ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ & ΣΥΣΤΗΜΑΤΩΝ
ΠΛΗΡΟΦΟΡΙΚΗΣ

Υλοποίηση Αλγορίθμου Δρομολόγησης και Ισοκατανομής Φορτίου σε ομότιμο δίκτυο στο περιβάλλον PlanetLab

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χαρικλής Κ. Πιτταράς

Επιβλέπων : Βασίλης Μάγκλαρης
Καθηγητής Ε.Μ.Π

Αθήνα, Ιούλιος 2008

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ & ΣΥΣΤΗΜΑΤΩΝ
ΠΛΗΡΟΦΟΡΙΚΗΣ

Υλοποίηση Αλγορίθμου Δρομολόγησης και Ισοκατανομής Φορτίου σε ομότιμο δίκτυο στο περιβάλλον PlanetLab

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χαρικλής Κ. Πιτταράς

Επιβλέπων : Βασίλης Μάγκλαρης
Καθηγητής Ε.Μ.Π

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 17η Ιουλίου 2008.

.....	.....	.....
Β.Μάγκλαρης	Σ.Παπαβασιλείου	Δ.Καλογεράς
Καθηγητής Ε.Μ.Π	Επ. Καθηγητής Ε.Μ.Π	Ερευνητής ΕΠΙΣΕΥ

Αθήνα, Ιούλιος 2008

.....

Χαρικλής Κ. Πιτταράς

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

ΠΕΡΙΛΗΨΗ

Τα ομότιμα (***P**eer-to-**P**eer*) δίκτυα έχουν αποκτήσεις τεράστιο ενδιαφέρον και έχουν γίνει πολύ δημοφιλείς στις μέρες μας. Εκτός από την ανταλλαγή αρχείων, πολύ μεγάλο ενδιαφέρον παρουσιάζουν και στην ακαδημαϊκή κοινότητα. Είναι συστήματα χωρίς κεντρική οργάνωση και ιεραρχικό έλεγχο και κάθε κόμβος-ομότιμος (***peer***) έχει τα ίδια δικαιώματα με τους υπόλοιπους. Τα συστήματα αυτά παρέχουν μια καλή υποδομή για δημιουργία συστημάτων πολύ μεγάλης κλίμακας. Για παράδειγμα, σε συστήματα με πολύ μεγάλα σύνολα δεδομένων ακόμα και η δεικτοδότηση τους είναι δύσκολο να γίνει από ένα μόνο κόμβο, για αυτό και χρειάζεται ένα κατανεμημένο σύστημα δεικτοδότησης.

Σκοπός της διπλωματικής είναι η δημιουργία ενός ομότιμου (***P2P***) συστήματος δεικτοδότησης για πολλών διαστάσεων δεδομένα και με λογαριθμική πολυπλοκότητα αναζήτησης σε περιβάλλον ***PlanetLab[12]*** (το *PlanetLab* είναι ένα παγκόσμιο ερευνητικό δίκτυο). Το σύστημα δεικτοδότησης βασίζεται στο *SkipIndex[2]*. Επίσης σε αυτό το σύστημα ενσωματώνουμε τον καινούριο αλγόριθμο Platon[3] για ισοκατανομή του φορτίου (*Load Balancing*) σε όλους τους peers.

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ

Ομότιμα δίκτυα, PlanetLab, P2P load balancing, Distributed K-D tree, Overlay Networks, Κατανεμημένοι Αλγόριθμοι, Αλγόριθμοι δρομολόγησης, Skip Index, Skip Graph, Skip List, Σύστημα Δεικτοδότησης,

ABSTRACT

Peer-to-Peer systems have obtained a significant interest and they have become very popular in our days. Except of the popular data sharing, peer-to-peer computing is also very popular in the academic community. They are systems which lack any centralized organization or hierarchical control and they provide a good substrate for creating large-scale systems. For example, in systems with huge data sets even their indexing can easily overwhelm the storage or processing capacity of a single node.

The purpose of this thesis is the implementation of a peer-to-peer indexing system for high dimensional data, with logarithmic complexity in search operation, in the *PlanetLab[12]* environment (PlanetLab is a global research network). This system is based in SkipIndex[2]. Additionally, in this system we implemented a new algorithm, Platon[3], used for load balancing.

KEYWORDS

Peer-to-Peer, PlanetLab, P2P load balancing, Distributed K-D tree, Overlay Networks, Distributed Scalable Algorithms, Lookup Protocols, Skip Index, Skip Graph, Skip List, Indexing Systems

Περιεχόμενα

1 Εισαγωγή

Σκοπός της διπλωματικής αυτής είναι η υλοποίηση ενός P2P συστήματος δεικτοδότησης για δεδομένα υψηλών διαστάσεων σε περιβάλλον *PlanetLab[12]*. Το ***PlanetLab*** είναι ένα παγκόσμιο ερευνητικό δίκτυο το οποίο υποστηρίζει την ανάπτυξη νέων υπηρεσιών δικτύου. Πολλοί ερευνητές από τον ακαδημαϊκό και γενικότερο ερευνητικό χώρο το χρησιμοποιούν για να εκτελέσουν τα πειράματα τους σε ένα κατανεμημένο περιβάλλον.

Στο P2P σύστημα μας κάθε peer κατέχει ένα μέρος του συνολικού πολυδιάστατου χώρου και σε αυτό το χώρο έχει το φορτίο του, που αναπαρίσταται σαν σημεία. Τα σημεία αυτά προέρχονται από hash κάποιων strings δηλαδή των χαρακτηριστικών (*attributes*) του κάθε στοιχείου που φυλάμε στο δίκτυο. Η αναζήτηση αυτών των σημείων βασίζεται στο ***SkipIndex[2]*** το οποίο περιγράφετε στην ενότητα 2.2, και τα βήματα που χρειάζονται είναι O(logN) όπου N ο συνολικός αριθμός των peers στο σύστημα. Με μετρήσεις που έγινα στο PlanetLab τα αποτελέσματα συμφωνούν με τα θεωρητικά. Στην ενότητα 2.1 δίνουμε κάποια γενικά στοιχεία για τα συστήματα P2P και στην ενότητα 2.2 αναλύουμε το SkipIndex. Στο κεφάλαιο 3 περιγράφουμε την υλοποίηση του P2P συστήματος.

Επίσης στο σύστημα αυτή υλοποιούμε το καινούριο αλγόριθμο ***Platon[3]*** για να διατηρούμε το δέντρο περιοχών ισοζυγισμένο, δηλαδή να εξασφαλίσουμε ότι όλοι οι peers θα έχουν ίσο αριθμό από στοιχεία. Επίσης με μετρήσεις που έχουν γίνει στο PlanetLab η πολυπλοκότητα του αλγορίθμου είναι γραμμική σε σχέση με το N, όπως προκύπτει και από την θεωρεία. Ο αλγόριθμος Platon και η υλοποίηση του περιγράφονται στο κεφάλαιο 4.

Στο κεφάλαιο 5 παρουσιάζουμε και αναλύουμε τις μετρήσεις που έχουμε πάρει σε περιβάλλον PlanetLab για αυτό το σύστημα.

2 Σχετική δουλειά σε Ομότιμα Δίκτυα (P2P)

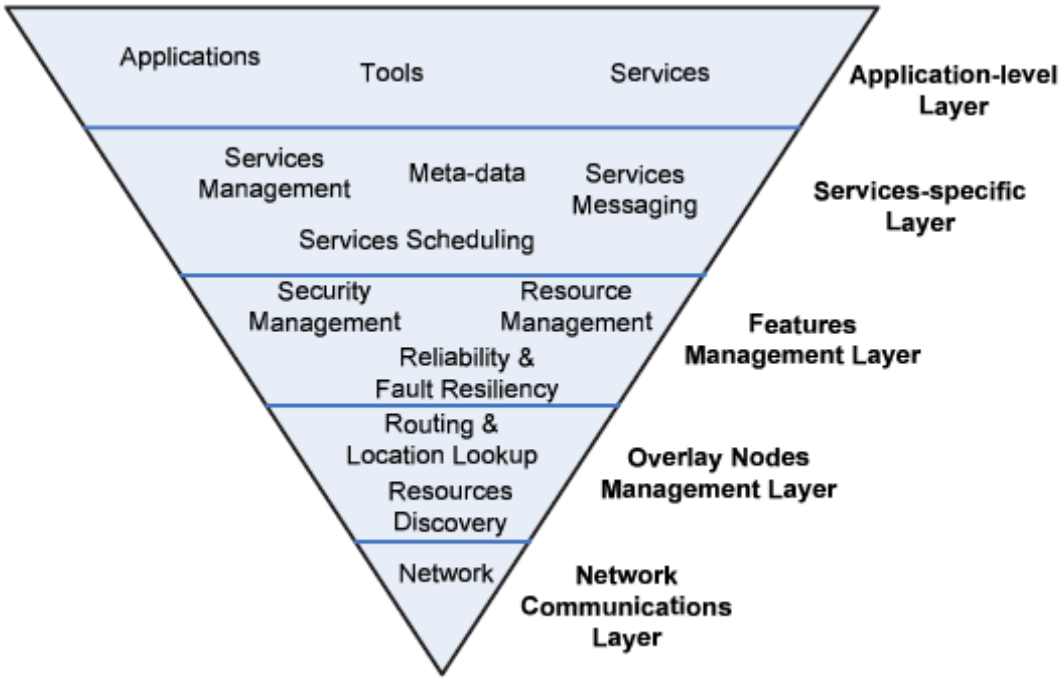
Σε αυτό το κεφάλαιο παραθέτουμε κάποια σχετική δουλεία σε ομότιμα (P2P) δίκτυα. Η παρακάτω ενότητα είναι μια γενική περιγραφή αυτών των δικτύων και βασίζεται κυρίως στο [1].

2.1 Επισκόπηση και Σύγκριση των Διαφόρων Ομότιμων Δικτύων

2.1.1 Εισαγωγή

Τα Peer-to-Peer (***P2P***) επικαλυπτόμενα (***overlay***) δίκτυα είναι κατανεμημένα συστήματα χωρίς καμιά ιεραρχική οργάνωση και κεντρικό έλεγχο. Τα δίκτυα αυτά μπορούν να οργανωθούν μόνα τους και δομούνται πάνω από υπάρχοντα δίκτυα IP, για αυτό και ονομάζονται επικαλυπτόμενα (***overlay***) δίκτυα. Προσφέρουν μια ποικιλία από χαρακτηριστικά όπως: ευρωστία, ευρείας περιοχής αρχιτεκτονική δρομολόγησης, αποτελεσματική αναζήτηση δεδομένων, επιλογή γειτονικών peers, ιεραρχική ονοματολογία, πιστοποίηση, ανωνυμία, τεράστια επεκτασιμότητα και ανεκτικότητα στα σφάλματα. Τα P2P overlay συστήματα πάνε πέρα από τις υπηρεσίες που προσφέρει το μοντέλο *πελάτη εξυπηρετητή* (***client server***), έχοντας συμμετρία στους ρόλους έτσι ώστε ένας peer να είναι client αλλά και server ταυτόχρονα. Επιτρέπουν πρόσβαση στους πόρους τους από άλλα συστήματα και υποστηρίζουν μοίρασμα των πόρων το οποίο απαιτεί ανεκτικότητα στα σφάλματα, αυτό οργάνωση και τεράστια επεκτασιμότητα.

Το σχήμα 2.1 δείχνει μια αφαιρετική P2P overlay αρχιτεκτονική, διευκρινίζοντας τα συστατικά του overlay επικοινωνιακού πλαισίου.

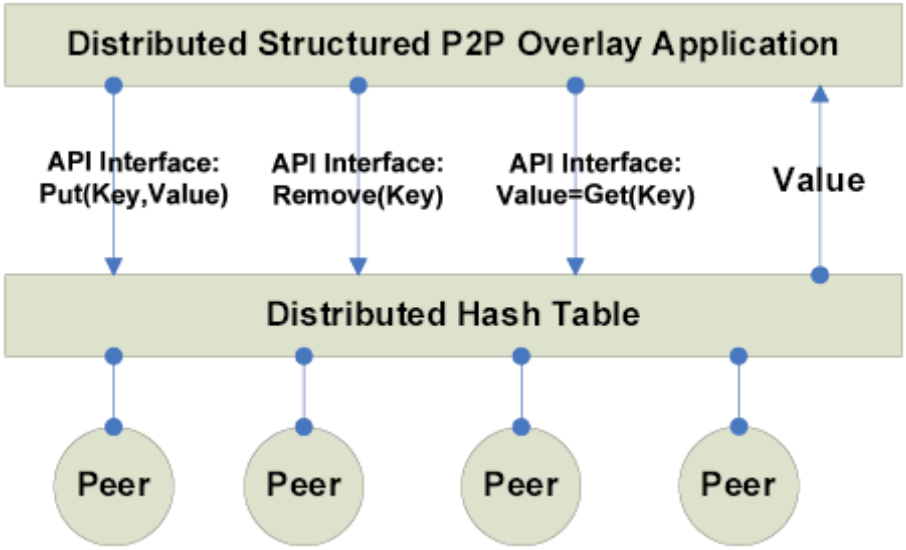


Σχημα 2.1: Αφαιρετική P2P overlay αρχιτεκτονική [1]

Το *Network Communications layer* περιγράφει τα χαρακτηριστικά του δικτύου από σταθερά μηχανήματα (desktop) τα οποία είναι συνδεδεμένα σε όλο το internet ή μικρές ασύρματες συσκευές ή συσκευές αισθητήρων οι οποίες είναι συνδεδεμένες σε ad-hoc δίκτυο.
Το *Overlay Nodes Management layer* καλύπτει το κομμάτι της διαχείρισης των peers, το οποίο περιλαμβάνει ανακάλυψη των peers και αλγόριθμοι δρομολόγησης για βελτιστοποίηση.
Το *Features Management layer* ασχολείται με την ασφάλεια, αξιοπιστία, ανασχηματισμό μετά από σφάλμα, και την συνολική διαθεσιμότητα των πόρων από την πλευρά της διατήρησης της ευρωστίας του P2P συστήματος.
Το *Services Specific layer* υποστηρίζει τα συγκεκριμένα συστατικά εφαρμογής μέσω των οποίων γίνεται η χρονοδρομολόγηση των παράλληλων και υπολογιστικά εντατικών δραστηριοτήτων, καθώς και την διαχείριση αρχείων. Μεταδομένα περιγράφουν το περιεχόμενο το οποίο αποθηκεύεται διαμέσου των peers καθώς και τις πληροφορίες τοποθεσίας.
Το *Application-level layer* ασχολείται με εργαλεία, εφαρμογές και υπηρεσίες οι οποίες υλοποιούνται με συγκεκριμένες λειτουργίες στην κορυφή του υποστρώματος υποδομής του P2P δικτύου .
Τέλος υπάρχουν δύο κλάσεις P2P overlay δικτύων, τα *δομημένα (Structured)* και *αδόμητα (Unstructured)*.

Ο όρος *δομημένα* σημαίνει ότι η τοπολογία του P2P overlay δικτύου είναι αυστηρά ελεγχόμενη και τα περιεχόμενα δεν τοποθετούνται τυχαία σε peers αλλά σε συγκεκριμένες τοποθεσίες έτσι ώστε επόμενα ερωτήματα να γίνονται πιο αποδοτικά. Τέτοια δομημένα P2P συστήματα χρησιμοποιούν Distributed Hash Table (DHT) ως υπόστρωμα (substrate) στο οποίο οι πληροφορίες τοποθεσίας των δεδομένων(data object) (ή τιμών) τοποθετούνται ντετερμινιστικά, στους peers με αναγνωριστικό το οποίο αντιστοιχεί στο μοναδικό κλειδί των δεδομένων. Τα βασισμένα σε DHT

συστήματα αναθέτουν με συνέπεια ομοιόμορφα τυχαία αναγνωριστικά (***NodeIDs***) στο σύνολο των peers από ένα μεγάλο χώρο από αναγνωριστικά.
Σε κάθε δεδομένο ανατίθεται μοναδικό αναγνωριστικό που ονομάζεται *κλειδί*, το οποίο επιλέγεται από τον ίδιο χώρο αναγνωριστικών. Τα κλειδιά διατάσσονται από το πρωτόκολλο overlay network σε ένα μοναδικό ενεργό peer στο overlay δίκτυο. Το P2P overlay δίκτυο υποστηρίζει την επεκτάσιμη αποθήκευση και ανάκτηση του ζευγαριού {**κλειδί, τιμή**} στο overlay δίκτυο, όπως εικονίζεται στο σχήμα 2.2 .



Σχήμα 2.2: Διεπαφή εφαρμογής για δομημένα DHT P2P overlay συστήματα. *[1]*

Δίνοντας ένα κλειδί η λειτουργία αποθήκευσης *(put(key,value))* ανατρέχει στη λειτουργία ανάκτησης *(value=get(key))* η οποία μπορεί να κληθεί και να αποθήκευση ή να επανάκτηση το δεδομένο που σχετίζεται με το συγκεκριμένο κλειδί. Η λειτουργία ανάκτησης εμπλέκει ρουτίνες κλήσεις στο peer που σχετίζεται με το κλειδί.

Κάθε peer διατηρεί ένα μικρό πίνακα δρομολόγησης ο οποίος αποτελείτε από NodeIDs γειτονικών peers και IP διευθύνσεις. Ερωτήματα αναζήτησης ή μηνύματα δρομολόγησης προωθούνται διαμέσου των overlay μονοπατιών, με μια προοδευτική μέθοδο, στους peers των οποίων τα NodeIDs είναι πιο κοντά στο κλειδί (στο χώρο αναγνωριστικών) του δεδομένου για το οποίο γίνεται η ερώτηση.

Στη θεωρεία τα DHT συστήματα μπορούν να εγγυηθούν ότι μπορεί να εντοπιστεί με μέσο ορό O(logN) overlay βήματα, δηλαδή βήματα στο overlay δίκτυο των peers, όπου N ο αριθμός των peers στο σύστημα. Όμως η διαδρομή στο δίκτυο υποστρώματος μεταξύ δύο peers (πχ το δίκτυο IP πάνω στο οποίο είναι δομημένο το overlay δίκτυο των peers) μπορεί να είναι σημαντική διαφορετική από τη διαδρομή στο DHT overlay δίκτυο. Ως αποτέλεσμα η καθυστέρηση αναζήτησης στο DHT P2P overlay δίκτυο μπορεί να είναι αρκετά υψηλότερη με δυσμενής αποτελέσματα στης εφαρμογές οι οποίες τρέχουν πάνω από αυτό. Το Plaxton προσφέρει ένα έξυπνο αλγόριθμο ο οποίος επιτυγχάνει σχεδόν βέλτιστη καθυστέρηση, όμως είναι απίθανο να επεκταθεί σε ένα μεγάλο αριθμό από peers.

Το 1999 το **Napster** πρωτοπόρησε με την ιδέα του Peer-to-Peer μοιραζόμενου συστήματος αρχείων το οποίο υποστηρίζει ένα κεντρικό σύστημα αναζήτησης αρχείων. Ήταν το πρώτο σύστημα το οποίο αναγνώριζε ότι αιτήσεις για δημοφιλή αρχεία δεν χρειαζόταν να στέλνονται σε ένα κεντρικό server αλλά θα μπορούσαν να χειρίζονται από πολλούς peers, οι οποίοι θα είχαν το αιτούμενο αρχείο.
Αυτά τα P2P μοιραζόμενων-αρχείων συστήματα είναι αυτό-κλιμακούμενα αφού όσο περισσότεροι peers ενώνονται στο σύστημα προσθέτουν στην αθροιστική download χωρητικότητα. Το Napster πέτυχε αυτό χρησιμοποιώντας ένα κεντρικό σύστημα αναζήτησης το οποίο βασίζεται στις λίστες αρχείων που παρέχει κάθε peer. Αυτό δεν απαιτεί υψηλό bandwidth για την αναζήτηση. Έχει όμως το μειονέκτημα της αποτυχίας όλου του συστήματος από την αποτυχία ενός μόνο σημείου λόγω της κεντρικής αναζήτησης.

Η **Gnutella** είναι ένα αποκεντρωμένο σύστημα το οποίο κατανέμει την ικανότητα της αναζήτησης και download σε όλους τους peers, εγκαθιδρύοντας έτσι ένα overlay δίκτυο από peers. Είναι το πρώτο σύστημα το οποίο χρησιμοποιεί το αδόμητο P2P overlay δίκτυο. Ένα *αδόμητο P2P* σύστημα συνθέτετε από peers οι οποίοι ενώνονται στο δίκτυο με κάποιους χαλαρούς δεσμούς, και χωρίς καμιά προηγούμενη γνώση της τοπολογίας του δικτύου. Το δίκτυο χρησιμοποιεί *πλημμύρα* ως το μηχανισμό για να στέλνει ερωτήματα διαμέσου του overlay δικτύου, με περιορισμένη εμβέλεια. Όταν ένας peer λάβει το ερώτημα από την πλημμύρα στέλνει μια λίστα, με όλα τα περιεχόμενα τα οποία ταιριάζουν στο ερώτημα, στον peer που έθεσε το ερώτημα. Αυτή η τεχνική είναι αποδοτική για εντοπισμό αντικειμένων για τα οποία υπάρχουν πολλά αντίγραφα, ενώ δεν ταιριάζει για αντικείμενα τα οποία είναι σπάνια. Επίσης αυτή η προσέγγιση δεν είναι επεκτάσιμη καθώς το φορτίο σε κάθε peer αυξάνει γραμμικά με το συνολικό αριθμό των ερωτημάτων και το μέγεθος του συστήματος.

Παρόλο που τα δομημένα P2P δίκτυα μπορούν αποτελεσματικά να εντοπίσουν σπάνια αντικείμενα αφού η ρουτίνα βασισμένη στο κλειδί είναι επεκτάσιμη, όμως προκαλούν σημαντικά ψηλότερη επιβάρυνση από ότι τα αδόμητα P2P δίκτυα για τα δημοφιλή αντικείμενα. Συνεπώς στο Internet σήμερα τα αδόμητα P2P δίκτυα είναι πιο διαδεδομένα.

2.1.2 Δομημένα P2P Overlay Δίκτυα:

Σε αυτή την κατηγορία, το overlay δίκτυο αναθέτει κλειδιά στα δεδομένα και οργανώνει τα peers του σε γράφο έτσι ώστε να προσδιορίζει κάθε κλειδί δεδομένου σε ένα peer. Αυτός ο δομημένος γράφος επιτρέπει αποδοτική ανακάλυψη των δεδομένων χρησιμοποιώντας το δοσμένο κλειδί. Παρόλο της απλής μορφής αυτή η κλάση συστημάτων δεν υποστηρίζει πολύπλοκα ερωτήματα. Παρακάτω παρουσιάζονται κάποια δομημένα P2P overlay Δίκτυα

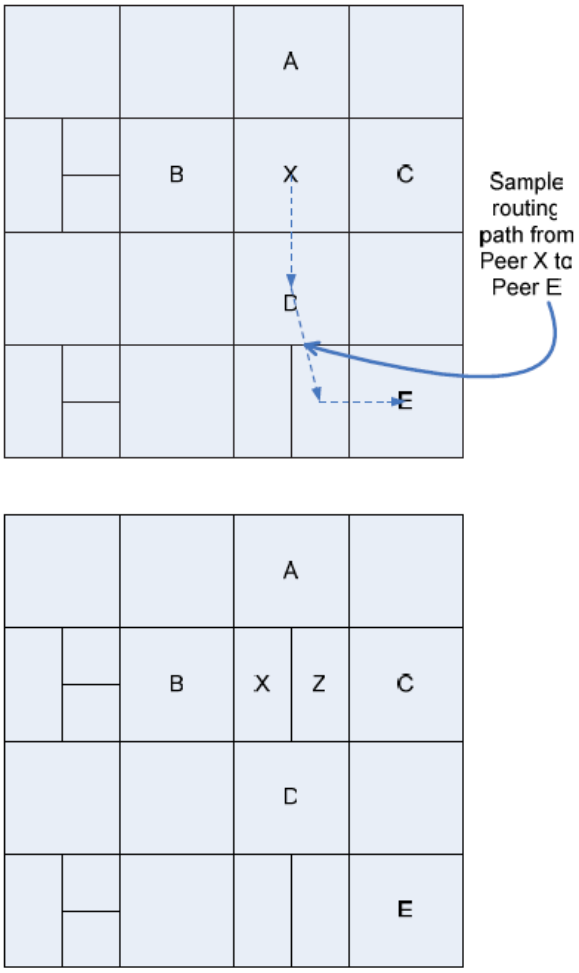
A. Content Addressable Network (CAN)

Το CAN είναι μια κατανεμημένη αποκεντρωμένη P2P υποδομή η οποία παρέχει λειτουργίες hash table πάνω σε κλίμακα internet. Είναι σχεδιασμένο έτσι ώστε να είναι επεκτάσιμο ανεκτικό στα σφάλματα και αυτό-οργανώσιμο.
Στο CAN υπάρχει ένας d διαστάσεων καρτεσιανός χώρος συντεταγμένων, σε ένα multi-torus, ο οποίος είναι απόλυτα λογικός. Ο χώρος αυτός μοιράζεται δυναμικά στους N peers του συστήματος έτσι ώστε κάθε peer να έχει το δικό του αποκλειστικό χώρο. Κάθε peer διατηρεί ένα πίνακα δρομολόγησης όπου κρατά τις IP διευθύνσεις των γειτονικών του peer στο χώρο συντεταγμένων. Τα μηνύματα CAN συμπεριλαμβάνουν και τις συντεταγμένες του παραλήπτη, έτσι οι peers χρησιμοποιώντας τις συντεταγμένες των γειτόνων τους, τις οποίες γνωρίζουν, προωθούν το μήνυμα στο γείτονα που είναι πιο κοντά στον παραλήπτη, χρησιμοποιώντας ένα απλό greedy αλγόριθμο δρομολόγησης. Για να φυλάξουμε ένα ζευγάρι {**κλειδί, τιμή**} το **κλειδί K** τοποθετείται ντετερμινιστικά σε ένα σημείο **P** στο χώρο συντεταγμένων χρησιμοποιώντας μια ομοιόμορφη hash συνάρτηση. Το πρωτόκολλο αναζήτησης μπορεί να ανακτήσει μια εγγραφή που σχετίζεται με το κλειδί K με τον εξής τρόπο, κάθε peer μπορεί να εφαρμόσει την ίδια συνάρτηση hash στο κλειδί K, και να εντοπίσει το σημείο P στο οποίο ανήκει, στην συνέχεια μπορεί να λάβει την **τιμή V** από το σημείο P. Η δρομολόγηση γίνεται μέσω των γειτονικών peers για τους οποίου κάθε peer γνωρίζει την IP διεύθυνση τους.

Όταν ένας νέος peer συνδέεται στο σύστημα έχει το δικό του μερίδιο στο χώρο συντεταγμένων το οποίο πρέπει να του παραχωρηθεί. Αυτό επιτυγχάνεται με το μοίρασμα του χώρου ενός άλλου peer, έτσι ώστε να παραχωρείται το μισό στο νέο peer και το υπόλοιπο να το κρατά ο παλιός. Το CAN έχει ένα σχετικό DNS domain name το οποίο δίνει την IP διεύθυνση ενός ή περισσοτέρων CAN Bootstrap peers. (οι οποίοι διατηρούν μια λίστα με μερικούς CAN peers). Όταν ένας νέος peer θα ενωθεί στο CAN δίκτυο τότε κοιτάζει στο DNS για το CAN domain και παίρνει την IP διεύθυνση ενός CAN Bootstrap peer. Ο Bootstrap peer δίνει τις IP διευθύνσεις κάποιων τυχαία επιλεγμένων peers του συστήματος . Ο νέος peer επιλέγει τυχαία ένα σημείο P και αφού πρώτα επικοινωνήσει με κάποιους peers για τους οποίους έλαβε προηγουμένως το IP τους από τον Bootstrap peer στέλνει μια αίτηση JOIN με προορισμό το σημείο P. Το μήνυμα στέλνεται από το peer που είχε προηγουμένως συνδεθεί μέσω του μηχανισμού δρομολόγησης του CAN. Όταν φτάσει το μήνυμα στο peer που βρίσκεται στο σημείο P, τότε αυτός μοιράζει το χώρο του στο μισό, παραχωρεί το ένα μισό στον καινούριο peer και κρατά το άλλο μισό. Επίσης τα ζεύγη {K,V} που αντιστοιχούν στο καινούριο peer μεταφέρονται στον καινούριο peer. Τέλος μαθαίνει από τον προηγούμενο κόμβο τις IP διευθύνσεις των γειτόνων του και καταχωρεί και τον προηγούμενο peer στο σύνολο των γειτόνων του.

Όταν ένας peer φεύγει από το CAN δίκτυο ένας αλγόριθμος αμέσως αναλαμβάνει τον έλεγχο έτσι ώστε να σιγουρέψει ότι κάποιος γείτονας του απελθόντος peer θα αναλάβει τη ζώνη που κατείχε. Ο peer τότε ανανεώνει το σύνολο τον γειτόνων του και απομακρύνει αυτούς που δεν είναι πλέων γείτονες του. Στη συνέχεια κάθε peer στο σύστημα πρέπει να στείλει μήνυμα κατάσταση ενημέρωσης έτσι ώστε όλοι οι γείτονες του να μάθουν για τις αλλαγές και να ανανεώσουν το σύνολο των γειτόνων τους. Το σύνολο το γειτόνων που κρατά κάθε peer εξαρτάτε μόνο από τις διαστάσεις του χώρου συντεταγμένων και είναι ανεξάρτητο από των αριθμό των peers στο σύστημα.

Το σύνολο γειτόνων του peer X είναι {A B C D}



Το νέο σύνολο γειτόνων του peer X είναι {A B D Z}
Το σύνολο γειτόνων του νέου peer Z είναι {A C D X}

Σχήμα 2.3: παράδειγμα από 2-d χώρο CAN πριν και μετά την ένωση του Z [1]

Το σχήμα 2.3 ένα απλό παράδειγμα δρομολόγησης από το peer X στο σημείο E, και την ένωση στο CAN δίκτυο ενός νέου peer Z. Για ένα χώρο d διαστάσεων χωρισμένο σε ίσες ζώνες το μέσω μήκος δρομολόγησης είναι $\frac{d}{4}n^{\frac{1}{d}}$ βήματα και κάθε peer διατηρεί μία λίστα από 2d γείτονες.

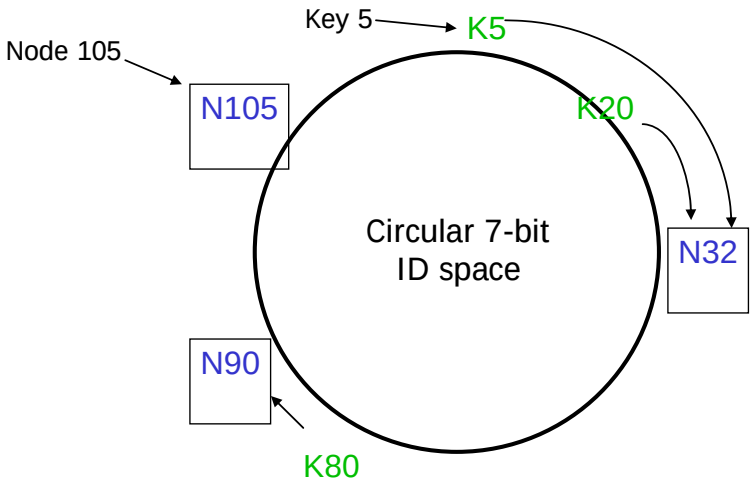
Επίσης το CAN έχει απόδοση δρομολόγησης $O(d.N^{\frac{1}{d}})$. Αφού υπάρχουν πολλά διαφορετικά μονοπάτια μεταξύ 2 σημείων, όταν ένας ή περισσότεροι γειτονικοί peers πέσουν ο peer θα μπορεί ακόμα να δρομολογεί μέσω του επόμενου καλύτερου μονοπατιού.

Βελτίωση στον αλγόριθμο του CAN μπορεί να γίνει διατηρώντας πολλαπλούς, ανεξάρτητους χώρους συντεταγμένων με κάθε peer στο σύστημα να αναλαμβάνει μια διαφορετική ζώνη σε κάθε χώρο συντεταγμένων ο οποίος ονομάζεται reality. Το CAN μπορεί να χρησιμοποιηθεί σε μεγάλης κλίμακας αποθηκευτικά συστήματα όπως OceanStore [8], Farsite [9], and Publius [10].

Αυτά τα συστήματα απαιτούν αποτελεσματική είσοδο και ανάκτηση των περιεχομένων σε ένα μεγάλο κατανεμημένο δίκτυο αποθήκευσης με επεκτάσιμο μηχανισμό δεικτοδότησης.

B. Chord [4]

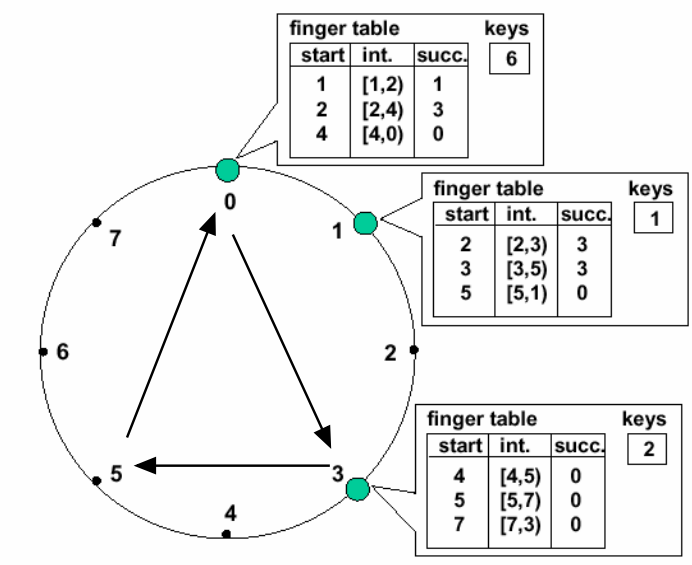
Το chord χρησιμοποιεί **συνεπές hashing** για να αναθέτει κλειδιά στους peers. Το αποκεντρωμένο σχήμα τείνει να ισοζυγήσει το φορτίο του συστήματος, αφού κάθε peer λαμβάνει περίπου τον ίδιο αριθμό από κλειδιά και υπάρχει μικρή μετακίνηση των κλειδιών όταν κάποιος peer συνδέεται ή αποχωρεί από το δίκτυο. Στο δίκτυο chord κάθε peer έχει το αναγνωριστικό του (***NodeID***) και κάθε δεδομένο ένα **κλειδί** τα οποία ανατίθενται από την συνεπής hash συνάρτηση και έχουν m-bit μήκος. Αυτά τα αναγνωριστικά βρίσκονται στον ίδιο χώρο σε ένα εύρος από 1 μέχρι 2^m. Δίνονται στους peer κάνοντας hash το IP του, ενώ αντίστοιχα στα δεδομένα μπορεί π.χ. να γίνει hash το όνομα. Αυτά τα αναγνωριστικά ταξινομούνται σε ένα κύκλο αναγνωριστικών με αρίθμηση από 0 μέχρι $2^m - 1$ ο οποίος ονομάζεται και δακτυλίδι chord. Κάθε κλειδί **K** τοποθετείτε μονοσήμαντα σε ένα peer, ο οποίος και ονομάζεται επιτυχών (***successor***) κόμβος για το συγκεκριμένο κλειδί, ως εξής: το K ανατίθεται στον πρώτο peer του οποίου το NodeID είναι ίσο ή ακολουθεί δεξιόστροφα το K στο χώρο αναγνωριστικών. Π.χ στο επόμενο παράδειγμα τα κλειδιά 5 και 20 ανατίθενται στο κόμβο 32 αφού αυτός ακολουθεί αμέσως μετά στο χώρο αναγνωριστικών, ενώ το κλειδί 80 στον κόμβο 90.



