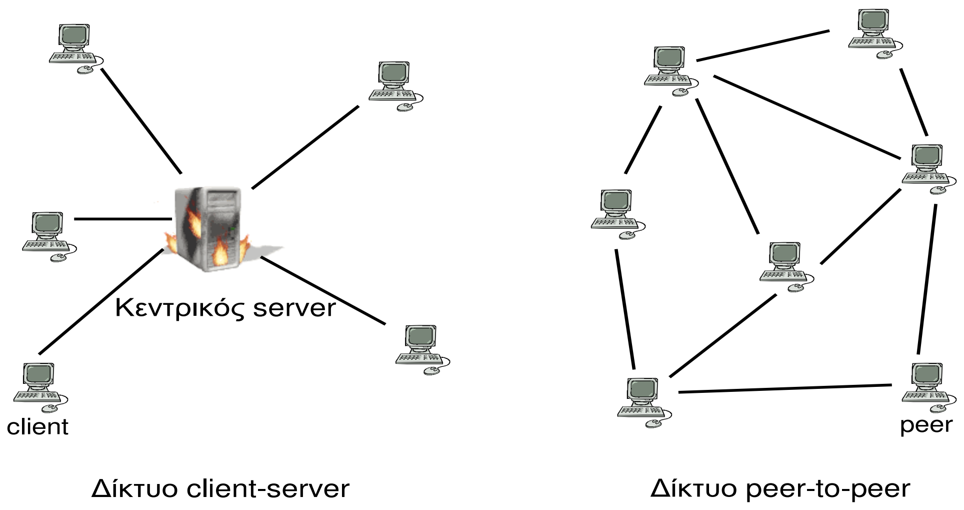
Επισκόπηση δικτύων Peer-To-Peer

Οι τεχνολογίες δικτύωσης που αναπτύσσονται περισσότερο στις μέρες μας είναι τα δίκτυα peer to peer (P2P) Υπάρχουν πάρα πολλά είδη τέτοιων δικτύων καθώς και πάρα πολλοί τομείς έρευνας που τα αφορούν γιαυτό κρίναμε σκόπιμο να κάνουμε μία σύντομη επισκόπηση των δικτύων αυτών καθώς και κάποιων βασικών μεθόδων λειτουργίας.

2.1 Ορισμός συστημάτων peer to peer

Η αρχιτεκτονική που επικρατούσε στο internet πριν την εμφάνιση των peer to peer ήταν βασισμένη στο μοντέλο πελάτη-εξυπηρέτη (client-server). Ο εξυπηρέτης ήταν συνήθως ένας ισχυρός υπολογιστής που παρείχε κάποιες υπηρεσίες και ήταν μονίμως διαθέσιμος (online), ενώ οι πελάτες ήταν οι υπολογιστές που χρησιμοποιούσαν τις υπηρεσίες του εξυπηρέτη και συνήθως οι clients δεν μπορούσαν να επικοινωνήσουν άμεσα μεταξύ τους (σχήμα 2.1). Στα συστήματα client-server δεν υπήρχε αυτο-οργάνωση στο δίκτυο, η οργάνωση και η διοίκηση ολόκληρου του συστήματος προκύπτει από την πολιτική του εξυπηρέτη.

Σήμερα, με την εμφάνιση ολοένα γρηγορότερων και συνεχώς ενεργών συνδέσεων στο internet (always on) ένα νέο είδος διασύνδεσης έχει αναδειχθεί. Η σύνδεση πλέον γίνεται ατομικά μεταξύ των client. Αντί να καλούμε αυτή την επικοινωνία client σε client την καλούμε peer-



Σχήμα 2.1: Από την αρχιτεκτονική πελάτη-εξυπηρέτη στα δίκτυα P2P

to-peer (ή δίκτυα ομοτίμων) αφού οι συνδέσεις γίνονται μεταξύ ομότιμων (ίσων) peers, ενώ η λέξη client υποδηλώνει μια υπο-υπηρεσία που εξαρτάται από κάποιον εξυπηρέτη. Το γεγονός ότι κάποιος κεντρικός εξυπηρέτης μπορεί να έχει βοηθήσει στην αρχική σύνδεση και επικοινωνία δεν έχει καμία σχέση με το που γίνεται η τελική δραστηριότητα. Το αποτέλεσμα είναι ότι όλες οι δραστηριότητες γίνονται από τους peers, ή αλλιώς στα άκρα του δικτύου. Αν για παράδειγμα θελήσει κάποιος να δημοσιεύσει κάτι, μπορεί να το κάνει στο δικό του υπολογιστή, αντί να το στείλει σε κάποιον εξωτερικό εξυπηρέτη. Τα δίκτυα P2Pείναι μια νέα επικαλυπτική (overlay) διασύνδεση πάνω στις τρέχουσες συνδέσεις του internet.

Στην πιο απλή του μορφή, P2Pείναι ένας υπολογιστής που ρωτάει έναν άλλο για κάποια υπηρεσία. Η υπηρεσία αυτή μπορεί να είναι αρχεία, λογισμικό, δυνατότητα εκτύπωσης, υπολογιστική ισχύς κτλ. Γενικά, στα peer to peer δίκτυα *κάθε συμμετέχων* κόμβος δρα και ως πελάτης αλλά και ως εξυπηρέτης (servent) παρέχοντας κάποιους πόρους ή υπηρεσίες. Οι peers μπορούν να συνδεθούν ή να αποσυνδεθούν ανά πάσα στιγμή και συνήθως δεν υπάρχει κάποια καθολική γνώση για όλο το σύστημα (πχ πόσοι peers υπάρχουν ή τι υπηρεσίες προσφέρει ο καθένας). Παρόλο που σε αυτή την ιδέα βασίζονται όλα τα p2p συστήματα, η αρχιτεκτονική που συναντάται από σύστημα σε σύστημα μπορεί να διαφέρει πολύ. Ένας αφαιρετικός και διαισθητικός ορισμός για τα peer to peer συστήματα δίνεται από τον Clay Shirkey [2].

Peer-to-peer είναι η κλάση των εφαρμογών που εκμεταλλεύονται resources όπως: αποθηκευτικός χώρος, επεξεργαστική ισχύς, υλικό, ανθρώπινη παρουσία, που υπάρχει διάσπαρτο σε κάθε μέρος του Internet. Επειδή η πρόσβαση σε αυτά τα μη κεντρικοποιημένα resources σημαίνει την λειτουργία σε ένα ασταθές περιβάλλον σύνδεσης και απρόβλεπτων IP διευθύνσεων, οι κόμβοι peer-to-peer πρέπει να λειτουργούν εκτός του DNS και να έχουν σημαντική ή πλήρη αυτονομία από κεντρικούς εξυπηρέτες.

Πολλοί πιστεύουν ότι P2Pείναι απλά ένας διαφορετικός όρος για τα *κατανεμημένα συστήματα* (distributed computing). Η σημαντική διαφορά μεταξύ των κατανεμημένων συστημάτων και των σημερινών δικτύων P2Pείναι ότι οι κόμβοι ενός συστήματος P2Pείναι συνήθως απλά PC και όχι μεγάλοι, συνεχώς συνδεδεμένοι εξυπηρέτες. Οι πελάτες είναι πλέον απλοί κόμβοι που βοηθούν στην επίτευξη αξιοπιστίας και ανεξαρτησίας *αλλά με μειωμένο κόστος*. Οι πόροι έχουν ήδη αγοραστεί (από του χρήστες) και απλά «κάθονται» και περιμένουν, αχρησιμοποίητοι και ανεκμετάλλευτοι (πχ οι επεξεργαστές των υπολογιστών ενός σχολικού εργαστηρίου την νύχτα). Τα δίκτυα P2Pμας επιτρέπουν να εκμεταλλευτούμε αυτά τα αχρησιμοποίητα resources.

Συμπερασματικά, πολλοί πιστεύουν ότι οι μέρες που χρειαζόμασταν έναν κεντρικό εξυπηρέτη για να φιλοξενήσουμε πληροφορίες έχουν περάσει. Οι υπολογιστές των πελατών είναι πλέον peers σε ένα δίκτυο, επιτρέποντας σε καθένα να φιλοξενεί και να προσφέρει τις δικές του πληροφορίες.

2.2 Χρήσεις συστημάτων P2P

Η χρήση των δικτύων P2Pδεν περιορίζεται μόνο στην ανταλλαγή μουσικών αρχείων. Υπάρχουν πολλές χρήσεις όπως:

- **Κατανεμημένοι υπολογισμοί (distributed computing):** Τα συστήματα αυτά κατανέμουν μια υπολογιστικά δύσκολη εργασία σε έναν αριθμό από peers με στόχο να χρησιμοποιήσουν τους υπολογιστικούς πόρους όλων αυτών των κόμβων για την επίλυσή της. Συνήθως οι peers δεν επικοινωνούν άμεσα μεταξύ τους αλλά μέσω ενός κεντρικοποιημένου εξυπηρέτη που διανέμει τα κομμάτια της εργασίας. Παραδείγματα τέτοιων συστημάτων είναι τα SETI@Home [3],United Devices [4], Entropia, GPU κτλ.

- **Κατανεμημένα κατεβάσματα αρχείων:** Γενικά, όταν ένα αρχείο προσφέρεται από έναν μόνο χρήστη, όσο περισσότεροι το κατεβάζουν, τόσο μικρότερες ταχύτητες επιτυγχάνουν. Αν όμως το αρχείο αυτό προσφέρεται από ένα P2Pσύστημα τότε όσοι κατεβάζουν το αρχείο το προσφέρουν και ταυτόχρονα (γίνονται και οι ίδιοι εξυπηρέτες). Αυτό έχει σαν αποτέλεσμα να αυξάνεται η συνολική διαθεσιμότητα (bandwidth) του συγκεκριμένου αρχείου, να κατανέμεται ο φόρτος σε περισσότερα από ένα σημεία του internet. Παραδείγματα τέτοιων δικτύων είναι το BitTorrent, Gnutella [5], Napster [1], LimeWire, Kazaa [6], Freenet [7] , Popular Power, E-donkey κτλ.

- **Μηχανές αναζήτησης:** Με την χρήση P2Pδικτύων μπορεί κάποιος να ψάξει αρχεία, υπηρεσίες, σελιδοδείκτες (bookmarks) κτλ που παρέχουν άλλοι χρήστες. Για παράδειγμα μπορεί μέσα σε μια εταιρία να υπάρχει ένα τοπικό δίκτυο που επιτρέπει σε προγραμματιστές να αναζητούν παραδείγματα κώδικα σε υπολογιστές συναδέλφων τους.

- **Επικοινωνία:** Υπάρχουν εφαρμογές chat που δεν βασίζονται σε έναν μόνο κεντρικοποιημένο εξυπηρέτη. Έτσι οι χρήστες για παράδειγμα μπορούν να ορίζουν μόνοι τους τους κανόνες, η και να επικοινωνούν απ'ευθείας μεταξύ τους για μεγαλύτερη ασφάλεια/ταχύτητα. Παραδείγματα τέτοιων υπηρεσιών είναι το netmeeting, ICQ [8], voice over IP.

- **Συνεργασία:** Μπορεί κάποιος να ξεκινήσει ένα έργο (project) με έναν αριθμό από φίλους, να διαμοιράζονται τα αρχεία του έργου, έναν πίνακα συζητήσεων (discussion board), κάποιο chat και άλλα απαραίτητα εργαλεία. Αργότερα μπορεί οποιοδήποτε μέλος να καλέσει άλλους να συμβάλλουν στο έργο μοιράζοντας και εκείνοι κάποιες υπηρεσίες.

- **Ασφάλεια:** Μπορούν κάποια μέλη σε μία κοινότητα να ενημερώσουν ο ένας τον άλλο για πιθανές τρύπες ασφαλείας, εισβολές κτλ. Έτσι μπορούν όλοι να λάβουν τα μέτρα τους. Η αποκεντρωμένη χρήση της υπηρεσίας αυτή σημαίνει ότι η απώλεια ενός μόνο κόμβου δεν δημιουργεί πρόβλημα στην ικανότητα των άλλων να αντιδρούν σε θέματα ασφαλείας.

- **Ιδιωτικότητα (privacy):** Για παράδειγμα στο Freenet [7], μπορεί ο καθένας να κάνει δημοσιεύσει την γνώμη του (ή κάποιο αρχείο) χωρίς τον φόβο να αποκαλυφθεί. Από την στιγμή που την δημοσιεύσει, αυτή θα παραμένει για όσο υπάρχουν χρήστες που την κατεβάζουν, χωρίς να υπάρχει τρόπος να την σβήσει ή να την τροποποιήσει κάποιος.

- **Έξυπνες συσκευές:** Φανταστείτε συσκευές όπως ένα δίκτυο από ανιχνευτές ή ακόμα και ένα δίκτυο μεταξύ αυτοκινήτων να επικοινωνούσαν μεταξύ τους ασύρματα με στόχο να προειδοποιήσουν για παράδειγμα για κάποιο ατύχημα ή για κάποια παρατήρηση.

2.3 Βασικές αρχές

Τα P2P συστήματα μπορούν να θεωρηθούν σαν ένα Internet σε επίπεδο εφαρμογής πάνω στο κλασσικό Internet. Οι βασικές αρχές που συναντάμε είναι:

- **Οι peers είναι ομότιμοι:** σε αντίθεση με τα συστήματα πελάτη εξυπηρέτη όπου κάθε κόμβος έχει τις δικές του αρμοδιότητες στα P2P δίκτυα οι peers λειτουργούν και ως πελάτες και ως εξυπηρέτες.

- **Διαμοιρασμός πόρων/υπηρεσιών:** Όλα τα P2P συστήματα αφορούν τον διαμοιρασμό πόρων όπως αρχεία (πχ τραγούδια, video, προγράμματα), εύρος δικτύου (bandwidth), υπηρεσίες (όπως υπηρεσίες πληροφόρησης). Με το να μοιράζονται οι πόροι μπορούν να υλοποιηθούν εφαρμογές που είναι πολύ βαριές για να βασιστούν σε ένα μόνο κόμβο. Αυτό ήταν και το κίνητρο συστημάτων όπως το Napster.

- **Άμεση συνεργασία:** Οι peers συνεργάζονται μεταξύ τους για την επίτευξη κάποιας συγκεκριμένης εργασία. Οι κόμβοι επικοινωνούν άμεσα μεταξύ τους και όχι με την χρήση κάποιου ενδιάμεσου εξυπηρέτη. Στα P2Pσυστήματα που υπάρχουν εξυπηρέτες, χρησιμοποιούνται μόνο για την ανακάλυψη των peers που προσφέρουν κάποιον απαιτούμενο πόρο και έπειτα οι peers συνεργάζονται χωρίς την παρέμβαση του εξυπηρέτη.

- **Αυτο-οργάνωση και αυτονομία:** Όταν έχουμε ένα πλήρως κατανεμημένο σύστημα, δεν υπάρχει κάποιος κόμβος που να διατηρεί μια βάση δεδομένων που να κρατάει καθολική πληροφορία για το σύστημα. Έτσι οι κόμβοι πρέπει να αυτοδιοργανωθούν *χρησιμοποιώντας μόνο τοπική πληροφορία και αλληλεπιδρώντας με γειτονικούς κόμβους*. Έτσι παίρνοντας τοπικές αποφάσεις μπορεί το σύστημα να δουλέψει συνολικά. Οι peers και οι συνδέσεις τους θεωρούνται προσωρινές. Έτσι δεν μπορούμε να κάνουμε εικασίες σχετικά με την τοπολογία.

- **Αποκέντρωση:** Η αποκέντρωση είναι ιδιαιτέρως χρήσιμη για την αποφυγή ενός σημείου αποτυχίας (single-point-of-failure) ή σημείων συμφόρησης (bottlenecks) της απόδοση ενός συστήματος με στόχο την κλιμάκωση (scalability). Όμως η κλιμάκωση αυτή βασίζεται κατά πολύ στο πρωτόκολλο του κάθε συστήματος. Πλήρη αποκεντρωμένα συστήματα είναι το Gnutella και το Freenet.

- **Λειτουργία σε ασταθές περιβάλλον:** Τα συστήματα P2Pβασίζονται σε κόμβους που συνδέονται και αποσυνδέονται συχνά από το δίκτυο. Επιπλέον επειδή οι προσωπικοί υπολογιστές που συμμετέχουν είναι πιο επηρεπείς σε τυχαίες αστοχίες (προβλήματα υλικού, προβλήματα συνδέσης στο internet, ιοί, χάκερς κτλ) οι αλγόριθμοι των δικτύων αυτών αντιμετωπίζουν το δυναμικό αυτό περιβάλλον σαν μια συνήθης διαδικασία και όχι σαν μια εξαίρεση.

- **Αναζήτηση:** Η αναζήτηση στα P2P συστήματα μοιάζει με στατικές αναζητήσεις όπως αυτές των Goggle ή Yahoo. Στην στατική αναζήτηση, η πληροφορία πριν αναζητηθεί πρέπει πρώτα να συλλεχθεί από κάποιον crawler (οι crawler συλλέγουν πληροφορίες από σελίδες και τις προσθέτουν σε βάσεις δεδομένων μηχανών αναζήτησης). Και μεταξύ δυο επισκέψεων ενός crawler σε κάποιο site μπορούν να περάσουν μήνες . Στο ενδιάμεσο η πληροφορία μπορεί να αλλάξει, να διαγραφεί κτλ με αποτέλεσμα τα αποτελέσματα της αναζήτησης να είναι αναξιόπιστα. Τα δίκτυα P2P δίνουν την δυνατότητα για αναζήτηση σε πραγματικό χρόνο. Όμως, ανάλογα με το πρωτόκολλο, κάποια πληροφορία μπορεί να μην βρεθεί ακόμα και αν υπάρχει στο δίκτυο. Επίσης δεν υπάρχει εγγύηση για την διάρκεια της αναζήτησης. Έτσι υπάρχει πρόβλημα ποιότητας υπηρεσιών (Quality Of Service ή QoS) στα συστήματα P2P .

2.4 Τύποι συστημάτων P2P

Κεντρικοποιημένα συστήματα (Centralized P2P):

Κάθε peer συνδέεται σε έναν εξυπηρέτη που συντονίζει τις επικοινωνίες μεταξύ των peers και όλες οι ανταλλαγές μηνυμάτων γίνονται μέσω του εξυπηρέτη (σχήμα 2.2). Πολλοί υποστηρίζουν ότι τα συστήματα αυτά δεν θα πρέπει να θεωρούνται P2Pαφού οι peers δεν επικοινωνούν απ'ευθείας. Απλώς ο κεντρικός εξυπηρέτης καταγράφει τις υπηρεσίες που προσφέρουν οι peers και αν κάποιος θελήσει να χρησιμοποιήσει μία από αυτές, τις χρησιμοποιεί μέσω του εξυπηρέτη. Τα πλεονεκτήματα αυτής της μεθόδου είναι ότι η διεύθυνση του εξυπηρέτη είναι γνωστή και είναι εύκολο να συνδεθεί κάποιος σε αυτόν. Επιπλέον, επειδή τα πάντα περνούν από τον εξυπηρέτη, μπορεί να χρησιμοποιηθεί κάποιο caching για να αυξηθεί η απόδοση. Δεν υπάρχει περίπτωση να έχουμε κατάτμηση του δικτύου αφού υπάρχει μόνο ένα επίπεδο συνδέσεων. Τα μειονεκτήματα είναι ότι ο εξυπηρέτης είναι σημείο συμφόρησης και σημείο αποτυχίας (αν «πέσει» ο εξυπηρέτης δεν λειτουργεί όλο το δίκτυο. Μερικά παραδείγματα τέτοιων εφαρμογών είναι εφαρμογές που διαμοιράζουν επεξεργαστικούς πόρους (CPU sharing applications) όπως το SETI@Home [3].

Συστήματα μεσίτη (Broker P2P):

Οι peers συνδέονται σε κάποιο εξυπηρέτη για να ανακαλύψουν άλλους peers και υπηρεσίες και μόλις βρουν αυτό που θέλουν επικοινωνούν απευθείας μεταξύ τους για να συνεργαστούν (σχήμα 2.3). Ο κάθε κόμβος ξέρει μόνο τον κεντρικό εξηπηρέτη και κάποιους peers που ο εξυπηρέτης του «σύστησε». Τα πλεονεκτήματα αυτών των δικτύων είναι ότι έχει την απόδοση ενός κεντρικοποιημένου εξυπηρέτη (όσον αφορά τις αναζητήσεις) αλλά ταυτόχρονα επιτρέπει τις απ'ευθείας συνδέσεις των peers που αποσυμφορίζουν τον κεντρικό εξυπηρέτη από τις εσωτερικές ανταλλαγές μηνυμάτων. Παραδείγματα τέτοιων δικτύων είναι το Napster [1].

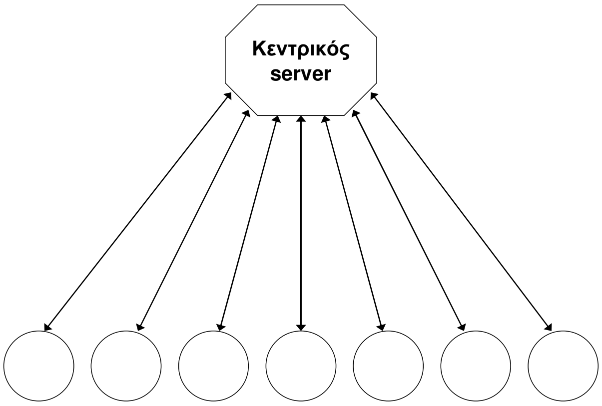
Αποκεντρωμένα συστήματα (Decentralized P2P):

Αυτή είναι η αληθινή αρχιτεκτονική peer-to-peer: δεν υπάρχει κανένας εξυπηρέτης, απλά peers (σχήμα 2.4). Ένας νέος κόμβος συνδέεται σε έναν υπάρχον κόμβο του δικτύου και αυτός τον συστήνει σε μερικούς ακόμα. Δεν υπάρχει κανένας κεντρικοποιημένος κόμβος που να συντονίζει το δίκτυο. Το δίκτυο λειτουργεί συνολικά με την λήψη μόνο τοπικών αποφάσεων σε κάθε peer. Δυο ξεχωριστά δίκτυα μπορούν να ενωθούν με έναν απλό κόμβο που συνδέεται και στα δύο. Τα πλεονεκτήματα είναι ότι δεν υπάρχει κάποιο αδύνατο σημείο στο δίκτυο όσον αφορά την συμφόρηση και τις αστοχίες. Το μειονεκτήμα είναι ότι οι αναζητήσεις είναι αργές αφού πρέπει να μεταδοθούν μέσω πολλών βημάτων σε όλους τους κόμβους (multi-hop searching). Παραδείγματα τέτοιων δικτύων είναι η Gnutella [5] και το Freenet [7].

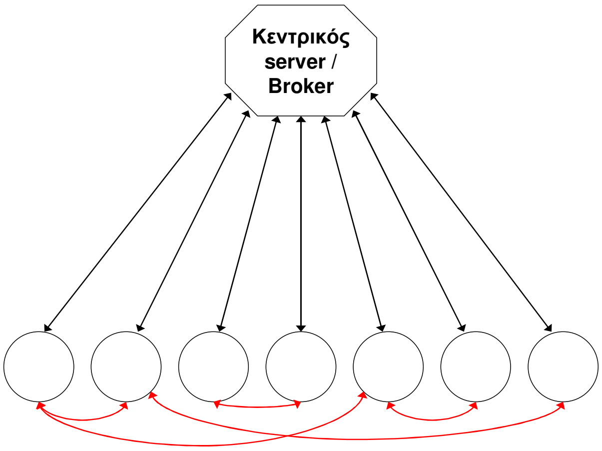
Μερικώς αποκεντρωμένα συστήματα (Partially decentralized P2P):
Τα δίκτυα αυτά είναι υβριδικά ανάμεσα στα συστήματα μεσίτη και στα αποκεντρωμένα. Στα συστήματα αυτά, όταν υπάρχουν κάποιοι κόμβοι που υπερτερούν σε σχέση με άλλους (καλύτερη σύνδεση στο internet, καλύτερο επεξεργαστή, μεγαλύτερη χωρητικότητα δίσκου, γνωρίζουν περισσότερους peers κ.α.) τότε αυτοί οι κόμβοι λειτουργούν ως super nodes. Οι super nodes ή super peers, λειτουργούν σαν μικροί εξυπηρέτες-μεσίτες μέσα σε ένα μεγαλύτερο αποκεντρωμένο δίκτυο. Κάθε κόμβος του δικτύου (συμπεραλαμβανομένων των super peers), συνδέεται σε έναν ή περισσότερους super peers και τους χρησιμοποιεί για εκτελέσει γρήγορες αναζητήσεις (σχήμα 2.5). Οι κόμβοι αυτοί δεν αποτελούν σημείο αποτυχίας (point-of-failure) για το δίκτυο αφού επιλέγονται δυναμικά και σε περίπτωση αποτυχίας το δίκτυο θα τους αντικαταστήσει με άλλους. Παραδείγματα τέτοιων δικτύων είναι το Kazaa [6] και το Morpheus.

2.5 Δομή δικτύων P2P

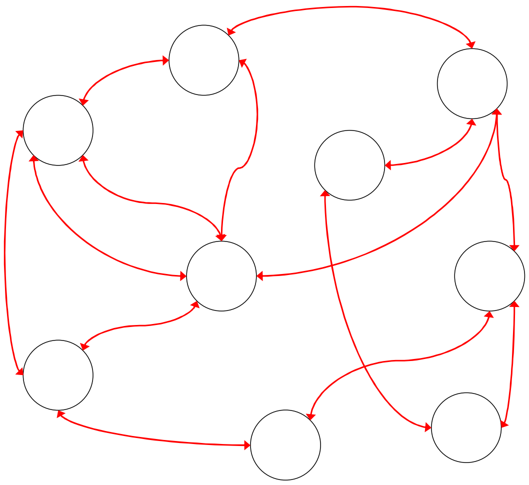
Τα συστήματα P2P αποτελούνται από δυναμικά δίκτυα με πολύπλοκη τοπολογία. Αυτή η τοπολογία δημιουργεί ένα δίκτυο επικάλυψης (overlay network) που μπορεί να μην έχει καμία σχέση με το φυσικό δίκτυο που συνδέει τους κόμβους. Τα P2P δίκτυα μπορεί να διαφοροποιηθούν ως προς το αν αυτό το overlay έχει κάποια δομή (structure) ή δημιουργείται τυχαία (ad-hoc). Με τον όρο *δομή (ή structure)*, εννοούμε τον τρόπο με τον οποίο κατανέμεται το περιεχόμενο σε σχέση με την τοπολογία: αν υπάρχει κάποιος τρόπος να πούμε απ'ευθείας ποιοι κόμβοι έχουν κάποια συγκεκριμένη πληροφορία ή αν πρέπει να αναζητήσουμε ολόκληρο



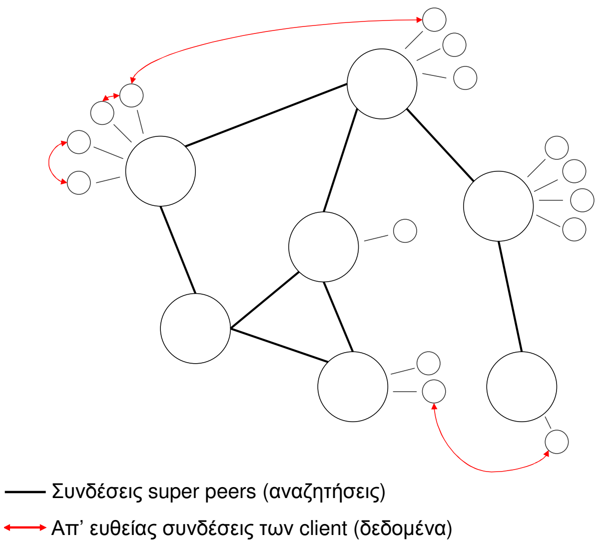
Σχήμα 2.2: Κεντρικοποιημένα P2Pσυστήματα



Σχήμα 2.3: Υβριδικά αποκεντρωμένα (brokered) P2P



Σχήμα 2.4: Πλήρως αποκεντρωμένα P2P



Σχήμα 2.5: Μερικώς αποκεντρωμένα P2P

το δίκτυο.

Αδόμητα P2P

Στα *P2Pσυστήματα χωρίς δομή (unstructured P2P)* όπως το Gnutella [5], η τοποθέτηση της πληροφορίας δεν έχει καμία σχέση με την overlay τοπολογία. Αφού δεν υπάρχει καμία πληροφορία για το ποιοι κόμβοι προσφέρουν ποια resources, η αναζήτηση σημαίνει τυχαία αναζήτηση σε κάποιους κόμβους του δικτύου. Το θετικό με αυτά τα δίκτυα είναι ότι μπορούν να εξυπηρετήσουν ένα μεγάλο και δυναμικό πληθυσμό από κόμβους. Όμως το κόστος της αναζήτησης είναι μεγάλο με αποτέλεσμα τα δίκτυα αυτά να θεωρούνται να μην θεωρούνται κλιμακούμενα (unscalable).

Δομημένα P2P

Με στόχο να επιτύχουμε μεγαλύτερη ικανότητα κλιμάκωσης (scalability), δημιουργήθηκαν τα *δομημένα συστήματα P2P (structured)*. Τέτοια συστήματα είναι το Chord, CAN, PAST, Tapestry κτλ. Στα δίκτυα αυτά η επικαλύπτουσα τοπολογία του δικτύου είναι ελεγχόμενη και οι πόροι τοποθετούνται σε συγκεκριμένες θέσεις. Αυτά τα συστήματα παρέχουν κάποια αντιστοίχηση (mapping) μεταξύ του πόρου και της θέσης του με την μορφή κάποιου πίνακα δρομολόγησης (routing table) έτσι ώστε η αναζήτηση να μπορεί να δρομολογηθεί στον κόμβο που έχει τον επιθυμητό πόρο. Τα δίκτυα αυτά προσφέρουν κλιμάκωση για αναζητήσεις κάποιων πόρων που είναι καλά ορισμένοι. Όμως τα δίκτυα αυτά είναι δύσκολο να διατηρήσουν την δομή τους σε ένα μεγάλο και δυναμικά παραγόμενο πληθυσμό από peers (κόμβοι συνδέονται και αποσυνδέονται συχνά).

Ελαφρώς δομημένα P2P

Μια τρίτη κατηγορία δικτύων είναι τα *ελαφρώς δομημένα (loosely structured)*. Τέτοιο παράδειγμα δικτύου είναι το Freenet. Τα δίκτυα αυτά ταξινομούνται κάπου ανάμεσα στα προηγούμενα δύο. Οι θέσεις των πόρων επηρεάζονται από κάποια δρομολόγηση αλλά δεν καθορίζονται από αυτή.

Έχουν αναπτυχθεί αρκετά συστήματα P2Pσε κάθε κατηγορία. Μπορείτε να δείτε που κατατάσσονται μερικά από τα πιο δημοφιλή συστήματα P2Pστον πίνακα 2.1.

	Αδόμητα	Ελαφρώς δομημένα	Δομημένα
Συστήματα μεσίτη	Napster		
Αποκεντρωμένα	Gnutella	Freenet	Chord, Can, Tapestry
Μερικώς αποκεντρωμένα	Kazaa, Gnutella		

Πίνακας 2.1: Κατηγορίες γνωστών P2P δικτύων

Κεφάλαιο 3

Περιγραφή μοντέλου συστήματος

Το περιβάλλον εργασίας των αλγορίθμων ενημέρωσης που θα παρουσιάσουμε παρακάτω (κεφ 5) είναι ένα σύστημα πολλαπλών πρακτόρων (multi agent system). Πρόκειται για ένα P2Pδίκτυο από πράκτορες που συνεργάζονται μεταξύ τους προσφέροντας ένα σύνολο από resources. Πιο συγκεκριμένα, το μοντέλο στο οποίο θα βασιστούμε είναι το μοντέλο κατανεμημένης cache (distributed cache model). Στις παραγράφους που ακολουθούν θα αναλύσουμε τις έννοιες πράκτορας (agent), συστήματα πολλαπλών πρακτόρων (multi agent system), distributed cache model και θα παρουσιάσουμε μερικούς αλγορίθμους αναζήτησης που λειτουργούν στο μοντέλο αυτό.

3.1 Πράκτορες

Ο πράκτορας είναι μια υπολογιστική οντότητα όπως για παράδειγμα ένα πρόγραμμα υπολογιστή ή ένα ρομπότ που αντιλαμβάνεται και αλληλεπιδρά με το περιβάλλον του και είναι αυτόνομο με την έννοια ότι η συμπεριφορά του μπορεί να βασίζεται, έστω και ελάχιστα, σε προηγούμενη εμπειρία. Ένας απλός ορισμός είναι [9]:

Ένας πράκτορας είναι ένα υπολογιστικό σύστημα που βρίσκεται μέσα σε κάποιο περιβάλλον και που είναι ικανός να δρά αυτόνομα μέσα σε αυτό με στόχο να επιτευχθούν *στόχοι για τους οποίους σχεδιάστηκε.*

Ένας πράκτορας μπορεί να θεωρηθεί «έξυπνος» (intelligent agents) με την έννοια ότι μπορεί να λειτουργήσει με ευελιξία και λογική σε μια πληθώρα καταστάσεων χρησιμοποιώντας διαδικασίες επίλυσης προβλημάτων, σχεδιασμού, λήψης αποφάσεων και μάθησης. Ένας πράκτορας είναι επίσης μια οντότητα που αλληλεπιδρά με το περιβάλλον στο οποίο βρίσκεται αφού μπορεί να επηρεαστεί από άλλους πράκτορες ή και από ανθρώπινες παρεμβάσεις.

Για παράδειγμα, οποιοδήποτε σύστημα που ελέγχει κάποια λειτουργία μπορεί να θεωρηθεί ως πράκτορας. Το κλασσικό παράδειγμα ενός τέτοιου συστήματος είναι ένας θερμοστάτης. Ο θερμοστάτης έχει έναν αισθητήρα που παίρνει κατευθείαν πληροφορίες από το *περιβάλλον* (π.χ. το δωμάτιο) και παράγει ως έξοδο δύο ειδών σήματα: ένα που υποδεικνύει ότι η θερμοκρασία είναι *χαμηλή* και ένα που υποδεικνύει ότι η θερμοκρασία είναι *εντάξει*. Έτσι οι δυνατές ενέργειες του θερμοστάτη είναι «ενεργοποίηση την θέρμανση» και «απενεργοποίησε την θέρμανση». Φυσικά αυτό είναι ένα απλό παράδειγμα αφού υπάρχουν πολύ πιο πολύπλοκα περιβάλλοντα και αποφάσεις όπως για παράδειγμα σε συστήματα «fly by wire» ή σε συστήματα ελέγχου πυρηνικών αντιδραστήρων.

3.2 Συστήματα πολλαπλών πρακτόρων (Multi agent systems)

Ένα multi-agent system (MAS) είναι ένα δίκτυο από πράκτορες που συνεργάζονται για να προσφέρουν κάποια κατανεμημένη υπηρεσία ή να επιλύσουν ένα πρόβλημα που δεν θα μπορούσε να το λύσει ένας μόνο κόμβος. Για να επιτύχουν αυτή την συνεργασία, κάθε πράκτορας παρέχει κάποιες υπηρεσίες (resources) τις οποίες μπορούν να χρησιμοποιήσουν και οι υπόλοιποι μέσω ενός δικτύου διασύνδεσης. Το δίκτυο αυτό μπορεί να είναι κάποιο δίκτυο P2P. Σε σχέση με ένα κεντρικοποιημένο σύστημα, τα MAS δεν έχουν κάποιο σημείο αστοχίας (point of failure). Επιπλέον, δεν έχουν κάποιο σημείο συμφόρησης ή κάποιο περιορισμό στον αριθμό και τις απαιτήσεις των υπηρεσιών που προσφέρονται (αφού τις διαχειρίζονται πολλοί πράκτορες και όχι μόνο ένας).

Για να εκπληρώσουν του στόχους τους, οι πράκτορες χρειάζονται πόρους που τους παρέχουν άλλοι πράκτορες. Όμως, για να χρησιμοποιήσουν μια υπηρεσία, πρέπει βρούμε και να επικοινωνήσουν με τον πράκτορα που την προσφέρει. Στα *κλειστά MAS (closed MAS)* κάθε πράκτορας γνωρίζει όλους τους άλλους πράκτορες που υπάρχουν στο σύστημα, αλλά στα *ανοιχτά MAS (open MAS)* δεν υπάρχει τέτοια γνώση, πόσο μάλλον γνώση για το ποιοι πράκτορες προσφέρουν ποια υπηρεσία.

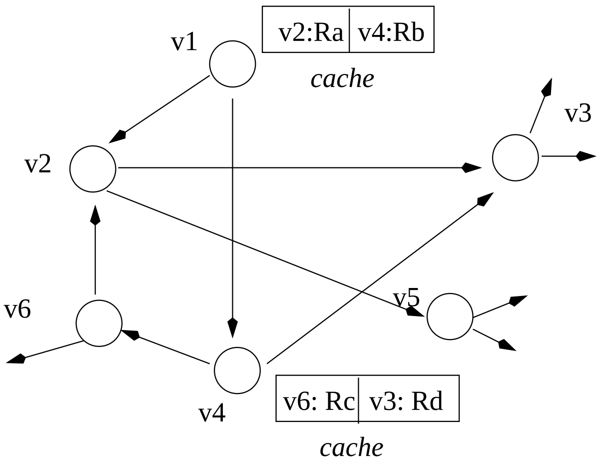
Μια κλασσική προσέγγιση για την επίλυση αυτού του προβλήματος είναι η ύπαρξη κάποιων *middle agents* που φυλάνε την αντιστοιχία αυτή. Έτσι όταν κάποιος πράκτορας θέλει να βρει μια υπηρεσία επικοινωνεί πρώτα με τον middle agent, λαμβάνει την απάντηση και έπειτα επικοινωνεί απ'ευθείας με τον πράκτορα που του σύστησε ο middle agent. Όμως, η μέθοδος αυτή μπορεί να αποτελέσει σημείο συμφόρησης και έρχεται σε αντίθεση με τους σχεδιαστικούς στόχους των MAS όσον αφορά την υπολογιστική ικανότητα, την αξιοπιστία, την επεκτασιμότητα, την στιβαρότητα, την ευκολία συντήρησης και την ευελιξία. Έτσι προτάθηκε ένας κατανεμημένος τρόπος για να μπορεί να εντοπιστούν οι πράκτορες που προσφέρουν κάποιο resource.

3.3 Μοντέλο συστήματος: Distributed cache

Για να μπορέσει κάποιος πράκτορας να αναζητήσει έναν άλλο, προτάθηκε μια πλήρως κατανεμημένη λύση όπου ο κάθε πράκτορας γνωρίζει έναν μικρό αριθμό από άλλους [10]. Ποιο συγκεκριμένα, κάθε πράκτορας διατηρεί μια προσωπική cache περιορισμένου μεγέθους που περιέχει πληροφορίες για τον εντοπισμό K διαφορετικών υπηρεσιών. Για κάθε μία από τις K υπηρεσίες, κρατά πληροφορίες για το πως θα επικοινωνήσει με τον πράκτορα που την παρέχει. Κατ'αυτόν τον τρόπο διατηρείται ένας κατανεμημένος κατάλογος που περιέχει πληροφορίες που αντιστοιχίζουν τις υπηρεσίες με τους πράκτορες που τις προσφέρουν. Έτσι εξαλείφονται τα προβλήματα του κεντρικοποιημένου καταλόγου αφού το μοντέλο αυτό προσφέρει καλύτερη ικανότητα κλιμάκωσης (scalability) και μεγαλύτερη αντοχή σε αποτυχίες.

Ουσιαστικά, δημιουργείται ένας *κατευθυνόμενος γράφος απο πράκτορες* $G(V, E)$ που καλείται *δίκτυο των caches (cache network)*. Υπάρχει μια ακμή από έναν πράκτορα A στον

B όταν όταν ο A έχει αποθηκευμένη στην cache του κάποια υπηρεσία που την παρέχει ο πράκτορας B. O A και ο B καλούνται *γείτονες* αφού η επικοινωνία μεταξύ τους είναι άμεση. Ένας πράκτορας μπορεί να προσφέρει παραπάνω από μια υπηρεσία, έτσι ο ίδιος πράκτορας μπορεί να εμφανίζεται παραπάνω από μια φορά σε κάποια cache. Παράδειγμα τέτοιου δικτύου μπορείτε να δείτε στο σχήμα 3.1.

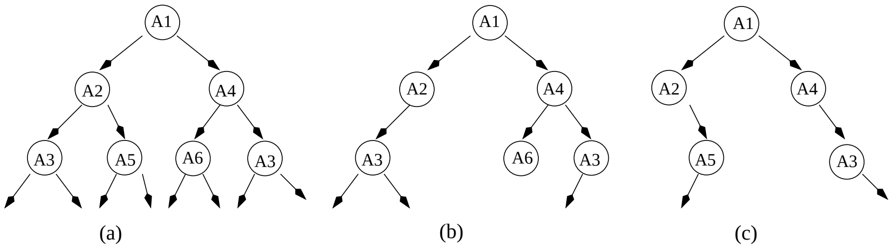


Σχήμα 3.1: Τμήμα του δικτύου όπου ο κάθε πράκτορας v_i κρατάει πληροφορίες για $k = 2$ resources

3.4 Αναζητήσεις στο δίκτυο των cache

Για να αναζητήσει κάποιος έναν πράκτορα που προσφέρει μια συγκεκριμένη υπηρεσία υπάρχουν πολλοί τρόποι. Η γενική πολιτική είναι ότι ο πράκτορας που ενδιαφέρεται ρωτάει ένα υποσύνολο από τους K πράκτορες που διατηρεί στην cache του. Αν και αυτοί δεν γνωρίζουν, προωθούν το μήνυμα στους δικούς τους γνωστούς κ.ο.κ. Μόλις βρεθεί κάποιος πράκτορας που γνωρίζει την απάντηση, επικοινωνεί απ'ευθείας με τον αρχικό κόμβο για να τον ενημερώσει.

Στο μοντέλο αυτό μπορούν να εφαρμοστούν κάποιοι από τους αλγορίθμους που υπάρχουν στα απλά αδόμητα δίκτυα P2Pμόνο που η αναζήτηση δεν γίνεται πάνω σε κάποιο σταθερό δίκτυο αλλά στο δίκτυο των caches. Αν οι cache κάποιου κόμβου δεν είναι ενημερωμένη, τότε δεν θα μπορέσει να προωθήσει σωστά το μήνυμα αφού δεν θα γνωρίζει τις θέσεις των γειτόνων



Σχήμα 3.2: Αναζήτηση στο δίκτυο των caches του σχήματος 3.1: a) flooding, b) teeming (ή teeming με decay), c) random paths ($p = 2$)

του. Θα περιγράψουμε έναν αριθμό από βασικούς αλγορίθμους που είναι κατάλληλοι για το μοντέλο αυτό.

Plain flooding

Στο απλό flooding, κάθε πράκτορας προωθεί *σε όλους* τους γείτονές του κάποιο μήνυμα αναζήτησης που λαμβάνει. Καθώς προωθούνται τα μηνύματα σχηματίζεται το δέντρο του σχήματος 3.2α. Ο όρος δέντρο ίσως να μην είναι αρκετά ακριβής γιατί κάποιος κόμβος μπορεί να υπάρχει παραπάνω από μία φορά στον γράφο αυτό (αφού υπάρχουν κύκλοι) αλλά βοηθάει στο να οπτικοποιήσουμε την επικοινωνία.

Για να περιοριστεί η ανεξέλεγκτη πλυμμήρα του δικτύου με μηνύματα η επικοινωνία σταματά όταν φτάσουμε σε ένα μέγιστο βάθος t ενώ αν ένας κόμβος λάβει το ίδιο μήνυμα παραπάνω από μία φορές δεν το επαναπροωθεί. Κάθε κόμβος εξετάζει αν η αναζητούμενη υπηρεσία βρίσκεται ή όχι στην cache του και αντίστοιχα ενημερώνει τον αρχικό κόμβο ή προωθεί το μήνυμα παραπέρα.

Το βασικό μειονέκτημα του flooding είναι ο πολύ μεγάλος αριθμός μηνυμάτων που πλημμυρίζουν το δίκτυο, ο οποίος μάλιστα αυξάνεται εκθετικά ως προς το βάθος t της αναζήτησης αλλά και ως προς το μέγεθος της cache k.

Teeming

Το teeming είναι μια ακόμη μέθοδος αναζήτησης που σχεδιάστηκε με στόχο να περιοριστεί ο αριθμός των μηνυμάτων σε σχέση με το απλό flooding. Στον αλγόριθμο αυτό, ο κάθε πράκτορας που λαμβάνει ένα μήνυμα αναζήτησης το προωθεί *σε ένα τυχαίο υποσύνολο των γειτόνων του.*

Ο κάθε πράκτορας επιλέγει με μία πιθανότητα φ να προωθήσει ή όχι το μήνυμα σε κάποιον γείτονά του. Έτσι, αν K είναι οι γείτονες ενός peer, το μέγεθος του υποσυνόλου αυτού μπορεί να είναι από 0 ως K, με μέσο μέγεθος $K\varphi$. Το απλό flooding μπορεί να θεωρηθεί ως μια ειδική περίπτωση του teeming με $\varphi = 1$. Παρόμοια τεχνική είναι το modified random BFS [11].

Ο αλγόριθμος αυτός εγγυάται μόνο πιθανοτικά την επιτυχία της αναζήτησης. Όσο πιο μικρό είναι το φ, τόσο λιγότεροι κόμβοι θα διερευνηθούν με αποτέλεσμα να έχουμε και μικρότερη πιθανότητα εύρεσης αποτελέσματος. Όμως περιορίζει αρκετά τον αριθμό των μηνυμάτων όπως έδειξαν στο [10].

Teeming με παράμετρο decay

Καθώς κατεβαίνουμε σε χαμηλότερα επίπεδα του δέντρου αναζήτησης υπάρχουν στο δίκτυο πολλαπλά αντίγραφα του ίδιου μηνύματος αναζήτησης. Για παράδειγμα, στο flooding τα μηνύματα αυτά αυξάνουν εκθετικά όπως μπόρεί να δεί κανείς και στο σχήμα 3.2. Για να αντιμετωπίσουμε αυτό το πρόβλημα, όσο το βάθος της αναζήτησης αυξάνεται, μειώνουμε σταδιακά το μέγεθος του υποσυνόλου των γειτόνων που επιλέγεται.

Έτσι η πιθανότητα φ δεν παραμένει σταθερή αλλά είναι συνάρτηση του βάθους της αναζήτησης. Για να ελέγξουμε το *πόσο γρήγορα μειώνεται* το μέγεθος του υποσυνόλου αυτού, ορίζουμε μια παράμετρο d που την ονομάζουμε «decay». Πλέον η πιθανότητα φ είναι συνάρτηση του βάθους της αναζήτησης s, αλλά και της παραμέτρου decay d: δηλαδή $\varphi = f(s, d)$.

Πειραματιστήκαμε με αρκετές συναρτήσεις μείωσης της πιθανότητας $f(s, d)$. Επειδή όπως προαναφέραμε κατά την λειτουργία ενός αλγορίθμου flooding τα μηνύματα στο δίκτυο αυξάνονται εκθετικά συναρτήσει του βάθους αναζήτησης, πήραμε τα καλύτερα αποτελέσματα όταν η πιθανότητα φ μειώνεται και αυτή εκθετικά. Δηλαδή όταν $\varphi = (1 - d)^s$.

Τυχαία μονοπάτια (Random paths) [12, 10]

Στον αλγόριθμο αυτό ο peer που ξεκινά την αναζήτηση επικοινωνεί με $p < k$ γείτονές του. Από εκεί και πέρα κάθε peer προωθεί το μήνυμα *μόνο σε έναν*, τυχαία επιλεγμένο, γείτονά του. Έτσι δημιουργούνται p μονοπάτια αναζήτησης μήκους t. Στον αλγόριθμο αυτό συνήθως επιλέγεται πολύ μεγαλύτερο TTL σε σχέση με τους προηγούμενους έτσι ώστε να αναζητήσουμε σε ένα επαρκές υποσύνολο του δικτύου. Είναι φανερό ότι αυτή η μέθοδος έχει το μικρότερο φόρτο μηνυμάτων, όμως έχει και την μικρότερη πιθανότητα επιτυχούς αναζήτησης.

Υπάρχουν πολλές παραλλαγές αυτού του αλγορίθμου. Ο k-walker random algorithm [13] παράγει k-walkers που προχωράνε στο δίκτυο επικοινωνώντας περιοδικά με τον query node για να μάθουν αν πρέπει να σταματήσουν ή να συνεχίσουν (αν η πληροφορία βρέθηκε). Στο Two-Level Random Walk Search Protocol [14] υπάρχουν δύο TTL. Μέχρι να λήξει το πρώτο TTL παράγονται walkers που συνεχίζουν μέχρι να λήξει το δεύτερο.

Iterative deepening

Η μέθοδος του iterative deepening προτάθηκε με στόχο να ελαχιστοποιήσει τα μηνύματα που προκαλούνται από ένα ευρύ flooding όταν αυτό δεν είναι απαραίτητο [15, 16, 12]. Την χρησιμοποιούμε όταν γνωρίζουμε ότι η πληροφορία έχει μεγάλες πιθανότητες να βρεθεί στην «γειτονιά» ενός κόμβου.

