Αλγόριθμοι Αποκεντρωμένων Δεδομένων

Τεχνική Αναφορά

Εργασία Εξαμήνου

Αθανάσιος Τάσης[1093503] — Σταύρος Μπαντζής[1097449]

Τμήμα Μηχανικών Η/Υ & Πληροφορικής
Πανεπιστήμιο Πατρών

29 Ιανουαρίου 2026

Περίληψη

Περίληψη: Σκοπός της παρούσας τεχνικής αναφοράς είναι η υλοποίηση και η αξιολόγηση Peer 2 Peer αλγόριθμων & πρωτοκώλων DHT(Distributed Hash Tables). Αφορά δηλαδή την προσωμοίωση ενός αποκεντρωμένου συστήματος αποθήκευσης και ανάκτησης δεδομένων χωρίς την ανάγκη για κεντρική ρυθμιστική οντότητα. Θα εξερευνήσουμε και θα αξιολογήσουμε τρόπους βελτίωσης της απόδησης αυτών των συστημάτων καθώς και την ανθεκτικότητά τους σε ανωμαλίες δικτύου.

Περιεχόμενα

1 Dataset

1.1 Python Virtual Enviroment

Για την υλοποίηση της παρούσας εργασίας χρησημοποιήσαμε Python σε συνδιασμό με packages που θα αναφέρονται σταδιακά όπου χρειάζεται.

Για την εκτέλεση των .py αρχείων αποφασίσαμε να υιοθετήσουμε virtual enviroment για να μπορούν να τρέχουν σε όλες τις συσκευές ομαλά και να διαχειριζόμαστε τα πακέτα και τις εκδόσεις τους.

1.2 Packages

Για την προεπεξεργασία των εγγράφων και την ορθή διαχείριση των δεδομένων χρησιμοποιήθηκαν τα ακόλουθα packages της Python (έκδοση 3.11):

- `os`: Διαχείριση διαδρομών και πρόσβαση στα αρχεία του συστήματος + υπολογισμός αριθμού πυρήνων για μέγιστη αξιοποίηση παραλληλίας.
- `Optional, List, Dict, Any`: Μετασχηματισμός αλφαριθμητικών δεδομένων. Πράξεις μεταξύ λιστών κτλ.
- `xxhash`: Αλγοριθμός κατακερματισμού των 64 bit.
- `pickle`: Αποθήκευση Distributed Hashed Table σε αρχείο .pkl που μας επιτρέπει την επαναχρησημοποίηση του χωρίς να το ανακατασκευάζω κάθε φορά που θέλω να αλλάξω κάτι στον κώδικα.
- `ProcessPoolExecutor`: Επιτρέπει την αξιοποίηση πυρήνων χωρίς το πρόβλημα του GIL (Global Intepreter Lock) της Python.
- `csv`: Read & Write σε .csv αρχεία.

Code 1: Packages Import

```
from typing import Optional, List, Dict, Any
import bisect
from node import Node
import xxhash
from concurrent.futures import ProcessPoolExecutor
import pickle
import csv
import os
```

1.3 Σύνολο Δεδομένων

Το dataset που χρησημοποιήσαμε είναι το Movies Metadata Cleaned Dataset (1900–2025) και αποτελείτε από 946,000 εγγραφές από ταινίες που σκηνοθετήθηκαν από το 1900 μέχρι το

2025.

Η κάθε εγγραφή αποθηκεύει 14 διαφορετικά στοιχεία για κάθε ταινία. Συγκεκριμένα:

Πίνακας 1: Dataset Columns Overview

id	Unique movie identifier
title	Official movie title
adult	Boolean flag indicating adult content
original_language	Original spoken language (ISO 639-1 code)
origin_country	List of production countries
release_date	Movie release date
genre_names	List of genres associated with the movie
company_names	Names of involved production companies
budget	Reported production budget (USD)
revenue	Worldwide gross revenue (USD)
runtime	Duration in minutes
popularity	Popularity score (as provided by TMDB)
vote_average	Average user rating
vote_count	Number of votes received

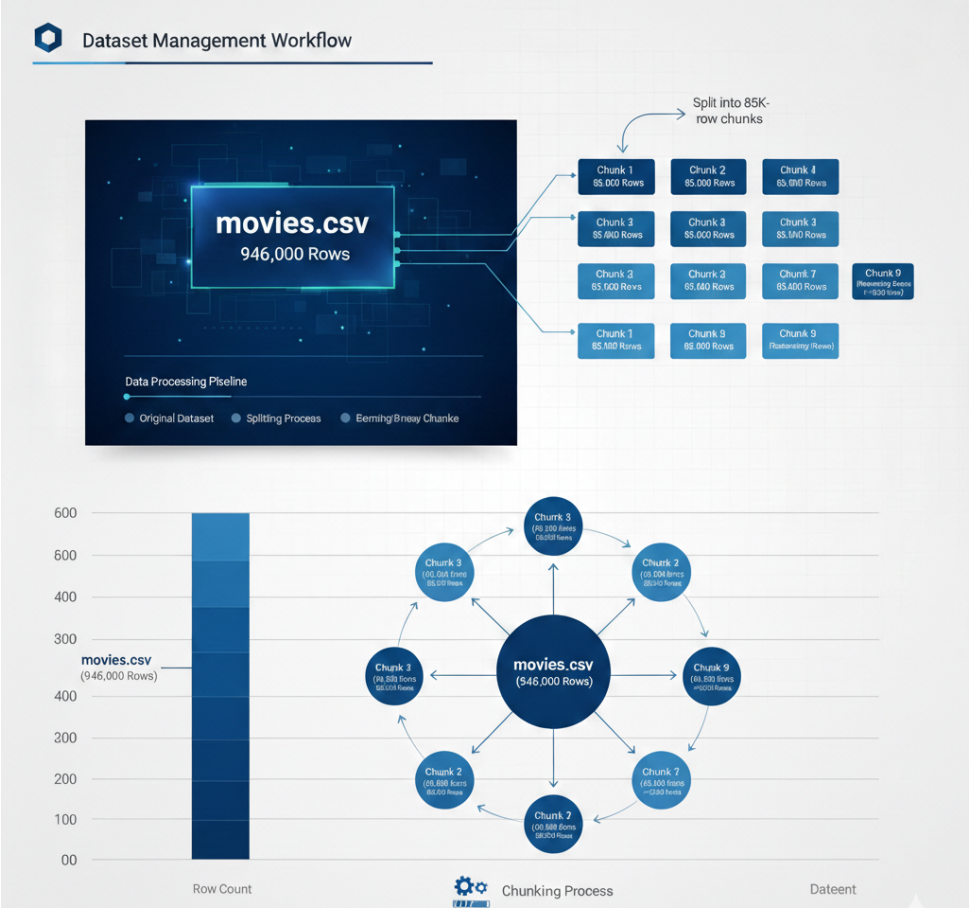
Το αρχείο .csv έχει μέγεθος 130.7 MB.

1.4 Εισαγωγή δεδομένων(Data Import)

Λόγο του μεγάλου αριθμού εγγραφών και μεγέθους του αρχείου, η απευθείας εισαγωγή των δεδομένων στην Python θα ήταν πολύ αργή και μη αποδοτική.

Η εισαγωγή λοιπόν γίνεται τμηματικά. ”Σπάμε” τις εγγραφές σε ”κομμάτια” των 85,000 και ορίζουμε, με την βοήθεια του *ProcessPoolExecutor*, 1 process σε κάθε πυρήνα που αναλαμβάνει την ανάγνωση των εγγραφών και την εισαγωγή τους στα αντίστοιχα nodes του DHT. Η επιλογή του αριθμού 85,000 έγινε εσκεμένα καθώς το σύστημα μας έχει 12 πυρήνες και έτσι:

$$12 \times 85{,}000 = 1{,}020{,}000 > 946{,}000$$

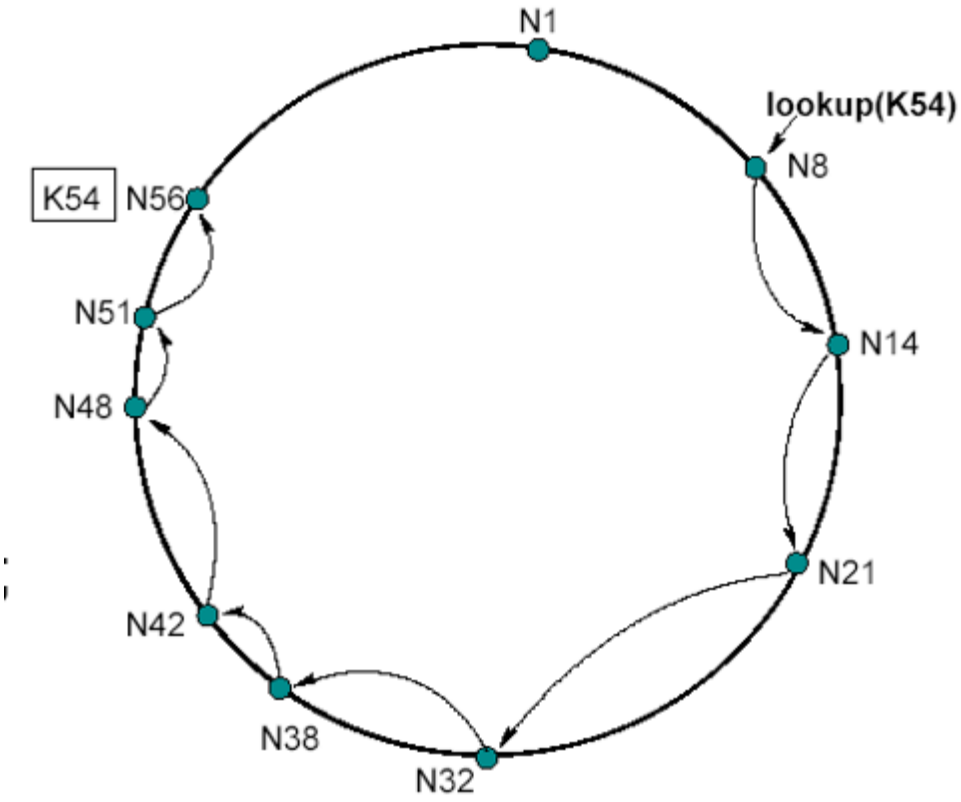


Σχήμα 1: Data Splitting

2 Chord

2.1 Ορισμός

Ο αλγόριθμος chord λειτουγεί ως εξής:



Σχήμα 2: Chord Ring with nodes

Πρόκειτε για ένα ιδεατό κύκλο που έχει αριθμίσημα σημεία. Ο αριθμός των σημείων αυτών ορίζεται από τον αλγόριθμο κατακερματισμού. Στην περίπτωσή μας ο xxhash μας δίνει την δυνατότητα 64 bit –> $2^{64} - 1$ σημεία.

2.1.1 Nodes

Nodes ή αλλιώς κόμβοι μπορεί να είναι ανεξάρτητοι χρήστες που υπολογιστική και αποθηκευτική δυνατότητα ή servers. Ο κάθε κόμβος παίρνει μια τιμή και συγκεκριμένα:

$$Node\,point\ =\ xxhash64(Node\,Name\,or\,Node\,IP\,Address)\ \text{mod}\ 2^{64}$$

Κάθε κόμβος αποθηκεύει τις εξής τιμές:

id	Η hashed τιμή του όπως αναφέραμε πρίν
successor	Ένα pointer στον επόμενο γείτονά του
predecessor	Ένα pointer στον προηγούμενο γειτονά του
data	Τα δεδομένα που κάνει host αυτός ο κόμβος
finger_table	Μια λίστα με συγκεκριμένους successor

Τα δεδομένα αποθηκεύονται στην μεταβλητή data ορίζοντας σαν key τον τίτλο μιας ταινίας και σαν μια εμφωλευμένη λίστα τα υπόλοιπα δεδομένα

Code 2: Node logic decleration

```python
class Node:
    def __init__(self, node_id: int, m_bits: int):
        self.id: int = node_id
        self.successor: "Node" = self
        self.predecessor: Optional["Node"] = None
        self.fingers: List["Node"] = [self] * m_bits
        self.data: Dict[str, List[Any]] = {}
```

2.1.2 Finger Table

Το finger table είναι ένας πίνακας μια routing μέθοδος η οποία μειώνει το κόστος αναζήτησης ενος κόμβου από $O(N)$ σε $O(\log N)$. Ο κάθε κόμβος έχει και από ένα πίνακα που αποτελείται από m εγγραφές. m είναι τα bit της συνάρτησης κατακερματισμού που χρησημοποιούμε.

For $i = 1, 2, \ldots, m$:

$$\text{finger}[i] = \begin{cases} \text{start}_i = (n + 2^{i-1}) \bmod 2^m \\ \text{node}_i = \text{successor}(\text{start}_i) \end{cases}$$

Meaning of Each Entry

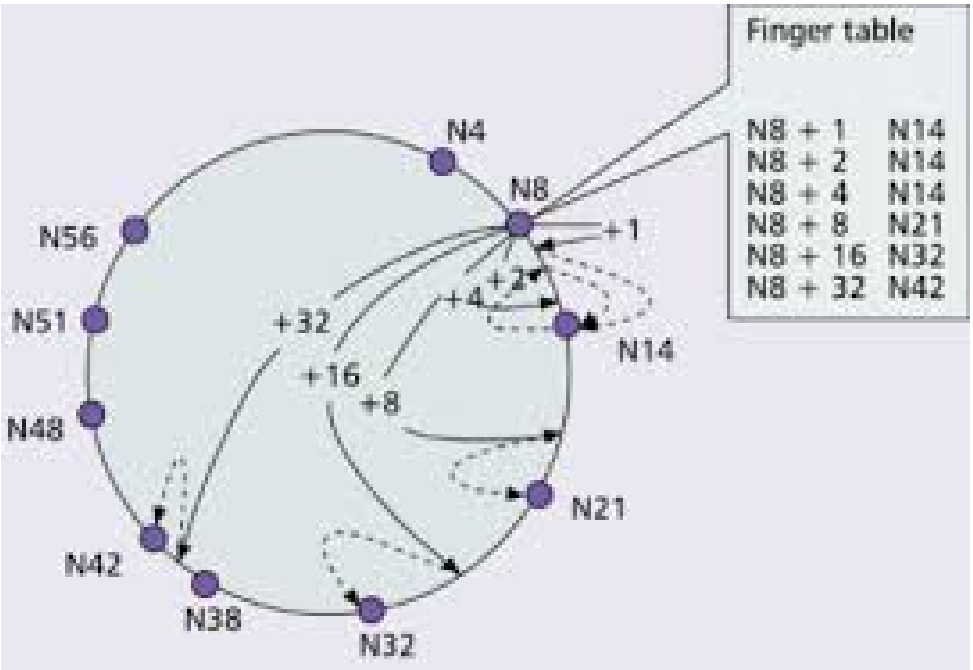
- start_i is the first identifier at distance 2^{i-1} clockwise from node n.
- node_i is the first node whose identifier is greater than or equal to start_i.
- Each finger entry covers an interval of exponentially increasing size.

Finger Table Representation

Entry i	Start $(n + 2^{i-1}) \bmod 2^m$	Stored Node
1	$n + 1$	successor$(n + 1)$
2	$n + 2$	successor$(n + 2)$
3	$n + 4$	successor$(n + 4)$
$\vdots$	$\vdots$	$\vdots$
m	$n + 2^{m-1}$	successor$(n + 2^{m-1})$

Πίνακας 2: Routing Performance of Chord

	Worst Case Time	**Average Case Time**
`Chord Finger Table`	$O(N)$	$O(0.5 \times N)$
`Chord Finger Table`	$O(\log N)$	$O(0.5 \times \log N)$



Σχήμα 3: Chord Ring with nodes

2.2 Σχεδίαση

2.2.1 Εύρεση Host Node

Η αναζήτηση του κοντινότερου κόμβου δωθέντος ένα query q είναι απαραίτητη για 2 λόγους. Αρχικά, είναι απαραίτητη κατα την διαδικασία εισαγωγής δεδομένων. Όταν εισάγουμε μια καινούργια ταινία στο σύστημα πρέπει να βρούμε σε ποιό node θα πρέπει να αποθηκευτεί. Επίσης κατα την αναζήτηση ενός τίτλου ταινίας από τον χρήστη θα πρέπει να κάνουμε locate τον κατάληλλο κόμβο που περιέχει την πληροφορία που αναζητεί. Ο αλγόριθμος εύρεσης είναι ο εξής:

1. Πάω στον πρώτο κόμβο του συστήματός μου.
2. Βλέπω το finger table του.
3. Βρίσκω σε ποιό διάστημα ανήκει ο τίτλος που ψάχνω.
4. Πάω στον μικρότερο κόμβο.
5. Κοιτάω αν αυτό που ψάχνω είναι στο διάστημα του κόμβου και του succesor του.
6. Αν ναί τοτε επιστρέφω τον succesor.
7. Αν όχι τότε πάω στο βήμα 2.

Code 3: find_succesor function

```
1    def find_successor(self, key_id: int):
2        curr = self
```

```
        hops = 0
        while not in_range(key_id, curr.id, curr.successor.id, True):
            next_node = curr.closest_preceding_node(key_id)
            if next_node == curr:
                break
            curr = next_node
            hops += 1
        return curr.successor, hops + 1

    def closest_preceding_node(self, key_id: int) -> "Node":
        for finger in reversed(self.fingers):
            if finger is not None and in_range(finger.id, self.id,
               key_id, False):
                return finger
        return self
```

2.2.2 Εισαγωγή κόμβου

Τα βήματα για την εισαγωγή ενός καινούργιου κόμβου στο DHT μας είναι:

1. Κάνω hash το όνομα του κόμβου ή την IP Address του.
2. Προσθέτω τον κόμβο στην λίστα κόμβων του DHT.
3. Φτιάχνω το finger table όπως αναφέραμε εδώ 2.1.2
4. Επαναυπολογίζω τα finger table για τους κόμβους πρίν απο αυτόν.