Σχήμα 2.4: ανάθεση κλειδιών [4]

Για να διατηρηθεί συνεπής τοποθέτηση των κλειδιών όταν ένας peer n συνδέεται στο δίκτυο όλα κλειδιά που είναι πριν από αυτό μέχρι και τον προηγούμενο peer και προηγουμένως είχαν ανατεθεί σε άλλο peer τώρα πρέπει να ξανά ανατεθούν στο καινούριο peer n. Επίσης όταν ένας peer εγκαταλείπει το δίκτυο όλα τα κλειδιά του πρέπει να ξανά ανατεθούν σε άλλο peer. Έτσι κάποιος peer συνδέεται ή εγκαταλείπει το σύστημα με απόδοση $(\log N)^2$. Για την δρομολόγηση των ερωτημάτων κάθε κόμβος διατηρεί ένα πίνακα ο οποίος ονομάζεται ***finger table*** και έχει m εγγραφές. Στην πραγματικότητα διατηρεί πληροφορίες για O(logN) peers όπου N ο αριθμός των peers. Η εγγραφή **i** του πίνακα στο peer **n** περιέχει τον peer **S** (NodeID, IP διεύθυνση

και άλλες σχετικές πληροφορίες) ο οποίος είναι επιτυχών για το συγκεκριμένο κλειδί $n+2^{i-1}$. Συγκεκριμένα έχουμε για κάθε εγγραφή του finger table, n.finger[k]= $(n+2^{i-1}) \bmod 2^m, 1 \le k \le m$. Με αυτό τον τρόπο οι peers αποθηκεύουν πληροφορίες για μικρό αριθμό peers και ξέρουν περισσότερα για peers οι οποίοι είναι μετά από αυτούς σε κοντινή απόσταση στον κύκλο αναγνωριστικών. Επίσης ο finger table δεν περιέχει αρκετή πληροφορίες για να μπορούμε απευθείας να καθορίσουμε τον successor ενός αυθαίρετου κλειδιού K. Τώρα όταν κάποιας peer θέλει εντοπίσει το successor του κλειδιού K, αν δεν έχει το ίδιο κλειδί K στο πίνακα του τότε στέλνει ένα lookup ερώτημα στο εγγραφή με το μεγαλύτερο κλειδί P που είναι μικρότερο από το K. δηλαδή στέλνει το ερώτημα στο επιτυχόντα peer του κλειδιού P. Η διαδικασία αυτή επαναλαμβάνεται, ώσπου το ερώτημα να φτάσει στον επιτυχών κόμβο s. η διαδικασία αυτή κοστίζει O(logN) hops όπου N ο αριθμός των peers.
Ακολουθεί ένα παράδειγμα όπου ο κόμβος 3 θέλει να βρει το επιτυχόντα του κλειδιού 6. Αφού δεν έχει το κλειδί 6 στο πίνακα του κάνει το ερώτημα στον επιτυχόντα του κλειδιού 5 το οποίο είναι το μεγαλύτερο κλειδί που υπάρχει στο πίνακα και προηγείται του 6. Ο επιτυχών peer του 5 είναι ο 0 ο οποίος έχει και το 6 και απαντά ανάλογα στο 3.



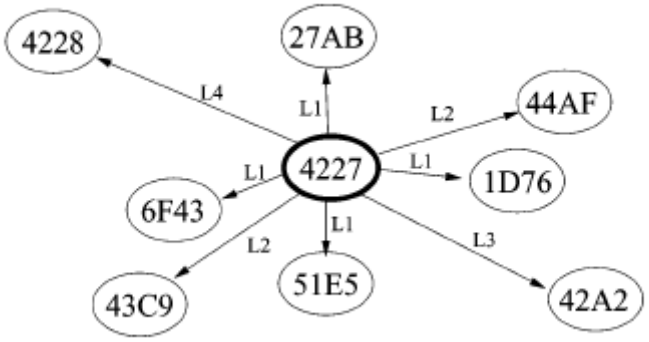
Σχήμα 2.5: δρομολόγηση [5]

Όταν ένας peer ενώνεται στο σύστημα, οι δείκτες που δείχνουν στους successor, σε μερικούς peers πρέπει να αλλαχθούν. Είναι σημαντικό οι successor δείκτες να είναι ενήμεροι και σωστοί σε κάθε στιγμή διότι αλλιώς δεν μπορεί να εγγυηθεί η ορθότητα της αναζήτησης. Το chord χρησιμοποιεί ένα πρωτόκολλο σταθεροποίησης το οποίο τρέχει περιοδικά στο background και ανανεώνει τους successor δείκτες και τις εγγραφές στο finger table. Η ορθότητα του πρωτοκόλλου chord βασίζεται στο γεγονός ότι κάθε peer είναι ενήμερος για τους successor του. Όταν ένας peer πέσει είναι πιθανό κάποιος άλλος peer να μην ξέρει τους νέους successor του, αφού θα έχουμε αλλαγή, και δεν έχει την ευκαιρία να μάθει για αυτό. Για να αποφύγουμε το γεγονός αυτό οι peers διατηρούν μια λίστα με successors μεγέθους r, η οποία περιέχει τους πρώτους r successor του peer. Όταν ένας successor peer δεν ανταποκρίνεται ο peer επικοινωνεί με τον επόμενο peer στη λίστα αυτή.

C. Tapestry

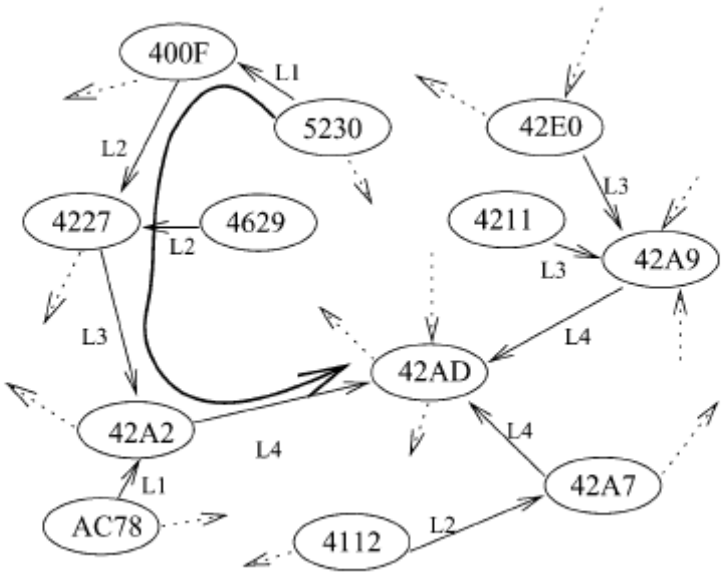
Το Tapestry έχει παρόμοια χαρακτηριστικά με το pastry, επίσης τυχαία αποκεντρωμένη τακτική για να επιτύχει κατανομή του φορτίου και δρομολόγηση. Η διαφορά μεταξύ pastry και Tapestry είναι το πώς χειρίζονται την τοποθεσία στο δίκτυο και τα αντίγραφα των δεδομένων. Η αρχιτεκτονική του Tapestry χρησιμοποιεί αρκετά από τη κατανεμημένη τεχνική αναζήτησης του Plaxton [7] με επιπρόσθετους μηχανισμούς για να παρέχει διαθεσιμότητα, επεκτασιμότητα, προσαρμογή στην παρουσία σφαλμάτων και επιθέσεων. Το Plaxton όπως και το Tapestry χρησιμοποιούν μια κατανεμημένη δομή δεδομένων το mesh. Στο Plaxton τα δεδομένα μπορούν να είναι ενωμένα σε ένα μόνο root peer ενώ στο Tapestry σε πολλούς. Στο Plaxton mesh οι peers μπορούν να παίζουν το ρόλο του server (έχει αποθηκευμένα δεδομένα) του client (δημιουργεί αιτήσεις) και router (προωθεί μηνύματα). Σε κάθε peer διατηρείτε τοπικός πίνακας δρομολόγησης έτσι ώστε τα μηνύματα να φτάνουν στο προορισμό τους (ID προορισμού) ταιριάζοντας σε κάθε βήμα ένα ψηφίο.

Στο παρακάτω σχήμα απεικονίζεται ένα παράδειγμα Tapestry mesh δρομολόγησης από την όψη ενός απλού κόμβου. (4227). Εξερχόμενοι γειτονικοί σύνδεσμοι δείχνου σε κόμβους με ένα κοινό πρόθεμα που ταιριάζει. Ψηλότερου επιπέδου σύνδεσμοι ταιριάζουν περισσότερα ψηφία, πχ στους συνδέσμου με L2 ταιριάζει 1 ψηφίο ενώ στου L3 2 ψηφία. Μαζί όλοι αυτοί οι σύνδεσμοι σχηματίζουν το τοπικό πίνακα δρομολόγησης.



Σχήμα 2.6:Tapestry mesh [6]

Χρησιμοποιώντας το τοπικό πίνακα δρομολόγησης τα μηνύματα δρομολογούνται στο ID του προορισμού ψηφίο με ψηφίο (π.χ. $4*** \Rightarrow 42** \Rightarrow 42A* \Rightarrow 42AD$ όπου * αντιπροσωπεύει τη wildcards). Ο $n_{ος}$ peer στον οποίο φτάνει το μήνυμα μοιράζεται πρόθεμα τουλάχιστο μήκους n με το ID προορισμού. Για να εντοπίσουμε το επόμενο peer to n+1 επίπεδο στο πίνακα δρομολόγησης εξετάζεται για να εντοπιστεί η εγγραφή η οποία ταιριάζει με την τιμή του επόμενου ψηφίου στο ID προορισμού. Αυτή η μέθοδος δρομολόγησης εγγυάται ότι οποιοσδήποτε υπάρχων peer στο δίκτυο μπορεί να εντοπιστεί με το πολύ $\log_B N$ λογικά βήματα στο overlay δίκτυο, με N των αριθμό των peers οι οποίοι χρησιμοποιούν NodeID με βάση το B.



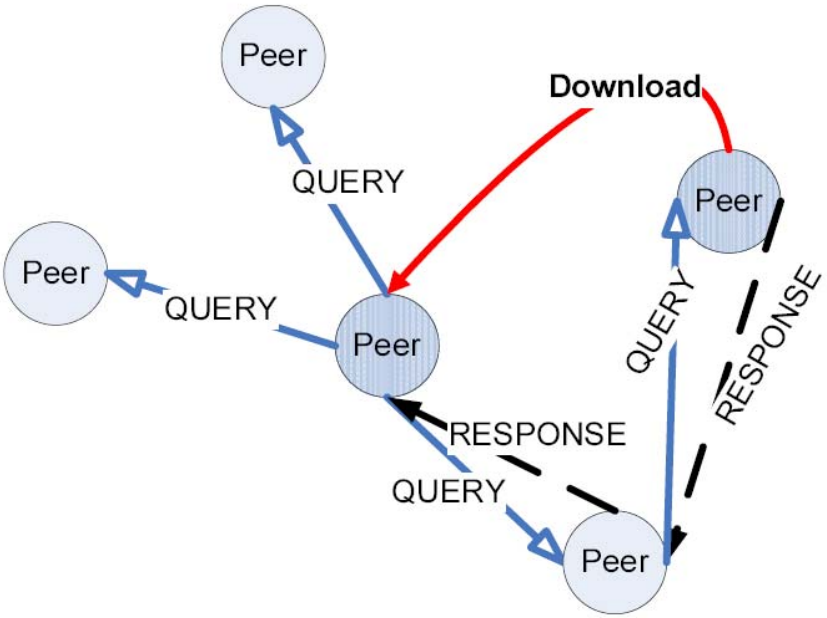
Σχήμα 2.7: Διαδρομή ενός μηνύματος. Το μήνυμα ξεκινά από τον κόμβο 5230 και έχει προορισμό τον κόμβο 42AD. Στο βήμα η τουλάχιστο πρόθεμα η ψηφίων ταιριάζει με το ID του προορισμού. [6]

Στο παράρτημα Α στον πίνακα 1 παρουσιάζονται περιληπτικά τα χαρακτηριστικά κάποιων δομημένων P2P overlay δικτύων

2.1.3 Αδόμητα P2P Overlay Δίκτυα

Σε αυτή την κατηγορία οι peers οργανώνονται στο overlay δίκτυο σε ένα τυχαίο γράφο με ένα επίπεδο ή ιεραρχικό τρόπο και χρησιμοποιούν πλημμύρα ή τυχαίους διαδρόμους κ.α. στο γράφο για να αναζητήσουν περιεχόμενα που είναι αποθηκευμένα στους peers. Αυτό έχει ως αποτέλεσμα η αναζήτηση για σπάνια δεδομένα να είναι αναποτελεσματική. Παρακάτω παρουσιάζονται μερικά από τα πιο κοινά αδόμητα P2P overlay δίκτυα.

A. Gnutella



Σχήμα 2.8: Το Gnutella χρησιμοποιεί μια αποκεντρωμένη αρχιτεκτονική για εντοπισμό και ανάκτηση δεδομένων. [1]

Το Gnutella είναι ένα αποκεντρωμένο πρωτόκολλο για κατανεμημένη αναζήτηση σε μια επίπεδη τοπολογία από peers (servents). Έχει χρησιμοποιηθεί ευρέως και υπάρχει πολύ δουλεία για την βελτίωση του. Κάθε peer είναι server και client και όπως φαίνεται στο σχήμα 2.8 χρησιμοποιεί μια αποκεντρωμένη αρχιτεκτονική για εντοπισμό και ανάκτηση δεδομένων. Σε αυτό το σύστημα δεν υπάρχει κεντρικός κατάλογος για αναζήτηση και ούτε διατηρεί κάποιο ακριβές έλεγχο στη τοπολογία του δικτύου και στην τοποθέτηση των αρχείων. Το δίκτυο σχηματίζεται από peers συνδέονται ακολουθώντας κάποιους χαλαρούς κανόνες. Η σχηματιζόμενη τοπολογία έχει κάποιες συγκεκριμένες ιδιότητες αλλά η τοποθέτηση των δεδομένων δεν βασίζεται σε κάποια γνώση της τοπολογίας όπως στα δομημένα P2P δίκτυα.

Η ποιο τυπική μέθοδος ερώτησης είναι η πλημμύρα όπου ο peer ρωτά τους γείτονες τους και αυτοί τους δικούς τους και άρα το ερώτημα πλημμυρίζεται σε όλους τους γείτονες με μια συγκεκριμένη όμως ακτίνα. Ένας τέτοιο δίκτυο μπορεί πολύ εύκολα να ανασχηματιστεί μετά από την εισαγωγή ή απομάκρυνση κάποιων peers από το

σύστημα. Όμως αυτός ο μηχανισμός αναζήτησης δεν είναι επεκτάσιμος και δημιουργεί μη προβλέψιμο φορτίο στο δίκτυο.

Οι peers εκτελούν εργασίες που είναι συνήθως συσχετισμένες και με client και server, παρέχοντας κάποια διεπαφή χρήστη (interface) για εκτέλεση ερωτημάτων από το χρήστη και επίσης δέχονται ερωτήματα από άλλους peers. Επίσης είναι υπεύθυνοι για την διαχείριση της κίνησης που εκτελείτε στο παρασκήνιο για την διασπορά της πληροφορίας έτσι ώστε να διατηρείτε το δίκτυο ολοκληρωμένο. Επίσης το πρωτόκολλο είναι υψηλά ανεκτικό σε σφάλματα καθώς οι λειτουργείς τους δικτύου δεν θα διακοπούν αν κάποια ομάδα από peers πάει offline.

Για να εισαχτεί στο σύστημα ένας νέος peer αρχικά ενώνεται σε κάποιο από τους πολλούς γνωστούς κόμβους οι οποίοι είναι σχεδόν πάντα διαθέσιμοι (π.χ. λίστα με τους peers μπορεί να βρεθεί στο `http://gnutellahosts.com`). Όταν ενωθεί στο δίκτυο οι peers στέλνουν μηνύματα αναμεταξύ τους. Αυτά τα μηνύματα είναι *broadcasted* (π.χ. στέλνονται σε όλους τους peers στους οποίους ο αποστολέας διατηρεί ανοικτή TCP σύνδεση) ή απλά είναι *προς τα πίσω διαδιδόμενα* (π.χ. στέλνεται σε μια συγκεκριμένη σύνδεση στην αντίθετη κατεύθυνση από την οποία πάρθηκε το αρχικό broadcasted μήνυμα). Πρώτο κάθε μήνυμα έχει ένα τυχαίο αναγνωριστικό. Δεύτερο, κάθε peer διατηρεί μια μικρή μνήμη με πρόσφατα δρομολογημένα μηνύματα, αποτρέποντας έτσι την ξανά αποστολή broadcasted μηνυμάτων και έτσι εφαρμόζει την προς τα πίσω διάδοση. Τρίτον, τα μηνύματα σημειώνονται με τα πεδία TTL (Time To Live) και τα βήματα που έχει κάνει.
Τα μηνύματα τα οποία επιτρέπονται στο δίκτυο είναι:

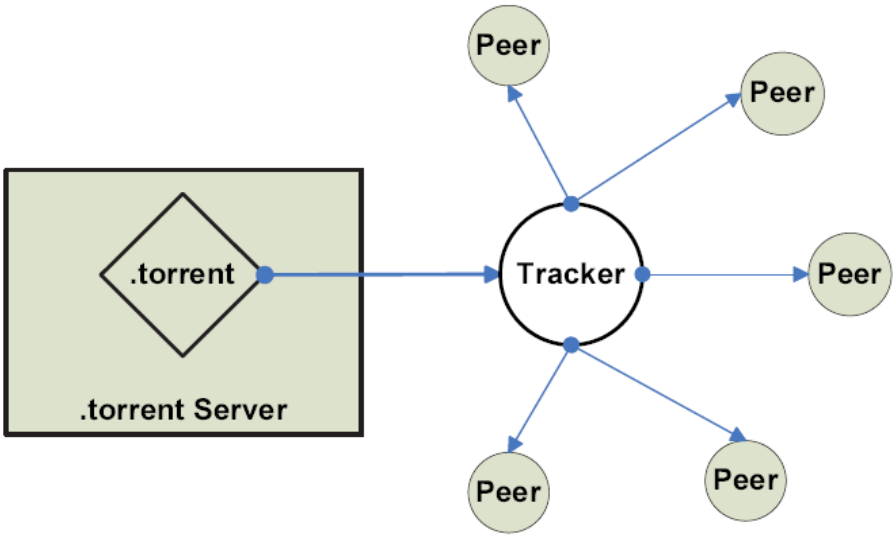
-*Μέλος ομάδας (PING and PONG) μηνύματα.* Ένας peer ο οποίος συνδέεται στο δίκτυο αρχικοποιεί ένα broadcasted μήνυμα PING για να ανακοινώσει την παρουσία του. Το μήνυμα αυτό στη συνέχεια στέλνεται στους γείτονες του και αρχικοποιούν ένα προς τα πίσω broadcasted μήνυμα PONG , το οποίο περιέχει πληροφορίες για τον peer όπως IP διεύθυνση, αριθμό και μέγεθος των δεδομένων.
-*Αναζήτησης (Ερώτημα και Απάντηση) μηνύματα.* Ένα ερώτημα περιέχει ένα string αναζήτησης το οποίο καθορίζεται από τον χρήστη, και κάθε peer ο οποίος λαμβάνει το μήνυμα αυτό πρώτα προσπαθεί να ταιριάσει το string αυτό με τα ονόματα των αρχείων τα οποία έχει ο ίδιος τοπικά φυλαγμένα και στη συνέχεια το κάνει broadcast. Οι απαντήσει είναι προς τα πίσω διαδιδόμενα προς τα ερωτήματα και περιλαμβάνουν απαραίτητες πληροφορίες για να γίνει download το αρχείο.
-*Μεταφοράς αρχείου (GET και PUSH) μηνύματα* . Το αρχείο γίνεται download απευθείας μεταξύ των δύο peers μέσω αυτών των δύο μηνυμάτων.

Έτσι για να γίνει μέλος τους δικτύου ένας peer απλά χρειάζεται να ανοίξει μια ή περισσότερες συνδέσεις με άλλους peers οι οποίοι είναι ήδη στο δίκτυο. Για την αντιμετώπιση της αναξιοπιστίας σε ένα τέτοιο δυναμικό δίκτυο, μετά την σύνδεση του ο peer περιοδικά στέλνει μηνύματα PING στους γείτονες του για να ανακαλύψει και άλλους peers. Οι peers αποφασίζουν που θα συνδεθούν στο δίκτυο με βάση τοπικών πληροφοριών. Έτσι όλο το δίκτυο επιπέδου εφαρμογής έχοντας τους peer και TCP συνδέσεις μεταξύ τους, σχηματίζει ένα δυναμικό αυτό οργανωμένο δίκτυο από ανεξάρτητες οντότητες.

B. BitTorrent

BitTorrent είναι ένα κεντρικοποιημένο P2P σύστημα το οποίο χρησιμοποιεί μια κεντρική τοποθεσία για την διαχείριση των downloads των χρηστών. Αυτό το δίκτυο κατανομής αρχείων χρησιμοποιεί ***tit-for-tat*** (ο peer ανταποδίδει με τον ίδιο τρόπο που οι άλλοι συνεργαζόμενοι peers συμπεριφέρθηκαν πριν). Οι peers με ψηλή upload ταχύτητα θα είναι πιθανών ικανοί να κατεβάσουν με μεγάλη ταχύτητα, και έτσι επιτυγχάνουμε υψηλή χρησιμοποίησης του bandwidth. Η ταχύτητα κατεβάσματος ενός peer θα μειωθεί αν η ταχύτητα ανεβάσματος έχει περιοριστεί. Αυτό θα εξασφαλίσει ότι το περιεχόμενο θα διασκορπιστεί σε όλους τους peers και θα αυξήσει την αξιοπιστία.

Η αρχιτεκτονική αποτελείτε από μια κεντρική τοποθεσία η οποία είναι ένας tracker και από τον οποίο κατεβάζεις ένα *.torrent* αρχείο, το οποίο περιέχει πληροφορίες για το αρχείο που θέλεις να καταβάσεις, όπως μήκος, όνομα πληροφορίες hashing και URL του tracker όπως φαίνεται και στο σχήμα 2.9.



Σχήμα 2.9: BitTorrent αρχιτεκτονική που αποτελείτε από τον κεντρικό tracker και το .torrent αρχείο [1]

Ο tracker διατηρεί ίχνος από όλους τους peers οι οποίοι έχουν το αρχείο (μερικό ή ολόκληρο) και αναζητά peers για να ενωθούν ο ένας με τον άλλο για κατέβασμα και ανέβασμα. Ο tracker χρησιμοποιεί ένα απλό πρωτόκολλο πάνω από το HTTP στο οποίο αυτός που θέλει να κατεβάσει ένα αρχείο (downloader) στέλνει πληροφορίες για αυτό και τον αριθμό της θύρας. Ο tracker ανταποκρίνεται με μια τυχαία λίστα με πληροφορίες επικοινωνίας για τους peers οι οποίοι κατεβάζουν το ίδιο αρχείο. Αυτοί που κατεβάζουν τότε το αρχείο χρησιμοποιούν αυτές τις πληροφορίες για να ενωθούν μεταξύ τους. Αυτός που έχει ολόκληρο το αρχείο ονομάζεται *seed* , και αρχικά τουλάχιστον ένας από αυτούς πρέπει να ξεκινήσει και να στείλει τουλάχιστον ένα αντίγραφο του αυθεντικού αρχείου.

Το BitTorrent κόβει τα αρχεία σε κομμάτια σταθερού μεγέθους (256 Kbytes) έτσι ώστε να μπορεί να παρακολουθεί το περιεχόμενο κάθε peer. Κάθε downloader peer ανακοινώνει σε όλους τους υπόλοιπους peers του τα κομμάτια τα οποία έχει, και

χρησιμοποιεί SHA1 για να κάνει hash το όλα τα κομμάτια τα οποία είναι περιλαμβάνονται στο *.torrents* αρχείο. Όταν ένας peer τελειώσει το κατέβασμα ενός κομματιού και ελέγξει ότι το hash ταιριάζει τότε ανακοινώνει ότι έχει το κομμάτι αυτό σε όλους τους υπόλοιπους peers του. Αυτό είναι για επαλήθευση της ακεραιότητας των δεδομένων. Επίσης οι συνδέσεις των peers είναι συμμετρικές. Αιτήσεις που δεν μπορούν να γραφτούν απευθείας στο TCP buffer παίρνουν ουρά στη μνήμη παρά στο application-level buffer του δικτύου, έτσι ώστε να μπορούν να απορριφτούν όταν ένα *choke* συμβεί.

Choking είναι μια προσωρινή άρνηση για upload. Το download μπορεί να συνεχίζεται κανονικά και η σύνδεση δεν χρειάζεται ανανέωση όταν το choking σταματά. Το choking γίνεται για πολλούς λόγους όπως: Όταν ο έλεγχος συμφόρησης του TCP συμπεριφέρεται πολύ φτωχά διότι στέλνουμε πολλές συνδέσεις ταυτόχρονα. Επιπρόσθετα το choking επιτρέπει σε κάθε peer να χρησιμοποιεί *tit-for-tat* αλγόριθμο για να εξασφαλίσει ότι έχει ένα σταθερό ρυθμό κατεβάσματος. Έχει αρκετά κριτήρια που πρέπει αν πλήρη ένας καλός αλγόριθμος choking. Πρέπει να περιορίζει τον αριθμό των ταυτόχρονων upload για καλή TCP απόδοση. Πρέπει να αποτρέπει το γρήγορα choking και unchoking, γνωστό ως ***fibrillation***. Τέλος πρέπει να δοκιμάζει αχρησιμοποίητες συνδέσεις καθώς μπορεί να είναι καλύτερες από τις τρέχων που χρησιμοποιούνται, γνωστό και ως οπτιμιστικό unchoking.

Στο παράρτημα Α στον πίνακα 2 συνοψίζονται οι συγκρίσεις για διάφορα αδόμητα P2P δίκτυα.

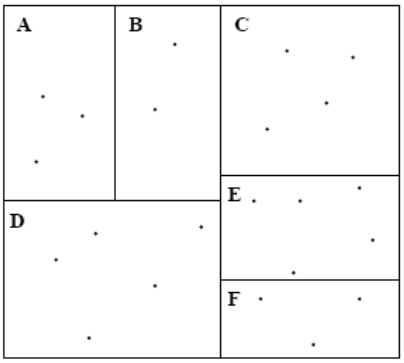
2.2 SkipIndex

Σε αυτή την ενότητα περιγράφουμε το *SkipIndex*[2] P2P σύστημα δεικτοδότησης για δεδομένα πολλών διαστάσεων και το οποίο έχουμε υλοποιήσει εν μέρη. Παραθέτουμε όμως και άλλες ενότητες του που αφορούν γενικότερα τα P2P για θέμα πληρότητας.

2.2.1 Εισαγωγή

Η συνεχής αύξηση της πυκνότητας αποθήκευσης καθώς και η συνεχής βελτίωση των συνδέσεων ευρείας ζώνης έχουν εκτοξεύσει την δημοτικότητα των τεράστιων και κατανεμημένων συλλογών δεδομένων. Μερικά από αυτά τα σύνολα δεδομένων μπορεί να είναι τέτοιας τεράστιας κλίμακας έτσι ώστε ακόμη και η δεικτοδότηση τους να υπερβαίνει την αποθηκευτική και επεξεργαστική ισχύ ενός απλού κόμβου. Η γεωγραφική κλίμακα τέτοιων κατανεμημένων συστημάτων μπορεί να εκτείνεται από ένα ζευγάρι συστοιχιών τα οποία είναι ενωμένα με ένα τοπικό δίκτυο σε ένα δωμάτιο μέχρι peer-to-peer συστήματα αποθήκευσης που εμπλέκουν πολλούς χρήστες σε μία ολόκληρη ήπειρο. Λόγω αυτής της τεράστιας κατανεμημένης κλίμακας, χρειαζόμαστε αποτελεσματικούς μεθόδους για να χειριζόμαστε την κατανεμημένη δεικτοδότηση. Στόχος είναι η δημιουργία ενός σχήματος δεικτοδότησης το οποίο να πληρεί τις παρακάτω απαιτήσεις:

- *Αποτελεσματική υποστήριξη για όμοιες αναζητήσεις και ευρεία ερωτήματα για δεδομένα σε χώρο υψηλών διαστάσεων:* Θέλουμε να μπορούμε εύκολα να βρίσκουμε σύνολα από αντικείμενα τα οποία είναι όμοια με το αντικείμενο στόχος το οποίο έχει καθοριστεί σε ένα χώρο υψηλών διαστάσεων (***όμοιες αναζητήσεις***). Επίσης θέλουμε να εκτελούμε ***ευρεία ερωτήματα*** τα οποία βρίσκουν αντικείμενα τα οποία περιέχονται σε μια περιοχή, η οποία έχει καθοριστεί από το ερώτημα, ενός χώρου υψηλών διαστάσεων. Αυτές οι λειτουργίες χρησιμοποιούνται ευρέως από πολλές εφαρμογές. Υπάρχων λύσεις όπως ο κατανεμημένος πίνακας κατακερματισμού (distributed hash table), ο οποίος χρησιμοποιεί hashing για την τοποθέτηση των αντικειμένων δεν είναι κατάλληλος για υποστήριξη πολύπλοκων ερωτημάτων όπως όμοιες αναζητήσεις και ευρεία ερωτήματα.
- *Υποστήριξη για πολλαπλή δεικτοδότηση:* Η κατανεμημένη υποδομή πρέπει να είναι ικανή να υποστήριξη πολλαπλά σύνολα δεδομένων, με διαφορετικές διαστάσεις και κατανομές το καθένα. Το σύστημα πρέπει να μπορεί να υποστηρίξει ένα μεγάλο αριθμό από μικρά σύνολα δεδομένων τα οποία το καθένα να είναι απλωμένο σε ένα μικρό αριθμό από κόμβους.
- *Ισοζύγιο φορτίου:* Κάθε κόμβος ο οποίος αποθηκεύει ένα μέρος της δεικτοδότησης πρέπει να είναι υπεύθυνος χονδρικά για τον ίδιο αριθμό από αντικείμενα. Η κατανομή των αντικειμένων σε ένα χώρο υψηλών διαστάσεων μπορεί να είναι πολύ ανομοιόμορφη όπως φαίνεται και στο σχήμα 2.10.



Σχήμα 2.10 [2]

- *Τοπικότητα:* δείκτες για όμοια ή σχετικά δεδομένα πρέπει να συγκεντρώνονται σε ένα μικρό αριθμό από κόμβους, επιτρέποντας έτσι την εκτέλεση ευρέων ή όμοιων ερωτημάτων και τον περιορισμό των κόμβο που επηρεάζονται από αυτά τα ερωτήματα. Έτσι βελτιώνεται ο παραλληλισμός των υποστηριζόμενων πολλαπλών ανεξάρτητων ερωτημάτων.
- *Υποστήριξη για την επεκτασιμότητα του επικαλύπτων (overlay) δικτύου:* Το σχήμα δεικτοδότησης πρέπει αν αναπτυχθεί πάνω από ένα P2P overlay δίκτυο μεγάλης κλίμακας. Όμως κάθε κόμβος πρέπει να διατηρεί μια μικρού μεγέθους κατάσταση (π.χ. ένας περιορισμένος αριθμός από δείκτες). Ενώ θα επιτρέπεται να τίθεται ερώτημα από οποιοδήποτε κόμβο και να προωθείται γρήγορα στο κόμβο στόχο ο οποίος και θα απαντά το ερώτημα. Επιπρόσθετα οι κόμβοι πρέπει να είναι ελεύθεροι να συνδέονται και να αποχωρούν από το overlay δίκτυο χωρίς να διακόπτουν την λειτουργία πολλών εναπομεινάντων κόμβων.

Παρακάτω εισάγεται το σύστημα δεικτοδότησης ***SkipIndex***, το οποίο περιέχει τους ακόλουθους μηχανισμούς:

(1) Το σύστημα χωρίζει το χώρο αναζήτησης σε ένα ιεραρχικό δένδρο, και οργανώνει τα φύλλα χρησιμοποιώντας SkipGraph-based κατανεμημένη δομή δεδομένων. Υποστηρίζει *όμοιες αναζητήσεις* και *ευρεία ερωτήματα* για δεδομένα σε χώρο υψηλών διαστάσεων και επίσης απαιτεί λογαριθμικό αριθμό από δείκτες σε peers (ως κατάσταση δεικτοδότησης σε κάθε peer) καθώς και λογαριθμικό αριθμό από overlay βήματα, μέχρι ένα ερώτημα να φτάσει στο προορισμό του.

(2) Δίνεται η δυνατότητα στο χρήστη να καθορίσει το επιθυμητό επίπεδο της ακρίβειας της αναζήτησης, και το σύστημα ελέγχει τον αριθμό των κόμβων τους οποίους θα ερωτήσει έτσι ώστε να ικανοποιεί το όριο λάθους.

(3) Διάφορα σύνολα δεδομένων με διαφορετικές διαστάσεις και κατανομές μπορούν να είναι αποθηκευμένα στην κατανεμημένη υποδομή την ίδια στιγμή.

(4) Το σύστημα επιτρέπει το δυναμικό διαχωρισμό του χώρου υψηλών διαστάσεων σε όλους του κόμβους που λαμβάνουν μέρος, με ένα τρόπο που διατηρεί το ισοζύγιο φορτίου.

2.2.2 Κατανεμημένη Οργάνωση Δεικτοδότησης

Σε μια κατανεμημένη υποδομή δεικτοδότησης κάθε κόμβος διατηρεί δείκτες για ένα υποσύνολο από τα κλειδιά που είναι αποθηκευμένα στο σύστημα την τρέχων χρονική στιγμή. Επίσης κάθε κόμβος διατηρεί ένα μικρό αριθμό από διευθύνσεις προς άλλους κόμβους σχηματίζοντας έτσι ένα overlay mesh. Η κατασκευή του συστήματος δεικτοδότησης καθώς και η επεξεργασία των ερωτημάτων πρέπει να εκτελούνται με ένα εντελώς αποκεντρωμένο τρόπο. Ακολούθως παρουσιάζονται και συγκρίνονται με το SkipIndex εναλλακτικοί υπάρχων τρόποι για διαμοιρασμό του χώρου αναζήτησης και αναζήτηση σε αυτό.