Ουσιαστικά καλούμε κάποια από τις παραπάνω μεθόδους αναζήτησης για $t = 1$. Αν η αναζήτηση δεν επιστρέφει κανένα αποτέλεσμα, επαναλαμβάνουμε την αναζήτηση με $t = 2$ κ.ο.κ. μέχρι να βρούμε την αναζητούμενη πληροφορία. Λόγω του ότι τα μηνύματα της αναζήτησης αυξάνονται εκθετικά όσο αυξάνεται το t, είναι πιθανό μια επαναληπτική αναζήτηση για $t = 1, 2, 3$, για παράδειγμα, να επιφέρει συνολικά λιγότερα μηνύματα σε σχέση με μια απευθείας αναζήτηση με $t = 4$.

Κεφάλαιο 4

Σχετικές εργασίες

Στο κεφάλαιο αυτό θα παρουσιάσουμε περιληπτικά άλλες ερευνητικές εργασίες που είναι σχετικές με το αντικείμενο της δικής μας εργασίας. Θα παρουσιάσουμε εργασίες που σχετίζονται με αλγορίθμους ενημέρωσης σε συστήματα P2P. Το τι θεωρείται ενημέρωση διαφέρει από εργασία σε εργασία. Οι αλγόριθμοι που θα παρουσιάσουμε χρησιμοποιούν τις επιδημικές μεθόδους push και pull για την ενημέρωση οπότε κρίνουμε σκόπιμο να παρουσιάσουμε εργασίες που εξετάζουν άλλες χρήσεις των αλγορίθμων αυτών. Μια άλλη ομάδα εργασιών σχετίζεται με την ενημέρωση αντιγράφων. Τα αντίγραφα αυτά μπορεί να είναι replicas μιας κατανεμημένης βάσης δεδομένων ή και απλά αρχεία. Τέλος θα δούμε που αλλού έχουν χρησιμοποιηθεί έννοιες όπως το snooping και το leasing που θα χρησιμοποιήσουμε παρακάτω.

Ενημερώσεις σε αναξιόπιστα συστήματα P2P

Οι επιδημικοί αλγόριθμοι χρησιμοποιήθηκαν για πρώτη φορά στον τομέα της ενημέρωσης στο [17]. Η εργασία αυτή επικεντρώνεται σε αναξιόπιστα, δυναμικά και ασταθή P2Pόπου οι κόμβοι μπορούν να είναι συχνά offline. Οι συγγραφείς υποθέτουν ένα πλήρως δυναμικό περιβάλλον όπου δεν υπάρχει καθολική γνώση για τους υπόλοιπους peers του συστήματος. Θεωρούν ότι οι αλλαγές στα διαμοιραζόμενα δεδομένα είναι πολλές και συχνές. Ως εκ τούτου, για να επιτύχουν αντοχή σε αποτυχίες προτείνουν ως μόνη λύση την αντιγραφή της πληροφορίας σε πολλούς peers (replication). Συνεπώς, στην εργασία αυτή καλούνται να βρούνε αλγορίθμους που να κρατούν τα αντίγραφά της πληροφορίας συνεπή.

Η μέθοδος που προτείνουν βασίζεται σε επιδημικούς αλγορίθμους (rumor spreading and epidemic algorithms) που εγγυούνται μόνο πιθανοτική επιτυχία ενημέρωσης. Ορίζουν την έννοια του push: Οι πράκτορες που αλλάζουν κάτι στα δεδομένα τους κάνουν push στο δίκτυο την πληροφορία αυτή ώστε να την μάθουν όλοι. Για να αντιμετωπίσουν το γεγονός ότι κάποιοι peers μπορεί να ήταν offline κατά την διάρκεια της ενημέρωσης συνδυάζουν την μέθοδο push με την μέθοδο pull, κατά την οποία αναζητούν πληροφορίες από το δίκτυο από ενημερωμένους πράκτορες.

Με στόχο πάντα την λειτουργία σε ασταθή περιβάλλοντα, ορίζουν τις παραμέτρους του push. Ορίζουν τον όρο time to live (TTL) που είναι ο μέγιστος αριθμός των προωθήσεων που γίνονται σε κάθε μήνυμα κατά την διάρκεια της πλημμύρας (flooding). Επίσης αναφέρουν ότι ο κάθε peer θα πρέπει να προωθεί το μήνυμα σε έναν τυχαίο υποσύνολο των γειτόνων του (και όχι σε όλους). Τέλος, για να αποφύγουν τις επαναμεταδώσεις ορίζουν ότι κάθε peer θα πρέπει να μεταδίδει το ίδιο μήνυμα το πολύ μια φορά.

Τέλος, ανέλυσαν την απόδοση των αλγορίθμων αυτών με ένα αναλυτικό μαθηματικό μοντέλο και έδειξαν ότι για κατάλληλες τιμές των παραμέτρων του push μπορεί να επιτευχθεί κλιμάκωση (scalability) του αλγορίθμου.

Ενημέρωση αντιγράφων σε συστήματα ανταλλαγής αρχείων

Στο [18] ασχολούνται με την ενημέρωση αντιγράφων σε P2Pσυστήματα ανταλλαγής αρχείων. Υποθέτουν ότι στα P2Pδίκτυα ανταλλαγής αρχείων για κάθε αρχείο υπάρχει ένας χρήστης που το δημοσίευσε για πρώτη φορά (*δημιουργός*) και πολλοί χρήστες που το έχουν κατεβάσει και το χρησιμοποιούν τοπικά. Το πρόβλημα είναι ότι όταν ο δημιουργός αλλάξει κάτι στο αρχείο τότε αυτή η ενημέρωση πρέπει να γίνει γνωστή και σε αυτούς που το κατέβασαν. Αυτό έχει δύο συνέπειες: πρώτον οι χρήστες που κατέβασαν το αρχείο θα πρέπει να χρησιμοποιούν την τοπική cache μόνο αν έχουν την πιο πρόσφατη έκδοση και δεύτερον θα πρέπει να απαντούν θετικά σε queries μόνο αν η έκδοση που έχουν είναι συνεπής με αυτή του δημιουργού.

Χρησιμοποίησαν τις μεθόδους push, pull και push with adaptive pull για να προωθήσουν αυτές τις ενημερώσεις. Η μέθοδος push ξεκινά από τον δημιουργό του αρχείου όταν αλλάξει κάτι. Ουσιαστικά, στέλνει μηνύματα ακύρωσης (invalidation messages) στο δίκτυο έτσι ώστε να διαγράψουν όλοι τα τοπικά αντίγραφα. Όσον αφορά την μέθοδο pull, επειδή οι πράκτορες δεν γνωρίζουν αν το αρχείο που χρησιμοποιούν είναι η τελευταία έκδοση θα πρέπει περιοδικά να κάνουν pull νέες εκδόσεις από το δίκτυο. Έτσι επικεντρώθηκαν στην μέθοδο *pull with adaptive TTR* όπου κάθε αντίγραφο έχει ένα χρόνο λήξης. Ο χρόνος λήξης υπολογίζεται με βάση στατιστικά στοιχεία και ουσιαστικά είναι το διάστημα που ο χρήστης «πιστεύει» ότι ο δημιουργός δεν θα αλλάξει κάτι στο αρχείο.

Η εργασία λοιπόν επικεντρώνεται στο να βρεθεί η τιμή αυτού του χρόνου με βάση παλιότερα στατιστικά. Η γενική ιδέα για τον υπολογισμό του χρόνου λήξης είναι ότι αυτός θα πρέπει να αυξάνεται κατα μια σταθερά αν σε δυο διαδοχικά pull δεν έχει αλλάξει η έκδοση το αρχείου και αντίστοιχα να μειώνεται όταν σε δυο διαδοχικά pull βρούμε το αρχείο αλλαγμένο.

Εξομοίωσαν ένα P2Pδίκτυο ανταλλαγής αρχείων που με βάση το δίκτυο Gnutella [5] και έδειξαν ότι μπορεί να επιτευχθεί αυτού του είδους η συνέπεια χρησιμοποιώντας τον συνδυασμό push with adaptive pull. Επίσης έδειξαν ότι η απόδοση του αλγορίθμου επηρεάζεται από το μέγεθος του δικτύου και από τον αριθμό των συνδέσεων που διατηρεί κάθε κόμβος.

Στη δική μας εργασία, εφαρμόζουμε την ίδια αρχική ιδέα των αλγορίθμων push/pull αλλά διαφοροποιούμαστε σε αρκετά σημεία. Πρώτον, δεν έχουμε πλέον αντίγραφα αρχείων αλλά θέσεις της υπηρεσίας στο δίκτυο. Στην δική μας περίπτωση είτε θα βρούμε μια υπηρεσία είτε όχι. Δεν υπάρχει περίπτωση να χρησιμοποιούμε μια παλιότερη έκδοση της υπηρεσίας και άρα ο αλγόριθμος pull with adaptive TTR δεν έχει εφαρμογή. Επιπλέον, υποθέτουν ότι το δίκτυο gnutella σχηματίζει συνεχώς ένα overlay δίκτυο κρατώντας έναν σταθερόελάχιστο αριθμό γειτόνων κάθε φορά. Στην δική μας περίπτωση το δίκτυο των caches είναι το επικαλύπτων δίκτυο. Ουσιαστικά εμείς ενημερώνουμε το ίδιο το δίκτυο και όχι αντίγραφα αρχείων που υπάρχουν στο δίκτυο. Τέλος, επεκτείνουμε τους αλγορίθμους push/pull με τους αλγορίθμους push with snooping directories και inverted cache push/pull.

Συνέπεια σε κατανεμημένες βάσεις δεδομένων

Οι αλγόριθμοι push/pull χρησιμοποιήθηκαν και σε ένα διαφορετικό πεδίο όπως αυτό των βάσεων δεδομένων. Στο [19] ασχολήθηκαν με το πρόβλημα της ενημέρωσης σε Virtual Data Warehouses (VW) χρησιμοποιώντας τους αλγορίθμους push/pull. Η ιδέα είναι ότι τα δεδομένα που μεταφέρονται από απομακρυσμένες βάσεις δεδομένων αποθηκεύονται σε ένα VW. Έτσι η cached αυτή πληροφορία χρησιμεύει στο να εξυπηρετήσει τις ανάγκες των χρηστών που θα συνδεθούν σε αυτή. Ο cache server αυτός είναι υπεύθυνος να αποθηκεύσει και να συντηρήσει την cache (συνέπεια).

Στο σύστημα αυτό, ο cache server χρησιμοποιεί τον αλγόριθμο push για να ενημερώσει του χρήστες (πελάτες) ότι έλαβε ένα νέο update από κάποιον απομακρυσμένο server. Ο ίδιος ο server διατηρεί την cache του συνεπή κάνοντας περιοδικά pulls για νέες εκδόσεις της πληροφορίας. Επομένων, η διαδικασία είναι απλή: Ο cache server ενημερώνεται με pull από τους απομακρυσμένους servers και προωθεί (push) την πληροφορία αυτή στους πελάτες του.

Η εργασία αυτή επικεντρώνεται και στο πότε πρέπει να γίνεται push ένα δεδομένο. Ασχολείται κυρίως με ένα μοντέλο μετοχών. Οπότε για παράδειγμα, δεν γίνεται push στους πελάτες η τιμή μιας μετοχής αν αυτή δεν αλλάξει περισσότερο από ένα δολάριο.

Leasing

Ο όρος lease time πρωτο-χρησιμοποιήθηκε [20] για να επιτύχει συνέπεια cache σε συστήματα κατανεμημένων αρχείων (distributed file systems). Επίσης, χρησιμοποιήθηκε [21] με στόχο να διατηρήσει ισχυρή συνέπεια στον παγκόσμιο ιστό (world wide web). Η ιδέα είναι ότι για να αποφύγουμε τον μεγάλο φόρτο στο internet, χρησιμοποιούμε proxy server. Όμως οι πληροφορίες που γίνονται cached στους proxy servers μπορεί να μην είναι πάντα σωστές. Οι περισσότεροι μηχανισμοί εγγυούνται μόνο *χαλαρή συνέπεια* παρέχοντας σταθερά time-to-live η κάνοντας περιοδικά pulls για να σιγουρεύουν ότι η cached πληροφορία είναι ακόμα έγκυρη.

Η έννοια του leasing προτάθηκε ώστε να διατηρηθεί αυστηρή συνέπεια με μικρό φόρτο μηνυμάτων. Έτσι ο server δίνει ένα χρόνο lease σε κάθε αίτηση κάποιου proxy. Ο χρόνος αυτός υποδηλώνει το χρονικό διάστημα στο οποίο ο server θα ενημερώσει των proxy για πιθανή αλλαγή. Εάν λήξει το lease time ο proxy server πρέπει να επικοινωνήσει και πάλι με τον server για να ανανεώσει το lease παίρνοντας ταυτόχρονα και την νέα έκδοση της πληροφορίας. Έτσι μπορούμε να επηρεάσουμε τον αριθμό των μηνυμάτων που ανταλλάσσονται (ανανεώσεις lease time) χρησιμοποιώντας διαφορετικές τιμές του lease time.

Snooping

Η έννοια του snooping χρησιμοποιήθηκε για θέματα συνέπειας της cache των επεξεργαστών σε πολυεπεξεργαστικά συστήματα [22]. Κάθε επεξεργαστής διατηρεί την προσωπική του cache για λόγους απόδοσης, φυλάει δηλαδή αντίγραφα της κεντρικής μνήμης με στόχο να επιταχύνει την προσπέλαση σε αυτά. Παρόλα αυτά, κάποιος άλλος επεξεργαστής μπορεί να αλλάξει δεδομένα στην κεντρική μνήμη με αποτέλεσμα τα δεδομένα στην cache κάποιου επεξεργαστή να πρέπει ακυρωθούν.

Το πρωτόκολλο snooping εφαρμόστηκε κυρίως σε συστήματα που όλοι οι επεξεργαστές είναι συνδεδεμένοι σε έναν κοινό δίαυλο. Στην αρχιτεκτονική αυτή, ό,τι πληροφορίες ανταλλάσσει κάποιος επεξεργαστής με την μνήμη γίνονται άμεσα γνωστές σε όλους αφού όλοι «ακούν» τον ίδιο δίαυλο . Ό όρος snooping σημαίνει λοιπόν ότι κάποιος επεξεργαστής παρακολουθεί τον δίαυλο και αν διαπιστώσει ότι κάποιος άλλος επεξεργαστής τροποποίησε κάποιο κομμάτι μνήμης που έχει στην cache του τότε ακυρώνει το αντίγραφό του (snooping = χώνω την μύτη μου, επειδή ο κάθε επεξεργαστής «άκουγε» τις επικοινωνίες των άλλων με την μνήμη).

Κεφάλαιο 5

Αλγόριθμοι ενημέρωσης

Ένα σύστημα open MAS είναι δυναμικό και συνεχώς μεταβαλλόμενο: κόμβοι μπορεί να τεθούν εκτός λειτουργίας ή και να μετακινηθούν. Με τον όρο *κινητικότητα* (*mobility*) εννοούμε ότι ένας πράκτορας ή μια υπηρεσία μπορεί να αλλάξει θέση στο δίκτυο. *Θέση* (*location*) ενός κόμβου, μπορεί να θεωρηθεί *η πληροφορία που χρειαζόμαστε για να επικοινωνήσουμε με τον συγκεκριμένο κόμβο.* Για παράδειγμα, θέση μπορεί να είναι η διεύθυνση IP σε ένα φυσικό TCP/IP δίκτυο. Οι μετακινήσεις είναι συχνό φαινόμενο σε πολλά P2Pδίκτυα και κυρίως σε δίκτυα από κινητούς πράκτορες (mobile agents).

Το μοντέλο που περιγράψαμε στο κεφάλαιο 3 είναι αρκετά αποτελεσματικό όσον αφορά την αναζήτηση αλλά απαιτεί το δίκτυο των cache να είναι συνεχώς ενημερωμένο. Όμως η μετακίνηση ενός πράκτορα ή μιας υπηρεσίας μπορεί να προκαλέσει *ασυνέπεια στην cache των υπολοίπων (cache inconsistency)* . Πιο συγκεκριμένα, αν υποθέσουμε ότι κάποιοι πράκτορες έχουν στην cache τους την θέση του πράκτορα A. Όταν ο A μετακινηθεί, οι πράκτορες αυτοί θα έχουν αποθηκευμένη την παλιά θέση του A. Η λανθασμένη αυτή (παλιά) πληροφορία ονομάζεται *μη-έγκυρη εγγραφή στην cache ή invalid cache entry.*

Τα πρόβλημα που δημιουργείται από την μετακίνηση ενός κόμβου είναι αρκετά σημαντικό. Κάθε μετακίνηση δημιουργεί έναν αριθμό μη έγκυρων εγγραφών στην cache και κάθε μη έγκυρο cache entry πρόκειται ουσιαστικά για μια λιγότερη ακμή στον γράφο που φαίνεται στο σχήμα 3.1. Αν έχουμε πολλές μετακινήσεις μπορεί γρήγορα ο γράφος να γίνει μη συνεκτικός δηλαδή να χωριστεί το δίκτυο μας σε παραπάνω από ένα υπο-δίκτυα που δεν συνδέονται μεταξύ τους. Επιπλέον οι αλγόριθμοι αναζήτησης (και γενικότερα οι αλγόριθμοι που βασίζονται στο flooding) δουλεύουν καλύτερα όταν ο βαθμός των κόμβων είναι ο μεγαλύτερος δυνατός και ένα invalid entry σημαίνει ένα connection λιγότερο.

Όμως, το σημαντικότερο πρόβλημα είναι ότι *η νέα θέση του πράκτορα δεν είναι γνωστή σε κανένα* αφού όλοι γνωρίζουν μόνο την παλιά. Αυτό σημαίνει ότι κανείς δεν μπορεί να επικοινωνήσει μαζί του, δηλαδή μετά την μετακίνηση ο πράκτορας είναι αποκομμένος.

Στο κεφάλαιο αυτό προτείνουμε κάποιες μεθόδους για να αντιμετωπίσουμε το πρόβλημα: προτείνουμε αλγορίθμους που φροντίζουν να διατηρηθούν στο ελάχιστο τις μη-έγκυρες εγγραφές στης cache (invalid cache entries). Οι αλγόριθμοι αυτοί ονομάζονται *αλγόριθμοι ενημέρωσης (update algorithms)* αφού φροντίζουν να ενημερώσουν τους πράκτορες του δικτύου για τις μετακινήσεις που έγιναν, ώστε και αυτοί με την σειρά τους να ενημερώσουν τις cache τους με τις νέες θέσεις.

Ουσιαστικά, με το να ενημερώνουμε τις cache των πρακτόρων, ενημερώνουμε τις ίδιες τις συνδέσεις τους με το δίκτυο. Κατά συνέπεια ενημερώνουμε την ίδια την τοπολογία του δικτύου.

Υπάρχουν δυο βασικές κατηγορίες αλγορίθμων: *pull based* που ξεκινούν από τους πράκτορες που έχουν τα invalid cache entries και *push based* που ξεκινούν από τους πράκτορες που μετακινούνται. Στο κεφάλαιο αυτό θα περιγράψουμε τις δύο αυτές βασικές κατηγορίες καθώς και κάποιους αλγορίθμους που προκύπτουν από την συνδυασμένη εφαρμογή τους.

Οι προτεινόμενοι αλγόριθμοι μελετήθηκαν και πειραματικά μέσω εξαντλητικών προσομοιώσεων προκειμένου να διαπιστωθεί η αποτελεσματικότητά τους και οι συγκριτικές επιδόσεις τους. Ένα τμήμα των αποτελεσμάτων αυτού του κεφαλαίου παρουσιάστηκε στο [23]

5.1 Αλγόριθμος pull

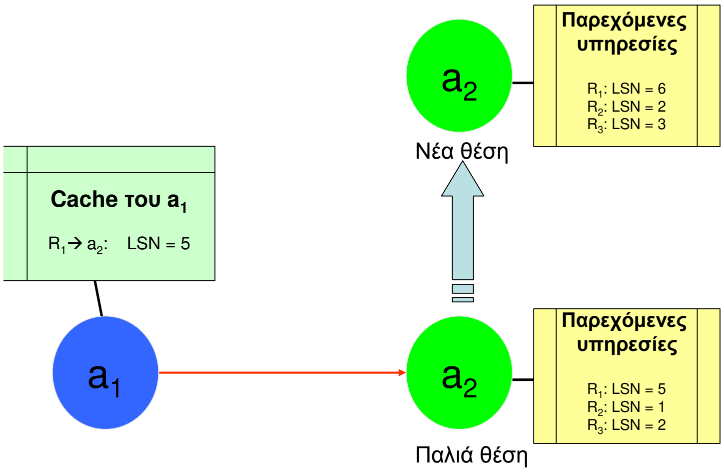
Ο διαδικασία pull ξεκινά από τους πράκτορες που θέλουν να ενημερωθούν για πιθανές αλλαγές που έγιναν. Αν οποιαδήποτε στιγμή κάποιος πράκτορας αντιληφθεί ότι η θέση μιας υπηρεσίας στην cache δεν είναι έγκυρη, ξεκινά μια διαδικασία αναζήτησης στο δίκτυο. Αναζητεί την νέα θέση της υπηρεσίας ώστε να αντικαταστήσει την λανθασμένη εγγραφή. Για την διαδικασία αυτή μπορεί να χρησιμοποιηθεί οποιοσδήποτε από τους αλγορίθμους αναζήτησης που περιγράψαμε στην παράγραφο 3.4.

Αν υποθέσουμε ότι τουλάχιστον ένας πράκτορας A ξέρει την νέα θέση της υπηρεσίας B, τότε εάν το μήνυμα της αναζήτησης φτάσει στον A, ο A θα ενημερώσει τον πράκτορα που ξεκίνησε την αναζήτηση για την σωστή θέση της υπηρεσίας B. Παρόλα αυτά, η διαδικασία αυτή μπορεί να μην επιστρέψει πάντα την σωστή θέση του B. Μπορεί η υπηρεσία B να έχει μετακινηθεί παραπάνω από μία φορά και ο A να γνωρίζει μεν μια νεότερη θέση, αλλά όχι την *πιο πρόσφατη* (αυτή που βρίσκεται τώρα).

Για να αντιμετωπίσουμε το παραπάνω πρόβλημα, για κάθε υπηρεσία κρατάμε έναν αύξοντα αριθμό που συμβολίζει τον αριθμό των μετακινήσεων και τον οποίο ονομάζουμε *location sequence number (LSN)*. Ο αριθμός αυτός αυξάνεται κάθε φορά που μετακινείται η υπηρεσία ή ο πράκτορας. Έτσι, το ότι ένας πράκτορας αναζητά νεότερη θέση κάποιας υπηρεσίας σημαίνει ότι ψάχνει για κάποιον άλλο πράκτορα που έχει στην cache του την υπηρεσία αυτή *με μεγαλύτερο location sequence number* από αυτό που έχει στην δική του .

Για παράδειγμα, όπως βλέπουμε στο σχήμα 5.1 όταν ο a_2 μετακινηθεί αυξάνει το LSN σε όλα τα resources που προσφέρει. Παρατηρούμε όμως ότι ο a_1 γνωρίζει ακόμα την παλιά (μη-έγγυρη) θέση του a_2. Όταν ο a_1 αποφασίσει να κάνει pull για την R_1 θα πρέπει να ψάξει στο δίκτυο για κόμβους που γνωρίζουν νεότερη θέση της υπηρεσίας, δηλαδή για κόμβους που έχουν στην cache τους θέση με $LSN > 5$.

Ο πράκτορας που αναζητά ενδέχεται να λάβει πολλές απαντήσεις από ένα μόνο search query (απο διαφορετικά paths του flooding). Στην περίπτωση αυτή, κρατάει το αποτέλεσμα με το μεγαλύτερο LSN. Επειδή και πάλι μπορεί μεν να πάρει μια πιο πρόσφατη θέση (μεγαλύτερο LSN) αλλά όχι την τελευταία, ίσως χρειαστεί να επαναλάβει την αναζήτηση, χρησιμοποιώντας



Σχήμα 5.1: Μεγαλύτερο location sequence number σημαίνει νεότερη πληροφορία

συνεχώς αυξανόμενο LSN. Δηλαδή, χρησιμοποιεί για την επόμενη αναζήτηση το μεγαλύτερο LSN που επέστρεψε η αμέσως προηγούμενη. Ο αλγόριθμος φαίνεται αναλυτικά στο σχήμα 5.2 Φυσικά, δεν μπορεί να επαναλαμβάνει την αναζήτηση επ῾άπειρο αφού μπορεί ο πράκτορας που προσφέρει την αναζητούμενη υπηρεσία να είναι απλώς εκτός σύνδεσης.

Η διαδικασία pull μπορεί να γίνει με δύο τρόπους. Ένας πράκτορας μπορεί να κάνει pull κάθε φορά που αντιλαμβάνεται ότι η cache του έχει invalid entry, για παράδειγμα, όταν προσπαθήσει να επικοινωνήσει με τον πράκτορα που έχει το resource και αποτύχει. Η διαδικασία αυτή ονομάζεται *pull καθ'απαίτηση ή on-demand pull* . Εναλλακτικά, μπορεί να κάνει pull την cache *περιοδικά (periodic pulling)*. Η διαδικασία αυτή έχει την έννοια του να ανανεώσει την cache προληπτικά, *πριν χρειαστεί να το ανακαλύψει όταν χρειαστεί την υπηρεσία (prefetching)* ώστε να αποφύγει καθυστερήσεις (communication latency). Επιπλέον, με το να επαναφέρουμε περιοδικά την cache όλων των πρακτόρων, κρατούμε όλο το δίκτυο ενημερωμένο με αποτέλεσμα να επιταχύνουμε και τις διαδικασίες αναζήτησης και ενημέρωσης. Παρόλα αυτά, τα περιοδικά pull δεν θα πρέπει να γίνονται πολύ συχνά γιατί ένας μεγάλος αριθμός από pulls μπορεί να οδηγήσει σε υψηλό φόρτο μηνυμάτων στο δίκτυο. Μπορούμε να χρησιμοποιήσουμε στατιστικά ώστε να κάνουμε pull όταν πιστεύουμε ότι ένα μεγάλο ποσοστό της cache κάποιου πράκτορα είναι μη έγκυρη.

```
Αρχικά θέτουμε search sequence = LSN (location sequence number στην cache)
μέχρι να βρεθεί η σωστή θέση
{
   ΑΝΑΖΗΤΗΣΕ
     ή κάποιον πράκτορα που έχει cached τη θέση με LSN > search sequence
     ή τον ίδιο το resource

   κρατάμε την απάντηση με το μεγαλύτερο LSN
   search sequence = το sequence της απάντησης

   αν ο πράκτορας είναι όντως στην θέση που επέστρεψε η αναζήτηση
   {
       ανανεώνουμε την cache (νέα θέση, νέο sequence)
       τερματισμός αλγορίθμου
   }
}
```

Σχήμα 5.2: Αλγόριθμος pull.

5.2 Αλγόριθμος push

Ο αλγόριθμος pull δεν μπορεί να δουλέψει από μόνος του γιατί *απαιτεί τουλάχιστον ένας πράκτορας να γνωρίζει την νέα θέση* της υπηρεσίας που μετακινήθηκε. Είναι φανερό ότι ο πράκτορας που μετακινείται πρέπει να γνωστοποιήσει την αλλαγή αυτή σε έναν αριθμό από άλλους.

Η γενική ιδέα είναι ότι ο πράκτορας που μετακινείται προωθεί (κάνει push) ένα μήνυμα που περιέχει την νέα του θέση στο δίκτυο. Ο στόχος της μετάδοσης αυτής είναι να φτάσει το μήνυμα *σε όσο το δυνατό περισσότερους πράκτορες του δικτύου* ώστε να φτάσει και σε όσο το δυνατό περισσότερους πράκτορες που έχουν στην cache τους την παλιά θέση. Όταν κάποιος λάβει το μήνυμα αυτό, εξετάζει αν χρειάζεται να κάνει κάποιες ενημερώσεις στην cache του και προωθεί το μήνυμα παραπέρα.

Μπορούμε να χρησιμοποιήσουμε οποιαδήποτε μέθοδο flooding επιθυμούμε για να διαδώσουμε αυτή την πληροφορία. Για παράδειγμα, μπορούμε να χρησιμοποιήσουμε κάποια από τις μεθόδους αναζήτησης που περιγράφηκαν στο κεφ 3.4, με κάποιες τροποποιήσεις. Μόνο που τώρα δεν πλημμυρίζουμε το δίκτυο με μηνύματα αναζήτησης αλλά με μηνύματα ενημέρωσης. Οι κόμβοι που λαμβάνουν τα μηνύματα δεν αποστέλλουν απαντήσεις, απλά ενημερώνονται για την αλλαγή και προωθούν το μήνυμα.

5.3 Συνδυασμός push και pull

Παραπάνω, περιγράψαμε δύο αρκετά γενικές μεθόδους που δεν μπορούν να χρησιμοποιηθούν όμως ξεχωριστά: Το pull δεν μπορεί να χρησιμοποιηθεί αν δεν υπάρξει push. Αντίστοιχα, χρησιμοποιώντας μόνο push δεν είναι δυνατό να είμαστε σίγουροι ότι όλοι οι πράκτορες θα λάβουν το μήνυμα τις ενημέρωσης. Αυτό γιατί το μήνυμα μπορεί να μην φτάσει στον πράκτορα όταν αυτός είναι offline ή και να μην φτάσει καθόλου (λόγω TTL κτλ). Ο κατάλληλος συνδυασμός των δύο παραπάνω μεθόδων μπορεί να οδηγήσει σε αποδοτικούς αλγορίθμους ενημέρωσης. Παρακάτω, θα παρουσιάσουμε τρεις διαφορετικές παραλλαγές που χρησιμοποιούν τις μεθόδους push/pull για να επιτύχουν την όσο το δυνατό ευρύτερη και γρηγορότερη ενημέρωση του δικτύου για κάποια αλλαγή.

5.3.1 Απλό push/pull

Στο απλό push/pull ο πράκτορας που μετακινείται απλώς μεταδίδει στο δίκτυο την νέα του θέση. Δηλαδή, στέλνει ένα μήνυμα σε ένα υποσύνολο των γειτόνων του οι οποίοι με την σειρά τους το προωθήσουν σε ένα υποσύνολο των δικών τους γειτόνων κ.ο.κ. Η ιδέα είναι απλή: επειδή ο πράκτορας δεν γνωρίζει ποιοι άλλοι τον έχουν στην cache τους, με το να πλημμυρίσει το δίκτυο με μηνύματα στοχεύει στο να ενημερώσει όσους περισσότερους μπορεί.

Επειδή θέλουμε το μήνυμα να φτάσει σε όσο το δυνατό περισσότερους πράκτορες χρησιμοποιώντας όσο το δυνατό μικρότερο αριθμό μηνυμάτων δεν ταιριάζουν όλοι οι αλγόριθμοι flooding. Ο απλός αλγόριθμος flooding (κεφ 3.4) έχει πολύ καλά αποτελέσματα όσον αφορά το ποσοστό των κόμβων που ενημερώνονται αλλά έχει μεγάλο overhead μηνυμάτων. Αν μειώσουμε κατά πολύ το TTL μπορούμε να περιορίσουμε τον αριθμό των μηνυμάτων αλλά οι απομακρυσμένοι κόμβοι δεν θα ενημερωθούν ποτέ. Από την άλλη, ο αλγόριθμος random paths (κεφ 3.4) έχει ελάχιστο message overhead αλλά οι κόμβοι που επισκέπτονται οι k-walkers είναι λίγοι ενώ η ενημέρωση μπορεί να αργήσει σημαντικά λόγω των μακρινών μονοπατιών που σχηματίζονται. Η μέθοδος που αρμόζει στο συγκεκριμένο πρόβλημα είναι η τροποποιημένη έκδοση του αλγορίθμου *teeming με decay*.

Ο αλγόριθμος *teeming με decay* ταιριάζει περισσότερο στο πρόβλημα γιατί ενημερώνει *γρήγορα* ένα μεγάλο αριθμό από πράκτορες (στα πρώτα βήματα του αλγορίθμου). Επιπλέον, επειδή κόβει τον τον ρυθμό μετάδοσης όσο περνάει ο χρόνος περιορίζει τις επαναμεταδόσεις επιτρέποντας όμως ταυτόχρονα την ενημέρωση απομακρυσμένων κόμβων. Ο χρήση του teeming with decay στον αλγόριθμο push περιγράφεται αναλυτικά στο σχήμα 5.3.

Οι βασικές παράμετροι του αλγορίθμου που επηρεάζουν το *εύρος του push* είναι το TTL και η παράμετρος decay. Ρυθμίζοντας τις δύο αυτές παραμέτρους μπορούμε να ρυθμίσουμε το ποσοστό των κόμβων που θα ενημερωθούν κατά την διαδικασία αυτή. Χρησιμοποιώντας μεγαλύτερο TTL και μικρότερη decay μπορούμε να ενημερώσουμε περισσότερους κόμβους αλλά με αυξημένο κόστος μηνυμάτων. Η χρήση μεγαλύτερου TTL επιτρέπει την ενημέρωση απομακρυσμένων κόμβων ενώ η χρήση μικρότερης παραμέτρου decay εκτός του ότι επιτρέπει την ενημέρωση περισσοτέρων κόμβων, συνεπάγει και την γρηγορότερη ενημέρωση αυτών. Αυτό συμβαίνει γιατί οι κόμβοι στο δέντρο του flooding (σχήμα 3.2) θα έχουν περισσότερα παιδιά και άρα περισσότεροι κόμβοι θα βρίσκονται στα ανώτερα levels του δέντρου.

Είναι φανερό ότι ο αλγόριθμος push δεν εγγυάται ότι όλοι οι κόμβοι θα λάβουν το μήνυμα ενημέρωσης. Αυτό μπορεί να γίνει για αρκετούς λόγους. Για παράδειγμα, μπορεί το TTL να είναι μικρότερο από την απόσταση που απαιτείται για να φτάσουμε τον πράκτορα (hops). Επίσης, μπορεί το μήνυμα να μην φτάσει λόγω της τοπολογίας του δικτύου αφού το δίκτυο μπορεί να είναι μη συνδετικό ή να μην είναι δημοφιλής ο πράκτορας που δεν ενημερώθηκε (όσοι περισσότεροι τον γνωρίζουν τόσο περισσότεροι μπορούν να του προωθήσουν μηνύματα ενημέρωσης). Τέλος, μπορεί να έχασε το μήνυμα ενημέρωσης λόγω του ότι ήταν εκτός λειτουργίας (offline) την ώρα της μετακίνησης.

Όταν κάποιος πράκτορας αντιληφθεί ότι η cache του έχει κάποιο μη έγκυρη εγγραφή, θα πρέπει να εκκινήσει την μέθοδο on-demand pull με στόχο να βρει κάποιον πράκτορα που έχει ενημερωθεί από το push που έχει προηγηθεί. Αν δεν βρει την νέα θέση μετά και από την

```
αν ο πράκτορας μετακινηθεί

   θέτει s = 1
   Για κάθε γείτονα A
      με πιθανότητα (1-d)^s
         στέλνει το μήνυμα M(agent, νέα θέση, s) στον A

-------------------------------------------------------------------------------

αν λάβει μήνυμα M(agent, νέα θέση, s) για την αλλαγή θέσης κάποιου άλλου

   αν έχει κάποια υπηρεσία του μετακινημένου πράκτορα στην cache
      ενημερώνει την cache για την νέα θέση.

   αν s < TTL
      s = s + 1
      Για κάθε γείτονα A
         με πιθανότητα (1-d)^s
            προωθεί το μήνυμα M(agent, νέα θέση, s) στον A
```

Σχήμα 5.3: Αλγόριθμος plain push όπου: s = βάθος ενημέρωσης, d = παράμετρος decay, TTL = time to live

διαδικασία pull, αυτό μπορεί να σημαίνει ότι ο πράκτορας που παρέχει την υπηρεσία είναι απλά εκτός σύνδεσης ή ότι το pull δεν έφτασε *σε κανένα* πράκτορα που γνωρίζει την νέα θέση της υπηρεσίας.

Για να επιτύχουμε σωστή συνέπεια στις cache των πρακτόρων του δικτύου πρέπει να κάνουμε χρήση και των δύο αλγορίθμων συνδυασμένα, χρησιμοποιώντας όμως παραμέτρους που θα μας εγγυηθούν σωστή ενημέρωση με μικρό αριθμό μηνυμάτων. Όμως, το εύρος του push (το πόσο μεγάλο flooding θα κάνουμε) σχετίζεται άμεσα με το εύρος του pull. Δηλαδή, αν για κάθε μετακίνηση κάνουμε ένα πολύ ευρύ push, μπορούμε να εκτελέσουμε ένα πιο περιορισμένο pull λόγω του ότι περιμένουμε ένα μεγάλο ποσοστό των πρακτόρων του δικτύου να έχει έγκυρη cache και άρα να βρούμε πιο εύκολα απάντηση στην αναζήτηση της νέας θέσης. Το ίδιο φυσικά ισχύει και αντίστροφα. Όπως θα δούμε και αργότερα στα πειραματικά αποτελέσματα, χρησιμοποιούμε περιορισμένο push μόνο αν ο ρυθμός των μετακινήσεων είναι πολύ μεγάλος (για να αποφύγουμε τον αυξημένο φόρτο μηνυμάτων).

5.3.2 Push/pull με καταλόγους snooping

Το απλό push/pull μπορεί να βελτιωθεί αν κάθε πράκτορας διατηρεί έναν *κατάλογο των πρακτόρων που μετακινήθηκαν πρόσφατα* σε συνδυασμό με την χρήση περιοδικού pulling (αντί για on-demand). Ο κατάλογος αυτός ονομάζεται *snooping directory*.

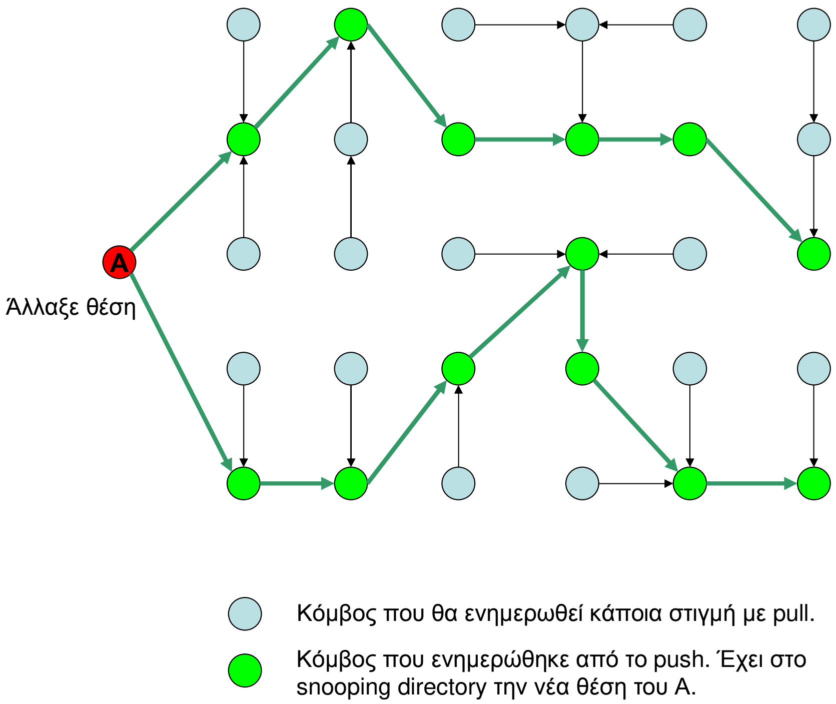
Όπως και στο απλό push/pull, όταν κάποιος πράκτορας μετακινείται απλά κάνει push στο δίκτυο την πληροφορία για την νέα θέση. Η διαφορά του συγκεκριμένου αλγορίθμου έγκειται στο ότι *κάθε πράκτορας* που θα λάβει το μήνυμα push αποθηκεύει την πληροφορία του σε έναν τοπικό κατάλογο ακόμα και αν δεν έχει στην cache κάποια υπηρεσία του μετακινημένου πράκτορα. Ο κατάλογος αυτός ονομάζεται *snooping directory*. Η πληροφορία αυτή αποθηκεύεται μόνο για ένα μικρό χρονικό διάστημα. Πιο συγκεκριμένα, κάθε φορά που προστίθεται ένα entry στο snooping directory λαμβάνει ημερομηνία λήξης (expiration time).

Η διαδικασία του pulling είναι παρόμοια με του προηγουμένου αλγορίθμου, μόνο που τώρα η αναζήτηση δε γίνεται μόνο στις cache αλλά και στους καταλόγους snooping των πρακτόρων που συμμετέχουν στην αναζήτηση. Η χρήση καταλόγων snooping θα ήταν άχρηστη αν κάποιος πράκτορας εφαρμόσει τον αλγόριθμο pull αφού περάσει αρκετό χρονικό διάστημα από την μετακίνηση του πράκτορα που ψάχνει. Πιο συγκεκριμένα, πρέπει να αναζητήσει πριν λήξει το expiration time του push γιατί αλλιώς δεν θα βρει καμιά πληροφορία στα snooping directories. Έτσι, οι πράκτορες θα πρέπει να *ενημερώνουν ολόκληρη την cache τους περιοδικά*. Ο χρόνος μεταξύ δύο διαδοχικών pulls θα πρέπει να είναι μικρότερος ή ίσος από το expiration time των entries στον κατάλογο snooping.

Η ύπαρξη του καταλόγου snooping μας επιτρέπει να χρησιμοποιήσουμε ένα πολύ περιορισμένο pull αφού πλέον δεν χρειάζεται να ψάξουμε σε όλο το δίκτυο κάποιον πράκτορα που να έχει στην cache του την υπηρεσία. Επειδή όλοι οι πράκτορες που λαμβάνουν το μήνυμα push «θυμούνται» για λίγο την νέα θέση της υπηρεσίας, *αρκεί να βρούμε τουλάχιστον ένα πράκτορα από τον οποίο πέρασε μήνυμα push*. Και επειδή οι πράκτορες αυτοί θα είναι πολλοί περισσότεροι, αρκεί ένα πολύ περιορισμένο pull.

Ταυτόχρονα, ο κατάλογος snooping μπορεί να οδηγήσει σε συνεπή cache χρησιμοποιώντας και πολύ περιορισμένο push. Αυτό ισχύει γιατί δεν μας ενδιαφέρει πλέον να ενημερώσουμε όσο το δυνατό περισσότερους πράκτορες αλλά απλά να φτάσει *ένα μήνυμα σε κάθε γειτονιά*. Το περιοδικό pulling που γίνεται θα εγγυηθεί ότι όλοι οι πράκτορες της γειτονιάς θα ενημερωθούν για την νέα θέση.

Για παράδειγμα αν κάθε κόμβος κάνει pull στην 2-hop γειτονιά, αυτό σημαίνει ότι αρκεί να ενημερώσουμε με το push έναν κόμβο σε κάθε 2-hop γειτονιά. Έτσι δεν είναι απαραίτητο να χρησιμοποιήσουμε ένα ευρύ push. Αρκεί να κάνουμε push χρησιμοποιώντας κάποιον «ελαφρύ» αλγόριθμο όπως το k-random paths (κεφ A.1.7). Στο παράδειγμα του σχήματος 5.4 βλέπουμε ότι κάθε πράσινος κόμβος έχει ενημερωθεί από το μήνυμα push. Έτσι σε κάθε πράσινο κόμβο υπάρχει στον κατάλογο snooping η πληροφορία ότι ο κόκκινος κόμβος έχει μετακινηθεί σε δοσμένη θέση. Αυτό αρκεί για να μάθουν και όλοι οι υπόλοιποι κόμβοι την πληροφορία όταν κάποια στιγμή κάνουν periodic-pull.



Σχήμα 5.4: Παράδειγμα push/pull με snooping directories όταν κάνουμε περιοδικό pull σε 2-hop γειτονιά. Όλοι οι κόμβοι σε αυτό το παράδειγμα θα ενημερωθούν

Ρυθμίζοντας σωστά τις παραμέτρους του αλγορίθμου (παράμετροι του push/pull, expiration time, περίοδος pull) ο αλγόριθμος αυτός μπορεί να επιτύχει το ίδιο ποσοστό ενημέρωσης με το απλό push/pull χρησιμοποιώντας πολύ λιγότερα μηνύματα.

5.3.3 Inverted cache push/pull

Για να αποφύγουμε τον μεγάλο όγκο μηνυμάτων που προκαλεί η φάση push, πρέπει να γνωρίζουμε που να στείλουμε τις ενημερώσεις ώστε να μην τις στέλνουμε τυφλά χρησιμοποιώντας κάποιον αλγόριθμο τύπου flooding. Η βασική ιδέα του «ενημερωμένου» push είναι ότι κάθε πράκτορας διατηρεί έναν κατάλογο από τους πράκτορες που έχουν στην cache τους κάποια υπηρεσία του. Διατηρεί δηλαδή μια *ανεστραμμένη cache* (inverted cache directory).

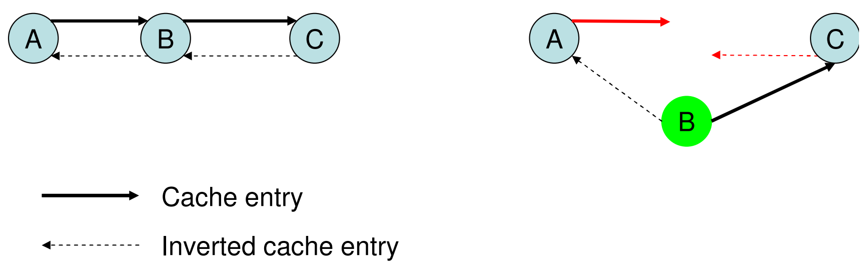
Όταν κάποιος πράκτορας μετακινείται, χρειάζεται να ενημερώσει μόνο τους πράκτορες που βρίσκονται στην ανεστραμμένη cache. Έτσι, όλοι ενημερώνονται *άμεσα*, με τον *βέλτιστο αριθμό μηνυμάτων* ενώ δεν απαιτείται πλέον η χρήση του αλγορίθμου pull (αφού όλοι ενημερώνονται).

Παρόλα αυτά, το να αποθηκεύεται ολόκληρη η λίστα των πράκτορες που γνωρίζουν τον πράκτορα μπορεί να μην είναι αποδοτικό. Για παράδειγμα, υπάρχουν πράκτορες που έχουν δημοφιλείς υπηρεσίες, δηλαδή υπηρεσίες που είναι cached από πολλούς, και ονομάζονται *δημοφιλείς πράκτορες*. Ο κατάλογος της ανεστραμμένης cache στους δημοφιλείς πράκτορες μπορεί να είναι απαγορευτικά μεγάλος με αποτέλεσμα να πρέπει να κρατήσουμε μόνο ένα ποσοστό του. Κατά συνέπεια, αφού δεν κρατείται ολόκληρος ο κατάλογος, με το push θα ενημερωθεί μόνο ένα ποσοστό των ενδιαφερομένων και άρα χρειαζόμαστε και τον αλγόριθμο (on-demand) pull. Ένας επιπρόσθετος λόγος που χρειάζεται το pulling είναι ότι ο πράκτορας μπορεί να μην ενημερωθεί ακόμα και αν είναι στην ανεστραμμένη cache λόγω του ότι ήταν εκτός σύνδεσης την ώρα της μετακίνησης.

Μια βελτίωση του παραπάνω προβλήματος είναι να συνδυαστεί με την ιδέα της μίσθωσης ή leasing. Η μέθοδος leasing προτάθηκε αρχικά στο [20] με στόχο να πιστοποιήσει ότι ένα κατεβασμένο αρχείο δεν έχει αλλαχθεί από τον αρχικό server για ένα συγκεκριμένο (leased) χρονικό διάστημα ώστε να μπορούμε να χρησιμοποιηθεί το κατεβασμένο τοπικό αντίγραφο. Στην δική μας περίπτωση, το διάστημα lease υποδουλώνει το διάστημα στο οποίο ο ιδιοκτήτης μίας υπηρεσίας (resource owner) θα ενημερώσει τον μισθωτή σε περίπτωση που μετακινηθεί. Αν το lease time λήξει, η εγγραφή μπορεί να σβηστεί από την ανεστραμμένη cache και ο πράκτορας να μην ενημερωθεί για την μετακίνηση. Αν ο leaser θελήσει να συνεχίσει και αργότερα να ενημερώνεται, πρέπει να ανανεώσει το lease.

Μία σημαντική διαφοροποίηση του αλγορίθμου αυτού είναι ότι πλέον απαιτείται η αποστολή μηνυμάτων κάθε φορά που ο μισθωτής θέλει να προσθέσει, σβήσει ή και αντικαταστήσει κάποια εγγραφή στην cache. Για παράδειγμα, όταν θελήσει να προσθέσει κάποια υπηρεσία στην cache του πρέπει να επικοινωνήσει με τον ιδιοκτήτη της και να περιμένει την απάντηση η οποία θα περιέχει το lease time. Ομοίως όταν θελήσει να σβήσει/αντικαταστήσει κάποια εγγραφή.

΄Οταν ένας πράκτορας B μετακινείται, πρέπει να ενημερώσει και τους πράκτορες της ανεστραμμένης *αλλά και τους πράκτορες της κανονικής* cache. Στο σχήμα 5.5, ο A έχει στην cache τον B και ο B έχει στην cache τον C. Αυτό σημαίνει ότι ο πράκτορας C έχει στην inverted cache ton B και ο B έχει στην inverted cache τον A. ΄Οταν ο B μετακινηθεί πρέπει να ενημερώσει τον πράκτορα A μέσω της ανεστραμμένης cache ώστε ο A να ανανεώσει την cache του. Επίσης πρέπει να ενημερώσει τον πράκτορα C μέσω της κανονικής του cache ώστε ο C να ανανεώσει την ανεστραμμένη cache του. Πρέπει λοιπόν να στείλουμε μηνύματα στους πράκτορες και των δύο cache.



Σχήμα 5.5: Η μετακίνηση ενός πράκτορα B απαιτεί την ενημέρωση των πρακτόρων και στην ανεστραμμένη αλλά και στην απλή cache.