2.2.3 Αποτυχία Κόμβου: Node Failure

Για να προσομοιώσουμε σωστά ενα node failure απλά αφαιρούμε τον κόμβο και διαγράφουμε όλα τα δεδομένα που περιέχει.

2.2.4 Εισαγωγή δεδομένων

Η διαδικασία εισαγωγής θα πρέπει να λαμβάνει υπόψην και την περίπτωση αποτυχίας κόμβου. Για να είναι robust το σύστημά μας και ανθεκτικό στις πολλαπλές αποτυχίες αναγκαζόμαστε να φτιάξουμε διάφορα backup. Στην ουσία κάθε ταινία έχει και από r αντίγραφα. Το r καθορίζετε στην συνέχεια.

Τα βήματα:

1. Καλώ την συνάρτηση find_successor 2.2.1 για να βρώ σε ποιό κόμβο θα πρέπει να αποθηκεύσω την ταινία.
2. Προσθέτω τα δεδομένα στο πεδίο data του node που βρήκα στο βήμα 1.
3. Βρίσκω τον ακριβώς απέναντι κόμβο και αποθηκεύω και εκεί την ταινία.

4. Αποθηκεύω επίσης τα δεδομένα και στους r successors του

Έτσι αν ο αρικός κόμβος αποτύχει, η ταινία υπάρχει ακόμα στον απέναντι κόμβο και σε r successors αυτού.

2.2.5 Αναζήτηση

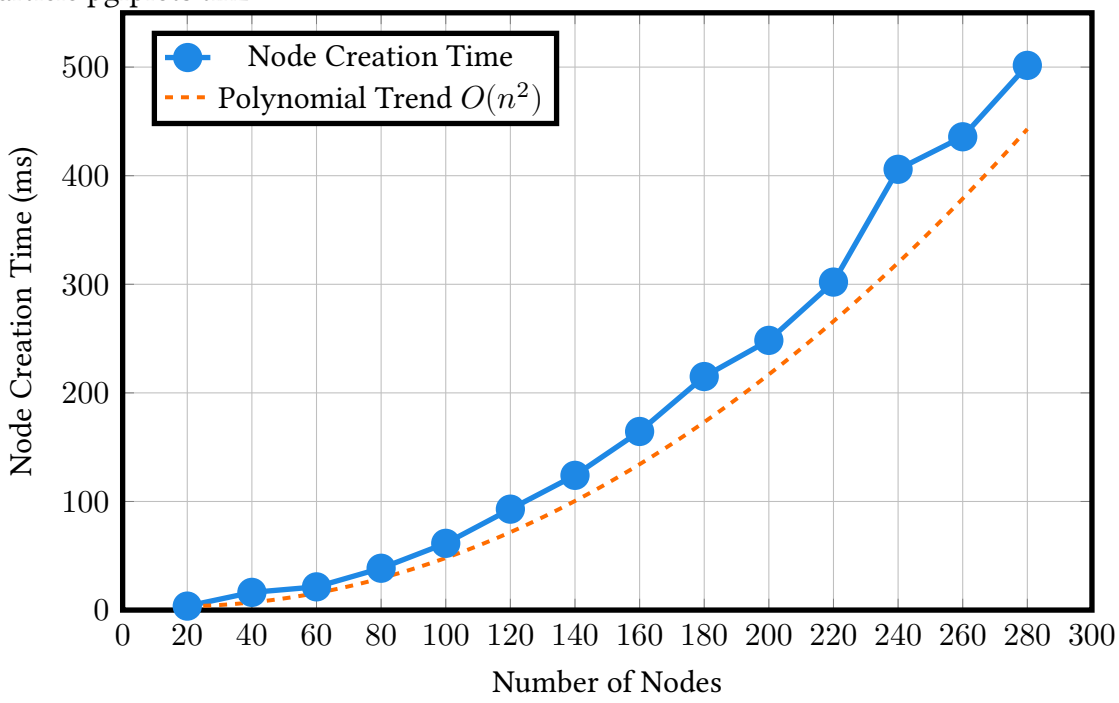
Όταν αναζητούμε τον τίλτο μιας ταινίας, αναμένουμε να μας επιστραφούν όλα τα στοιχεία των ταινιών που οι τίτλοι τους έχουν exact match με το query μας. Η διαδικασία της αναζήτησης αποτελείται απο 2 στάδια. Το πρώτο είναι η εύρεση του κόμβου που κάνει host την ταινία και το δεύτερο είναι η εύρεση της ακριβούς τοποθεσίας μέσα στο node. Για το δεύτερο κομμάτι εφαρμόσαμε την build-in function της Python 3.11 get() και όχι κάποιο τύπο B+ Trees, R-Trees, KD-Trees etc καθώς λόγο του μεγέθους του Dataset σε σχέση με τον αριθμό των κόμβων έχουμε μικρή πυκνότητα ανα κόμβο με περίπου 15-30K movies/node. Αν χρησημοποιούσαμε local indexes τότε η ταχύτητα ανάκτησης θα μειωνόταν. Με πιό πυκνούς κόμβους θα είχαμε καλύτερα αποτελέσματα. Το πρώτο στάδιο υλοποιήθηκε με την χρήση της συνάρτησης find_successor.

