ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ
UNIVERSITY OF PATRAS

Τεχνολογίες & Αλγόριθμοι Αποκεντρωμένων Δεδομένων

Project_1: Implementation and Experimental Evaluation of Basic DHTs

https://github.com/AggelosVer/Texnologies-Apokentrwmenwn-Dedomenwn

Μέλη Ομάδας

ΒΕΡΥΚΙΟΣ ΑΓΓΕΛΟΣ ΑΜ: 1100500

up1100500@upnet.gr

ΒΟΓΙΑΝΤΖΗΣ ΑΝΑΣΤΑΣΙΟΣ ΑΜ: 1100506

up1100506@upnet.gr

ΚΟΛΥΒΡΑΣ ΚΩΝΣΤΑΝΤΙΝΟΣ ΑΜ: 1103826

up1103826@upnet.gr

Περιεχόμενα

Εισαγωγή και Στόχος

Σε αυτή την εργασία, ο σκοπός μας ήταν να φτιάξουμε και να συγκρίνουμε δύο συστήματα P2P (Peer-to-Peer), το **Chord** και το **Pastry**. Αυτά τα συστήματα ανήκουν στην κατηγορία των **Distributed Hash Tables (DHT)** και μας επιτρέπουν να αποθηκεύουμε πληροφορίες σε ένα δίκτυο κόμβων χωρίς να υπάρχει ένας κεντρικός server που να τα ελέγχει όλα.

Τι θέλουμε να πετύχουμε;

- Να φτιάξουμε τον αλγόριθμο του **Chord** που χρησιμοποιεί έναν δακτύλιο για να οργανώνει τους κόμβους.

- Να φτιάξουμε τον αλγόριθμο του **Pastry** που χρησιμοποιεί προθέματα (prefixes) για να βρίσκει πιο γρήγορα τους κόμβους.

- Να μετρήσουμε πόσα "βήματα" (hops) χρειάζεται το κάθε σύστημα για να βρει μια ταινία.

- Να δούμε πώς συμπεριφέρεται το δίκτυο όταν προσθέτουμε ή αφαιρούμε κόμβους (Join/Leave).

Πώς το υλοποιήσαμε;

Χρησιμοποιήσαμε **Python** και προσομοιώσαμε το δίκτυο με:

Threads για τους κόμβους, ώστε να τρέχουν ταυτόχρονα.

Sockets για να επικοινωνούν μεταξύ τους με μηνύματα.

B+ Trees τοπικά σε κάθε κόμβο, για να μπορούμε να κάνουμε γρήγορη αναζήτηση μέσα στα δεδομένα που κατέχει ο κάθε peer.

Κατακερματισμός και Γενική Λειτουργία DHT

Αναγνωριστικά και Hashing (SHA-1)

Το πρώτο βήμα είναι να δώσουμε μια "ταυτότητα" (ID) τόσο στους κόμβους (τον κάθε υπολογιστή) όσο και στα κλειδιά μας (τους τίτλους των ταινιών). Για να το κάνουμε αυτό ομοιόμορφα, χρησιμοποιήσαμε τη συνάρτηση κατακερματισμού **SHA-1**.

Στον κώδικα μας (αρχείο `dht_hash.py`), η συνάρτηση `hash_key` παίρνει ένα string (π.χ. τον τίτλο "Inception") και βγάζει έναν αριθμό 160-bit.

```python
def hash_key(self, key: str) -> int:
    normalized_key = key.strip().lower()
    key_bytes = normalized_key.encode('utf-8')
    hash_obj = hashlib.sha1(key_bytes)
    hex_digest = hash_obj.hexdigest()
    hash_int = int(hex_digest, 16)
    identifier = hash_int % self.ring_size
    return identifier
```