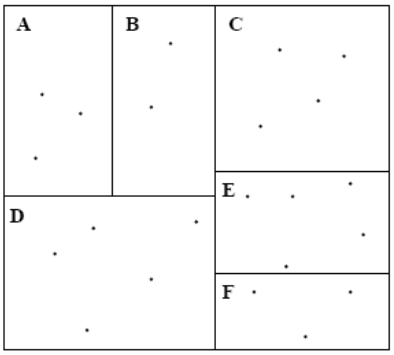
- ***Μέθοδοι διαμοιρασμού του χώρου αναζήτησης.***

- Content Addressable Networks (CAN)

Το CAN περιγράφεται σε προηγούμενη ενότητα (2.1.2A), εδώ απλά τονίζεται το γεγονός ότι η συνάρτηση hash που χρησιμοποιείται για την χαρτογράφηση των κλειδιών σε περιοχές του χώρου καταστρέφει τη λογική ολοκλήρωση του χώρου κλειδιών. Δηλαδή γίνεται δύσκολο για αποτελεσματική υποστήριξη πολύπλοκων όμοιων αναζητήσεων και ευρέων ερωτημάτων, αφού π.χ. όμοια αντικείμενα δεν είναι συγκεντρωμένα σε μια μικρή περιοχή του χώρου, δηλαδή σε λίγους γειτονικούς κόμβους, αλλά διάσπαρτα σε ολόκληρο το χώρο. Μια εναλλακτική στρατηγική για υποστήριξή αυτών των ερωτημάτων είναι να μην χρησιμοποιείται η hash συνάρτηση για χαρτογράφηση των κλειδιών αλλά αντιθέτως τα κλειδιά να χαρτογραφούνται σε σημεία στο χώρο βασισμένα σε μια απλή συνάρτηση η οποία διατηρεί την λογική σειρά των κλειδιών. Βέβαια η δυνατότητα υποστήριξης πολύπλοκων ερωτημάτων με αυτή τη μέθοδο έχει το κόστος της το οποίο είναι, ότι κάποιοι κόμβοι οι οποίοι θα τυγχάνει να έχουν πολύ δημοφιλές ζώνες θα αντιμετωπίζουν ψηλό αποθηκευτικό κόστος και βαρύ φορτίο ερωτημάτων ενώ άλλοι κόμβοι θα είναι σχεδόν αδρανείς, άρα ανατρέπεται το ισοζύγιο φορτίου.

- Απλό CAN

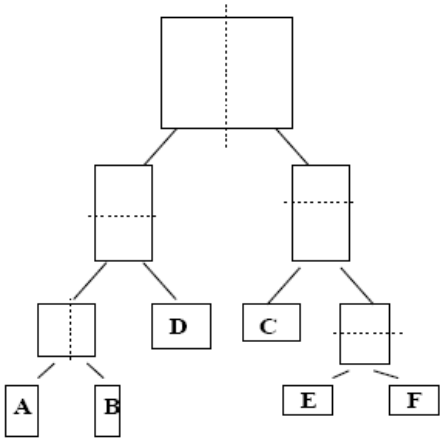
Για την αντιμετώπιση του προβλήματος με το ισοζύγιο φορτίου που υπάρχει στο CAN το οποίο χρησιμοποιεί unhashed κλειδιά εισάγεται η παρακάτω απλή μέθοδος, *pSearch*, για την ανάθεση περιοχών του χώρου στους κόμβους. Ένας συνδεόμενος κόμβος αντί να παίρνει ένα τυχαίο σημείο του χώρου για την επιλογή της περιοχής του, να διαλέγει ένα κλειδί από κάποια αντικείμενα τα οποία υπάρχουν ήδη στο σύστημα, στη συνέχεια λειτουργεί όπως και προηγουμένως. Με αυτό τον τρόπο πυκνές με αντικείμενα περιοχές είναι πιο πιθανών να χωριστούν, από τους νεοεισερχόμενους κόμβους και η κατανομή των δεδομένων θα είναι πιο ισοζυγισμένη σε σύγκριση με την περίπτωση που το σημείο επιλέγεται τυχαία. Όμως αφού ο συνδεόμενος κόμβος χρησιμοποιεί μόνο ένα κλειδί αυτό πιθανών να μην αντικατοπτρίζει την ακριβή κατανομή των δεδομένων.



Το πλέγμα στο παραπάνω σχήμα σχηματίστηκε χρησιμοποιώντας αυτή την τεχνική. Το ανισοζύγιο φορτίου είναι ορατό σε πολλά κελιά!

➢ Ισοζυγισμένη Ανάθεση Περιοχών

Αυτή η τεχνική προσπαθεί να πετύχει καλύτερα αποτελέσματα, από την προηγούμενη μέθοδο, στο ισοζύγιο φορτίου. Αρχικά υπάρχει ένας μόνο κόμβος στο σύστημα ο οποίος ελέγχει και όλο το χώρο κλειδιών. Καθώς προστίθενται και άλλα κλειδιά στο σύστημα όταν ο αριθμός τους ξεπεράσει το κατώφλι του αριθμού που πρέπει να ανατίθενται σε κάθε κόμβο, ο αρχικός κόμβος μοιράζει το χώρο σε δύο μη επικαλυπτόμενες περιοχές με ίδιο αριθμό κλειδιών σε κάθε μια. Ο χώρος μοιράζεται στην διάσταση με την μεγαλύτερη πυκνότητα. Η μια περιοχή δίνεται σε ένα ανενεργό κόμβο. Η διαδικασία διαμοιρασμού συνεχίζεται σχηματίζοντας ένα δυαδικό δένδρο. Το παρακάτω σχήμα παρουσιάζει ένα απλοποιημένο παράδειγμα από ένα 2-διαστάσεων χώρο ο οποίος είναι διαμοιρασμένος σε 6 κόμβους.



Σχήμα 2.11 [2]

➢ ***Ιεραρχική οργάνωση διαμοιρασμού***

Οι ιεραρχικές πολυδιάστατες δενδρικές δομές (όπως K-D tree , R-tree , R*-tree , X-tree) έχουν λάβει σημαντική έρευνα στη κοινότητα των βάσεων δεδομένων. Αυτές οι δομές ιεραρχικά χωρίζουν το χώρο αναζήτησης και τα σύνολα δεδομένων σε μικρότερες και μικρότερες περιοχές. Λειτουργίες εισαγωγής και αναζήτησης

πλοηγούνται από την ρίζα του δένδρου προς τον κατάλληλο κόμβο φύλλο. Παρακάτω εξετάζουμε αν μπορούν αυτές οι δομές να προσαρμοστούν για την κατασκευή αποτελεσματικού κατανεμημένου συστήματος δεικτοδότησης.

➢ Κατανεμημένο Δένδρο Αναζήτησης

Μπορούμε να δημιουργήσουμε κατανεμημένη υποδομή δεικτοδότησης κατανέμοντας μέρη του ιεραρχικού δένδρου αναζήτησης σε διαφορετικούς επεξεργαστές (κόμβους). Οι κόμβοι του δένδρου οι οποίοι αναπαριστούν περιοχές του χώρου πρέπει να είναι κατανεμημένα αντικείμενα τα οποία περιέχουν δείκτες σε άλλους κόμβους. Μη τοπικοί δείκτες του δέντρου πρέπει να αναπαρίσταται ως παγκόσμιες διευθύνσεις που καθορίζονται από το ID του επεξεργαστή καθώς και από την τοπική διεύθυνση του μηχανήματος στον οποίο βρίσκετε ο κόμβος. Η πλοήγηση στο δένδρο για ερωτήματα και εισαγωγές απαιτεί ανταλλαγή μηνυμάτων μεταξύ των κόμβων. Η θεμελιώδης διαφορά μεταξύ του κατανεμημένου δένδρου αναζήτησης και του CAN συστήματος είναι στο πως εκτελείτε η δρομολόγηση στα διάφορα ερωτήματα. Στα κατανεμημένου δένδρου αναζήτησης όλες οι λειτουργίες ακολουθούν τους ιεραρχικούς δείκτες του δένδρου για να φτάσουν στον κόμβο στόχο του δένδρου. Αντιθέτως στα CAN συστήματα εκτελείτε δρομολόγηση στο καρτεσιανό χώρο για να φτάσουμε στην περιοχή στόχο.

➢ Κατανεμημένο Δένδρο Αναζήτησης χρησιμοποιώντας DHT

Ένας κατανεμημένος πίνακας hash (DHT) διαμοιράζει τα αντικείμενα με τέτοιο τρόπο ώστε να διατηρεί το ισοζύγιο φορτίου και επιτρέπει αποτελεσματική αναζήτηση των αντικειμένων με βάση το ID τους. Χρησιμοποιώντας DHT για πρόσβαση στα αντικείμενα ενός δένδρου αναζήτησης τότε το σύστημα θα αναφέρεται στα αντικείμενα του δένδρου με κάποια μοναδικά ID παρά με τις καθολικές διευθύνσεις, δηλαδή την καθολική διεύθυνση του peer (κόμβου) στον όποιο βρίσκεται. Έτσι τα αντικείμενα μπορούν διαφανή να μεταφέρονται από τον ένα peer στον άλλο για διατήρηση του ισοζυγίου φορτίου.

Η αναζήτηση στο δένδρο πρέπει να γίνεται από πάνω από την ρίζα προς τα κάτω στα φύλλα, ο peer ο οποίος είναι στην ρίζα τείνει να γίνει στενωπό σημείο του συστήματος και είναι επίσης σημείο αποτυχίας του συστήματος, δηλαδή με την πτώση του, το σύστημα δεν λειτουργεί. Επίσης πρέπει να λαμβάνεται πρόνοια ώστε ταυτόχρονες ενημερώσεις να εκτελούνται με ασφάλεια και ορθότητα. Επίσης η διατήρηση ενός ισοζυγισμένου δένδρου (R-tree) είναι ένα ακόμα πιο πολύπλοκο θέμα, από τις ταυτόχρονες ενημερώσεις.

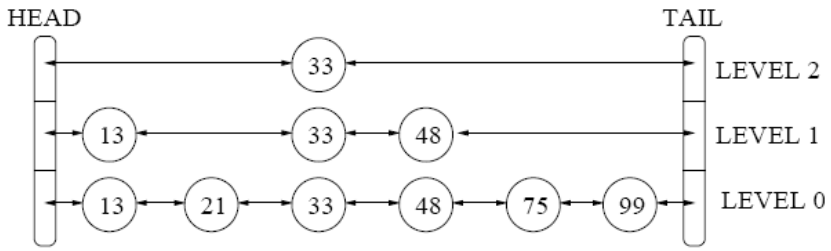
➢ ***SkipIndex: οργανώνοντας περιοχές σε ένα SkipGraph***

Ο χωρισμός του χώρου γίνεται όπως φαίνεται στο σχήμα 2.11 και ονομάζεται ***δένδρο περιοχών (ή δένδρο)***. Συσχετίζουμε ένα μονοδιάστατο κλειδί σε κάθε περιοχή του συστήματος για να πετύχουμε μια καθολική σειρά των περιοχών. Αυτό το κλειδί περιέχει τον ιεραρχική τρόπο με τον οποίο αυτή η περιοχή δημιουργήθηκε. Αυτά τα κλειδιά χρησιμοποιούνται για την αποθήκευση των περιοχών φύλλα σε ένα

SkipGraph το οποίο υποστηρίζει εισαγωγές και αναζητήσεις με βάση μονοδιάστατο κλειδί. Με αυτή την δομή δεν έχουμε hashing και άρα διατηρούμε την λογική ολοκλήρωση του χώρου κλειδιών. Επιπρόσθετα δίνοντας μας ένα δεδομένο (το σημείο του δεδομένου) και μια περιοχή φύλλο, μπορούμε να αποφανθούμε αν η περιοχή που περιέχει το σημείο αυτό είναι πριν ή μετά την δοσμένη περιοχή φύλλο στην καθολική σειρά. Για τον εντοπισμό μιας περιοχής η οποία περιέχει ένα σημείο, το ερώτημα δρομολογείτε διαμέσου του SkipGraph με τέτοιο τρόπο έτσι ώστε η απόσταση μέχρι την περιοχή προορισμό, μετρούμενη σε σχέση με την καθολική σειρά, να γίνεται πιθανόν μισή σε κάθε βήμα δρομολόγησης. Το δένδρο περιοχών δεν χρησιμοποιείται για σκοπούς πλοήγησης, αντιθέτως κάθε κόμβος επεξεργαστής διατηρεί μια μερική όψη του δένδρου για βοήθεια στην επεξεργασία των ερωτημάτων και για τον καθορισμό της σειράς μεταξύ της περιοχής και του σημείου προορισμού.

➢ SkipGraphs

Το σχήμα δρομολόγησης SkipIndex βασίζεται πάνω στο *SkipGraphs* το οποίο είναι μια γενίκευση του ***SkipList*** για μία διάσταση. Το SkipList κατασκευάζεται με βάση πιθανότητες. Κάθε στοιχείο στο SkipList πιθανών να περιέχεται σε αρκετά επίπεδα από συνδεδεμένες λίστες. Η λίστα του χαμηλότερου αποτελείτε από όλα τα στοιχείο ταξινομημένα με βάση τα κλειδιά τους. Κάθε στοιχείο το οποίο εμφανίζεται στην λίστα του επιπέδου i μπορεί να εμφανίζεται και στην λίστα του επιπέδου i+1 με πιθανότητα p. Σε κάθε επίπεδο τα στοιχεία διατηρούν δείκτες στους αριστερά και δεξιά γείτονές τους. Για εντοπισμό ενός κλειδιού αρχικά ψάχνουμε το ψηλότερο επίπεδο το οποίο περιέχει και τα λιγότερα στοιχεία, και πιθανών πολύ λίγα, και στη συνέχεια πέφτουμε κάτω στα χαμηλότερα και πυκνότερα επίπεδα αν χρειάζεται. Ο μέσος όρος των επιπέδων είναι O(log n), όπου n ο αριθμός των στοιχείων στο χαμηλότερο επίπεδο. Άρα η αναζήτηση θα διατρέξει περίπου O(log n) κλειδιά προτού φτάσει στο προορισμό.
Στο σχήμα 2.12α φαίνεται ένα παράδειγμα μιας SkipList δομής



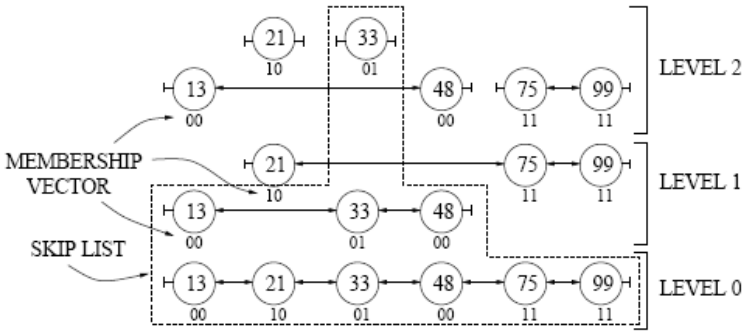
Σχήμα 2.12α: SkipList [11]

Το ***SkipGraph*** επεκτείνει το SkipList για κατανεμημένα περιβάλλοντα, προσθέτοντας πλεονάζουσα συνδεσιμότητα και διαχείριση στην δομή δεδομένων. Είναι ισοδύναμο με μια συλλογή από το πολύ n SkipLists(όπου n ο αριθμός των στοιχείων στο κατώτερο επίπεδο). Κάθε στοιχείο λαμβάνει μέρος σε όλα τα επίπεδα σε ακριβώς μια SkipList. Μερικά από τα κατώτερα επίπεδα να μοιράζονται μεταξύ πολλών SkipLists.

Κάθε στοιχείο (peer στην περίπτωσή μας) έχει ένα τυχαίο κλειδί από 0 και 1 μήκους όσα και τα επίπεδα της SkipList δηλαδή log(N) όπου N το πλήθος των peers. Για να κατασκευάσει την SkipList του ο peer, σε κάθε επίπεδο της παίρνει πρόθεμα του

κλειδιού του ίσο με K και το ελέγχει με το πρόθεμα K των κλειδιών των υπόλοιπων peers και σε όσους είναι το ίδιο τους εισάγει στο επίπεδο αυτό, όπου K το επίπεδο (0-log(N)). Συγκεκριμένα σε κάθε επίπεδο οι κόμβοι σε μια λίστα πρέπει να έχουν κοινό πρόθεμα ανάλογα με το επίπεδο, π.χ. στο επίπεδο 0 να έχουν κοινό πρόθεμα μήκους 0 στο επίπεδο 1 μήκους 1 κοκ. Έτσι κάθε peer κατασκευάζει τη δική του SkipList και όλες μαζί αποτελούν το SkipGraph.

Η αυξημένη συνδεσιμότητα παρέχει μεγαλύτερη ανεκτικότητα στα σφάλματα, αφού κάθε στοιχειό μπορεί να εντοπιστεί χρησιμοποιώντας οποιοδήποτε από τα στοιχεία των ανώτερων επιπέδων των διαφορετικών SkipLists.



Σχήμα 2.12β: SkipGraph με logN=3 επίπεδα [11]

➢ Περιγραφή της SkipIndex δρομολόγησης

Αρχικά περιγράφουμε κάποιους όρους που χρησιμοποιούνται στην περιγραφή.

Περιοχές: Μια περιοχή αντιπροσωπεύεται από ένα κόμβο στο δένδρο περιοχών (σχήμα 2.11). Κάθε περιοχή είναι ένα υπέρ-ορθογώνιο στο χώρο αναζήτησης. Μια ενδιάμεση περιοχή δημιουργεί δυο περιοχές παιδιά όταν χωρίζεται σε μια από τις διαστάσεις της. Περιοχές φύλλα συσχετίζονται με φυσικά μηχανήματα τα οποία είναι υπεύθυνα για την διαχείριση αυτού του κομματιού δεικτοδότησης.

Ιστορικό διαχωρισμού μιας περιοχής αντιπροσωπεύει το μονοπάτι στο δένδρο περιοχών από την ρίζα στην περιοχή αυτή. Συγκεκριμένα είναι μια λίστα από εγγραφές $<dim_{split}, pos_{split}>$ με κάθε μια να καθορίζει την διάσταση και την θέση στην οποία έγινε ο διαχωρισμός.

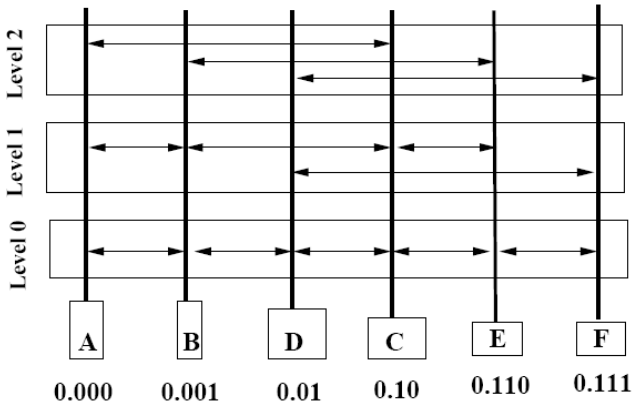
Κωδικός περιοχής είναι ένα string από 0 και 1 το οποίο αντιπροσωπεύει πώς μια περιοχή δημιουργείτε από την διαδικασία διαχωρισμού. Όταν μια περιοχή χωρίζεται σε δύο μέρη ο κωδικός περιοχής για το αριστερό παιδί (το οποίο έχει και την περιοχή με τις μικρότερες συντεταγμένες στην διάσταση που έγινε ο διαχωρισμός) δημιουργείτε προσθέτοντας 0 στο τέλος του κωδικού περιοχής της αρχικής περιοχής από την οποίο προήλθε. Ο κωδικός περιοχής του δεξιού παιδιού υπολογίζεται προσθέτοντας 1 στο τέλος του κωδικού περιοχής της αρχικής περιοχής. Ο κωδικός αναπαρίστανται σε μια δυαδική κλασματική μορφή με μια υποδιαίρεση αριστερά από το σημαντικότερο bit. Όταν ο κωδικός περιοχής συνδυάζεται με το ιστορικό διαχωρισμού περιγράφει πλήρως τις συντεταγμένες τις περιοχής.

Σειρά της περιοχής R και του σημείου p: θεωρούμε το R_p την περιοχή φύλλο η οποία περιέχει το σημείο p, αν ο κωδικός (R_p) είναι γνωστός τότε μπορούμε να χρησιμοποιήσουμε τους παρακάτω απλούς ελέγχους για να καθορίσουμε την σειρά μεταξύ της περιοχής R και του σημείου p. Αν ο κωδικός R περιέχεται στο R_p, π.χ. είναι πρόθεμα του, τότε ισχύει $p \in R$, αφού η περιοχή R_p δημιουργήθηκε μετά από διαχωρισμό της περιοχής R. Αλλιώς αν $code(R) < code(R_p)$ τότε ισχύει $R < p$ αλλιώς $p < R$. Όμως αν ο κωδικός R_p δεν είναι γνωστός, πρέπει να χρησιμοποιήσουμε το ιστορικό διαχωρισμού του R για να καθορίσουμε που εμφανίζεται το σημείο p με βάση τους διάφορους διαχωρισμούς περιοχών που συνέβησαν για να δημιουργηθεί η περιοχή R. Παρακάτω ακολουθεί ο αλγόριθμος αυτός.

for *i*th tuble [$\dim_{\text{split}}$, pos_{split}] **in** *History(R)* **do**
  c←*i*th bit in *Code(R)*
  if (c=0 & $p[\dim_{\text{split}}] \geq pos_{split}$) **then**
    return ($p>R$)
  else if (c=1 & $p[\dim_{\text{split}}] < pos_{split}$) **then**
    return ($p<R$)
return ($p \in R$)

Δηλαδή για κάθε εγγραφή του ιστορικού διαχωρισμού παίρνουμε το αντίστοιχο bit του κωδικού περιοχής και αν η περιοχή R είχε πάει στην αριστερή πλευρά (bit=0) και το σημείο p στην διάσταση που έγινε ο διαχωρισμός, έχει μεγαλύτερη τιμή από την θέση που έγινε ο διαχωρισμός τότε το p είναι μεγαλύτερο από το R ως προς αυτήν την διάσταση διαχωρισμού. Αλλιώς αν το bit=1 και αυτή η διάσταση του p είναι μικρότερη από το σημείο όπου έγινε διαχωρισμός τότε το p είναι μικρότερο από το R ως προς αυτήν την διάσταση. Αλλιώς το p περιέχεται στο R.

Οι περιοχές φύλλα οργανώνονται σε ένα SkipGraph χρησιμοποιώντας τους κωδικούς περιοχής τους ως κλειδιά. Το σχήμα 2.13 εικονίζει τους κωδικούς περιοχής που ανατέθηκαν στις περιοχές που φαίνονται στο δένδρο του σχήματος 2.11 και την τελική SkipGraph οργάνωση των περιοχών.



Σχήμα 2.13: SkipGraph [2]

Ένας κόμβος ο οποίος είναι υπεύθυνος για μια περιοχή φύλλο διατηρεί την ακόλουθη κατάσταση για τους γείτονες τους, η οποία χρησιμοποιείτε για την δρομολόγηση ερωτημάτων: α) για κάθε επίπεδο του SkipGraph, ο κόμβος ο οποίος κατέχει μια περιοχή διατηρεί τα IDs και διευθύνσεις των κόμβων οι οποίοι κατέχουν διπλανές περιοχές. β) για κάθε επίπεδο του SkipGraph, ο κόμβος ο οποίος κατέχει μια περιοχή διατηρεί επίσης το ιστορικό διαχωρισμού και τους κωδικούς περιοχής των διπλανών περιοχών.

Όταν ένας κόμβος ο οποίος κατέχει μια περιοχή A του τεθεί το ερώτημα του εντοπισμού της περιοχής φύλλου η οποία περιέχει το σημείο στόχος p τότε εκτελεί τις ακόλουθες ενέργειες.

Algorithm 1 SkipIndex Routing
if ($p \in A$) **then**
  A is the destination region
else if ($p > A$) **then**
  /*move right */
  for *i=max-level* down to 0 **do**
    If *(*! *right_region*[i]>*p*) **then**
      forward to *right_neighbor*[i]
else /*move left, omitted*/

Με άλλα λόγια ο κόμβος ο οποίος κατέχει την περιοχή A εξετάζει τις γειτονικές του περιοχές αρχίζοντας από το ψηλότερο επίπεδο του SkipGraph και δρομολογεί το ερώτημα στο μακρινότερο γείτονα του χωρίς βέβαια να προσπερνά το σημείο προορισμός, δηλαδή το μακρινότερο γείτονα που είναι όμως πριν το σημείο προορισμού. Καθώς το ερώτημα πλησιάζει το σημείο προορισμού, ο αλγόριθμος δρομολόγησης χρησιμοποιεί σταδιακά χαμηλότερα επίπεδα του SkipGraph. Η απόσταση δρομολόγησης και ο αριθμός των peers που εμπλέκονται για την δρομολόγηση ενός ερωτήματος, έχουν όριο O(log N) όπου N οι κόμβοι στο SkipGraph. Αυτό το όριο είναι ανεξάρτητο από την κατανομή των δεδομένων και μπορεί να επιτευχθεί ακόμα και όταν τα μεγέθη των περιοχών είναι πολύ ανόμοια ή διαφορετικά.

Πρέπει να σημειωθεί ότι όλες οι περιοχές φύλλα που σχηματίζονται από επαναληπτικό χωρισμό μιας μόνο ενδιάμεσης περιοχής είναι συνοριακές στο χαμηλότερο επίπεδο του SkipGraph. Αυτό μας δίνει την δυνατότητα να κάνουμε broadcast ένα ερώτημα σε όλες τις περιοχές φύλλα που έχουν σχηματιστεί από μία ενδιάμεση περιοχή.

Στο skipIndex χρησιμοποιείτε μια δυναμική έκδοση για ισοζυγισμένη παραχώρηση περιοχών. Ένας κόμβος φέρνει ένα ανενεργό κόμβο για να του δώσει περιοχή όταν ανιχνευτεί υπερφόρτωση. Η λεπτομέρειες για τον πώς ανιχνεύεται η υπερφόρτωση και πώς δημιουργείτε ο ανενεργής κόμβος περιγράφονται στην ενότητα 2.2.4. Ο καινούριο κόμβος λαμβάνει από τον κόμβο που εκτελεί το διαχωρισμό το ιστορικό διαχωρισμού, και το μερίδιο των κλειδιών τα οποία πρέπει να παραχωρηθούν στον καινούριο κόμβο. Μετά το διαχωρισμό και οι δύο κόμβοι τροποποιούν το ιστορικό διαχωρισμού τους καθώς και το κωδικό περιοχής τους. Συγκεκριμένα ο αρχικός κόμβος κρατά το χαμηλότερο μισό της αρχικής περιοχής και προσθέτει ‘0’ στο τέλος

του κωδικού περιοχής του. Ο καινούριος κόμβος λαμβάνει το πάνω μισό της περιοχής και προσθέτει ‘1’ στο τέλος του κωδικού περιοχής του. Ο αρχικός κόμβος δεν αλλάζει τις τιμές των κλειδιών κατά την διάρκεια της διαδικασίας διαχωρισμού και ο καινούριος κόμβος γίνεται ο δεξιός γείτονας του, στο χαμηλότερο επίπεδο του SkipGraph. Ο καινούριος κόμβος συνδέεται στο SkipGraph με τον κωδικό περιοχής του. Η διαδικασία σύνδεσης περιγράφεται στην ενότητα 2.2.4. Μετά την εισαγωγή ο κόμβος που εκτελεί το διαχωρισμό ειδοποιεί τους peers του για αλλαγές στο ιστορικό διαχωρισμού τους και τον κωδικό περιοχής τους.

2.2.3 Κατανεμημένη Διαδικασία Ερωτημάτων

Το SkipIndex υποστηρίζει πολλών ειδών ερωτήματα: ερωτήματα σημείου, ευρεία ερωτήματα, ερωτήματα για τα k-κοντινότερα γειτονικά. Το ερώτημα σημείου είναι ίσο με την δρομολόγηση του ερωτήματος στον κόμβο ο οποίος κατέχει την περιοχή που περιλαμβάνει το σημείο προορισμού.

Το ***ευρύ ερώτημα(range query)*** επιστρέφει όλα τα σημεία δεδομένων τα οποία περιλαμβάνονται σε μια δοσμένη περιοχή υπέρ-ορθογωνίου ή υπέρ-σφαίρας ή άλλο σχήμα το οποίο μπορεί να προκύψει χρησιμοποιώντας μέτρα αποστάσεων με διαφορετικά βάρη για τις διάφορες διαστάσεις. Αυτά τα ερωτήματα μπορούν να εκτελεστούν με μια ευρεία λειτουργία πολυεκπομπής η οποία στέλνει το ερώτημα σε όλους τους κόμβους τους οποίους η περιοχή τους τέμνει την ***ευρεία περιοχή του ερωτήματος (query range)***. Αυτή η λειτουργία μπορεί να εκτελεστεί με λογαριθμικό αριθμό από βήματα, ακόμα και αν πολλοί κόμβοι διατηρούν περιοχές που τέμνουν τη συγκεκριμένη περιοχή του ερωτήματος.

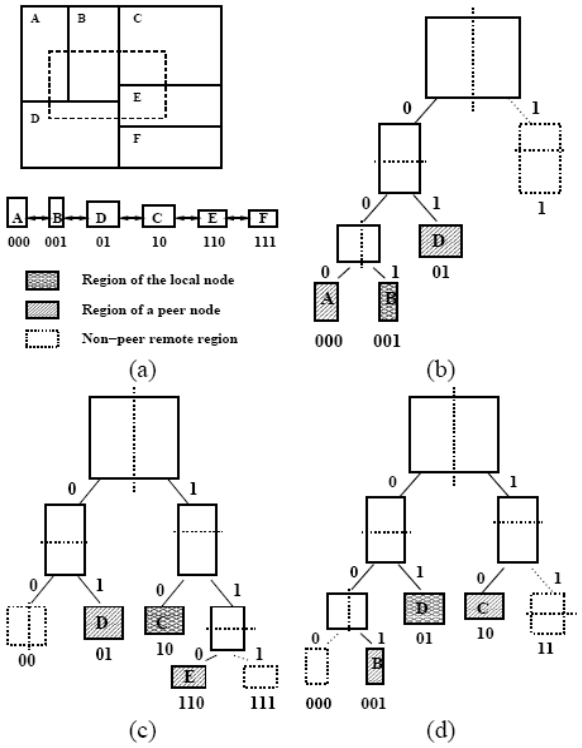
- ***<u>Ευρύ Ερώτημα με Πολύεκπομπή</u>***

Το SkipIndex εκτελεί διάσχιση του δέντρου σε ένα ευρύ ερώτημα ξεκινώντας από την ρίζα και επαναληπτικά επισκέπτεται κάθε υποδέντρο του οποίου τα όρια της περιοχής του τέμνονται με την *ευρεία περιοχή του ερωτήματος (query range)*. Αυτή η διάσχιση γίνεται με ένα κατανεμημένο τρόπο χωρίς να απαιτείται οποιοσδήποτε κόμβος να διατηρεί την καθολική όψη της δομής δεικτοδότησης.

Κάθε κόμβος διατηρεί μια *μερική όψη* του δένδρου. Η *μερική ή τοπική όψη του δέντρου* σε κάθε κόμβου περιλαμβάνει το ιστορικό διαχωρισμού της τοπικής περιοχής του καθώς και τα ιστορικά διαχωρισμού των περιοχών που διατηρούνται από τους γειτονικούς του peers στο SkipGraph. Κάθε ιστορικό παρέχει πληροφορίες για τη διαδρομή από την ρίζα του δέντρου μέχρι τη περιοχή φύλλο. Το σχήμα 2.14 παρουσιάζει τις μερικές όψεις για 3 κόμβους, όπως βλέπουμε π.χ. ο κόμβος B στο σχήμα 2.14.b διατηρεί ιστορικό για την περιοχή του B, και για τους γειτονικούς κόμβους A και D, ενώ για τους υπόλοιπους κόμβους C, E και F δεν διατηρεί ιστορικό, αλλά τους βλέπει σαν μια και μόνο άγνωστη περιοχή

Υπάρχουν τριών ειδών “φύλλα” στη μερική όψη δέντρου των κόμβων.1. Η τοπική περιοχή του ίδιου κόμβου,2. οι περιοχές των peers του δηλαδή των γειτονικών peers στο SkipGraph και οι 3. “άγνωστες” περιοχές οι οποίες αντιστοιχούν στις άγνωστες

περιοχές του δέντρου. Τις δύο τελευταίες περιοχές θα τις ονομάσουμε ***απομακρυσμένες περιοχές (remote regions)***. Μια άγνωστη περιοχή μπορεί να είναι μια απλή περιοχή φύλλο ή μια ομάδα από περιοχές φύλλα. Ο τοπικός κόμβος δεν έχει ικανοποιητική πληροφόρηση για αυτές τις περιοχές.



Σχήμα 2.14: Μερικές όψεις δέντρου για διάφορους κόμβους.

Ένα *ευρύ ερώτημα* ξεκινά δρομολογώντας στο κέντρο της *ευρεία περιοχή του ερωτήματος*-***qrange***, π.χ. στο σχήμα 2.15 το κέντρο της *qrange* είναι το σημείο q το οποίο βρίσκεται στην περιοχή B. Ο κόμβος ο οποίος έχει τη κεντρική αυτή περιοχή εκτελεί ένα τοπικό ευρύ ερώτημα και διασχίζει το τοπικό μερικό δέντρο του για να βρει ποιες άλλες περιοχές τέμνουν το *qrange*. Αυτό είναι ίσο με την διάσχιση δέντρου που γίνεται σε ένα συγκεντρωτικό δέντρο δεικτοδότησης εκτός του γεγονός ότι επιστρέφετε ένα σύνολο από περιοχές για αναζήτηση σε απομακρυσμένα μηχανήματα. Τότε το ερώτημα κάνει πολυεκπομπή σε κάθε περιοχή του συνόλου για να συνεχιστεί η αναζήτηση.

Για μια *άγνωστη περιοχή προορισμού*-***dregion*** **,** δρομολογούμε το μήνυμα σε κάποιους κόμβους οι οποίοι δεικτοδοτούν τουλάχιστο μια περιοχή της *dregion.* Αυτοί οι κόμβοι μπορούν, αν είναι απαραίτητο, να αποσυνθέσουν περεταίρω τη *dregion* σε μικρότερες περιοχές και να προωθήσουν ένα μήνυμα στον κατάλληλο κόμβο, αφού όπως αναφέραμε προηγουμένως μια άγνωστη περιοχή μπορεί να είναι μια απλή περιοχή φύλλο ή μια ομάδα από περιοχές φύλλα Το μήνυμα περιλαμβάνει το αυθεντικό *qrange* καθώς και το *dregion* δηλαδή την περιοχή την οποία ο παραλήπτης κόμβος πρέπει να αποσυνθέσει περεταίρω για να μπορεί να προωθήσει το μήνυμα. Όταν ένας κόμβος προωθεί μηνύματα σε άλλους κόμβους καθορίζει μη επικαλυπτόμενες *dregion* για να αποτρέψει την δημιουργία διπλών μηνυμάτων για ένα δοσμένο ερώτημα.