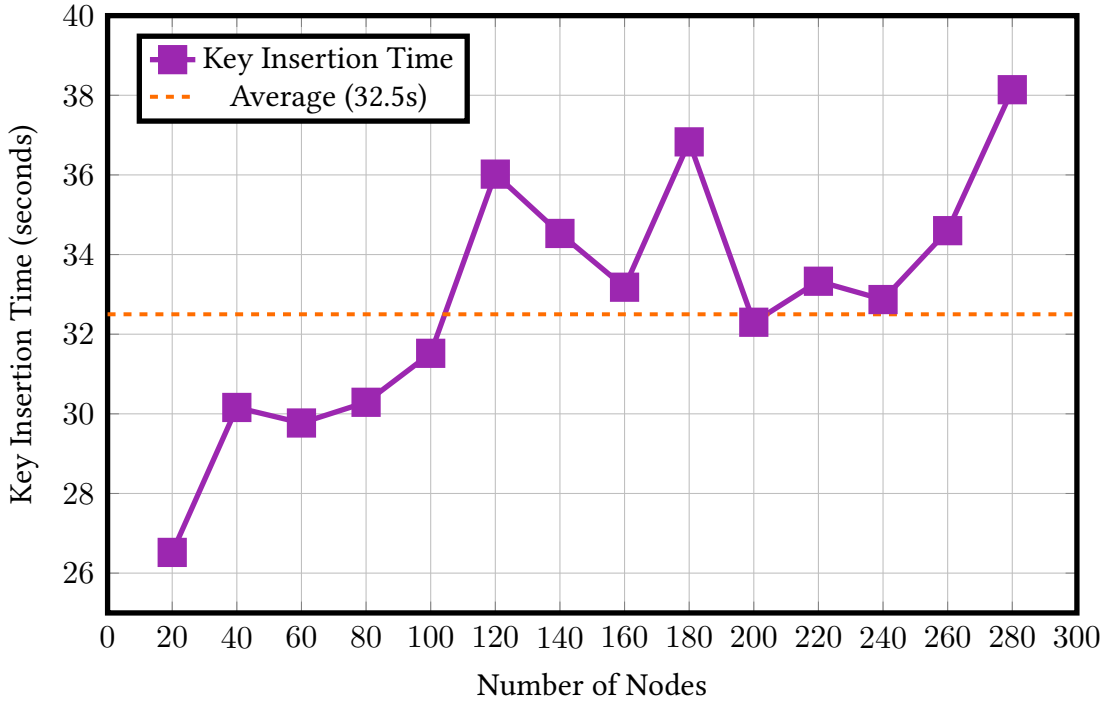
2.3 Κατασκευή

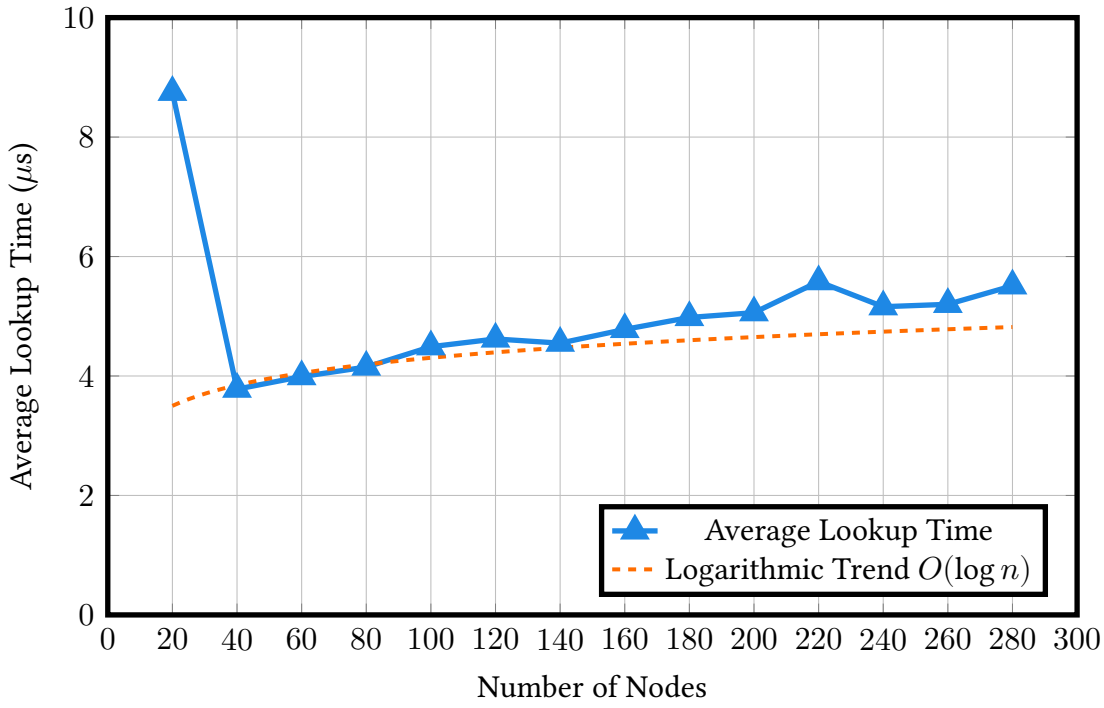
2.3.1 Βέλτιστος αριθμός κόμβων

Για την επιλογή του βέλτιστου αριθμού κόμβων τρέξαμε κάποια τεστ που μετράνε τον μέσο χρόνο αναζήτησης 1 ταινίας. Πρώτα επιλέξαμε 30,000 random τίτλους απο το Dataset και τους αποθηκεύσαμε σε ενα .csv αρχείο. Στην συνέχεια υπολογίσαμε: 1) Χρόνο Κατασκευής DHT, 2) Χρόνο εισαγωγής όλου του Dataset, 3) M.O. Lookup time

article pgfplots tikz

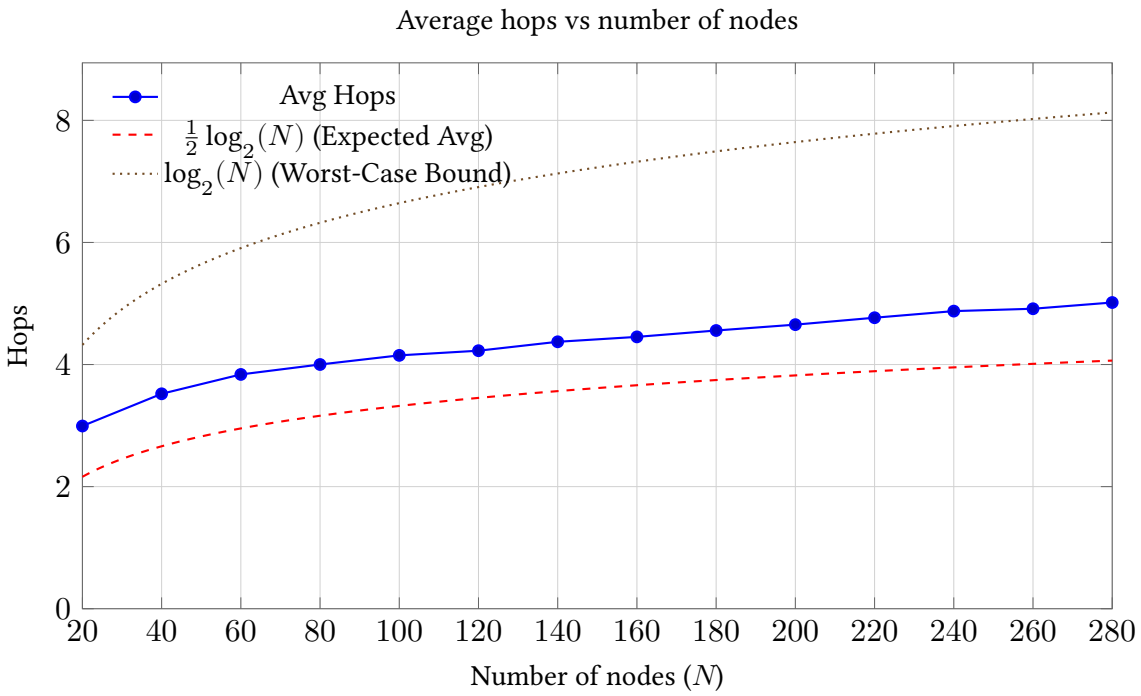






2.3.2 Hops

Επίσης υπολογίσαμε και τον Μ.Ο. των hops που χρειάζονται για να βρεθεί μια ταινία. Στην ουσία μέσα απο πόσα nodes θα 'πηδήξει' ο αλγόριθμος για να φτάσει στην απάντηση.



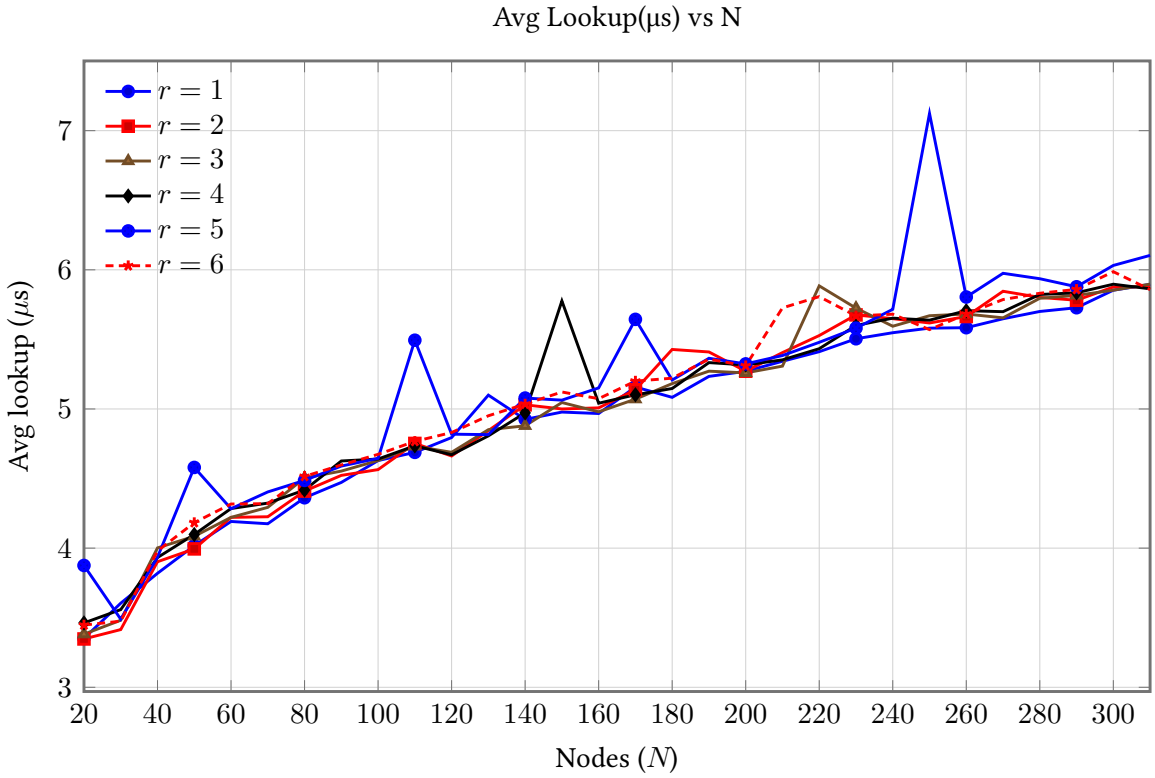
Σχήμα 4: Average hops compared to average & worst case senario.