Επειδή ο ιδιοκτήτης διατηρεί μόνο τις εγγραφές που έχουν έγκυρα lease times, *μικρότερα lease times συνεπάγουν μικρότερους καταλόγους ανεστραμμένης cache.* Η τιμή του lease time που δίνει ο κάθε πράκτορας θα πρέπει να βασίζεται σε δύο παράγοντες: στο πόσο συχνά μετακινείται και στο πόσο δημοφιλής είναι. Αν ο πράκτορας είναι πολύ δημοφιλής πρέπει να έχει μικρό lease time. Αν πάλι ο πράκτορας μετακινείται συχνά θα πρέπει να δίνει μικρά lease times ώστε να ενημερώνει όσο το δυνατό λιγότερους κόμβους κάθε φορά που μετακινείται.

Ο αλγόριθμος αυτός είναι ιδιαίτερα αποδοτικός όταν έχουμε συχνές μετακινήσεις αφού ενημερώνει τους απαιτούμενους πράκτορες άμεσα και με τον μικρότερο δυνατό αριθμό μηνυμάτων. Όμως, απαιτεί την ύπαρξη επιπρόσθετης μνήμης και επιφέρει φόρτο μηνυμάτων όταν έχουμε συχνές προσθήκες/διαγραφές/αλλαγές στις cache. Τέλος, οι ιδιοκτήτες των υπηρεσιών πρέπει να διατηρούν καταλόγους με πληροφορίες για την διευκόλυνση άλλων πρακτόρων, δηλαδή δεσμεύουν πόρους για την εξυπηρέτηση άλλων.

5.4 Πειραματικά αποτελέσματα

5.4.1 Περιγραφή του εξωμοιωτή

Για την αξιολόγηση της συμπεριφοράς των παραπάνω προτεινόμενων αλγορίθμων, γράψαμε ένα πρόγραμμα που εξομοιώνει ένα P2Pδίκτυο από κινητούς πράκτορες. Υλοποιήσαμε και τους τρεις παραπάνω αλγορίθμους στον εξομοιωτή με στόχο να εξάγουμε χρήσιμα συμπεράσματα για την συμπεριφορά τους σε ένα αληθινό δίκτυο P2P. Εδώ θα παρουσιάζουμε συνοπτικά την λειτουργία του ενώ λεπτομέρειες δίνονται στα παραρτήματα B-E.

Στην εξομοίωση αυτή αρχικά δημιουργούμε έναν δίκτυο από από πράκτορες. Κάθε πράκτορας κατέχει έναν αριθμό από υπηρεσίες (resources) τις οποίες μοιράζεται με τους άλλους. Μια υπηρεσία μπορεί να βρεθεί σε έναν μόνο πράκτορα αλλά ένας πράκτορας μπορεί να κατέχει παραπάνω από μια υπηρεσίες. Επιπλέον, κάθε πράκτορας έχει cache σταθερού μεγέθους k στην οποία φυλάσσει διευθύνσεις υπηρεσιών. Ξεκινάμε την εξομοίωση του δικτύου φτιάχνοντας έναν τυχαίο γράφο συνδέσεων: κάθε πράκτορας επιλέγει k υπηρεσίες τυχαία και τις προσθέτει στην cache του.

Σε ένα πραγματικό δίκτυο P2Pκάποιες υπηρεσίες είναι πιο δημοφιλείς από άλλες. Έτσι, για να προσεγγίσουμε περισσότερο τέτοιο δίκτυο, στις παραμέτρους της εξομοίωσης μπορούμε να ορίσουμε έναν αριθμό υπηρεσιών που θεωρούνται δημοφιλείς, καθώς και το πόσο πιο δημοφιλείς είναι σε σχέση με τις άλλες. Εναλλακτικά μπορεί να χρησιμοποιηθεί κάποια κατανομή σύμφωνα με την οποία θα οριστεί η δημοτικότητα των πρακτόρων4 ολόκληρου του δικτύου.

Ένα άλλο σημείο που πρέπει να τονίσουμε είναι ότι αρχικά οι εγγραφές στις cache των πρακτόρων του δικτύου είναι όλες έγκυρες. Έτσι ξεκινάμε από ένα «καλής ποιότητας» δίκτυο. Γιαυτό και στα γραφήματα που θα παραθέσουμε παρακάτω το ποσοστό των έγκυρων cache στην αρχή της εξομοίωσης είναι πάντα εκατό τοις εκατό.

Μετά την αρχικοποίηση του γράφου, ξεκινά η εξομοίωση του κάθε αλγορίθμου ενημέρωσης. Η εξομοίωση μας γίνεται σε βήματα (turn based simulation). Σε κάθε βήμα του simulation ο κάθε πράκτορας μπορεί να εκτελέσει κάποια από τις εξής λειτουργίες :

- Να αλλάξει θέση με δοσμένη πιθανότητα P_{move}.
- Να χρησιμοποιήσει μια υπηρεσία στην cache με πιθανότητα P_{use}. Αν δεν βρεθεί ο πράκτορας που παρέχει αυτή την υπηρεσία τότε ξεκινά αυτομάτως διαδικασία on-demand pull.
- Να προσθέσει/αντικαταστήσει/σβήσει μια υπηρεσία από την cache του me πιθανότητα P_{change}.
- Να εκτελέσει κάποιο περιοδικό pull, ανάλογα με τον αλγόριθμο αναζήτησης που χρησιμοποιούμε.
- Να σβήσει κάποια «ληγμένη» (expired) καταχώρηση από το snooping directory.
- Να σβήσει κάποια «ληγμένη» καταχώρηση (λόγω lease time) από την inverted cache.
- Να μην κάνει τίποτα (sleep).

Μπορούμε να εξομοιώσουμε οποιοδήποτε συνδυασμό των μεθόδων push και pull που περιγράψαμε στο κεφάλαιο 5. Στα πειράματά μας μπορούμε να μεταβάλουμε κάθε παράμετρο που επηρεάζει τα χαρακτηριστικά του δικτύου που προκύπτει, την συμπεριφορά κάθε πράκτορα, καθώς και την απόδοση κάθε αλγορίθμου ώστε να εκτιμήσουμε τις επιπτώσεις των αλλαγών αυτών στο σύστημα. Οι παράμετροι αυτοί είναι:

- Ο αριθμός των πρακτόρων στο σύστημα.

- Το μέγεθος της cache K.
- Τον ελάχιστο, μέγιστο και μέσο αριθμό resources που κατέχει κάθε πράκτορας.
- Την κατανομή των δημοφιλών resources (πόσες είναι, πόσο πιο δημοφιλείς είναι).
- Την πιθανότητα να μετακινηθεί κάποιος πράκτορας σε κάθε γύρο P_{move}.
- Την πιθανότητα να χρησιμοποιήσει ένα resource P_{use}.
- Για τον αλγόριθμο teeming με decay που χρησιμοποιείται στο push/pull (κεφ. 3.4)

  Η παράμετρος decay d.

  Το μέγιστο βάθος TTL.
- Για τον αλγόριθμο push με καταλόγους snooping (κεφ. 5.3.2).

  Expiration time στον κατάλογο snooping.

  Περίοδος του periodic pulling.
- Για τον αλγόριθμο Inverted cache push/pull με leasing.

  Ο χρόνος του lease που εκδίδει ο ιδιοκτήτης της υπηρεσίας.

  Η πιθανότητα να γίνει ανανέωση η μίσθωση πριν λήξει.

Κατά την διάρκεια της εξομοίωσης, διατηρούμε στατιστικά που αφορούν:

- τον φόρτο μηνυμάτων που προκαλείται από τους αλγορίθμους push και pull ξεχωριστά
- το ποσοστό της cache που είναι έγκυρο σε κάθε βήμα
- το ελάχιστο, μέσο και μέγιστο μέγεθος των καταλόγων snooping και inverted cache
- το πόσο γρήγορα (σε πόσα hops) φτάνει η ενημέρωση κατά μέσο όρο κατά το push
- πόσοι πράκτορες μαθαίνουν σε κάθε hop την ενημέρωση (σε πόσους πράκτορες φτάνει το μήνυμα push σε κάθε βήμα του δέντρο (σχήμα 3.2)

Τέλος θα πρέπει να αναφερθεί ότι σε όλες τις εξομοιώσεις, κρατήσαμε σχετικά χαμηλά την πιθανότητα ένας πράκτορας να χρησιμοποιήσει μια υπηρεσία. Αυτό γιατί θέλαμε να κρατήσουμε σε χαμηλό επίπεδο τις επιπτώσεις των on-demand pulls και να επικεντρωθούμε στα αποτελέσματα αυτών καθεαυτών των αλγορίθμων ενημέρωσης.

Τα αποτελέσματα που θα παρουσιάσουμε παρακάτω είναι για ένα δίκτυο 1000 πράκτορες που κατέχουν 3000 υπηρεσίες (resources). Αφήνουμε την εξομοίωση να τρέξει για 250 βήματα. Οι δημοφιλής πράκτορες είναι λίγοι (2%) αλλά είναι δέκα φορές περισσότερο γνωστοί. Τα αποτελέσματα προκύπτουν παίρνοντας τον μέσο όρο αρκετών εκτελέσεων του κάθε πειράματος.

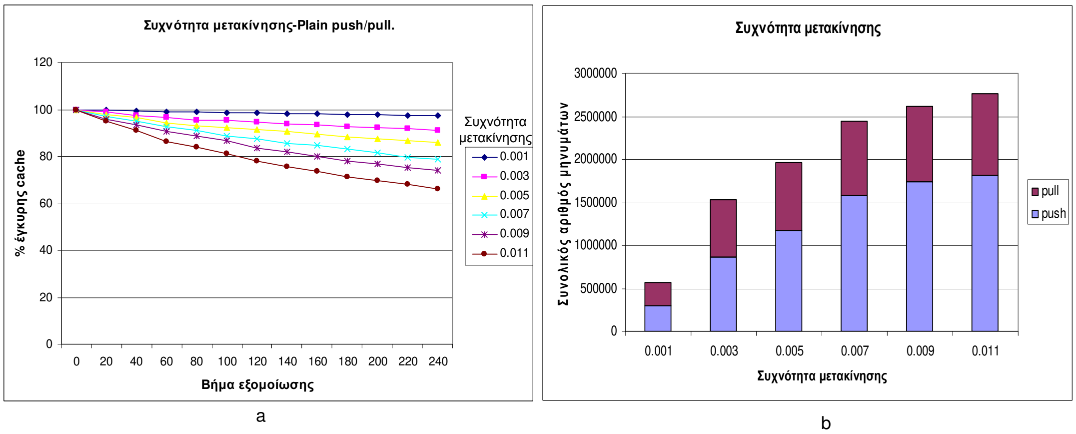
5.4.2 Απλό push/pull

Πειραματιστήκαμε με αρκετούς αλγορίθμους για το flooding του απλού push (π.χ. teeming, random paths). Θα παρουσιάσουμε τα αποτελέσματα χρησιμοποιώντας τον αλγόριθμο teeming με decay που παρουσίασε τα καλύτερα αποτελέσματα όσον αφορά τον φόρτο μηνυμάτων και την αποτελεσματικότητα ενημέρωσης. Χρησιμοποιήσαμε την συνάρτηση decay: $\phi = (1-d)^s$, όπου ϕ είναι η πιθανότητα να προωθήσουμε τον μήνυμα σε έναν γείτονα, s το βάθος του μηνύματος και d η παράμετρος decay. Έτσι το μέγεθος του υποσυνόλου που επιλέγεται κάθε φορά για την προώθηση του μηνύματος μειώνεται εκθετικά με το βάθος της ενημέρωσης. Και μάλλιστα, όσο μεγαλύτερη είναι η παράμετρος decay, τόσο πιο γρήγορα γίνεται αυτή η εκθετική μείωση. Για $d = 0$ ο αλγόριθμος teeming with decay είναι ουσιαστικά ο αλγόριθμος plain flooding. Επειδή το μήνυμα πάντα προωθείται σε τουλάχιστον έναν γείτονα (μέχρι να λήξει το TTL), για $d = 1$ ο αλγόριθμος ταυτίζεται με τον K-random paths.

5.4.2.1 Συχνότητα μετακίνησης (κινητικότητα)

Όπως φαίνεται στο σχήμα 5.6a, όταν οι πράκτορες μετακινούνται συχνά το ποσοστό των έγκυρων εγγραφών στην cache μειώνεται πιο γρήγορα. Αυτό είναι φυσικό αφού μεγαλύτερη κινητικότητα σημαίνει γενικότερα ένα πιο δυναμικό δίκτυο όπου χρειάζεται συχνότερες ενημερώσεις. Έτσι, σε δίκτυα που υπάρχουν πολλές μετακινήσεις, πρέπει να εκτελέσουμε ένα πιο ευρύ push flooding αν θέλουμε να διατηρήσουμε την ποιότητα του δικτύου σε υψηλά επίπεδα.

Όπως φαίνεται στο σχήμα 5.6b η συχνότητα μετακίνησης επιφέρει αύξηση στον αριθμό των



Σχήμα 5.6: απλό push/pull - συχνότητα μετακίνησης

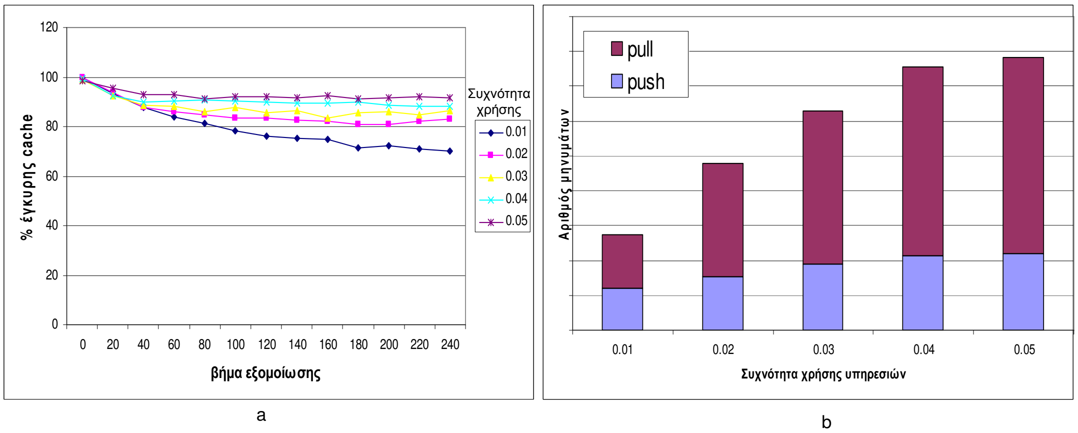
a) Το ποσοστό της cache που παραμένει έγκυρο κατα την διάρκεια της εξομοίωσης (βήμα) για διαφορετικές τιμές της πιθανότητας μετακίνησης. b) Ο συνολικός αριθμός μηνυμάτων που στάλθηκαν για την ενημέρωση για διαφορετικές συχνότητες μετακίνησης. Μεγαλύτερες τιμές στην συχνότητα μετακίνησης σημαίνουν ότι σε κάθε γύρο μετακινούνται περισσότεροι πράκτορες.

μηνυμάτων. Η αύξηση προκαλείται κυρίως λόγω των μηνυμάτων push, αφού κάθε φορά που μετακινείται ένας πράκτορας κάνει push στο δίκτυο. Ο ρυθμός της αύξησης είναι γραμμικός σε σχέση με την συχνότητα μετακίνησης. Παρόλα αυτά, παρατηρούμε ότι μετά από ένα σημείο ο ρυθμός αυτός πέφτει (πχ για τιμές > 0.006). Αυτό συμβαίνει γιατί για μεγάλες συχνότητες μετακίνησης, η ποιότητα του δικτύου δεν είναι πολύ καλή (όπως φαίνεται στο 5.6a) οπότε και οι αλγόριθμοι push/pull δεν μπορούν να δουλέψουν αποτελεσματικά (δεν μπορεί να βρεθεί αρκετά μεγάλο υποσύνολο γειτόνων για να προωθηθεί το μήνυμα).

5.4.2.2 Συχνότητα χρήσης υπηρεσιών

Όπως φαίνεται στο σχήμα 5.7a, όσο περισσότερο χρησιμοποιείται μια υπηρεσία τόσο περισσότερα on-demand pull γίνονται και άρα το ποσοστό των μη-έγκυρων εγγραφών cache είναι μικρότερο. Έτσι συχνότερη χρήση κάποιας υπηρεσίας σημαίνει ότι υπάρχει μικρή πιθανότητα να βρεθεί μια μη-έγκυρη εγγραφή της.

Όπως φαίνεται στο σχήμα 5.7b όσο συχνότερα χρησιμοποιούνται οι υπηρεσίες, τόσο περισσότερες φορές γίνονται on-demand pull και άρα τόσο περισσότερα μηνύματα στέλνονται. Η αύξηση αυτή είναι υπο-γραμμική ως προς την συχνότητα χρήσης, αυτό συμβαίνει κυρίως γιατί όσο μεγαλύτερη είναι η πιθανότητα χρήσης, τόσο καλύτερα ενημερωμένο είναι το δίκτυο και άρα δεν χρειάζονται γραμμικά περισσότερα pulls.



Σχήμα 5.7: απλό push/pull - συχνότητα χρήσης μιας υπηρεσίας
a) Το ποσοστό της cache που παραμένει έγκυρο κατα την διάρκεια της εξομοίωσης (βήμα) για διαφορετικές τιμές της πιθανότητας χρήσης μιας υπηρεσίας (κατά συνέπεια και συχνότητα on-demand pull) b) Ο συνολικός αριθμός μηνυμάτων που στάλθηκαν για την ενημέρωση για διαφορετικές συχνότητες χρήσης.

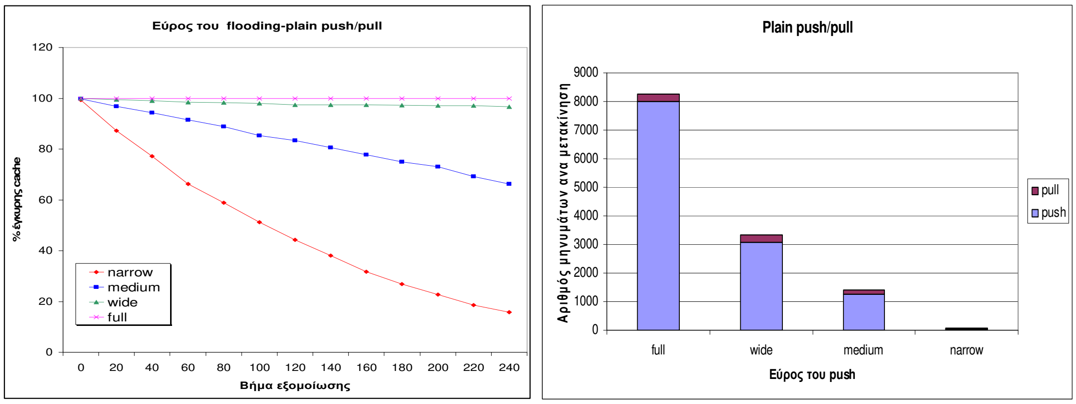
5.4.2.3 Εύρος του flooding

Μεταβάλλοντας την παράμετρο decay και το μέγιστο αριθμό βημάτων t, μπορούνε να ελέγξουμε το *εύρος του push* όπως αναφέραμε στο κεφάλαιο 5.3.1. Χρησιμοποιήσαμε διάφορες τιμές για να εξομοιώσουμε την συμπεριφορά των αλγορίθμων push και για να παρουσιάσουμε τα αποτελέσματα τις χωρίσαμε σε τέσσερις κατηγορίες που φαίνονται στον πίνακα 5.1. Για τον αλγόριθμο on-demand pull χρησιμοποιήσαμε ταν αλγόριθμο k-random paths με $t = 3$.

Εύρος του push	d	t
Full flooding	0.0	5
Wide flooding	0.2	5
Medium flooding	0.3	5
Narrow flooding	0.4	4

Πίνακας 5.1: Οι παράμετροι του push. d = παράμετρος decay, t time to live (TTL) Όσο μικρό είναι το d και όσο πιο μεγάλο το t τόσο πιο «ευρύ» το push flooding.

Στο σχήμα 5.8a βλέπουμε ότι το εύρος του push παίζει σημαντικό ρόλο σε ότι αφορά την ενημέρωση των πρακτόρων για κάποια μετακίνηση. Έτσι αν κάνουμε ένα πολύ περιορισμένο push, στο τέλος της εξομοίωσης θα έχουμε μόλις το 10% των cache ενημερωμένο. Αντίθετα, βλέπουμε ότι για ένα ευρύ push (full, wide) η πλειοψηφία των εγγραφών στην cache παραμένει ενημερωμένη (πάνω από 90%).



a b

Σχήμα 5.8: απλό push/pull - εύρος του push

a) Το ποσοστό της cache που παραμένει έγκυρο κατα την διάρκεια της εξομοίωσης (βήμα) για διαφορετικό εύρος του push.
b) Ο συνολικός αριθμός μηνυμάτων που στάλθηκαν για την ενημέρωση για για διαφορετικό εύρος του push.

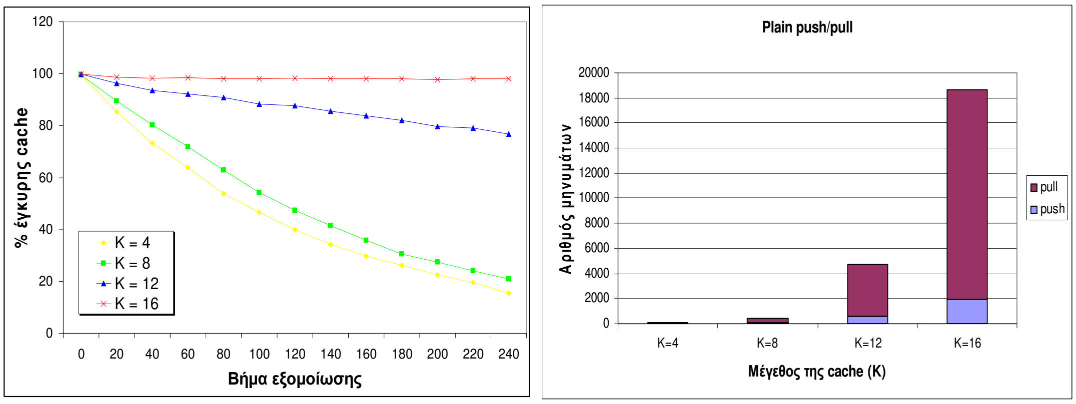
Παρόλα αυτά, ένα ευρύ push σημαίνει και μεγάλο όγκο μηνυμάτων (σχήμα 5.8b). Παρατηρούμε ότι ο αριθμός μηνυμάτων αυξάνεται εκθετικά. Έτσι, ένα «wide» push έχει σχεδόν τα ίδια αποτελέσματα στην ποιότητα της cache με ένα «full» push, όμως η διαφορά στο κόστος των μηνυμάτων είναι πολύ μεγάλη (απαιτούνται λιγότερο από τα μισά μηνύματα). Πρέπει λοιπόν να γίνει σωστή επιλογή των παραμέτρων του flooding.

Τέλος, όταν περιορίζουμε το push, περιορίζουμε ταυτόχρονα και τον αριθμό των μηνυμάτων του pull. Αυτό συμβαίνει γιατί με ένα πολύ περιορισμένο push η ποιότητα του δικτύου είναι αρκετά κακή με αποτέλεσμα να μην μπορεί να δουλέψει αποτελεσματικά ούτε ο αλγόριθμος pull.

5.4.2.4 Μέγεθος της cache

Το μέγεθος της cache επηρεάζει την απόδοση του αλγορίθμου push/pull. Αυτό συμβαίνει γιατί όταν ο κάθε πράκτορας έχει περισσότερους γείτονες (πιο πυκνός γράφος) πρέπει κάθε φορά να προωθεί το μήνυμα σε περισσότερους πράκτορες. Όπως φαίνεται στο σχήμα 5.9a, για πολύ αραιά γραφήματα πρέπει να χρησιμοποιήσουμε πιο ευρύ push αν θέλουμε να έχουμε αποτελεσματική ενημέρωση.

Όμως, σε πυκνά δίκτυα ο αριθμός των μηνυμάτων αυξάνεται και μάλιστα εκθετικά (σχήμα 5.9b). Έτσι καλό είναι να ρυθμίζουμε τις παραμέτρους του push ανάλογα με το



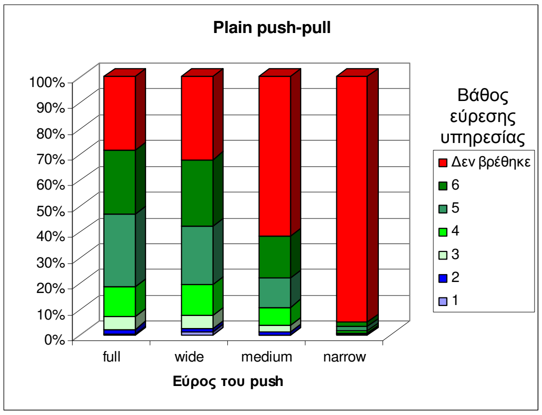
a b

Σχήμα 5.9: απλό push/pull - μέγεθος της cache

a) Το ποσοστό της cache που παραμένει έγκυρο κατα την διάρκεια της εξομοίωσης για διαφορετικό μέγεθος της cache (K). b) Ο συνολικός αριθμός μηνυμάτων που στάλθηκαν για την ενημέρωση για διαφορετικό μέγεθος της cache (K)

μέγεθος του δικτύου.

5.4.2.5 Πιθανότητα εύρεσης μιας τυχαίας υπηρεσίας



Σχήμα 5.10: απλό push/pull - τυχαία αναζήτηση

Το ποσοστό των αναζητήσεων που το αποτέλεσμα βρέθηκε σε βάθος 1, βάθος 2..., καθώς και το ποσοστό των αναζητήσεων που δεν επέστρεψαν σωστό αποτέλεσμα. Οι αναζητήσεις γίνονται μετά το τέλος του simulation του αλγορίθμου ενημέρωσης plain push/pull

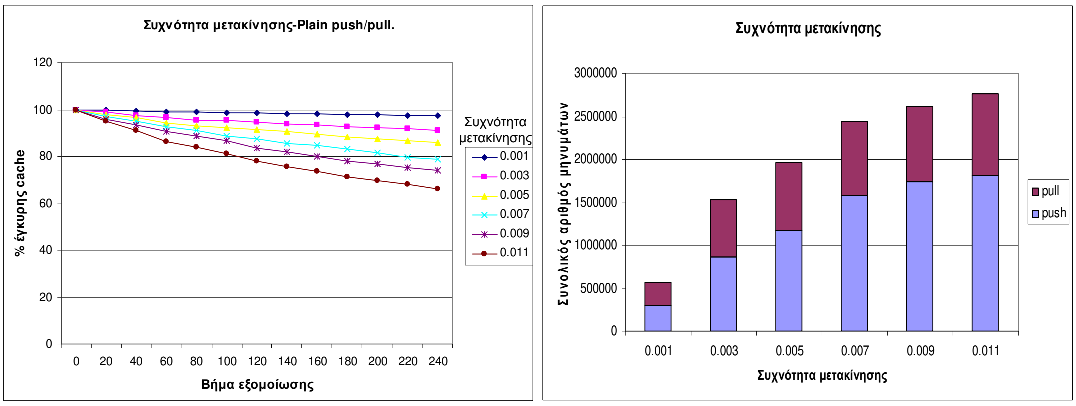
Στο δίκτυο που προκύπτει μετά από το τέλος της εξομοίωσης κάνουμε αναζητήσεις τυχαίων υπηρεσιών από τυχαίους πράκτορες. Θέλουμε να δούμε το πως οι αλγόριθμοι ενημέρωσης επηρεάζουν το δίκτυο όσον αφορά τις αναζητήσεις. Στο σχήμα 5.10 παρουσιάζεται η πιθανότητα να βρούμε τελικά την υπηρεσία και μάλιστα στην περίπτωση που βρεθεί, σε τι βάθος αναζήτησης βρέθηκε. Έτσι, αν χρησιμοποιήσουμε ένα narrow push για την ενημέρωση στο δίκτυο παρατηρούμε ότι η πιθανότητα να βρεθεί κάποια υπηρεσία μετά είναι πολύ μικρή.

Όσο πιο ευρύ push χρησιμοποιούμε τόσο πιο μεγάλη είναι η πιθανότητα να βρούμε κάποια υπηρεσία. Μάλιστα, όσο πιο ευρύ push κάνουμε, τόσο πιο νωρίς θα βρεθεί η αναζητούμενη πληροφορία (στα πρώτα βήματα του αλγορίθμου αναζήτησης).

5.4.3 Push/pull με καταλόγους snooping

Στην εξομοίωση του αλγορίθμου αυτού οι πράκτορες κάνουν περιοδικό pull σε ολόκληρη την 2-hop γειτονιά τους. Χρησιμοποιούμε τον ίδιο αλγόριθμο push με πριν έτσι ώστε να έχουμε συγκρίσιμα αποτελέσματα. Στον αλγόριθμο αυτό, πέρα από τα αποτελέσματα που αφορούν την ποιότητα του δικτύου και τον φόρτο μηνυμάτων, μας ενδιαφέρει και το μέγεθος του καταλόγου snooping.

5.4.3.1 Συχνότητα μετακίνησης (κινητικότητα)

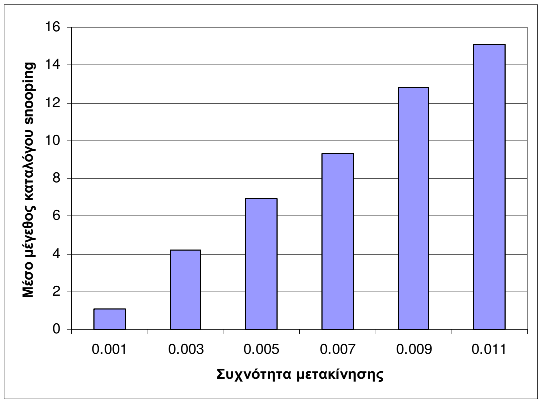


a b

Σχήμα 5.11: push με καταλόγους snooping - συχνότητα μετακίνησης

a) Το ποσοστό της cache που παραμένει έγκυρο κατα την διάρκεια της εξομοίωσης (βήμα) για διαφορετικές τιμές της πιθανότητας μετακίνησης. b) Ο συνολικός αριθμός μηνυμάτων που στάλθηκαν για την ενημέρωση για διαφορετικές συχνότητες μετακίνησης.Μεγαλύτερες τιμές στην συχνότητα μετακίνησης σημαίνουν ότι σε κάθε γύρο μετακινούνται περισσότεροι πράκτορες.

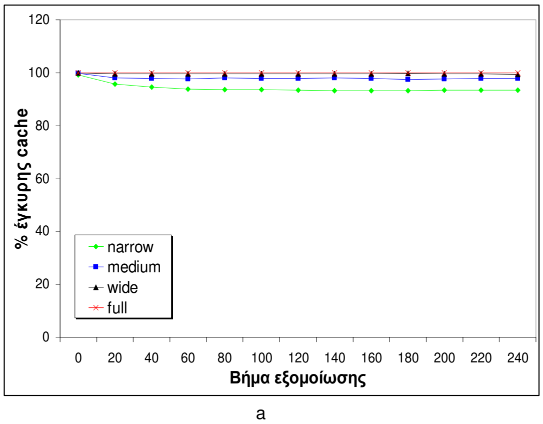
Στο σχήμα 5.11 βλέπουμε ότι ο αλγόριθμος αυτός συμπεριφέρεται παρόμοια με τον προηγούμενο όσον αφορά την κινητικότητα. Όμως, τα καταφέρνει καλύτερα όσον αφορά την ποιότητα του δικτύου αλλά όπως παρατηρούμε το κόστος των μηνυμάτων είναι αυξημένο λόγω των περιοδικών pulls. Συνεπώς, η χρήση του αλγορίθμου αυτού συμφέρει όταν έχουμε αρκετές μετακινήσεις που δικαιολογούν το αυξημένο κόστος του περιοδικού pull (σε σχέση με το push). Αν έχουμε πολύ λίγες μετακινήσεις, ίσως είναι καλύτερο να χρησιμοποιήσουμε απλό push/pull που προκαλεί φόρτο μηνυμάτων κυρίως όταν μετακινούνται πράκτορες.

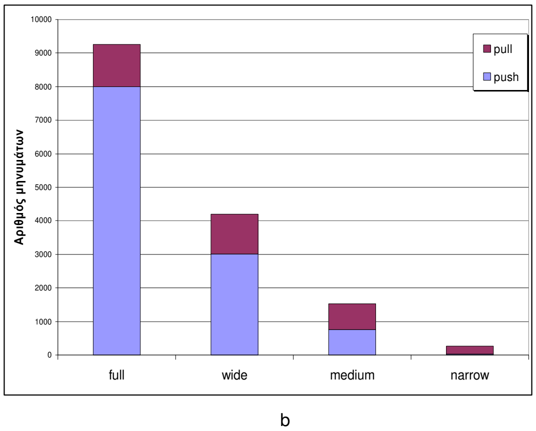


Σχήμα 5.12: push με καταλόγους snooping - συχνότητα μετακίνησης
Το μέσο μέγεθος του καταλόγου snooping που κρατά κάθε πράκτορας για διαφορετικές συχνότητες μετακίνησης

Στο σχήμα 5.12 βλέπουμε ότι το μέγεθος του καταλόγου snooping εξαρτάται από την συχνότητα των μετακινήσεων στο δίκτυο. Αυτό είναι φυσικό αν σκεφτεί κανείς ότι όσο περισσότερες μετακινήσεις έχουμε, τόσο περισσότερες μετακινήσεις θα μαθαίνει κάθε πράκτορας και άρα τόσο περισσότερες εγγραφές θα πρέπει να προσθέτει στους καταλόγους snooping. Η αύξηση αυτή γίνεται με γραμμικό τρόπο.

5.4.3.2 Εύρος του push



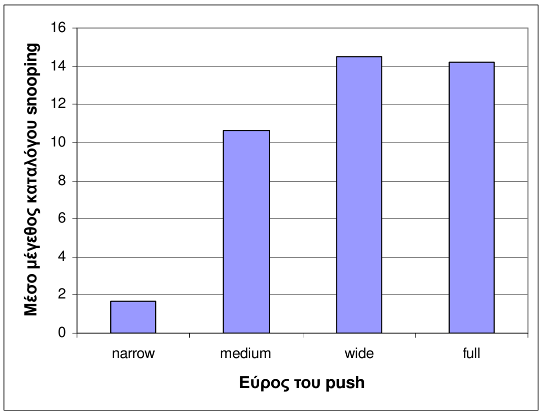


Σχήμα 5.13: push με καταλόγους snooping - εύρος του push
a) Το ποσοστό της cache που παραμένει έγκυρο κατα την διάρκεια της εξομοίωσης (βήμα) για διαφορετικό εύρος του push.
b) Ο συνολικός αριθμός μηνυμάτων που στάλθηκαν για την ενημέρωση για για διαφορετικό εύρος του push.

Το εύρος του push (σχήμα 5.13a) επηρεάζει την ποιότητα του δικτύου με παρόμοιο τρόπο σε σχέση με το απλό push/pull (εικ 5.8a). Η διαφορά εδώ είναι ότι πετυχαίνουμε καλύτερη ποιότητα δικτύου χρησιμοποιώντας τις ίδιες παραμέτρους (πίνακας 5.1). Έτσι βλέπουμε ότι ακόμα και ένα «narrow» push αρκεί για να κρατήσουμε την πλειοψηφία της cache έγκυρη στο τέλος της εξομοίωσης. Εντούτοις, όπως παρατηρούμε στην εικόνα 5.13b ο όγκος των μηνυμάτων που ανταλλάσσονται έχει αυξηθεί σε σχέση με το απλό push/pull (εικ 5.8b). Αυτό συμβαίνει κυρίως λόγω των περιοδικών pull που γίνονται στο δίκτυο.

Αν συγκρίνουμε τα αποτελέσματα του απλού push/pull (σχήμα 5.8) και του push με καταλόγους snooping (εικ 5.13) παρατηρούμε ότι αν χρησιμοποιήσουμε snooping directories και περιοδικά pulls μπορούμε να πετύχουμε την ίδια ποιότητα δικτύου χρησιμοποιώντας λιγότερο ευρύ push. Για παράδειγμα, μπορούμε να χρησιμοποιήσουμε ένα «medium» snooping push και να πετύχουμε την ίδια συνέπεια στην cache με ένα full απλό push/pull. Έτσι πετυχαίνουμε το ίδιο αποτέλεσμα χρησιμοποιώντας μια τάξη λιγότερα μηνύματα όπως φαίνονται και στις εικόνες 5.13b και εικ 5.8b.

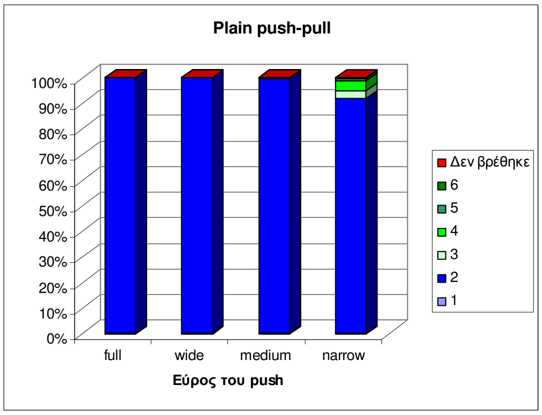
Συμπερασματικά, η χρήση των καταλόγων snooping και των περιοδικών pull μας επιτρέπει να διατηρήσουμε ιδια συνέπεια στις cache των πρακτόρων χρησιμοποιώντας πολύ λιγότερα μηνύματα σε σχέση με το απλό push/pull. Φυσικά υπάρχει το κόστος της μνήμης που απαιτείται για την διατήρηση των snooping directories.



Σχήμα 5.14: push με καταλόγους snooping - εύρος του push
Το μέσο μέγεθος του καταλόγου snooping που κρατά κάθε πράκτορας για διαφορετικές παραμέτρους του push

Στο σχήμα 5.14 βλέπουμε ότι το εύρος του αλγορίθμου push επηρεάζει κατά πολύ το μέγεθος των καταλόγων snooping που κρατά κάθε πράκτορας. Αυτό συμβαίνει γιατί όσο πιο ευρύ push χρησιμοποιούμε, τόσο περισσότερους πράκτορες ενημερώνουμε και άρα τόσο περισσότεροι προσθέτουν εγγραφές στους καταλόγους snooping. Βλέπουμε ότι η αύξηση αυτή είναι αρχικά εκθετική αλλά μετά σταθεροποιείται λόγω των περιορισμών που θέσαμε στον αλγόριθμο snooping (expiration κτλ).

5.4.3.3 Πιθανότητα εύρεσης μιας τυχαίας υπηρεσίας

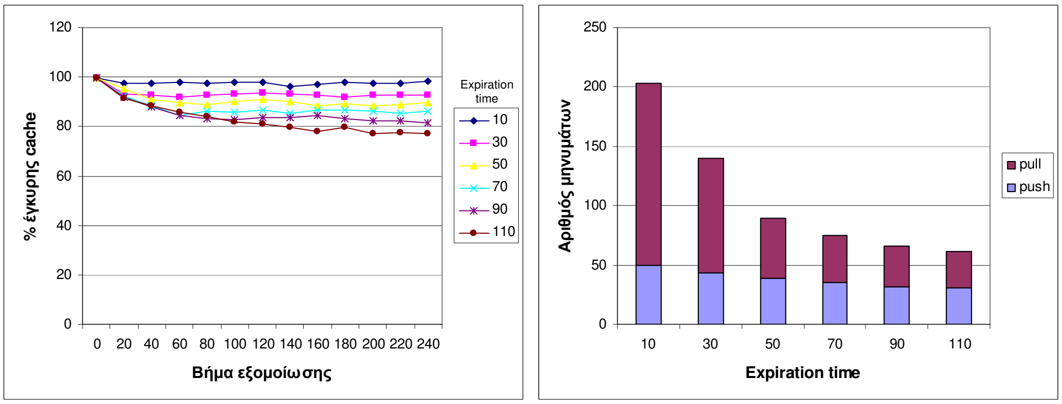


Σχήμα 5.15: push με καταλόγους snooping - τυχαία αναζήτηση
Το ποσοστό των αναζητήσεων που το αποτέλεσμα βρέθηκε σε βάθος 1, βάθος 2..., καθώς και το ποσοστό των αναζητήσεων που δεν επέστρεψαν σωστό αποτέλεσμα. Οι αναζητήσεις γίνονται μετά το τέλος του simulation του αλγορίθμου ενημέρωσης push με καταλόγους snooping

Παρατηρούμε ότι η ύπαρξη καταλόγων snooping οδηγεί σε πολύ καλά αποτελέσματα (σχήμα 5.15). Στις περισσότερες των περιπτώσεων η πληροφορία βρίσκεται στο πρώτο ή και στο δεύτερο βήμα του αλγορίθμου αναζήτησης. Αυτό γίνεται για δύο λόγους: Λόγω της καλύτερης συνέπειας στο δίκτυο και λόγω του ότι μπορεί να βρεθεί πληροφορία και στους καταλόγους snooping. Οι κατάλογοι snooping λειτουργούν δηλαδή σαν ένα είδος προσωρινών αντιγράφων (replicas).

5.4.3.4 Expiration time στον κατάλογο snooping

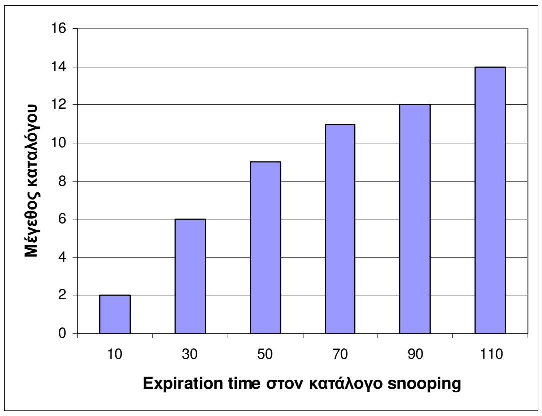
Στο σχήμα 5.16a βλέπουμε ότι το expiration time δεν επηρεάζει κατά πολύ την απόδοση του αλγορίθμου όσον αφορά την ποιότητα των cache. Όμως παίζει σημαντικό ρόλο στον αριθμό μηνυμάτων pull (εικ 5.16b) αφού όσο μικρότερο είναι το expiration time τόσο συχνότερα πρέπει να γίνονται περιοδικά pull. Όπως είναι φυσικό, το expiration time επηρεάζει επίσης το μέγεθος των καταλόγων snooping όπως φαίνεται και στην εικόνα 5.17.



a b

Σχήμα 5.16: push με καταλόγους snooping - expiration time

a) Το ποσοστό της cache που παραμένει έγκυρο κατα την διάρκεια της εξομοίωσης (βήμα) για διαφορετικό expiration time στους καταλόγους snooping. b) Ο συνολικός αριθμός μηνυμάτων που στάλθηκαν για την ενημέρωση για για διαφορετικό expiration time.



Σχήμα 5.17: push με καταλόγους snooping - expiration time

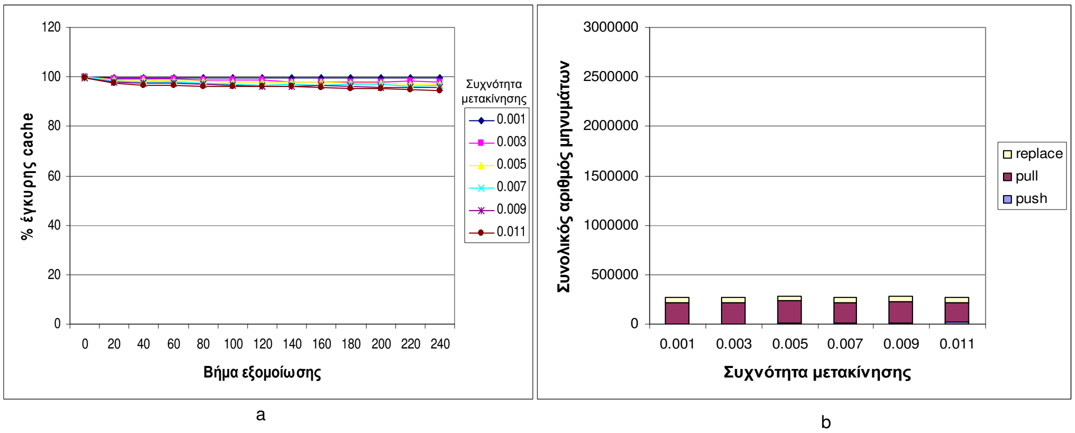
Το μέσο μέγεθος του καταλόγου snooping που κρατά κάθε πράκτορας για διαφορετικές τιμές του expiration time

Έτσι τίθεται ένα θέμα επιλογής. Πρέπει να επιλέξουμε ανάμεσα σε μεγάλο expiration time που σημαίνει λίγα μηνύματα periodic pull αλλά μεγάλους καταλόγους snooping και σε μικρό expiration time που σημαίνει πολλά μηνύματα και μικρούς καταλόγους.

5.4.4 Inverted cache push/pull

Υλοποιήσαμε τον αλγόριθμο αυτό χρησιμοποιώντας και την μέθοδο leasing. Κάθε πράκτορας για να προσθέσει μια υπηρεσία στην cache ενημερώνει τον ιδιοκτήτη της και αυτός απαντά με ένα lease time που είναι σταθερό για όλους τους πράκτορες. Οι πράκτορες δεν ανανεώνουν τα lease times παραμόνο αν χρειαστεί να χρησιμοποιήσουν την υπηρεσία ξανά. Τέλος οι πράκτορες σβήνουν εγγραφές από την inverted cache μόνο αν η μνήμη ξέπεράσει ένα (άνω) κατώφλι.

5.4.4.1 Συχνότητα μετακίνησης (κινητικότητα)

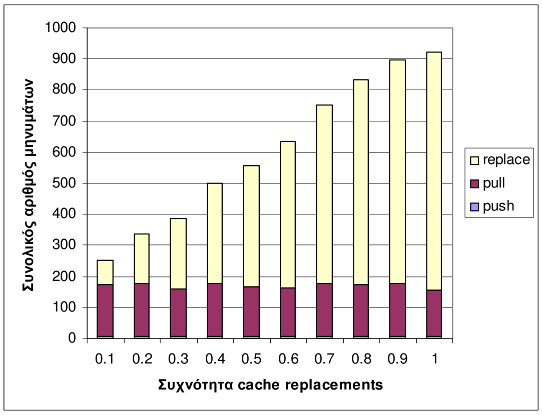


Σχήμα 5.18: inverted cache push/pull - συχνότητα μετακίνησης
a) Το ποσοστό της cache που παραμένει έγκυρο κατα την διάρκεια της εξομοίωσης (βήμα) για διαφορετικές τιμές της πιθανότητας μετακίνησης. b) Ο συνολικός αριθμός μηνυμάτων που στάλθηκαν για την ενημέρωση για διαφορετικές συχνότητες μετακίνησης.Μεγαλύτερες τιμές στην συχνότητα μετακίνησης σημαίνουν ότι σε κάθε γύρο μετακινούνται περισσότεροι πράκτορες.

Στο σχήμα 5.18b παρατηρούμε ότι ο αλγόριθμος αυτός έχει πολύ μικρότερο φόρτο μηνυμάτων σε σχέση με τους προηγούμενους δυο. Έτσι ακόμα και όταν έχουμε πολύ συχνές μετακινήσεις τα μηνύματα που χρειαζόμαστε είναι πολύ λίγα. Ο λόγος που δεν υπάρχει ουσιαστική αύξηση είναι ότι όταν αυξάνονται οι μετακινήσεις, αυξάνονται τα μηνύματα push. Όμως, στον αλγόριθμο αυτό χρειαζόμαστε πολύ λίγα μηνύματα push (ενημερώνουμε μόνο τους πράκτορες στην απλή και στην inverted cache). Τέλος οι μετακινήσεις επηρεάζουν με παρόμοιο τρόπο σε σχέση με τους προηγούμενους αλγορίθμους το ποσοστό των έγκυρων καταχωρήσεων στις cache.

5.4.4.2 Συχνότητα cache replacements

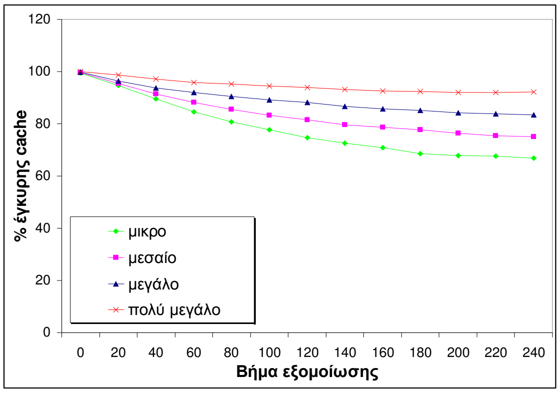
Για πρώτη φορά βλέπουμε ότι οι αλλαγές στην cache (cache replacements) προκαλούν κίνηση μηνυμάτων (σχήμα 5.19). Έτσι όσο πιο συχνές είναι οι αλλαγές στην cache των πρακτόρων, τόσο περισσότερα είναι τα μηνύματα που ανταλλάσσονται. Είναι σημαντικό να σημειωθεί



Σχήμα 5.19: inverted cache push/pull - συχνότητα αλλαγών στην cache
Η συχνότητα των cache replacement επηρεάζει τον φόρτο μηνυμάτων του αλγορίθμου inverted cache push/pull

ότι η αύξηση αυτή γίνεται γραμμικά και ότι γενικά ο αριθμός των μηνυμάτων είναι πολύ μικρός (δεν ξεπερνάει τα 100 συνολικά σε όλη την εξομοίωση).