Χρησιμοποιούμε τον συμβολισμό $r \cap qrange$ για να υποδηλώσουμε ότι η περιοχή r τέμνει την *ευρεία περιοχή του ερωτήματος* και $r_1 \subseteq r_2$ για να υποδηλώσουμε ότι η περιοχή r_1 περικλείετε από την περιοχή r_2. Παρακάτω ακολουθεί ο αλγόριθμος ο οποίος εκτελείτε από ένα κόμβο που παραλαμβάνει το ερώτημα <*qrange,dregion*>

Algorithm 2 Range-Limited Multicast
Upon node receive query<qrange, dregion>:
if *local_region(node)* $\subseteq$ *dregion* **then**
perform local query(*qrange*)
for all remote region *R* in partial tree view **do**
if $R \subseteq dregion$ **and** $R \cap qrange$ **then**
forward query<*qrange, R*> **to** *R*

Το βάθος της πολυεκπομπής δηλαδή ο μέγιστος αριθμός βημάτων για να φτάσουμε στην περιοχή φύλλο το οποία ανήκει στην ευρύτερη περιοχή προορισμού, δεν εξαρτάτε από το ύψος του δέντρου το οποίο μπορεί αν είναι O(N) στην χειρότερη περίπτωση. Αντιθέτως για πολυεκπομπή ενός ερωτήματος σε μια δοσμένη περιοχή η διαδικασία πολυεκπομπής προωθεί τα μηνύματα χρησιμοποιώντας συνδέσεις του SkipGraph και σε κάθε βήμα αυτής της διαδικασίας επιτυγχάνει ένα από τους παρακάτω στόχους: α) η απόσταση από το σύνολο των κόμβων που αντιπροσωπεύουν την περιοχή προορισμού μοιράζεται, ή β) η περιοχή προορισμού αποσυντίθεται σε υποπεριοχές, οι οποίες έχουν το πολύ το μισό μέγεθος από την αρχική περιοχή προορισμού. Όταν η *ευρεία περιοχή του ερωτήματος* είναι μικρή, π.χ. μερικές γειτονικές περιοχές εμπλέκονται, το βάθος της πολυεκπομπής είναι τυπικά O(1), καθώς αυτές οι περιοχές ήδη περιλαμβάνονται στην μερική όψη δέντρου του αρχικού κόμβου.

➢ ***Αλγόριθμος ερωτήματος k-κοντινότερων γειτόνων***

Το αναζήτηση ***k-κοντινότερων γειτόνων (ΚΚΓ)*** είναι παρόμοια με ένα σφαιρικό ευρύ ερώτημα, με την διαφορά ότι η ακτίνα δεν προκαθορίζεται αλλά προσδιορίζεται δυναμικά κατά την διάρκεια της αναζήτησης.

Στο πρώτο στάδιο, δρομολογούμε την αναζήτηση των κ-κοντινότερων γειτόνων στο κόμβο Α ο οποίος κατέχει την περιοχή που περιέχει το *σημείο ερωτήματος* ***q***. Ο κόμβος Α τότε εκτελεί ένα τοπικό ερώτημα για να προσδιορίσει ένα αρχικό σύνολο υποψήφιων από κ-κοντινότερα γειτονικά σημεία , οι οποίοι υποδηλώνονται με $\boldsymbol{S_{\kappa\kappa\gamma}}$. Η μέγιστη απόσταση από το σημείο q σε οποιοδήποτε σημείο στο $S_{\kappa\kappa\gamma}$ είναι η αρχική τιμή της ακτίνας $\boldsymbol{R_{\kappa\kappa\gamma}}$ του ΚΚΓ ερωτήματος. Αν το μέγεθος του $S_{\kappa\kappa\gamma}$ είναι μικρότερο από το κ, τότε η ακτίνα $R_{\kappa\kappa\gamma}$ τίθεται σε τιμή τέτοια ώστε να καλύπτει ολόκληρο το d-διαστάσεων χώρο. Ο κόμβος Α διατηρεί επίσης μια ουρά προτεραιότητας Q_{search}, με περιοχές στις οποίες πρέπει να γίνει αναζήτηση, και είναι ταξινομημένες σε σχέση με την ελάχιστη απόσταση από το σημείο ερωτήματος q. Το αρχικό περιεχόμενο της ουράς Q_{search} αποτελείτε από περιοχές που τέμνουν την τρέχων σφαίρα ερωτήματος <q, $R_{\kappa\kappa\gamma}$ >, και καθορίζονται μετά από διάσχιση του μερικού δέντρου του κόμβου Α.

Ο αλγόριθμος εκτελεί τα ερωτήματα στις περιοχές στην ουρά Q_{search} με σειρά ανάλογη με την απόσταση της κάθε περιοχής από το σημείο q. Ο Α αποσπά κάθε φορά από την ουρά την περιοχή **R** με την μικρότερη απόσταση από το σημείο q, και στέλνει ένα μήνυμα ερώτημα <*q*, $R_{κκγ}$,*R*> προς την περιοχή R. Όταν αυτό το μήνυμα λαμβάνεται από ένα κόμβο, ο οποίος μπορεί να έχει περισσότερες λεπτομερής πληροφορίες για την περιοχή R, τότε η περιοχή αναζήτησης φιλτράρεται (refined), βασιζόμενη στο μερικό δέντρο του κόμβου αυτού και προωθείται προς την υπο-περιοχή του R η οποία είναι κοντινότερα στο σημείο q. Όταν η περιοχή αναζήτησης τελικά φιλτράρεται σε μια περιοχή φύλλο και λαμβάνεται από τον αντίστοιχο κόμβο, ένα τοπικό ερώτημα εκτελείτε για να καθοριστούν σημεία που περιέχονται στην περιοχή η οποία είναι πιο κοντά σε σχέση με την τρέχων ΚΚΓ εκτιμωμένη απόσταση $R_{κκγ}$. Τα αποτελέσματα αναφέρονται πίσω στον Α μαζί με τις νέες υποπεριοχές που έχουν ανακαλυφτεί και τέμνουν τη σφαίρα ερωτήματος (query ball). Ο Α τότε ενημερώνει το $S_{κκγ}$ και $R_{κκγ}$, και εισάγει τις υποπεριοχές που βρέθηκαν κατά τη διάρκεια του συγκεκριμένου βήματος στην Q_{search}. Ο Α επαναλαμβάνει το στάδιο ερώτησης μέχρι να μην υπάρχουν πλέον περιοχές στην Q_{search} με απόσταση $R_{κκγ}$.
Ο παρακάτω αλγόριθμος 3 περιγράφει αυτήν την διαδικασία.

Algorithm 3 Basic KNN query algorithm

A does a local KNN search and initializes S_{knn} and R_{knn} based on the results
A traverses its partial tree view and initializes Q_{search} with all regions R, such *that mindist(R,q)* $\leq R_{knn}$
while (Q_{search} not empty) **do**
  $R \leftarrow$ extract_min(Q_{search})
  forward query<*q*, R_{knn},*R*> **to** *R*
  receive result < Set_p, Set_R >
  insert regions in Set_R into Q_{search}
  update S_{knn} and R_{knn} with Set_p
  prune Q_{search} with new R_{knn}
return S_{knn}

Upon node B receiving the query< *q*, R_{knn},*R* > :
if (*local_region(B)==R*) **then**
  $Set_p \leftarrow K$ nearest points within R_{knn}
  $Set_R \leftarrow$ leaf regions inside R from local tree view
  reply to *A* the result < Set_p, Set_R >
else
  find the leaf R_t in local tree view with minimum *mindist(q,R)*
  forward query< *q*, R_{knn},*R* > **to** R_t

Η σειριακή φύση αυτών των ερωτημάτων μπορεί να προκαλέσει υψηλή καθυστέρηση. Αυτό το πρόβλημα μπορεί να αντιμετωπιστεί αρχικοποιώντας τις αναζητήσεις στις Μ κοντινότερες περιοχές, αντί για μία περιοχή κάθε φορά. Έτσι αυξάνεται ο παραλληλισμό και βελτιώνεται η καθυστέρηση όμως γίνονται μερικά αχρείαστα ερωτήματα.

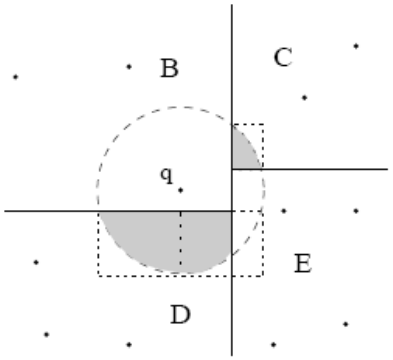
➢ ***Προσεγγιστική Αναζήτηση***

Καθώς αυξάνουμε τις διαστάσεις όλο και περισσότερες οριακές περιοχές στην δομή δεικτοδότησης τέμνουν την σφαίρα του ερωτήματος και τελικά η αναζήτηση πρέπει να σαρώσει όλα το σύνολο δεδομένων. Αυτό αναφέρεται ως '*Η κατάρα των διαστάσεων'*. Για πολλές εφαρμογές αυτό μπορεί να αποφευχθεί χρησιμοποιώντας προσεγγιστική αναζήτηση για τους παρακάτω λόγους:

Πρώτον αρκετές εφαρμογές, μπορούν να ανεχτούν μικρή ανακρίβεια στα αποτελέσματα του ερωτήματος. Επιπρόσθετα οι περισσότεροι τρόποι μέτρησης των αποστάσεων σε αυτές τις εφαρμογές είναι ευριστικής φύσης σε πρώτο μέρος, και έτσι παρέχοντας ελαφρά ανακριβή αποτελέσματα πιθανών να μην έχει σοβαρές συνέπειες.

Δεύτερο, παρόλο που η απόσταση προς τους κ-κοντινότερους γείτονες, και συνεπώς η ακτίνα του ερωτήματος αυξάνονται σταδιακά με την αύξηση των διαστάσεων, ο όγκος της σφαίρας του ερωτήματος τυπικά δεν αυξάνεται. Η πλειοψηφία των οριακών περιοχών τέμνουν την σφαίρα ερωτήματος με μόνο μια από τις γωνίες τους. Αυτές οι τομές είναι συνεπώς αμελητέες στον όγκο. Αν τα σημεία δεδομένων είναι ομοιόμορφα κατανεμημένα στο τοπικό εύρος του ερωτήματος, τότε αυτές οι μικροσκοπικές περιοχές τομής μπορούν να αγνοηθούν καθώς έχουν πολύ μικρή πιθανότητα να έχουν κάποια σημεία δεδομένων.

Βασιζόμενοι σε αυτήν την παρατήρηση μπορούμε να εισάγουμε ένα ακριβές όριο για τα ερωτήματα που βασίζονται στον όγκο της περιοχής αναζήτησης. Δηλαδή όταν ο όγκος της εξεταζόμενης περιοχής ο οποίος βρίσκεται μέσα στην σφαίρα ερωτήματος ξεπερνά το επιθυμητό όριο ακρίβειας, η αναζήτηση θα τερματίζεται. Το σχήμα 2.15 δείχνει μια σφαίρα ερωτήματος με κέντρο το σημείο q η οποία τέμνει τέσσερις ορθογώνιες περιοχές. Αφού η τομή της σφαίρας με την περιοχή C έχει αμελητέο όγκο, μπορούμε να παραλείψουμε να ρωτήσουμε την περιοχή C χωρίς να θυσιάζουμε την ακρίβεια της αναζήτησης.



Σχήμα 2.15: Τομή της σφαίρας ερωτήματος με περιοχές

➢ Εκτίμηση του όγκου τομής

Παρουσιάζεται μια μέθοδος η οποία υπολογίζει το άνω όριο του όγκου που τέμνεται. Θεωρώντας την τομή της περιοχής C με την σφαίρα ερωτήματος στο σχήμα 2.15 και ένα ορθογώνιο συνοριακό κουτί (BB) το οποίο εμπεριέχει αυτήν την τομή. Ο λόγος του όγκου μιας περιοχής που τέμνει τη σφαίρα με τον όγκο του συνοριακού κουτιού αυτής της τομής φράσσεται άνω από τον λόγω ολόκληρης της σφαίρας με τον όγκο του περιγεγραμμενου τετραγώνου της σφαίρας. Αυτό το τετράγωνο, αν η ακτίνα τις σφαίρας είναι R, έχει όγκο $(2R)^D$ και έτσι παίρνουμε το παρακάτω όριο όγκου:

$$V_{\text{int}ersect} \le \frac{V_{sphere}}{(2R_{\text{ΚΚΓ}})^D} * V_{BB}$$

Ο V_{sphere} είναι συνάρτηση των $R_{\text{ΚΚΓ}}$ και D. Το V_{BB} είναι το ορθογώνιο συνοριακό κουτί το οποίο εμπεριέχει την τομή και μπορεί να υπολογιστεί καθορίζοντας τα σημεία τομής του υπερκύβου με την υπερσφαίρα. Έτσι είναι δυνατόν να υπολογίσουμε το πάνω όριο του $V_{\text{int}ersect}$.

Για τον όγκο τομής της σφαίρας με τις περιοχές D και E είναι λίγο πιο δύσκολο να βρεθεί το πάνω όριο τους. Αυτές οι περιοχές επικαλύπτονται με περισσότερα από ένα τεταρτημόρια της σφαίρας ερωτημάτων. Σε αυτές τις περιπτώσεις επεκτείνουμε την περιοχή έτσι ώστε οι γωνιές της να μην βρίσκονται μέσα στην σφαίρα, σπάζοντας την εκτεταμένη περιοχή σε διαφορετικά κομμάτια έτσι ώστε κάθε κομμάτι να τέμνει την σφαίρα σε ακριβώς ένα τεταρτημόριο, και τότε χρησιμοποιούμε τη μέθοδο υπολογισμού του πάνω όριο που περιγράψαμε προηγουμένως για να υπολογίσουμε τον όγκο τομής για κάθε ένα από αυτά τα κομμάτια.

➢ Νωρίς τερματισμός της προσεγγιστικής αναζήτησης

Υποθέτουμε ότι κάθε ερώτημα για κ-κοντινότερους γείτονες καθορίζει ένα όριο λάθους **ε.** υποθέτοντας ομοιόμορφη κατανομή δεδομένων γύρω από το σημείο ερωτήματος επιτρέπεται να αγνοήσουμε ένα ποσοστό ε της σφαίρας ερωτήματος. Τώρα τροποποιούμε τον αλγόριθμο κ-κοντινότερων γειτόνων έτσι ώστε για κάθε περιοχή στην Q_{search}, να κρατά πληροφορία για το εκτεταμένο οριακό κουτί της τομής της περιοχής με την σφαίρα ερωτήματος. Η συνθήκη τερματισμού της αναζήτησης τροποποιείτε στην:

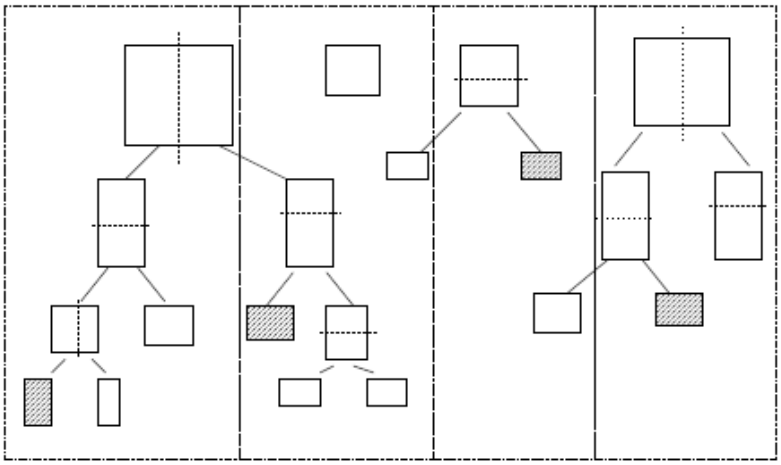
$$\sum_{\forall R \in Q_{search}} (V_{EBB(R)}) < \varepsilon * (2R_{KK\Gamma})^D$$

Δηλαδή $\varepsilon = \dfrac{V_{\text{int}ersect}}{V_{sphere}}$

2.2.4 Υπηρεσίες Δεικτοδότησης

- ***<u>Συνύπαρξη ποικίλων μορφών δεικτοδότησης</u>***

Το SkipIndex μπορεί να επεκταθεί έτσι ώστε να υποστηρίζει πολλαπλές μορφές δεικτοδότησης. Το σχήμα 2.16 παρουσιάζει την κύρια ιδέα. Τα ευρετήρια ταξινομούνται με βάση τα ID τους. Έτσι τα δέντρα περιοχών σχηματίζουν ένα δάσος. Χρησιμοποιούμε το SkipIndex για να οργανώσουμε τις περιοχές φύλλα σε μια σειρά στο δάσος με βάση τα κλειδιά που παράγονται στην μορφή <*indexID, treecode*>. Με τον μεγάλο αριθμό από ευρετήρια συνδυάζουμε γειτονικά δέντρα ή τμήματα σε ένα κόμβο όπως δείχνουν και τα κουτιά με διακεκομμένη γραμμή στο σχήμα 2.16. το κλειδί σε κάθε κόμβο δίνεται από το κλειδί του ποιο αριστερού φύλλου που διατηρείτε από τον κόμβο, όπως φαίνεται από τις σκιαγμένες περιοχές του σχήματος 2.16.



Σχήμα 2.16: Χωρισμός ενός δάσους από ευρετήρια

Πολλές ιδιότητες του σχήματος αυτού επισημαίνονται παρακάτω:

- Η απόσταση δρομολόγησης και η ποσότητα της κατάστασης δρομολόγησης είναι ανεξάρτητα από τον αριθμό των ευρετηρίων και εξαρτώνται μόνο από τον αριθμό των κόμβων.
- Η διαστάσεις μιας συγκεκριμένης δεικτοδότησης είναι ασυσχέτιστες με την δομή δρομολόγησης
- Μετακινώντας ένα κομμάτι του ευρετηρίου από ένα κόμβο σε ένα γείτονα του διαμέσου του συνόρου, αλλάζει το κλειδί του κόμβου αλλά δεν αλλάζει η δομή του SkipGraph
- Απομακρύνοντας ένα κόμβο τις ευθύνες του αναλαμβάνει ο αριστερός γείτονας.

Η πρώτη ιδιότητα εξασφαλίζει την επεκτασιμότητα του SkipIndex. Η δεύτερη παρέχει την ποικιλία των συνόλων δεδομένων. Οι δύο τελευταίες ιδιότητες βοηθούν στο ισοζύγιο φορτίου και στο μηχανισμό ανάκτησης μετά από σφάλμα.

➢ ***Δυναμικό Ισοζύγιο Φορτίου***

Εδώ περιγράφεται κάποια μέθοδος για διατήρηση του ισοζυγίου φορτίου η οποία περιγράφεται στο SkipIndex[2]. Για ισοζύγιο φορτίου εμείς έχουμε υλοποιήσει τον αλγόριθμο Platon[3], αλλά για μεγαλύτερη πληρότητα παραθέτουμε και αυτή την μέθοδο.

Σε ένα πραγματικό σύστημα δεικτοδότησης υπάρχουν δύο δυναμικά θέματα που πρέπει να προσεχθούν σε ένα διαχωρισμό: πως θα αποφασίσουμε για το κατώφλι υπερφόρτωσης με ένα κατανεμημένο τρόπο, δηλαδή το ελάχιστο φορτίο για το οποίο ένας κόμβος θεωρείτε υπερφορτωμένος. Και πώς θα βρούμε τον ανενεργό κόμβο ο οποίος θα αναλάβει το επιπλέων φορτίο.

Δεν χρησιμοποιείτε κεντρική μέθοδος για τον προφανή λόγω της επεκτασιμότητας. Αντιθέτως υπάρχει μια συνεχής 'συνομιλία' μεταξύ των peers έτσι ώστε η πληροφορίες για τους υπερφορτωμένους και ελάχιστα φορτωμένους κόμβους να διασκορπιστούν σε όλο το δίκτυο. Κάθε κόμβος παίρνει την δική του απόφαση για το πώς θα κάνει το διαχωρισμό.

Οι κόμβοι στο SkipIndex δεν διατηρούν λεπτομερής πληροφορίες για το φορτίο στους άλλους κόμβους. Περιοδικά οι κόμβοι συνομιλούν με τους γείτονες τους για το τρέχων φορτίο τους καθώς και για τους k ψηλότερους και χαμηλότερους φορτωμένους κόμβους που ξέρουν. Η συνομιλίες αυτές έρχονται με χρονοσφραγίδες από τον αρχικό κόμβο για τον έλεγχο της συνέπειας. Όταν λαμβάνονται μηνύματα ο κόμβος κρατά μόνο τις καταχωρήσεις (μηνύματα) οι οποίες έχουν χρονοσφραγίδες νεότερες από τις τοπικές εγγραφές που διατηρεί ο κόμβος αυτός. Επίσης όταν ένας κόμβος Α λάβει μήνυμα από τον κόμβο Β με παλιά χρονοσφραγίδα τότε στην επόμενη συνομιλία με τον κόμβο Β θα του στείλει την τοπική εγγραφή που διατηρεί και έχει νεότερη χρονοσφραγίδα.

Το κατώφλι υπερφόρτωσης αποφασίζετε τοπικά από κάθε κόμβο ως:
MAX{k ψηλότερο φορτίο, κ χαμηλότερο φορτίο*C, T}, όπου T είναι ένα σταθερό κατώφλι ως το χαμηλότερο όριο για υπερφόρτωση, και το C είναι μια σταθερά για σταθεροποίηση μεγαλύτερη του 1.

Όταν R_t ένας κόμβος εντοπίσει υπερφόρτωση για μια σημαντική χρονική περίοδο τότε καλεί την διαδικασία διαχωρισμού. Ο κόμβος ο οποίος θα αναλάβει το επιπλέον φορτίο επιλέγεται ως εξής:

- Αν κάποιος από τους κοντινότερους γείτονες έχει χαμηλό φορτίο (underloaded), τότε επίλεξε τον.
- Αλλιώς αν υπάρχει κάποιος ανενεργός κόμβος στην τοπική όψη, τότε επίλεξε τον
- Αλλιώς αν υπάρχει κάποιος φυσικός κόμβος με χαμηλό φορτίο, ο οποίος έχει λιγότερους από V εικονικούς κόμβους, τότε δημιούργησε ένα εικονικό κόμβο εκεί.
- Αλλιώς πάρε ένα κόμβο με χαμηλό φορτίο, κάνε τον ανενεργό μετακινώντας τα δεδομένα του στο γείτονα του (***συμπίεση***)

Ένα μηχάνημα με πολλούς εικονικούς κόμβους μπορεί να φιλοξενεί μεγάλο αριθμό από συνδέσεις των peers, με αποτέλεσμα να αυξάνεται το κόστος της διατήρησης των peers και υπάρχει και η πιθανότητα απώλειας της σύνδεσης από αστοχία του κόμβου.

Η συμπίεση από την άλλη μεριά, προκαλεί υψηλή καθυστέρηση λόγω της μεταφοράς των δεδομένων. Στην πράξη η συμπίεση γίνεται στο παρασκήνιο για την διατήρηση ενός συνόλου από ανενεργούς κόμβους.

➢ ***Φυσική Γειτνίαση***

Στο SkipIndex χρησιμοποιείτε μια ευριστική μέθοδος για την βελτίωση της τοπικότητας στην δρομολόγηση δηλαδή γειτονικοί κόμβοι στο overlay δίκτυο να είναι γειτονικοί και στο φυσικό δίκτυο. Έτσι όταν ένας κόμβος διαχωρίζεται επιλέγει τον κοντινότερο στο φυσικό δίκτυο από τους K λιγότερο φορτωμένους κόμβους. Έτσι peers στο SkipGraph οι οποίοι βρίσκονται στα χαμηλότερα επίπεδα τείνουν να είναι πιο κοντά ο ένας στον άλλο. Στα ψηλότερα όμως επίπεδα οι κόμβοι είναι πιο τυχαία κατανεμημένοι. Από την τοπικότητα αυτή επωφελούνται περισσότερο λειτουργίες ερώτησης παρά λειτουργίες εισαγωγής.

➢ ***Σύνδεση Νέου Κόμβου***

Ένας νέος κόμβος συνδέεται στο SkipIndex με δύο βήματα. Πρώτο προσαρτάτε σε κάποιους υπάρχων ενεργούς κόμβους σαν ένας ανενεργός κόμβος. Ο ενεργός κόμβος μπορεί να δημοσιεύσει, στην συνομιλία του με άλλους ενεργούς κόμβους, αυτούς τους προσαρτημένους ανενεργούς κόμβος ως κόμβους με μηδενικό φορτίο. Αργότερα ένας υπερφορτωμένος ενεργός κόμβος, μετατρέπει τον ανενεργό κόμβο σε ενεργό μέσω ενός ατομικού πρωτοκόλλου σύνδεσης.

Το πρώτο βήμα δεν περιέχει κανένα θέμα συνέπειας. Ο καινούριος κόμβος επιλέγει τυχαία κάποιους κωδικούς μέλους του SkipGraph και δρομολογεί αιτήσεις προσάρτησης σε κόμβους που έχουν το μεγαλύτερο πρόθεμα που ταιριάζει με αυτούς τους κωδικούς. Χρησιμοποιούμε κωδικούς μέλους αντί για τα κλειδιά του SkipGraph διότι αυτά δεν κατανέμονται τυχαία και ομοιόμορφα. Οι κόμβοι που αποδέχονται τον ανενεργό κόμβο είναι υπεύθυνοι για να δημοσιεύουν πληροφορίες για αυτό το κόμβο στις συζητήσεις τους με τους άλλους κόμβους και να ελέγχουν περιοδικά αν ο νέος κόμβος είναι ζωντανός.

Η εισαγωγή ενός ανενεργού κόμβου στο ενεργό SkipGraph μπορεί να είναι περίπλοκη όταν έχουμε να κάνουμε με ταυτόχρονες συνδέσεις. Η εισαγωγή ξεκινά από το επίπεδο 0 δρομολογώντας ένα μήνυμα σύνδεσης στο κόμβο με κωδικό περιοχής πιο κοντά στον καινούριο κόμβο. Ο κόμβος αυτό εκτελεί διαχωρισμό της περιοχής δίνει το δεξί κομμάτι στο καινούριο κόμβο, ο οποίος είναι στα δεξιά του διαχωριζόμενου κόμβου στο δέντρο περιοχών, και επίσης προωθεί την αίτηση σύνδεσης στους κόμβους αυτού του επιπέδου οι οποίοι πρέπει να είναι peers του νέου κόμβου, δηλαδή των έχουν γείτονα στο SkipGraph στο επίπεδο 0. Μετά την εγκατάσταση του στο επίπεδο 0 ο καινούριους κόμβος εκτελεί μια διάσχιση του επιπέδου 0 για να βρει κόμβους οι οποίοι έχουν ως peer τον καινούριο κόμβο στο επίπεδο 1 δηλαδή είναι γείτονες του στο επίπεδο 1 του SkipGraph. Η διαδικασία αυτή συνεχίζεται μέχρι να φτάσει στο ανώτερο επίπεδο. Η διαδικασία σύνδεσης χρειάζεται *O(log N)* βήματα και μηνύματα.

Ο παραπάνω αλγόριθμος συνδυάζεται με ένα ατομικό πρωτόκολλο όπως το πρωτόκολλο συντονισμού με κλείδωμα σε δύο φάσεις (two-phase locking). Αυτό γίνεται για να αποφευχθούν παρεμβολές στην κανονική δρομολόγηση σε συνθήκες ανταγωνισμού και αποτυχίας στη διάρκεια της διαδικασίας σύνδεσης. Οι ενεργοί κόμβοι καταγράφουν την διαδικασία σύνδεσης, όμως δεν την χρησιμοποιούν για κανονική δρομολόγηση μέχρι να λάβουν σήμα επιτυχής εκπλήρωσης της από τον συνδεόμενο κόμβο. Αν ο κόμβος αποτύχει πριν να γίνει η σύνδεση η καταγραφή αυτή λήγει μετά από κάποιο χρόνο. Ο κόμβος μπορεί να στείλει το σήμα επιτυχούς σύνδεσης μόλις εγκαταστήσει σύνδεση με τους δύο κοντινότερούς peers του επιπέδου 0.

➢ ***Ανάνηψη από αποτυχία***

Υπάρχουν πολλές αιτίες οπού ένα σύστημα χρειάζεται επιδιόρθωση όπως α) ένας κόμβος αποχωρεί εθελοντικά β) αποτυγχάνει χωρίς προειδοποίηση λόγω πτώσης του ή πτώσης της σύνδεσης του με το δίκτυο γ) το δίκτυο που ενώνει δυο peers αποτυγχάνει.

Για την επιδιόρθωση της δομής δεικτοδότησης όταν ένας κόμβος ‘εξαφανίζεται’, διατηρείται ένα σύνολο από ‘backup’ κόμβους για κάθε μηχάνημα. Το backup αυτό επιλέγεται από τους κοντινότερους κόμβους στο SkipGraph και ονομάζεται ***κοντινοί φίλοι (close buddies).*** Μια λειτουργία που τρέχει στο παρασκήνιο αντιγράφει τα δεδομένα ανάμεσα σε όλους τους κοντινούς φίλους. Αυτή η αντιγραφή εκτός από τη χρησιμότητα της σε περίπτωση αποτυχίας, επιπρόσθετα βοηθά στην μείωση της καθυστέρησης των ερωτημάτων καθώς ένας ανενεργός κόμβος μπορεί να εξυπηρετήσει ερωτήματα στη θέση ενός απασχολημένου φίλου.

Όταν ένας κόμβος αποχωρεί εθελοντικά μεταφέρει την περιοχή του στον κοντινότερο φίλο και μετά πληροφορεί όλους τους peers για αυτή την μεταφορά. Αποτυχία ή αποσύνδεση κάποιου κόμβου ανιχνεύεται από την περιοδική συνομιλία μεταξύ των peers. Συγκεκριμένα όταν ένας κόμβος Α συνεχόμενα λείπει από πολλές συζητήσεις, ο κόμβος Β που ανιχνεύει αυτήν την απουσία δρομολογεί πολλά μηνύματα ανίχνευσης διαμέσου άλλων peers. Αν κάποιο από αυτά τα μηνύματα φτάσει στο προορισμό τότε η σύνδεση μεταξύ των κόμβων Α και Β σημειώνεται ως αποκομμένη και κάθε άλλη περεταίρω επικοινωνία θα δρομολογείτε από παράκαμψη.

Αλλιώς αν κανένα από τα ανιχνευτικά μηνύματα είναι επιτυχημένο, ο κόμβος δηλώνεται νεκρός και απομακρύνεται από τον πίνακα δρομολόγησης. Κάθε μεταγενέστερο μήνυμα προς την περιοχή τους νεκρού κόμβου θα φτάνει στον κοντινότερο του φίλο στην αριστερή πλευρά. Έτσι αυτός ο κόμβος αναλαμβάνει την ευθύνη.

3 SkipIndex P2P Πλατφόρμα στο PlanetLab

Σε αυτή την ενότητα έχουμε υλοποιήσει μια πλατφόρμα P2P η οποία βασίζεται στο *SkipIndex** [2] το οποίο περιγράφηκε στο κεφάλαιο 2.2 και η οποία μπορεί να λειτουργήσει σε περιβάλλον *PlanetLab* [12] καθώς και σε ένα ή ομάδα από τοπικά PC. Συγκεκριμένα έχουμε υλοποιήσει την αναζήτηση *SkipIndex* όπως περιγράφετε στην ενότητα 2.2.2 Παρακάτω περιγράφουμε γενικά την υλοποίηση και τις αρχές λειτουργίας της.

3.1 Γενική περιγραφή

Η πλατφόρμα αποτελείτε από 4 μέρη τα οποία περιγράφονται παρακάτω.

3.1.1 Peer-to-Peer (P2P)

Είναι η καθαυτών εφαρμογή P2P που υλοποιεί τα ερωτήματα και εισαγωγές και τρέχει στους κόμβους του PlanetLab. Είναι μια Server-Client εφαρμογή, και ο Server είναι πολυνηματικός. Ανάλογα με την μεταβλητή ***multiport*** στην κλάση Variables όλοι οι peer servers είτε ακούνε στην ίδια θύρα 45967 (*multiport=false*), και αυτή είναι η έκδοση που χρησιμοποιούμε συνήθως στο PlanetLab. Ή αν *multiport=true* κάθε peer server ακούει σε διαφορετική θύρα η οποία είναι η 45967+peerID. Με αυτό τον τρόπο μπορούμε να ελέγξουμε την εφαρμογή σε ένα ή περισσότερα τοπικά PC ή ακόμα μπορούμε σε κάθε μηχάνημα του PlanetLab να τρέχουμε περισσότερους από ένα peers.

Το KD-δένδρο που κατασκευάζεται στις εισαγωγές των νέων κόμβων αποτελείτε από αντικείμενα της κλάσης Node και το αριστερό παιδί κάθε αντικειμένου είναι η διεύθυνση του peer που βρίσκετε εκεί. Βέβαια κανένας peer δεν έχει ολόκληρο το KD-δένδρο. Κάθε peer με την εισαγωγή του στο δίκτυο, αν η μεταβλητή ***addNewPointsFromNewPeer*** της κλάσης Variables είναι *true*, προσθέτει σε αυτό και τα δικά του σημεία τα οποία αν η μεταβλητή ***useStrings*** στη κλάση Variables είναι true, τότε τα σημεία είναι strings τα οποία διαβάζονται από ένα αρχείο και μετατρέπονται σε ακέραιους αριθμούς με hash για να αποθηκευτούν στο χώρο διαστάσεων του κατάλληλου peer. Αλλιώς αν η *useStrings* είναι false παράγονται τυχαία κάποια σημεία.

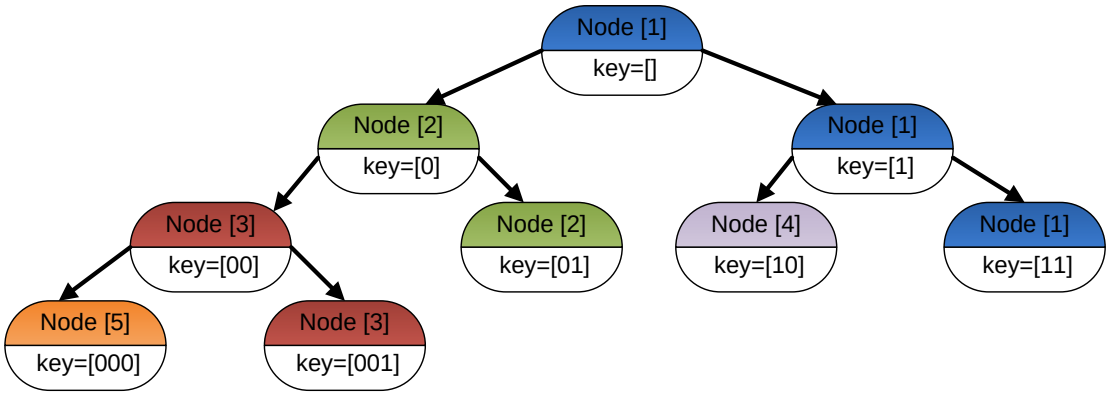
Ο χώρος κάθε peer στον οποίο αποθηκεύονται τα σημεία είναι D- διαστάσεων και το D καθορίζεται από την μεταβλητή ***dimension*** της κλάσης Variables. Έτσι μπορούμε να αποθηκεύουμε και να εκτελέσουμε ερωτήματα για στοιχεία με D χαρακτηριστικά, και το D το καθορίζουμε εμείς ανάλογα με τα στοιχεία που θέλουμε να αποθηκεύσουμε στο δίκτυο.

Σε κάθε peer, αν έχουμε το standard input και output του, μπορούμε να θέσουμε απευθείας ερωτήματα όπως να τυπώσει τα όρια του χώρου τον οποίο κατέχει καθώς και να εκτελέσουμε αναζήτηση για ένα σημείου σε όλο το δίκτυο με τον αλγόριθμο του skipIndex ή με flooding (δηλαδή ρωτώντας ένα ένα τους peers μέχρι να βρούμε το σημείο που ψάχνουμε). Ανάλογα με την μεταβλητή *useStrings* το σημείο για αναζήτηση δίνεται είτε ως συντεταγμένες από αριθμούς ή ως strings που μετατρέπονται στη συνέχεια σε ακέραιους με hash.