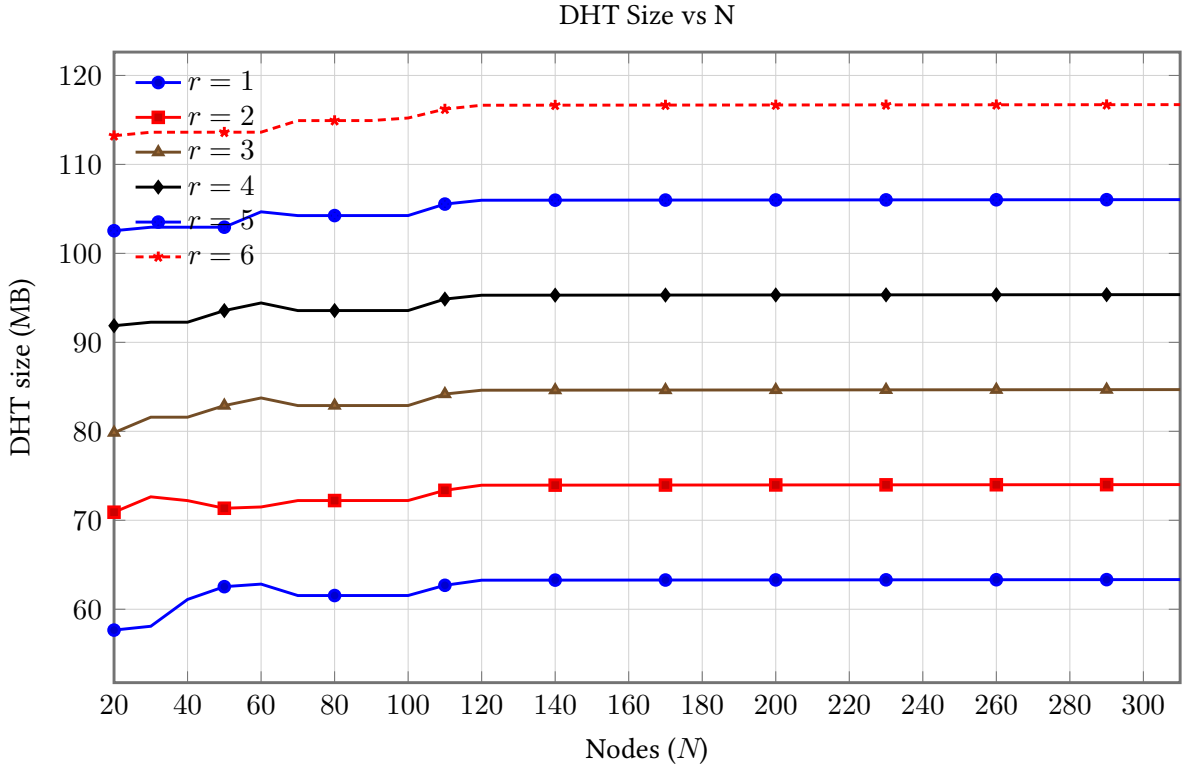
Επιλογή N

Βάση των αποτελεσμάτων, η λογική λέει οτι ο βέλτιστος αριθμός κόμβων είναι 20. Όμως ο μικρός αριθμός κόμβων επιφέρει και ένα μεγάλο αρνητικό αποτέλεσμα. Αυτό είναι η έλειψη redundancy. Αν μια περιοχή κόμβων αποτύχουν τότε, ακόμη και μόνο 5 κόμβοι, θα αποτελούν μεγάλο ποσοστό των συνολικών κόμβων. (25 % !!!).

Για αυτόν τον λόγο επιλέξαμε να χρησημοποιήσουμε 240 κόμβους καθώς το trade-off μεταξύ χρόνου υπολογισμού - robustness είναι πολύ μικρό. Δηλαδή με μια μικρή καθυστέρηση στον χρόνο υπολογισμού, ασφαλίζω το σύστημά μου απο πολλαπλά failures.

Σύγκριση Χώρου - N & Χρόνου - N





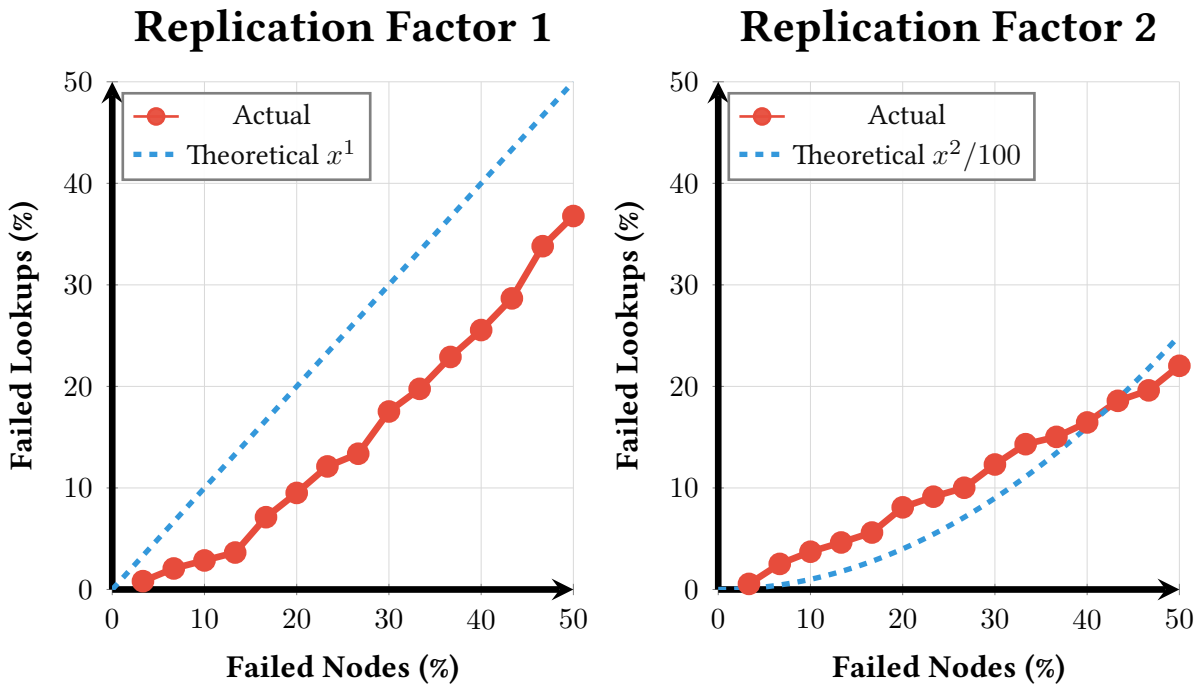
Παρατηρούμε οτι ο αριθμός των κόμβων δεν επηρεάζει μέγεθος του αρχείου, αλλά επιφέρει μια αμελητέα επιβάρυνση στον χρόνο. Ο οριθμός των αντιγράφων από την άλλη, r, αυξάνει σημαντικά το μέγεθος του DHT και συγκεκριμένα μπορούμε να εκφράσουμε το μέγεθος του DHT ως:

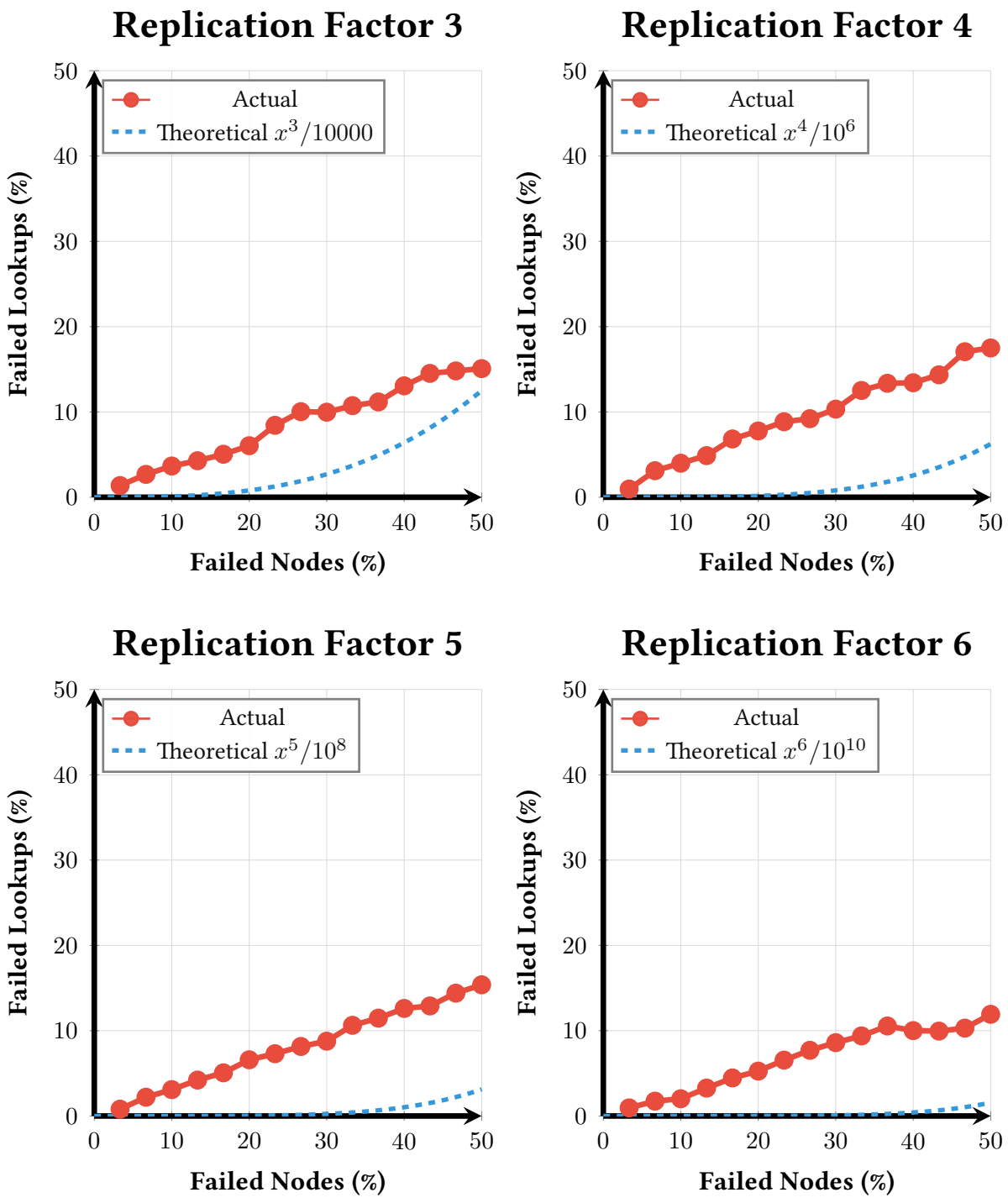
$$\text{Size(DHT)} = 63.33 + 10.68 \cdot r$$