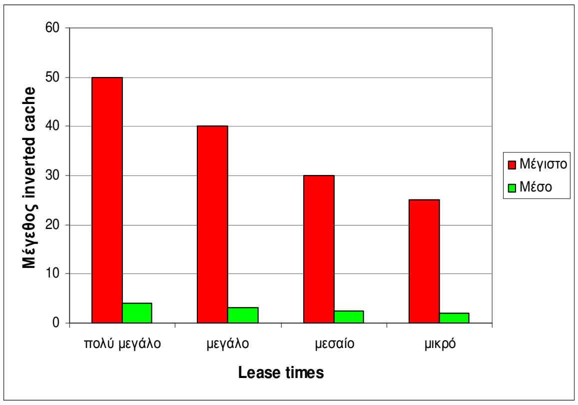
5.4.4.3 Χρόνος Lease time



Σχήμα 5.20: inverted cache push/pull - χρόνος lease
Το lease time επηρεάζει το ποσοστό των έγκυρων cache entries

Το lease time επηρεάζει την ποιότητα του δικτύου (σχήμα 5.20). Όσο μεγαλύτερο το lease time, τόσο περισσότεροι entries κρατά ένας πράκτορας στην inverted cache και άρα τόσο περισσότερους θα ενημερώσει όταν μετακινηθεί. Παρατηρούμε όμως ότι ακόμα και για μικρές τιμές του lease time (5 γύρους) η πλειοψηφία των καταχωρήσεων στις cache παραμένει έγκυρο στο τέλος της εξομοίωσης .

Επίσης ο χρόνος του lease επηρεάζει και το μέγεθος της inverted cache (σχήμα 5.21). Βλέ-

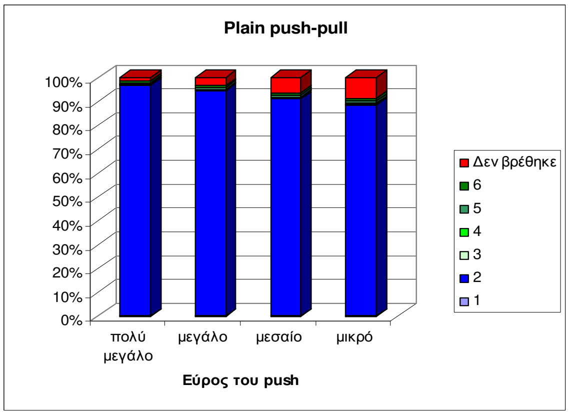


Σχήμα 5.21: inverted cache push/pull - χρόνος lease

Το lease time επηρεάζει το μέγεθος του καταλόγου inverted cache.

πουμε ότι τόσο το μέγιστο όσο και το μέσο μέγεθος της inverted cache επηρεάζεται γραμμικά σε σχέση με τον χρόνο μίσθωσης. Η διαφορά μεταξύ της μέσης και της μέγιστης τιμής οφείλεται στους δημοφιλής πράκτορες.

5.4.4.4 Πιθανότητα εύρεσης μιας τυχαίας υπηρεσίας



Σχήμα 5.22: inverted cache push/pull - τυχαία αναζήτηση

Το ποσοστό των αναζητήσεων που το αποτέλεσμα βρέθηκε σε βάθος 1, βάθος 2..., καθώς και το ποσοστό των αναζητήσεων που δεν επέστρεψαν σωστό αποτέλεσμα. Οι αναζητήσεις γίνονται μετά το τέλος του simulation του αλγορίθμου ενημέρωσης inverted cache

Και ο αλγόριθμος αυτός έχει καλά αποτελέσματα όσον αφορά την πιθανότητα εύρεσης μιας αναζητούμενης πληροφορίας μετά το τέλος μιας αναζήτησης (σχήμα 5.22). Φυσικά, όσο μεγαλύτερα lease time χρησιμοποιήσουμε τόσο πιο συνεπές θα είναι το δίκτυο και άρα τόσο πιο εύκολα θα βρούμε το αναζητούμενο αποτέλεσμα.

5.4.5 Συζήτηση

Όταν δεν θέλουμε να χρησιμοποιήσουμε παραπάνω μνήμη για τους αλγορίθμους ενημέρωσης, τότε η μόνη λύση είναι ο απλός push/pull αλγόριθμος. Ο αλγόριθμος αυτός παρουσιάζει ικανοποιητικά αποτελέσματα (όσον αφορά την συνέπεια των cache) μόνο όταν χρησιμοποιηθεί ένα ευρύ push flooding. Επιπλέον, η απλότητα του αλγορίθμου, το γεγονός ότι κάποιος πράκτορας δεν βασίζεται σε άλλους για να ενημερωθεί (δεν χρειάζεται να υπάρχουν άλλοι που κρατούν «σωστούς» καταλόγους snooping/inverted cache) και το γεγονός ότι αν κάποιος δεν ενημερωθεί μπορεί να στραφεί στον αλγόριθμο on-demand pull, κάνουν τον αλγόριθμο αυτό κατάλληλο για δυναμικά open-MAS συστήματα. *Έτσι, ο αλγόριθμος αυτός μπορεί να εφαρμοστεί σε περιβάλλοντα όπου οι κόμβοι βγαίνουν συχνά εκτός σύνδεσης, υπάρχουν κακόβουλοι πράκτορες και γενικά σε αναξιόπιστα και ασταθή περιβάλλοντα.*

Τα αρνητικά σημεία του αλγορίθμου plain push/pull είναι ο μεγάλος φόρτος μηνυμάτων. Επειδή χρειάζεται ευρύ push με στόχο να *ενημερωθούν όσο περισσότεροι πράκτορες γίνεται*, αναγκαστικά πρέπει να σταλθούν πολλά μηνύματα. Μάλιστα, όπως είδαμε, τα μηνύματα αυτά αυξάνονται πολύ όταν έχουμε μεγάλη κινητικότητα στο δίκτυο. Έτσι το απλό push/pull θα πρέπει να αποφεύγεται σε συστήματα που έχουμε πάρα πολύ συχνές μετακινήσεις υπηρεσιών.

Η ύπαρξη καταλόγων snooping και περιοδικών pull βελτιώνει (κατά πολύ) τον αλγόριθμο plain push/pull. Χρησιμοποιώντας τις ίδιες ακριβώς παραμέτρους με το απλό push/pull, η συνέπεια των εγγραφών στις cache των πράκτορες του του δικτύου είναι αρκετά καλύτερη. Όπως υποδεικνύουν τα πειραματικά αποτελέσματα, ακόμα και ένα «narrow» push αρκεί για να επιτύχουμε ικανοποιητική ποιότητα στην cache. Παρόλο που *για το ίδιο εύρος push* έχουμε μεγαλύτερο φόρτο μηνυμάτων (λόγω των αυξημένων μηνυμάτων που προκαλούνται από το περιοδικό pull), στην πραγματικότητα ο αλγόριθμος αυτός είναι καλύτερος γιατί μας επιτρέπει να επιτύχουμε *την ίδια ποιότητα στην cache χρησιμοποιώντας πιο περιορισμένο push* (εικόνες 5.8 και 5.13). Αυτό αντίστοιχα σημαίνει ότι έχουμε το ίδιο ποιοτικό αποτέλεσμα χρησιμοποιώντας πολύ λιγότερα μηνύματα. Τέλος, λόγω της ύπαρξης των snooping directories, αν χρησιμοποιήσουμε σχετικά μεγάλο expiration time μπορούμε να επιτύχουμε ένα είδος replication κάτι που όπως είδαμε μας επιτρέπει να έχουμε πιο γρήγορα αποτελέσματα στις αναζητήσεις (σχήμα 5.15).

Παρόλα αυτά, ο αλγόριθμος push με καταλόγους snooping απαιτεί την ύπαρξη και την συντήρηση αυτών καθεαυτών των καταλόγων. Έτσι κάθε πράκτορας πρέπει να αφιερώσει ένα μικρό ποσό μνήμης. Το μέγεθος του καταλόγου αυξάνεται με την συχνότητα μετακίνησης, το εύρος των push και το expiration time. Παρόλα αυτά, χρησιμοποιώντας κατάλληλες τιμές του expiration time μπορούμε να περιορίσουμε το μέγεθος του καταλόγου σε συστήματα με μεγάλη κινητικότητα.

Με το να γνωρίζει που να στείλει τα updates, ο αλγόριθμος inverted cache push/pull επιφέρει ελάχιστο φόρτο μηνυμάτων. Επειδή τα μηνύματα push είναι σχεδόν αμελητέα ο αλγόριθμος αυτός είναι κατάλληλος για συστήματα όπου έχουμε πολύ συχνές μετακινήσεις. Επιπλέον η ενημέρωση αυτή μεταδίδεται στιγμιαία (σε ένα βήμα, hop). Όπως είδαμε, μπορούμε να κρατήσουμε την cache συνεπή χρησιμοποιώντας μικρές τιμές lease time κάτι που σημαίνει ότι δεν χρειάζεται να κρατούμε μεγάλους καταλόγους inverted cache.

Ωστόσο, η μέθοδος αυτή έχει και αρκετά μειονεκτήματα. Καταρχήν, και η μέθοδος αυτή απαιτεί την χρήση μνήμης για την διατήρηση των καταλόγων inverted cache η οποίοι μπορεί να γίνουν αρκετά μεγάλοι για πράκτορες που κατέχουν δημοφιλείς υπηρεσίες. Επιπλέον, αυτή είναι η μοναδική μέθοδος που οι αλλαγές στην cache των πρακτόρων επιφέρει φόρτο μηνυμάτων (cache replacements). Το κόστος για κάθε αλλαγή δεν είναι υψηλό (δύο μηνύματα), αλλά ίσως να μη συμφέρει να χρησιμοποιήσουμε τον συγκεκριμένο αλγόριθμο αν στο δίκτυο συμβαίνουν *πολλές αλλαγές στις cache και ελάχιστες μετακινήσεις*. Τέλος, ο αλγόριθμος αυτός δεν είναι κατάλληλος για αναξιόπιστα συστήματα open-MAS. Αυτό γιατί ο κάθε πράκτορας βασίζεται σε άλλους στο να τον ενημερώσουν. Έτσι ο αλγόριθμος αυτός δεν μπορεί να δουλέψει αποτελεσματικά όταν υπάρχουν αναξιόπιστοι και μη διαθέσιμοι (offline) κόμβοι.

Κεφάλαιο 6

Δημιουργία και ενημερώσεις αντιγράφων

Σε όλα τα συστήματα P2Pγίνονται αναζητήσεις υποθέτοντας ότι η αναζητούμενη πληροφορία μπορεί να βρεθεί σε έναν αριθμό από κόμβους. Έτσι υπάρχουν πολλαπλά *αντίγραφα (replicas)* του ίδιου αντικειμένου τα οποία προσφέρονται από διαφορετικούς κόμβους*(replication)*. Σε συστήματα όπως το δίκτυο Gnutella υπάρχει σαφής τρόπος δημιουργίας αντιγράφων: μόνο οι κόμβοι που χρησιμοποιούν κάποιο αντικείμενο το αντιγράφουν τοπικά. Συστήματα όπως το Freenet επιτρέπουν ένα αντικείμενο να αντιγραφεί σε κάποιον κόμβο ακόμα και αν ο κόμβος δεν το έχει χρησιμοποιήσει ποτέ.

Όμως, δημιουργείται ένας αριθμός από βασικά ερωτήματα: πόσα αντίγραφα κάθε αντικειμένου πρέπει να δημιουργηθούν ώστε το κόστος των αναζητήσεων να βελτιστοποιηθεί, δεδομένου ότι ο αποθηκευτικός χώρος για όλα τα αντικείμενα στο δίκτυο είναι σταθερός; Επιπλέον, πώς μπορούμε να κρατήσουμε τα αντίγραφα ενός αντικειμένου συνεπή αν υποθέσουμε ότι το αρχικό αντικείμενο μπορεί να αλλάξει;

Στο κεφάλαιο αυτό θα μελετήσουμε ποια πολιτική δημιουργίας αντιγράφων είναι βέλτιστη ως προς το κόστος των αναζητήσεων. Έπειτα, θα προσπαθήσουμε να εφαρμόσουμε τους αλγορίθμους που παρουσιάσαμε στο προηγούμενο κεφάλαιο με στόχο την δημιουργία αλλά και την συντήρηση των αντιγράφων αυτών.

6.1 Δημιουργία αντιγράφων

Στο σημείο αυτό θα παρουσιάσουμε την βασική θεωρία που αφορά την δημιουργία αντιγράφων σε ένα δίκτυο P2P. Θα ανακεφαλαιώσουμε μέρος της μελέτης που έχει παρουσιαστεί στα [13, 18].

Ας υποθέσουμε ότι έχουμε n κόμβους και m αντικείμενα. Κάθε αντικείμενο i αντιγράφεται σε r_i τυχαίους κόμβους. Έτσι, συνολικά στο δίκτυο αποθηκεύονται $\sum_{i=1}^{m} r_i = R$ αντικείμενα. Υποθέτουμε ότι οι στρατηγικές δημιουργίας αντιγράφων είναι τέτοιες ώστε $1 << r_i \leq n$. Τέλος, υποθέτουμε ότι ένα αντικείμενο i αναζητείται με σχετικό ρυθμό q_i έτσι ώστε $\sum_i q_i = 1$ (π.χ. αν όλα τα αντικείμενα αναζητούνται με τον ίδιο ρυθμό $q_i = \frac{1}{m} \forall i$).

Στο [13] πρότειναν ότι ο αριθμός των αντιγράφων που δημιουργείται θα πρέπει να ακολουθεί κάποια από τις ακόλουθες κατανομές:

- *Ομοιόμορφη*: Όλα τα αντικείμενα έχουν τον ίδιο αριθμό αντιγράφων:

$$\text{Ομοιόμορφη: } r_i \propto \frac{R}{m} \tag{6.1}$$

- *Αναλογική*: Ο αριθμός των αντιγράφων μιας υπηρεσίας i εξαρτάται από το πόσο δημοφιλές είναι *όσον αφορά τις αναζητήσεις*. Αν θεωρήσουμε ότι κατά μέσο όρο γίνονται q_i ερωτήσεις για το αντικείμενο αυτό τότε:

$$\text{Αναλογική: } r_i \propto q_i \tag{6.2}$$

- *Τετραγωνικής ρίζας*: Ο αριθμός των αντιγράφων ενός αντικειμένου i είναι ανάλογος της τετραγωνικής ρίζας των ερωτήσεων που γίνονται για αυτό:

$$\text{Τετραγωνικής ρίζας: } r_i \propto \sqrt{q_i} \tag{6.3}$$

Για να αναζητήσουμε ένα αντικείμενο σε ένα σύστημα P2Pεκτελούμε συνήθως κάποιον αλγόριθμο αναζήτησης που βασίζεται στην μέθοδο flooding. Αυτό ισοδυναμεί με το να ρωτήσουμε τυχαία έναν αριθμό από κόμβους μέχρι να βρούμε το αντικείμενο που μας ενδιαφέρει. Ρωτώντας τυχαία κόμβους, η πιθανότητα $Pr_i(k)$ να βρεθεί κάποιο αντικείμενο στον k-οστό κόμβο είναι:

$$Pr_i(k) = \frac{r_i}{n}(1 - \frac{r_i}{n})^{k-1}$$

Δηλαδή, κατά μέσο όρο πρέπει να ερωτηθούν $A_i = \frac{n}{r_i}$ κόμβοι μέχρι να βρεθεί το αντικείμενο. Οπότε, το *μέσο μέγεθος της αναζήτησης* για ένα αντικείμενο είναι αντιστρόφως ανάλογο με τον αριθμό των αντιγράφων του, κάτι που σημαίνει ότι όσο περισσότερα αντίγραφα υπάρχουν τόσο λιγότερους κόμβους θα χρειαστεί να ρωτήσουμε.

Αυτό που προσπαθούμε να βελτιώσουμε είναι *το μέσο μέγεθος όλων των αναζητήσεων στο δίκτυο* A, όπου $A = \sum_i q_i A_i = n \sum_i \frac{q_i}{r_i}$. Αν δεν υπήρχε κάποιο όριο στον αριθμό αντιγραφων r_i, τότε η βέλτιστη στρατηγική θα ήταν να αντιγράφουμε τα πάντα παντού αφού αν $r_i = n$, όλες οι αναζητήσεις γίνονται τετριμμένες. Αντιθέτως, υποθέτουμε ότι ο μέσος αριθμός αντιγράφων σε κάθε κόμβο $\rho = \frac{R}{n}$ είναι σταθερός και μικρότερος από m. Οπότε η βασική ερώτηση είναι πως να κατανείμουμε αυτά τα R αντίγραφα των m αντικειμένων.

Η πιο απλή στρατηγική είναι να δημιουργήσουμε τον ίδιο αριθμό αντιγράφων για κάθε αντικείμενο: $r_i = \frac{R}{m}$. Αυτή η στρατηγική καλείται *ομοιόμορφη* και σε αυτή την περίπτωση το μέσο μέγεθος της αναζήτησης είναι:

$$A_{\text{ομοιομορφη}} = \sum_i q_i \frac{m}{\rho} = \frac{m}{\rho}$$

το οποίο δεν εξαρτάται από την κατανομή των ερωτήσεων στο δίκτυο.

Είναι προφανές ότι το να κατανέμουμε τα αντίγραφα ομοιόμορφα, ακόμα και αυτά που δεν αναζητούνται συχνά, δεν είναι αποδοτικό. Μια πιο φυσική πολιτική θα ήταν κάποιος κόμβος που αναζητά ένα αντικείμενο να το αντιγράφει και τοπικά αφού το βρει. Έτσι ο αριθμός των αντιγράφων είναι *ανάλογος* με τον αριθμό των ερωτήσεων: $r_i = Rq_i$. Η στρατηγική αυτή θα πρέπει να μειώσει το μέγεθος των αναζητήσεων που αφορούν δημοφιλή αντικείμενα.

Παρόλα αυτά, μπορεί να αποδειχθεί ότι το μέσο μέγεθος αναζήτησης παραμένει το ίδιο:

$$A_{\text{αναλογικη}} = n \sum_i \frac{q_i}{Rq_i} = \frac{m}{\rho} = A_{\text{ομοιομορφη}}$$

Συμπερασματικά, η ομοιόμορφη και η αναλογική κατανομή των αντιγράφων έχουν ακριβώς το ίδιο αποτέλεσμα στο μέσο μέγεθος αναζήτησης και το μέγεθος αυτό είναι ανεξάρτητο της κατανομής των ερωτήσεων. Στην αναλογική κατανομή το μέσο μέγεθος αναζήτησης για κάθε αντικείμενο διαφέρει με τα δημοφιλή αντικείμενα να έχουν μικρότερο μέσο φόρτο σε σχέση με τα λιγότερο δημοφιλή. Οπότε αντικείμενα που ο ρυθμός αναζήτησης είναι παραπάνω από τον μέσο όρο $\frac{1}{m}$ επωφελούνται από την αναλογική κατανομή και τα υπόλοιπα από την ομοιόμορφη.

6.1.1 Κατανομή τετραγωνικής ρίζας

Έχοντας ως δεδομένο ότι η ομοιόμορφη και η αναλογική πολιτική δημιουργίας αντιγράφων έχουν το ίδιο μέσο μέγεθος αναζήτησης, δημιουργείται το ερώτημα του ποια είναι η καλύτερη πολιτική δέσμευσης αντιγράφων έτσι ώστε να ελαχιστοποιηθεί το μέσο μέγεθος αναζήτησης. Έχει αποδειχθεί ότι η βέλτιστη πολιτική είναι η δημιουργία αντιγράφων που ακολουθούν την κατανομή *τετραγωνικής ρίζας (Square-Root replication)* [24].

Έτσι τα αντίγραφα ενός αντικειμένου είναι ανάλογα ως προς την τετραγωνική ρίζα των αναζητήσεων του: $r_i = \lambda\sqrt{q_i}$, όπου $\lambda = \frac{R}{\sum_i \sqrt{q_i}}$. Σε αυτή την περίπτωση, το μέσο μέγεθος αναζήτησης είναι

$$A_{\beta\varepsilon\lambda\tau\iota\sigma\tau\eta} = \frac{1}{\rho}(\sum_i \sqrt{q_i})^2$$

6.2 Επίτευξη κατανομής τετραγωνικής ρίζας

Ενώ η ομογενής και η αναλογική κατανομή απέχουν πολύ από την βέλτιστο όσον αφορά το μέσο μέγεθος αναζήτησης, έχουν το πλεονέκτημα ότι είναι εύκολο να υλοποιηθούν σε ένα κατανεμημένο σύστημα. Η ομογενής κατανομή ορίζει την δημιουργία σταθερού αριθμού αντιγράφων για κάθε αντικείμενο ενώ η αναλογική κατανομή ορίζει την δημιουργία σταθερού αριθμού αντιγράφων μετά από κάθε αναζήτηση του. Το βασικό ερώτημα είναι αν μπορούμε να επιτύχουμε την βέλτιστη κατανομή τετραγωνικής ρίζας σε ένα πλήρως κατανεμημένο σύστημα.

Επειδή δεν μπορούμε να γνωρίζουμε τον ρυθμό q_i με τον οποίο γίνονται ερωτήσεις για κάθε αντικείμενο i, προτάθηκε στο [13] μια ευριστική μέθοδος για να προσεγγιστεί η βέλτιστη κατανομή τετραγωνικής ρίζας.

Έστω σε κάθε αναζήτηση ενός αντικειμένου γνωρίζουμε τον αριθμό των κόμβων που ερωτήθηκαν x. Όπως αναφέρθηκε, ο μέσος όρος τον κόμβων αυτών είναι $x = \frac{n}{r_i}$ δηλαδή είναι αντιστρόφως ανάλογος του αριθμού των αντιγράφων του αντικειμένου. Η μέθοδος αυτή ορίζει ότι αμέσως μετά την αναζήτηση δημιουργούμε cx αντίγραφα, όπου c σταθερά. Άρα μετά από κάθε επιτυχημένη αναζήτηση ενός αντικειμένου i δημιουργούνται κατά μέσο όρο $c\frac{n}{r_i}$ αντίγραφα. Όπως διαπιστώθηκε πειραματικά η μέθοδος αυτή δημιουργεί αντίγραφα που προσεγγίζουν την κατανομή τετραγωνικής ρίζας, υποθέτοντας ότι το σύστημα είναι σε σταθερή κατάσταση (δηλαδή ταυτόχρονα με την δημιουργία νέων αντιγράφων διαγράφονται τα παλιά με ίδιο ρυθμό).

Σε αυτή την κατεύθυνση, σημαντικό είναι το ζήτημα της πολιτικής διαγραφών ώστε να οδηγηθούμε σε *σταθερή κατάσταση (steady state)*. Υποθέτουμε ότι με τον χρόνο κάποια αντίγραφα εξαφανίζονται και νέα παίρνουν την θέση τους. Για να διατηρηθεί το σύστημα σε σταθερή κατάσταση, ο ρυθμός των διαγραφών θα πρέπει να είναι ο ίδιος με τον ρυθμό της δημιουργίας νέων αντιγράφων. Υπάρχουν αρκετές πολιτικές διαγραφής. Μερικές πολιτικές ορίζουν ότι οι διαγραφές είναι ανεξάρτητες ως προς το id του αντιγράφου η την συχνότητα ερωτήσεων που το αφορούν. Έτσι αναθέτουν έναν σταθερό χρόνο λήξης στο κάθε αντίγραφο και κατά συνέπεια τα αντίγραφα αντικαθίστανται με την λογική First In First Out (FIFO). Σε άλλες περιπτώσεις, λαμβάνεται υπ'όψη πληροφορίες όπως το πιο αντίγραφο χρησιμοποιήθηκε παλιότερα (Least Recently Used, LRU), το ποιο αντίγραφο χρησιμοποιείται πιο σπάνια (Least Frequently Used, LFU). Οποιαδήποτε από τις παραπάνω πολιτικές μπορεί να εφαρμοστεί στην κατανομή τετραγωνική ρίζας με στόχο να περιορίσουμε τον συνολικό αριθμό αντιγράφων στο σύστημα.

6.2.1 Πρακτικοί αλγόριθμοι

Παρά την προσεγγιστική λύση που παρουσιάστηκε στην προηγούμενη παράγραφο, οπού δημιουργούσαμε αντίγραφα που είναι ανάλογα τον κόμβων που ερωτήθηκαν, σε ένα δυναμικό σύστημα δεν είναι εύκολο να γίνει γνωστός ο αριθμός τον κόμβων που ερωτήθηκαν. Έτσι προτάθηκαν κάποιες πολιτικές δημιουργίας αντιγράφων που είναι εύκολες να υλοποιηθούν. Η πρώτη πολιτική ονομάζεται «owner replication» στην οποία δημιουργείται ένα αντίγραφο μόνο στον κόμβο που ξεκίνησε την αναζήτηση. Η δεύτερη πολιτική είναι το «path replication» όπου το αντικείμενο αποθηκεύεται σε όλους του κόμβους κατά μήκος του μονοπατιού από τον αναζητητή ως τον παροχέα. Η πρώτη πολιτική χρησιμοποιείται σε συστήματα όπως στο Gnutella και η δεύτερη σε συστήματα όπως το Freenet. Μια βελτίωση της δεύτερης μεθόδου είναι η «random replication» στην οποία το αντικείμενο αντιγράφεται σε κάποιους τυχαίους κόμβους του μονοπατιού.

6.3 Εφαρμογή αλγορίθμων push/pull για την δημιουργία αντιγράφων

Όπως αναφέραμε παραπάνω, η βέλτιστη πολιτική δημιουργίας αντιγράφων ενός αντικειμένου είναι να δημιουργούμε αντίγραφα που είναι ανάλογα της τετραγωνικής ρίζας των ερωτήσεων που γίνονται για αυτό. Υποθέσαμε ότι για να επιτύχουμε αυτή την πολιτική πρέπει σε κάθε αναζήτηση να δημιουργούμε αντίγραφα που είναι ανάλογα με τον αριθμό τον κόμβο που ερωτήθηκαν.

Κάθε αναζήτηση στο δίκτυο μπορεί να θεωρηθεί ως διαδικασία pull. Επειδή δεν γνωρίζουμε πόσοι κόμβοι ερωτήθηκαν κατά το pull, μπορούμε να δημιουργήσουμε τα αντίγραφα αυτά χρησιμοποιώντας κατάλληλα τον αλγόριθμο push. Για να προσεγγίσουμε τον αριθμό κόμβων που ερωτήθηκαν, αμέσως μετά από μια επιτυχημένη αναζήτηση κάποιου αντικειμένου κάνουμε push στο δίκτυο ώστε να δημιουργηθούν αντίγραφά του. Όταν ένας κόμβος λάβει το μήνυμα push μπορεί (με κάποια πιθανότητα) να δημιουργήσει ένα αντίγραφο του δεδομένου ή/και να προωθήσει το μήνυμα παραπέρα. Ονομάζουμε την μέθοδο αυτή *pull then push*.

Επειδή θέλουμε ο αριθμός τον κόμβων που θα λάβουν το μήνυμα του push να είναι περίπου ίσος με τον αριθμό των κόμβων που ερωτήθηκαν κατά το pull, το TTL του αλγορίθμου push θα πρέπει να ισούται με το βάθος στο οποίο βρέθηκε η πληροφορία κατά το προηγηθέν pull. Με αυτόν τον τρόπο αν το αντικείμενο βρέθηκε μετά από πολλά βήματα τότε θα αντιγραφεί σε περισσότερους κόμβους κατά το push. Έτσι, τα αντίγραφα είναι ανάλογα με τον αριθμό των κόμβων που ερωτήθηκαν και άρα επιτυγχάνουμε την προσέγγιση της κατανομής τετραγωνικής ρίζας.

6.4 Πειραματικά αποτελέσματα

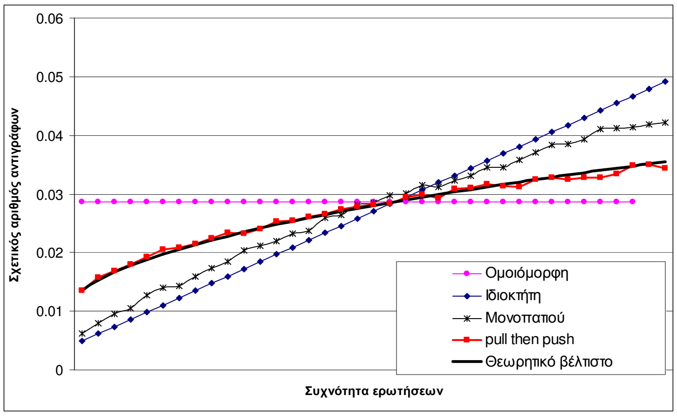
Για να εξετάσουμε την συμπεριφορά της προτεινόμενης μεθόδου, τροποποιήσαμε τον εξομοιωτή που παρουσιάσαμε στην παράγραφο 5.4.

Στην εξομοίωση δημιουργούμε ένα τυχαίο δίκτυο από κόμβους οι οποίοι προσφέρουν κάποια αντικείμενα. Αρχικά δεν υπάρχει *κανένα αντίγραφο* αντικειμένου. Μετά τον σχηματισμό του δικτύου αρχίζουν τυχαίες αναζητήσεις (pulls για τυχαία αντικείμενα από τυχαίους κόμβους). Οι αναζητήσεις για κάθε αντικείμενο γίνονται με διαφορετικές πιθανότητες ώστε να μελετήσουμε τον αριθμό των αντιγράφων r_i συναρτήσει του ρυθμού ερωτήσεων q_i. Μετά από μια επιτυχημένη αναζήτηση δημιουργούνται αντίγραφα στο δίκτυο σύμφωνα με τις εξής πολιτικές:

- Ομοιόμορφη, όπου δημιουργείται σταθερός αριθμός αντιγράφων r_i για κάθε αντικείμενο i, ανεξάρτητα από τον ρυθμό ερωτήσεων.
- Ιδιοκτήτη, όπου μόνο ο κόμβος που ξεκινά την αναζήτηση δημιουργεί αντίγραφο
- Μονοπατιού, όπου αντίγραφα δημιουργούνται κατά μήκος του μονοπατιού μεταξύ του αναζητητή και του παροχέα.
- Pull then push, είναι η μέθοδος που προτείναμε (6.5) και στην οποία αμέσως μετά από ένα επιτυχημένο pull κάνουμε push (με το ίδιο TTL) στο δίκτυο το αντικείμενο ώστε να δημιουργηθούν αντίγραφα. Όταν ένας κόμβος λάβει το μήνυμα push δημιουργεί ένα τοπικό αντίγραφο της υπηρεσίας με πιθανότητα $P_{replicate}$.

Στόχος μας είναι να εξετάσουμε το κατα πόσο η προτεινόμενη μέθοδος push/pull είναι ικανή να δημιουργήσει αντίγραφα που ακολουθούν την κατανομή τετραγωνικής ρίζας.

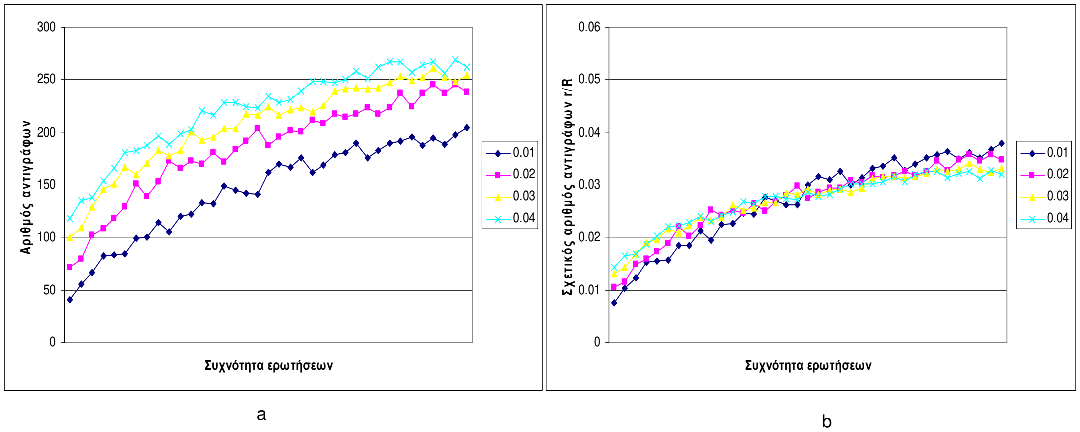
Στο σχήμα 6.1 βλέπουμε τον σχετικό αριθμό αντιγράφων $\frac{r_i}{R}$ για κάθε αντικείμενο ανάλογα με την συχνότητα ερωτήσεων q_i. Παρατηρούμε ότι σε όλες τις μεθόδους δημιουργίας αντιγρά-



Σχήμα 6.1: Η συχνότητα ερωτήσεων για ένα αντικείμενο σε σχέση με τον αριθμό αντιγράφων του. Φαίνεται και η βέλτιστη κατανομή τετραγωνικής ρίζας.

φων εκτός από την ομοιόμορφη, αντικείμενα που αναζητούνται σπάνια έχουν πολύ λιγότερα αντίγραφα σε σχέση με αυτά που αναζητούνται συχνά. Στο σχήμα φαίνεται και η βέλτιστη κατανομή των αντιγράφων δηλαδή η κατανομή τετραγωνικής ρίζας.

Η ομοιόμορφη δημιουργία αντιγράφων έχει την χειρότερη συμπεριφορά αφού ο αριθμός αντιγράφων που δημιουργούνται είναι ανεξάρτητος από την συχνότητα αναζητήσεων. Η μέθοδος «ιδιοκτήτη» δημιουργεί αντίγραφα που αυξάνονται γραμμικά ως προς τις αναζητήσεις. Ελαφρώς καλύτερη προσέγγιση είναι αυτή της μεθόδου «μονοπατιού» αφού πλησιάζει λίγο περισσότερο στην βέλτιστη κατανομή. Όπως βλέπουμε, η μέθοδος που προτείνουμε (pull then push) δημιουργεί αντίγραφα με κατανομή που είναι πολύ κοντά στην επιθυμητή βέλτιστη.



Σχήμα 6.2: a) Πως η πιθανότητα $P_{replicate}$ επηρεάζει τον αριθμό αντιγράφων που δημιουργούνται. b) Πως η πιθανότητα $P_{replicate}$ επηρεάζει την κατανομή τετραγωνικής ρίζας

Στο σχήμα 6.2a βλέπουμε ότι η πιθανότητα $P_{replicate}$ επηρεάζει τον αριθμό των αντιγράφων που δημιουργούνται στο σύστημα. Αφού σε κάθε αναζήτηση δημιουργούνται $P_{replicate}A_i = P_{replicate}\frac{n}{r_i}$ αντίγραφα, όσο μεγαλύτερη είναι η πιθανότητα αυτή, τόσα περισσότερα αντίγραφα θα δημιουργηθούν. Παρόλα αυτά, όπως βλέπουμε στο 6.2b, η κατανομή τετραγωνικής ρίζας δεν αλλάζει. Αυτό που αλλάζει είναι ο αριθμός των αντιγράφων με την επιτυχία επιτυγχάνεται.

6.5 Εφαρμογή αλγορίθμων push/pull για την ενημέρωση των αντιγράφων

Στην παράγραφο αυτή θα ασχοληθούμε με ένα άλλο πρόβλημα που αφορά την ενημέρωση των αντιγράφων. Υποθέτουμε ότι υπάρχει ένας αριθμός από αντίγραφα που έχουν δημιουργηθεί με τον τρόπο που περιγράψαμε στην παράγραφο και θέλουμε τα αντίγραφα αυτά να είναι συνεπή μεταξύ τους.

Ορίζουμε ως *ιδιοκτήτη* ενός αντικειμένου τον κόμβο που το πρόσφερε πρώτη φορά στο δίκτυο. Η βασική αρχή είναι ότι κάθε φορά που ο ιδιοκτήτης ενός αντικειμένου προβεί σε αλλαγές, όλα τα αντίγραφα του αντικειμένου στο δίκτυο θα πρέπει να ενημερωθούν. Για παράδειγμα, αν το αντικείμενο είναι ένα πρόγραμμα και ο κόμβος που το διαμοίρασε για πρώτη φορά δημοσίευσε μια νέα έκδοση του, θα πρέπει με κάποιο τρόπο να προσπαθήσει να ενημερώσει όλους τους κόμβους που το «κατέβασαν» να πάψουν να χρησιμοποιούν την παλιά έκδοση και να κατεβάσουν την καινούργια.

Στο κεφάλαιο 5 περιγράψαμε πως ο αλγόριθμος push μπορεί να χρησιμοποιηθεί για να κοινοποιήσει την μετακίνηση μιας υπηρεσίας στο δίκτυο. Στην περίπτωση των πολλαπλών αντιγράφων μετακίνηση μπορεί να θεωρηθεί η ενημέρωση ενός αντικειμένου από τον «ιδιοκτήτη» του. Έτσι, όταν προβεί σε κάποια αλλαγή, θα πρέπει να ενημερώσει το δίκτυο εκτελώντας τον αλγόριθμο push. Οι κόμβοι που θα λάβουν το μήνυμα push έχουν δύο επιλογές: ή να *ακυρώσουν* το αντίγραφό τους (*invalidation messages*), ή να ενημερώσουν την έκδοση που έχουν, επικοινωνώντας με τον ιδιοκτήτη.

Όμως, ο αλγόριθμος pull δεν είναι πλέον και τόσο απλός. Στην περίπτωση του δικτύου των caches ένας πράκτορας ήταν σε θέση να γνωρίζει πότε η πληροφορία που έχει στην cache δεν είναι έγκυρη: προσπαθούσε να χρησιμοποιήσει κάποια υπηρεσία και αποτύγχανε γιατί είχε μετακινηθεί. Στην περίπτωση των πολλαπλών αντιγράφων ο κάθε κόμβος χρησιμοποιεί το τοπικό αντίγραφο της υπηρεσίας και δεν είναι σε θέση να καταλάβει ότι η έκδοση που έχει είναι παλιά αφού δεν μπορεί να γνωρίζει αν ο ιδιοκτήτης της παρέχει κάποια ενημερωμένη έκδοση.

Έτσι, οι κόμβοι που έχουν κάποιο αντίγραφο πρέπει να ελέγχουν περιοδικά αν υπάρχει νέα έκδοση (periodic pulls). Η συχνότητα αυτής της αναζήτησης μπορεί να βασίζεται σε παλιότερα στατιστικά [18]. Έστω $V_{\alpha\nu\tau}$ είναι ο αριθμός έκδοσης κάποιου αντιγράφου ενός αντικειμένου, $V_{\iota\delta}$ ο αριθμός έκδοσης του ιδιοκτήτη. Προφανώς σε κάθε κόμβο του δικτύου $V_{\alpha\nu\tau} \leq V_{\iota\delta}$. Τέλος, έστω TTP είναι ο χρόνος για το επόμενο περιοδικό pull.

Υποθέτουμε ότι αν σε δύο διαδοχικά pull δεν βρεθεί κάποια νέα έκδοση της πληροφορίας ($V_{\iota\delta} = V_{\alpha\nu\tau}$) τότε μάλλον θα πρέπει να χρησιμοποιούμε μεγαλύτερο TTP. Άρα ο επόμενος χρόνος αναζήτησης θα αυξηθεί κατά μια σταθερά C:

$$TTP' = TTP + C$$

Αν μεταξύ δυο διαδοχικών pull δημοσιευθεί μια νεότερη έκδοση (δηλαδή $V_{\iota\delta} > V_{\alpha\nu\tau}$) τότε θα πρέπει να μειώσουμε το TTP και μάλιστα η μείωση αυτή να είναι μεγάλη αν το αντικείμενο άλλαξε παραπάνω από μια φορά:

$$TTP' = \omega \times TTP - (1 - \omega) \times \frac{TTP}{V_{\iota\delta} - V_{\alpha\nu\tau} + \alpha}$$

όπου $\alpha > 0$ και $0 < \omega < 1$ είναι παράμετροι που ορίζουν το πόσο γρήγορα η αργά θα γίνει αυτή η μείωση.

6.5.1 Ενημέρωση αντιγράφων που δημιουργήθηκαν με push/pull

Όπως είδαμε, για να επιτύχουμε την κατανομή τετραγωνικής ρίζας πρέπει μετά από μια επιτυχημένη αναζήτηση ο κάθε κόμβος να κάνει push ώστε να δημιουργηθούν αντίγραφά στην ευρύτερη γειτονιά του. Μπορούμε να επωφεληθούμε από αυτόν τον τρόπο κατασκευής αντιγράφων ώστε να επιτύχουμε πιο αποτελεσματική ενημέρωση.

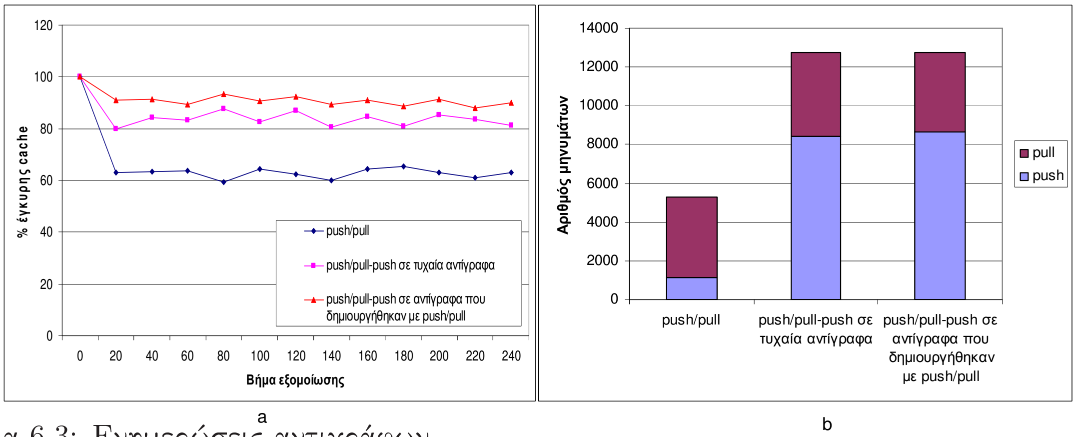
Ας υποθέσουμε ότι κάποιος κόμβος που είχε αναζητήσει παλιότερα ένα αντικείμενο λάβει μια ενημερωμένη έκδοση από κάποιο on-demand pull. Σε αυτή την περίπτωση θα πρέπει να προωθήσει (push) και την ενημέρωση αυτή γειτονιά του. Εκμεταλλευόμαστε δηλαδή τον μηχανισμό κατασκευής των αντιγράφων ώστε να προωθήσουμε την ενημέρωση σε ένα υποσύνολο κόμβων που υπάρχουν πολλαπλά αντίγραφα τα οποία χρειάζονται ενημέρωση. Ονομάζουμε την μέθοδο αυτή *push/pull-push* αφού οι ιδιοκτήτες που κάνουν αλλαγές κάνουν push (όπως και στο απλό push/pull), ενώ οι κόμβοι που κάνουν on-demand pull κάνουμε αμέσως μετά και οι ίδιοι push.

Οι βέλτιστες τιμές των παραμέτρων (decay, TTL) για το push της ενημέρωσης είναι να χρησιμοποιήσουμε τις ίδιες με αυτές του push δημιουργίας αντιγράφων. Στην περίπτωση που δεν επιθυμούμε να κρατήσουμε αυτή την πληροφορία, μπορούμε να χρησιμοποιήσουμε το βάθος στο οποίο βρέθηκε η ενημέρωση υποθέτοντας ότι έχουν ενημερωθεί οι κόμβοι που είναι πιο «μακρυά».

6.6 Πειραματικά αποτελέσματα

Θα εξετάσουμε τώρα αν οι αλγόριθμοι push/pull είναι ικανοί να κρατήσουν τα αντίγραφα ενημερωμένα. Αφού φτάσουμε το σύστημα σε σταθερή κατάσταση όσον αφορά τον αριθμό των αντιγράφων καταλήγουμε σε ένα δίκτυο όπου υπάρχουν αντίγραφα αντικειμένων που ακολουθούν την τετραγωνική κατανομή. Αρχικά, όλα τα αντικείμενα και τα αντίγραφα έχουν αριθμό έκδοσης $V_i = 1$. Έπειτα επιλέγουμε τυχαία αντικείμενα και αυξάνουμε τον αριθμό έκδοσης του αντιγράφου που υπάρχει στον «ιδοκτήτη» του αντικειμένου. Στόχος μας είναι να εξετάσουμε αν οι πολιτικές που περιγράψαμε παραπάνω είναι ικανές να ενημερώσουν τα υπόλοιπα ανίγραφα

Όπως παρατηρούμε στο σχήμα 6.3a, ακόμα και ένα απλό push/pull μπορεί να κρατήσει τα αντίγραφα συνεπή. Μάλιστα, λόγω του ότι τα αντίγραφα δημιουργούνται και σβήνονται



Σχήμα 6.3: Ενημερώσεις αντιγράφων

Ενημέρωση χρησιμοποιώντας το απλό push/pull, καθώς και την βελτίωση όπου μετά κάποιο pull γίνεται push σε δίκτυο όπου τα αντίγραφα έχουν προκύψει τυχαία και σε δίκτυο που έχουν προκύψει με την μέθοδο push/pull

συχνά, στο συγκεκριμένο παράδειγμα χρησιμοποιήθηκε ένα πολύ περιορισμένο push ($d = 0.9, t = 4$) που όμως είναι ικανό να διατηρήσει ένα μεγάλο ποσοστό των αντιγράφων σε συνεπή κατάσταση.

Όπως είναι φυσικό, η βελτίωση του να χρησιμοποιεί κάποιος push αμέσως μετά κάποιο επιτυχημένο pull βελτιώνει το αποτέλεσμα τόσο στο δίκτυο με τα τυχαία αντίγραφα όσο και στο δίκτυο που τα αντίγραφα φτιάχτηκαν με push/pull. Όμως, όταν τα αντίγραφα δεν έχουν φτιαχτεί με τυχαίο τρόπο αλλά σύμφωνα με την μέθοδο push-pull, τα αποτελέσματα είναι πολύ καλύτερα. Οπότε η υπόθεση που κάναμε ότι ένα push μετά από ένα επιτυχημένο on-demand pull εκμεταλλεύεται τον τρόπο δημιουργίας των αντιγράφων, επιβεβαιώνεται και πειραματικά. Και όπως βλέπουμε στην εικόνα 6.3b αυτό γίνεται χωρίς επιπλέον κόστος μηνυμάτων σε σχέση με το τυχαίο δίκτυο.

Κεφάλαιο 7

Συμπεράσματα

Στην εργασία αυτή μελετήσαμε το πρόβλημα των ενημερώσεων της cache σε ένα peer to peer δίκτυο από κινητούς πράκτορες. Στο μοντέλο αυτό, κάθε πράκτορας διατηρεί μια προσωπική cache περιορισμένου μεγέθους, που περιέχει πληροφορίες για τον εντοπισμό Κ διαφορετικών υπηρεσιών. Για κάθε μία από τις Κ υπηρεσίες, κρατά πληροφορίες για το πού θα βρει τον πράκτορα που την παρέχει. Αυτό έχει σαν αποτέλεσμα να διατηρείται ένας κατανεμημένος κατάλογος που περιέχει πληροφορίες που αντιστοιχίζουν υπηρεσίες με τους πράκτορες που τα προσφέρουν.

Το πρόβλημα που εξετάσαμε είναι το τι συμβαίνει στο παραπάνω δίκτυο όταν υπάρχει κινητικότητα. Όπως είδαμε, τα πρόβλημα που δημιουργούνται από την μετακίνηση μιας υπηρεσίας είναι αρκετά σημαντικά αφού μια τέτοια μετακίνηση μπορεί να προκαλέσει *ασυνέπεια στην cache των υπολοίπων πρακτόρων (cache inconsistency)*. Έτσι προτείναμε μια σειρά από αλγορίθμους ενημέρωσης των εγγραφών cache. Ουσιαστικά, με το να ενημερώνουμε τις εγγραφές cache των πρακτόρων, ενημερώνουμε τις συνδέσεις τους και κατα συνέπεια ενημερώνουμε την ίδια την τοπολογία του δικτύου.

Οι αλγόριθμοι ενημέρωσης που προτείναμε, συνδυάζουν την τεχνική *pull*, η οποία ξεκίνα από τους πράκτορες που χρειάζονται την ενημέρωση, και την τεχνική *push*, που ξεκινά από τους πράκτορες που μετακινούνται. Και η τεχνική push αλλά και η τεχνική pull βασίζονται σε κατάλληλες παραλλαγές της μεθόδου πλημμύρας (flooding). Μελετήσαμε την χρήση περιοδικών pull όπου οι πράκτορες περιοδικά ενημερώνουν ολόκληρη την cache τους και την καθ'απαίτηση (on-demand) χρήση pull όπου οι πράκτορες ενημερώνουν κάποια εγγραφή στην cache μόνο όταν ανακαλύψουν ότι δεν είναι έγκυρη. Κατα το push, όταν ο πράκτορας μετακινηθεί ενημερώνει τους υπολοίπους για την νέα θέση του. Μελετήσαμε αρκετές παραλλαγές αλγορίθμων flooding-push με στόχο να ενημερώσουμε όσο περισσότερους πράκτορες γίνεται χρησιμοποιώντας όσο το δυνατό λιγότερα μηνύματα.

Προτείνουμε επίσης μία νέα παραλλαγή flooding στην οποία οι πράκτορες που λαμβάνουν πληροφορία σχετικά με την μετακίνηση άλλων, την κρατούν για λίγο σε κάποιον κατάλογο που ονομάζουμε *κατάλογο «snooping»*. Συγκρίνουμε αυτές τις μεθόδους (που βασίζονται στο flooding) με την χρήση ενός «ενημερωμένου» push όπου οι πράκτορες κρατούν επιπρόσθετα και μία ανεστραμμένη cache (inverted cache). Με στόχο να μειώσουμε το μέγεθος του καταλόγου inverted cache συνδυάσαμε τον αλγόριθμο αυτό με την τεχνική «leasing» όπου κάθε πράκτορας μισθώνει τον χρόνο του στην inverted cache κάποιου άλλου.