Πάντα υπάρχει σε κάθε peer μια ενημερωμένη και σωστά διατεταγμένη λίστα από όλους τους peers του συστήματος και τις διευθύνσεις τους. Η λίστα αυτή αποτελείτε από αντικείμενα της κλάσης ***Peer*** (η οποία το κλειδί διαχωρισμού και το τυχαίο κλειδί του peer, τη διεύθυνση και τη θύρα που ακούει) και περιγράφουμε στη *Β φάση εισαγωγής* πως κρατείτε πάντα ενημερωμένη. Με αυτήν τη λίστα και με βάση το τυχαίο κλειδί κάθε peer κατασκευάζουμε το ***SkipList*** του το οποίο χρησιμοποιείτε στις αναζητήσεις. Όλα μαζί τα SkipList αποτελούν το κατανεμημένο ***SkipGraph.*** Το τυχαίο κλειδί για ένα συγκεκριμένο peer είναι το ίδιο σε όλους τους peers του δικτύου. Κάθε επίπεδο του SkipList αποτελείτε από μια συνδεδεμένη λίστα την οποία ονομάζουμε ***LinkList***. Κάθε peer εκτός από το δικό του *SplitHistory*, το οποίο αποτελείτε από αντικείμενα της κλάσης ***SplitHistory***, διατηρεί και τα SplitHistory των γειτονικών του peer σε κάθε επίπεδο του *SkipList*, δηλαδή σε κάθε *LinkList*. Αυτά τα SplitHistory ενημερώνονται μετά από κάθε εισαγωγή νέου peer και έτσι πάντα είναι ενημερωμένα και ορθά. Περισσότερα για τα ερωτήματα περιγράφονται στην ενότητα 3.3

Επίσης το P2P σύστημα υλοποιεί τον αλγόριθμο *PLATON* για να διατηρούμε το δέντρο ισοζυγισμένο κατά την εισαγωγή νέων στοιχείων. Ο τρόπος υλοποίησης και ο αλγόριθμος περιγράφονται στο κεφάλαιο 4.

Στο σχήμα 3.1.1 εικονίζεται το global KD-δένδρο από αντικείμενα Node για ένα δίκτυο από 5 peers. Ολόκληρο αυτό το δέντρο δεν το έχει κανένας peer, αλλά κάθε peer έχει ένα κομμάτι του. Κάθε κομμάτι εμφανίζεται με διαφορετικό χρώμα και επίσης αναγράφεται σε ποιο peer ανήκει.



Σχήμα 3.1.1: Global KD-tree for 5 peers

Στο σχήμα 3.1.2 φαίνεται η λίστα από αντικείμενα Peer με όλους τους peers του δικτύου για το ποιο πάνω δένδρο. Από αυτή την λίστα σχηματίζεται το SkipGraph

Σχήμα 3.1.2: Peers list

Τέλος μερικά από τα request που εξυπηρετεί ο server του P2P είναι τα ακόλουθα:

- *join request* το οποίο στέλνεται από ένα νέο peer για την εισαγωγή του στο P2P δίκτυο.
- *SuccessfulJoin request* το οποίο λαμβάνεται από το νέο peer, μετά την επιτυχημένη εισαγωγή του στο δίκτυο. Αρχικά ο νέος peer στέλνει ένα join request στο ***SplitPeer*** ο οποίος θα διαχωρίσει την περιοχή του και θα δώσει στον νέο peer τη μίση. Την διεύθυνση του SplitPeer την παίρνει ο νέος peer από τον Bootstrap. Τέλος ο SplitPeer όταν τελείωση τη διαδικασία εισαγωγής επικοινωνεί πίσω με το νέο peer με ένα μήνυμα SuccessfulJoin request και του στέλνει όλες τις πληροφορίες που χρειάζεται ο νέος peer για να είναι ένα ομότιμο μέλος στο P2P δίκτυο.
- *updatePeersList request* το οποίο στέλνεται σε όλους τους peers του δικτύου, μετά την επιτυχημένη εισαγωγή ενός καινούριου peer, για να ενημερώσουν τη λίστα με τους peers που διατηρούν. Η πληροφορία που στέλνεται είναι δύο αντικείμενα Peer, του νέου peer και αυτού που διαχώρισε την περιοχή του, και με βάση αυτή την πληροφορία κάθε peer είναι σε θέση να ενημέρωση σωστά τη λίστα του. Μετά την ενημέρωση της λίστας με τους peers ενημερώνεται και το *SkipList*. Το πώς η πληροφορία αυτή αποστέλλεται σε όλους τους peers του συστήματος περιγράφεται σε επόμενο σημείο.
- *callfindPoint request* αποστέλλεται σε ένα peer μαζί με τις κατάλληλες παραμέτρους για να εκτελέσει τοπικά την μέθοδο *findPoint* της κλάσης *Node*. Κάθε τέτοια αποστολή μετρά ως ένα βήμα στην διαδικασία της αναζήτησης.
- *resultOfFindPoint request* λαμβάνεται από ένα peer ο οποίος έχει ξεκινήσει μια αναζήτηση και αποστέλλεται από τον peer στον οποίο έχει βρεθεί το αρχείο το οποίο αναζητούσε.
- *getSplitHistory request* αποστέλλεται από ένα peer ο οποίος πρέπει να ενημερώσει το SplitHistory σε κάποιο επίπεδο του *LinkList*, μετά την ενημέρωση του *LinkList* μετά από μια εισαγωγή νέου peer στο σύστημα. Αποστέλλετε στο νέο peer ή στον παλιό peer ο οποίος έχει κάνει το διαχωρισμό, αφού μόνο αυτοί έχουν καινούρια splitHistory τα οποία δεν γνωρίζουν οι υπόλοιποι. Ή αποστέλλεται από κάποιο νέο peer ο οποίος φτιάχνει το *LinkList* του.
- *Kill request* λαμβάνεται από ένα peer ο οποίος θέλουμε να τερματιστεί. Χρησιμοποιείται από το control center το οποίο περιγράφεται παρακάτω.
- *queryArgs request* αποστέλλετε από το control center, μαζί με τους παραμέτρους αναζήτησης, σε ένα peer στον οποίο θέλουμε να ξεκινήσουμε μια αναζήτηση.
- *printMySpace request* αποστέλλετε από το control center σε ένα peer για τον οποίο θέλουμε να μάθουμε τις συντεταγμένες του χώρου τον οποίο κατέχει.
- *addNewPoints request* αποστέλλετε από ένα νέο peer λίγο πριν κάνει join, για να στείλει τα σημεία του σε όλους τους υπόλοιπους peers και ο κάθε ένας να αποθήκευση αυτά που ανήκουν στο χώρο του.

- *flooding request* αποστέλλεται από ένα peer, ο οποίος εκτελεί flooding ερώτημα, σε όλους τους υπόλοιπούς με την σειρά για να κοιτάξουν αν έχουν το σημείο το οποίο ψάχνει.
- *getThePoints request* αποστέλλετε σε ένα peer από τον οποίο θέλουμε να μας στείλε κάποιο συγκεκριμένο αριθμό από τα σημεία του και τα οποία είναι τα πιο κοντινά σε ένα άξονα τον οποίο στέλνουμε ως παράμετρο. Χρησιμοποιείται από τον αλγόριθμο PLATON
- *loadBalance request* αποστέλλετε σε ένα peer μαζί με τις κατάλληλες παραμέτρους, για να εκτελέσει του αναδρομικού αλγόριθμο PLATON load balance.
- *changeSpace request* αποστέλλετε σε ένα peer για να τροποποιήσει κατάλληλα τα όρια της περιοχής του μετά από μια μετακίνηση άξονα που έγινε κατά την εκτέλεση του αλγόριθμου PLATON
- *changeSpace_RemovePoints request* λαμβάνεται από ένα peer για να τροποποιήσει κατάλληλα τα όρια της περιοχής του και να απομακρύνει τα τυχών σημεία του που δεν είναι πλέων στην περιοχή του. Δημιουργείτε μετά από μια μετακίνηση άξονα που έγινε κατά την εκτέλεση του αλγόριθμου PLATON
- *updateSplitHistory request* αποστέλλετε σε ένα peer, μαζί με το τροποποιημένο αντικείμενο *SplitHistory* , για να ενημερώσει το SplitHistory tree του καθώς και όλα τα SplitHistory που διατηρεί για τους γείτονες του στο *SkipList*, μετά από μια μετακίνηση άξονα που έγινε κατά την εκτέλεση του αλγόριθμου PLATON.

3.1.2 StartPeer Server (SPS)

Είναι μια απλή εφαρμογή server, η οποία τρέχει συνέχεια στους κόμβους του PlanetLab. Ο server ακούει στην θύρα 12225 και δεν είναι πολυνηματικός. Εξυπηρετεί τα παρακάτω δύο requests.

- *startPeer request* αποστέλλεται από το control center για να εκκινήσει ο SPS το P2P το οποίο βρίσκεται στο ίδιο μηχάνημα με τον SPS.
- *kill request* λαμβάνεται από τον SPS και τερματίζεται.

3.1.3 Bootstrap Server

Είναι ένας server ο οποίος ακούει στην θύρα 46998 και τρέχει σε ένα μηχάνημα του PlanetLab και χρειάζεται για την εισαγωγή των νέων peer στο δίκτυο. Κάθε peer αμέσως μόλις τρέξει επικοινωνεί μαζί του με ένα *join request*. Αν είναι ο πρώτος peer του συστήματος απλά ενημερώνεται για αυτό και κάνει μόνος του την εισαγωγή, αλλιώς λαμβάνει μια ενημερωμένη, για την ώρα, λίστα (από αντικείμενα Peer) με όλους τους peers του συστήματος καθώς και τη διεύθυνση του ***SplitPeer*** με τον οποίο θα επικοινωνήσει με ένα join request και θα ξεκινήσει η διαδικασία εισαγωγής του νέου peer. Ο SplitPeer είναι αυτός που θα διαμοιράσει την περιοχή του και θα δώσει την μισή στον νέο peer. Και στις δύο περιπτώσεις ο Bootstrap αποστέλλει στο νέο peer την διεύθυνση, και την θύρα που ακούει το control center

Ο Bootstrap μπορεί να βρει ποιος είναι ο SplitPeer από την λίστα με τους peers που διατηρεί καθώς και με τη γνώση του προηγούμενου SplitPeer. Συγκεκριμένα ο *SplitPeer* είναι ο peer που κατέχει το αριστερότερο φύλλο στο K-D δένδρο το οποίο έχει την μεγαλύτερη περιοχή, δηλαδή δεν έχει ακόμα διαμοιραστεί για το επόμενο επίπεδο. Π.χ. στο σχήμα 3.1.1 για τον καινούριο peer 6 ο SplitPeer είναι ο 2 διότι η περιοχή του είναι η αριστερότερη περιοχή η οποία δεν έχει διαμοιραστεί ακόμα.

Μετά την επιτυχημένη εισαγωγή του, ο νέος peer επικοινωνεί με τον Bootstrap με ένα *updatePeersList request* και του αποστέλλει δύο αντικείμενα Peer, του νέου peer και του SplitPeer, έτσι ο Bootstrap ενημερώνει τη λίστα του.

3.1.4 Control Center

Είναι μια εφαρμογή η οποία τρέχει στο τοπικό PC και μπορούμε με αυτή να κάνουμε μαζικά τα insert των peers, διαβάζοντας τις διευθύνσεις των κόμβων από ένα αρχείο. Έτσι δεν χρειάζεται να ενωθούμε με ssh σε κάθε μηχάνημα ξεχωριστά και να τρέχουμε το P2P. Επίσης μπορούμε να εκτελέσουμε ερώτημα αναζήτησης σε κάποιο peer, δίνοντας τις συντεταγμένες ή strings ανάλογα με τη μεταβλητή *useStrings* της κλάσης Variables. Καθώς και ερώτημα για τις συντεταγμένες του χώρου κάποιου peer και να παίρνουμε τα αποτελέσματα πίσω. Για το ερώτημα αναζήτησης παίρνουμε πίσω την διαδρομή από τους peers την οποία ακολούθησε το ερώτημα, τα βήματα που έκανε και τον χρόνο που διάρκεσε. Επίσης μπορούμε να στείλουμε μαζικό σήμα τερματισμού σε όλους τους P2P που τρέχουν καθώς και σε όλους τους SPS. Η εφαρμογή αυτή είναι μια server-client εφαρμογή και ο server ακούει στην θύρα 36418. Όταν ο αλγόριθμος Platon είναι ενεργός, σε κάθε εκτέλεση του το control center μαζεύει από όλους τους peers διάφορες πληροφορίες για την εκτέλεση του αλγόριθμου και στο τέλος παρουσιάζονται τα συνολικά στατιστικά στοιχεία της εκτέλεσης . Περισσότερες πληροφορίες δίνονται στο κεφάλαιο 4

Τις διευθύνσεις των peers για τα ερωτήματα μπορούμε να τις πάρουμε από το αρχείο, από το οποίο έγιναν και τα insert ή σε περίπτωση multi ports από τον Bootstrap μαζί και με τη θύρα που ακούει κάθε peer.
Ο server αυτός εξυπηρετεί τα ακόλουθα request:

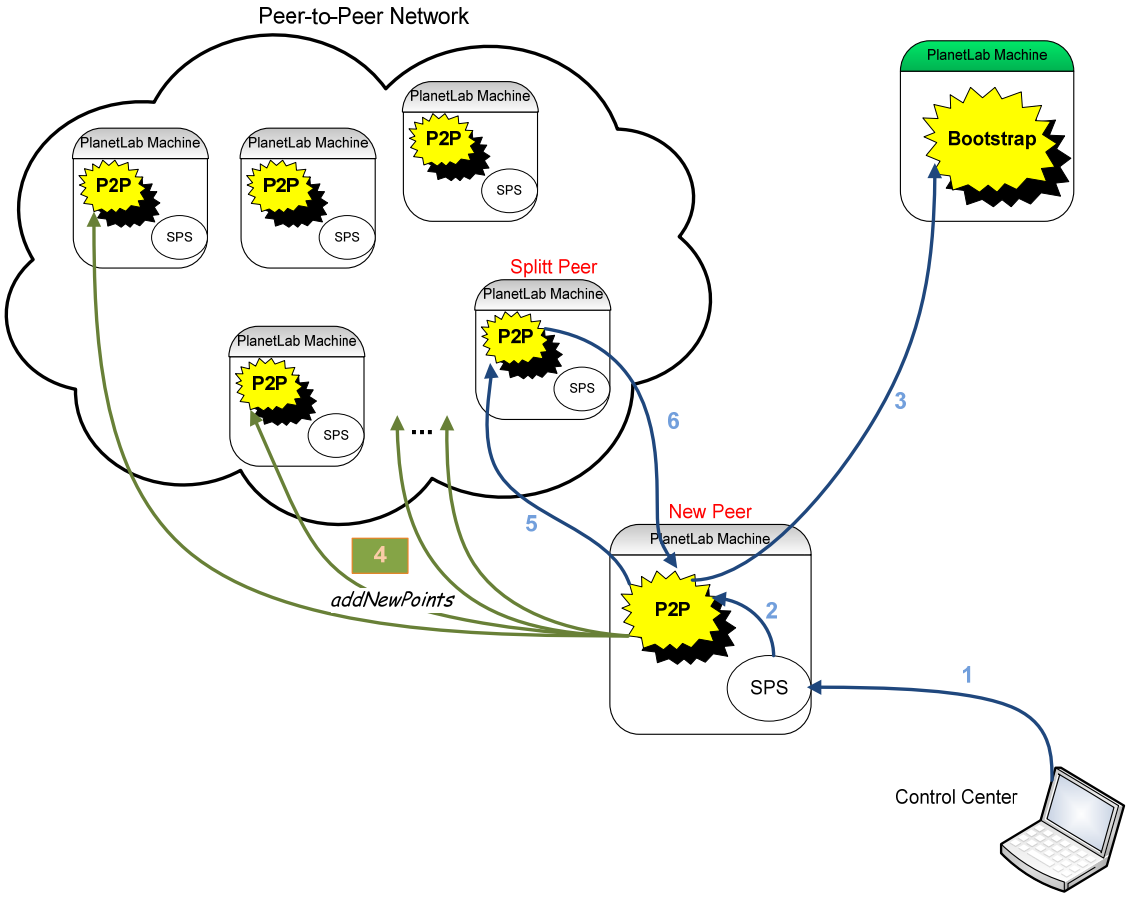
- *successJoin request* το οποίο λαμβάνεται όταν η τρέχων διαδικασία εισαγωγής ενός peer έχει τελειώσει και μπορεί να συνεχίσει στην εισαγωγή του επόμενου peer. Επίσης λαμβάνεται και το ID του peer που μόλις συνδέθηκε για έλεγχο σφαλμάτων.
- *findPointResult request* το οποίο αποστέλλεται από τον peer στον οποίο έχει βρεθεί το σημείο το οποίο αναζητούσαμε και αποστέλλονται τα αποτελέσματα της αναζήτησης.

3.2 Εισαγωγή Νέου Peer

Η εισαγωγή ενός νέου peer στο σύστημα γίνεται σε δύο φάσεις οι οποίες περιγράφονται παρακάτω.

3.2.1 Α Φάση Εισαγωγής

Στην Α φάση εισαγωγής ο νέος peer συνδέεται στο δίκτυο, παίρνει την περιοχή του και όλες τις άλλες πληροφορίες που χρειάζεται. Η φάση αυτή φαίνεται σχηματικά στο σχήμα 3.2.

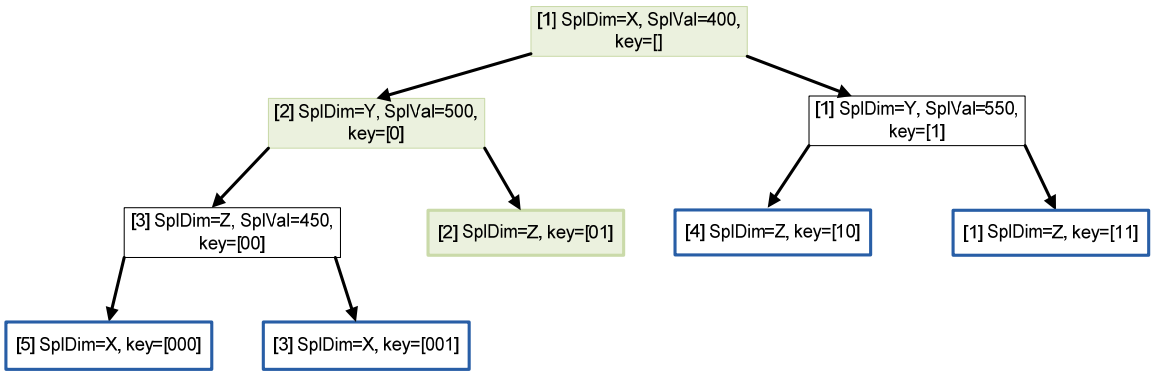


Σχήμα 3.2: Α Φάση Εισαγωγής Νέου Peer

Αρχικά στο βήμα 1 το control center στέλνει *startPeer request* στον SPS που τρέχει στο ίδιο μηχάνημα με τον peer που θέλουμε να εισάγουμε στο δίκτυο. Ακολούθως στο βήμα 2 ο SPS εκκινεί τον P2P και στην συνέχεια ο P2P στο βήμα 3 επικοινωνεί με το Bootstrap server για να του δώσει τις απαραίτητες πληροφορίες, που περιγράψαμε προηγουμένως,(ενότητα 3.1.3) για την εισαγωγή του στο δίκτυο. Το βήμα 4 (το οποίο μπορεί να απενεργοποιηθεί με την μεταβλητή *addNewPointsFromNewPeer* στην κλάση Variables) συμβαίνει μόνο αν η μεταβλητή *loadBalance* της κλάσης Variables είναι false αλλιώς αν είναι true περιγράφουμε στο κεφάλαιο 4 πως γίνεται η εισαγωγή των νέων στοιχείων. Ο νέος peer στο βήμα 4

στέλνει σε όλους τους υπόλοιπους peers που βρίσκονται στο δίκτυο ένα *addNewPoints request* μαζί με τα σημεία που διαθέτει και κάθε peer που λαμβάνει αυτό το request αποθηκεύει τα σημεία που ανήκουν στο χώρο του. Στη συνέχεια στο βήμα 5 ο νέος peer επικοινωνεί με τον *SplitPeer* με ένα *join request*, του οποίου την διεύθυνση έχει πάρει από τον Bootstrap. Ο *SplitPeer* διαχωρίζει την περιοχή του για δώσει στον νέο peer τη μίση. Δίνει στο νέο peer το αριστερό κομμάτι της περιοχής του, δηλαδή τα χαμηλότερα νούμερα, και ο κωδικός διαχωρισμού του νέου peer προκύπτει με την προσθήκη 0 στο κωδικό διαχωρισμού του παλιού peer. Ο κωδικός του παλιού peer προκύπτει με προσθήκη 1 στον προηγούμενο κωδικό του. Η τιμή διαχωρισμού (split value) προκύπτει ως το μέσω των υπάρχων σημείων του παλιού peer, δηλαδή οι δύο περιοχές που προκύπτουν να έχουν τον ίδιο αριθμό σημείων. Τέλος φτιάχνει το πρώτο αντικείμενο Node του νέου peer και στο βήμα 6 επικοινωνεί πίσω μαζί του, με ένα *SuccessfulJoin request*, και του στέλνει το αντικείμενο Node, το splitHistory του νέου peer καθώς και τα δύο αντικείμενα Peer, το δικό του και του νέου peer έτσι ώστε να μπορεί ο νέος peer να ενημερώσει την λίστα που έχει πάρει από τον Bootstrap. Όλες οι πληροφορίες διαχωρισμού (όπως splitValue, splitDimension, key) φυλάσσονται σε αντικείμενα SplitHistory τα οποία δημιουργούν ένα δέντρο και χρησιμοποιούνται για τον καθορισμό της περιοχής ενός peer στις αναζητήσεις, όπως ακριβώς περιγράφει το skipIndex[2] (κεφάλαιο 2.1). Ο νέος peer έχει συνδεθεί επιτυχώς στο δίκτυο και τώρα μένει να φτιάξει το *SkipList* του και να ενημερώσει τους υπόλοιπους peer για την εισαγωγή αυτή.

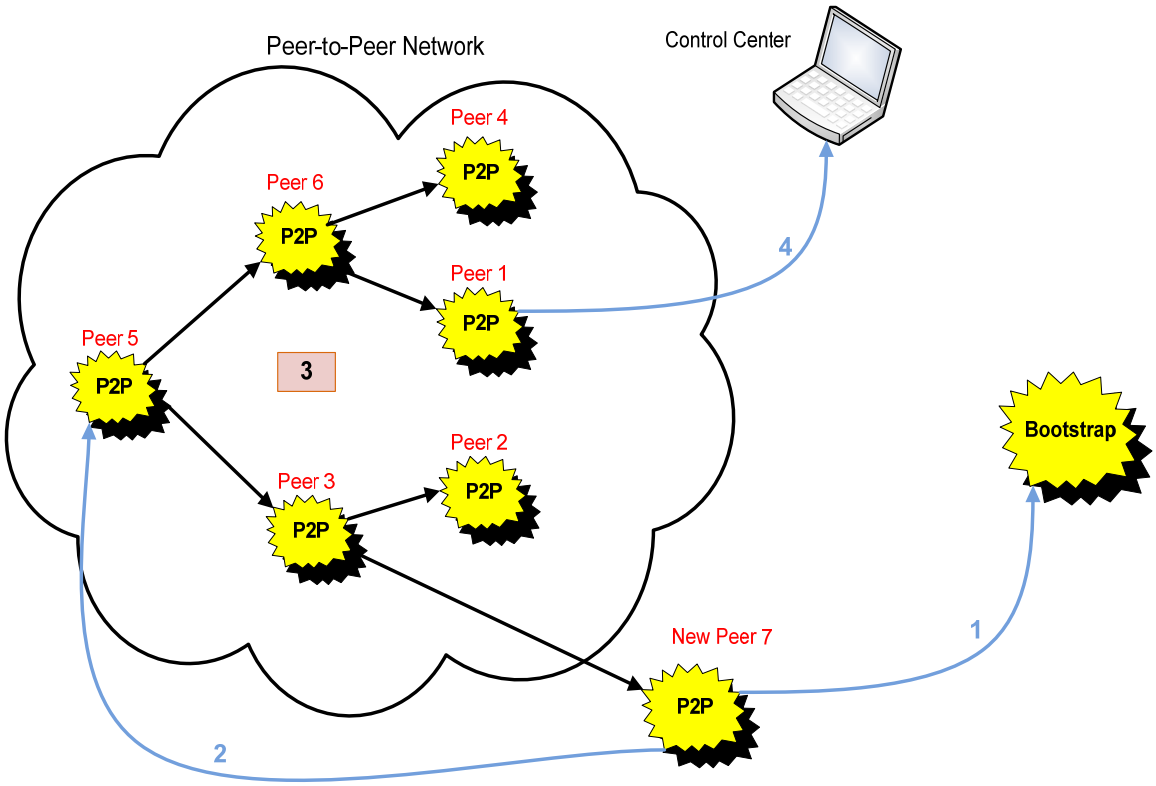
Ένα παράδειγμα 3-διάστατου δέντρου για 5 peers φαίνεται στο σχήμα 3.3. Κάθε αντικείμενο SplitHistory κρατά τη διάσταση όπου έγινε ο διαχωρισμός (SplDim) την τιμή στο άξονα όπου έγινε ο διαχωρισμός (SplVal) το id του peer καθώς και το κλειδί. Τα φύλλα είναι περιοχές των peer όπου δεν έγινε ακόμα διαχωρισμός για αυτό δεν υπάρχει SplVal, υπάρχει όμως SplDim διότι ξέρουμε σε ποια διάσταση θα κάνουμε τον επόμενο διαχωρισμό. Το σκιασμένο μέρος του δέντρου είναι το μερικό SplitHistory tree του peer 2 και αυτό μονό χρειάζεται για να καθορίσει το χώρο του. Βέβαια στο *SkipList* διατηρεί και τα μερικά SplitHistory tries των γειτόνων του σε κάθε επίπεδο. Π.χ. για το επίπεδο 0 θα χρειαστεί των peers 3 και 4.



Σχήμα 3.3: 3D SplitHistory tree

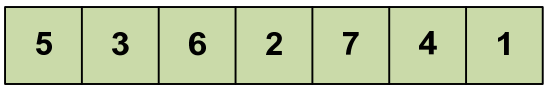
3.2.2 Β Φάση Εισαγωγής

Στην Β φάση εισαγωγής ο νέος peer πρέπει να ενημερώσει όλους τους υπόλοιπους peer του δικτύου καθώς και τον Bootstrap για την εισαγωγή αυτή. Συγκεκριμένα πρέπει να τους στείλει 2 αντικείμενα Peer, του νέου peer (του ιδίου δηλαδή) και του SplitPeer (για να μπορούν να ξέρουν που ακριβώς τοποθετήθηκε ο νέος peer) έτσι ώστε να μπορέσουν να ενημερώσουν τη λίστα με όλους τους peers που έχουν. Μετά την ενημέρωση τις λίστας αυτής κάθε peer ξεχωριστά ενημερώνει το *SkipList* του. Μετά την ενημέρωση του *SkipList* κάθε peers ελέγχει όλα τα επίπεδα του *SkipList* και αν σε κάποιο από αυτά έχει γείτονα το νέο peer ή τον SplitPeer, πρέπει να στείλε σε αυτόν ένα *getSplitHistory request* για να πάρει το SplitHistory του διότι είτε δεν το έχει (νέος peer) ή το έχει αλλά αυτό έχει αλλάξει (SplitPeer). Βέβαια μια φορά θα στείλουμε το *getSplitHistory request* στον ίδιο peer ανεξάρτητα σε πόσα διαφορετικά επίπεδα υπάρχει ως γείτονας. Το ίδιο συμβαίνει και στην κατασκευή του *SkipList* από το νέο peer. Η φάση αυτή φαίνεται σχηματικά στο σχήμα 3.4, με ένα παράδειγμα κατά την εισαγωγή του peer 7. Για χάρη ευκολίας δεν φαίνεται όλο το μηχάνημα με το SPS αλλά μόνο τα P2P.

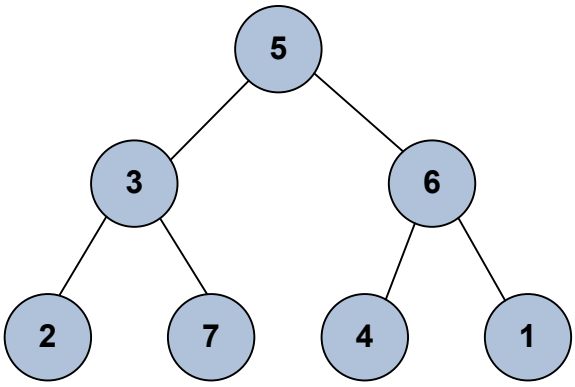


Σχήμα 3.4: Β Φάση Εισαγωγής Νέου Peer

Αρχικά στο βήμα 1 ο νέος peer ενημερώνει τον Bootstrap, στην συνέχεια παίρνει την ενημερωμένη λίστα με τους peers και σχηματίζει ένα εικονικό δένδρο. Ο σχηματισμός αυτού του δένδρου φαίνεται παρακάτω με ένα παράδειγμα με τους 7 peers.



Σχήμα 3.5.1: Η λίστα με τους peers



Σχήμα 3.5.2: Το εικονικό δένδρο από τη λίστα

Στο βήμα 2 ο νέος peer στέλνει ένα *updatePeersList request*, μαζί με τα δύο αντικείμενα peer που προαναφέραμε, στον peer ρίζα του εικονικού δένδρου, στο παράδειγμα μας στον peer 5. Ο peer 5 αφού ενημερώσει την λίστα του, σχηματίζει το ίδιο δένδρο και βλέπει ότι έχει για παιδιά τους peer 3 και 6 στους οποίους στέλνει με την σειρά του το *updatePeersList request* . Η διαδικασία αυτή συνεχίζεται μέχρι να ενημερωθούν όλοι οι peers του συστήματος, στο σχήμα 3.4 η διαδικασία αυτή εικονίζεται ως βήμα 3. Η διαδικασία αυτή χρειάζεται logN βήματα για να ενημερωθούν όλοι οι peers και αποφεύγουμε με αυτό τον τρόπο την περίπτωση ο νέος peer να πρέπει να ενημερώσει ένα ένα όλους τους peers του δικτύου, το οποίο θα ήταν πιο χρονοβόρο. Ο νέος peer καθώς και ο SplitPeer, έχουν ήδη ενημερωμένη τη λίστα και όταν θα λάβουν το *updatePeersList request* απλά θα το προωθήσουν στα παιδία τους αν αυτά υπάρχουν. Τέλος όταν ο τελευταίος peer της λίστας (Ο peer 1 στο παράδειγμα μας) λάβει το request αυτό, στέλνει στο control center ένα *successJoin request* για να τον ενημέρωση ότι η διαδικασία εισαγωγής έχει τελειώσει επιτυχώς και ότι τώρα μπορεί να συνεχίζει με την επόμενη εισαγωγή. Στο σχήμα 3.4 αυτό φαίνεται ως βήμα 4.

Βέβαια με αυτά τα δεδομένα μπορεί να ξεκινήσει η επόμενη εισαγωγή χωρίς να έχουν ενημερωθεί όλοι οι peers. Αυτό έχει ως συνέπεια σε κάποιους peers να φτάνουν ενημερώσεις ταυτόχρονα ή ακόμα και με λανθασμένη σειρά! Ή να φθάνουν join request πριν να έχει φτάσει μια προηγούμενη ενημέρωση. Εξασφαλίζουμε ότι σε κάθε peer όλα τα update και join request θα εκτελεστούν με τη σωστή σειρά, καθώς και ότι δεν θα εκτελούνται ταυτόχρονες ενημερώσεις, συγχρονίζοντας τα νήματα που εξυπηρετούν αυτά τα request με τη βοήθεια σηματοφόρων και conditions.

Αξίζει να σημειωθεί ότι αυτή η ενημέρωση κοστίζει N μηνύματα στο δίκτυο (όπου N ο συνολικός αριθμός των peers). Στο SkipIndex ενότητα 2.2.4 αναφέρει ότι η ενημέρωση μπορεί να γίνει με logN μηνύματα. Μπορούσε να γίνει πιο εύκολα η ενημέρωση με logN μηνύματα απλά ενημερώνοντας ο νέος peer τους γείτονες του για την παρουσία του, καθώς και ο SplitPeer για την αλλαγή του SplitHistory του, σε όλα

τα επίπεδα του SkipList τους. Έχουμε υλοποιήσει όμως με αυτό τον τρόπο την ενημέρωση διότι η υλοποίηση του αλγόριθμου load balance PLATON που περιγράφεται στο κεφάλαιο 4 χρησιμοποιεί την λίστα με όλους τους peers για να βρει του οριακούς peers σε κάθε επίπεδο.

Ένα άλλο θέμα όμως με την logN ενημέρωση είναι η περίπτωση που ο peer θα θέλει να προσθέσει νέο επίπεδο στην *SkipList* του, τότε θα κοιτάζει τους γείτονες του στο χαμηλότερο επίπεδο αν είναι γείτονες και στο επόμενο. Αν δεν είναι αυτοί τότε θα τους προωθεί ένα μήνυμα για να ψάξουν τους δικούς τους γείτονες σε αυτό το επίπεδο για να δούναι αν αυτοί είναι οι γείτονες του. Δηλαδή αν θέλει ένας peer να φτιάξει τη λίστα στο επίπεδο K δεν θα αποστέλλεται η αίτηση αναζήτησης του γείτονα μέσω του επιπέδου 0 αλλά μέσω του επιπέδου K-1, το οποίο θα είναι και πιο αποδοτικό. Αυτό συμβαίνει διότι οποιοσδήποτε peer δεν είναι στην λίστα του επιπέδου K-1 τότε αποκλείετε να είναι στην λίστα του επιπέδου K και δεν χρειάζεται καν να τον ρωτήσουμε.