2.3.3 Έλεγχος Ανθεκτικότητας - Robustness Check

Για να ελέγξουμε το πόσο ανθεκτικό είναι το DHT μας σε αποτυχίες κόμβων θα ορίσουμε ένα σταθερό αριθμό κόμβων N = 300 και θα δοκιμάσουμε να κάνουμε manually fail από 10 έως 150 κόμβους. Για κάθε τέστ, θα αξιολογήσουμε πόσες από τις 30,000 ταινίες, που έχουμε ως sample από προηγούμενο βήμα, κατάφεραν να ανακτηθούν. Βάση του paper του Chord, το ποσοστό των failed lookups πρέπει να είναι:

$$\text{Failed lookup \%} = 100 \cdot \left(\frac{\text{failed nodes}}{N}\right)^r$$





Παρατήρηση

Παρατηρώ ότι για r > 3 η διαφορά στα ποσοστά αποτυχίας βελτιώνεται ελάχιστα. Για αυτόν τον λόγο θα χρησημοποιήσω r = 3. Η χρήση μεγαλύτερου r θα αυξήσει δραματικό τον αποθηκευτικό χώρο που χρειάζομαι και το κόστος κατασκευής του DHT.

Παρατήρηση II

Οι λόγοι που για r>3 τα failed lookups είναι μεγαλύτερα από τα ideal failed lookups είναι 2:

- Τα αντίγραφα αποθηκέυονται σε γειτονικούς κόμβους, με αποτέλεσμα, ενα τοπικό 'blackout' να αφανίσει την ταινία τελείως από το DHT
- Δεν υπάρχει αλγόριθμος που να μας βεβαιώνει τον αριθμό των αντιγράφων μιας ταινίας όλες τις στιγμές.

3 Pastry

Περίληψη Το Pastry είναι ένα αποκεντρωμένο πρωτόκολλο *Distributed Hash Table* (DHT) που σχεδιάστηκε για αποδοτική δρομολόγηση και εντοπισμό πόρων σε μεγάλης κλίμακας peer-to-peer συστήματα. Κάθε κόμβος και κάθε αντικείμενο αντιστοιχίζεται σε ένα μοναδικό αναγνωριστικό από έναν μεγάλο και αραιό χώρο τιμών, επιτρέποντας ομοιόμορφη κατανομή φορτίου. Το σύστημα εγγυάται ότι κάθε αίτημα για ένα κλειδί θα δρομολογηθεί στον υπεύθυνο κόμβο σε αναμενόμενο λογαριθμικό αριθμό αλμάτων. Στην παρούσα αναφορά παρουσιάζονται η θεωρητική βάση του Pastry, η δομή της τοπολογίας και ο αλγόριθμος δρομολόγησης, οι μηχανισμοί εισόδου και αποχώρησης κόμβων, καθώς και ζητήματα απόδοσης, ανθεκτικότητας και πρακτικής χρήσης.

3.1 Τι είναι το Pastry DHT

Το Pastry αποτελεί ένα overlay δίκτυο το οποίο υλοποιεί key-based routing πάνω από το υπάρχον δίκτυο IP. Σε αντίθεση με παραδοσιακά συστήματα πελάτη-εξυπηρετητή, το Pastry δεν βασίζεται σε κεντρικούς κόμβους, αλλά κατανέμει την ευθύνη αποθήκευσης και δρομολόγησης σε όλους τους συμμετέχοντες κόμβους. Κάθε κόμβος αναλαμβάνει την εξυπηρέτηση ενός τμήματος του χώρου κλειδιών, γεγονός που επιτρέπει στο σύστημα να κλιμακώνεται σε πολύ μεγάλους πληθυσμούς κόμβων. Θεμελιώδης εγγύηση του Pastry είναι ότι η δρομολόγηση ολοκληρώνεται σε $O(\log N)$ άλματα, υπό την παραδοχή ομοιόμορφης κατανομής των αναγνωριστικών.

3.2 Πώς λειτουργεί (υψηλού επιπέδου)

3.2.1 Χώρος αναγνωριστικών

Ο χώρος αναγνωριστικών του Pastry είναι συνήθως μεγέθους 128 bit και παράγεται μέσω κρυπτογραφικών συναρτήσεων κατακερματισμού. Η χρήση hash εξασφαλίζει ότι τα nodeIds κατανέμονται ομοιόμορφα, μειώνοντας την πιθανότητα ανισοκατανομής φορτίου. Κάθε αντικείμενο ή υπηρεσία που αποθηκεύεται στο σύστημα αντιστοιχίζεται επίσης σε ένα κλειδί από τον ίδιο χώρο. Ο υπεύθυνος κόμβος για ένα κλειδί ορίζεται ως εκείνος του οποίου το nodeId είναι αριθμητικά πλησιέστερο στο κλειδί, γεγονός που δημιουργεί μια σαφή και ντετερμινιστική ανάθεση ευθυνών.

3.2.2 Βασική ιδέα δρομολόγησης

Η δρομολόγηση στο Pastry βασίζεται στη σύγκριση προθεμάτων των αναγνωριστικών σε βάση 2^b. Σε κάθε βήμα της δρομολόγησης, το μήνυμα προωθείται σε έναν κόμβο που μοιράζεται μεγαλύτερο κοινό πρόθεμα με το ζητούμενο κλειδί σε σχέση με τον τρέχοντα κόμβο. Αυτό το αναλλοίωτο (invariant) της αύξησης του κοινού προθέματος εγγυάται ότι δεν δημιουργούνται κύκλοι και ότι η διαδικασία θα τερματίσει σε πεπερασμένο αριθμό βημάτων. Υπό κανονικές συνθήκες, ο αριθμός των αλμάτων είναι λογαριθμικός ως προς το πλήθος των κόμβων $O(\log_{2^b} N)$.

3.3 Τοπολογία και Δρομολόγηση στο Pastry

3.3.1 Αλγόριθμος Λογικής Δρομολόγησης

Για κάθε εισερχόμενο αίτημα με κλειδί K, ο κόμβος ακολουθεί τα εξής βήματα:

1. **Έλεγχος Leaf Set:** Αν το K εμπίπτει στο εύρος των αναγνωριστικών του leaf set, το αίτημα προωθείται απευθείας στον κόμβο που είναι αριθμητικά πλησιέστερος στο K.

2. **Έλεγχος Routing Table:** Αν το K δεν καλύπτεται από το leaf set, ο κόμβος αναζητά στον πίνακα δρομολόγησης έναν κόμβο που μοιράζεται μεγαλύτερο κοινό πρόθεμα με το K από τον τρέχοντα.

3. **Σπάνια Περίπτωση (Convergence):** Αν οι παραπάνω έλεγχοι αποτύχουν, το αίτημα προωθείται σε έναν κόμβο που έχει το ίδιο μήκος προθέματος αλλά είναι αριθμητικά πλησιέστερος στο K.

3.3.2 Πίνακας δρομολόγησης (routing table)

Κάθε κόμβος διατηρεί έναν πίνακα δρομολόγησης δομημένο σε επίπεδα, όπου κάθε επίπεδο αντιστοιχεί σε ένα μήκος κοινού προθέματος. Για κάθε πιθανή τιμή του επόμενου ψηφίου, ο πίνακας περιέχει έναν ή περισσότερους υποψήφιους κόμβους. Το μέγεθος του πίνακα είναι $O(\log_{2^b} N)$, ενώ η επιλογή της παραμέτρου b επηρεάζει άμεσα το trade-off μεταξύ μεγέθους πίνακα και αριθμού αλμάτων. Συνήθως χρησιμοποιείται $b = 4$, που σημαίνει ότι τα αναγνωριστικά εξετάζονται σε δεκαεξαδική μορφή (βάση 16). Κάθε βήμα δρομολόγησης "διορθώνει" ένα ψηφίο του προθέματος. Αυτό εξηγεί γιατί ο πίνακας δρομολόγησης έχει 16 στήλες (σε κάθε επίπεδο).