Για να συγκρίνουμε τις μεθόδους αυτούς υλοποιήσαμε έναν εξομοιωτή. Κύριος στόχος μας είναι να εξετάσουμε την ποιότητα του δικτύου που προκύπτει με την χρήση του κάθε αλγορίθμου ενημέρωσης καθώς και τον φόρτο μηνυμάτων που προκαλεί κάθε αλγόριθμος.

Όταν δεν θέλουμε να χρησιμοποιήσουμε παραπάνω μνήμη για τους αλγορίθμους ενημέρωσης, τότε η μόνη λύση είναι ο απλός push/pull αλγόριθμος. Ο αλγόριθμος αυτός παρουσιάζει ικανοποιητικά αποτελέσματα (όσον αφορά την συνέπεια των cache) μόνο όταν χρησιμοποιηθεί ένα ευρύ push flooding αλλά επιφέρει μεγάλο φόρτο μηνυμάτων.

Η ύπαρξη καταλόγων snooping και περιοδικών pull βελτιώνει (κατά πολύ) τον αλγόριθμο plain push/pull αλλά ως αντάλλαγμα έχει αυξημένο κόστος σε μνήμη. Χρησιμοποιώντας τις ίδιες ακριβώς παραμέτρους με το απλό push/pull, η συνέπεια των εγγραφών στις cache των πράκτορες του του δικτύου είναι αρκετά καλύτερη και άρα μας επιτρέπει να επιτύχουμε *την ίδια ποιότητα στην cache χρησιμοποιώντας λιγότερα μηνύματα.*

Ο αλγόριθμος inverted cache push/pull επιφέρει ελάχιστο φόρτο μηνυμάτων και στιγμιαία ενημέρωση (1-hop) και άρα είναι κατάλληλος για συστήματα στα οποία γίνονται πολύ συχνές μετακινήσεις. Ωστόσο απαιτεί την χρήση μνήμης για την διατήρηση των καταλόγων inverted cache οι οποίοι μπορεί να γίνουν αρκετά μεγάλοι για πράκτορες που κατέχουν δημοφιλείς υπηρεσίες. Επιπλέον, οι αλλαγές στην cache των πρακτόρων (cache replacements) επιφέρουν φόρτο μηνυμάτων. Τέλος, ο αλγόριθμος αυτός δεν είναι κατάλληλος για αναξιόπιστα συστήματα open-MAS αφού ο κάθε πράκτορας βασίζεται σε άλλους στο να τον ενημερώσουν.

Τα αποτελέσματα αυτής της δουλειάς παρουσιάστηκαν πρόσφατα και στο [23].

Ένα δεύτερο θέμα το οποίο μελετήσαμε είναι η δυνατότητα εφαρμογής των αλγορίθμων push/pull στην δημιουργία και την διατήρηση πολλαπλών αντιγράφων σε ένα δίκτυο P2P. Όπως είδαμε ο βέλτιστος αριθμός αντιγράφων ενός αντικειμένου είναι ανάλογος της τετραγωνικής ρίζας αριθμού αναζητήσεων που γίνονται για αυτό. Μια καλή προσέγγιση για την επίτευξη αυτής της κατανομής είναι σε κάθε αναζήτηση να δημιουργούμε αντίγραφα που είναι ανάλογα με τον αριθμό των κόμβων που ερωτήθηκαν. Σε παλιότερες εργασίες αυτό είχε προσεγγιστεί με τις τεχνικές «owner replication», όπου μόνο ο κόμβος που έκανε την αναζήτηση δημιουργεί αντίγραφο, και «path replication» όπου δημιουργούνται αντίγραφα κατά μήκος του μονοπατιού του αναζητητή και του παροχέα.

Στην δική μας εργασία προτείναμε ότι η κατανομή τετραγωνικής ρίζας μπορεί να προσεγγιστεί αν αμέσως μετά απο μια επιτυχημένη αναζήτηση ο κόμβος που την ξεκίνησε κάνει push το αποτέλεσμα στο δίκτυο με TTL ίσο με το βάθος εύρεσης. Τα αντίγραφα θα δημιουργηθούν στους κόμβους που θα λάβουν το μήνυμα push. Όπως είδαμε και πειραματικά, η μέθοδος αυτή προσεγγίζει καλύτερα από τις προηγούμενες την βέλτιστη κατανομή τετραγωνικής ρίζας.

Επιπρόσθετα, δείξαμε πως μπορούν να εφαρμοστούν οι αλγόριθμοι push/pull στην διατήρηση της συνέπειας των αντιγράφων. Όταν ο ιδιοκτήτης ενός αντικειμένου προβεί σε αλλαγές κάνει push στο δίκτυο, ενώ οι κόμβοι που έχουν αντίγραφα εξετάζουν περιοδικά αν υπάρχει νέα έκδοση κάνοντας ένα on-demand pull. Παρουσιάσαμε επίσης έναν απλό τρόπο για να προσαρμόζεται χρόνος μεταξύ δυο περιοδικών pull.

Λόγω του ότι τα αντίγραφα δημιουργούνται με την χρήση ενός push μετά από κάποια αναζήτηση, προτείναμε ότι αν ο κόμβος που τα είχε δημιουργήσει λάβει κάποια νεότερη έκδοση (μέσω ενός on-demand pull για παράδειγμα) θα πρέπει να προωθήσει και την ενημέρωση αυτή με ένα push. Έτσι, μετά από κάθε επιτυχημένο on-demand pull θα πρέπει να γίνεται push με στόχο να επιτύχουμε την ενημέρωση όσο το δυνατό περισσοτέρων αντιγράφων. Τα πειραματικά αποτελέσματα έδειξαν ότι αν τα αντίγραφα έχουν φτιαχτεί με push η τεχνική αυτή είναι αποτελεσματική.

7.1 Μελλοντική δουλειά

Υπάρχουν πολλά ερευνητικά θέματα όσον αφορά τις ενημερώσεις σε ένα δυναμικό περιβάλλον όπως είναι τα δίκτυα P2P. Επιπλέον, οι αλγόριθμοι push/pull μπορούν να βρουν εφαρμογή σε πάρα πολλά πεδία. Εξαρτάται από το τι ορίζεται κάθε φορά ως «μετακίνηση» και τι ως «ενημέρωση». Αν θεωρήσουμε ότι μετακίνηση είναι οποιαδήποτε *νέα* πληροφορία η οποία πρέπει να γίνει γνωστή σε άλλους, τότε οι αλγόριθμοι αυτοί μπορούν να εφαρμοστούν και σε αρκετές άλλες περιπτώσεις.

Όπως είδαμε οι αλγόριθμοι αυτοί εξαρτώνται κατά πολύ από τις παραμέτρους που χρησιμοποιούνται κατά το push και το pull (decay, TTL, συχνότητα περιοδικών pull, expiration time, lease time κτλ). Αν και μελετήσαμε τις επιπτώσεις της κάθε παραμέτρου στο σύστημα θα πρέπει να μελετηθεί πως οι παράμετροι αυτοί θα πρέπει να αυτορρυθμίζονται σε ένα δυναμικό P2Pσύστημα ανάλογα με το μέγεθος του δικτύου και την πυκνότητα των συνδέσεων κ.α. Στόχος θα είναι να μπορούν από μόνοι τους οι κόμβοι να ρυθμίζουν τις παραμέτρους αυτές ώστε να βελτιστοποιηθεί τόσο η ποιότητα της ενημέρωσης όσο και ο φόρτος των μηνυμάτων που ανταλλάσσονται.

Επιπλέον, ενδιαφέρουσα ερευνητική κατεύθυνση είναι να μελετηθεί η εφαρμογή των μεθόδων push/pull σε ασύρματα ad-hoc δίκτυα. Σε αυτή την περίπτωση μπορούμε να εκμεταλλευτούμε πληροφορίες όπως η γεωγραφική θέση και «φυσική» μετακίνηση (με κάποια ταχύτητα και διεύθυνση) για να επιτύχουμε αποτελεσματικότερη ενημέρωση. Μπορούμε να εφαρμόσουμε τις μεθόδους αυτές για το πολύ σημαντικό θέμα της δρομολόγησης. Συγκεκριμένα μπορούμε να χρησιμοποιήσουμε τους αλγορίθμους push/pull για να ενημερώσουμε τους πίνακες δρομολόγησης που διατηρεί κάθε κόμβος σε ένα τέτοιο δίκτυο.

Τέλος θα μπορούσαν να ερευνηθούν νέοι τρόποι ενημέρωσης με στόχο την ελαχιστοποίηση του όγκου των μηνυμάτων. Για παράδειγμα η εύρεση καλύτερης συνάρτησης decay, νέων αλγορίθμων flooding, ελαφρώς δομημένες ενημερώσεις (πχ super nodes που έχουν ως στόχο να συγκεντρώνουν ενημερώσεις) κτλ.

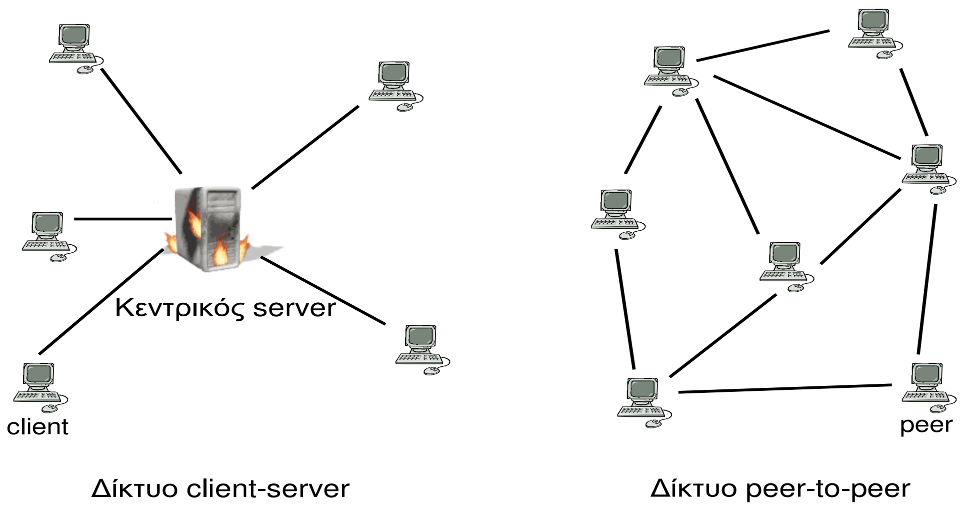
Παράρτημα A

Επισκόπηση δικτύων Peer-To-Peer και Ad-Hoc

Τα δίκτυα P2Pκαι ad-hoc έγιναν δημοφιλές αντικείμενο τόσο ερευνητικά όσο και ως εφαρμογές χρηστών. Ως αποτέλεσμα, είχαμε μια έκρηξη προτάσεων, μελετών, συστημάτων και εφαρμογών. Κρίθηκε λοιπόν απαραίτητο να γίνει μια πληρέστερη επισκόπηση των δικτύων αυτών. Για λόγους πληρότητας και ανεξαρτησίας του παραρτήματος αυτού, κάποιες πληροφορίες που παρουσιάσαμε στο κεφάλαιο 2 θα επαναληφθούν.

A.1 Συστήματα peer to peer

Η αρχιτεκτονική που επικρατούσε στο internet πριν την εμφάνιση των peer to peer ήταν βασισμένη στο μοντέλο πελάτη-εξυπηρέτη (client-server). Ο εξυπηρέτης ήταν συνήθως ένας ισχυρός υπολογιστής που παρείχε κάποιες υπηρεσίες και ήταν μονίμως διαθέσιμος (online), ενώ οι πελάτες ήταν οι υπολογιστές που χρησιμοποιούσαν τις υπηρεσίες του εξυπηρέτη και συνήθως οι clients δεν μπορούσαν να επικοινωνήσουν άμεσα μεταξύ τους (σχήμα A.1). Στα συστήματα client-server δεν υπήρχε αυτο-οργάνωση στο δίκτυο, η οργάνωση και η διοίκηση ολόκληρου του συστήματος προκύπτει από την πολιτική του εξυπηρέτη.



Σχήμα A.1: Από την αρχιτεκτονική πελάτη-εξυπηρέτη στα δίκτυα P2P

Σήμερα, με την εμφάνιση ολοένα γρηγορότερων και συνεχώς ενεργών συνδέσεων στο internet (always on) ένα νέο είδος διασύνδεσης έχει αναδειχθεί. Η σύνδεση πλέον γίνεται ατομικά μεταξύ των client. Αντί να καλούμε αυτή την επικοινωνία client σε client την καλούμε peer-to-peer (ή δίκτυα ομοτίμων) αφού οι συνδέσεις γίνονται μεταξύ ομότιμων (ίσων) peers, ενώ η λέξη client υποδηλώνει μια υπο-υπηρεσία που εξαρτάται από κάποιον εξυπηρέτη. Το γεγονός ότι κάποιος κεντρικός εξυπηρέτης μπορεί να έχει βοηθήσει στην αρχική σύνδεση και επικοινωνία δεν έχει καμία σχέση με το που γίνεται η τελική δραστηριότητα. Το αποτέλεσμα είναι ότι όλες οι δραστηριότητες γίνονται από τους peers, ή αλλιώς στα άκρα του δικτύου. Αν για παράδειγμα θελήσει κάποιος να δημοσιεύσει κάτι, μπορεί να το κάνει στο δικό του υπολογιστή, αντί να το στείλει σε κάποιον εξωτερικό εξυπηρέτη. Τα δίκτυα P2Pείναι μια νέα επικαλυπτική (overlay) διασύνδεση πάνω στις τρέχουσες συνδέσεις του internet.

Στην πιο απλή του μορφή, P2Pείναι ένας υπολογιστής που ρωτάει έναν άλλο για κάποια υπηρεσία. Η υπηρεσία αυτή μπορεί να είναι αρχεία, λογισμικό, δυνατότητα εκτύπωσης, υπολογιστική ισχύς κτλ. Γενικά, στα peer to peer δίκτυα *κάθε συμμετέχων* κόμβος δρα και ως πελάτης αλλά και ως εξυπηρέτης (servent) παρέχοντας κάποιους πόρους ή υπηρεσίες. Οι peers μπορούν να συνδεθούν ή να αποσυνδεθούν ανά πάσα στιγμή και συνήθως δεν υπάρχει κάποια καθολική γνώση για όλο το σύστημα (πχ πόσοι peers υπάρχουν ή τι υπηρεσίες προσφέρει ο καθένας). Παρόλο που σε αυτή την ιδέα βασίζονται όλα τα p2p συστήματα, η αρχιτεκτονική που συναντάται από σύστημα σε σύστημα μπορεί να διαφέρει πολύ. Ένας αφαιρετικός και διαισθητικός ορισμός για τα peer to peer συστήματα δίνεται από τον Clay Shirkey [2].

Peer-to-peer είναι η κλάση των εφαρμογών που εκμεταλλεύονται resources όπως: αποθηκευτικός χώρος, επεξεργαστική ισχύς, υλικό, ανθρώπινη παρουσία, που υπάρχει διάσπαρτο σε κάθε μέρος του Internet. Επειδή η πρόσβαση σε αυτά τα μη κεντρικοποιημένα resources σημαίνει την λειτουργία σε ένα ασταθές περιβάλλον σύνδεσης και απρόβλεπτων IP διευθύνσεων, οι κόμβοι peer-to-peer πρέπει να λειτουργούν εκτός του DNS και να έχουν σημαντική ή πλήρη αυτονομία από κεντρικούς εξυπηρέτες.

Πολλοί πιστεύουν ότι P2Pείναι απλά ένας διαφορετικός όρος για τα *κατανεμημένα συστήματα* (distributed computing). Η σημαντική διαφορά μεταξύ των κατανεμημένων συστημάτων και των σημερινών δικτύων P2Pείναι ότι οι κόμβοι ενός συστήματος P2Pείναι συνήθως απλά PC και όχι μεγάλοι, συνεχώς συνδεδεμένοι εξυπηρέτες. Οι πελάτες είναι πλέον απλοί κόμβοι που βοηθούν στην επίτευξη αξιοπιστίας και ανεξαρτησίας *αλλά με μειωμένο κόστος*. Οι πόροι έχουν ήδη αγοραστεί (από του χρήστες) και απλά «κάθονται» και περιμένουν, αχρησιμοποίητοι και ανεκμετάλλευτοι (πχ οι επεξεργαστές των υπολογιστών ενός σχολικού εργαστηρίου την νύχτα). Τα δίκτυα P2Pμας επιτρέπουν να εκμεταλλευτούμε αυτά τα αχρησιμοποίητα resources.

Συμπερασματικά, πολλοί πιστεύουν ότι οι μέρες που χρειαζόμασταν έναν κεντρικό εξυπηρέτη για να φιλοξενήσουμε πληροφορίες έχουν περάσει. Οι υπολογιστές των πελατών είναι πλέον peers σε ένα δίκτυο, επιτρέποντας σε καθένα να φιλοξενεί και να προσφέρει τις δικές του πληροφορίες.

A.1.1 Χρήσεις συστημάτων P2P

Η χρήση των δικτύων P2Pδεν περιορίζεται μόνο στην ανταλλαγή μουσικών αρχείων. Υπάρχουν πολλές χρήσεις όπως:

- **Κατανεμημένοι υπολογισμοί (distributed computing):** Τα συστήματα αυτά κατανέμουν μια υπολογιστικά δύσκολη εργασία σε έναν αριθμό από peers με στόχο να χρησιμοποιήσουν τους υπολογιστικούς πόρους όλων αυτών των κόμβων για την επίλυσή της. Συνήθως οι peers δεν επικοινωνούν άμεσα μεταξύ τους αλλά μέσω ενός κεντρικοποιημένου εξυπηρέτη που διανέμει τα κομμάτια της εργασίας. Παραδείγματα τέτοιων συστημάτων είναι τα SETI@Home [3],United Devices [4], Entropia, GPU κτλ.

- **Κατανεμημένα κατεβάσματα αρχείων:** Γενικά, όταν ένα αρχείο προσφέρεται από έναν μόνο χρήστη, όσο περισσότεροι το κατεβάζουν, τόσο μικρότερες ταχύτητες επιτυγχάνουν. Αν όμως το αρχείο αυτό προσφέρεται από ένα P2Pσύστημα τότε όσοι κατεβάζουν το αρχείο το προσφέρουν και ταυτόχρονα (γίνονται και οι ίδιοι εξυπηρέτες). Αυτό έχει σαν αποτέλεσμα να αυξάνεται η συνολική διαθεσιμότητα (bandwidth) του συγκεκριμένου αρχείου, να κατανέμεται ο φόρτος σε περισσότερα από ένα σημεία του internet. Παραδείγματα τέτοιων δικτύων είναι το BitTorrent, Gnutella [5], Napster [1], LimeWire, Kazaa [6], Freenet [7] , Popular Power, E-donkey κτλ.

- **Μηχανές αναζήτησης:** Με την χρήση P2Pδικτύων μπορεί κάποιος να ψάξει αρχεία, υπηρεσίες, σελιδοδείκτες (bookmarks) κτλ που παρέχουν άλλοι χρήστες. Για παράδειγμα μπορεί μέσα σε μια εταιρία να υπάρχει ένα τοπικό δίκτυο που επιτρέπει σε προγραμματιστές να αναζητούν παραδείγματα κώδικα σε υπολογιστές συναδέλφων τους.

- **Επικοινωνία:** Υπάρχουν εφαρμογές chat που δεν βασίζονται σε έναν μόνο κεντρικοποιημένο εξυπηρέτη. Έτσι οι χρήστες για παράδειγμα μπορούν να ορίζουν μόνοι τους τους κανόνες, η και να επικοινωνούν απ'ευθείας μεταξύ τους για μεγαλύτερη ασφάλεια/ταχύτητα. Παραδείγματα τέτοιων υπηρεσιών είναι το netmeeting, ICQ [8], voice over IP.

- **Συνεργασία:** Μπορεί κάποιος να ξεκινήσει ένα έργο (project) με έναν αριθμό από φίλους, να διαμοιράζονται τα αρχεία του έργου, έναν πίνακα συζητήσεων (discussion board), κάποιο chat και άλλα απαραίτητα εργαλεία. Αργότερα μπορεί οποιοδήποτε μέλος να καλέσει άλλους να συμβάλλουν στο έργο μοιράζοντας και εκείνοι κάποιες υπηρεσίες.

- **Ασφάλεια:** Μπορούν κάποια μέλη σε μία κοινότητα να ενημερώσουν ο ένας τον άλλο για πιθανές τρύπες ασφαλείας, εισβολές κτλ. Έτσι μπορούν όλοι να λάβουν τα μέτρα τους. Η αποκεντρωμένη χρήση της υπηρεσίας αυτή σημαίνει ότι η απώλεια ενός μόνο κόμβου δεν δημιουργεί πρόβλημα στην ικανότητα των άλλων να αντιδρούν σε θέματα ασφαλείας.

- **Ιδιωτικότητα (privacy):** Για παράδειγμα στο Freenet [7], μπορεί ο καθένας να κάνει δημοσιεύσει την γνώμη του (ή κάποιο αρχείο) χωρίς τον φόβο να αποκαλυφθεί. Από την στιγμή που την δημοσιεύσει, αυτή θα παραμένει για όσο υπάρχουν χρήστες που την κατεβάζουν, χωρίς να υπάρχει τρόπος να την σβήσει ή να την τροποποιήσει κάποιος.
- **Έξυπνες συσκευές:** Φανταστείτε συσκευές όπως ένα δίκτυο από ανιχνευτές ή ακόμα και ένα δίκτυο μεταξύ αυτοκινήτων να επικοινωνούσαν μεταξύ τους ασύρματα με στόχο να προειδοποιήσουν για παράδειγμα για κάποιο ατύχημα ή για κάποια παρατήρηση.

A.1.2 Βασικές αρχές

Τα P2P συστήματα μπορούν να θεωρηθούν σαν ένα Internet σε επίπεδο εφαρμογής πάνω στο κλασσικό Internet. Οι βασικές αρχές που συναντάμε είναι:

- **Οι peers είναι ομότιμοι:** σε αντίθεση με τα συστήματα πελάτη εξυπηρέτη όπου κάθε κόμβος έχει τις δικές του αρμοδιότητες στα P2P δίκτυα οι peers λειτουργούν και ως πελάτες και ως εξυπηρέτες.
- **Διαμοιρασμός πόρων/υπηρεσιών:** Όλα τα P2P συστήματα αφορούν τον διαμοιρασμό πόρων όπως αρχεία (πχ τραγούδια, video, προγράμματα), εύρος δικτύου (bandwidth), υπηρεσίες (όπως υπηρεσίες πληροφόρησης). Με το να μοιράζονται οι πόροι μπορούν να υλοποιηθούν εφαρμογές που είναι πολύ βαριές για να βασιστούν σε ένα μόνο κόμβο. Αυτό ήταν και το κίνητρο συστημάτων όπως το Napster.
- **Άμεση συνεργασία:** Οι peers συνεργάζονται μεταξύ τους για την επίτευξη κάποιας συγκεκριμένης εργασία. Οι κόμβοι επικοινωνούν άμεσα μεταξύ τους και όχι με την χρήση κάποιου ενδιάμεσου εξυπηρέτη. Στα P2Pσυστήματα που υπάρχουν εξυπηρέτες, χρησιμοποιούνται μόνο για την ανακάλυψη των peers που προσφέρουν κάποιον απαιτούμενο πόρο και έπειτα οι peers συνεργάζονται χωρίς την παρέμβαση του εξυπηρέτη.
- **Αυτο-οργάνωση και αυτονομία:** Όταν έχουμε ένα πλήρως κατανεμημένο σύστημα, δεν υπάρχει κάποιος κόμβος που να διατηρεί μια βάση δεδομένων που να κρατάει καθολική πληροφορία για το σύστημα. Έτσι οι κόμβοι πρέπει να αυτοδιοργανωθούν

χρησιμοποιώντας μόνο τοπική πληροφορία και αλληλεπιδρώντας με γειτονικούς κόμβους. Έτσι παίρνοντας τοπικές αποφάσεις μπορεί το σύστημα να δουλέψει συνολικά. Οι peers και οι συνδέσεις τους θεωρούνται προσωρινές. Έτσι δεν μπορούμε να κάνουμε εικασίες σχετικά με την τοπολογία.

- **Αποκέντρωση:** Η αποκέντρωση είναι ιδιαιτέρως χρήσιμη για την αποφυγή ενός σημείου αποτυχίας (single-point-of-failure) ή σημείων συμφόρησης (bottlenecks) της απόδοση ενός συστήματος με στόχο την κλιμάκωση (scalability). Όμως η κλιμάκωση αυτή βασίζεται κατά πολύ στο πρωτόκολλο του κάθε συστήματος. Πλήρη αποκεντρωμένα συστήματα είναι το Gnutella και το Freenet.

- **Λειτουργία σε ασταθές περιβάλλον:** Τα συστήματα P2Pβασίζονται σε κόμβους που συνδέονται και αποσυνδέονται συχνά από το δίκτυο. Επιπλέον επειδή οι προσωπικοί υπολογιστές που συμμετέχουν είναι πιο επηρεπείς σε τυχαίες αστοχίες (προβλήματα υλικού, προβλήματα συνδέσης στο internet, ιοί, χάκερς κτλ) οι αλγόριθμοι των δικτύων αυτών αντιμετωπίζουν το δυναμικό αυτό περιβάλλον σαν μια συνήθης διαδικασία και όχι σαν μια εξαίρεση.

- **Αναζήτηση:** Η αναζήτηση στα P2P συστήματα μοιάζει με στατικές αναζητήσεις όπως αυτές των Goggle ή Yahoo. Στην στατική αναζήτηση, η πληροφορία πριν αναζητηθεί πρέπει πρώτα να συλλεχθεί από κάποιον crawler (οι crawler συλλέγουν πληροφορίες από σελίδες και τις προσθέτουν σε βάσεις δεδομένων μηχανών αναζήτησης). Και μεταξύ δυο επισκέψεων ενός crawler σε κάποιο site μπορούν να περάσουν μήνες . Στο ενδιάμεσο η πληροφορία μπορεί να αλλάξει, να διαγραφεί κτλ με αποτέλεσμα τα αποτελέσματα της αναζήτησης να είναι αναξιόπιστα. Τα δίκτυα P2P δίνουν την δυνατότητα για αναζήτηση σε πραγματικό χρόνο. Όμως, ανάλογα με το πρωτόκολλο, κάποια πληροφορία μπορεί να μην βρεθεί ακόμα και αν υπάρχει στο δίκτυο. Επίσης δεν υπάρχει εγγύηση για την διάρκεια της αναζήτησης. Έτσι υπάρχει πρόβλημα ποιότητας υπηρεσιών (Quality Of Service ή QoS) στα συστήματα P2P .

A.1.3 Τύποι συστημάτων P2P

Κεντρικοποιημένα συστήματα (Centralized P2P)

Κάθε peer συνδέεται σε έναν εξυπηρέτη που συντονίζει τις επικοινωνίες μεταξύ των peers και όλες οι ανταλλαγές μηνυμάτων γίνονται μέσω του εξυπηρέτη (σχήμα A.2). Πολλοί υποστηρίζουν ότι τα συστήματα αυτά δεν θα πρέπει να θεωρούνται P2Ραφού οι peers δεν επικοινωνούν απ'ευθείας. Απλώς ο κεντρικός εξυπηρέτης καταγράφει τις υπηρεσίες που προσφέρουν οι peers και αν κάποιος θελήσει να χρησιμοποιήσει μία από αυτές, τις χρησιμοποιεί μέσω του εξυπηρέτη. Τα πλεονεκτήματα αυτής της μεθόδου είναι ότι η διεύθυνση του εξυπηρέτη είναι γνωστή και είναι εύκολο να συνδεθεί κάποιος σε αυτόν. Επιπλέον, επειδή τα πάντα περνούν από τον εξυπηρέτη, μπορεί να χρησιμοποιηθεί κάποιο caching για να αυξηθεί η απόδοση. Δεν υπάρχει περίπτωση να έχουμε κατάτμηση του δικτύου αφού υπάρχει μόνο ένα επίπεδο συνδέσεων. Τα μειονεκτήματα είναι ότι ο εξυπηρέτης είναι σημείο συμφόρησης και σημείο αποτυχίας (αν «πέσει» ο εξυπηρέτης δεν λειτουργεί όλο το δίκτυο. Μερικά παραδείγματα τέτοιων εφαρμογών είναι εφαρμογές που διαμοιράζουν επεξεργαστικούς πόρους (CPU sharing applications) όπως το SETI@Home [3].

Συστήματα μεσίτη (Broker P2P):

Οι peers συνδέονται σε κάποιο εξυπηρέτη για να ανακαλύψουν άλλους peers και υπηρεσίες και μόλις βρουν αυτό που θέλουν επικοινωνούν απευθείας μεταξύ τους για να συνεργαστούν (σχήμα A.3). Ο κάθε κόμβος ξέρει μόνο τον κεντρικό εξηπηρέτη και κάποιους peers που ο εξυπηρέτης του «σύστησε». Τα πλεονεκτήματα αυτών των δικτύων είναι ότι έχει την απόδοση ενός κεντρικοποιημένου εξυπηρέτη (όσον αφορά τις αναζητήσεις) αλλά ταυτόχρονα επιτρέπει τις απ'ευθείας συνδέσεις των peers που αποσυμφορίζουν τον κεντρικό εξυπηρέτη από τις εσωτερικές ανταλλαγές μηνυμάτων. Παραδείγματα τέτοιων δικτύων είναι το Napster [1].

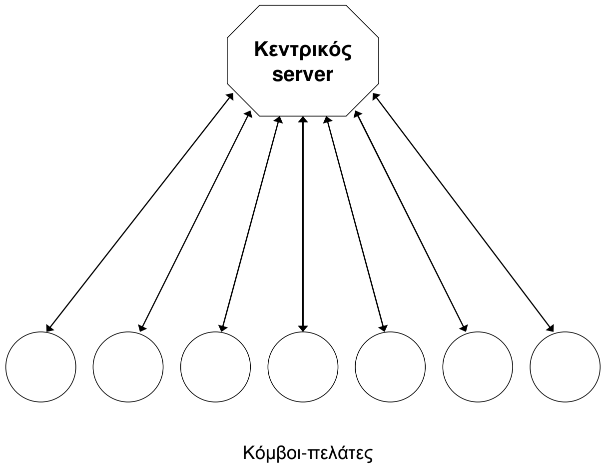
Αποκεντρωμένα συστήματα (Decentralized P2P):

Αυτή είναι η αληθινή αρχιτεκτονική peer-to-peer: δεν υπάρχει κανένας εξυπηρέτης, απλά peers (σχήμα A.4). Ένας νέος κόμβος συνδέεται σε έναν υπάρχον κόμβο του δικτύου και αυτός τον συστήνει σε μερικούς ακόμα. Δεν υπάρχει κανένας κεντρικοποιημένος κόμβος που να συντονίζει το δίκτυο. Το δίκτυο λειτουργεί συνολικά με την λήψη μόνο τοπικών αποφάσεων σε κάθε peer. Δυο ξεχωριστά δίκτυα μπορούν να ενωθούν με έναν απλό κόμβο που συνδέεται και στα δύο. Τα πλεονεκτήματα είναι ότι δεν υπάρχει κάποιο αδύνατο σημείο στο δίκτυο όσον αφορά την συμφόρηση και τις αστοχίες. Το μειονεκτήμα είναι ότι οι αναζητήσεις είναι αργές αφού πρέπει να μεταδοθούν μέσω πολλών βημάτων σε όλους τους κόμβους (multi-hop searching). Παραδείγματα τέτοιων δικτύων είναι η Gnutella [5] και το Freenet [7].

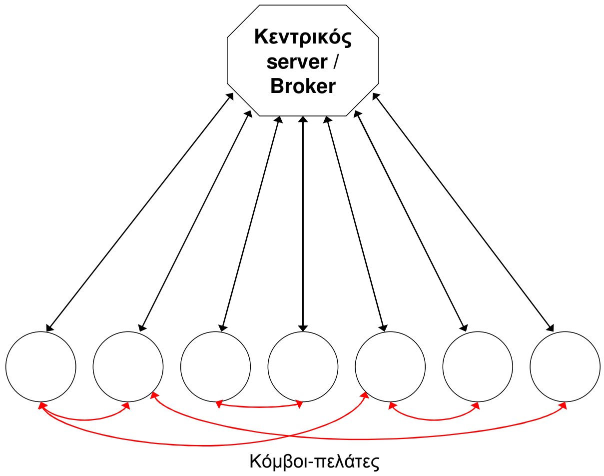
Μερικώς αποκεντρωμένα συστήματα (Partially decentralized P2P):
Τα δίκτυα αυτά είναι υβριδικά ανάμεσα στα συστήματα μεσίτη και στα αποκεντρωμένα. Στα συστήματα αυτά, όταν υπάρχουν κάποιοι κόμβοι που υπερτερούν σε σχέση με άλλους (καλύτερη σύνδεση στο internet, καλύτερο επεξεργαστή, μεγαλύτερη χωρητικότητα δίσκου, γνωρίζουν περισσότερους peers κ.α.) τότε αυτοί οι κόμβοι λειτουργούν ως super nodes. Οι super nodes ή super peers, λειτουργούν σαν μικροί εξυπηρέτες-μεσίτες μέσα σε ένα μεγαλύτερο αποκεντρωμένο δίκτυο. Κάθε κόμβος του δικτύου (συμπεραλαμβανομένων των super peers), συνδέεται σε έναν ή περισσότερους super peers και τους χρησιμοποιεί για εκτελέσει γρήγορες αναζητήσεις (σχήμα A.5). Οι κόμβοι αυτοί δεν αποτελούν σημείο αποτυχίας (point-of-failure) για το δίκτυο αφού επιλέγονται δυναμικά και σε περίπτωση αποτυχίας το δίκτυο θα τους αντικαταστήσει με άλλους. Παραδείγματα τέτοιων δικτύων είναι το Kazaa [6] και το Morpheus.

A.1.4 Δομή P2Pδικτύων

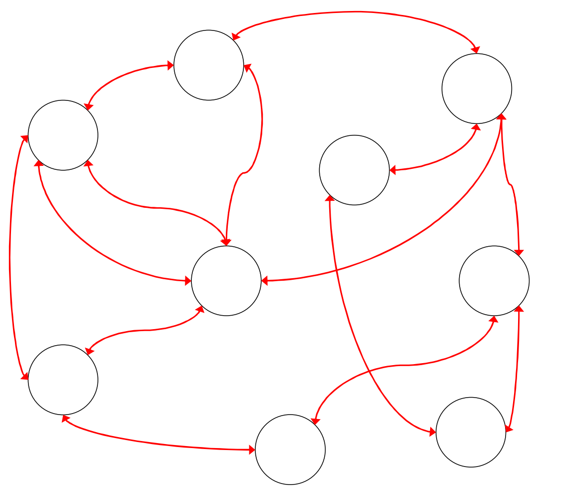
Τα συστήματα P2P αποτελούνται από δυναμικά δίκτυα με πολύπλοκη τοπολογία. Αυτή η τοπολογία δημιουργεί ένα δίκτυο επικάλυψης (overlay network) που μπορεί να μην έχει καμία σχέση με το φυσικό δίκτυο που συνδέει τους κόμβους. Τα P2P δίκτυα μπορεί να διαφοροποιηθούν ως προς το αν αυτό το overlay έχει κάποια δομή (structure) ή δημιουργείται τυχαία (ad-hoc). Με τον όρο *δομή (ή structure)*, εννοούμε τον τρόπο με τον οποίο κατανέμεται το περιεχόμενο σε σχέση με την τοπολογία: αν υπάρχει κάποιος τρόπος να πούμε απ'ευθείας ποιοι κόμβοι έχουν κάποια συγκεκριμένη πληροφορία ή αν πρέπει να αναζητήσουμε ολόκληρο το δίκτυο.



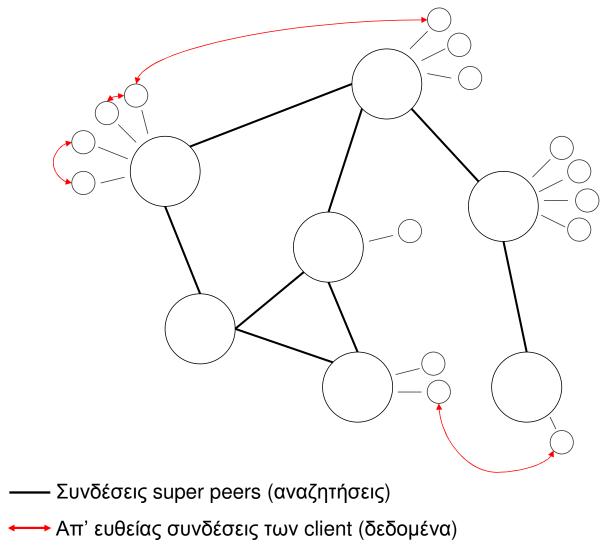
Σχήμα A.2: Κεντρικοποιημένα P2Pσυστήματα



Σχήμα A.3: Υβριδικά αποκεντρωμένα (brokered) P2P



Σχήμα Α.4: Πλήρως αποκεντρωμένα P2P



Σχήμα Α.5: Μερικώς αποκεντρωμένα P2P

Αδόμητα P2P

Στα *P2Pσυστήματα χωρίς δομή (unstructured P2P)* όπως το Gnutella [5], η τοποθέτηση της πληροφορίας δεν έχει καμία σχέση με την overlay τοπολογία. Αφού δεν υπάρχει καμία πληροφορία για το ποιοι κόμβοι προσφέρουν ποια resources, η αναζήτηση σημαίνει τυχαία αναζήτηση σε κάποιους κόμβους του δικτύου όπως θα δούμε και στο κεφάλαιο A.1.7. Το θετικό με αυτά τα δίκτυα είναι ότι μπορούν να εξυπηρετήσουν ένα μεγάλο και δυναμικό πληθυσμό από κόμβους. Όμως το κόστος της αναζήτησης είναι μεγάλο με αποτέλεσμα τα δίκτυα αυτά να θεωρούνται να μην θεωρούνται κλιμακούμενα (unscalable).

Δομημένα P2P

Με στόχο να επιτύχουμε μεγαλύτερη ικανότητα κλιμάκωσης (scalability), δημιουργήθηκαν τα *δομημένα συστήματα P2P (structured)*. Τέτοια συστήματα είναι το Chord, CAN, PAST, Tapestry κτλ. Στα δίκτυα αυτά η επικαλύπτουσα τοπολογία του δικτύου είναι ελεγχόμενη και οι πόροι τοποθετούνται σε συγκεκριμένες θέσεις. Αυτά τα συστήματα παρέχουν κάποια αντιστοίχηση (mapping) μεταξύ του πόρου και της θέσης του με την μορφή κάποιου πίνακα δρομολόγησης (routing table) έτσι ώστε η αναζήτηση να μπορεί να δρομολογηθεί στον κόμβο που έχει τον επιθυμητό πόρο. Τα δίκτυα αυτά προσφέρουν κλιμάκωση για αναζητήσεις κάποιων πόρων που είναι καλά ορισμένοι. Όμως τα δίκτυα αυτά είναι δύσκολο να διατηρήσουν την δομή τους σε ένα μεγάλο και δυναμικά παραγόμενο πληθυσμό από peers (κόμβοι συνδέονται και αποσυνδέονται συχνά).

Ελαφρώς δομημένα P2P

Μια τρίτη κατηγορία δικτύων είναι τα *ελαφρώς δομημένα (loosely structured)*. Τέτοιο παράδειγμα δικτύου είναι το Freenet. Τα δίκτυα αυτά ταξινομούνται κάπου ανάμεσα στα προηγούμενα δύο. Οι θέσεις των πόρων επηρεάζονται από κάποια δρομολόγηση αλλά δεν καθορίζονται από αυτή.

Έχουν αναπτυχθεί αρκετά συστήματα P2Pσε κάθε κατηγορία. Μπορείτε να δείτε που κατατάσσονται μερικά απότ τα πιο δημοφιλή συστήματα P2Pστον πίνακα A.1.

	Unstructured Networks	Loosely Structured Networks	Structured Networks
Hybrid Decentralized	Napster		
Pure Decentralized	Gnutella	Freenet	Chord, Can, Tapestry
Partially Centralized	Kazaa, Gnutella		

Πίνακας A.1: Κατηγορίες γνωστών P2P δικτύων

A.1.5 Παραδείγματα αδόμητων δικτύων P2P

Παρακάτω θα περιγράψουμε περιληπτικά την βασική αρχιτεκτονική και τον βασικό τρόπο λειτουργίας μερικών από τα πιο δημοφιλή συστήματα P2Pπου αναπτύχθηκαν. Θα αναφερθούμε πρώτα σε συστήματα αδόμητα όπως το Napster, Seti@Home, Kazaa, Morpheus, Gnutella, Freenet και έπειτα σε δομημένα συστήματα όπως τον Chord και Can.

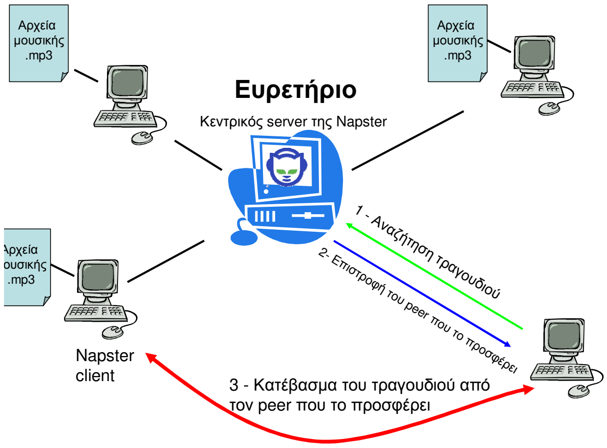
Napster

Το Napster [1] είναι το διάσημο mp3 file sharing σύστημα που δημιουργήθηκε το 1999. Ήταν επαναστατικό γιατί ήταν από τα πρώτα που επέτρεπε στους χρήστες αφού συνδεθούν σε έναν κεντρικοποιημένο εξυπηρέτη, να ψάξουν στα αρχεία των συνδεδεμένων χρηστών και μετά να τα κατεβάζουν κατευθείαν από αυτούς.

Ο χρήστες όταν συνδέονται στον κεντρικό εξυπηρέτη στέλνουν την λίστα με τα mp3 που κάνουν share. Ο κεντρικός εξυπηρέτης προσθέτει την λίστα αυτή σε μία βάση δεδομένων που περιέχει τα τραγούδια και ποιοι συνδεδεμένοι χρήστες τα προσφέρουν. Όταν κάποιος αναζητά ένα αρχείο, ο κεντρικός εξυπηρέτης ψάχνει στην βάση δεδομένων και του επιστρέφει τους peers που το έχουν. Έπειτα ο χρήστης συνδέεται απ'ευθείας στον peer που έχει το αρχείο και το κατεβάζει χωρίς καμία περαιτέρω παρέμβαση του κεντρικού εξυπηρέτη. Παράδειγμα του δικτύου Napster είναι αυτό του σχήματος A.6

Seti@Home

Το Seti@Home [3] ξεκίνησε το 1996 ως ένα επιστημονικό πείραμα που χρησιμοποιεί τους υπολογιστές που είναι συνδεδεμένοι στο internet για να ψάξει για εξωγήινη νοημοσύνη (extraterrestrial intelligence, SETI). Αποτελείται από έναν πελάτη υπό μορφή screen saver που αναλαμβάνει να αναλύσει δεδομένα από ραδιοτηλεσκόπια. Ο πελάτης συνδέεται σε

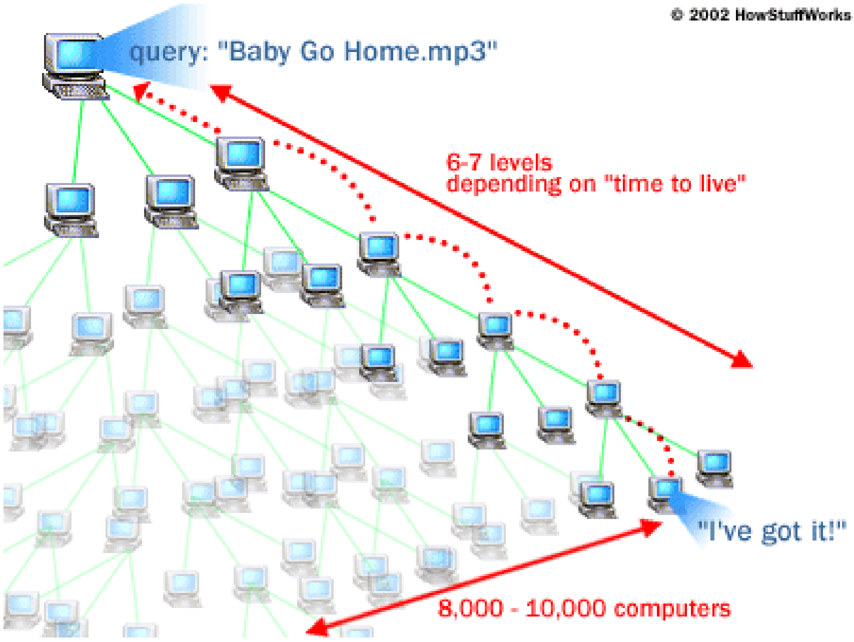


Σχήμα A.6: Το δίκτυο ανταλλαγής τραγουδιών Napster

έναν κεντρικό εξυπηρέτη, κατεβάζει τα δεδομένα που πρόκειται να επεξεργαστεί και μετά τα επεξεργάζεται μέχρι να λύσει το πρόβλημα και στέλνει τα αποτελέσματα πίσω στον εξυπηρέτη. Αν η εκτέλεση δεν επιτύχει τότε αγνοείται. Παρόμοια συστήματα έχουν χρησιμοποιηθεί αποκρυπτογράφηση πακέτων και από εταιρίες όπως η United Devices και η Entropia.

Kazaa, Morpheus

Το Kazaa [6] και το Morpheus [25] είναι δύο παρόμοια P2P συστήματα που ανήκουν στην κατηγορία partially centralized αφού χρησιμοποιούν την ιδέα των *SuperNodes*. Στους κόμβους αυτούς ανατίθεται δυναμικά να εξυπηρετούν ένα υποσύνολο των peers του δικτύου με το να κάνουν caching και indexing των αρχείων τους. Οι peers εκλέγονται αυτόματα για να γίνουν supernodes αν έχουν αρκετό εύρος δικτύου και επεξεργαστική ισχύς. Στο Morpheus ένας κεντρικός εξυπηρέτης κρατά λίστα με κάποιους από τους SuperNodes. Οι αναζητήσεις στέλνονται μόνο σε supernodes και όχι σε απλούς peers. Αν κάποιος supernode αποτύχει οι συνδεδεμένοι σε αυτόν peers μπορούν να συνδεθούν σε άλλο ή κάποιος από αυτούς να γίνει ο νέος supernode.



Σχήμα A.7: Διαδικασία αναζήτησης στο δίκτυο gnutella. Εικόνα από το site «how stuff works» [26]

Gnutella

Το Gnutella [5] είναι το πρώτο αληθινό ομότιμο P2P σύστημα. Δεν υπάρχει κάποιος κεντρικός εξυπηρέτης για αναζήτηση, οργάνωση και επικοινωνία. Οι χρήστες συνδέονται για πρώτη φορά στο δίκτυο με κάποιον peer χρησιμοποιώντας έναν εξυπηρέτη αναζήτησης (lookup server) που ονομάζεται GnuCache. Αλλά αυτό δεν είναι υποχρεωτικό: μπορούν να συνδεθούν βρίσκοντας την διεύθυνση κάποιου peer μέσω Newsgroups, websites κτλ. Αφού συνδεθεί με ένα peer είναι εύκολο να βρει και άλλους κόμβους για να συνδεθεί κάνοντας κάποια αναζήτηση μέσω του πρώτου . Οι κόμβοι στο Gnutella συνήθως συνδέονται με τρεις άλλους κόμβους και μετά εκτελούν αναζητήσεις χρησιμοποιώντας κάποιο broadcast σε όλους τους γείτονες που με την σειρά τους το προωθούν στους δικούς τους γείτονες κ.ο.κ. Η διαδικασία αυτή ονομάζεται *πλημμύρα ή flooding*. Αφού βρεθεί ο κόμβος που έχει το αρχείο η επικοινωνία γίνεται απευθείας μεταξύ των δύο κόμβων όπως και στα υπόλοιπα P2P δίκτυα (σχήμα A.7)

Freenet

Το freenet [7] είναι ένα αποκεντρωμένο P2P σύστημα. Υπάρχει μερική δομή αλλά οι peers οργανώνονται μεταξύ τους. Ουσιαστικά χρησιμοποιεί τον ελεύθερο χώρο στον δίσκο των peers για να δημιουργήσει ένα κοινό εικονικό file system. Εκεί είναι και η κεντρική διαφορά του με το Gnutella το οποίο είναι ένα file sharing σύστημα ενώ το freenet είναι ένα file-storage σύστημα. Ενώ στο Gnutella τα αρχεία αντιγράφονται όταν ένας χρήστης θελήσει να τα κατεβάσει, στο freenet τα αρχεία μπορούν να προωθηθούν σε άλλους κόμβους για αποθήκευση ή replication.

Το freenet έχει σχεδιαστεί ώστε να παρέχει *ασφάλεια και ανωνυμία* στους χρήστες του. Είναι ανέφικτο να ανακαλυφθεί η πραγματική πηγή του αρχείου (ο χρήστης που το δημιούργησε ή το έκανε share) γιατί τα δεδομένα απλά περνάνε από κόμβο σε κόμβο και είναι αδύνατο κάποιος να καθορίσει τα περιεχόμενα που υπάρχουν στον δικό του κόμβο (πόσο μάλλον να κρατηθεί υπεύθυνος για αυτά).