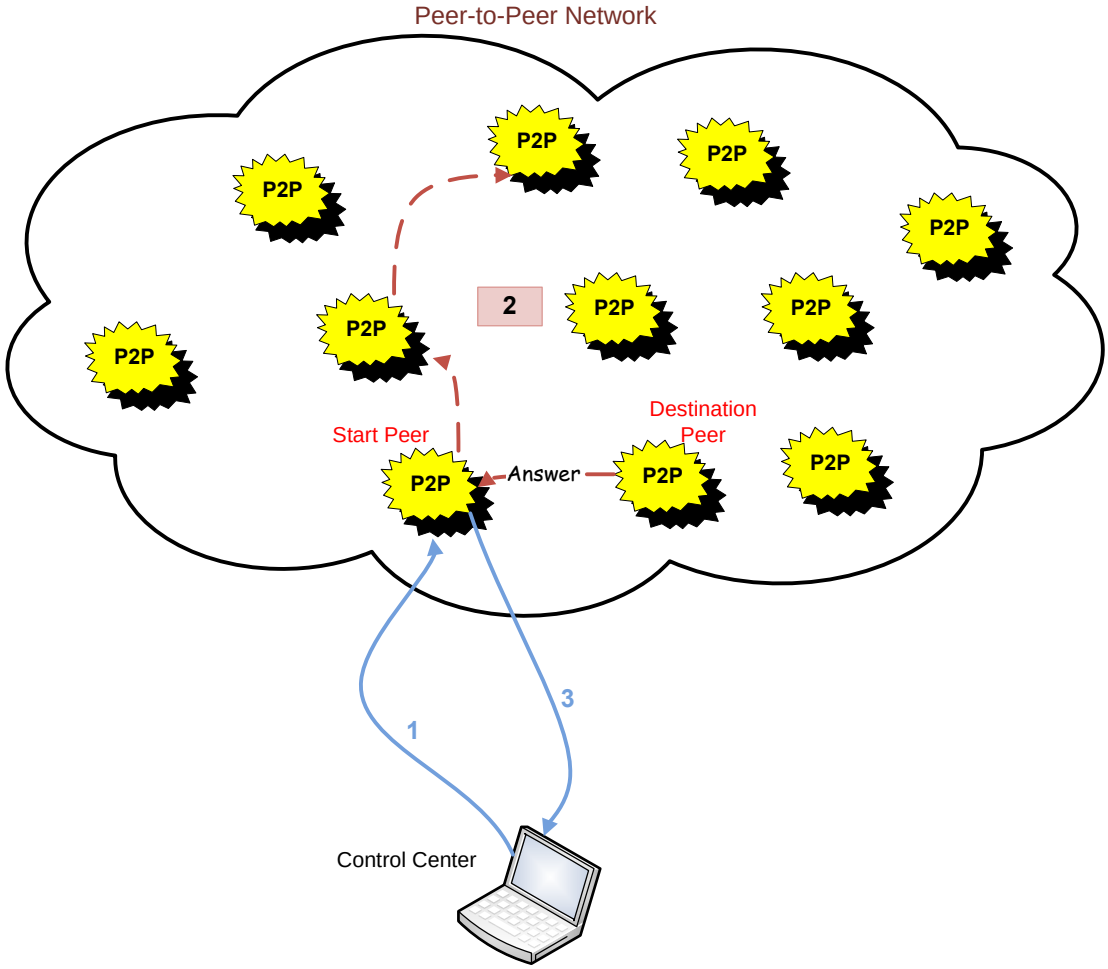
Βέβαια μπορεί να γίνει ο σχηματισμός του *SkipList* χωρίς να έχει ο νέος peer ολόκληρη τη λίστα με τους υπόλοιπους peer (που τώρα παίρνει από τον Bootstrap). Σε αυτή την περίπτωση μπορεί να σχηματίσει τη λίστα του επιπέδου 0 από τις πληροφορίες που θα πάρει από τον SplitPeer. Αυτό μπορεί να γίνει αφού δεξιά του νέου peer στο επίπεδο 0 είναι ο SplitPeer, ενώ αριστερά του είναι ο πρώην αριστερός γείτονας του SplitPeer. Έτσι αφού σχηματίσει τη λίστα του επιπέδου 0 μπορεί να δρομολογήσει κάποιες αιτήσεις αναζήτησης σε αυτούς τους γείτονες έτσι ώστε να βρούνε τους γείτονες του στο επίπεδο 1. Στη συνέχεια αφού βρεθούν οι γείτονες του στο επιπέδου 1 δρομολογεί σε αυτούς αιτήσεις αναζήτησης για τους γείτονες του στο επόμενο επίπεδο. Οι νέες αυτές αιτήσεις δρομολογούνται στο επίπεδο 1 του SkipGraph, όπως ακριβώς περιγράφετε στην προηγούμενη παράγραφο. Αυτή η διαδικασία συνεχίζεται μέχρι να φτάσει στο ψηλότερο επίπεδο στο οποίο είναι μόνος του. Αυτές οι αιτήσεις όταν φτάσουν στους γείτονες του νέου peer τότε μαθαίνουν ότι νέος peer είναι γείτονας τους και πρέπει να ενημερώσουν και αυτοί το *SkipList* τους. Αυτός ο τρόπος είναι υλοποιημένος επίσης στην πλατφόρμα SkipIndex ως μια επιπλέων επιλογή όταν ο αλγόριθμος Platon είναι απενεργοποιημένος.

3.3 Αναζήτηση

Η αναζήτηση λειτουργεί όπως ακριβώς περιγράφετε από το SkipIndex[2] στην ενότητα 2.2.2. Από την λίστα με όλους του peers του δικτύου που διατηρεί κάθε peer σχηματίζει το *SkipList*. Κάθε peer έχει ένα τυχαίο κλειδί από 0 και 1, το οποίο βρίσκεται στο αντικείμενο Peer και είναι ίδιο για το συγκεκριμένο peer σε όλους τους υπόλοιπους peers. Κάθε peer συγκρίνει τα K πρώτα ψηφία του κλειδιού του με τα K πρώτα ψηφία των κλειδιών των υπόλοιπων peers και αν κάποιου είναι τα ίδια τότε τον εισάγει σε αυτό το επίπεδο του *SkipList*. Όπου K το επίπεδο του *SkipList (LinkList)*, π.χ. για το επίπεδο 0 συγκρίνουμε 0 ψηφία και άρα όλοι οι peers μπαίνουν, για το επίπεδο 1 συγκρίνουμε 1 ψηφίο και άρα υπάρχει πιθανότητα $\frac{1}{2}$ να μπει

κάποιος peer κοκ. Για να βρεθεί ένας peer στο επίπεδο K υπάρχει πιθανότητα $\frac{1}{2^K}$, ενώ ένας peer που βρίσκετε στο επίπεδο K μπορεί να βρεθεί στο επίπεδο K+1 με πιθανότητα $\frac{1}{2}$. Βέβαια ο peer στον οποίο ανήκει το συγκεκριμένο *SkipList* βρίσκετε σε όλα τα επίπεδα του. Όπως αναφέρθηκε και προηγουμένως κάθε peer διατηρεί το SplitHistory tree της περιοχής του καθώς και όλως των γειτόνων του στο *SkipList*. Έτσι από αυτά μπορεί να ξέρει αν ένα σημείο είναι μέσα σε αυτές τις περιοχές δεξιά ή αριστερά τους.

Τώρα το ερώτημα μπορεί να γίνει από το control center δίνοντας τον αριθμό του peer στον οποίο θέλουμε να ξεκινήσει η αναζήτηση καθώς και το σημείο που ψάχνουμε. Το D-διαστάσεων σημείο μπορεί να δοθεί ως συντεταγμένες ή σαν strings ανάλογα με την μεταβλητή *useStrings*. Η διαδικασία φαίνεται σχηματικά στο σχήμα 3.6. Φυσικά το ερώτημα μπορεί να τεθεί και απευθείας στον peer αν έχουμε το standard input και output του. Στο βήμα 1 το control center στέλνει ένα *queryArgs request,* μαζί με τις συντεταγμένες του σημείου που ψάχνουμε, στον ***StartPeer*** ο οποίος θα ξεκινήσει την αναζήτηση. Ο StartPeer κοιτάζει αρχικά αν έχει αυτός το σημείο αυτό, αν όχι ξέρει σε ποιο κατεύθυνση βρίσκετε η περιοχή που περιέχει το σημείο αυτό. Έτσι κοιτάζει αρχικά στο ψηλότερο επίπεδο του *SkipList* του το γείτονα του προς αυτή την κατεύθυνση. Από το splitHistory του γείτονα του το οποίο διαθέτει, κοιτάζει αν δεν προσπέρνα η περιοχή του το σημείο αυτό και αν όχι τότε του στέλνει ένα c*allfindPoint request.* Αλλιώς εκτελεί την ίδια διαδικασία σε χαμηλότερα επίπεδα του *SkipList*. Δηλαδή από το *SkipList* βρίσκει τον πιο μακρινό peer, ο οποίος όμως δεν προσπερνά το σημείο αυτό και του στέλνει ένα c*allfindPoint request* μαζί με τις κατάλληλες παραμέτρους. Ο peer αυτός με την σειρά του εκτελεί την ίδια διαδικασία. Η όλη διαδικασία συνεχίζεται μέχρι το c*allfindPoint request* φτάσει στον destination peer ο οποίος κατέχει το χώρο στον οποίο ανήκει το σημείο αυτό. Ο destination peer στέλνει στον StartPeer ένα *resultOfFindPoint request* για να του επιστρέψει τα αποτελέσματα της αναζήτησης. Αν έχει βρεθεί το σημείο, ο destination peer στέλνει και ένα αντικείμενο Point το οποίο είναι το σημείο που αναζητούσαμε αλλιώς στέλνει null σε αυτή την παράμετρο. Στο σχήμα 3.6 η διαδικασία αυτή φαίνεται ως βήμα 2 και κοστίζει O(logN) βήματα, όπως περιγράφεται στο SkipIndex[2], και όπου N ο συνολικός αριθμός των peers στο δίκτυο. Τέλος στο βήμα 3 ο StartPeer στέλνει ένα *findPointResult request* στο control center για να του επιστρέψει τα αποτελέσματα της αναζήτησης. Τέλος στο control center παρουσιάζεται το σημείο, ο start και destination peer, όλοι οι ενδιάμεσοι κόμβοι από τους οποίους πέρασε το ερώτημα καθώς και ο χρόνος που διήρκεσε το ερώτημα μέχρι η απάντηση να ληφθεί από τον StartPeer. Αν το σημείο έχει βρεθεί και η αναζήτηση έγινε με strings παρουσιάζεται και ο owner του σημείου αυτού.



Σχήμα 3.6: Διαδικασία αναζήτησης

3.4 Γενική Περιγραφή της Ανάπτυξης και Λειτουργίας της Εφαρμογής

Για να θέσουμε σε λειτουργία την εφαρμογή σε πολλούς κόμβους του PlanetLab χρησιμοποιούμε το εργαλείο PlMan[13]. Με αυτό μπορούμε να ανεβάσουμε ταυτόχρονα στους κόμβους που θέλουμε την εφαρμογή, (P2P και PeerServer) και να εκτελέσουμε ταυτόχρονα σε όλου τους κόμβους τον PeerServer. Επίσης με τη βοήθεια αυτού του εργαλείου μπορούμε να δούμε την έξοδο κάθε peer, δηλαδή όλα τα μηνύματα που τυπώνει, και άρα να παρακολουθήσουμε όλη τη διαδικασία της εισαγωγής της εκτέλεσης ερωτημάτων και του Load Balancing. Βέβαια πριν ανεβάσουμε την εφαρμογή πρέπει να την παραμετροποιήσουμε θέτοντας κατάλληλα τις μεταβλητές στην κλάση Variables. Επίσης κατά την εκκίνηση του Bootstrap πρέπει να του δώσουμε ως παράμετρο την σωστή διεύθυνση του Control Center.

Ακολούθως από το Control Center μπορούμε να ελέγξουμε την εφαρμογή. Να εκτελέσουμε τις εισαγωγές των peers αυτοματοποιημένα, να εκτελέσουμε ερωτήματα καθώς και να τερματίσουμε την εφαρμογή σε όλους τους κόμβους. Ένα παράδειγμα εκτέλεσης του Control Center παρουσιάζεται παρακάτω. Με κόκκινο bold χρώμα σημειώνονται ότι εισάγει ο χρήστης.

```
Give the number of peers: 4
The control center server is started...

Do you want to make the inserts [in], make query [qu],
kill the Peers [kp], kill the PeersServer [ks] or exit [exit] ?
in
Now I will start 4 peers...

I will start the peer(planetlab1.cs.unibo.it)  1
Attempting connection to planetlab1.cs.unibo.it, port=12225
Now I wait for peer 1
I get successJoin
The peer 1 was successful joined [OK]

I will start the peer(planetlab1.cs.purdue.edu)  2
Attempting connection to planetlab1.cs.purdue.edu, port=12225
Now I wait for peer 2
I get message from : 2, 1,

***************Results for PLATON [2]******************

All the message to border are 1, all the move points are 55 and the time is:
0.995 sec (995.0 msec)
The new points are :120, the Threshold is :0.0 and the load are:

Peer[2] load=65
Peer[1] load=66

***************END of PLATON [2] results***************

The peer 2 was successful joined [OK]

I will start the peer(planetlab2.cs.purdue.edu)  3
Attempting connection to planetlab2.cs.purdue.edu, port=12225
Now I wait for peer 3
I get successJoin
The peer 3 was successful joined [OK]
```

```
I will start the peer(planet5.berkeley.intel-research.net)  4
Attempting connection to planet5.berkeley.intel-research.net, port=12225
Now I wait for peer 4
I get message from : 3, 2, 4, 1,

***************Results for PLATON [4]******************

All the message to border are 5, all the move points are 138 and the time
is: 4.791 sec (4791.0 msec)
The new points are :800, the Threshold is :0.0 and the load are:

Peer[3] load=232
Peer[2] load=233
Peer[4] load=233
Peer[1] load=233

***************END of PLATON [4] results***************

The peer 4 was successful joined [OK]

Do you want to make the inserts [in], make query [qu],
kill the Peers [kp], kill the PeersServer [ks] or exit [exit] ?
qu
Do you want to read from the file or from boostrap [b]  the peers ? :

Do you want to make a query? [q], to print a space [p] or to exit [ex] :p

Give the number of peer which will be quered :2

Attempting connection to planetlab1.cs.purdue.edu, port=45967
Peers [2] space is :D1L=-2.147483648E9, D1max=5.9279929E7, D2L=-9.3682487E8,
D2max=2.147483647E9, D3L=-2.147483648E9, D3max=2.147483647E9,

Do you want to make a query? [q], to print a space [p] or to exit [ex] :q

Give the number of peer which will be quered :1

Do you want to give a point to search for them? yes[y]/no[n] :y

Give the string 1 :TWOqqszwb54

Give the string 2 :TWOqqszwb55

Give the string 3 :TWOqqszwb56

What type of query, skipIndex[s] or flooding[f] ? :s

Execute Skip Index query...
Attempting connection to planetlab2.cs.purdue.edu, port=45967
I wait from the result...
Do you want to make a query? [q], to print a space [p] or to exit [ex] :

Result for point: D1=-1.605410339E9, D2=-1.605410338E9, D3=-1.605410337E9,
from peer 3
The hops are: 2, the StartPeer is 1, and the time is 0.0 sec (784 msec)
The track list is: 2, 3,
The point is found !!
The strings are: TWOqqszwb54, TWOqqszwb55, TWOqqszwb56, And the owner is:
owner2
ex

Do you want to make the inserts [in], make query [qu],
kill the Peers [kp], kill the PeersServer [ks] or exit [exit] ?
kp
Do you want to read from the file [f] or from boostrap [b]  the peers ? :f

Now I send kill signal to 4 nodes...
```

```
I will kill the peer(planetlab1.cs.unibo.it)  1
Attempting connection to planetlab1.cs.unibo.it, port=45967
The peer 1 was successful killed
I will kill the peer(planetlab1.cs.purdue.edu)  2
Attempting connection to planetlab1.cs.purdue.edu, port=45967
The peer 2 was successful killed
I will kill the peer(planetlab2.cs.purdue.edu)  3
Attempting connection to planetlab2.cs.purdue.edu, port=45967
The peer 3 was successful killed
I will kill the peer(planet5.berkeley.intel-research.net)  4
Attempting connection to planet5.berkeley.intel-research.net, port=45967
The peer 4 was successful killed
Do you want to make the inserts [in], make query [qu],
kill the Peers [kp], kill the PeersServer [ks] or exit [exit] ?
```

Αρχικά δίνουμε τον αριθμό των peers ο οποίος χρησιμοποιείτε στην εισαγωγή και στις διαγραφές. Ακολούθως με ***in*** ξεκινάμε την εισαγωγή για 4 peers. Προηγούμενος έχουμε εκκινήσει τον PeeServer στους 4 αυτούς κόμβους και τον Bootstrap σε ένα άλλο ξεχωριστό μηχάνημα του οποίο την διεύθυνση έχουμε καταχωρήσει στη μεταβλητή *Bootstrap* της κλάσης Variables. Επίσης είναι ενεργοποιημένος ο αλγόριθμος Platon Load Balancing για αυτό και σε κάθε peer με ID δύναμη του 2 βλέπουμε τα συνολικά στατιστικά στοιχεία της εκτέλεσης του αλγορίθμου. Οι διευθύνσεις των peers διαβάζονται από το αρχείο *nodes.hosts.* Μετά την εκτέλεση των εισαγωγών μπορούμε να εκτελέσουμε ερωτήματα στους peers ή να τερματίσουμε την εφαρμογή σε όλους τους κόμβους. Δίνουμε ***qu*** για μπούμε σε λειτουργία ερωτημάτων, και αν έχουμε ρυθμίσει τους peers να ακούνε σε διαφορετικές πόρτες μας δίνεται αρχικά η δυνατότητα να επιλέξουμε να πάρουμε τις διευθύνσεις και τις πόρτες που ακούνε οι peers από τον Bootstrap ο οποίος διαθέτει αυτή την λίστα. Με ***p*** μπορούμε να ρωτήσουμε κάποιο peer και να τυπώσουμε τα όρια της **D**-διαστάσεων περιοχή του. Ακολούθως δίνουμε ***q*** για να εκτελέσουμε ερώτημα αναζήτησης. Αν δεν επιλέξουμε να δώσουμε συντεταγμένες τότε παράγονται D τυχαίοι αριθμοί και εκτελείτε αυτόματα το ερώτημα, αλλιώς δίνουμε εμείς τα D χαρακτηριστικά του στοιχείου που ψάχνουμε (αν είναι ενεργοποιημένη η επιλογή για Strings τότε αυτά μετατρέπονται σε double με hash πριν να σταλούν στον αρχικό peer). Το ερώτημα εκτελείτε στο peer που επιλέξαμε προηγουμένως (StartPeer) και τα αποτελέσματα επιστρέφονται πίσω ασύγχρονα από τον αρχικό peer μόλις λάβει ο ίδιος την απάντηση. Αν έχει βρεθεί το δεδομένο τότε παίρνουμε και τον ιδιοκτήτη του ο οποίος δεν είναι αυτός που έχει το δεδομένο στο χώρο του αλλά αυτός που το εισήγαγε στο δίκτυο. Ακολούθως με ***kp*** τερματίζουμε την εφαρμογή σε όλους τους κόμβους και με ***ks*** αν θέλουμε τερματίζουμε και τους PeerServer.

4 Αλγόριθμος Ισοκατανομής Φορτίου PLATON

Σε αυτό το κεφάλαιο έχουμε υλοποιήσει τον αλγόριθμο ***PLATON*** [3] στην προηγούμενη πλατφόρμα SkipIndex P2P, έτσι ώστε το πολυδιάστατο δέντρο να μένει πάντα ισοζυγισμένο μετά από κάθε εισαγωγή νέων στοιχειών. Έτσι κάθε peer έχει ίσο αριθμό από στοιχεία με του υπόλοιπους κάθε φορά. Στην παρούσα έκδοσή του ο αλγόριθμος μπορεί να λειτουργήσει μόνο με αριθμό από peers οι οποίοι είναι δύναμη του δύο. Άρα κατά τη διαδικασία της εισαγωγής νέου peer, μόνο συγκεκριμένοι νέοι peers μπορούν να εκτελέσουν τον αλγόριθμο αυτό. Βέβαια ο αλγόριθμος μπορεί να εκτελεστεί οποιαδήποτε χρονική στιγμή θέλουμε να προσθέσουμε καινούρια σημεία στο δίκτυο, από οποιοδήποτε peer, φτάνει ο συνολικός αριθμός των peers να είναι δύναμη του 2. Η επιλογή για την λειτουργία ή όχι του αλγόριθμου αυτού γίνεται από την μεταβλητή ***loadBalance*** της κλάσης Variables. Ακολουθεί η περιγραφή του αλγορίθμου Platon η οποία βασίζεται στο [3]

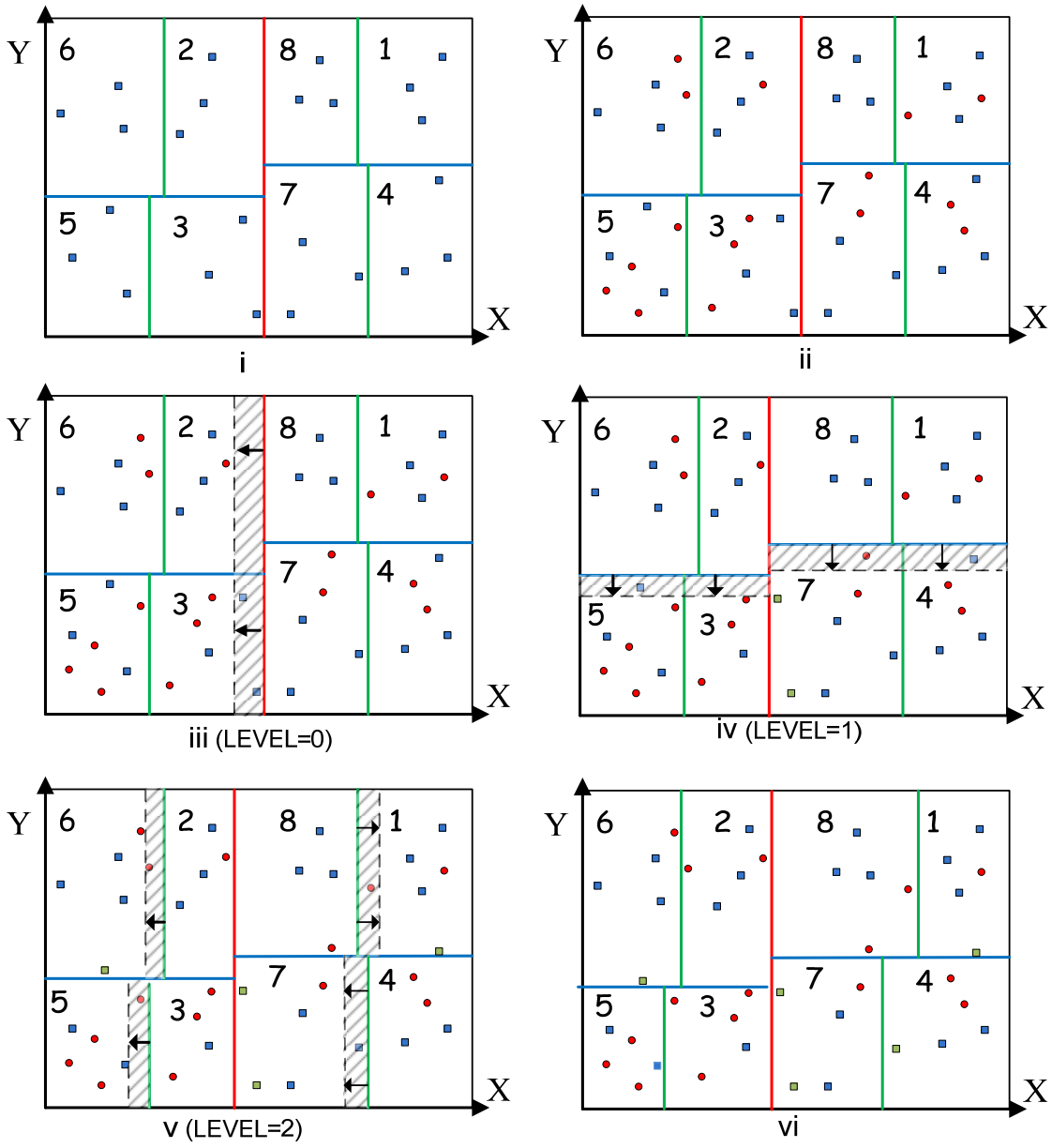
4.1 Περιγραφή του Αλγόριθμου PLATON

Κατά την εισαγωγή ενός νέου peer, μπορεί να θέλει να εισάγει νέου πόρους στο δίκτυο, οι συντεταγμένες των οποίων όμως μπορεί να βρίσκονται οπουδήποτε στο παγκόσμιο χώρο συντεταγμένων. Έτσι το συνολικό K-D δέντρο πιθανών να μην είναι πλέων ισοζυγισμένο. Σε κάθε περίπτωση ανεξάρτητα από τις τιμές και την κατανομή των νέων σημείων στο K διαστάσεων χώρο πρέπει να εγγυηθούμε ότι το συνολικό K-D δένδρο θα παραμείνει ισοζυγισμένο μετά την προσθήκη των νέων σημείων. Αυτό θα εξασφαλίσει ότι η πολυπλοκότητα για εκτέλεση K-χαρακτηριστικών ερωτήματα (ακριβής ή range) θα παραμείνει λογαριθμική σε σχέση με το συνολικό αριθμό των peers στο σύστημα.

Για να δείξουμε την ανάγκη του αλγορίθμου στην προσθήκη νέων πόρων καθώς και τη λειτουργία του, παρουσιάζουμε στο σχήμα 4.1 ένα παράδειγμα με 8 peers οι οποίοι μοιράζονται ένα χώρο 2 διαστάσεων. Στο σχήμα 4.1(i), φαίνεται η αρχική κατανομή των σημείων όπου κάθε peer έχει 3 σημεία. Αυτό μπορεί να προκύψει αν τα σημεία αυτά (24 σημεία) είναι γνωστά από την αρχή και κατά την εισαγωγή νέων peers κόβουμε κατάλληλα το χώρο κάθε φορά. Ο γεγονός αυτός όμως , να ξέρουμε από την αρχή όλα τα σημεία, εισάγει σημαντικούς περιορισμούς στη δομή του K-D δένδρου καθώς νέοι πόροι μπορεί να εμφανιστούν. Κατά την εισαγωγή των νέων πόρων στο παγκόσμιο χώρο οι κατανομές των σημείων στις περιοχές των peers θα αλλάξει με αποτέλεσμα να έχουμε περιοχές με διαφορετικό αριθμό από σημεία.

Στο σχήμα 4.1(ii) φαίνεται η κατανομή όπως θα ήταν μετά την εισαγωγή των νέων σημείων και χωρίς την εκτέλεση του αλγορίθμου Load Balance. Όπως παρατηρούμε η κατανομή δεν είναι ισοζυγισμένη αφού οι peers έχουν από 0 μέχρι 4 καινούρια σημεία. Αυτή η ανισοκατανομή των σημείων σημαίνει ότι η διαδικασία της αναζήτησης στο δίκτυο των peers θα χάσει την λογαριθμική του πολυπλοκότητα, μετρούμενη με αριθμό βημάτων μεταξύ περιοχών (δηλαδή peers). Έτσι είναι αναγκαίο να παρέχουμε ένα load balance μηχανισμό ικανό να κατανέμει ίσα τα νέα

σημεία μεταξύ των peers και έτσι οι περιοχές να έχουν ίσο αριθμό από σημεία και η πολυπλοκότητα των ερωτημάτων να παραμείνει λογαριθμική.



Σχήμα 4.1: Λειτουργία του αλγορίθμου Load Balance

Για να λύσουμε το πρόβλημα του Load Balancing όταν ένας νέος peer θέλει να εισάγει νέους πόρους στο δίκτυο, παρουσιάζουμε ένα επαναληπτικό αλγόριθμο ο οποίος ακολουθεί τη φιλοσοφία του *"διαίρει και βασίλευε"*. Ο ψευδοκώδικας παρουσιάζεται στο σχήμα 4.2 και παρουσιάζει τον αλγόριθμο PLATON LOAD_BALANCE ο οποίος εκτελείτε από τον peer *N* όταν θέλει να προσθέσει νέα σημεία στον παγκόσμιο χώρο συντεταγμένων. Τα νέα σημεία βρίσκονται μέσα στο πίνακα POINTS. Ο αριθμός των νέων σημείων είναι αυθαίρετος και ανεξάρτητος από τις τιμές και την κατανομή των ήδη υπαρχουσών σημείων του χώρου τα οποία μοιράζονται μεταξύ των peers του δικτύου. Η μεταβλητή LEVEL δηλώνει το επίπεδο στο οποίο ο επαναληπτικός αλγόριθμος εκτελείτε. Αρχικά το LEVEL είναι 0 και αυξάνεται κατά ένα όταν μια επανάληψη συμβαίνει, διασχίζοντας κάτω το δέντρο σε χαμηλότερα επίπεδα.

```
LOAD_BALANCE(N , POINTS[], LEVEL)

1.  SPLIT_VALUE=ROOT [LEVEL].SPLIT_VALUE;
2.  LOCAL_SIZE=CALCULATE_POINTS (N, POINTS, SPLIT_VALUE);
3.  REMOTE_SIZE=POINTS.SIZE-LOCAL_SIZE;
4.  IF (LEVEL>0)
    {
      MY_REMOVE_POINTS=FIND_MY_BORDER_REMOVED_POINTS(N, LEVEL);
      OPP_REMOVE_POINTS=FIND_OPP_BORDER_REMOVED_POINTS(N, LEVEL);
      LOCAL_SIZE= LOCAL_SIZE- MY_REMOVE_POINTS;
      REMOTE_SIZE= REMOTE_SIZE- OPP_REMOVE_POINTS;
    }
5.  N*=SELECT_PEER(N, LEVEL);
6.  IF (LOCAL_SIZE>REMOTE_SIZE)
    {
      MOVE_SIZE= (LOCAL_SIZE-REMOTE_SIZE)/2;
      OLD_POINTS =FIND_BORDER_POINTS  (N, LEVEL, MOVE_SIZE);
      NEW_SPLIT_VALUE=MOVE_AXIS (OLD_POINTS +POINTS, MOVE_SIZE);
      POINTS_TO_MOVE= SELECT_MOVE_POINTS(OLD_POINTS +POINTS);
      LOCAL_POINTS= SELECT_LOCAL_POINTS(POINTS);
      REMOVE_POINTS= SELECT_REMOVE_POINTS(POINTS);
      LOAD_BALANCE (N, LOCAL_POINTS, LEVEL+1);
      LOAD_BALANCE (N*, REMOTE_POINTS+POINTS_TO_MOVE, LEVEL+1);
    }
    ELSE
    {
      MOVE_SIZE= (REMOTE_SIZE -LOCAL_SIZE)/2;
      OLD_POINTS =FIND_BORDER_POINTS  (N*, LEVEL, MOVE_SIZE);
      NEW_SPLIT_VALUE=MOVE_AXIS (OLD_POINTS +POINTS,  MOVE_SIZE);
      POINTS_TO_MOVE= SELECT_MOVE_POINTS(OLD_POINTS +POINTS);
      LOCAL_POINTS= SELECT_LOCAL_POINTS(POINTS);
      REMOVE_POINTS= SELECT_REMOVE_POINTS(POINTS);
      LOAD_BALANCE (N, LOCAL_POINTS+POINTS_TO_MOVE, LEVEL+1);
      LOAD_BALANCE (N*, REMOTE_POINTS, LEVEL+1);

    }
```

Σχήμα 4.2: Ψευδοκώδικας για τον αλγόριθμο Load Balance

Η κύρια ιδέα πίσω από τον αλγόριθμο είναι η ακόλουθη. Ας πάρουμε ως παράδειγμα την κατανομή του φορτίου στο σχήμα 4.1(ii). Το παρακάτω σχήμα 4.3 παρουσιάζει το αντίστοιχο K-D δένδρο του χώρου δεδομένων που φαίνεται στο σχήμα 4.1. Υποθέτουμε ότι ο peer 8 είναι ο peer ο οποίος θα προσθέσει τα νέα σημεία στο δισδιάστατο χώρο. Αρχικά ο peer 8 συμβουλεύεται το SplitHistory του για να ελέγξει ποσά από τα νέα σημεία ανήκουν στο αριστερό υποδέντρο του K-D δένδρου στο επίπεδο 0 και πόσα ανήκουν στο δεξί. Στο σχήμα 4.2 αυτό φαίνεται στα βήματα 1-3. Έτσι βρίσκει το *LOCAL_SIZE* και *REMOTE_SIZE* και αφού το επίπεδο είναι 0 δεν μπαίνει στο if της γραμμής 4. Έτσι μπορεί να δει ότι 10 νέα σημεία πάνε στο αριστερό υποδέντρο και 6 στο δεξί. Αυτό σημαίνει ότι αν ο peer 8 καταφέρει να μετακινήσει 2 σημεία από το αριστερό υποδέντρο στο δεξί τότε το K-D δέντρο θα είναι ισοζυγισμένο στο επίπεδο 0, έχοντας 10-2=8 σημεία αριστερά και 6+2=8 σημεία δεξιά. Για να το πετύχει αυτό το πρόβλημα για τον peer 8 είναι πώς θα καθορίσει ποία σημεία πρέπει να μετακινηθούν από το αριστερό στο δεξί υποδέντρο. Στο παράδειγμά μας τα σημεία που πρέπει να μετακινηθούν για το επίπεδο 0, βρίσκονται στη σκιασμένη περιοχή του σχήματος 4.1(iii). Για να βρει αυτά τα σημεία ο peer 8 κάνει την έξης παρατήρηση: τα επιθυμητά σημεία είναι σημεία που ανήκουν σε peers οι οποίοι γειτνιάζουν με τον άξονα διαχωρισμού μεταξύ του αριστερού και δεξιού υποδέντρου, δηλαδή είναι ***οριακοί peers*** από την αριστερή πλευρά. Ο άξονας διαχωρισμού είναι ο X για ολόκληρο το δένδρο, ο Y για τα δύο υπόδεντρα που ξεκινούν στο επίπεδο 1 κοκ. Στο παράδειγμα μας οι peers οι οποίοι έχουν τα σημεία που θα μετακινηθούν είναι οι 2 και 3. Οι οριακοί peers βρίσκονται μέσα στην συνάρτηση ***FIND_BORDER_POINTS*** . Η συνάρτηση αυτή, της οποίας ο ψευδοκώδικας δίνεται στο σχήμα 4.4, βρίσκει τους οριακούς peers, και ζητά από αυτούς αριθμό σημείων ίσο με τον αριθμό αυτών που πρέπει να μετακινηθούν. Τέλος επιστρέφει όλα αυτά τα σημεία στο πίνακα *OLD_POINTS* όπως φαίνετε στο σχήμα 4.2.

Στο Platon[3] περιγράφετε ένας τρόπος ανεύρεσης των οριακών peers μέσω μια επέκτασης του SkipGraph. Εμείς σε αυτή την υλοποίηση βρίσκουμε τους οριακούς peers από την λίστα με όλους τους peers την οποία διαθέτουμε.

Η πολυπλοκότητα της *FIND_BORDER_POINTS* σε σχέση με τα μηνύματα στο δίκτυο που θα στείλει είναι ο αριθμός των οριακών peers, που θα βρεθούν σε κάθε κλήση της συνάρτησης. Σε ένα δίκτυο με N peers και K-διαστάσεων χώρο αυτός ο αριθμός είναι $N^{1-1/K}$, έτσι η πολυπλοκότητα της *FIND_BORDER_POINTS* είναι $N^{1-1/K}$.

Πηγαίνοντας πίσω στον αλγόριθμο LOAD_BALANCE μετά την εκτέλεση της συνάρτησης *FIND_BORDER_POINTS* εκτελείτε η συνάρτηση ***MOVE_AXIS*** η οποία παίρνει τα καινούρια σημεία και τα παλιά σημεία που επέστρεψε η *FIND_BORDER_POINTS* από τους οριακούς peers και αφού τα ταξινομήσει όλα μαζί σε ένα πίνακα μπορεί να βρει από εκεί το καινούριο split value. Συγκεκριμένα βρίσκει σε αυτό το πίνακα τη θέση του παλιού split value και μετακινεί τον άξονα MOVE_SIZE σημεία δεξιά η αριστερά ανάλογα. Επίσης κάνει όλες τις απαραίτητες ενέργειες ενημέρωσης που πρέπει να γίνουν για την μετακίνηση του άξονα. Στη συνέχεια η συνάρτηση ***SELECT_MOVE_POINTS*** επιλέγει τα σημεία *POINTS_TO_MOVE* τα οποία είναι αυτά που θα μετακινηθούν. Αυτά τα σημεία αποτελούνται από καινούρια και παλιά σημεία από τους οριακούς peers και είναι τα σημεία μεταξύ των δύο split value, δηλαδή του παλιού και νέου split value.