3.3.3 Leaf set

Το leaf set αποτελείται από τους L κόμβους με τα αριθμητικά πλησιέστερα nodeIds. Ο ρόλος του είναι κρίσιμος για τη διασφάλιση τοπικής ορθότητας, ιδιαίτερα όταν το ζητούμενο κλειδί βρίσκεται κοντά στο nodeId του κόμβου. Ακόμη και αν ο πίνακας δρομολόγησης είναι ελλιπής ή περιέχει ξεπερασμένες εγγραφές, το leaf set επιτρέπει στο σύστημα να καταλήξει στον σωστό υπεύθυνο κόμβο.

3.3.4 Neighborhood set

Το neighborhood set χρησιμοποιείται για βελτιστοποίηση εγγύτητας σε επίπεδο δικτύου. Περιλαμβάνει κόμβους που είναι γεωγραφικά ή δικτυακά κοντά, με βάση μετρήσεις καθυστέρησης. Αν και δεν επηρεάζει την ορθότητα της δρομολόγησης, συμβάλλει στη μείωση της συνολικής καθυστέρησης των αιτημάτων. Το Neighborhood Set δεν χρησιμοποιείται στη δρομολόγηση (routing) για την εύρεση του κλειδιού, αλλά χρησιμοποιείται κυρίως κατά την είσοδο ενός νέου κόμβου (Join) για να βρει ο νέος κόμβος άλλους "κοντινούς" του κόμβους και να αντιγράψει τους πίνακες δρομολόγησής τους.

3.4 Είσοδος νέων κόμβων και αποτυχίες κόμβων

3.4.1 Είσοδος νέου κόμβου

Η διαδικασία εισόδου ενός νέου κόμβου πραγματοποιείται χωρίς κεντρικό συντονισμό. Ο νέος κόμβος επιλέγει τυχαίο nodeId και επικοινωνεί με έναν κόμβο bootstrap. Μέσω ενός μηνύματος join, συλλέγει σταδιακά τις απαραίτητες πληροφορίες για να αρχικοποιήσει τις δομές του. Το κόστος της διαδικασίας είναι $O(\log N)$ μηνύματα, γεγονός που καθιστά το Pastry κατάλληλο για δυναμικά περιβάλλοντα. Ο νέος κόμβος X λαμβάνει το Leaf Set από τον πλησιέστερο κόμβο Z και τα επίπεδα του Routing Table από όλους τους ενδιάμεσους κόμβους της διαδρομής A...Z

3.4.2 Αποτυχίες και αυτο-επιδιόρθωση

Το Pastry έχει σχεδιαστεί ώστε να αντιμετωπίζει αποτυχίες κόμβων με τοπικούς μηχανισμούς επιδιόρθωσης. Η ύπαρξη πλεονασμού τόσο στο leaf set όσο και στον πίνακα δρομολόγησης επιτρέπει την αντικατάσταση αποτυχημένων κόμβων χωρίς παγκόσμιο συντονισμό. Αυτό αυξάνει σημαντικά την ανθεκτικότητα του συστήματος σε συνθήκες υψηλού churn.

3.5 Απόδοση

3.5.1 Κλιμάκωση

Η λογαριθμική πολυπλοκότητα τόσο στη δρομολόγηση όσο και στη συντήρηση της κατάστασης επιτρέπει στο Pastry να κλιμακώνεται σε πολύ μεγάλα δίκτυα. Υπό ιδανικές συνθήκες, η απόδοση παραμένει σταθερή ακόμη και με χιλιάδες ή εκατομμύρια κόμβους.

3.5.2 Καθυστέρηση και εγγύτητα

Η ενσωμάτωση κριτηρίων εγγύτητας επιτρέπει στο Pastry να επιτυγχάνει χαμηλότερη end-to-end καθυστέρηση σε σύγκριση με DHTs που αγνοούν τη φυσική τοπολογία του δικτύου.

3.5.3 Κόστος συντήρησης

Η συντήρηση των δομών δεδομένων απαιτεί περιοδική ανταλλαγή μηνυμάτων, γεγονός που δημιουργεί επιπλέον overhead. Η σωστή επιλογή παραμέτρων είναι καθοριστική για την ισορροπία μεταξύ ανθεκτικότητας και κόστους.

3.6 Πλεονεκτήματα και Μειονεκτήματα

Το Pastry προσφέρει αποκεντρωμένη λειτουργία, καλή κλιμάκωση και ανθεκτικότητα, ωστόσο παρουσιάζει αυξημένο κόστος συντήρησης και ευαισθησία σε κακόβουλες επιθέσεις αν δεν συνδυαστεί με μηχανισμούς ασφάλειας.

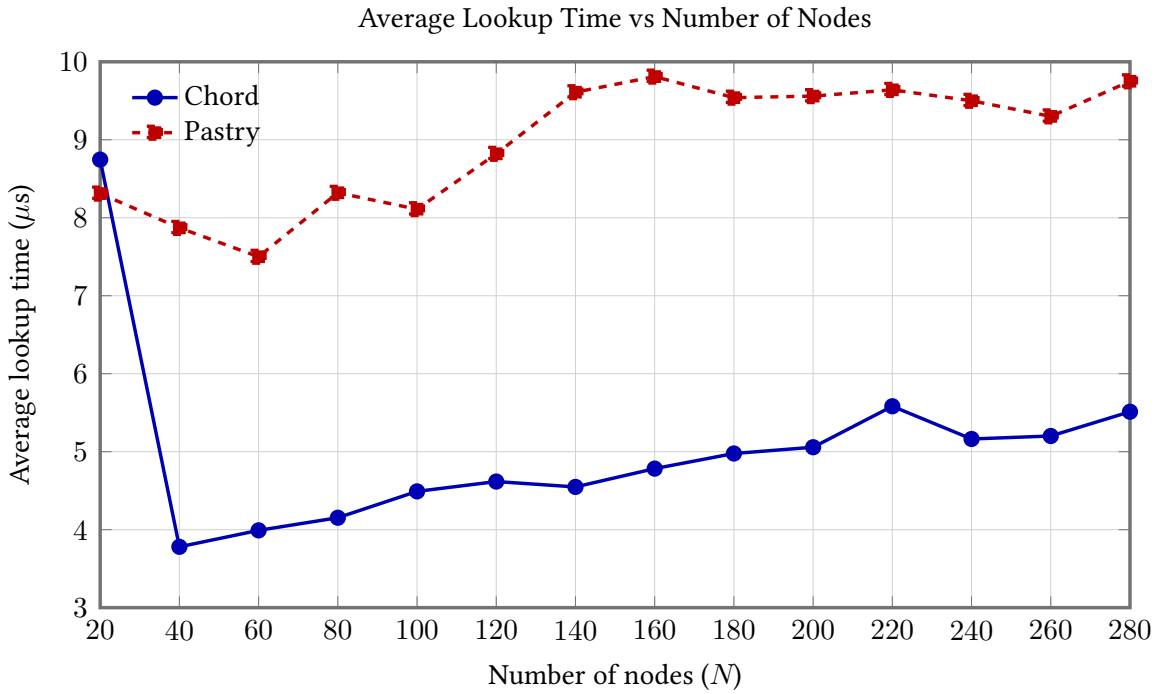
3.7 Συμπεράσματα

Το Pastry αποτελεί ένα από τα θεμελιώδη πρωτόκολλα DHT και έχει επηρεάσει σημαντικά τον σχεδιασμό μεταγενέστερων συστημάτων. Παρότι δεν χρησιμοποιείται αυτούσιο σε όλα τα σύγχρονα συστήματα, οι ιδέες του παραμένουν ιδιαίτερα επίκαιρες και χρήσιμες για την κατανόηση της θεωρίας των κατανεμημένων overlay δικτύων.

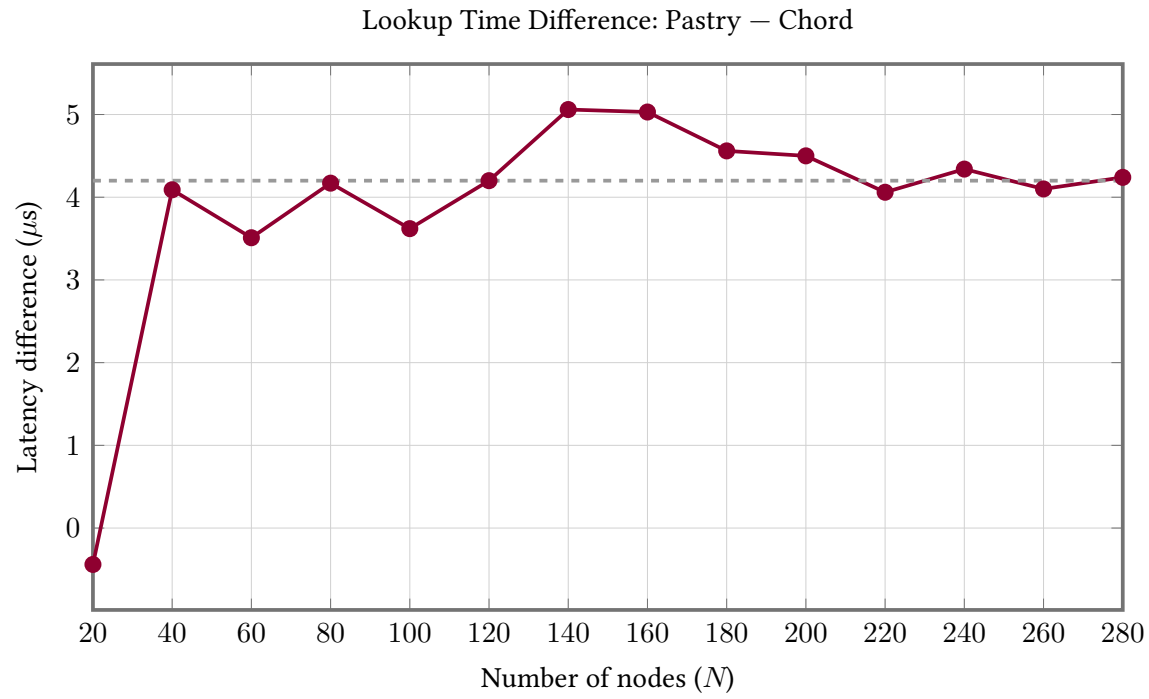
4 Σύγκριση

4.1 Αποτελέσματα σύγκρισης χρόνου

Σε αυτό το σημείο συγκρίνουμε την ταχύτητα των lookups σε σχέση με των αριθμό των κόμβων και για τους 2 αλγορίθμους.