Όταν ένα νέο αρχείο τοποθετείται στο σύστημα, υπολογίζεται ένα key για αυτό και το αρχείο αυτό τοποθετείται σε peers που έχουν αρχεία με παρόμοιο key. Αν κάποιος αναζητήσει αυτό το αρχείο θα επιστραφεί στον αναζητητή μέσω ενός μονοπατιού και όχι κατευθείαν με αποτέλεσμα να είναι αδύνατος ο εντοπισμός του. Οποιοσδήποτε ενδιάμεσος κόμβος κάνει cache το αρχείο και μπορεί να αλλάξει τον αποστολέα (βάζοντας τον εαυτό του). Αυτή είναι άλλη μια δικλείδα για να διατηρηθεί η ανωνυμία αφού οι peers που έκαναν cache το δεδομένο δεν μπορούν να θεωρηθούν υπεύθυνοι μιας και αυτό έγινε χωρίς δικιά τους παρέμβαση.

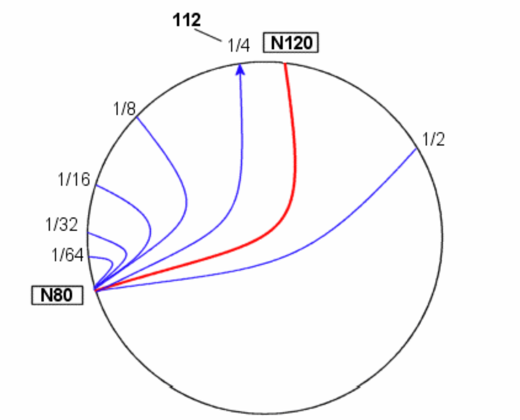
Κάθε κόμβος έχει ένα πίνακα δρομολόγησης με τους κόμβους που ξέρει και τα keys τους. Το Freenet είναι loosely structured γιατί ένας κόμβος μπορεί να υπολογίζει ποιοι peers είναι πιο πιθανό να έχουν το δεδομένο της αναζήτησης και να προωθήσει το μήνυμα μόνο σε αυτούς. Έτσι δημιουργείται ένα μονοπάτι που έχει μεγάλες πιθανότητες να οδηγήσει στο δεδομένο.

Περισσότερες πληροφορίες σχετικά με το πρωτόκολλο και την αρχιτεκτονική του freenet μπορείτε να βρείτε στην ιστιοσελίδα του [7].

A.1.6 Παραδείγματα δομημένων δικτύων P2P

Chord

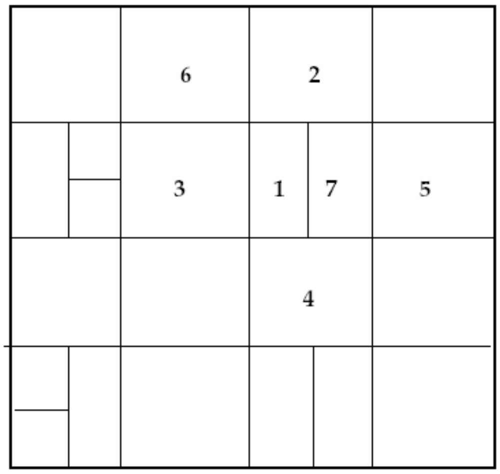
Το chord [27] είναι ένα πρωτόκολλο που επιτρέπει μια scalable γρήγορη αναζήτηση κόμβων που περιέχουν συγκεκριμένα δεδομένα. Στο chord ο κάθε κόμβος αναλαμβάνει να εξυπηρετήσει συγκεκριμένα δεδομένα και χρειάζονται O(log n) βήματα για να βρούμε οποιοδήποτε κόμβο από οποιοδήποτε άλλο κόμβο. Οι κόμβοι οργανώνονται σε έναν δακτύλιο και κάθε κόμβους ξέρει τον αμέσως προηγούμενο κόμβο, τον αμέσως επόμενο, τον 2 hops επόμενο, τον 4 hops επόμενο, τον 8 hops επόμενο κοκ. Λόγω της οργάνωσης ξέρει πολύ περισσότερους κοντινούς κόμβους παρά μακρινούς. Παράδειγμα της τοπολογίας chord μπορείτε να δείτε στο σχήμα A.8.



Σχήμα A.8: Το δίκτυο chord

Για την αναζήτηση ουσιαστικά πρωοθούμε την ερώτηση στον κατάλληλο κόμβο από αυτούς που ξέρουμε: στέλνουμε την ερώτηση στον τελευταίο κόμβο που είναι πριν από τον κόμβο που ψάχνουμε. Αυτός προωθεί στον αμέσως προηγούμενο κόμβο που ξέρει κ.ο.κ. μέχρι να βρούμε τον τελικό κόμβο.

Με την είσοδό του ένας κόμβος πρέπει να θέσει τους δείκτες στον προηγούμενο και τους επόμενους κόμβους σωστά. Αλλά αυτό δεν είναι αρκετό αφού όλοι οι κόμβοι στο δίκτυο πρέπει να αναδιοργανώσουν τα δεδομένα τους ώστε να κάνουν χώρο για δεδομένα που θα αναλάβει ο νέος κόμβος. Επειδή αυτό απαιτεί χρόνο $O(log^2 n)$ αυτό γίνεται περιοδικά. Παρόμοιες καταστάσεις αντιμετωπίζονται και κατά την αποχώρηση ενός κόμβου. Περισσότερες πληροφορίες μπορείτε να βρείτε στο [27].



Σχήμα A.9: Το δίκτυο CAN

CAN

Το CAN [28] είναι ουσιαστικά ένας μηχανισμός για scalable indexing. Κάθε δεδομένο (value) παίρνει ένα κλειδί (key). Η αποθήκευση αυτής της αντιστοιχίας (mapping) γίνεται κατανεμημένα σε όλους τους κόμβους του δικτύου. Έτσι, κάθε κόμβος φυλάει ένα μέρος (ζώνη) των ζευγαριών (Key,Value). Επιπλέον κρατά και πληροφορίες για τους κόμβους γειτονικών ζωνών. Στο σύστημα αυτό οι κόμβοι χωρίζονται σε έναν D-dimensional καρτεσιανό χώρο. Κάθε κόμβος αναλαμβάνει μία τέτοια ζώνη (μπορείτε να την φανταστείτε σαν ένα ορθογώνιο παραλληλόγραμμο) και σε κάθε ζώνη υπάρχει μόνο ένας κόμβος (σχήμα A.9).

Το hash table (Key, Value) κατανέμεται ως εξής: ο κόμβος περιέχει όλα τα δεδομένα για τα οποία το Key τους αφού πέρασε από μία uniform hash function H(key) τα έστειλε σε σημείο P που ανήκει μέσα στο τετράγωνο της ζώνης. Όταν ψάχνουμε για ένα δεδομένο με key K, τότε το περνάμε από την ίδια hash function η οποία μας κατευθύνει πάλι στο σημείο P που είναι μέσα στην ζώνη του κόμβου που έχει αποθηκευμένο το (Key, Value). Επειδή στο ενδιάμεσο μπορεί να έχει αλλάξει η ζώνη (να έχει προστεθεί νέος κόμβος), αν δεν βρεθεί η τιμή στην ζώνη αυτή, κοιτάμε στους γείτονες.

Όταν ένας νέος κόμβος συνδέεται επιλέγει ένα τυχαίο σημείο P στον καρτεσιανό χώρο. Μετά επικοινωνεί με τον κόμβο στον οποίο ανήκει η ζώνη και αυτός του δίνει τον μισό από τον χώρο της ζώνης μεταδίδοντας του το hash table που αντιστοιχεί στον χώρο αυτό και ενημερώνοντάς τον για τους γείτονες. Αντίστοιχα στην αποχώρηση παραδίδει την ζώνη του σε κάποιον από τους γείτονες (merge).

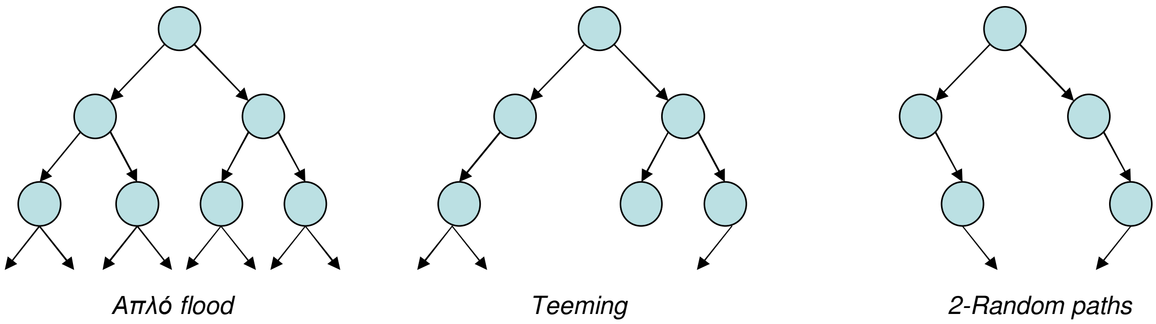
Αναλυτική περιγραφή του συστήματος και των αλγορίθμων του θα βρείτε στο [28].

A.1.7 Αναζητήσεις σε αδόμητα δίκτυα P2P

Στα unstructured P2P δίκτυα, όταν κάποιος θέλει να αναζητήσει μία πληροφορία ή ένα αρχείο, πρέπει αναγκαστικά να ξεκινήσει μια διαδικασία αναζήτησης σε όλο το δίκτυο ώστε να βρει κάποιον peer που θα το έχει.

Η γενική μέθοδος αναζήτησης σε αυτά τα δίκτυα είναι η πλημμύρα (flooding): Ο peer που θέλει να κάνει την αναζήτηση πλημμυρίζει το δίκτυο με το αίτημα του μέχρι να φτάσει σε κάποιον που να έχει την αναζητούμενη πληροφορία. Πιο συγκεκριμένα, αν η πληροφορία δεν βρεθεί σε κάποια τοπική cache, ο peer στέλνει ένα μήνυμα αναζήτησης σε ένα υποσύνολο των κόμβων με τους οποίους συνδέεται (γείτονες). Με την σειρά τους αυτοί που έλαβαν το μήνυμα το προωθούν σε υποσύνολο των δικών τους γειτόνων και ούτω καθεξής. Όταν κάποιος peer ανακαλύψει ότι έχει την πληροφορία, επικοινωνεί με τον peer που ξεκίνησε την αναζήτηση και δεν προωθεί το μήνυμα παραπέρα.

Για λόγους απόδοσης, η αναζήτηση αυτή δεν συνεχίζεται απ' άπειρον: το μήνυμα προωθείται μόνο t φορές. Η παράμετρος t ονομάζεται time to live (TTL). Επιπλέον ένας peer μπορεί να λάβει το ίδιο μήνυμα αναζήτησης παραπάνω από μια φορά λόγω των πιθανών κύκλων που μπορεί να υπάρχουν στον γράφο των συνδέσεων. Έτσι, για να αποφευχθούν οι επαναμεταδώσεις, αν ένας κόμβος προωθήσει ένα μήνυμα, δεν θα το προωθήσει ξανά στο μέλλον (και πάλι δεν εξαλείφονται πλήρως οι πολλαπλές λήψεις του ίδιου ερωτήματος). Οι τεχνικές αυτές συνήθως εισάγουν μεγάλο όγκο μηνυμάτων στο δίκτυο. Αναπτύχθηκαν αρκετές παραλλαγές της τεχνικής αυτής για την αναζήτηση σε ένα αδόμητο peer to peer δίκτυο.



Σχήμα A.10: Μέθοδοι αναζήτησης σε unstructured P2Pδίκτυα

Plain flooding

Στο απλό flooding, ο κάθε peer προωθεί το μήνυμά του σε όλους τους γείτονές του. Καθώς προωθούνται τα μηνύματα σχηματίζεται το δέντρο του σχήματος A.10α. Ο όρος δέντρο ίσως να μην είναι αρκετά ακριβής γιατί κάποιος κόμβος μπορεί να υπάρχει παραπάνω από μία φορά στον γράφο αυτό (αφού υπάρχουν κύκλοι) αλλά βοηθάει στο να οπτικοποιήσουμε την επικοινωνία. Φυσικά η επικοινωνία σταματά όταν φτάσουμε σε ένα μέγιστο βάθος t και δεν επαναπροωθούνται ίδια queries.

Το βασικό μειονέκτημα του flooding είναι ο πολύ μεγάλος αριθμός μηνυμάτων που πλημμυρίζουν το δίκτυο ο οποίος αυξάνεται εκθετικά ως προς το βάθος t της αναζήτησης αλλά και τον βαθμό του κάθε κόμβου (αριθμός γειτόνων). Όμως ο αλγόριθμος αυτός έχει και την μεγαλύτερη πιθανότητα να βρει την πληροφορία, αν αυτή υπάρχει στο δίκτυο.

Teeming

Το teeming είναι μια πιθανοτική μέθοδος αναζήτησης που σχεδιάστηκε με στόχο να μειωθεί ο αριθμός των μηνυμάτων. Ο κάθε peer προωθεί το μήνυμά του σε ένα τυχαίο υποσύνολο των γειτόνων του. Ορίζουμε την πιθανότητα φ να προωθήσουμε το μήνυμα σε κάποιον γείτονα. Έτσι αν k είναι οι γείτονες ενός peer, το μέγεθος του υποσυνόλου αυτού μπορεί να είναι από 0 ως k, με μέσο μέγεθος $k \cdot \varphi$. Το απλό flooding μπορεί να θεωρηθεί ως μια ειδική περίπτωση του teeming με $\varphi = 1$ (σχήμα A.10). Παρόμοια τεχνική είναι το modified random BFS [11].

Ο αλγόριθμος αυτός εγγυάται πιθανοτική επιτυχία της αναζήτησης. Όσο πιο μικρό είναι το φ, τόσο λιγότεροι κόμβοι θα διερευνηθούν με αποτέλεσμα να έχουμε και μικρότερη πιθανότητα εύρεσης αποτελέσματος. Όμως περιορίζει αρκετά τον αριθμό των μηνυμάτων όπως έδειξαν στο [10].

Teeming with decay

Καθώς κατεβαίνουμε σε χαμηλότερα επίπεδα του δέντρου αναζήτησης υπάρχουν πολλαπλά αντίγραφα του μηνύματος αναζήτησης στο δίκτυο. Στο flooding για παράδειγμα τα μηνύματα αυτά αυξάνουν εκθετικά. Για να αντιμετωπιστεί αυτό το πρόβλημα μπορούμε καθώς το επίπεδο αυξάνεται, να μειώνουμε το υποσύνολο των γειτόνων στο οποίο θα προωθηθεί το μήνυμα. Έτσι η πιθανότητα φ δεν παραμένει σταθερή αλλά είναι συνάρτηση του βάθους της αναζήτησης. Πιο συγκεκριμένα, $\varphi = f(s, d)$ όπου s είναι το βάθος του κόμβου στο δέντρο (step) και d είναι μία παράμετρος που την ονομάζουμε decay parameter. Μια αποτελεσματική συνάρτηση f είναι αυτή που μειώνει εκθετικά την πιθανότητα φ δηλαδή $\varphi = (1 - d)^s$.

Random paths [12, 10]

Στον αλγόριθμο αυτό ο peer που ξεκινά την αναζήτηση επικοινωνεί με p γείτονές του ($p < k$). Από εκεί και πέρα κάθε peer προωθεί το μήνυμα μόνο σε έναν τυχαία επιλεγμένο γείτονά του. Έτσι δημιουργούνται p μονοπάτια αναζήτησης μήκους t. Στον αλγόριθμο αυτό επιλέγεται συνήθως πολύ μεγαλύτερο TTL t. Είναι φανερό ότι ο αλγόριθμος αυτός έχει το μικρότερο φόρτο μηνυμάτων, όμως έχει και την μικρότερη πιθανότητα επιτυχούς αναζήτησης (σχήμα A.10).

Υπάρχουν πολλές παραλλαγές αυτού του αλγορίθμου. Ο k-walker random algorithm [13] παράγει k-walkers που προχωράνε στο δίκτυο επικοινωνώντας περιοδικά με τον query node για να μάθουν αν πρέπει να σταματήσουν ή να συνεχίσουν (αν η πληροφορία βρέθηκε). Στο Two-Level Random Walk Search Protocol [14] υπάρχουν δύο TTL. Μέχρι να λήξει το πρώτο TTL παράγονται walkers που συνεχίζουν μέχρι να λήξει το δεύτερο.

Iterative deepening

Η μέθοδος του iterative deepening προτάθηκε ([15, 16, 12]) με στόχο να ελαχιστοποιήσει τα μηνύματα. Χρησιμοποιείται όταν γνωρίζουμε ότι η πληροφορία έχει μεγάλες πιθανότητες να βρεθεί στην γειτονιά ενός κόμβου. Ουσιαστικά καλούμε μία μέθοδο αναζήτησης με TTL = 1. Αν αποτύχει, επαναλαμβάνουμε την αναζήτηση με TTL = 2 κ.ο.κ. μέχρι να βρούμε την πληροφορία. Λόγω της εκθετικής αύξησης των μηνυμάτων με το TTL, είναι πιθανό μια επαναληπτική αναζήτηση για TTL 1,2,3 μπορεί να επιφέρει συνολικά λιγότερα μηνύματα σε σχέση με μια απευθείας αναζήτηση με TTL = 4 για παράδειγμα.

Directed BFS and intelligent search [12]

Η ιδέα πίσω από αυτόν τον teeming αλγόριθμο είναι ότι επιλέγεται ένα υποσύνολο που έχει μεγάλη πιθανότητα να δώσει γρήγορα επιτυχές αποτέλεσμα αναζήτησης. Για την επιλογή των «καλών» γειτόνων, κάθε κόμβος φυλά απλά στατιστικά για τους γείτονές του. Για παράδειγμα:

- Τον μέγιστο αριθμό επιτυχημένων απαντήσεων ως τώρα
- Το μικρότερο hop-count που χρειάστηκε για μια προηγούμενη αναζήτηση (latency)
- Το πόσο σταθερός είναι ο γείτονας (αν υπάρχει για καιρό εκεί)
- Ο γείτονας με το μικρότερο message queue (ο λιγότερο απασχολημένος).

Με την σωστή επιλογή γειτόνων μπορούμε να μειώσουμε τον αριθμό των μηνυμάτων, να έχουμε ποιοτικές αναζητήσεις και μικρότερο response time. Στο [12] μόνο ο query node χρησιμοποιούσε στατιστικά για να προωθήσει το μήνυμα. Όλοι οι άλλοι κόμβοι έκαναν απλό flooding. Στο [11] όμως χρησιμοποιήθηκαν «keywords» και έτσι γεννήθηκε η ιδέα του keyword query που αποτελούνταν από τέσσερις μηχανισμούς: αναζήτησης, profiling, αξιολόγηση των peer, εύρεσης παραπλήσεων ερωτήσεων. Έτσι ο κάθε κόμβος επιλέγει γείτονες που απάντησαν γρήγορα σε παρόμοιες ερωτήσεις σε παρελθόν.

Local indices based search [12]

Κάθε κόμβος φυλάει ευρετήριο με δεδομένα αναζήτησης για τους k-hop γείτονές του. Έτσι αν έρθει ένα query μπορεί γρήγορα να απαντήσει αν κάποιος από τους k-hop γείτονες έχει την πληροφορία. Έτσι μπορούμε να απαντήσουμε σε μία ερώτηση με την ίδια ακρίβεια απασχολώντας λιγότερους κόμβους. Οι κόμβοι επεξεργάζονται ένα μήνυμα αναζήτησης μόνο αν το βάθος αναζήτησης d ανήκει σε ένα κοινό policy P που είναι ίδιο για όλους τους κόμβους. Αν το βάθος d δεν ανήκει στο P τότε απλά προωθούν το μήνυμα στους γείτονες με βάθος d+1.

Για παράδειγμα αν $k = 2$ και $P = 0, 3, 6$ τότε ο κόμβος αναζήτησης (βάθος 0) θα επεξεργαστεί το μήνυμα (δηλαδή θα δει αν υπάρχει αποτέλεσμα στο ευρετήριο της 2-hop γειτονιάς). Αν δεν υπάρχει θα προωθήσει το μήνυμα στους γείτονες με βάθος $d = 1$. Αυτοί δεν θα επεξεργαστούν το μήνυμα γιατί το $d = 1$ δεν ανήκει στο policy P. Έτσι, το μήνυμα απλά θα προωθηθεί στους γείτονες με βάθος $d = 2$ και μετά στους γείτονες με βάθος $d = 3$. Οι κόμβοι με βάθος $d = 3$ ανήκουν στο P οπότε θα γίνει αναζήτηση στο τοπικό ευρετήριο κ.οκ.

Τα ευρετήρια αυτά πρέπει να ανανεωθούν κάθε φορά που κάποιος peer εισέρχεται ή φεύγει από το σύστημα. Η ανίχνευση για failures γίνεται με χρήση μηνυμάτων ping. Περισσότερες πληροφορίες θα βρείτε στο [12].

Routing indices based search [29]

Η μέθοδος αυτή μοιάζει πολύ με την μέθοδο Directed BFS and intelligent search [12]. Ενώ η intelligent search βασίζεται σε στατιστικά προηγουμένων ερωτήσεων για να επιλέξει το υποσύνολο που θα προωθηθεί το μήνυμα, η routing indices αποθηκεύει πληροφορίες ως προς τα θέματα ή τον αριθμό τον αρχείων που φυλάσσει ο κάθε γείτονας. Έτσι η μέθοδος αυτή ταιριάζει σε αναζητήσεις ως προς το περιεχόμενο (content search queries) αντί για αναζητήσεις ως προς κάποιο filename. Τα αρχεία μπορούν να ανήκουν σε παραπάνω από ένα topic και κάθε κόμβος φυλάσσει πληροφορίες (keywords) τα δικά του αρχεία. Έτσι σε κάποια αναζήτηση επιλέγονται οι m καλύτεροι γείτονες. Καλοί γείτονες θεωρούνται αυτοί που έχουν αρχεία με topics κοντά ως προς αυτό που αναζητάμε ή έχουν πολλά αρχεία.

Adaptive probabilistic search [30, 31]

Στην μέθοδο αυτή υποθέτουμε ότι αποθηκεύονται αντικείμενα στο δίκτυο με μία συγκεκριμένη κατανομή. Ο αλγόριθμος βασίζεται στον k-walker random walk and probabilistic forwarding. Ο κάθε κόμβος προωθεί τον μήνυμα στον γείτονα με την μεγαλύτερη πιθανότητα να απαντήσει. Η πιθανότητα αυτή υπολογίζεται με βάση προηγούμενα queries και ανανεώνεται με βάση το αποτέλεσμα του τωρινού query (αν η αναζήτηση ήταν θετική η πιθανότητα αυξάνεται ενώ αν απέτυχε μειώνεται).

Για να επιλέξουμε τον κατάλληλο γείτονα, κάθε κόμβος φυλάει ένα ευρετήριο με τους γείτονές του. Στο ευρετήριο κάθε γείτονα υπάρχει ένα ευρετήριο για κάθε object για το οποίο ερωτήθηκε και προώθησε σε αυτόν. Η τιμή στο ευρετήριο αναπαριστά την πιθανότητα να προωθήσουμε το query για το συγκεκριμένο object στον συγκεκριμένο γείτονα.

Αρχικά όλες οι πιθανότητες είναι ίδιες και ο κόμβος που θα προωθηθεί το μήνυμα επιλέγεται τυχαία. Αν είναι επιτυχής η αναζήτηση η πιθανότητα αυξάνεται αλλιώς μειώνεται. Έτσι σιγά σιγά μπορούμε να βρούμε ποιοι είναι οι πιο καλοί γείτονες για να μας απαντήσουν στο συγκεκριμένο object query.

Dominating set based search [32]

Στην μέθοδο αυτή επιλέγεται ένας αριθμός από κόμβους ώστε να φτιάξουν ένα connected dominating set (CDS). Ένα CDS σε ένα P2P δίκτυο είναι ένα υποσύνολο των κόμβων που συνδέονται άμεσα μεταξύ του. Όλοι οι άλλοι κόμβοι έχουν τουλάχιστον ένα γείτονα στο CDS. Η αναζήτηση επιτυγχάνεται με ένα τυχαίο περπάτημα πάνω στο dominating set. Η κατασκευή του dominating set κατασκευάζεται μόνο με τοπική πληροφορία (2-hop) όπως έδειξαν στο [32].

Η αναζήτηση γίνεται ως εξής. Αν ο κόμβος αναζήτησης δεν είναι στο dominating set, προωθεί το μήνυμά του σε έναν dominating γείτονα. Ο κόμβος στο dominating set ψάχνει στην τοπική βάση δεδομένων και αν δεν βρει απάντηση προωθεί το μήνυμα στους γείτονές του. Οι γείτονες που είναι στο dominating set θα προωθήσουν παραπέρα την πληροφορία ενώ οι υπόλοιποι απλά θα ψάξουν τοπικά για απάντηση.

Η οργάνωση της μεθόδου αυτής ταιριάζει σε συστήματα που έχουν super peers όπως το kazaa και το morpheus.

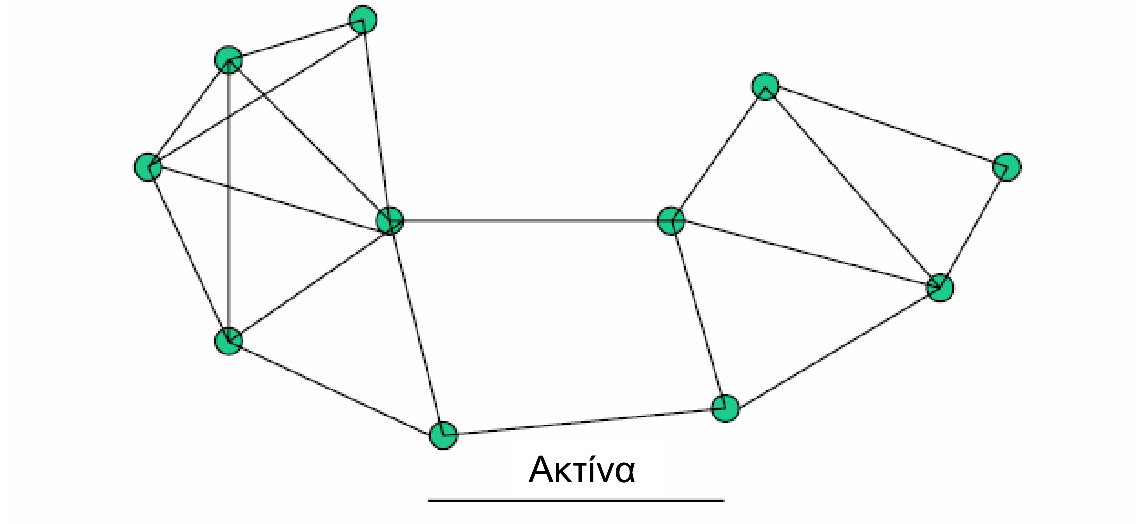
A.2 Mobile ad-hoc networks

Τα mobile ad-hoc δίκτυα (MANETs) αποτελούνται από ασύρματους κόμβους που επικοινωνούν μεταξύ τους χωρίς κάποια σταθερή δικτυακή υποδομή. Χρησιμοποιούνται σε καταστάσεις καταστροφής, σε πεδία μάχης, σε συνέδρια, και έχουν κεντρίσει το ενδιαφέρον τα τελευταία χρόνια κυρίως με την χρήση των δικτύων από ανιχνευτές (sensor networks). Τέτοια δίκτυα είναι πιθανό να υλοποιηθούν στο άμεσο μέλλον με στόχο να καταγράφουν και να ελέγχουν το φυσικό περιβάλλον σε απομακρυσμένες περιοχές. Με το να δικτυώσουμε αυτούς τους ανιχνευτές, επιτυγχάνουμε ακριβέστερη καταγραφή της πληροφορίας μέσω της συνεργασίας μεταξύ τους. Μητροπολιτικά δίκτυα έχουν αναπτυχθεί σε κάποιες πόλης χρησιμοποιώντας κεραίες στις οροφές των κτηρίων ως ένα εναλλακτικό ασύρματο δίκτυο.

Στα ad-hoc δίκτυα, ασύρματοι κόμβοι επικοινωνούν μεταξύ τους αφού δεν υπάρχει άλλη σταθερή δικτυακή υποδομή. Ως επί το πλείστων, τα δίκτυα αυτά αποτελούνται από ομότιμους κόμβου που επικοινωνούν ασύρματα χωρίς κάποιον κεντρικοποιημένο έλεγχο. Στα ασύρματα δίκτυα η επικοινωνία μεταξύ δύο κόμβων μπορεί να γίνει απ'ευθείας μόνο αν οι δύο κόμβοι είναι εντός εμβελείας του ασύρματου εξοπλισμού. Αν κάτι τέτοιο δεν είναι δυνατό τότε πρέπει να γίνει επικοινωνία *χρησιμοποιώντας ενδιάμεσους κόμβους* (multi-hop communication). Αυτή είναι και η βασικότερη διαφορά με τα P2P δίκτυα όπου δυο κόμβοι θεωρούμε ότι μπορούν να επικοινωνήσουν απευθείας μέσω του internet.

Ένα αποδεκτό μοντέλο για την θεωρητική μελέτη τέτοιων δικτύων είναι το *unit graph model*: δύο κόμβοι A και B συνδέονται πάντα όταν η απόσταση τους είναι μικρότερη από R, όπου R είναι η ακτίνα μετάδοσης. Το μοντέλο αυτό κάνει πολλές υποθέσεις αφού θεωρεί ότι όλοι οι κόμβοι έχουν την ίδια εμβέλεια R, ότι δεν υπάρχουν εμπόδια (υπο-γράφοι του unit-graph), ότι δεν υπάρχουν κατευθυντικές κεραίες (στην πράξη μπορεί να μην συνδεόμαστε με όλους τους κόμβους που είναι μέσα σε μία ακτίνα αλλά μόνο σε όσους είναι μπροστά μας για παράδειγμα). Όμως είναι και το μοναδικό μοντέλο που μελετάται στην βιβλιογραφία. Το σχήμα A.11 μας δείχνει ένα παράδειγμα ενός τέτοιου γράφου με την δοσμένη ακτίνα R.

A.2.1 Παραδείγματα MANET



Σχήμα A.11: Το μοντέλο unit graph: Κάθε κόμβος που βρίσκεται σε απόσταση μικρότερη της ακτίνας συνδέεται

Δίκτυα ανιχνευτών (Sensor networks) [33]

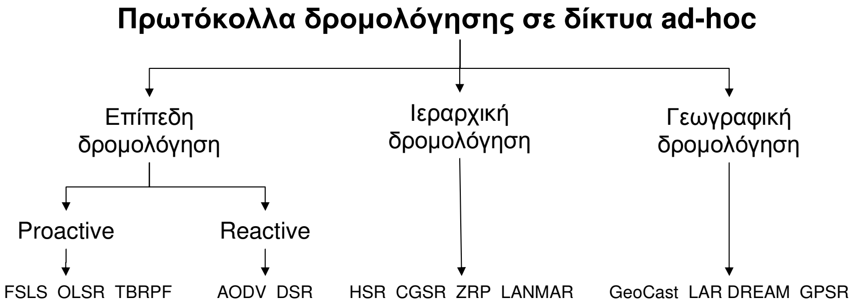
Τα δίκτυα αυτά, καλούνται επίσης υβριδικά ad-hoc δίκτυα, χρησιμοποιούνται για να συλλέγουν δεδομένα όπως θερμοκρασία, ανίχνευση χημικών, κίνηση κτλ. Πρόκειται για ένα δίκτυο από ανιχνευτές που επικοινωνούν μεταξύ τους με στόχο να συνεργαστούν ώστε να ανταλλάξουν πληροφορίες και να εξάγουν ένα συμπέρασμα.

Μητροπολιτικά δίκτυα [34, 35, 36]

Μερικά δίκτυα συνδέονται με σταθερή υποδομή χρησιμοποιώντας wireless access points. Για παράδειγμα, μπορεί να φτιαχτεί ένα πλέγμα από κεραίες σε ταράτσες κτιρίων με στόχο την δημιουργία ενός εναλλακτικού, δωρεάν, δικτύου των κατοίκων μιας πόλης.

Δίκτυα οχημάτων [34, 35, 36]

Τα οχήματα σε έναν δρόμο μπορούν να δημιουργήσουν ένα ad-hoc δίκτυο με στόχο να καταγράψουν πληροφορίες για το κυκλοφοριακό, να πληροφορήσουν για πιθανά ατυχήματα κ.α. Το κάθε όχημα μπορεί για παράδειγμα όταν ανιχνεύσει ένα κυκλοφοριακό πρόβλημα να μεταδώσει σε όλα τα οχήματα εντός εμβέλειας την πληροφορία και εκείνα να την προωθήσουν παραπέρα.



Σχήμα A.12: Κατηγορίες πρωτοκόλλων δρομολόγησης σε MANET (ad-hoc routing protocols)

Τηλεφωνικά δίκτυα [37]
Τα multihop δίκτυα κινητής τηλεφωνίας έχουν προταθεί πρόσφατα ως ένας εναλλακτικός τρόπος επικοινωνίας σε καταστάσεις όπου υπάρχουν πολλοί χρήστες σε περιορισμένη περιοχή όπως για παράδειγμα ένα στάδιο.

Φυσικά υπάρχουν εκατοντάδες άλλες χρήσεις των συστημάτων MANET.

A.2.2 Δρομολόγηση σε ad-hoc networks (routing)

Είναι φανερό ότι λόγω της multi-hop επικοινωνίας, το βασικό πρόβλημα που καλείται να αντιμετωπίσει κάποιος σε ad-hoc δίκτυα είναι η δρομολόγηση των πακέτων από έναν κόμβο σε έναν άλλο. Υπάρχει πολλή ερευνητική δουλεία γύρω από το θέμα που συνοψίζεται στα surveys [38, 39]. O Xiaoyan Hong χωρίζει τα πρωτόκολλα αυτά σε τρεις βασικές κατηγορίες όπως φαίνεται και στο σχήμα A.12

- **Flat routing:** 'Ολοι οι κόμβοι είναι το ίδιο και δεν χρησιμοποιείται γεωγραφική πληροφορία.
  - **Proactive:** Υπάρχει κάποια background πληροφορία που αφορά το routing. Οι κόμβοι συλλέγουν πληροφορίες από παλιότερες δρομολογήσεις και τις κρατάνε

σε κάποια βάση δεδομένων για να δρομολογήσουν τις επόμενες. Δεν έχουμε (συνήθως) διαφορετική διαδρομή κάθε φορά που υπάρχει μια επικοινωνία.

  – **Reactive:** On demand route αναζήτηση για κάθε επικοινωνία.

- **Hierarchical routing:** υπάρχουν κάποιοι ειδικοί κόμβοι στο δίκτυο που αναλαμβάνουν να βοηθούν ή να κάνουν το routing.
- **Geographic positioning assisted routing:** υπάρχει κάποιος GPS receiver σε κάθε κόμβο. Η γεωγραφική πληροφορία βοηθάει στο routing.

A.2.2.1 Flat routing Proactive αλγόριθμοι

FSR (Fisheye State Routing)
Ανταλλάσσει περιοδικά όλες τις routing πληροφορίες που έχει αποθηκευμένες με τους γείτονες. Η συχνότητα μετάδοσης αυτών των πληροφοριών εξαρτάται από την απόσταση των κόμβων (hop-distance). 'Ετσι routes για μακρινούς κόμβους μεταδίδονται στους γείτονες λιγότερο συχνά.

OLSR (Optimized Link State Routing)
Ο αλγόριθμος αυτός προσπαθεί να αφαιρέσει μερικούς από τους γείτονές του που δεν χρειάζονται στο routing για να αποφύγει επαναμεταδώσεις.. Κάποιος κόμβος μαθαίνει από τους γείτονες του όλους τους 2hop γείτονες και μετά υπολογίζει το minimum set 1-hop t κόμβων που έχουν ως γείτονες όλους τους 2-hop κόμβους. Θα επικοινωνεί μόνο με αυτούς για το routing από εκεί και πέρα. Ταιριάζει σε πυκνά δίκτυα αφού οι MPR κόμβοι θα είναι λίγοι ενώ σε αραιά δίκτυα κάθε κόμβος θα λαμβάνει μέρος στο routing οπότε ο αλγόριθμος γίνεται ισοδύναμος με τον προηγούμενο.

TBRPF (Topology Broadcast Based on Reverse Path Forwarding)
Αποτελείται από δύο μέρη: το μέρος της ανακάλυψης γειτόνων και το routing μέρος. Στο πρώτο μέρος, στέλνονται μηνύματα hello που αναφέρουν αλλαγές στους γείτονες. Συλλέγονται τοπικές πληροφορίες για την τοπολογία και φτιάχνεται ένα τοπικό σκελετικό δέντρο με

τα minimum-hop paths. Το routing γίνεται χρησιμοποιώντας αυτούς τους κόμβους μόνο. Update στέλνουμε μόνο αν οι αλλαγές επηρεάζουν το δέντρο αυτό.

A.2.2.2 Flat routing On demand αλγόριθμοι

AODV (Ad Hoc On-demand Distance Vector routing)

Χρησιμοποιεί την έννοια του backward learning: Με το που λάβουμε ένα query, ο κόμβος μετάβασης μαθαίνει και αποθηκεύει το route από την πηγή ως αυτόν. Έτσι μαθαίνει σιγά σιγά routes για κόμβους του δικτύου. Αν κάποιος κόμβος έχει το route για τον προορισμό στέλνει την απάντηση και δεν προωθεί το μήνυμα παραπέρα.

DSR (Dynamic Source Routing)

Στην ερώτηση κάθε ενδιάμεσος κόμβος προσθέτει πληροφορία για τον εαυτό του στο πακέτο. Κατά την απάντηση το πακέτο επιστρέφεται με την αντίθετη φορά και όλοι οι κόμβοι κάνουν cache τις πληροφορίες για τα routes που προκύπτουν.

A.2.2.3 Ιεραρχική δρομολόγηση

Όταν το μέγεθος του δικτύου αυξάνεται πολύ, το flat routing γίνεται ασύμφορο. Η ιεραρχική δρομολόγηση βασίζεται στην οργάνωση των κόμβων σε ομάδες και στην ανάθεση διαφορετικών λειτουργιών για εργασίες που αφορούν εσωτερικές ή εξωτερικές υποθέσεις της ομάδας.

CGSR (Clusterhead-Gateway Switch Routing)

Χωρίζει το δίκτυο σε clusters και σε κάθε cluster ορίζεται ένας επικεφαλής. Κάθε κόμβος που ανήκει σε δύο ή και παραπάνω clusters ονομάζεται gateway. Όταν ένας κόμβος λάβει κάποιο μήνυμα το προωθεί στον επικεφαλή και εκείνος γνωρίζει αν ανήκει στο δικό του cluster ή όχι. Αν δεν ανήκει τότε έχει routes για το gateway που θα το στείλει ώστε να παρωθηθεί στο επόμενο cluster.

HSR (Hierarchical State Routing)

Παρόμοιο με το CGSR, μόνο που υπάρχουν τριών ειδών κόμβοι: cluster heads, gateways, and internal nodes. Πλέον υπάρχουν πολλαπλά επίπεδα και οι επικεφαλείς κάποιου επιπέδου είναι κόμβοι του αμέσως υψηλότερου επιπέδου.

ZRP (Zone Routing Protocol)

Πρόκειται για ένα υβριδικό πρωτόκολλο που συνδυάζει proactive και on-demand routing. Οι κόμβοι χωρίζονται σε ζώνες και εσωτερικά της ζώνης χρησιμοποιούμε proactive routing (routing tables) ενώ για επικοινωνία μεταξύ ζωνών χρησιμοποιούμε on-demand routing.

LANMAR (Landmark Ad Hoc Routing Protocol)

Παρόμοιο με το ZRP μόνο που το κάθε group έχει μία σημαδούρα που χρησιμεύει στο να ανιχνευθεί η θέση ολόκληρου του group. Η μέθοδος αυτή είναι χρήσιμη όταν έχουμε μετακινήσεις σε groups.

A.2.2.4 Geographic Position Information Assisted Routing

Geocast

Χρησιμοποιεί συγκεκριμένη γεωγραφική πληροφορία για να καθορίσει τον προορισμό. Υπάρχει ένας ειδικός (centralized) υπολογιστής που είναι υπεύθυνος να λαμβάνει και να στέλνει αυτές τις πληροφορίες (GeoHost). Ο GeoHost είναι υπεύθυνος και στο να στέλνει τα πακέτα στον κατάλληλοGeoRouter. Για καλύτερη διαχείριση το δίκτυο εξυπηρετείται από πολλούς GeoRouters που είναι ιεραρχικά εκλεγμένοι και κάθε GeoRouter αναλαμβάνει να εξυπηρετήσει μία συγκεκριμένη γεωγραφική περιοχή.

LAR (Location-Aided Routing)

Ο LAR χρησιμοποιεί γεωγραφική πληροφορία για να περιορίσει τον χώρο αναζήτησης ενός νέου route σε μία μικρότερη ζώνη. Μοιάζει με τον DSR: χρησιμοποιεί την έννοια του limited flooding. Αρχικά υπολογίζει την κυκλική περιοχή που μπορεί να είναι ο προορισμός (π.χ.

από την παλιά του θέση και την απόσταση που είναι δυνατόν να διάνυσε). Κατά την διάρκεια του flooding το μήνυμα προωθείται μόνο από κόμβους που είναι μέσα στο τετράγωνο που ορίζει αυτός ο κύκλος και ο κόμβος της ερώτησης. Εναλλακτικά, το μήνυμα προωθείται σε κάποιο γείτονα μόνο αν ο είναι πιο κοντά στον προορισμό (δεν πάμε προς τα πίσω).

DREAM (Distance Routing Effect Algorithm for Mobility)

Κάθε κόμβος κρατάει ένα location table για όλους τους κόμβους. Στέλνονται control packets σχετικά με τις νέες θέσεις των κόμβων ανά τακτά χρονικά διαστήματα. Η συχνότητα που στέλνονται αυτά τα updates εξαρτάται από δύο παραμέτρους. Distance effect, όσο μεγαλύτερη είναι η απόσταση που χωρίζει δύο κόμβους τόσο αραιότερα στέλνονται αυτά τα μηνύματα. Mobility rate: όσο πιο γρήγορα κινείται ένας κόμβος τόσο πιο συχνά πρέπει να διαφημίζει την νέα του θέση.

GPSR (Greedy Perimeter Stateless Routing)

Χρησιμοποιεί μόνο τοπική πληροφορία για τους γείτονές του για να προωθήσει το μήνυμα. Κάθε φορά που ένας κόμβος μετακινείται, ενημερώνει μόνο τους γείτονες του κάτι που σημαίνει μικρή 1-hop επικοινωνία. Τα μηνύματα προωθούνται χρησιμοποιώντας μεθόδους greedy forwarding: κάθε φορά προωθούμε το μήνυμα στον γείτονα που είναι πιο κοντά στον προορισμό. Αν αποτύχει αυτή η διαδικασία χρησιμοποιεί perimeter-forwarding χρησιμοποιώντας τον κανόνα του δεξιού χεριού: κάθε κόμβος προωθεί το μήνυμα στον δεξιότερο γείτονα του που όμως είναι αριστερά από την ευθεία που ορίζει η πηγή με τον προορισμό με αποτέλεσμα να κινούμαστε περιμετρικά της ευθείας.

Παράρτημα B

Ευρετήριο υλοποίησης

B.1 Ομάδες συναρτήσεων

Για την διευκόλυνση του simulation, οι συναρτήσεις χωρίστηκαν σε πέντε ομάδες ανάλογα με την εργασία που επιτελούν. Ακολουθεί η λίστα όλων των ομάδων:

B.2 Δομές Δεδομένων

Ακολουθούν οι δομές δεδομένων χρησιμοποιούνται στον simulator με σύντομες περιγραφές:

B.3 Λίστα Αρχείων

Ακολουθεί μια λίστα όλων των αρχείων με σύντομες περιγραφές:

Παράρτημα C

Τεκμηρίωση μονάδων

C.1 Παράμετροι του simulation

Μπορούμε να εξομοιώσουμε οποιοδήποτε συνδυασμό των μεθόδων push και pull που περιγράψαμε στο κεφάλαιο 5. Στα πειράματά μας μπορούμε να μεταβάλουμε κάθε παράμετρο που επηρεάζει την απόδοση κάθε αλγορίθμου ώστε να εκτιμήσουμε τις επιπτώσεις της στο σύστημα. Οι παράμετροι αυτοί είναι:

- Ο αριθμός των πρακτόρων στο σύστημα.
- Το μέγεθος της cache K
- Τον ελάχιστο, μέγιστο και μέσο αριθμό resources που κατέχει κάθε πράκτορας.
- Την κατανομή των δημοφιλών resources (πόσες είναι, πόσο πιο δημοφιλείς είναι).
- Την πιθανότητα να μετακινηθεί κάποιος agent σε κάθε γύρο P_{move}.
- Την πιθανότητα να χρησιμοποιήσει ένα resource P_{use}.
- Για τον αλγόριθμο teeming with decay που χρησιμοποιείται στο push/pull (κεφ. 3.4)

  Η παράμετρος decay d

  Το μέγιστο βάθος TTL
- Για τον αλγόριθμο push με καταλόγους snooping (κεφ. 5.3.2)

  Expiration time στον κατάλογο snooping.

  Περίοδος του periodic pulling.

- Για τον αλγόριθμο Inverted cache push/pull με leasing

  Ο χρόνος του lease που εκδίδει ο ιδιοκτήτης της υπηρεσίας.

  Η πιθανότητα να γίνει ανανέωση η μίσθωση πριν λήξει.

Παρακάτω ακολουθούν τα ονόματα και οι περιγραφή των μεταβλητών στον κώδικα του εξωμοιωτή.

Μεταβλητές

- float **exp_time**

  Ο χρόνος σε turns στον οποίο λήγει το snooping entry.

- int **algorithm**

  Ο αλγόριθμος που χρησιμοποιείται αυτή την στιγμή.

- double **p_change**

  Πιθανότητα να γίνει αλλαγή στην cache κάποιου agent σε κάθε γύρο.

- float **p_move**

  Πιθανότητα να μετακινηθεί ένας agent σε κάθε γύρο.

- double **p_pull**

  Πιθανότητα να εκτελέσουμε on-demand pull σε κάθε γύρο.

- int **use_replication**

  Αν θα χρησιμοποιηθεί replication ή όχι.

- int **replica_time**

  Πόσο χρόνο θα κρατάμε τα replicas.

- double **p_replicate**

Η πιθανότητα να κάνουμε replicate σε συγκεκριμένο κόμβο του path.

- int **maximum_level**

  Το μέγιστο βάθος του push.

- int **maximum_pull_level**

  Το μέγιστο βάθος του pull.

- float **decay_par**

  Η παράμετρος decay.

- int **do_pull**

  Αν θα γινονται on-demand pulls.

- int **pull_decay_par**

  Το decay κατά το pull.

- float **replace_prob**

  Η πιθανότητα να προβούμε σε αντικατάσταση σε κάθε γύρο.

Τεκμηρίωση Μεταβλητών

C.1.0.5 int algorithm

Ο αλγόριθμος που χρησιμοποιείται αυτή την στιγμή.

Παίρνει τιμές του enum **alg_types**(σελ. 197). Ορισμός στη γραμμή 17 του αρχείου main.c.

Αναφορά από initialize(), pull_algorithm(), push_to_network(), set_simulation_parameters(), και simulation().

C.1.0.6 float decay_par

Η παράμετρος decay.

Ορισμός στη γραμμή 95 του αρχείου main.c.

Αναφορά από push_to_network(), και set_simulation_parameters().

C.1.0.7 int do_pull

Αν θα γινονται on-demand pulls.

Ορισμός στη γραμμή 96 του αρχείου main.c.

Αναφορά από set_simulation_parameters().

C.1.0.8 float exp_time

Ο χρόνος σε turns στον οποίο λήγει το snooping entry.

Ορισμός στη γραμμή 15 του αρχείου main.c.

Αναφορά από initialize(), push_to_network(), set_simulation_parameters(), και validate_agents_cache().

C.1.0.9 int maximum_level

Το μέγιστο βάθος του push.

Ορισμός στη γραμμή 93 του αρχείου main.c.

Αναφορά από push_to_network(), set_simulation_parameters(), simulation(), και stats_initialize_round().

C.1.0.10 int maximum_pull_level

Το μέγιστο βάθος του pull.

Ορισμός στη γραμμή 94 του αρχείου main.c.

Αναφορά από initialize_stats(), pull_algorithm(), set_simulation_parameters(), stats_finalize_round(), και stats_initialize_round().

C.1.0.11 double p_change

Πιθανότητα να γίνει αλλαγή στην cache κάποιου agent σε κάθε γύρο.

Ορισμός στη γραμμή 20 του αρχείου main.c.

C.1.0.12 float p_move

Πιθανότητα να μετακινηθεί ένας agent σε κάθε γύρο.

Ορισμός στη γραμμή 21 του αρχείου main.c.

Αναφορά από set_simulation_parameters(), και simulation().

C.1.0.13 double p_pull

Πιθανότητα να εκτελέσουμε on-demand pull σε κάθε γύρο.

Ορισμός στη γραμμή 22 του αρχείου main.c.

Αναφορά από set_simulation_parameters(), και simulation().

C.1.0.14 double p_replicate

Η πιθανότητα να κάνουμε replicate σε συγκεκριμένο κόμβο του path.

Ορισμός στη γραμμή 90 του αρχείου main.c.

Αναφορά από pull_algorithm(), και set_simulation_parameters().

C.1.0.15 int pull_decay_par

Το decay κατά το pull.