Ακολούθως πρέπει να δημιουργήσουμε ακόμα 2 λίστες από σημεία, τη *LOCAL_POINTS* στην οποία βρίσκονται τα καινούρια σημεία που θα τοποθετηθούν στην περιοχή του peer που εκτελεί τον αλγόριθμο και τα *REMOVE_POINTS* στην οποία βρίσκονται τα καινούρια σημεία που θα τοποθετηθούν στην απέναντι περιοχή. Την εργασία αυτή εκτελούν οι συναρτήσεις ***SELECT_LOCAL_POINTS*** και ***SELECT_REMOVE_POINTS*** αντίστοιχα.

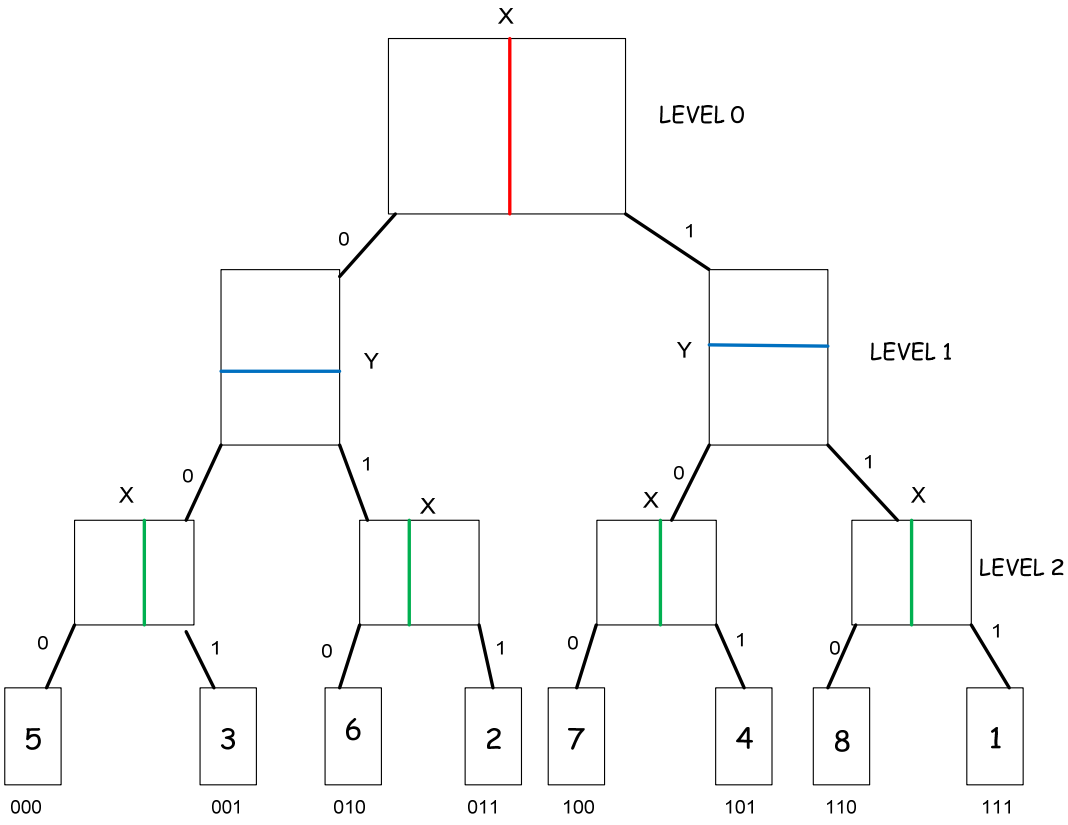
Ο N* είναι ένας οποιοσδήποτε peer ο οποίος βρίσκεται στην απέναντι περιοχή του peer που εκτελεί τον αλγόριθμο σε σχέση με τον τρέχων άξονα διαχωρισμού. Π.χ. για επίπεδο =0 αν εκτελεί ο peer 8, τότε απέναντι peers είναι οι 2, 3, 5 και 6 και για επίπεδο =1 απέναντι του peer 8 είναι οι peers 7 και 4. Ο N* βρίσκεται με την συνάρτηση ***SELECT_PEER*** .

Ακολούθως εκτελούμε αναδρομικά τον αλγόριθμο για επίπεδο αυξημένο κατά ένα σε σχέση με το τρέχων επίπεδο. Συγκεκριμένα εκτελείτε από τον τρέχων peer με όρισμα καινούρια σημεία τα *LOCAL_POINTS* και αν ο άξονας έχει μετακινηθεί προς την απέναντι περιοχή και τα *POINTS_TO_MOVE*. Επίσης στέλνεται ένα μήνυμα σε κάποιο απέναντι peer N* για να εκτελέσει και αυτός τον αλγόριθμο με καινούρια σημεία τα *REMOVE_POINTS* και τα *POINTS_TO_MOVE* αν αυτά είναι στην απέναντι περιοχή.

Για επίπεδα >0 πρέπει να εκτελέσουμε ακόμη μια ενέργεια για να λειτουργήσει ορθά ο αλγόριθμος. Π.χ. όπως βλέπουμε στο σχήμα 4.1 (iii) τα παλιά σημεία που έχουμε μετακινήσει βρίσκονται όλα στον peer 3. Άρα στο αριστερό υποδέντρο όταν θα εκτελέσουμε τον αλγόριθμο για επίπεδο=1 οι δύο πλευρές δεν θα είναι ισοζυγισμένες ως προς τα παλιά σημεία. Έτσι με την συνάρτηση ***FIND_MY_BORDER_REMOVED_POINTS*** μαθαίνουμε για όλους του οριακούς peers, για επίπεδα μικρότερα του τρέχων επίπεδου, για τα σημεία που έχουν αφαίρεση μέχρι τώρα σε όλες τις προηγούμενες εκτέλεσης του τρέχων αλγόριθμου LOAD BALANCE. Οι οποίοι όμως οριακοί peers βρίσκονται μέσα στην τρέχων τοπική περιοχή του peer και στους οποίους ο άξονας έχει μετακινηθεί μέσα στην περιοχή τους και άρα πιθανών να τους αφαίρεσε σημεία. Π.χ. για την εκτέλεση του αλγορίθμου για επίπεδο=1 στο δεξιό υποδέντρο αν εκτελείτε από τον peer 6 τότε τοπικός οριακός peer είναι ο 2 ενώ απέναντι ο 3. Σημειώστε ότι στο δεξιό υποδέντρο δεν χρειάζεται να μάθουμε για τους οριακούς peers 7 και 8 από τον peer που εκτελεί τον αλγόριθμο εκεί διότι στο επίπεδο 0 ο άξονας δεν έχει μετακινηθεί μέσα στην περιοχή τους και άρα αποκλείετε να έχουν αφαιρέσει σημεία. Αντίστοιχα με την ***FIND_OPP_BORDER_REMOVED_POINTS*** ρωτάμε τους απέναντι peers και οι τιμές που παίρνουμε αφαιρούνται από το LOCAL_SIZE και REMOTE_SIZE αντίστοιχα. Αξίζει να σημειωθεί ότι οι δύο αυτές συναρτήσεις δεν στέλνουν μηνύματα στο δίκτυο αλλά παίρνουν την πληροφορία που θέλουν από ένα ιστορικό που στέλνεται από τον peer ο οποίος έστειλε την αίτηση για εκτέλεση του load balance.

Επιστρέφοντας πίσω στο παράδειγμα μας του σχήματος 4.1 βλέπουμε στο σχήμα 4.1(iv) την κατανομή μετά την πρώτη εκτέλεση του αλγόριθμου. Τώρα το δέντρο είναι ισοζυγισμένο στο επίπεδο 0 αλλά το ισοζύγιο φορτίου στις υπόλοιπες περιοχές θα επιτευχθεί με την αναδρομική εκτέλεση του αλγόριθμου. Όπως φαίνεται στο σχήμα 4.1(iv) μέχρι 4.1(vi). Στο σχήμα 4.1(iv) φαίνεται ταυτόχρονη επαναληπτική εκτέλεση του αλγόριθμου για επίπεδο=1, στις δύο υποπεριοχές του δέντρου. Ας

πάρουμε την αριστερή περιοχή στην οποία ο άξονας Y χωρίζει τις δύο περιοχές που αποτελούνται από τους peers 2,6 πάνω και 3,5 κάτω. Ο αλγόριθμος εκτελείτε από κάποιο από αυτούς του 4 peers. Στην πάνω περιοχή προστίθενται 3 καινούρια σημεία ενώ στην κάτω 7. Παράλληλα όμως στην κάτω περιοχή έχουν αφαιρεθεί κατά την μετακίνηση του άξονα στο επίπεδο 0 δύο σημεία (από τον peer 3) άρα στην κάτω περιοχή προστίθενται 7-2=5 σημεία και 3 πάνω, άρα έχουμε 5-3=2 σημεία διαφορά και πρέπει να μετακινήσουμε ένα σημείο από την κάτω περιοχή στην πάνω για να είναι ισοζυγισμένο το αριστερό υποδέντρο στο επίπεδο 1. Άρα ο άξονας μετακινείτε προς τα κάτω. Αντίστοιχα στο δεξί υποδέντρο έχουμε 2 καινούρια σημεία πάνω και 6 κάτω, μαζί με τα παλιά που έχουν μετακινηθεί. Άρα πρέπει να μετακινήσουμε 6-2=4/2=2 σημεία από την κάτω περιοχή στην πάνω, μετακινώντας τον άξονα προς τα κάτω. Στο σχήμα 4.1(v) βλέπουμε την παράλληλη επαναληπτική εκτέλεση του αλγόριθμου για το επίπεδο 2 στις τέσσερις διαφορετικές υποπεριοχές. Π.χ. στην κάτω αριστερή περιοχή μεταξύ των peers 3 και 5 έχουμε 4 καινούρια σημεία στον peer 5 και 3 στον peer 3. Όμως σε προηγούμενες επαναλήψεις του αλγόριθμου έχουμε αφαιρέσει 1 σημείο από τον peer 5 (επίπεδο=1) και 2 σημεία από τον peer 3 (επίπεδο=0), άρα 4-1=3 νέα σημεία για τον peer 5 και 3-2=1 νέο σημείο για τον peer 3. Άρα πρέπει να μετακινήσουμε ένα σημείο από τον peer 5 στον 3. Τέλος στο σχήμα 4.1(vi) φαίνεται η τελική ισοζυγισμένη περιοχή όπου κάθε peer έχει 5 σημεία.



Σχήμα 4.3: K-D δέντρο αντίστοιχο του χώρου δεδομένων στο σχήμα 4.1

```
FIND_BORDER_POINTS  (N, LEVEL, MOVE_SIZE)

1. BORDER_NODES []= ADVISE_PLATON_PEER_LIST (N, LEVEL);
2. i=0;  foreach node n in BORDER_NODES[];
   {
      POINTS_TEMP[i++]=n.SELECT_POINTS(MOVE_SIZE);
   }
3. return POINTS_TEMP [];
```

Σχήμα 4.4: Ψευδοκώδικας για την συνάρτηση FIND_BORDER_POINTS

Ακολούθως αναλύουμε την πολυπλοκότητα που χρειάζεται ο αλγόριθμος load balance για την ανακατανομή του φορτίου στις περιοχές των peers όταν νέα σημεία με K χαρακτηριστικά στην γενική περίπτωση, εισάγονται στο δίκτυο. Στο προηγούμενο παράδειγμά μας ο αριθμός K των διαστάσεων ήταν 2, όμως ο αλγόριθμος μπορεί να λειτουργήσει με K διαστάσεις ακριβώς με τον ίδιο τρόπο όπως στις 2 διαστάσεις.

4.1.1 Ανάλυση της πολυπλοκότητας του αλγόριθμου PLATON

Ας δηλώσουμε **T(N)** την πολυπλοκότητα του αλγόριθμου PLATON load balance [3] όταν ο αριθμός των peers είναι N. Θα υπολογίσουμε την πολυπλοκότητα του αλγόριθμου σε σχέση με των αριθμό των μηνυμάτων που στέλνονται στο δίκτυο για την επικοινωνία μεταξύ των peers. Συνεπώς η πολυπλοκότητα την οποία προσπαθούμε να αναλύσουμε αφορά την συνολική επιβάρυνση του δικτύου που προκαλεί ο αλγόριθμος.

Ο αλγόριθμος load balance καλεί την συνάρτηση FIND_BORDER_POINTS σε κάθε επανάληψη. Αυτή είναι η μόνη συναρτήση που απαιτεί μηνύματα στο δίκτυο από ένα peer σε ένα σύνολο από κάποιους άλλους απομακρυσμένους peers, για να βρεθούν τα σημεία που πρέπει να αφαιρεθούν και με βάση αυτών και το νέο split value. Για να το πετύχουμε αυτό πρέπει αν ρωτήσουμε όλους τους peers που γειτνιάζουν με το τρέχων άξονα διαχωρισμού αλλά από την μια πλευρά μόνο. Ας δηλώσουμε **f(N)** την πολυπλοκότητα της FIND_BORDER_POINTS όπως φαίνεται στο σχήμα 4.4. Τώρα μπορούμε να δούμε την πολυπλοκότητα του αλγόριθμου LOAD_BALANCE όπως δίνεται από την παρακάτω εξίσωση.

$$T(n) = 2 * T(\frac{n}{2}) + f(n) \quad \textbf{(1)}$$

Αυτό ισχύει διότι σε κάθε επαναληπτική κλήση θα διασχίσουμε το παγκόσμιο K-D δέντρο σε ένα επίπεδο πιο κάτω κάθε φορά καλώντας την LOAD_BALANCE στα δύο υποδέντρα και το καθένα έχει τους μισούς peers από το προηγούμενο επίπεδο.

Θα εξετάσουμε τώρα την πολυπλοκότητα της f(N) της συνάρτησης FIND_BORDER_POINTS . Η πολυπλοκότητας σε σχέση με τα μηνύματα στο δίκτυο που στέλνονται είναι ίσο με τον αριθμό των οριακών peers που βρίσκονται σε κάθε

κλήση της μεθόδου, και σε ένα δίκτυο με n peers και K-διαστάσεων χώρο αυτή είναι $(n^{1/K})^{K-1} = n^{1-1/K}$. Έτσι η πολυπλοκότητα της FIND_BORDER_POINTS είναι $n^{1-1/K}$

Η εξίσωση 1 τότε γίνεται:

$$T(n) = 2 * T(\frac{n}{2}) + n^{1-1/K} \quad \textbf{(2)}$$

Θα λύσουμε την εξίσωση 2 εφαρμόζοντας το master theorem.

$T(n) = a * T(\frac{n}{b}) + f(n)$ *για* $a \geq 1, b > 1$ *και*
$f(n) = O(n^{\log_b a - \varepsilon})$

Τότε $T(n) = \Theta(n^{\log_b a})$

Στην περίπτωση μας *α=2 και b=2* τότε $f(n) = O(n^{\log_b a - \varepsilon}) = \mathrm{O}(n^{\log_2 2 - \varepsilon}) = \mathrm{O}(n^{1-\varepsilon})$ *για* $\varepsilon = \frac{1}{K}$. Αυτό σημαίνει ότι μπορούμε να εφαρμόσουμε το master theorem για $\varepsilon = \frac{1}{K}$ και τελικά παίρνουμε την παρακάτω πολυπλοκότητα:

Υπενθυμίζουμε ότι αυτά είναι τα συνολικά μηνύματα που στέλνονται στο δίκτυο έτσι ώστε να μπορέσουμε να βρούμε τις νέα τιμές των αξόνων. Τα μηνύματα αυτά όμως στέλνονται παράλληλα από πολλούς peers αφού ο αλγόριθμος δουλεύει παράλληλα και αναδρομικά.

$$T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$$

4.2 Υλοποίηση του αλγόριθμου PLATON στην πλατφόρμα SkipIndex σε Περιβάλλον PlanetLab

Στην εκτέλεση του αλγόριθμου κάνουμε την παραδοχή ότι το δέντρο αρχικά είναι ισοζυγισμένο και όλοι οι peers έχουν ίσο αριθμό από στοιχεία. Επίσης σε αυτή την υλοποίηση του αλγορίθμου οι οριακοί peers σε κάθε επίπεδο βρίσκονται τοπικά από τους peers χρησιμοποιώντας την λίστα (από αντικείμενα Peer) με όλους τους peers του συστήματος την οποία διαθέτουν.

Η πρώτη εκτέλεση του αλγόριθμου γίνεται με την εκτέλεση της μεθόδου ADD της κλάσης Node με ορίσματα τα καινούρια σημεία που θέλει να εισάγει ο peer και level=0. Στην συνέχεια αυτή καλεί την ADD μέθοδο της PLATON με τα ίδια ορίσματα. Εκεί αρχικά βρίσκουμε τον αριθμό των καινούριων σημείων που θα τοποθετηθούν στην μεριά του peer καθώς και στην απέναντι μεριά με βάση το επίπεδο 0. Οι δύο μεριές προκύπτουν με βάση το split dimension, καθώς και το επίπεδο που έγινε ο διαχωρισμός αυτός. Π.χ. στο σχήμα 4.5 με κόκκινο χρώμα βλέπουμε το διαχωρισμό που έγινε στη Χ διάσταση στο επίπεδο 0, και τις δύο μεριές αριστερά με τους peers 2,3,5 και 6 και δεξιά με τους peers 1,4,7 και 8. Με μπλε το διαχωρισμό στη Υ διάσταση στο επίπεδο 1 και τέλος με πράσινο χρώμα το διαχωρισμό στη Χ διάσταση στο επίπεδο 2. Αντίστοιχα στο σχήμα 4.6 βλέπουμε το δέντρο για τους 8 peers. Στο σχήμα 4.7 εικονίζονται ένα ένα τα βήματα για το διαχωρισμό του δισδιάστατου χώρου

Την διαφορά αυτών των δύο αριθμών την ονομάζουμε *move_size* και αν είναι μεγαλύτερη από τον αριθμό των σημείων, που προκύπτει από τα συνολικά καινούρια σημεία πολλαπλασιαζόμενα με ένα της εκατό ποσοστό που δίνουμε ως μεταβλητή ($thresholdPlaton = \frac{abs(LOCAL\_POINTS.size - REMOTE\_POINTS.size)}{POINTS.size} * 100$) στη κλάση Variables, τότε πρέπει να μετακινήσουμε τον άξονα κατάλληλα για να διατηρήσουμε ισοζυγισμένο το δέντρο στο επίπεδο μηδέν. Δηλαδή *move_size/2* σημεία πρέπει να μετακινηθούν προς την μεριά με τα λιγότερα σημεία. Ανάλογα με το που είναι τα περισσότερα σημεία θα μετακινηθεί και ο άξονας, π.χ. αν είναι αριστερά τότε θα μετακινηθεί αριστερά ο άξονας και αντίστοιχα δεξιά. Προς την μεριά που θα μετακινηθεί ο άξονας στέλνουμε ένα μήνυμα στους οριακούς peers και τους ζητάμε να μας στείλουν κάποια σημεία τους, τα ποιο κοντινά στον άξονα, και πλήθος ίσο με τα συνολικά σημεία που θέλουμε να μετακινήσουμε. Π.χ. αν ο άξονας στο επίπεδο 0 (κόκκινη γραμμή) θα μετακινηθεί αριστερά τότε θα στείλουμε το μήνυμα αυτό στους peers 2 και 3.

Τους οριακούς(border) peers σε κάθε επίπεδο διαχωρισμού βρίσκουμε με τη συνάρτηση *findBorderPeers* της κλάσης Variables. Για παράδειγμα ο peer 6, με βάση το σχήμα 4.5, στο επίπεδο 0 έχει ως οριακούς peers στη μεριά του τους 2 και 3 ενώ από την άλλη μεριά τους 8 και 7. Στο επίπεδο 1 έχει ως οριακούς peers στη μεριά του τους 2 και 6 ενώ από την άλλη μεριά τους 5 και 3. Και τέλος στο επίπεδο 2 έχει ως οριακό peer στη μεριά του τον εαυτό του ενώ από την άλλη μεριά των 2. Η *findBorderPeers* βρίσκει αυτές τις πληροφορίες από την λίστα με όλους τους peers και με βάση τα κλειδιά διαχωρισμού τους.

Με βάση τα σημεία που θα πάρουμε από τους οριακούς peers, τα καινούρια σημεία που θέλουμε να εισάγουμε και το παλιό split value για τον συγκεκριμένο άξονα

μπορούμε να βρούμε το νέο split value έτσι ώστε το δέντρο στο επίπεδο μηδέν να είναι ισοζυγισμένο. Συγκεκριμένα διατάσουμε όλα αυτά τα σημεία κατά την διάσταση διαχωρισμού και αφού βρούμε που βρίσκετε το παλιό split value μετακινούμαστε δεξιά ή αριστερά ανάλογα κατά *move_size* /2 σημεία και βρίσκουμε το νέο split value. Όταν βρεθεί το νέο split value ο peer που εκτελεί τον αλγόριθμο ενημερώνει το SplitHistory του σε αυτό το επίπεδο με την νέα τιμή καθώς και όλα τα SplitHistory που διατηρεί για τους γείτονες του στο SkipList. Στην συνέχεια στέλνει το αλλαγμένο αντικείμενο SplitHistory με ένα *updateSplitHistory request* σε όλους τους υπόλοιπους peers για να κάνουν και αυτοί όλες τις απαραίτητες ενημερώσεις αν χρειάζεται. Ακολούθως στέλνει στους οριακούς peers που έχουν απομακρυνθεί σημεία από την περιοχή τους, ένα *changeSpace_RemovePoints request* μαζί με τα παλιά σημεία που θα πρέπει να μετακινηθούν. Οι οριακοί αυτοί peers απομακρύνουν από την περιοχή τους τα σημεία τους που έχουν μετακινηθεί και ενημερώνουν το χώρο τους για τις νέες διαστάσεις. Στους οριακούς peers από την άλλη μεριά στέλνουμε ένα *changeSpace request* για να αλλάξουν απλά τα όρια της περιοχής τους.

Στη συνέχεια σχηματίζουμε 3 λίστες από σημεία. Την *logalPoints* η οποία αποτελείτε από καινούρια σημεία τα οποία θα τοποθετηθούν στην μεριά του peer ο οποίος εκτελεί τον αλγόριθμο. Την *remotePoints* η οποία αποτελείτε από καινούρια σημεία τα οποία θα τοποθετηθούν στην απέναντι μεριά. Και την *POINTS_X2_ALL* η οποία αποτελείτε από τα παλιά σημεία τα οποία θα μετακινηθούν και τα καινούρια σημεία τα οποία βρίσκονται μεταξύ newSplitValue και oldSplitValue αν newSplitValue < oldSplitValue ή μεταξύ oldSplitValue και newSplitValue αν oldSplitValue < newSplitValue.

Ο έλεγχος στην συνέχεια περνά στην μέθοδο ADD της κλάσης Node. Αν τα *POINTS_X2_ALL* σημεία είναι στην μεριά του peer που εκτελεί τον αλγόριθμο τότε θα εκτελεστεί αναδρομικά ο προηγούμενος αλγόριθμος με σημεία τα *logalPoints+ POINTS_X2_ALL* και level=1 και θα βρεθεί ένας peer από την απέναντι μεριά (επιλέγουμε ένα οριακό peer από την άλλη μεριά) του οποίου θα αποσταλεί ένα *loadBalance request* με σημεία τα *remotePoints* και level=1. Αλλιώς αν τα *POINTS_X2_ALL* είναι στην απέναντι μεριά, θα προστεθούν στα *remotePoints* και θα αποσταλούν όπως προηγουμένως στον απομακρυσμένο peer.

Ο αλγόριθμος αυτός εκτελείτε αναδρομικά μέχρι το level γίνει ίσο με το επίπεδο του δέντρου-1. Σε αυτή την επανάληψη ο peer που εκτελεί τον αλγόριθμο θα προσθέσει όλα τα σημεία που πρέπει να μπουν στην περιοχή του και θα στείλει ένα *loadBalance request* στον peer από την απέναντι μεριά με την *stopFlag* όμως ίσον με 1. Δηλαδή λαμβάνοντας αυτό το request ο απομακρυσμένος peer απλά θα προσθέσει τα σημεία στην περιοχή του χωρίς να ξαναεκτελέσει τον αλγόριθμο.

Κάθε peer που εκτελεί τον αλγόριθμο στο τέλος στέλνει στο control center ένα *PlatonResult request* μαζί με τον αριθμό όλων των μηνυμάτων που έστειλε σε οριακούς peers για τις ανάγκες του αλγορίθμου (εξαιρούνται οι ενημερώσεις), το συνολικό αριθμό των σημείων που χρειάστηκε να μετακινήσει καθώς και το τελικό φορτίο του. Επίσης ο νέος peer ο οποίος ξεκίνησε τον αλγόριθμο Platon στέλνει τον χρόνο που χρειάστηκε για να εκτελέσει τον αλγόριθμο και αφού ο αλγόριθμος εκτελείτε παράλληλα και από άλλους peers αυτός ο χρόνος είναι ενδεικτικός για το συνολικό χρόνο που χρειάζεται για να εκτελεστεί ο αλγόριθμος. Επίσης του στέλνει

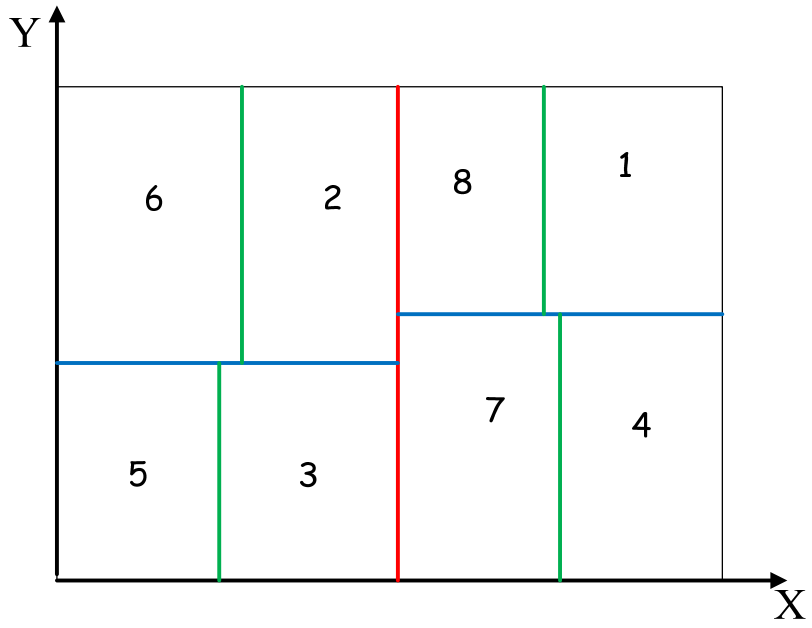
τον αριθμό των καινούριων σημείων που προστέθηκαν και το threshold. Όλες αυτές οι πληροφορίες μαζεύονται και αθροίζονται, όπου χρειάζεται, από το control center το οποίο μόλις λάβει από όλους τους peers τα μηνύματα παρουσιάζει τα συνολικά στατιστικά εκτέλεσης του αλγορίθμου.

Κατά τις επόμενες επαναλήψεις με επίπεδο μεγαλύτερο του 0 υπάρχει και μια άλλη παράμετρος που πρέπει να λάβουμε υπόψη μας. Έστω η περίπτωση μετακίνησης τους άξονα Χ στο επίπεδο 0 αριστερά. Τότε θα πρέπει να απομακρυνθούν κάποια σημεία από τους οριακούς peers 2 και 3 όπως έχει περιγραφεί προηγούμενος. Υπάρχει όμως το ενδεχόμενο αυτά τα σημεία να μην τα λάβουμε με ίσο αριθμό από τους οριακούς peers. Έτσι μπορεί π.χ. από τον peer 3 να λάβουμε πολύ περισσότερα σημεία από ότι από τον peer 2, (όπως φαίνεται στο σχήμα 4.1(iii)) και τελικά οι περιοχές στα δυο υποδέντρα να μην είναι πλέων ισοζυγισμένες. Βέβαια στο επίπεδο μηδέν το δέντρο θα είναι ισοζυγισμένο. Έτσι στις επόμενες επαναλήψεις του αλγόριθμου κατά τον υπολογισμό των καινούριων σημείων που θα τοποθετηθούν στη μεριά του peer καθώς και στην απέναντι πρέπει να λάβουμε υπόψη μας και αυτή την παράμετρο. Π.χ. στο επίπεδο 1 αυτό επιτυγχάνεται βρίσκοντας ο peer τους οριακούς peers του προηγούμενου επιπέδου που βρίσκονται στη μεριά του. Π.χ. έστω ο peer 6 εκτελεί τον αλγόριθμο για level=1, δηλαδή για τον άξονα Υ (μπλε άξονας). Οι οριακοί peers του επιπέδου μηδέν οι οποίοι βρίσκονται στην μεριά του είναι οι 2 και 3. Αυτοί οι peers λόγο της προηγούμενης μετακίνησης του άξονα αριστερά μπορεί να έχουν διαφορετικό φορτίο χαλώντας έτσι το ισοζύγιο φορτίου για το τρέχων επίπεδο (επίπεδο 1 στο παράδειγμα μας). Στη συνέχεια ο peer βρίσκει ποιοι από αυτούς τους οριακούς peers είναι στην μεριά του και ποίοι απέναντι με βάση το τρέχων επίπεδο. Στο παράδειγμά μας ο peer 2 είναι στην ίδια μεριά ενώ ο 3 στην απέναντι. Ακολούθως μαθαίνει για τους οριακούς peers που βρήκε προηγούμενος και στους οποίους ο άξονας έχει μετακινηθεί μέσα στην περιοχή τους και πιθανών να αφαίρεσαν σημεία τον αριθμό των σημείων που έχει μέχρι τώρα απομακρύνει στη τρέχουσα εκτέλεση του αλγόριθμου Platon. Αυτό τον αριθμό αν είναι οριακός peer ο οποίος είναι στην μεριά του με βάση το τρέχων επίπεδο τον αφαιρεί από τον αριθμό των σημείων που θα τοποθετηθούν στην μεριά του ή αντίστοιχα αν είναι peer της απέναντι μεριάς τον αφαιρεί από τον αριθμό των σημείων που θα τοποθετηθούν απέναντι. Στην συνέχεια ο αλγόριθμος λειτουργεί όπως προηγουμένως.

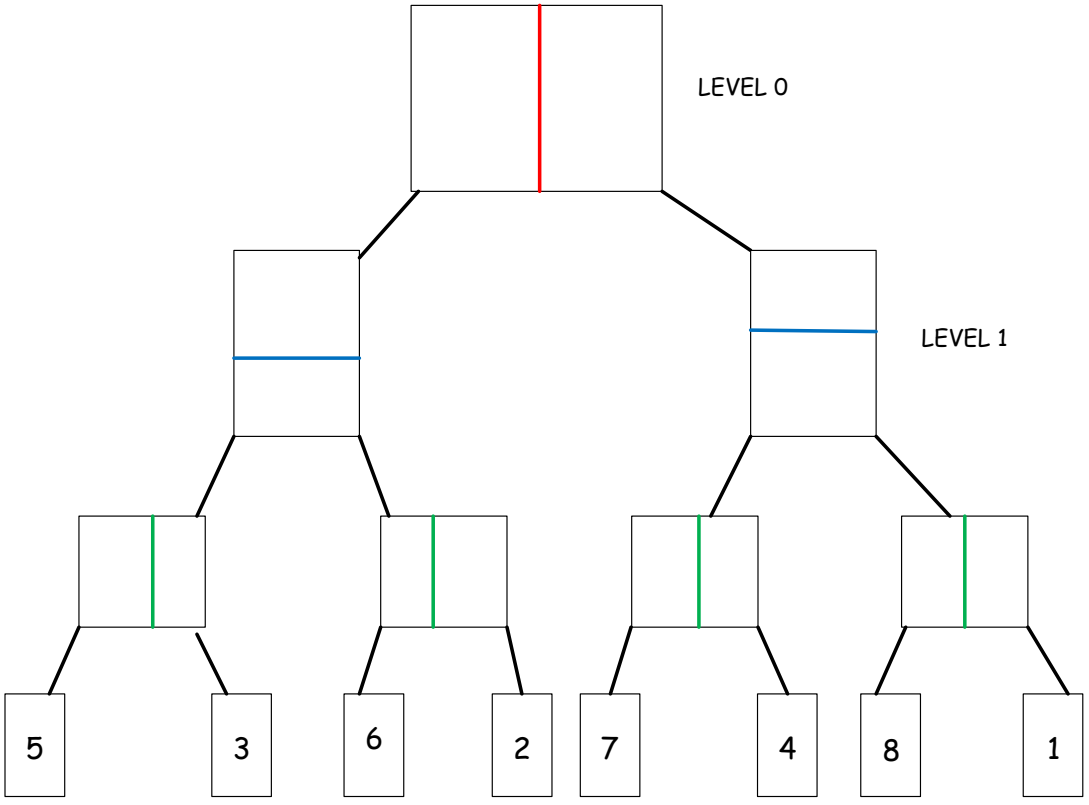
Για την πληροφορία αυτή δεν στέλνουμε κανένα μήνυμα στο δίκτυο. Κάθε peer που λαμβάνει αίτημα για εκτέλεση του αλγόριθμου load balance λαμβάνει από τον προηγούμενο peer ένα ιστορικό με την κατεύθυνση που έχουν μετακινηθεί οι άξονες καθώς και με τον αριθμό των σημείων που έχει αφαιρέσει κάθε οριακός peer . Αυτές τις πληροφορίες τις ενημερώνει ο peer που λαμβάνει το αίτημα και τις προωθεί μαζί με το αίτημα/τα load balance που θα στείλει στην συνέχεια. Για να λειτουργήσει σωστά αυτό, βασιζόμαστε στο γεγονός ότι ένας peer για να εκτελέσει τον αλγόριθμο δεν χρειάζεται να ξέρει όλες τις μετακινήσεις των αξόνων και τα σημεία που έχουν αφαιρέσει όλοι οι peer του δικτύου αλλά μια ομάδα από αυτούς. Π.χ. στο σχήμα 4.1(v) κατά την μετακίνηση των αξόνων στο επίπεδο 2 στις περιοχές 6-2 και 5-3 δεν χρειάζεται να ξέρουμε πως έχει μετακινηθεί ο άξονας και τους οριακούς peers του επιπέδου 1 δεξιά για τον άξονα μεταξύ των 8,1 και 7,4.