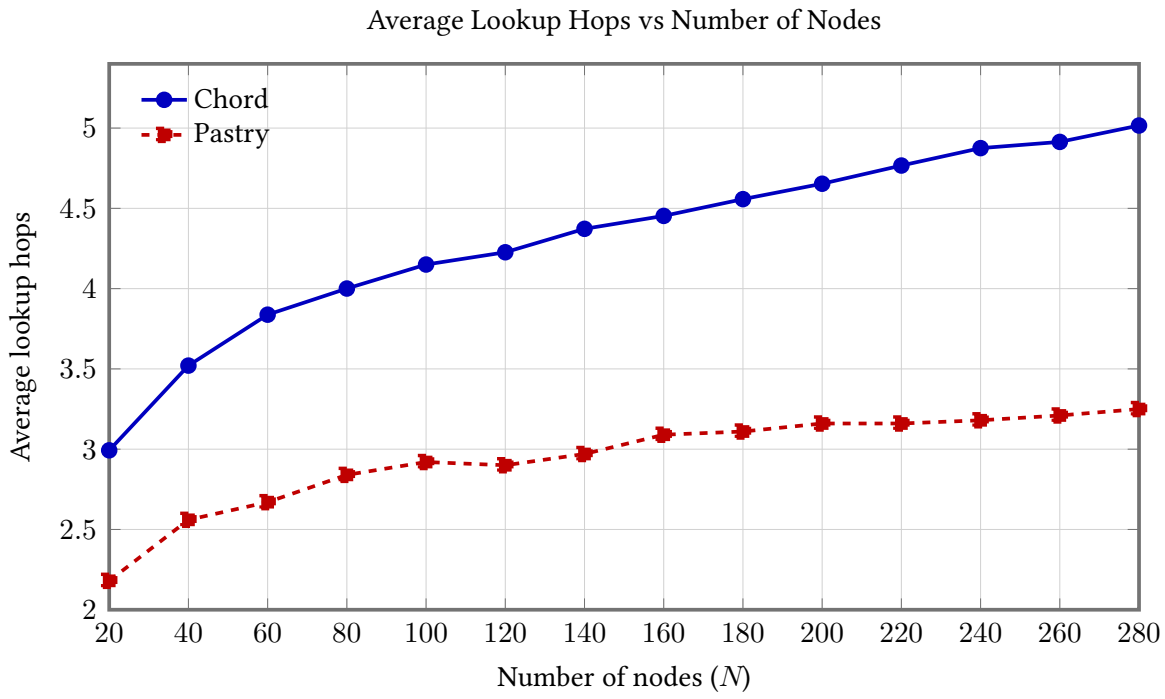
Σχήμα 5: Σύγκριση απόδοσης Chord - Pastry



Σχήμα 6: Διαφορά χρόνων Pastry - Chord

4.2 Αποτελέσματα σύγκρισης hops

Σε αυτό το σημείο συγκρίνουμε τον Μ.Ο. των lookups σε σχέση με των αριθμό των κόμβων και για τους 2 αλγορίθμους.

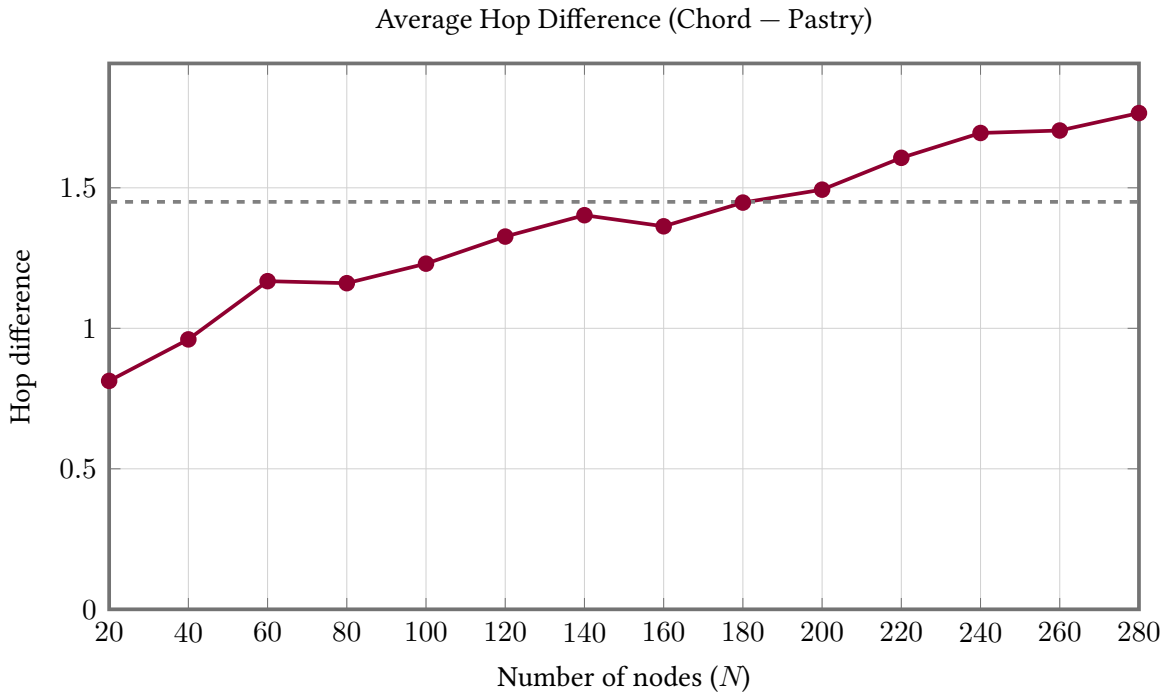


Σχήμα 7: Μ.Ο. Hops για Chord - Pastry

Παρατηρούμε οτι και οι αλγόριθμοι έχουν την μορφή:

$$\text{Chord: } \mathbb{E}[\text{hops}] \approx 0.60 \log_2 N$$

$$\text{Pastry: } \mathbb{E}[\text{hops}] \approx 0.42 \log_2 N$$



Σχήμα 8: Διαφορά hops μεταξύ chord & pastry

Παρατήρηση

Το Chord και το Pastry παρουσιάζουν παρόμοια ασυμπτωτική συμπεριφορά, καθώς και οι δύο αλγόριθμοι επιτυγχάνουν δρομολόγηση σε O(logN) άλματα. Ωστόσο, διαφοροποιούνται ως προς τον συντελεστή του λογαριθμικού όρου, λόγω της διαφορετικής οργάνωσης των δομών δρομολόγησης.

Chord χρησιμοποιεί πίνακες δακτύλων με εκθετικά βήματα στον χώρο αναγνωριστικών, γεγονός που οδηγεί σε απλούστερη διαχείριση κατάστασης αλλά σε ελαφρώς μεγαλύτερο μέσο αριθμό αλμάτων. Αντίθετα, το Pastry εφαρμόζει δρομολόγηση βασισμένη σε κοινά προθέματα, διατηρώντας πλουσιότερη δρομολογιακή πληροφορία ανά κόμβο, η οποία μειώνει τον αριθμό των αλμάτων ανά αναζήτηση.

συνέπεια, η διαφορά στην απόδοση δεν αφορά την κλιμάκωση, αλλά τον σταθερό παράγοντα, ο οποίος είναι μικρότερος στο Pastry λόγω του αυξημένου κόστους σε μνήμη και συντήρηση δρομολογιακής κατάστασης.

5 Links

- GitHub Repo: Distributed Systems
- Latex: Chord & Pastry Technical Report
- Athanasios Tasis: up1093503@upnet.gr
- Stavros Bantzis: up1097449@upnet.gr
- ChatGPT chat: OpenAI
- Claude AI: Anthropic
- Nano-Banana Pro: Image