Ορισμός στη γραμμή 97 του αρχείου main.c.

Αναφορά από pull_algorithm(), και set_simulation_parameters().

C.1.0.16 float replace_prob

Η πιθανότητα να προβούμε σε αντικατάσταση σε κάθε γύρο.

Ορισμός στη γραμμή 99 του αρχείου main.c.

Αναφορά από set_simulation_parameters(), και simulation().

C.1.0.17 int replica_time

Πόσο χρόνο θα κρατάμε τα replicas.

Ορισμός στη γραμμή 89 του αρχείου main.c.

Αναφορά από pull_algorithm(), και set_simulation_parameters().

C.1.0.18 int use_replication

Αν θα χρησιμοποιηθεί replication ή όχι.

Ορισμός στη γραμμή 88 του αρχείου main.c.

Αναφορά από initialize(), pull_algorithm(), push_to_network(), set_simulation_parameters(), και simulation().

C.2 Κατασκευή του συνδεδεμένου γράφου.

Στην εξομοίωση αυτή αρχικά δημιουργούμε έναν δίκτυο από από πράκτορες. Κάθε πράκτορας κατέχει έναν αριθμό από υπηρεσίες (resources) τις οποίες μοιράζεται με τους άλλους. Μια υπηρεσία μπορεί να βρεθεί σε έναν μόνο agent αλλά ένας agent μπορεί να κατέχει παραπάνω από μια υπηρεσίες. Επιπλέον, κάθε πράκτορας έχει cache σταθερού μεγέθους k που φυλάσσει διευθύνσεις υπηρεσιών. Ξεκινάμε την εξομοίωση του δικτύου φτιάχνοντας έναν τυχαίο γράφο συνδέσεων: κάθε πράκτορας επιλέγει k υπηρεσίες τυχαία και τις προσθέτει στην cache του. Για να προσεγγίσουμε περισσότερο ένα πραγματικό P2Pδίκτυο, κάποιες υπηρεσίες είναι πιο δημοφιλείς από άλλες. Μπορούμε στις παραμέτρους της εξομοίωσης να ορίσουμε το πόσες είναι οι δημοφιλείς υπηρεσίες και το πόσο δημοφιλής είναι η κάθε υπηρεσία ή να ορίσουμε μια κατανομή. Αρχικά, οι εγγραφές στις cache των πρακτόρων του δικτύου είναι όλες έγκυρες. Έτσι ξεκινάμε από ένα "καλής ποιότητας" δίκτυο.

Εδώ θα βρείτε τις συναρτήσεις και μεταβλητές που έχουν ως στόχο την δημιουργία και αρχικοποίηση του γράφου των agents.

Συναρτήσεις

- int **ran** (double probability)

  Συνάρτηση που επιστρέφει 1 με δοσμένη πιθανότητα.

- void **print_G** ()

  Τυπώνει τον γράφο των agent.

- void **initialize** ()

  Αρχικοποιεί τον γράφο.

- void **destroy_graph** ()

  Καταστρέφει τον γράφο των agent (free memory).

- void **print_graph_stats** ()

Τυπώνει στατιστικά για τον γράφο μας.

- void **print_lists** (int cur_round)

  Τυπώνει λίστες με γείτονες για κάθε agent.

Μεταβλητές

- int **K**

  Το μέγεθος της cache.

- float **per_cache**

  Το ποσοστό των agent που υπάρχουν στην cache.Το K προκύπτει από αυτό.

- float **percentage_of_popular_agents**

  Το ποσοστό των agent που είναι δημοφηλείς.

- float **popularity**

  Το πόσο δημοφηλείς είναι οι δημοφηλείς agents.

Τεκμηρίωση Συναρτήσεων

C.2.0.19 void destroy_graph ()

Καταστρέφει τον γράφο των agent (free memory).

Ορισμός στη γραμμή 109 του αρχείου net_funcs.c.

References agents, και num_agents.

Αναφορά από batch_run().

```
110 {
111     int i;
112     for (i = 0; i < num_agents; i++) {
113
114         free(agents[i].cache);
115         free(agents[i].known_location);
116     }
117 }
```

C.2.0.20 void initialize ()

Αρχικοποιεί τον γράφο.

Δεσμεύει μνήμη για κάθε agent και αρχικοποιεί τις τιμές. Συνδέει τυχαία τον γράφο βάζοντας στις caches μόνο valid δεδομένα. Άν έχουμε simulation με replicas φτιάχνει αυτόματα ένα σύνολο από αρχικά replicas Ορισμός στη γραμμή 45 του αρχείου net_funcs.c.

References add_to_cache(), agents, algorithm, agent_t::cache, exp_time, agent_t::f_cache_head, agent_t::f_cache_size, agent_t::f_cache_tail, K, agent_t::known_location, LEASING, agent_t::location, num_agents, per_cache, percentage_of_popular_agents, popularity, agent_t::popularity, pull_algorithm(), pull_level_found, ran(), agent_t::time_to_validate, και use_replication.

Αναφορά από batch_run().

```
46 {
47     int i, j, k;
48     float total_prob = 0.0;     /* keep here the sum of the probabilities */
49
50     srand(time(NULL));
51
52
53
54     K = ((float) num_agents * per_cache);
55     if (K == 0)
56         K = 1;
57
58
59     for (i = 0; i < num_agents; i++) {
```

```

        agents[i].cache = (int *) malloc(sizeof(int) * K);
        agents[i].known_location = (int *) malloc(sizeof(int) * K);
        agents[i].location = 1;
        agents[i].f_cache_head = NULL;
        agents[i].f_cache_tail = NULL;
        agents[i].f_cache_size = 0;
        agents[i].time_to_validate =
            exp_time * ((float) rand() / RAND_MAX);

        if (ran(percentage_of_popular_agents))
            agents[i].popularity = popularity;
        else
            agents[i].popularity = 1.0;

        total_prob += agents[i].popularity;

    }

    for (i = 0; i < num_agents; i++) {
        k = 0;
        while (k < K) {
            agents[i].known_location[k] = 1;
            agents[i].cache[k] = num_agents * rand() / RAND_MAX;

            if (ran
                (((float) agents[agents[i].cache[k]].popularity /
                  total_prob))) {
                if (algorithm == LEASING) {
                    add_to_cache(agents[i].cache[k], exp_time, i, 1);
                }
                k++;

            }

        }
    }
    if (use_replication)
        for (i = 0; i < num_agents; i++)
            for (k = 0; k < K; k++)
                 if (ran(0.3)) {
                     pull_level_found = -1;
                     pull_algorithm(i, 1, agents[i].cache[k], 0);
```

```
103                 }
104 }
```

C.2.0.21 void print_G ()

Τυπώνει τον γράφο των agent.

Ορισμός στη γραμμή 24 του αρχείου net_funcs.c.

References agents, K, και num_agents.

```
25 {
26     int i, j, k = 0;
27
28
29     for (i = 0; i < num_agents; i++) {
30         printf("%d knows\n", i);
31         for (j = 0; j < K; j++) {
32             printf("%d ", agents[i].cache[j]);
33
34         }
35         printf("\n\n");
36     }
37 }
```

C.2.0.22 void print_graph_stats ()

Τυπώνει στατιστικά για τον γράφο μας.

Τυπώνει πληροφορίες όπως τον βαθμό (rank) κάθε κόμβου, το πόσο δημοφιλής είναι κτλ... Ορισμός στη γραμμή 124 του αρχείου net_funcs.c.

References agents, agent_t::cache, inv_cache_percentage, K, και num_agents.

Αναφορά από batch_run().

```
125 {
```

```
    int i, j;
    int rank[num_agents];
    int max = 0;
    int av = 0;


    int av_pop = 0;
    int num_pop = 0;

    int av_non_pop = 0;
    int num_non_pop = 0;

    for (i = 0; i < num_agents; i++)
        rank[i] = 0;

    for (i = 0; i < num_agents; i++)
        for (j = 0; j < K; j++)
            rank[agents[i].cache[j]]++;


/*
        for (i = 0 ; i < num_agents; i++){
                printf("%d, pop = %f, agents = %d\n"
                        ,i, agents[i].popularity, rank[i]);
                if (agents[i].popularity == 1) {
                        av_non_pop += rank[i];
                        num_non_pop++;
                }
                else {
                        av_pop += rank[i];
                        num_pop++;
                }
        }
        printf("NON POPULAR %d->av = %f\nPOPULAR %d->av = %f\n"
                ,num_non_pop
                ,(float)av_non_pop/num_non_pop
                ,num_pop,(float)av_pop/num_pop);
*/
    for (i = 0; i < num_agents; i++) {
        if (rank[i] > max)
            max = rank[i];
        av += rank[i];
    }
```

```
    av = av / num_agents;
    printf("MAX:%f AV:%f\n\n", max * inv_cache_percentage,
           av * inv_cache_percentage);
}
```

C.2.0.23 void print_lists (int cur_round)

Τυπώνει λίστες με γείτονες για κάθε agent.

Παράμετροι:

cur_round ο γύρος του simulation που είμαστε τώρα.

Ορισμός στη γραμμή 177 του αρχείου net_funcs.c.

References agents, flood_cache_t::expire, agent_t::f_cache_head, flood_cache, flood_cache_t::next, και num_agents.

```
{
    int i;
    flood_cache *p;

    printf("%d\n", cur_round);
    for (i = 0; i < num_agents; i++) {
        p = agents[i].f_cache_head;
        printf("%d (%d) :", i, agents[i].f_cache_size);
        while (p != NULL) {
            printf("%d->", p->expire);
            p = p->next;
        }
        printf("\n");
    }
}
```

C.2.0.24 int ran (double probability)

Συνάρτηση που επιστρέφει 1 με δοσμένη πιθανότητα.

Παράμετροι:

probability double. Με την πιθανότητα αυτή θα επιστραφεί 1.

Επιστρέφει:

Επιστρέφεται 1 ή 0 με πιθανότητα probability.

Ορισμός στη γραμμή 14 του αρχείου net_funcs.c.

Αναφορά από initialize(), move_inverted(), pull_algorithm(), push_to_network(), και simulation().

```
15 {
16     if ((double) rand() / RAND_MAX <= probability)
17         return 1;
18     else
19         return 0;
20 }
```

Τεκμηρίωση Μεταβλητών

C.2.0.25 int K

Το μέγεθος της cache.

Ορισμός στη γραμμή 31 του αρχείου main.c.

Αναφορά από final_consistency(), initialize(), move_inverted(), print_G(), print_graph_stats(), pull_algorithm(), push_to_network(), replace(), simulation(), και validate_agents_cache().

C.2.0.26 float per_cache

Το ποσοστό των agent που υπάρχουν στην cache.Το K προκύπτει από αυτό.

Ορισμός στη γραμμή 32 του αρχείου main.c.

Αναφορά από initialize(), και set_simulation_parameters().

C.2.0.27 float percentage_of_popular_agents

Το ποσοστό των agent που είναι δημοφηλείς.

Ορισμός στη γραμμή 33 του αρχείου main.c.

Αναφορά από initialize(), και set_simulation_parameters().

C.2.0.28 float popularity

Το πόσο δημοφηλείς είναι οι δημοφηλείς agents.

Π.Χ. αν popularity = 2, τους ξέρουν δύο φορές περισσότεροι agents κατα μέσο όρο Ορισμός στη γραμμή 34 του αρχείου main.c.

Αναφορά από initialize(), και set_simulation_parameters().

C.3 Συναρτήσεις στατιστικών

Οι συναρτήσεις αυτές χρησιμοποιούνται κατα την διάρκεια της εξομοίωσης για την εξαγωγή και φύλαξη αποτελεσμάτων αλλά και στο τέλος αυτής για και τον υπολογισμό χρήσιμων στατιστικών στοιχείων . Τα στατιστικά αυτά γράφονται σε comma seperated files (.csv) που είναι κατάλληλα για προγράμματα οπτικοποίησης όπως είναι το gnuplot και το microsoft excel.

Κατά την διάρκεια της εξομοίωσης, κρατούμε στατιστικά που αφορούν:

- τον φόρτο μηνυμάτων που προκαλείται από τους αλγορίθμους push και pull ξεχωριστά
- το ποσοστό της cache που είναι έγκυρο σε κάθε βήμα
- το ελάχιστο, μέσο και μέγιστο μέγεθος των καταλόγων snooping και inverted cache
- το πόσο γρήγορα (σε πόσα hops) φτάνει η ενημέρωση κατά μέσο όρο κατά το push
- πόσοι πράκτορες μαθαίνουν σε κάθε hop την ενημέρωση (σε πόσους agents φτάνει το μήνυμα push σε κάθε βήμα του δέντρο (σχήμα 3.2)

Συναρτήσεις

- float **final_consistency** (int turn)

  Υπολογίζει το ποσοστό των έγκυρων cache entries.

- void **initialize_stats** ()

  Αρχικοποιεί τις δομές και τα αρχεία που κρατούν τα στατιστικά στην αρχή.

- void **stats_initialize_round** ()

  Αρχικοποιεί τις δομές και τα αρχεία που κρατούν τα στατιστικά κάθε φορά που τρέχουμε το simulation για διαφορετική τιμή στο range που κινείται η **val_to_test**(σελ. 158).

- void **stats_finalize_round** ()

  Στο τέλος κάθε simulation (τέλος των **turns**(σελ. 157)*) γράφει τα στατιστικά στα αρχεία.*

Μεταβλητές

- int **messages**

  Κρατά τον αριθμό μηνυμάτων του push μόνο.

- int **pull_messages**

  Κρατά τον αριθμό μηνυμάτων του pull μόνο.

- int **pull_level_found**

  Εδώ αποθηκεύεται το βάθος (στο path) στο οποίο βρέθηκε η πληροφορία κατά το pull.

- int **rep_messages**

  Μηνύματα που χρειάστηκαν για cache replacement.

- int **moves**

  Αριθμός μετακινήσεων που έγιναν κατα την διάρκεια του simulation.

- int **pulls**

  Αριθμός pull που έγιναν κατα την διάρκεια του simulation.

- int **av_dir_size**

  Μέσο μέγεθος snooping directory.

- int **agents_inf_at_each_level** [max_flood_level]

  Κρατάει στατιστικά για το πόσοι agents ενημερώθηκαν σε κάθε βάθος του push.

- int **pull_found_at_each_level** [max_flood_level]

  Κρατάει στατιστικά για το πόσοι agents ερωτήθηκαν σε κάθε βάθος του pull.

- int **pulls_performed**

  Ο αριθμός των pulls που έγιναν κατα την διάρκεια του simulation.

- FILE ∗ **f_cache**

Αρχείο αποθήκευσης στατιστικών: ποσοστό valid cache.

- FILE * **f_mes**

  Αρχείο αποθήκευσης στατιστικών: αριθμός μηνημάτων.

- FILE * **f_dir_size**

  Αρχείο αποθήκευσης στατιστικών: μέσο μέγεθος καταλόγου snooping.

- FILE * **f_levels**

  Αρχείο αποθήκευσης στατιστικών: πόσα pulls απαντήθηκαν σε κάθε level ή και δεν βρέθηκε σωστή απάντηση.

Τεκμηρίωση Συναρτήσεων

C.3.0.29 float final_consistency (int turn)

Υπολογίζει το ποσοστό των έγκυρων cache entries.

Σαρώνει όλους τους agents βρίσκει το ποσοστό των cache entries που είναι έγκυρο. Στο τέλος αποθηκεύει το αποτέλεσμα στο .csv file

Παράμετροι:

turn Το turn στο οποίο κλήθηκε η συνάρτηση.

Επιστρέφει:

Το ποσοστό που υπολογίστηκε.

Ορισμός στη γραμμή 28 του αρχείου statistics.c.

References agents, agent_t::cache, f_cache, K, agent_t::known_location, agent_t::location, και num_agents.

Αναφορά από simulation().

```
29 {
30     int i, k;
31     int location;
32
33     int temp_inf = 0, temp_not_inf = 0;
34
35     for (i = 0; i < num_agents; i++) {
36         for (k = 0; k < K; k++)
37             if (agents[(agents[i].cache[k])].location ==
38                 agents[i].known_location[k]) {
39
40                 temp_inf++;
41             } else {
42                 temp_not_inf++;
43             }
44     }
45
46
47     fprintf(f_cache, "%f;",
48             100 * (float) temp_inf / (temp_not_inf + temp_inf));
49     return 100 * (float) temp_inf / (temp_not_inf + temp_inf);
50 }
```

C.3.0.30 void initialize_stats ()

Αρχικοποιεί τις δομές και τα αρχεία που κρατούν τα στατιστικά στην αρχή.

Ανοίγει τα αρχεία, τοποθετεί σε αυτά επικεφαλίδες (που θα φανούν και στους γράφους όταν ανοιχθούν απο τό Excel). Ορισμός στη γραμμή 60 του αρχείου statistics.c.

References f_cache, f_dir_size, f_levels, f_mes, maximum_pull_level, και turns.

Αναφορά από batch_run().

```
61 {
62     int i;
63
64     f_cache = fopen("informed.csv", "w");
65     f_mes = fopen("messages.csv", "w");
66     f_dir_size = fopen("dir_size.csv", "w");
```

```
    f_levels = fopen("inf_per_level.csv", "w");

    // initialize stats cvf file (to open with ms excel)
    fprintf(f_cache, ";");
    for (i = 0; i < turns; i += 20)
        fprintf(f_cache, "%d;", i);
    fprintf(f_cache, "\n");


    fprintf(f_levels, ";not_found;");
    for (i = 0; i < maximum_pull_level + 1; i++)
        fprintf(f_levels, "%d;", i);
    fprintf(f_levels, "\n");

    fprintf(f_mes, " ;push;pull\n");

    printf("value to test\t"
           "|perc\t" "|cons\t\t" "|av dir\t" "|ps_mes\t" "|pl_mes\t");
    printf
        ("\n-------------------------------------------------------------------");
}
```

C.3.0.31 void stats_finalize_round ()

Στο τέλος κάθε simulation (τέλος των **turns**(σελ. 157)) γράφει τα στατιστικά στα αρχεία.

Ουσιαστικά γράφουμε

- Τον συνολικό αριθμό μηνυμάτων
- Τα μηνύματα λόγω pull
- Τα μηνύματα λόγω push
- Το μέσο μέγεθος των directories
- Σε κάθε level πόσοι agents ενημερώθηκαν
- Διάφορα άλλα finalizations σε άλλα αρχεία (πχ \n)

Ορισμός στη γραμμή 132 του αρχείου statistics.c.

References av_dir_size, f_cache, f_dir_size, f_levels, f_mes, maximum_pull_level, messages, moves, pull_found_at_each_level, pull_messages, pulls_performed, rep_messages, και val_to_test.

Αναφορά από batch_run().

```
133 {
134     int i;
135     float sum_of_agents_informed = 0.0;
136
137     fprintf(f_cache, "\n");     // change line to the stats file
138     moves = 1;
139     fprintf(f_mes, "%f;%d;%d;%d\n", *val_to_test, (messages) / moves,
140             (pull_messages) / moves, (rep_messages) / moves);
141     fprintf(f_dir_size, "%f;%d\n", *val_to_test, av_dir_size);
142
143
144     for (i = 0; i < maximum_pull_level + 2; i++) {
145         sum_of_agents_informed =
146             (float) pull_found_at_each_level[i] / pulls_performed;
147         fprintf(f_levels, "%f;", sum_of_agents_informed);
148         printf("\n %d %f\n", i, sum_of_agents_informed);
149     }
150     fprintf(f_levels, "\n");
151
152 }
```

C.3.0.32 void stats_initialize_round ()

Αρχικοποιεί τις δομές και τα αρχεία που κρατούν τα στατιστικά κάθε φορά που τρέχουμε το simulation για διαφορετική τιμή στο range που κινείται η **val_to_test**(σελ. 158).

Μηδενίζει κάποιες μεταβλητές (ώστε να τις προετοιμάσει για τον επόμενο γύρο) και γράφει στα .csv τις επικεφαλίδες του γύρου (Πχ την τιμή της μεταβλητής **val_to_test**(σελ. 158) που θα εξεταστεί). Ορισμός στη γραμμή 97 του αρχείου statistics.c.

References agents_inf_at_each_level, f_cache, f_levels, maximum_level, maximum_pull_level, messages, pull_found_at_each_level, pull_messages, pulls_performed, rep_messages, και val_to_test.

Αναφορά από batch_run().

```
{
    int i;

     for (i = 0; i < maximum_level + 1; i++) {
         agents_inf_at_each_level[i] = 0;
     }
     for (i = 0; i < maximum_pull_level + 2; i++) {
         pull_found_at_each_level[i] = 0;
     }

     pulls_performed = 0;

     messages = 0;
     pull_messages = 0;
     rep_messages = 0;
     fprintf(f_cache, "%f;", *val_to_test);
     fprintf(f_levels, "%f;", *val_to_test);
}
```

Τεκμηρίωση Μεταβλητών

C.3.0.33 int agents_inf_at_each_level[max_flood_level]

Κρατάει στατιστικά για το πόσοι agents ενημερώθηκαν σε κάθε βάθος του push.

Πρόκειται για πίνακα όπου, στην θέση 1 έχει το πόσοι agents ενημερώθηκαν σε TTL=1, στην θέση 2 το πόσοι ενημερώθηκαν σε βάθος 2 κτλ. Το άθροισμα όλων αυτών των θέσεων είναι ο αριθμός των agent που ενημερώθηκαν με το push συνολικά Ορισμός στη γραμμή 51 του αρχείου main.c.

Αναφορά από push_to_network(), και stats_initialize_round().

C.3.0.34 int av_dir_size

Μέσο μέγεθος snooping directory.

Ορισμός στη γραμμή 50 του αρχείου main.c.

Αναφορά από simulation(), και stats_finalize_round().

C.3.0.35 FILE* f_cache

Αρχείο αποθήκευσης στατιστικών: ποσοστό valid cache.

Ορισμός στη γραμμή 56 του αρχείου main.c.

Αναφορά από final_consistency(), initialize_stats(), stats_finalize_round(), και stats_-initialize_round().

C.3.0.36 FILE * f_dir_size

Αρχείο αποθήκευσης στατιστικών: μέσο μέγεθος καταλόγου snooping.

Ορισμός στη γραμμή 56 του αρχείου main.c.

Αναφορά από initialize_stats(), και stats_finalize_round().

C.3.0.37 FILE * f_levels

Αρχείο αποθήκευσης στατιστικών: πόσα pulls απαντήθηκαν σε κάθε level ή και δεν βρέθηκε σωστή απάντηση.

Ορισμός στη γραμμή 56 του αρχείου main.c.

Αναφορά από initialize_stats(), stats_finalize_round(), και stats_initialize_round().

C.3.0.38 FILE * f_mes

Αρχείο αποθήκευσης στατιστικών: αριθμός μηνημάτων.

Ορισμός στη γραμμή 56 του αρχείου main.c.

Αναφορά από initialize_stats(), και stats_finalize_round().

C.3.0.39 int messages

Κρατά τον αριθμό μηνυμάτων του push μόνο.

Ορισμός στη γραμμή 44 του αρχείου main.c.

Αναφορά από move_inverted(), push_to_network(), simulation(), stats_finalize_round(), και stats_initialize_round().

C.3.0.40 int moves

Αριθμός μετακινήσεων που έγιναν κατα την διάρκεια του simulation.

Ορισμός στη γραμμή 48 του αρχείου main.c.

Αναφορά από simulation(), και stats_finalize_round().

C.3.0.41 int pull_found_at_each_level[max_flood_level]

Κρατάει στατιστικά για το πόσοι agents ερωτήθηκαν σε κάθε βάθος του pull.

Πρόκειται για πίνακα όπου, στην θέση 1 έχει το πόσοι agents ρωτήθηκαν σε TTL=1, στην θέση 2 το πόσοι σε βάθος 2 κτλ. Το άθροισμα όλων αυτών των θέσεων είναι ο αριθμός των agent που ρωτήθηκαν κατα την διάρκεια του pull συνολικά.

Ορισμός στη γραμμή 52 του αρχείου main.c.

Αναφορά από pull_algorithm(), stats_finalize_round(), και stats_initialize_round().

C.3.0.42 int pull_level_found

Εδώ αποθηκεύεται το βάθος (στο path) στο οποίο βρέθηκε η πληροφορία κατά το pull.

Ορισμός στη γραμμή 46 του αρχείου main.c.

Αναφορά από initialize(), pull_algorithm(), και simulation().

C.3.0.43 int pull_messages

Κρατά τον αριθμό μηνυμάτων του pull μόνο.

Ορισμός στη γραμμή 45 του αρχείου main.c.

Αναφορά από pull_algorithm(), simulation(), stats_finalize_round(), stats_initialize_round(), και validate_agents_cache().

C.3.0.44 int pulls

Αριθμός pull που έγιναν κατα την διάρκεια του simulation.

Ορισμός στη γραμμή 49 του αρχείου main.c.

C.3.0.45 int pulls_performed

Ο αριθμός των pulls που έγιναν κατα την διάρκεια του simulation.

Ορισμός στη γραμμή 53 του αρχείου main.c.

Αναφορά από pull_algorithm(), stats_finalize_round(), και stats_initialize_round().

C.3.0.46 int rep_messages

Μηνύματα που χρειάστηκαν για cache replacement.

Ορισμός στη γραμμή 47 του αρχείου main.c.

Αναφορά από replace(), stats_finalize_round(), και stats_initialize_round().

C.4 Συντονισμός και αυτοματοποίηση της εξομοίωσης.

Δημιουργήθηκε μία μηχανή αυτόματης εκτέλεσης ώστε να εξάγουμε στατιστικά χωρίς πολλές παρεμβάσεις από τον χρήστη. Έτσι δημιουργήθηκαν συναρτήσεις που αφορούν τον συντονισμό του simulation ώστε να τρέχει με αυτοματοποιημένο τρόπο. Ποιο συγκεκριμένα χρησιμοποιείται για

- Τον αυτόματο υπολογισμό αποτελεσμάτων για διαφορετικές τιμές κάποιων μεταβλητών. Έτσι πχ δίνεται η μεταβλητή που θέλουμε να μεταβάλουμε, το μέγιστο/ελάχιστο/βήμα της μεταβολής και αυτόματα τρέχει ένα simulation για κάθε τιμή που ανήκει σε αυτό το διάστημα. Μάλλιστα τα αποτελέσματα γίνονται plot στο ίδιο αρχείο (σε διαφορετικές καμπύλες).
- Την αυτόματη λήψη μέσου όρου πολλαπλών εκτελέσεων του ίδιου simulation.

Συναρτήσεις

- void **set_simulation_parameters** ()

  Στην συνάρτηση αυτή δηλώνουμε τις τιμές που θέλουμε να θέσουμε σε κάθε παράμετρο για το simulation.

- void **simulation** ()

  Τρέχει το simulation για έναν αριθμό απο turns κρατώντας στατιστικά Πρόκειται για το loop που ξεκινά από τον αρχικοποιημένο γράφο και τρέχει το πείραμα για έναν αριθμό από **turns**(σελ. 157).

- int **batch_run** ()

  Καλεί την **simulation()**(σελ. 153) *για κάθε τιμή του range της παραμέτρου* **val_to_test**(σελ. 158) *που θέλουμε να εξετάσουμε.*

Μεταβλητές

- float * **val_to_test**

Στον δείκτη αυτό βάζουμε την παράμετρο που θέλουμε να μελετήσουμε.

- float **l_bound**

  Η αρχική τιμή της παραμέτρου, με αυτή θα ξεκινήσει το simulation **val_to_test**(σελ. 158).

- float **up_bound**

  Όταν φτάσουμε σε αυτή την τιμή σταματάμε το simulation **val_to_test**(σελ. 158)

- float **incr**

  Η παράμετρος αυξάνεται κατά τόσο σε κάθε κύκλο απο turns **val_to_test**(σελ. 158).

- int **turns**

  Ο αριθμός των turns του simulation.

Λεπτομερής Περιγραφή

Συναρτήσεις και παράμετροι που αφορούν τον συντονισμό του simulation ώστε να τρέχει με αυτοματοποιημένο τρόπο.

Για την εξομοίωση ο χρήστης ορίζει στον simulator ποιά **παράμετρο**(σελ. 121) θέλει να μελετήσει (**val_to_test**(σελ. 158)) και για ποιές τιμές (**l_bound**(σελ. 157), **up_bound**(σελ. 157), **incr**(σελ. 157)). Από εκεί και πέρα αυτόματα για κάθε τιμή της παραμέτρου τρέχει το simulation για έναν αριθμό από **turns**(σελ. 157). Μόλις τελειώσουν τα turns, αποθηκεύονται τα αποτελέσματα, αυξάνεται η παράμετρος κατα **incr**(σελ. 157) και τρέχει για την επόμενη τιμή κ.ο.κ.

Τεκμηρίωση Συναρτήσεων

C.4.0.47 int batch_run ()

Καλεί την **simulation()**(σελ. 153) για κάθε τιμή του range της παραμέτρου **val_to_test**(σελ. 158) που θέλουμε να εξετάσουμε.

Στην συνάρτηση αυτή η μεταβλητή **val_to_test**(σελ. 158) αυξάνεται η μειώνεται σε κάθε loop και καλείται η **simulation()**(σελ. 153) για να εξωμειώσουμε το σύστημα με αυτή την τιμή.

Επιπλέον γράφει και η συνάρτηση αυτή στα αρχεία των στατιστικών (αλλάζει γραμμή στο comma seperated excel file και στην πρώτη στήλη γράφει επικεφαλίδες με την τιμή που εξετάζουμε)

Κοιτάξτε επίσης :

simulation(σελ. 153) **set_simulation_parameters**(σελ. 151) **val_to_test**(σελ. 158) **up_bound**(σελ. 157) **l_bound**(σελ. 157) **incr**(σελ. 157)

Ορισμός στη γραμμή 236 του αρχείου simulation.c.

References destroy_graph(), incr, initialize(), initialize_stats(), l_bound, print_graph_stats(), simulation(), stats_finalize_round(), stats_initialize_round(), up_bound, και val_to_test.

```
237 {
238     int i;
239     initialize_stats();
240
241     for (*val_to_test = l_bound; *val_to_test <= up_bound;
242          *val_to_test += incr) {
243         /*for (i = 0; i < 4; i ++){
244            switch (i){
245            case 3: decay_par = 0.0;
246            maximum_level = 5;
247            break;
248            case 2: decay_par = 0.04;
249            maximum_level = 5;
250            break;
251            case 1: decay_par = 0.1;
252            maximum_level = 5;
253            break;
254            case 0: decay_par = 0.6;
255            maximum_level = 4;
256            break;
257            }
258            val_to_test = &decay_par;
259
```

```
         */
        printf("\n");
        initialize();           //initialize graph

        stats_initialize_round();
        print_graph_stats();
        /****************************/
        simulation();
        /****************************/
        stats_finalize_round();
        destroy_graph();        //free memory



    }



}
```

C.4.0.48 void set_simulation_parameters ()

Στην συνάρτηση αυτή δηλώνουμε τις τιμές που θέλουμε να θέσουμε σε κάθε παράμετρο για το simulation.

Ο χρήστης δίνει τις τιμές για τις παραμέτρους που θα μείνουν σταθερές (π.χ. τον αριθμό των **turns**(σελ. 157), το μέγεθος του **K**(σελ. 135), το TTL κτλ...)

Επίσης θέτοντας κάποιες παραμέτρους επιλέγει το είδος του αλγορίθμου που θα τρέξει (**use_replication**(σελ. 126), **do_pull**(σελ. 124), **algorithm**(σελ. 123) κτλ...)

Τέλος σε κάθε simulation μπορεί αυτόματα να εξεταστεί και ένα range τιμών κάποιας παραμέτρου. Η παράμετρος που θα εξεταστεί δηλώνεται στο στην μεταβλητή **val_to_test**(σελ. 158) και το range χρησιμοποιώντας τις μεταβλητές **l_bound**(σελ. 157), **up_bound**(σελ. 157), **incr**(σελ. 157). Ορισμός στη γραμμή 28 του αρχείου simulation.c.

References algorithm, decay_par, do_pull, exp_time, incr, inv_cache_percentage, l_bound, LEASING, maximum_level, maximum_pull_level, p_move, p_pull, p_replicate, per_cache, percentage_of_popular_agents, popularity, pull_decay_par, replace_prob, replica_time, turns, up_bound, use_replication, και val_to_test.

```
{

    /* Number of agents = 1000 */
    /* graph values */
    per_cache = 0.004;
    percentage_of_popular_agents = 0.01;
    popularity = 10.0;          /* three times more agents know them */

    /* simulation values */
    p_move = 0.005;             /* probability to move (each agent at each round) */
    p_pull = 0.01;

    turns = 250;                /* number of simulated turns */


    /* default flooding values */
    maximum_level = 5;
    decay_par = 0.4;

    do_pull = 0;
    /* pulling values */
    maximum_pull_level = 5;     /* look in my 2 hop neighborhood */
    pull_decay_par = 0.5;       /* with no decay */


    /* replication parameters */
    use_replication = 0;
    replica_time = 6;
    p_replicate = 0.3;

    /* keep cache for moving agents (algorithm 3) */

    exp_time = 20;              /* TTL for the snoop cache */


    replace_prob = 0.5;

    //LEASING
    inv_cache_percentage = 0.8;
```

```

    // ALGORITHM SELECTION
    //PLAIN_PUSH 1
    //SNOOP      2
    //LEASING    3

    algorithm = LEASING;

    // VALUE TO TEST
    val_to_test = &inv_cache_percentage;
    l_bound = 0.5;
    up_bound = 1.0;
    incr = 0.1;

}
```

C.4.0.49 void simulation ()

Τρέχει το simulation για έναν αριθμό απο turns κρατώντας στατιστικά. Πρόκειται για το loop που ξεκινά από τον αρχικοποιημένο γράφο και τρέχει το πείραμα για έναν αριθμό από **turns**(σελ. 157).

Σε κάθε γύρο αρχικοποιεί τις μεταβλητές που κρατάν τα στατιστικά, και για κάθε turn εκτελέι τις κατάλληλες ενέργειες (ανάλογα με τις παραμέτρους και τον αλγόριθμο που χρησιμοποιείται). Για παράδειγμα αν έχει επιλεχθεί snooping κοιτά αν έχουν λήξει snooping directory entries στον τρέχων γύρο, επιλέγει ποιοι agents θα μετακινηθούν, ποιοί θα κάνουν replace μέρη της cache τους κτλ...

Στο τέλος κάθε γύρου γράφει τα αποτελέσματα σε αρχεία που κρατάν τα στατιστικά και τυπώνει στην οθόνη μηνύματα για να γνωρίζουμε το status του simulation σε πραγματικό χρόνο.

use_replication Ορισμός στη γραμμή 98 του αρχείου simulation.c.

References flood_cache_t::agent, agents, algorithm, av_dir_size, agent_t::cache, delete_-expired_cache(), delete_old_replicas(), agent_t::f_cache_size, final_consistency(), K, agent_-t::known_location, LEASING, levels, agent_t::location, flood_cache_t::location, maximum_-level, messages, move_inverted(), moves, num_agents, p_move, p_pull, pull_algorithm(), pull_level_found, pull_messages, push_to_network(), ran(), replace(), replace_prob, agent_t::replica_head, SNOOP, turns, use_replication, val_to_test, και validate_expired_cache().

Αναφορά από batch_run().

```
{
    int i, j, agnt, k;
    float cons;
    int max = 0;
    int av_dir = 0;
    int dest;
    int max_dir = 0;
    levels = (int *) malloc((maximum_level + 1) * sizeof(int));
    for (i = 0; i < maximum_level + 1; i++)
        levels[i] = 0;
    messages = 0;
    pull_messages = 0;
    moves = 0;
    av_dir = 0;
    for (k = 0; k < turns; k++) {
        if (use_replication)
            delete_old_replicas(k);
        if (algorithm == SNOOP) {
            validate_expired_cache(k);
            delete_expired_cache(k);
            for (i = 0; i < num_agents; i++) {
                av_dir += agents[i].f_cache_size;
                if (agents[i].f_cache_size > max_dir)
                    max_dir = agents[i].f_cache_size;
            }
        }
        /*
```

```
           if (algorithm == PLAIN_PUSH){
           validate_expired_cache(k);
           }
         */

        if (algorithm == LEASING) {
            for (i = 0; i < num_agents; i++) {
                av_dir += agents[i].f_cache_size;

                if (ran(replace_prob))
                    replace();
            }
            validate_expired_cache(k);
        }

        for (agnt = 0; agnt < num_agents; agnt++) {
            if (ran(p_move)) {
                if (algorithm == LEASING)
                    move_inverted(agnt);
                else
                    push_to_network(agnt, k);
                moves++;
            }

            if (ran(p_pull)) {

                if (use_replication) {
                    int k_loc = 0;
                    /* pull to update replica */

                    /* pick a resource to pull */

                    /* temp just pull an existing old replica */
                    if (agents[agnt].replica_head != NULL) {
                        dest = agents[agnt].replica_head->agent;
                        k_loc = agents[agnt].replica_head->location;
                    } else {
                        dest = rand() % num_agents;
                        k_loc = 0;
                    }
```

```
                pull_level_found = -1;
                pull_algorithm(agnt, k_loc, dest, k);

            } else {
                /* pull to update cached locations */
                for (i = 0; i < K; i++) {
                    dest = agents[agnt].cache[i];
                    if (agents[agnt].known_location[i] !=
                        agents[dest].location) {
                        pull_level_found = -1;
                        agents[agnt].known_location[i] =
                            pull_algorithm(agnt,
                                           agents[agnt].
                                           known_location[i], dest, k);
                        break;
                    }
                }
            }
        }
    }


    /* gother stats */
    if (1 /*!use_replication */ ) {
        if (k % 20 == 0)
            cons = final_consistency(k);
        // keep_agents_informed_at_each_level(k);

        /* print stats every 10 rounds to the screen */
        if ((k != 0 && (k % 10) == 0) || k == turns - 1)
            printf("\rvalue %6.4f:\t"
                   "|%d%%:\t"
                   "|%f%%\t"
                   "|%d\t"
                   "|%d\t"
                   "|%d\t", *val_to_test, (k + 1) * 100 / turns, cons,
                   av_dir / (num_agents * k)
                   , (messages) / turns, (pull_messages / turns)

                );
    }

}
```

```
    av_dir_size = av_dir / (num_agents * turns);
    printf("MAX DIR = %d\n", max_dir);
}
```

Τεκμηρίωση Μεταβλητών

C.4.0.50 float incr

Η παράμετρος αυξάνεται κατά τόσο σε κάθε κύκλο απο turns **val_to_test**(σελ. 158).

Ορισμός στη γραμμή 78 του αρχείου main.c.

Αναφορά από batch_run(), και set_simulation_parameters().

C.4.0.51 float l_bound

Η αρχική τιμή της παραμέτρου, με αυτή θα ξεκινήσει το simulation **val_to_test**(σελ. 158).

Ορισμός στη γραμμή 76 του αρχείου main.c.

Αναφορά από batch_run(), και set_simulation_parameters().

C.4.0.52 int turns

Ο αριθμός των turns του simulation.

Ορισμός στη γραμμή 79 του αρχείου main.c.

Αναφορά από initialize_stats(), set_simulation_parameters(), και simulation().

C.4.0.53 float up_bound

Όταν φτάσουμε σε αυτή την τιμή σταματάμε το simulation **val_to_test**(σελ. 158)

Ορισμός στη γραμμή 77 του αρχείου main.c.

Αναφορά από batch_run(), και set_simulation_parameters().

C.4.0.54 float* val_to_test

Στον δείκτη αυτό βάζουμε την παράμετρο που θέλουμε να μελετήσουμε.

Κατά την διάρκεια του simulation αυτή η παράμετρος θα αυξάνεται και θα μειώνεται αυτόματα ανάλογα με το **l_bound**(σελ. 157), **up_bound**(σελ. 157), **incr**(σελ. 157) Ορισμός στη γραμμή 75 του αρχείου main.c.

Αναφορά από batch_run(), set_simulation_parameters(), simulation(), stats_finalize_round(), και stats_initialize_round().

C.5 Συναρτήσεις που αφορούν την εξομοίωση των αλγορίθμων.

Οι συναρτήσεις αυτές ουσιαστικά χρησιμεύουν άμεσα για την εξομοίωση των αλγορίθμων push, pull, push with snooping directories, inverted cache push/pull κτλ.

int f_no; int f_agents[num_agents][2]; int f_pull_no; int f_pull_agents[num_agents][2];

Συναρτήσεις

- void **move_inverted** (int **agent**)

  Μετακινεί έναν δοσμένο agent σε νέα θέση.

- void **replace** ()

  Αντικαθιστά ένα entrie στην cache κάποιου agent.

- void **clear_flood_marks** ()

  Καθαρίζει τα σημάδια που βάζουμε κατά την διάρκεια του αλγόριθμου pull.

- int **pull_algorithm** (int sour_agent, int known_location, int dest_agent, int turn)

  Ο αλγόριθμος pull.

- void **clear_marks** ()

  Καθαρίζει τα σημάδια που βάζουμε κατά την διάρκεια του αλγόριθμου push.

- void **push_to_network** (int **agent**, int turn)

  Ο αλγόριθμος pull.

- void **delete_old_replicas** (int cur_round)

  Σβήνει τα παλιά replicas.

- void **create_replica** (int to_agent, int expr, int **agent**, int location)

  Φτιάχνει replica μιας πληροφορίας σε κάποιον agent.

- void **delete_expired_cache** (int cur_round)

  Σβήνει τα expired snooping cache entries.

- void **add_to_cache** (int to_agent, int expr, int **agent**, int location)

  Φτιάχνει ενα snoopig entrie σε κάποιον agent.

- void **validate_agents_cache** (int **agent**, int cur_turn)

  Εκτελεί το periodic pull του αλγορίθμου snooping.

- void **validate_expired_cache** (int cur_turn)

  Ενημερώνει την cache όλων των agent.

Λεπτομερής Περιγραφή

int f_no; int f_agents[num_agents][2]; int f_pull_no; int f_pull_agents[num_agents][2];

float inv_cache_percentage;/**< Το ποσοστό της inverted cache που κρατάμε (σχετίζεται με το leasing time)

Τεκμηρίωση Συναρτήσεων

C.5.0.55 void add_to_cache (int to_agent, int expr, int agent, int location)

Φτιάχνει ενα snoopig entrie σε κάποιον agent.

Παράμετροι:

to_agent σε ποιόν agent θα φτιαχτεί το snooping entrie.

expr σε ποιο γύρο θα λήξει

agent σε ποιον agent ανήκει η πληροφορία που θα αντιγράψουμε

location η θέση (η το version) αυτή την στιγμή.

Η συνάρτηση αυτή καλείται από το push και βάζει στους κόμβους που συνάντησε ένα snooping entry.Ορισμός στη γραμμή 49 του αρχείου snooping_funcs.c.

References flood_cache_t::agent, agent, agents, flood_cache_t::expire, agent_t::f_cache_head, agent_t::f_cache_size, agent_t::f_cache_tail, flood_cache, flood_cache_t::location, και flood_cache_t::next.

Αναφορά από initialize(), και push_to_network().

```
50 {
51     int i = to_agent;
52     flood_cache *p;
53
54     p = (flood_cache *) malloc(sizeof(flood_cache));
55     p->agent = agent;
56     p->expire = expr;
57     p->location = location;
58     p->next = NULL;
59
60     i = to_agent;
61     if (agents[i].f_cache_tail == NULL)
62         agents[i].f_cache_tail = agents[i].f_cache_head = p;
63     else {
64         agents[i].f_cache_tail->next = p;
65         agents[i].f_cache_tail = p;
66     }
67
68     agents[i].f_cache_size++;
69
70 }
```

C.5.0.56 void clear_flood_marks ()

Καθαρίζει τα σημάδια που βάζουμε κατά την διάρκεια του αλγόριθμου pull.

Κατά τον αλγόριθμο pull, κάθε agent που λαμβάνει το μήνυμα μαρκάρεται για να μην επαναπροωθήσει το μήνυμα μελλοντικά. Πριν κάποιο pull καθαρίζουμε τα σημάδια που βάλαμε κατά την διάρκεια του προηγούμενου pull. Ορισμός στη γραμμή 14 του αρχείου pull.c.

References agents, agent_t::flood_marks, και num_agents.

Αναφορά από pull_algorithm().

```
{
    int i;
    for (i = 0; i < num_agents; i++)
        agents[i].flood_marks = -1;

}
```

C.5.0.57 void clear_marks ()

Καθαρίζει τα σημάδια που βάζουμε κατά την διάρκεια του αλγόριθμου push.

Κάθε agent που λαμβάνει το μήνυμα μαρκάρεται για να μην επανα-προωθήσει το μήνυμα μελλοντικά. Πριν κάποιο push καθαρίζουμε τα σημάδια που βάλαμε κατά την διάρκεια του προηγούμενου push. Ορισμός στη γραμμή 17 του αρχείου push.c.

References agents, agent_t::marked, και num_agents.

Αναφορά από push_to_network().

```
{
    int i;
    for (i = 0; i < num_agents; i++)
        agents[i].marked = -1;

}
```

C.5.0.58 void create_replica (int to_agent, int expr, int agent, int location)

Φτιάχνει replica μιας πληροφορίας σε κάποιον agent.

Παράμετροι:

to_agent σε ποιόν agent θα φτιαχτεί η replica.

expr σε ποιο γύρο θα λήξει η replica

agent σε ποιον agent ανήκει η πληροφορία που θα αντιγράψουμε

location η θέση (η το version) αυτή την στιγμή.

Φτιάχνουμε στον to_agent μια replica της πληροφορίας που κρατάει ο agent. Η replica θα έχει version number location. Στο μέλλον, ενδέχεται ο agent να αυξήσει το location αλλά στην replica θα έχουμε ακόμα το παλιό version. Ορισμός στη γραμμή 49 του αρχείου replication.c.

References flood_cache_t::agent, agent, agents, flood_cache_t::expire, flood_cache, flood_cache_t::location, flood_cache_t::next, agent_t::replica_head, agent_t::replica_size, και agent_t::replica_tail.

Αναφορά από pull_algorithm().

```
50 {
51     int i = to_agent;
52     flood_cache *p;
53 
54     p = (flood_cache *) malloc(sizeof(flood_cache));
55     p->agent = agent;
56     p->expire = expr;
57     p->location = location;
58     p->next = NULL;
59 
60     i = to_agent;
61     if (agents[i].replica_tail == NULL)
62         agents[i].replica_tail = agents[i].replica_head = p;
63     else {
64         agents[i].replica_tail->next = p;
65         agents[i].replica_tail = p;
66     }
67 
68     agents[i].replica_size++;
69 
70 }
```

C.5.0.59 void delete_expired_cache (int cur_round)

Σβήνει τα expired snooping cache entries.

Παράμετροι:

cur_round ο τρέχων γύρος. Σαρώνει τους agents και σβήνει τα snooping cache entries που έχουν λήξει.

Ορισμός στη γραμμή 15 του αρχείου snooping_funcs.c.

References agents, flood_cache_t::expire, agent_t::f_cache_head, agent_t::f_cache_size, agent_t::f_cache_tail, flood_cache, flood_cache_t::next, και num_agents.

Αναφορά από simulation().

```
{
    int i;
    flood_cache *tmp;
    // print_lists(cur_round);
    //  system("pause");

    for (i = 0; i < num_agents; i++) {
        while (agents[i].f_cache_head != NULL
               && agents[i].f_cache_head->expire <= cur_round) {
            agents[i].f_cache_size--;

            tmp = agents[i].f_cache_head;
            if (agents[i].f_cache_head == agents[i].f_cache_tail) {
                agents[i].f_cache_head = NULL;
                agents[i].f_cache_tail = NULL;
            } else
                agents[i].f_cache_head = agents[i].f_cache_head->next;

            free(tmp);
        }
    }
}
```

C.5.0.60 void delete_old_replicas (int cur_round)

Σβήνει τα παλιά replicas.

Παράμετροι:

cur_round ο τρέχων γύρος. Σαρώνει τους agents και σβήνει τα replicas που έχουν λήξει.

Ορισμός στη γραμμή 16 του αρχείου replication.c.

References agents, flood_cache_t::expire, flood_cache, flood_cache_t::next, num_agents, agent_t::replica_head, agent_t::replica_size, και agent_t::replica_tail.

Αναφορά από simulation().

```
{
    int i;
    flood_cache *tmp;
    // print_lists(cur_round);
    //  system("pause");

    for (i = 0; i < num_agents; i++) {
        while (agents[i].replica_head != NULL
               && agents[i].replica_head->expire <= cur_round) {
            agents[i].replica_size--;

            tmp = agents[i].replica_head;
            if (agents[i].replica_head == agents[i].replica_tail) {
                agents[i].replica_head = NULL;
                agents[i].replica_tail = NULL;
            } else
                agents[i].replica_head = agents[i].replica_head->next;

            free(tmp);
        }
    }
}
```

C.5.0.61 void move_inverted (int agent)

Μετακινεί έναν δοσμένο agent σε νέα θέση.

Όταν κληθεί, ενημερώνει όλους τους agents που είναι στο inverted cache για την νέα θέση και μετά μετακινείται σε αυτή αυξάνοντας το **agent_t::location**(σελ. 182). Εναλλακτικά μπορεί να ενημερωθεί μόνο ένα ποσοστό αυτών που καθορίζεται από την μεταβλητή **inv_cache_percentage**(σελ. 205).

Παράμετροι:

agent O agent που θέλουμε να μετακινήσουμε.

Ορισμός στη γραμμή 17 του αρχείου inverted_cache.c.

References flood_cache_t::agent, agent, agents, agent_t::cache, agent_t::f_cache_head, flood_cache, inv_cache_percentage, K, agent_t::known_location, agent_t::location, messages, flood_cache_t::next, και ran().