Σε επίπεδα μεγαλύτερα του 1 λειτουργούμε με τον ίδιο τρόπο μαθαίνοντας για τα σημεία που απομάκρυναν μόνο για τους peer που βρίσκονται κοντά σε άξονες, οι οποίοι έχουν μετακινηθεί σε προηγούμενες εκτελέσεις του Platon σε ψηλότερα

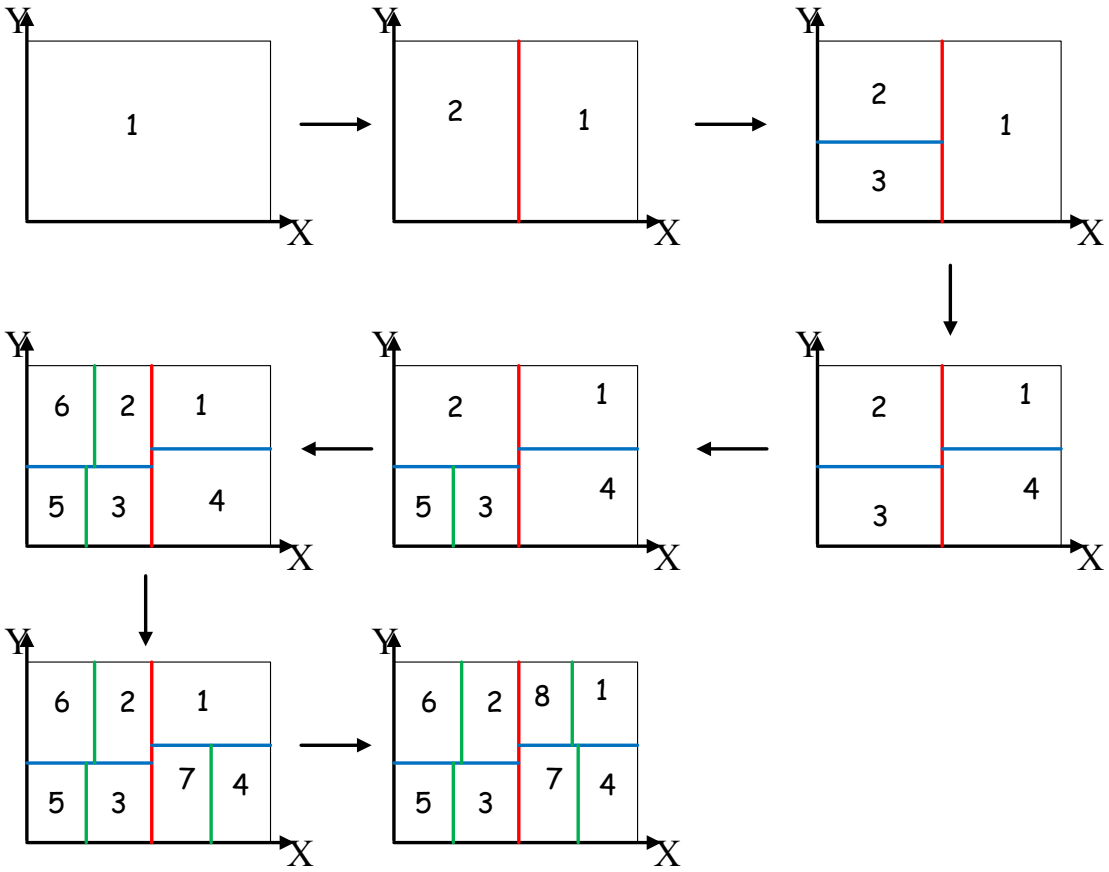
επίπεδα μέσα στην περιοχή του border peer και άρα πολύ πιθανών να αφαίρεσαν σημεία.



Σχήμα 4.5: Δισδιάστατος χώρος με 8 peers



Σχήμα 4.6: Δισδιάστατο δέντρο για 8 peers



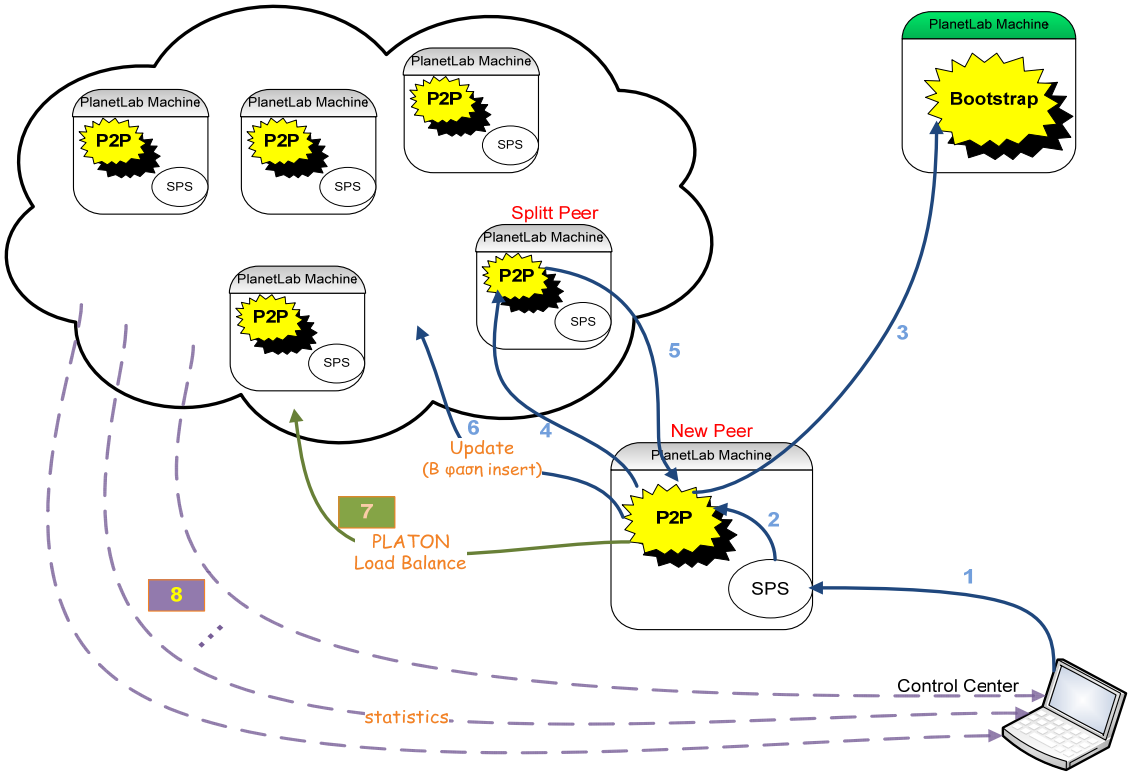
Σχήμα 4.7: Διαδικασία διαχωρισμού δισδιάστατου χώρου για 8 peers

4.2.1 Διαδικασία Εισαγωγής Νέου Peer με τον Αλγόριθμο PLATON Ενεργοποιημένο

Η διαδικασία εισαγωγής νέου peer με ενεργό τον αλγόριθμος PLATON είναι η ίδια όπως περιγράφηκε στο κεφάλαιο 3 με κάποιες μικροδιαφορές. Όπως φαίνεται και στο σχήμα 4.8, μέχρι το βήμα 5 γίνεται όπως και προηγουμένως και ο νέος peer εισάγετε επιτυχημένα στο δίκτυο χωρίς όμως την πρόσθεση των καινούριων σημείων του. Στο βήμα 6 ξεκινά την ενημέρωση του δικτύου για την νέα εισαγωγή (Β φάση εισαγωγής) και αμέσως μετά στο βήμα 7 αρχίζει την εκτέλεση του αλγορίθμου PLATON για την εισαγωγή των καινούριων σημείων διατηρώντας όμως το δέντρο ισοζυγισμένο. Αυτά τα δύο βήματα στη συνέχεια μπορεί να εκτελούνται παράλληλα! Στο βήμα 8 το control center, όπως περιγράφηκε και προηγούμενος, λαμβάνει από κάθε peer μετά το τέλος της εκτέλεσης του αλγορίθμου κάποια στατιστικά στοιχεία, μόλις λάβει από όλους τους peers τα στοιχεία αυτά τα παρουσιάζει και ξεκινά την επόμενη εισαγωγή αν υπάρχει. Βέβαια υπάρχει το ενδεχόμενο κάποια updates ή request να έρθουν με λανθασμένη σειρά, αυτές οι περιπτώσεις είναι οι παρακάτω.

Υπάρχει το ενδεχόμενο ένας peer να λάβει request για εκτέλεση του PLATON χωρίς προηγούμενος να λάβει την ενημέρωση για την εισαγωγή του νέου peer που ξεκίνησε τον αλγόριθμο PLATON! Με σηματοφόρους και condition εξασφαλίζουμε ότι αν

αυτά τα δύο request φτάσουν με λανθασμένη σειρά τα νήματα που τα εξυπηρετούν θα εκτελεστούν με την ορθή σειρά. Επίσης εξασφαλίζουμε ότι κάθε φορά ένα μόνο νήμα μπορεί να ενημερώνει τα split histories, και τα request για ενημέρωση του split histories που έρχονται πριν να γίνει η ενημέρωση για την εισαγωγή του νέου peer περιμένουν. Επίσης σε ένα peer δεν μπορούν να εκτελούνται ταυτόχρονα ο αλγόριθμος PLATON και insert (μέθοδος insert της κλάσης Node). Αν ένα request για insert ληφθεί χωρίς προηγουμένως ο αλγόριθμος Platon να εκτελεστεί το νήμα που εκτελεί το insert θα περιμένει. Π.χ. ο peer 3 λάβει ένα join request από τον νέο peer 5 αλλά στον peer 3 δεν έχει εκτελεστεί ακόμα ο αλγόριθμος Platon από την εισαγωγή του peer 4, τότε ο 3 θα περιμένει μέχρι να γίνει αυτό και μετά να συνεχίσει με την εισαγωγή του 5.



Σχήμα 4.8: Διαδικασία εισαγωγής νέου peer με ενεργό τον αλγόριθμο PLATON

5 Μετρήσεις

Παρακάτω παρουσιάζονται κάποιες μετρήσεις που έχουν ληφθεί στο ***PlanetLab*** για τους αλγόριθμους *SkipIndex* και *Platon*. Όλοι οι χρόνοι που παρουσιάζονται είναι για το συγκεκριμένο στιγμιότυπο που έγινε η μέτρηση, και αν πάρουμε τις ίδιες μετρήσεις σε διαφορετική χρονική περίοδο αυτές οι τιμές θα διαφέρουν ελαφρός ή και σημαντικά! Αυτό συμβαίνει διότι το *PlanetLab* είναι ένα πραγματικό περιβάλλον και ανάλογα με τη χρονική στιγμή που λαμβάνουμε τη μέτρηση για ένα ερώτημα η διάρκεια του μπορεί να διαφέρει. Π.χ. σε κάποιες χρονικές περιόδους κάποιοι κόμβοι μπορεί να είναι πιο φορτωμένοι και αργοί. Επίσης αφού για την επικοινωνία χρησιμοποιούνται οι συνδέσεις του παγκόσμιου internet δύο διαδοχικά ίδια ερωτήματα πολύ πιθανόν να διαφέρουν στη διάρκεια τους ελαφρά. Παρόλο όμως που αυτοί οι χρόνοι που μετράμε δεν είναι σταθεροί μας δίνουν μια καλή εκτίμηση για το χρόνο που χρειάζονται τα διάφορα ερωτήματα. Βέβαια τα υπόλοιπα στοιχεία όπως βήματα, αριθμός μηνύματα που στέλνονται κ.α. δεν αλλάζουν ανάλογα με τη χρονική στιγμή που εκτελούμε τη μέτρηση.

5.1 SkipIndex Μετρήσεις

Εδώ παρουσιάζονται μετρήσεις για την αναζήτηση SkipIndex. Ακολουθούν 3 πίνακες με μετρήσεις για 8, 16 και 32 peers οι οποίες έγιναν σε μηχανήματα του PlanetLab. Τα ερωτήματα έγιναν σε τρισδιάστατο χώρο συντεταγμένων, με τη λειτουργία των string ενεργοποιημένη (*useStrings=true*). Μετά από κάθε πίνακα υπάρχει μια λίστα που δείχνει τη διεύθυνση του φυσικό μηχανήματος στο οποίο βρισκόταν κάθε peer. Η λίστα αυτή είναι διατεταγμένη όπως είναι η λίστα με όλους τους peers στο επίπεδο 0 του SkipGraph.

Αρχικός peer είναι ο peer στον οποίο τέθηκε το ερώτημα, *τελικός peer* είναι ο peer στον χώρο του οποίου βρέθηκε το σημείο ή ανήκει το σημείο σε περίπτωση που δεν υπάρχει. *Αριθμός βημάτων* είναι τα μηνύματα που στέλνουν μεταξύ τους οι peer μέχρι το ερώτημα αναζήτησης φτάσει στον τελικό peer. Ο *χρόνος* μετράτε από τον αρχικό peer και είναι από την στιγμή που θα λάβει αίτημα για αναζήτηση μέχρι τα αποτελέσματα να του επιστραφούν πίσω από τον τελικό peer. *Διαδρομή* είναι η ακολουθία από peers που ακολουθεί το ερώτημα μέχρι να φτάσει στον τελικό peer.

Μετρήσεις με 8 Peers στο PlanetLab

Αρχικός Peer	Τελικός Peer	Αριθμός βημάτων	Χρόνος (msec)	Διαδρομή
1	8	1	217	8
3	2	1	16	2
2	5	2	366	3, 5
4	1	2	955	8, 1
5	8	4	1678	6, 7, 4, 8
7	1	1	198	1
8	2	2	942	4, 2

Πίνακας 5.1α

Λίστα με τους Peers

Peer	Διεύθυνση
5	planet6.berkeley.intel-research.net
3	planetlab2.cs.purdue.edu
6	planetlab1.inf.ethz.ch
2	planetlab1.cs.purdue.edu
7	planetlab1.lkn.ei.tum.de
4	planet5.berkeley.intel-research.net
8	planetlab-1.ic.ac.uk
1	planetlab1.cs.unibo.it

Πίνακας 5.1β

Μετρήσεις με 16 Peers στο PlanetLab

Αρχικός Peer	Τελικός Peer	Αριθμός βημάτων	Χρόνος (msec)	Διαδρομή
3	15	4	1568	13, 14, 4, 15
5	5	0	0	
6	11	1	448	11
8	14	1	220	14
9	6	2	1168	3, 6
9	7	3	311	12, 13, 7
9	1	4	1806	12, 8, 16, 1
12	4	3	895	13, 14, 4
16	11	4	1250	13, 12, 6, 11
16	9	3	907	13, 3, 9

Πίνακας 5.2α

Λίστα με τους Peers

Peer	Διεύθυνση
9	planetlab-1.ic.ac.uk
5	planet5.berkeley.intel-research.net
10	planetlab-2.ic.ac.uk
3	planetlab1.cs.purdue.edu
11	planetlab2.csail.mit.edu
6	planet6.berkeley.intel-research.net
12	194.36.10.154
2	ricepl-2.cs.rice.edu
13	planetlab1.tmit.bme.hu
7	planetlab1.inf.ethz.ch
14	planetlab3.inf.ethz.ch
4	planetlab2.cs.purdue.edu
15	planetlab-3.ic.ac.uk
8	planetlab1.lkn.ei.tum.de
16	planetlab-4.ic.ac.uk
1	planetlab1.cs.unibo.it

Πίνακας 5.2β

Μετρήσεις με 32 Peers στο PlanetLab

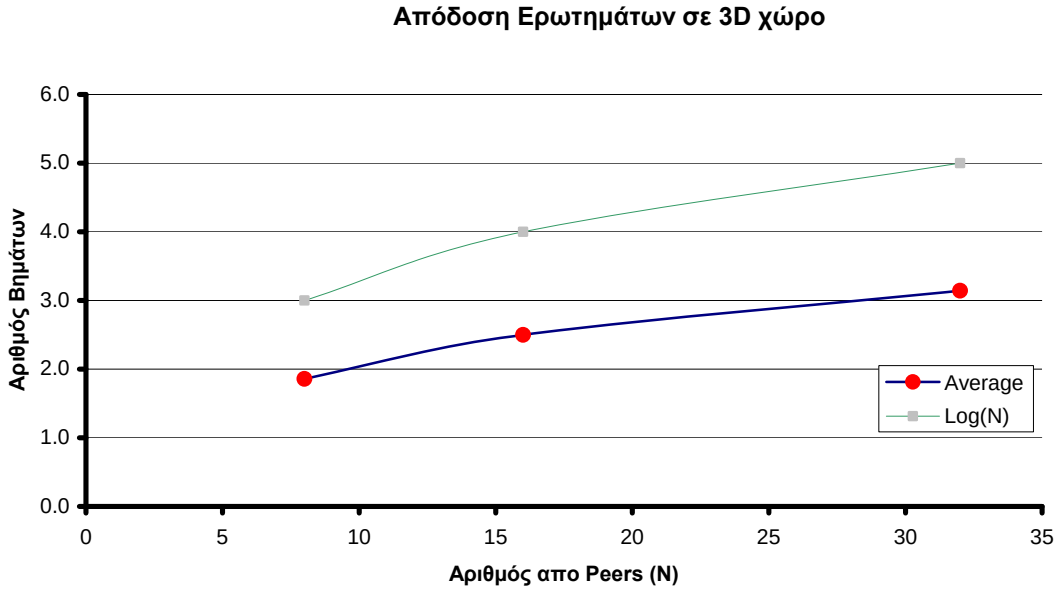
Αρχικός Peer	Τελικός Peer	Αριθμός βημάτων	Χρόνος (msec)	Διαδρομή
16	28	3	3060	31, 8, 28
4	14	1	779	14
7	6	2	1102	12, 6
11	4	5	2269	22, 2, 27, 14, 4
13	18	4	1255	24, 23, 10, 18
16	24	3	1718	31, 13, 24
17	16	1	9	16
20	22	3	851	3, 11, 22
18	22	4	839	20, 3, 11, 22
24	20	2	1148	23, 20
27	25	3	1185	7, 26, 25
28	24	2	1882	13, 24
31	20	4	2315	13, 24, 23, 20

Πίνακας 5.3α

Λίστα με τους Peers

Peer	Διεύθυνση
17	planetlab-4.ic.ac.uk
9	planetlab-2.ssvl.kth.se
18	198.7.242.41
5	planet5.berkeley.intel-research.net
19	planetlab1.comp.nus.edu.sg
10	planetlab-1.imperial.ac.uk
20	198.7.242.42
3	planetlab1.cs.purdue.edu
21	vicky.planetlab.ntua.gr
11	planetlab-2.imperial.ac.uk
22	planet2.att.nodes.planet-lab.org
6	planet6.berkeley.intel-research.net
23	193.167.187.187
12	planetlab2.csail.mit.edu
24	planetlab2.cs.unibo.it
2	ricepl-2.cs.rice.edu
25	planetlab1.csail.mit.edu
13	planetlab1.nrl.dcs.qmul.ac.uk
26	planetlab4.inf.ethz.ch
7	planetlab1.inf.ethz.ch
27	planet2.scs.stanford.edu
14	planetlab1.tmit.bme.hu
28	planetlab3.csail.mit.edu
4	planetlab2.cs.purdue.edu
29	planetlab3.cs.uiuc.edu
15	planetlab3.inf.ethz.ch
30	planet0.jaist.ac.jp
8	planetlab1.lkn.ei.tum.de
31	planet1.jaist.ac.jp
16	planetlab-3.imperial.ac.uk
32	planetlab2.comp.nus.edu.sg
1	planetlab1.cs.unibo.it

Πίνακας 5.3β



Γραφική 5.1

Παρατηρούμε ότι ο μέσος αριθμός των βημάτων που χρειάζεται για κάθε ερώτημα είναι μικρότερος από το Log(N), όπου N ο συνολικός αριθμός των peers στο σύστημα. Δηλαδή η πολυπλοκότητα του ερωτήματος αναζήτησης είναι O(log(N)) βήματα όπως προκύπτει και στο SkipIndex[2]. Όπου αριθμό βημάτων εννοούμε τον αριθμό των peer τους οποίους χρειάζεται να ρωτήσουμε μέχρι να βρούμε το δεδομένο που ψάχνουμε.

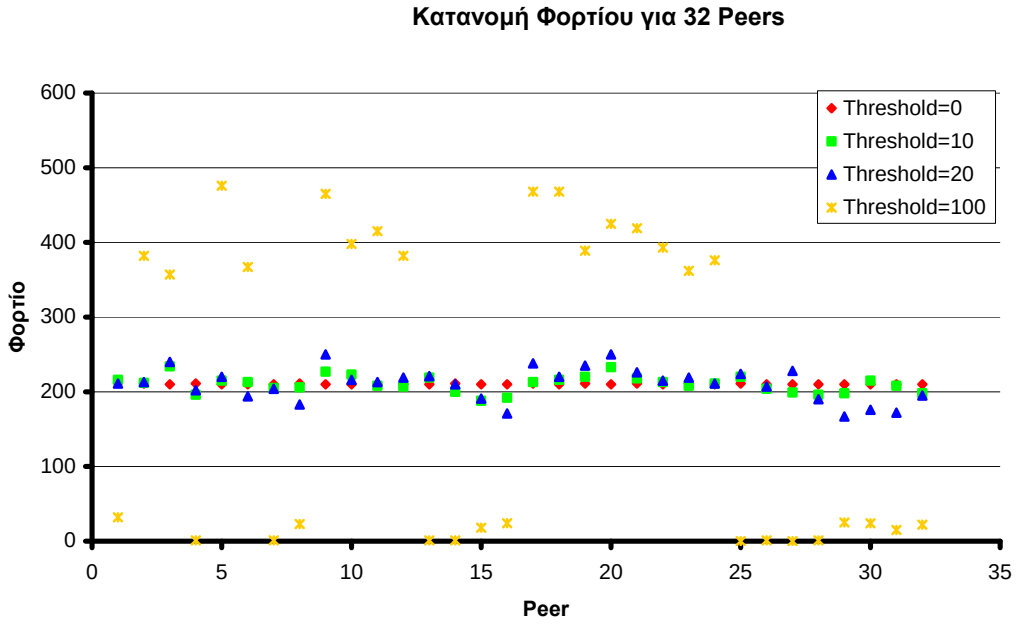
5.2 PLATON Μετρήσεις

Μετρήσεις στο PlanetLab με 32 Peers και 3D Χώρο
(Αρχικός Αριθμός Σημείων=11)

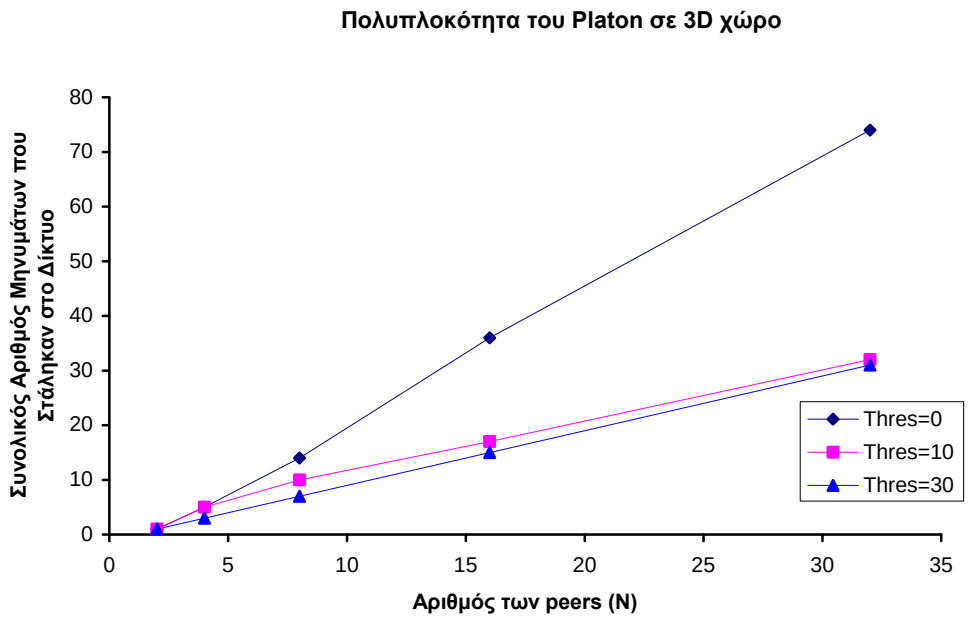
Αριθμός peers	Αριθμός Νέων Σημείων	Κατώφλι (%)	Μηνύματα στο Δίκτυο	Σημεία που Μετακινήθηκαν	Χρόνος (sec)	Load Imbalance (max/min)
2	120	0	1	55	1.05	1.01
2	120	10	1	55	1.05	1.01
2	120	20	1	55	1.04	1.01
2	120	30	1	55	1.06	1.01
2	120	100	1	0	0.34	10.89
4	800	0	5	138	2.52	1.00
4	800	10	5	123	1.78	1.14
4	800	20	3	0	0.7	1.60
4	800	30	3	0	0.6	1.60
4	800	100	3	0	0.408	77.16
8	1600	0	14	326	25.36	1.00
8	1600	10	10	235	23.15	1.08
8	1600	20	12	682	12.4	1.26
8	1600	30	7	0	4.7	2.10
8	1600	100	7	0	4.59	229.66
16	1600	0	36	127	17.92	1.00
16	1600	10	17	23	3.8	1.19
16	1600	20	15	0	1.3	1.38
16	1600	30	15	0	1.4	2.44
16	1600	100	15	0	1.43	558
32	2600	0	74	652	392.28	1.01
32	2600	10	32	111	6.43	1.24
32	2600	20	32	62	28	1.50
32	2600	30	31	0	11.6	2.76
32	2600	100	31	0	4.01	468

Πίνακας 5.4

Σε κάποιες μετρήσεις βλέπουμε μεγάλους χρόνους. Αυτό συμβαίνει διότι στις περιπτώσεις αυτές είχαμε πολλές μετακινήσεις των αξόνων και άρα πολλές ενημερώσεις για την αλλαγή του SplitHistory. Αυτές οι ενημερώσεις κοστίζουν πολύ αλλά υπάρχει δυνατότητα να βελτιωθούν. Επίσης κάποιοι κόμβοι του PlanetLab είναι αρκετά αργοί και έτσι σε αυτές τις ενημερώσεις αυξάνουν τρομακτικά το χρόνο.



Γραφική 5.2



Γραφική 5.3

Παρατηρούμε ότι ο συνολικός αριθμός των μηνυμάτων που χρειάζεται ο αλγόριθμος για να εκτελεστεί έχει γραμμική σχέση με τον συνολικό αριθμό των peers στο σύστημα(N), όπως προκύπτει και από την απόδειξη στην ενότητα 4.1.1

6 Σχόλια-Μελλοντική Δουλειά

Σε αυτή τη διπλωματική εργασία έχουμε πετύχει να υλοποιήσουμε ένα κατανεμημένο peer-to-peer σύστημα δεικτοδότησης το οποίο μπορεί να λειτουργήσει σε περιβάλλον PlanetLab και μπορούμε να το ελέγχουμε κεντρικά από ένα τοπικό PC μέσω του Control Center. Συγκεκριμένα έχουμε υλοποιήσει την κατανεμημένη εισαγωγή των peers στο σύστημα, καθώς και την εκτέλεση ερωτημάτων αναζήτησης σε D-διαστάσεων χώρο σύμφωνα με το πρότυπο SkipIndex[2]. Σύμφωνα με μετρήσεις που έχουμε λάβει σε περιβάλλον PlanetLab τα ερωτήματα αναζήτησης κοστίζουν O(log(N)) βήματα όπου N ο συνολικός αριθμός των peers.

Επίσης στην πλατφόρμα αυτή έχουμε ενσωματώσει τον καινούριο αλγόριθμο Platon[3] για να κρατάμε το φορτίο στο D-διαστάσεων χώρο ισοζυγισμένο μεταξύ των peers. Με μετρήσεις που έχουμε λάβει σε περιβάλλον PlanetLab ο συνολικός αριθμός μηνυμάτων που στέλνονται σε οριακούς peers για την εκτέλεση του αλγορίθμου έχει γραμμική σχέση με τον συνολικό αριθμό των peers N στο δίκτυο, όπως αποδεικνύεται και στην ενότητα 4.1.1

6.1 Μελλοντική Δουλειά

Ως μελλοντική δουλεία μπορεί να γίνει σε πρώτο στάδιο η τροποποίηση της συνάρτησης που βρίσκει τους οριακούς peers στον αλγόριθμο Platon και η οποία τώρα χρησιμοποιεί τη λίστα με όλους τους peer του δικτυού. Συγκεκριμένα οι οριακοί peers θα βρίσκονται όπως περιγράφεται στο [3] και με βάση το *Extended SkipGraph*. Επιπρόσθετα στο Platon μπορεί να βελτιωθεί ο τρόπος που ενημερώνονται οι peers για κάθε μετακίνηση κάποιου άξονα. Επίσης μπορούν να υλοποιηθούν τα *ευρέα ερωτήματα (Range Query)* και ερωτήματα *αναζήτησης k-κοντινότερων γειτόνων (ΚΚΓ),* τα οποία περιγράφονται και τα δύο στην ενότητα 2.2.3(Κατανεμημένη Διαδικασία Ερωτημάτων).

Σε επόμενο και ποίο προχωρημένο στάδιο μπορεί να γίνει η λειτουργία αποχώρησης κάποιου peer από το δίκτυο καθώς και λειτουργία ανάνηψης μετά από αποτυχία κάποιου peer. Επίσης μπορεί να τροποποιηθεί η λειτουργία εισαγωγής νέου peer έτσι ώστε ο SplitPeer να μην δίνεται από τον Bootstrap αλλά να προκύπτει μέσα από το δίκτυο των peers με κάποια κατανεμημένη διαδικασία.

Αναφορές

[1] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma and Steven Lim. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. *IEEE COMMUNICATIONS SURVEY AND TUTORIAL, MARCH 2004*

[2] Chi Zhang, Arvind Krishnamurthy, Randolph Y. Wang. SkipIndex: Towards a Scalable Peer-to-Peer Index Service for High Dimensional Data

[3] Leonidas Lymberopoulos. PLATON: Peer to Peer Load Adjusting Tree Overlay Networks. *Technical Report June 2008*

[4] Robert Morris, Ion Stoica, David Karger, M. Frans Kaashoek, Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

[5] Vincent Matossian. Description of CHORD's Location and routing mechanisms. *October 12th 2001 ECE 579*

[6] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, Member, IEEE, and John D. Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment

[7] C. Plaxton, R. Rajaraman, and A. Richa, "Accessing nearby copies of replicated objects in a distributed environment," *in Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures, 1997*.

[8] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," *in Processings of the ACM ASPLOS, November 2002*

[9] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer, "Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs," *in Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, June 2000, pp. 34–43.*

[10] M. Waldman, A. D. Rubin, and L. F. Cranor, "Publius: a robust, tamperevident, censorship-resistant, web publishing system," *in Processings of the Ninth USENIX Security Symposium, Denver, CO, USA, 2000.*

[11] James Aspnes, Gauri Shah. Skip Graphs

[12] *PlanetLab*. http://www.planet-lab.org/

[13] *PlMan* . http://www.cs.washington.edu/research/networking/cplane/

Παράρτημα
Α

TABLE I
A COMPARISON OF VARIOUS *Structured* P2P OVERLAY NETWORK SCHEMES

Algorithm	Structured P2P Overlay Network Comparisons					
Taxonomy	CAN	Chord	Tapestry	Pastry	Kademlia	Viceroy
Decentralization	DHT functionality on Internet-like scale					
Architecture	Multi-dimensional ID coordinate space.	Uni-directional and Circular NodeID space.	Plaxton-style global mesh network.	Plaxton-style global mesh network.	XOR metric for distance between points in the key space.	Butterfly network with connected ring of predecessor and successor links, data managed by servers.
Lookup Protocol	key,value pairs to map a point P in the coordinate space using uniform hash function.	Matching Key and NodeID.	Matching suffix in NodeID.	Matching Key and prefix in NodeID.	Matching Key and Node-ID based routing.	Routing through levels of tree until a peer is reached with no downlinks; vicinity search performed using ring and leveling links.
System Parameters	N-number of peers in network d-number of dimensions.	N-number of peers in network.	N-number of peers in network B-base of the chosen peer identifier.	N-number of peers in network b-number of bits ($B = 2^b$) used for the base of the chosen identifier.	N-number of peers in network b-number of bits ($(B = 2^b)$ of NodeID.	N-number of peers in network.
Routing Performance	$O(d.N^{\frac{1}{d}})$	$O(logN)$	$O(log_B N)$	$O(log_B N)$	$O(log_B N) + c$ where c = small constant	$O(logN)$
Routing State	$2d$	$logN$	$log_B N$	$Blog_B N + Blog_B N$	$Blog_B N + B$	$logN$
Peers Join/Leave	$2d$	$(logN)^2$	$log_B N$	$log_B N$	$log_B N + c$ where c = small constant	$logN$
Security	Low level. Suffers from man-in-middle and Trojan attacks.					
Reliability/Fault Resiliency	Failure of peers will not cause network-wide failure. Multiple peers responsible for each data item. On failures, application retries.	Failure of peers will not cause network-wide failure. Replicate data on multiple consecutive peers. On failures, application retries.	Failure of peers will not cause network-wide failure. Replicate data across multiple peers. Keep track of multiple paths to each peer.	Failure of peers will not cause network-wide failure. Replicate data across multiple peers. Keep track of multiple paths to each peer.	Failure of peers will not cause network-wide failure. Replicate data across multiple peers.	Failure of peers will not cause network-wide failure. Load incurred by lookups routing evenly distributed among participating lookup servers.

Πίνακας 1 [1]

TABLE II

A COMPARISON OF VARIOUS *Unstructured* P2P OVERLAY NETWORK SCHEMES

Algorithm	Unstructured P2P Overlay Network Comparisons				
Taxonomy	Freenet	Gnutella	FastTrack/KaZaA	BitTorrent	Overnet/eDonkey2000
Decentralization	Loosely DHT functionality.	Topology is flat with equal peers.	No explicit central server. Peers are connected to their Super-Peers.	Centralized model with a Tracker keeping track of peers.	Hybrid two-layer network composed of clients and servers.
Architecture	Keywords and descriptive text strings to identify data objects.	Flat and Ad-Hoc network of servents (peers). Flooding request and peers download directly.	Two-level hierarchical network of Super-Peers and peers.	Peers request information from a central Tracker.	Servers provide the locations of files to requesting clients for download directly.
Lookup Protocol	Keys,Descriptive Text String search from peer to peer.	Query flooding.	Super-Peers.	Tracker.	Client-Server peers.
System Parameters	None	None	None	*.torrent* file.	None
Routing Performance	Guarantee to locate data using *Key* search until the requests exceeded the Hops-To-Live (HTL) limits.	No guarantee to locate data; Improvements made in adapting Ultrapeer-client topologies; Good performance for popular content.	Some degree of guarantee to locate data, since queries are routed to the Super-Peers which has a better scaling; Good performance for popular content.	Guarantee to locate data and guarantee performance for popular content.	Guarantee to locate data and guarantee performance for popular content.
Routing State	Constant	Constant	Constant	Constant but choking (temporary refusal to upload) may occur.	Constant
Peers Join/Leave	Constant	Constant	Constant	Constant	Constant with bootstrapping from other peer and connect to server to register files being shared.
Security	Low; Suffers from man-in-middle and Trojan attacks.	Low; Threats: flooding, malicious content, virus spreading, attack on queries, and denial of service attacks.	Low; Threats: flooding, malicious or fake content, viruses, etc. Spywares monitor the activities of peers in the background.	Moderate; Centralized Tracker manage file transfer and allows more control which makes it much harder faking IP addresses, port numbers, etc.	Moderate; Similar threats as the FastTrack and BitTorrent.
Reliability/Fault Resiliency	No hierarchy or central point of failure exists.	Degradation of the performance; Peer receive multiple copies of replies from peers that have the data; Requester peers can retry.	The ordinary peers are reassigned to other Super-Peers.	The Tracker keeps track of the peers and availability of the pieces of files; Avoid Choking by fibrillation by changing the peer that is choked once every then seconds.	Reconnecting to another server; Will not receive multiple replies from peers with available data.

Πίνακας 2 [1]