Αναφορά από simulation().

```
{
    flood_cache *p;
    int dest_agent;
    int i;

    (agents[agent].location)++;

    p = agents[agent].f_cache_head;
    while (p != NULL) {
        if (ran(inv_cache_percentage)) {
            for (i = 0; i < K; i++)
                if (agents[p->agent].cache[i] == agent)
                    agents[p->agent].known_location[i] =
                        agents[agent].location;

            messages += 2;
        }

        p = p->next;
```

```
    }

}
```

C.5.0.62 int pull_algorithm (int sour_agent, int known_location, int dest_agent, int turn)

Ο αλγόριθμος pull.

Παράμετροι:

sour_agent Ο agent που ξεκινά την αναζήτηση

known_location Η θέση που γνώριζε ο sour_agent πριν την αναζήτηση (ψάχνει κάποια πιο καινούργια).

dest_agent Ο agent που ψάχνουμε (στο replication ψάχνουμε κάποια πιο σύγχρονη replica αυτού του agent και όχι τον ίδιο τον agent)

turn Η turn που ξεκίνησε η αναζήτηση.

Επιστρέφει:

Η νέα θέση του agent ή το νέο version number της replicas. Επιστρέφεται πάντα το **μεγαλύτερο** αποτέλεσμα που μας επέστρεψαν τα paths του pull.

Γίνεται flooding στο δίκτυο από τον agent που ξεκινά το pull μέχρι να βρεθεί η πληροφορία που αναζητούμε. Τότε επιστρέφεται το αποτέλεσμα στον αρχικό agent.

Κάθε κόμβος που λαμβάνει το query message εξετάζει

- Το snooping directory του (αν υπάρχει)
- Τις replicas (αν χρησιμοποιούμε replication)
- Την cache του (αν δεν χρησιμοποιούμε replication)

Αν δεν βρεί την αναζητούμενη πληροφορία, πρωθεί το μήνυμα στους γείτονές του λαμβάνοντας υπόψην τις παραμέτρους **maximum_pull_level**(σελ. 125), **pull_decay_par**(σελ. 126) κτλ.

Αν χρησιμοποιούμε replication τότε με πιθανότητα **p_replicate**(σελ. 125) φτιάχνουμε replica της αναζητούμενης πληροφορίας στον τρέχων agent.

Επίσης κρατούνται στατιστικά όπως ο αριθμός μηνυμάτων και το βάθος στο οποίο βρέθηκε η πληροφορία (ή αν δεν βρέθηκε). Ορισμός στη γραμμή 54 του αρχείου pull.c.

References flood_cache_t::agent, agents, algorithm, agent_t::cache, clear_flood_marks(), create_replica(), agent_t::f_cache_head, f_pull_agents, f_pull_no, flood_cache, agent_t::flood_marks, K, agent_t::known_location, agent_t::location, flood_cache_t::location, maximum_pull_level, flood_cache_t::next, p_replicate, pull_decay_par, pull_found_at_each_level, pull_level_found, pull_messages, pulls_performed, ran(), agent_t::replica_head, replica_time, SNOOP, και use_replication.

Αναφορά από initialize(), simulation(), και validate_agents_cache().

```
56 {
57 
58     int i, j, k, l;
59     float prob = 1;
60     int level = 0;
61 
62     int ret_location = known_location;
63     int flood_dest_agent;
64 
65     int flood;
66     int flooded_nodes;
67 
68     int stats_found = 0;
69 
70     flood_cache *p;
71 
72     clear_flood_marks();
73     agents[sour_agent].flood_marks = 0;
74 
75     f_pull_no = 1;
76     f_pull_agents[0][0] = sour_agent;
77 
78     for (level = 0; level <= maximum_pull_level; level++) {
           /* for each flood level */
79         flooded_nodes = 0;
80         for (l = 0; l < f_pull_no; l++) {
```

```
pull_messages++;
i = f_pull_agents[l][level % 2];
flood = 1;




// if we use snooping algorithm, check snoop cache.
if (algorithm == SNOOP) {
    p = agents[i].f_cache_head;
    while (p != NULL) {
        if (p->agent == dest_agent)
            if (p->location > known_location) {
                if (ret_location < p->location)
                    ret_location = p->location;
                flood = 0;
                pull_level_found = level;

                // pull_messages ++;
            }
         p = p->next;
     }
 }

 if (use_replication) {
     p = agents[i].replica_head;
     while (p != NULL) {
         if (p->agent == dest_agent)
             if (p->location > known_location) {
                 if (ret_location < p->location)
                     ret_location = p->location;
                 flood = 0;
                 pull_level_found = level;
                 // pull_messages ++;
            }
         p = p->next;
     }
 }

 //if the agent is in my cache
   return the result and do not forward the message.
 //if we use replication
   the cache is used only as a network infastrucrure.
```

```
if (!use_replication)
    for (j = 0; j < K; j++) {
        if (agents[i].cache[j] == dest_agent)
            if (agents[i].known_location[j] > known_location) {
                if (ret_location < agents[i].known_location[j]) {
                    ret_location = agents[i].known_location[j];
                    flood = 0;
                    pull_level_found = level;
                }
                //   pull_messages ++;

            }
    }


if (i == dest_agent) {
    ret_location = agents[i].location;
    flood = 0;
    pull_level_found = level;
}


/* create replication paths */
if (use_replication)
    /* with a probability create replica to the pulling agent*/
    if (ran(p_replicate) || i == sour_agent)
        create_replica(i, turn + replica_time, dest_agent,
                       agents[dest_agent].location);


// else forward the message (flooding)
if (flood == 1) {
    for (j = 0; j < K; j++)
        if (ran(prob)) {
            /* for each agent in the cache INFORM HIM */
            flood_dest_agent = agents[i].cache[j];
            if (use_replication
                || agents[i].known_location[j] ==
                agents[flood_dest_agent].location)
                /* if the agent is there, we send him a message */
                /* there is a probability to forward the message */
                if (agents[flood_dest_agent].flood_marks == -1) {
                    /* if this agent hasn't received the message yet */
```

```

                                f_pull_agents[flooded_nodes][(level +
                                                              1) % 2] =
                                    flood_dest_agent;
                                flooded_nodes++;
                                agents[flood_dest_agent].flood_marks = 1;
                            } else
                                pull_messages++;
                        // pull_messages++;



                    }
            } else {

                if (stats_found == 0
                    && ret_location == agents[dest_agent].location) {
                    //      printf("ton brika se level %d\n",level);
                    stats_found = level + 1;
                }
            }

        }
        f_pull_no = flooded_nodes;
        prob = prob * (1 - pull_decay_par);
        if (prob <= (float) 1.0 / K)
            prob = (float) 1.0 / K;
    }

    pull_found_at_each_level[stats_found]++;

    pulls_performed++;
    return ret_location;
}
```

C.5.0.63 void push_to_network (int agent, int turn)

Ο αλγόριθμος pull.

Παράμετροι:

agent Ο agent που μετακινείται (αν έχουμε move) ή ο agent που αλλάζει version στο resource του (αν έχουμε replication).

turn Η turn που ξεκίνησε η αναζήτηση.

Κάθε φορά που την καλούμε τότε ο agent αλλάζει θέση (αξάνει το **agent_t::location**(σελ. 182)) και ενημερώνει το δίκτυο χρησιμοποιώντας τον αλγόριθμο push.

Κάθε κόμβος που λαμβάνει το message

- Βάζει την πληροφορία στο snooping directory (αν υπάρχει)
- Αν χρησιμοποιούμε replicas εξετάζουμε αν έχουμε κάποια replica εμείς για να την ανανεώσουμε
- Αν δεν χρησιμοποιούμε replicas, αν έχουμε την θέση του agent στην cache την ανανεώνουμε

Έπειτα. πρωθούμε το μήνυμα στους γείτονές του λαμβάνοντας υπόψην τις παραμέτρους **maximum_level**(σελ. 124), **decay_par**(σελ. 124) κτλ.

Επίσης κρατούνται στατιστικά όπως ο αριθμός μηνυμάτων και ο αριθμώς των agent που ενημερώθηκε σε κάθε βάθος του path. Ορισμός στη γραμμή 53 του αρχείου push.c.

References add_to_cache(), flood_cache_t::agent, agent, agents, agents_inf_at_each_level, algorithm, agent_t::cache, clear_marks(), decay_par, exp_time, f_agents, f_no, flood_cache, K, agent_t::known_location, flood_cache_t::location, agent_t::location, agent_t::marked, maximum_level, messages, flood_cache_t::next, ran(), agent_t::replica_head, SNOOP, και use_replication.

Αναφορά από simulation().

```
54 {
55     int i, j, k, l;
56     float prob = 1;
57     int level = 0;
58     int new_location;
59     int old_location;
60
61     int destination_agent;
```

```
    int retries;

    int old_k_location;

    int flooded_nodes;
    flood_cache *p;

    old_location = agents[agent].location;
    new_location = ++(agents[agent].location);

    clear_marks();
    agents[agent].marked = 0;

    f_no = 1;
    f_agents[0][0] = agent;

    for (level = 0; level <= maximum_level; level++) {
        /* for each flood level */
        flooded_nodes = 0;
        for (l = 0; l < f_no; l++) {
            messages++;

            /* get the number of the agent that forwards the message now */
            i = f_agents[l][level % 2];
            // printf("------->%d\n",i);

            for (j = 0; j < K; j++)
            /* for each agent in the cache INFORM HIM */
                if (ran(prob)) {
                    destination_agent = agents[i].cache[j];
                    retries = 0;
                    old_k_location = agents[i].known_location[j];

                    if (agents[i].known_location[j] ==
                        agents[destination_agent].location
                        || use_replication)
                        /* if we push for location update
                            only push if the agent has not moved

                           if we push for replica update
                           then we assume that the network is always
                           correct (no moves)
                           so push can continue
```

```
                    */
                    if (agents[destination_agent].marked == -1) {
                    /* if this agent hasn't received the message yet */
                        if (algorithm == SNOOP)     /* snooping */
                            add_to_cache(destination_agent,
                                         (turn + exp_time), agent,
                                         new_location);
                        /* mark this agent. It will forward
                           the message at the next level */
                        f_agents[flooded_nodes][(level + 1) % 2] =
                            destination_agent;
                        flooded_nodes++;
                        agents[destination_agent].marked = level;
                        agents_inf_at_each_level[level]++;





                        /* UPDATES */
                        if (use_replication) {
                            /*update just the replica */
                            p = agents[i].replica_head;
                            while (p != NULL) {
                                if (p->agent == destination_agent)
                                    p->location = new_location;
                                p = p->next;
                            }
                        } else {
                            /* update location */
                            for (k = 0; k < K; k++)
                                if (agents[destination_agent].
                                    cache[k] == agent)
                                    agents[destination_agent].
                                        known_location[k] =
                                        new_location;
                        }



                    } else
                        messages++;
```

```
                }               // if prob
        }                       // for every agent that forwards
        prob = prob * (1 - decay_par);
        if (prob <= (float) 1.0 / K)
            prob = (float) 1.0 / K;
        f_no = flooded_nodes;
    }                           // for every level
}
```

C.5.0.64 void replace ()

Αντικαθιστά ένα entrie στην cache κάποιου agent.

Χρησιμοποιείται στον αλγόριθμο inverted cache όπου η αντικατάσταση ενός entrie προκαλεί κίνηση μηνυμάτων.Ορισμός στη γραμμή 48 του αρχείου inverted_cache.c.

References K, και rep_messages.

Αναφορά από simulation().

```
49 {
50     rep_messages += K;
51 }
```

C.5.0.65 void validate_agents_cache (int agent, int cur_turn)

Εκτελεί το periodic pull του αλγορίθμου snooping.

Παράμετροι:

agent ποιος agent θα κάνει το περιοδικό pull

cur_turn ο τρέχων γύρος.

Αυτή η συνάρτηση καλείται περιοδικά από κάθε agent. Πρέπει να κληθεί σε διαστήματα που είναι μικρότερα από το expiration time του snooping directory. Αυτό που κάνει είναι ότι στέλνει μηνύματα pull για κάθε cache entry του agent ώστε να ενημερώσει ολόκληρη την cache του.Ορισμός στη γραμμή 82 του αρχείου snooping_funcs.c.

References agent, agents, exp_time, K, agent_t::known_location, pull_algorithm(), pull_messages, και agent_t::time_to_validate.

Αναφορά από validate_expired_cache().

```
83 {
84     int i;
85     int new_loc;
86     int tmp_mess;
87
88     tmp_mess = pull_messages;
89     pull_messages = 0;
90     agents[agent].time_to_validate = cur_turn + exp_time;
91     for (i = 0; i < K; i++) {
92
93         new_loc =
94             pull_algorithm(agent, agents[agent].known_location[i],
95                            agents[agent].cache[i], cur_turn);
96         if (new_loc > agents[agent].known_location[i]) {
97             agents[agent].known_location[i] = new_loc;
98
99         }
100     }
101     pull_messages = tmp_mess + pull_messages / K;
102 }
```

C.5.0.66 void validate_expired_cache (int cur_turn)

Ενημερώνει την cache όλων των agent.

Παράμετροι:

cur_turn ο τρέχων γύρος.

Σαρώνει όλους τους agents καλώντας την **validate_agents_cache()**(σελ. 175)Ορισμός στη γραμμή 112 του αρχείου snooping_funcs.c.

References agents, num_agents, agent_t::time_to_validate, και validate_agents_cache().

Αναφορά από simulation().

```
{
    int i;

    for (i = 0; i < num_agents; i++)
        if (agents[i].time_to_validate <= cur_turn)
            validate_agents_cache(i, cur_turn);
}
```

Παράρτημα D

Τεκμηρίωση δομών δεδομένων

D.1 agent_t Αναφορά Δομής

Ουσιαστικά, αυτή η δομή είναι ο agent. Περιέχει πληροφορίες όπως η τωρινή του θέση, η cache του, η inverted cache (άν έχει), τον κατάλογο snooping (αν υπάρχει), το πόσο δημοφιλής είναι, το ποια resources προσφέρει κτλ.

`#include <globals.h>`

Πεδία Δεδομένων

- int **location**

  Η θέση του.

- int **marked**

  Το αν χρησιμοποιήθηκε στο push.

- int **flood_marks**

  Το αν χρησιμοποιήθηκε στο pull.

- float **popularity**

Το πόσο δημοφιλής είναι.

- int **time_to_validate**

  Χρησιμοποιείται για το snooping.Σε ποιό turn θα κάνει poll από την γειτονιά

- **flood_cache ∗ f_cache_head**

  Το head της λίστας των agent που έχουμε στο snooping directory μας.

- **flood_cache ∗ f_cache_tail**

  Το tail της λίστας των agent που έχουμε στο snooping directory μας.

- int **f_cache_size**

  Ο αριθμός των agent που έχουμε στο snooping directory.

- **flood_cache ∗ replica_head**

  Ομοίως με το **agent_t::f_cache_head**(σελ. 181) *μόνο που αφορά συνδεδεμένη λίστα με replicas.*

- **flood_cache ∗ replica_tail**

  Ομοίως με το **agent_t::f_cache_tail**(σελ. 181) *μόνο που αφορά συνδεδεμένη λίστα με replicas.*

- int **replica_size**

  Ομοίως με το **agent_t::f_cache_size**(σελ. 181).

- int ∗ **cache**

  Εδώ είναι οι K θέσεις με την cache του agent.

- int ∗ **known_location**

  Η θέση των agent που ξέρουμε.

Ορισμός στη γραμμή 61 του αρχείου globals.h.

Τεκμηρίωση Πεδίων

D.1.0.67 int* agent_t::cache

Εδώ είναι οι K θέσεις με την cache του agent.

Πρόκειται για έναν K-διάστατο πίνακα Ορισμός στη γραμμή 77 του αρχείου globals.h.

Αναφορά από final_consistency(), initialize(), move_inverted(), print_graph_stats(), pull_algorithm(), push_to_network(), και simulation().

D.1.0.68 flood_cache* agent_t::f_cache_head

Το head της λίστας των agent που έχουμε στο snooping directory μας.

Ορισμός στη γραμμή 69 του αρχείου globals.h.

Αναφορά από add_to_cache(), delete_expired_cache(), initialize(), move_inverted(), print_lists(), και pull_algorithm().

D.1.0.69 int agent_t::f_cache_size

Ο αριθμός των agent που έχουμε στο snooping directory.

Ορισμός στη γραμμή 71 του αρχείου globals.h.

Αναφορά από add_to_cache(), delete_expired_cache(), initialize(), και simulation().

D.1.0.70 flood_cache * agent_t::f_cache_tail

Το tail της λίστας των agent που έχουμε στο snooping directory μας.

Ορισμός στη γραμμή 69 του αρχείου globals.h.

Αναφορά από add_to_cache(), delete_expired_cache(), και initialize().

D.1.0.71 int agent_t::flood_marks

Το αν χρησιμοποιήθηκε στο pull.

Εσωτερικό σημάδι για να μήν τον ξαναχρησιμοποιήσουμε Ορισμός στη γραμμή 64 του αρχείου globals.h.

Αναφορά από clear_flood_marks(), και pull_algorithm().

D.1.0.72 int* agent_t::known_location

Η θέση των agent που ξέρουμε.

Για κάθε έναν από τους agent που ξέρουμε στο **agent_t::cache**(σελ. 181), η θέση που γνωρίζουμε ότι βρίσκεται. Μπορεί να είναι και λάθος αν έχει μετακινηθεί στο ενδιάμεσο. Ορισμός στη γραμμή 78 του αρχείου globals.h.

Αναφορά από final_consistency(), initialize(), move_inverted(), pull_algorithm(), push_to_network(), simulation(), και validate_agents_cache().

D.1.0.73 int agent_t::location

Η θέση του.

Ορισμός στη γραμμή 62 του αρχείου globals.h.

Αναφορά από final_consistency(), initialize(), move_inverted(), pull_algorithm(), push_to_network(), και simulation().

D.1.0.74 int agent_t::marked

Το αν χρησιμοποιήθηκε στο push.

Εσωτερικό σημάδι για να μήν τον ξαναχρησιμοποιήσουμε Ορισμός στη γραμμή 63 του αρχείου globals.h.

Αναφορά από clear_marks(), και push_to_network().

D.1.0.75 float agent_t::popularity

Το πόσο δημοφιλής είναι.

Ορισμός στη γραμμή 66 του αρχείου globals.h.

Αναφορά από initialize().

D.1.0.76 flood_cache* agent_t::replica_head

Ομοίως με το **agent_t::f_cache_head**(σελ. 181) μόνο που αφορά συνδεδεμένη λίστα με replicas.

Όταν έχουμε replication, φυλλάμε εδώ όλες τις replicas που κρατάμε για resources άλλων κόμβων. Ορισμός στη γραμμή 73 του αρχείου globals.h.

Αναφορά από create_replica(), delete_old_replicas(), pull_algorithm(), push_to_network(), και simulation().

D.1.0.77 int agent_t::replica_size

Ομοίως με το **agent_t::f_cache_size**(σελ. 181).

Ορισμός στη γραμμή 75 του αρχείου globals.h.

Αναφορά από create_replica(), και delete_old_replicas().

D.1.0.78 flood_cache * agent_t::replica_tail

Ομοίως με το **agent_t::f_cache_tail**(σελ. 181) μόνο που αφορά συνδεδεμένη λίστα με replicas.

Όταν έχουμε replication, φυλλάμε εδώ όλες τις replicas που κρατάμε για resources άλλων κόμβων. Όταν έχουμε replication, φυλλάμε εδώ όλες τις replicas που κρατάμε για resources άλλων κόμβων. Ορισμός στη γραμμή 73 του αρχείου globals.h.

Αναφορά από create_replica(), και delete_old_replicas().

D.1.0.79 int agent_t::time_to_validate

Χρησιμοποιείται για το snooping.Σε ποιό turn θα κάνει poll από την γειτονιά

Ορισμός στη γραμμή 68 του αρχείου globals.h.

Αναφορά από initialize(), validate_agents_cache(), και validate_expired_cache().

Η τεκμηρίωση για αυτή η δομή δημιουργήθηκε απο το ακόλουθο αρχείο:

- **globals.h**

D.2 flood_cache_t Αναφορά Δομής

Πρόκειται για συνδεδεμένη λίστα στην οποία αποθηκεύουμε το snooping directory και το inverted directory του κάθε agent. Έτσι έχει πληροφορίες για τον ποιον agent αφορά, την θέση που μάθαμε, σε πιο γύρο λήγει η πληροφορία αυτή κτλ. Επίσης χρησιμοποιείται και για τα replicas.

`#include <globals.h>`

Πεδία Δεδομένων

- int **agent**

  Ο agent τον οποίο αφορά η πληροφορία.

- int **location**

  Η θέση του agent που μάθαμε.

- int **expire**

  Σε ποιό turn λήγει η cache αυτή (θα πρέπει να διαγραφεί).

- **flood_cache_t** ∗ **next**

  Το επόμενο στοιχείο, για να φτιαχθεί η συνδεδεμένη λίστα.

Λεπτομερής Περιγραφή

Προσωρινή cache που κρατά ο κάθε agent και αφορά locations που έμαθε κατά το push/pull.

Συνδεδεμένη λίστα που χρησιμοποιείται για το snooping και για τα replicas.

Για παράδειγμα, κάθε φορά που μας έρχεται ένα snooping μήνυμα που αφορά την θέση ενός agent, αποθηκεύουμε την θέση αυτή στην λίστα μέχρι να λήξει το expiration time **flood_cache_t::expire**(σελ. 186)

Ορισμός στη γραμμή 52 του αρχείου globals.h.

Τεκμηρίωση Πεδίων

D.2.0.80 int flood_cache_t::agent

Ο agent τον οποίο αφορά η πληροφορία.

Ορισμός στη γραμμή 53 του αρχείου globals.h.

Αναφορά από add_to_cache(), create_replica(), move_inverted(), pull_algorithm(), push_to_network(), και simulation().

D.2.0.81 int flood_cache_t::expire

Σε ποιό turn λήγει η cache αυτή (θα πρέπει να διαγραφεί).

Ορισμός στη γραμμή 55 του αρχείου globals.h.

Αναφορά από add_to_cache(), create_replica(), delete_expired_cache(), delete_old_replicas(), και print_lists().

D.2.0.82 int flood_cache_t::location

Η θέση του agent που μάθαμε.

Ορισμός στη γραμμή 54 του αρχείου globals.h.

Αναφορά από add_to_cache(), create_replica(), pull_algorithm(), push_to_network(), και simulation().

D.2.0.83 struct flood_cache_t* flood_cache_t::next

Το επόμενο στοιχείο, για να φτιαχθεί η συνδεδεμένη λίστα.

Ορισμός στη γραμμή 56 του αρχείου globals.h.

Αναφορά από add_to_cache(), create_replica(), delete_expired_cache(), delete_old_replicas(), move_inverted(), print_lists(), pull_algorithm(), και push_to_network().

Η τεκμηρίωση για αυτή η δομή δημιουργήθηκε απο το ακόλουθο αρχείο:

- **globals.h**

Παράρτημα Ε

Τεκμηρίωση αρχείων

Ε.1 globals.h Αναφορά Αρχείου

Περιέχει όλες τις global μεταβλητές που χρησιμοποιούνται στον simulator.

```
#include <stdio.h>

#include <stdlib.h>
```

Δομές Δεδομένων

- struct **flood_cache_t**

  Προσωρινή cache που κρατά ο κάθε agent και αφορά locations που έμαθε κατά το push/pull.

- struct **agent_t**

  Η δομή που περιέχει όλες τις πληροφορίες που χρειάζεται ένας agent.

Ορισμοί

- #define **num_agents** 1000

  Ο αριθμός των agent που έχουμε στο σύστημα.

- #define **max_flood_level** 20

  Το μέγιστο flood level που μπορούμε να χρησιμοποιήσουμε από τους αλγορίθμους μας.

Ορισμοί Τύπων

- typedef **flood_cache_t flood_cache**

  Προσωρινή cache που κρατά ο κάθε agent και αφορά locations που έμαθε κατά το push/pull.

- typedef **agent_t agent**

  Η δομή που περιέχει όλες τις πληροφορίες που χρειάζεται ένας agent.

Απαριθμήσεις

- enum { **PLAIN_PUSH**, **SNOOP**, **LEASING** }

  Τα ονόματα των βασικών αλγορίθμων.

Συναρτήσεις

- int **ran** (double probability)

  Συνάρτηση που επιστρέφει 1 με δοσμένη πιθανότητα.

- float **final_consistency** (int turn)

  Υπολογίζει το ποσοστό των έγκυρων cache entries.

- void **create_replica** (int to_agent, int expr, int **agent**, int location)

  Φτιάχνει replica μιας πληροφορίας σε κάποιον agent.

- void **add_to_cache** (int to_agent, int expr, int **agent**, int location)

  Φτιάχνει ενα snoopig entrie σε κάποιον agent.

Μεταβλητές

- enum { ... } **alg_types**

  Τα ονόματα των βασικών αλγορίθμων.

- float **exp_time**

  Ο χρόνος σε turns στον οποίο λήγει το snooping entry.

- int **algorithm**

  Ο αλγόριθμος που χρησιμοποιείται αυτή την στιγμή.

- double **p_change**

  Πιθανότητα να γίνει αλλαγή στην cache κάποιου agent σε κάθε γύρο.

- float **p_move**

  Πιθανότητα να μετακινηθεί ένας agent σε κάθε γύρο.

- double **p_pull**

  Πιθανότητα να εκτελέσουμε on-demand pull σε κάθε γύρο.

- int **K**

  Το μέγεθος της cache.

- float **per_cache**

  Το ποσοστό των agent που υπάρχουν στην cache.Το K προκύπτει από αυτό.

- float **percentage_of_popular_agents**

  Το ποσοστό των agent που είναι δημοφηλείς.

- float **popularity**

  Το πόσο δημοφηλείς είναι οι δημοφηλείς agents.

- int **turns**

  Ο αριθμός των turns του simulation.

- int **f_no**
- int **f_agents** [num_agents][2]
- int **f_pull_no**
- int **f_pull_agents** [num_agents][2]
- float **inv_cache_percentage**
- **agent agents** [num_agents]

  Ο βασικός πίνακας που περιέχει όλους τους agents του συστήματος (δείκτες στην δομή **agent**(σελ. 196)*).*

- int **messages**

  Κρατά τον αριθμό μηνυμάτων του push μόνο.

- int **pull_messages**

  Κρατά τον αριθμό μηνυμάτων του pull μόνο.

- int **pull_level_found**

  Εδώ αποθηκεύεται το βάθος (στο path) στο οποίο βρέθηκε η πληροφορία κατά το pull.

- int **rep_messages**

  Μηνύματα που χρειάστηκαν για cache replacement.

- int **moves**

  Αριθμός μετακινήσεων που έγιναν κατα την διάρκεια του simulation.

- int **pulls**

  Αριθμός pull που έγιναν κατα την διάρκεια του simulation.

- int **av_dir_size**

  Μέσο μέγεθος snooping directory.

- int **agents_inf_at_each_level** [max_flood_level]

  Κρατάει στατιστικά για το πόσοι agents ενημερώθηκαν σε κάθε βάθος του push.

- int **pull_found_at_each_level** [max_flood_level]

  Κρατάει στατιστικά για το πόσοι agents ερωτήθηκαν σε κάθε βάθος του pull.

- int **pulls_performed**

  Ο αριθμός των pulls που έγιναν κατα την διάρκεια του simulation.

- FILE * **f_cache**

  Αρχείο αποθήκευσης στατιστικών: ποσοστό valid cache.

- FILE * **f_mes**

  Αρχείο αποθήκευσης στατιστικών: αριθμός μηνημάτων.

- FILE * **f_dir_size**

  Αρχείο αποθήκευσης στατιστικών: μέσο μέγεθος καταλόγου snooping.

- FILE * **f_levels**

  Αρχείο αποθήκευσης στατιστικών: πόσα pulls απαντήθηκαν σε κάθε level ή και δεν βρέθηκε σωστή απάντηση.

- float * **val_to_test**

Στον δείκτη αυτό βάζουμε την παράμετρο που θέλουμε να μελετήσουμε.

- float **l_bound**

  Η αρχική τιμή της παραμέτρου, με αυτή θα ξεκινήσει το simulation **val_to_test**(σελ. 158).

- float **up_bound**

  Όταν φτάσουμε σε αυτή την τιμή σταματάμε το simulation **val_to_test**(σελ. 158)

- float **incr**

  Η παράμετρος αυξάνεται κατά τόσο σε κάθε κύκλο απο turns **val_to_test**(σελ. 158).

- int * **levels**
- int **use_replication**

  Αν θα χρησιμοποιηθεί replication ή όχι.

- int **replica_time**

  Πόσο χρόνο θα κρατάμε τα replicas.

- double **p_replicate**

  Η πιθανότητα να κάνουμε replicate σε συγκεκριμένο κόμβο του path.

- int **maximum_level**

  Το μέγιστο βάθος του push.

- int **maximum_pull_level**

  Το μέγιστο βάθος του pull.

- float **decay_par**

  Η παράμετρος decay.

- int **do_pull**

  Αν θα γινονται on-demand pulls.

- int **pull_decay_par**

  Το decay κατά το pull.

- float **replace_prob**

  Η πιθανότητα να προβούμε σε αντικατάσταση σε κάθε γύρο.

Λεπτομερής Περιγραφή

Περιέχει όλες τις global μεταβλητές που χρησιμοποιούνται στον simulator.

Οι μεταβλητές αφορούν παραμέτρους του simulation, εσωτερικές δομές για την λειτουργία του, μεταβλητές που κρατάν στατιστικά κτλ.

Ορισμός στο αρχείο **globals.h**.

Τεκμηρίωση Ορισμών

E.1.0.84 #define max_flood_level 20

Το μέγιστο flood level που μπορούμε να χρησιμοποιήσουμε από τους αλγορίθμους μας.

Ουσιαστικά χρησιμοποιείται για να δηλώσουμε στατικούς πίνακες.

Προειδοποίηση:

Το μέγεθος του flood για τους αλγορίθμους push/pull πρέπει να είναι μικρότερο για να μην έχουμε segmentation faults

Ορισμός στη γραμμή 16 του αρχείου globals.h.

E.1.0.85 #define num_agents 1000

Ο αριθμός των agent που έχουμε στο σύστημα.

Ορισμός στη γραμμή 13 του αρχείου globals.h.

Αναφορά από clear_flood_marks(), clear_marks(), delete_expired_cache(), delete_old_replicas(), destroy_graph(), final_consistency(), initialize(), print_G(), print_graph_stats(), print_lists(), simulation(), και validate_expired_cache().

Τεκμηρίωση Ορισμών Τύπων

E.1.0.86 typedef struct agent_t agent

Η δομή που περιέχει όλες τις πληροφορίες που χρειάζεται ένας agent.

Αναφορά από add_to_cache(), create_replica(), move_inverted(), push_to_network(), και validate_agents_cache().

E.1.0.87 typedef struct flood_cache_t flood_cache

Προσωρινή cache που κρατά ο κάθε agent και αφορά locations που έμαθε κατά το push/pull.

Συνδεδεμένη λίστα που χρησιμοποιείται για το snooping και για τα replicas.

Για παράδειγμα, κάθε φορά που μας έρχεται ένα snooping μήνυμα που αφορά την θέση ενός agent, αποθηκεύουμε την θέση αυτή στην λίστα μέχρι να λήξει το expiration time **flood_cache_t::expire**(σελ. 186)

Αναφορά από add_to_cache(), create_replica(), delete_expired_cache(), delete_old_replicas(), move_inverted(), print_lists(), pull_algorithm(), και push_to_network().

Τεκμηρίωση Απαριθμήσεων

E.1.0.88 anonymous enum

Τα ονόματα των βασικών αλγορίθμων.

Θέτοντας αυτή την τιμή επιλέγεται ο κατάλληλος αλγόριθμος

Τιμές Απαριθμήσεων:

PLAIN_PUSH

SNOOP

LEASING

Ορισμός στη γραμμή 19 του αρχείου globals.h.

```
19 { PLAIN_PUSH, SNOOP, LEASING } alg_types;
```

Τεκμηρίωση Μεταβλητών

E.1.0.89 agent agents[num_agents]

Ο βασικός πίνακας που περιέχει όλους τους agents του συστήματος (δείκτες στην δομή **agent**(σελ. 196)).

Ορισμός στη γραμμή 111 του αρχείου main.c.

Αναφορά από add_to_cache(), clear_flood_marks(), clear_marks(), create_replica(), delete_expired_cache(), delete_old_replicas(), destroy_graph(), final_consistency(), initialize(), move_inverted(), print_G(), print_graph_stats(), print_lists(), pull_algorithm(), push_to_network(), simulation(), validate_agents_cache(), και validate_expired_cache().

E.1.0.90 enum { ... } alg_types

Τα ονόματα των βασικών αλγορίθμων.

Θέτοντας αυτή την τιμή επιλέγεται ο κατάλληλος αλγόριθμος

E.1.0.91 int f_agents[num_agents][2]

Αναφορά από push_to_network().

E.1.0.92 int f_no

Αναφορά από push_to_network().

E.1.0.93 int f_pull_agents[num_agents][2]

Αναφορά από pull_algorithm().

E.1.0.94 int f_pull_no

Αναφορά από pull_algorithm().

E.1.0.95 float inv_cache_percentage

Ορισμός στη γραμμή 116 του αρχείου main.c.

Αναφορά από move_inverted(), print_graph_stats(), και set_simulation_parameters().

E.1.0.96 int* levels

Ορισμός στη γραμμή 82 του αρχείου main.c.

Αναφορά από simulation().

E.2 inverted_cache.c Αναφορά Αρχείου

Συναρτήσεις που χρειάζονται για τον αλγόριθμο inverted cache.

`#include "globals.h"`

Συναρτήσεις

- void **move_inverted** (int **agent**)

  Μετακινεί έναν δοσμένο agent σε νέα θέση.

- void **replace** ()

  Αντικαθιστά ένα entrie στην cache κάποιου agent.

Ορισμός στο αρχείο **inverted_cache.c**.

E.3 main.c Αναφορά Αρχείου

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include "globals.h"
```

Ορισμοί

- #define **num_agents** 1000

Συναρτήσεις

- int **main** ()

Μεταβλητές

- float **exp_time**

  Ο χρόνος σε turns στον οποίο λήγει το snooping entry.

- int **algorithm**

  Ο αλγόριθμος που χρησιμοποιείται αυτή την στιγμή.

- double **p_change**

  Πιθανότητα να γίνει αλλαγή στην cache κάποιου agent σε κάθε γύρο.

- float **p_move**

  Πιθανότητα να μετακινηθεί ένας agent σε κάθε γύρο.

- double **p_pull**

  Πιθανότητα να εκτελέσουμε on-demand pull σε κάθε γύρο.

- int **K**

  Το μέγεθος της cache.

- float **per_cache**

  Το ποσοστό των agent που υπάρχουν στην cache.Το K προκύπτει από αυτό.

- float **percentage_of_popular_agents**

  Το ποσοστό των agent που είναι δημοφηλείς.

- float **popularity**

  Το πόσο δημοφηλείς είναι οι δημοφηλείς agents.

- int **messages**

  Κρατά τον αριθμό μηνυμάτων του push μόνο.

- int **pull_messages**

  Κρατά τον αριθμό μηνυμάτων του pull μόνο.

- int **pull_level_found**

  Εδώ αποθηκεύεται το βάθος (στο path) στο οποίο βρέθηκε η πληροφορία κατά το pull.

- int **rep_messages**

  Μηνύματα που χρειάστηκαν για cache replacement.

- int **moves**

  Αριθμός μετακινήσεων που έγιναν κατα την διάρκεια του simulation.

- int **pulls**

  Αριθμός pull που έγιναν κατα την διάρκεια του simulation.

- int **av_dir_size**

  Μέσο μέγεθος snooping directory.

- int **agents_inf_at_each_level** [max_flood_level]

  Κρατάει στατιστικά για το πόσοι agents ενημερώθηκαν σε κάθε βάθος του push.

- int **pull_found_at_each_level** [max_flood_level]

  Κρατάει στατιστικά για το πόσοι agents ερωτήθηκαν σε κάθε βάθος του pull.

- int **pulls_performed**

  Ο αριθμός των pulls που έγιναν κατα την διάρκεια του simulation.

- FILE * **f_cache**

  Αρχείο αποθήκευσης στατιστικών: ποσοστό valid cache.

- FILE * **f_mes**

  Αρχείο αποθήκευσης στατιστικών: αριθμός μηνημάτων.

- FILE * **f_dir_size**

  Αρχείο αποθήκευσης στατιστικών: μέσο μέγεθος καταλόγου snooping.

- FILE * **f_levels**

  Αρχείο αποθήκευσης στατιστικών: πόσα pulls απαντήθηκαν σε κάθε level ή και δεν βρέθηκε σωστή απάντηση.

- float * **val_to_test**

  Στον δείκτη αυτό βάζουμε την παράμετρο που θέλουμε να μελετήσουμε.

- float **l_bound**

  Η αρχική τιμή της παραμέτρου, με αυτή θα ξεκινήσει το simulation **val_to_test**(σελ. 158).

- float **up_bound**

Όταν φτάσουμε σε αυτή την τιμή σταματάμε το simulation **val_to_test**(σελ. 158)

- float **incr**

  Η παράμετρος αυξάνεται κατά τόσο σε κάθε κύκλο απο turns **val_to_test**(σελ. 158).

- int **turns**

  Ο αριθμός των turns του simulation.

- int ∗ **levels**
- int **use_replication**

  Αν θα χρησιμοποιηθεί replication ή όχι.

- int **replica_time**

  Πόσο χρόνο θα κρατάμε τα replicas.

- double **p_replicate**

  Η πιθανότητα να κάνουμε replicate σε συγκεκριμένο κόμβο του path.

- int **maximum_level**

  Το μέγιστο βάθος του push.

- int **maximum_pull_level**

  Το μέγιστο βάθος του pull.

- float **decay_par**

  Η παράμετρος decay.

- int **do_pull**

  Αν θα γινονται on-demand pulls.

- int **pull_decay_par**

  Το decay κατά το pull.

- float **replace_prob**

  Η πιθανότητα να προβούμε σε αντικατάσταση σε κάθε γύρο.

- **agent agents** [num_agents]

  Ο βασικός πίνακας που περιέχει όλους τους agents του συστήματος (δείκτες στην δομή **agent**(σελ. 196)*).*

- float **inv_cache_percentage**

Τεκμηρίωση Ορισμών

E.3.0.97 #define num_agents 1000

Ορισμός στη γραμμή 7 του αρχείου main.c.

Τεκμηρίωση Συναρτήσεων

E.3.0.98 int main ()

Ορισμός στη γραμμή 125 του αρχείου main.c.

```
{

    set_simulation_parameters();
    batch_run();

    printf("\n");
    system("pause");

    return 0;
}
```

Τεκμηρίωση Μεταβλητών

E.3.0.99 agent agents[num_agents]

Ο βασικός πίνακας που περιέχει όλους τους agents του συστήματος (δείκτες στην δομή **agent**(σελ. 196)).

/** @} Ορισμός στη γραμμή 111 του αρχείου main.c.

Αναφορά από add_to_cache(), clear_flood_marks(), clear_marks(), create_replica(), delete_-expired_cache(), delete_old_replicas(), destroy_graph(), final_consistency(), initialize(), move_inverted(), print_G(), print_graph_stats(), print_lists(), pull_algorithm(), push_to_-network(), simulation(), validate_agents_cache(), και validate_expired_cache().

E.3.0.100 float inv_cache_percentage

Ορισμός στη γραμμή 116 του αρχείου main.c.

Αναφορά από move_inverted(), print_graph_stats(), και set_simulation_parameters().

E.3.0.101 int* levels

Ορισμός στη γραμμή 82 του αρχείου main.c.

Αναφορά από simulation().

E.4 net_funcs.c Αναφορά Αρχείου

Το αρχείο αυτό περιέχει συναρτήσεις που αφορούν την κατασκευή και την συντήρηση του δικτύου των agents.

```
#include "globals.h"
```

Συναρτήσεις

- int **ran** (double probability)

  Συνάρτηση που επιστρέφει 1 με δοσμένη πιθανότητα.

- void **print_G** ()

  Τυπώνει τον γράφο των agent.

- void **initialize** ()

  Αρχικοποιεί τον γράφο.

- void **destroy_graph** ()

  Καταστρέφει τον γράφο των agent (free memory).

- void **print_graph_stats** ()

  Τυπώνει στατιστικά για τον γράφο μας.

- void **print_lists** (int cur_round)

  Τυπώνει λίστες με γείτονες για κάθε agent.

Ορισμός στο αρχείο **net_funcs.c**.

E.5 pull.c Αναφορά Αρχείου

Συναρτήσεις που χρειάζονται για τον αλγόριθμο pull.

```
#include "globals.h"
```

Συναρτήσεις

- void **clear_flood_marks** ()

  Καθαρίζει τα σημάδια που βάζουμε κατά την διάρκεια του αλγόριθμου pull.

- int **pull_algorithm** (int sour_agent, int known_location, int dest_agent, int turn)

  Ο αλγόριθμος pull.

Ορισμός στο αρχείο **pull.c**.

E.6 push.c Αναφορά Αρχείου

Συναρτήσεις που χρειάζονται για τον αλγόριθμο push.

`#include "globals.h"`

Συναρτήσεις

- void **clear_marks** ()

  Καθαρίζει τα σημάδια που βάζουμε κατά την διάρκεια του αλγόριθμου push.

- void **push_to_network** (int **agent**, int turn)

  Ο αλγόριθμος pull.

Ορισμός στο αρχείο **push.c**.

E.7 replication.c Αναφορά Αρχείου

Συναρτήσεις που αφορούν τον χειρισμό των αντιγράφων.

```
#include "globals.h"
```

Συναρτήσεις

- void **delete_old_replicas** (int cur_round)

  Σβήνει τα παλιά replicas.

- void **create_replica** (int to_agent, int expr, int **agent**, int location)

  Φτιάχνει replica μιας πληροφορίας σε κάποιον agent.

Ορισμός στο αρχείο **replication.c**.

E.8 simulation.c Αναφορά Αρχείου

Περιέχει τις συναρτήσεις που ενορχηστρώνουν το simulation.

```
#include "globals.h"
```

Συναρτήσεις

- void **set_simulation_parameters** ()

  Στην συνάρτηση αυτή δηλώνουμε τις τιμές που θέλουμε να θέσουμε σε κάθε παράμετρο για το simulation.

- void **simulation** ()

  Τρέχει το simulation για έναν αριθμό απο turns κρατώντας στατιστικά Πρόκειται για το loop που ξεκινά από τον αρχικοποιημένο γράφο και τρέχει το πείραμα για έναν αριθμό από **turns**(σελ. 157).

- int **batch_run** ()

  Καλεί την **simulation()**(σελ. 153) *για κάθε τιμή του range της παραμέτρου* **val_to_test**(σελ. 158) *που θέλουμε να εξετάσουμε.*

Λεπτομερής Περιγραφή

Με την χρήση των παρακάτω συναρτήσεων το simulation γίνεται με έναν αυτοματοποιημένο τρόπο. Απλά δηλώνουμε στην set_simulation_parameters τις τιμές των παραμέτρων που θέλουμε να κρατήσουμε σταθερές και κάποιο range μιας μεταβλητής που θέλουμε να εξετάσουμε και μετά αναλαμβάνει αυτόματα να τρέξει το simulation για κάθε τιμή του range (και για έναν αριθμό απο turns) και να κρατήσει στατιστικά.

Ορισμός στο αρχείο **simulation.c**.

E.9 snooping_funcs.c Αναφορά Αρχείου

Συναρτήσεις που χρειάζονται για το snooping.

```
#include "globals.h"
```

Συναρτήσεις

- void **delete_expired_cache** (int cur_round)

  Σβήνει τα expired snooping cache entries.

- void **add_to_cache** (int to_agent, int expr, int **agent**, int location)

  Φτιάχνει ενα snoopig entrie σε κάποιον agent.

- void **validate_agents_cache** (int **agent**, int cur_turn)

  Εκτελεί το periodic pull του αλγορίθμου snooping.

- void **validate_expired_cache** (int cur_turn)

  Ενημερώνει την cache όλων των agent.

Ορισμός στο αρχείο **snooping_funcs.c**.

E.10 statistics.c Αναφορά Αρχείου

Περιέχει συναρτήσεις που βοηθούν στην συγκέντρωση, τον υπολογισμό και την αποθήκευση στατιστικών.

```
#include "globals.h"
```

Συναρτήσεις

- float **final_consistency** (int turn)

    Υπολογίζει το ποσοστό των έγκυρων cache entries.

- void **initialize_stats** ()

    Αρχικοποιεί τις δομές και τα αρχεία που κρατούν τα στατιστικά στην αρχή.

- void **stats_initialize_round** ()

    Αρχικοποιεί τις δομές και τα αρχεία που κρατούν τα στατιστικά κάθε φορά που τρέχουμε το simulation για διαφορετική τιμή στο range που κινείται η **val_to_test**(σελ. 158).

- void **stats_finalize_round** ()

    Στο τέλος κάθε simulation (τέλος των **turns**(σελ. 157)*) γράφει τα στατιστικά στα αρχεία.*

Ορισμός στο αρχείο **statistics.c**.

Βιβλιογραφία

[1] Napster official site:. http://www.napster.com.

[2] Clay Shirky. P2p definition. In *http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html*.

[3] SETI@HOME:. http://setiathome.ssl.berkeley.edu/.

[4] United Devices:. http://www.ud.com/.

[5] The Gnutella Protocol Specification v0.4, Document Revision 1.2, Clip2. http://www.clip2.com, protocols@clip2.com.

[6] Kazaa:. http://www.kazaa.com.

[7] Freenet project. http://freenet.sourceforge.net/.

[8] I seek you (ICQ):. http://www.icq.com/.

[9] Hyacinth S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):205–244, 1996.

[10] Vassilios V. Dimakopoulos and Evaggelia Pitoura. A Peer-to-Peer Approach to Resource Discovery in Multi-Agent Systems.

[11] Vana Kalogeraki, Dimitrios Gunopulos, and D. Zeinalipour-Yazti. A local search mechanism for peer-to-peer networks. In *Proceedings of the eleventh international conference on Information and knowledge management*, pages 300–307. ACM Press, 2002.

[12] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. In *Proc. of the 22nd IEEE International Conference on Distributed Computing (IEEE ICDCS'02), 2002.*

[13] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th international conference on Supercomputing*, pages 84–95. ACM Press, 2002.

[14] I. Jawhar and J. Wu. A Two-Level Random Walk Search Protocol for Peer-to-Peer Networks. In *Proc. of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics, 2004.*

[15] Chandan K. Dubey. Searching strategies in peer to peer databases. In *Department of Computer Science and Engineering Indian Institute of Technology, Kanpur.*

[16] Xiuqi Li and Jie Wu. Searching techniques in peer-to-peer networks. In *Department of Computer Science and Engineering, Florida Atlantic University Boca Raton, FL 33431.*

[17] Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *Proc. of ICDCS 2003, 23rd International Conference on Distributed Computing Systems*, pages 76–85, Providence, Rhode Island, May 2003.

[18] Jiang Lan, Xiaotao Liu, Prashant Shenoy, and Krithi Ramamritham. Consistency maintenance in peer-to-peer file sharing networks. In *Proc. of WIAPP'03, 3rd IEEE Workshop on Internet Applications*, pages 76–85, San Jose, CA, June 2003.

[19] R. Srinivasan, C. Liang, and Krithi Ramamritham. Maintaining temporal coherency of virtual data warehouses. In *RTSS*, pages 60–, 1998.

[20] C. Gary and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *In Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, pages 202-210*, 1989.

[21] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive lease: A strong consistency mechanism for the world wide web. Technical Report UM-CS-1999-041, , 1999.

[22] S. J. Eggers and R. H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 2–15. ACM Press, 1989.

[23] Elias Leontiadis, Vassilios V. Dimakopoulos, and Evaggelia Pitoura. Cache updates in a peer-to-peer network of mobile agents. In *4th International Conference on Peer-to-Peer Computing (P2P 2004), August 2004, Zurich, Switzerland*. IEEE Computer Society, 2004.

[24] Edidth Cohen and Scott Shenker. Replication strategies in unstructured peer-to-peer netwokrs. In *nanuscript 2001.*

[25] Morpheus:. http://www.morpheus.com/.

[26] How stuff works website. http://www.howstuffworks.com/.

[27] Emma Brunskill. Building peer-to-peer systems with chord, a distributed lookup service. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 81. IEEE Computer Society, 2001.

[28] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.

[29] Arturo Crespo and Hector Garcia-Molina. Routing indices for peer-to-peer systems. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 23. IEEE Computer Society, 2002.

[30] D. Tsoumakos and N. Roussopoulos. Adaptive probabilistic search in peer-to-peer networks. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03), 2003*.

[31] D. Tsoumakos and N. Roussopoulos. Adaptive probabilistic search in peer-to-peer networks. In *technical report, CS-TR-4451, 2003.*

[32] Jie Wu and Hailan Li. A dominating-set-based routing scheme in ad hoc wireless networks. *Telecommunication Systems*, 18(1–3):13–36, 2001.

[33] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–114, Aug. 2002.

[34] Wireless node database. http://www.nodedb.com.

[35] Athens metropolitan wireless network. http://www.atwn.com.

[36] Sonic rooftop networks. http://www.sonic.net/sales/rooftop/faq.shtml.

[37] Ying-Dar Jason Lin and Yu-Ching Hsu. Multihop cellular: A new architecture for wireless communications. In *INFOCOM (3)*, pages 1273–1282, 2000.

[38] X. Hong, K. Xu, and M. Gerla. Scalable routing protocols for mobile ad hoc networks. In *IEEE Network 16*, pages 11–21. IEEE Computer Society, 2002.

[39] S. Giordano, I. Stojmenovic, and L. Blazevie. Position based routing algorithms for ad hoc networks: a taxonomy, 2001.

Ευρετήριο