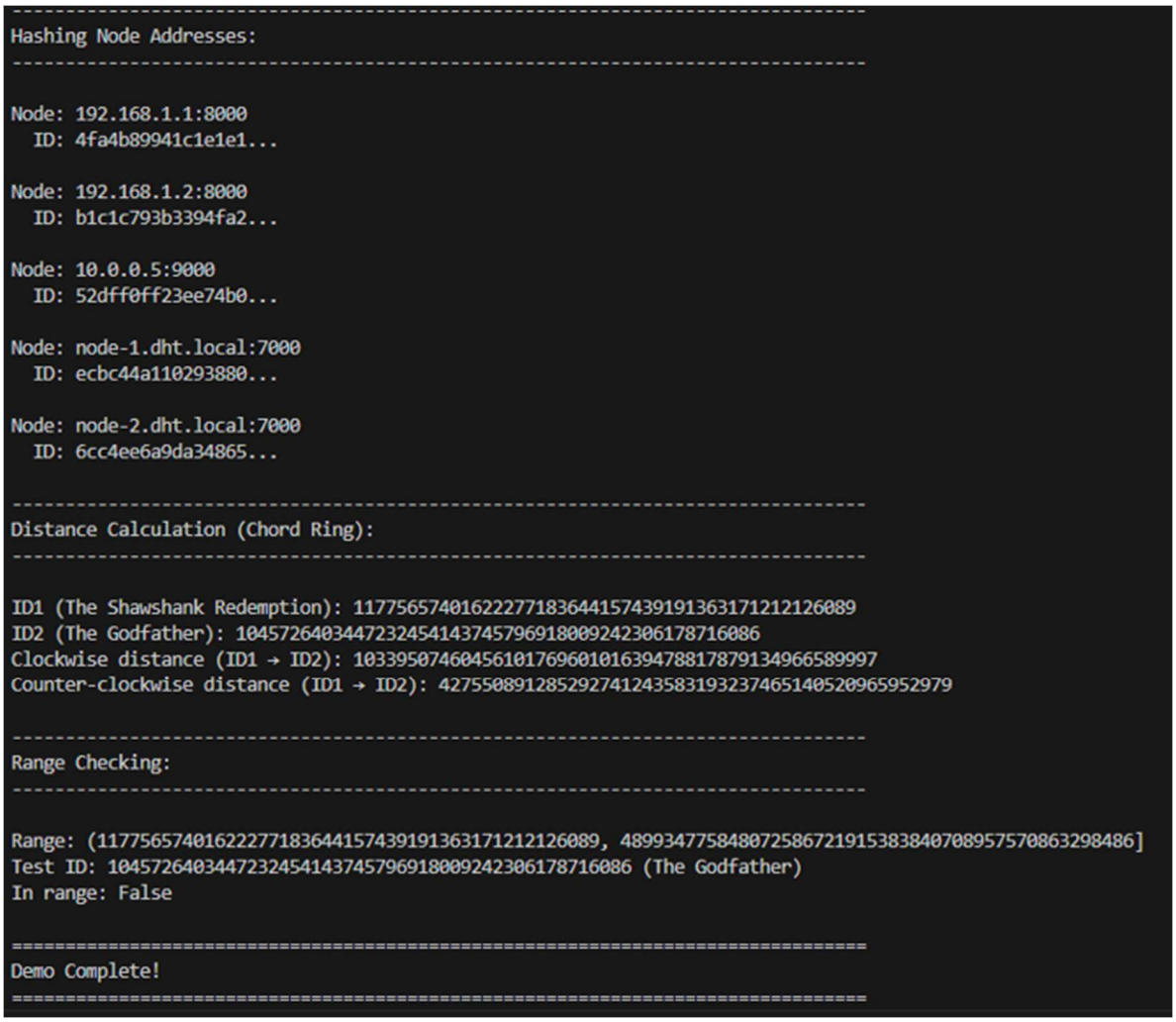
Αυτή η μέθοδος είναι πολύ σημαντική γιατί διασφαλίζει ότι οι ταινίες θα "μοιραστούν" ομοιόμορφα σε όλο το δίκτυο.

Παράδειγμα Hashing Ταινιών

python dht_hash.py -> βλέπουμε πώς μετατρέπονται οι τίτλοι σε IDs

```
Identifier space: 160 bits
Ring size: 2^160 = 1461501637330902918203684832716283019655932542976

----------------------------------------------------------------------
Hashing Movie Titles:
----------------------------------------------------------------------

Title: The Shawshank Redemption
  Decimal ID: 1177565740162227718364415743919136317121212 6089
  Hex ID:     021009eae1cef4159785df5e4fedd0de03cce789
  Pastry Prefix (8 digits): 021009ea

Title: The Godfather
  Decimal ID: 1045726403447232454143745796918009242306178716086
  Hex ID:     b72bfe636513419e9ee53525f5c061cb33c811b6
  Pastry Prefix (8 digits): b72bfe63
```

```
Title: The Godfather
  Decimal ID: 104572640344723245414374579691800924230617871608 6
  Hex ID:     b72bfe636513419e9ee53525f5c061cb33c811b6
  Pastry Prefix (8 digits): b72bfe63

Title: The Dark Knight
  Decimal ID: 489934775848072586721915383840708957570863298486
  Hex ID:     55d16f6733b52149c7f16c5a90b4421338a177b6
  Pastry Prefix (8 digits): 55d16f67

Title: Pulp Fiction
  Decimal ID: 10288696799329532920626357801206017489980028790 50
  Hex ID:     b4381cb13bda96acb6cd6c916e9291f8d748aa4a
  Pastry Prefix (8 digits): b4381cb1

Title: Forrest Gump
  Decimal ID: 589684587161502137061127765580717381925907927853
  Hex ID:     674a5f14703431c3bc739c4eff849f4f77b7132d
  Pastry Prefix (8 digits): 674a5f14

Title: Inception
  Decimal ID: 68446726115916459567805551425376085410908823194 8
  Hex ID:     77e492d109db59d1fe3093815f1ee944242db20c
  Pastry Prefix (8 digits): 77e492d1

Title: The Matrix
  Decimal ID: 207213117316447108432154057522119256765704721660
  Hex ID:     244bc1ee0ded1368169ebb4cc4e8e20ac26c54fc
  Pastry Prefix (8 digits): 244bc1ee

Title: Goodfellas
  Decimal ID: 86918641198416180017720621296344476149923221336 7
  Hex ID:     983faaacbf47c7d81952769baf48a58b232d9977
  Pastry Prefix (8 digits): 983faaac

Title: The Silence of the Lambs
  Decimal ID: 998117092075344586053108672941216815449283498733
  Hex ID:     aed51e4bfbfdcbedf618a7ceaf719679bef012ed
  Pastry Prefix (8 digits): aed51e4b

Title: Interstellar
  Decimal ID: 134587805285153441061867090836195917158306323155 9
  Hex ID:     ebbf42580e83b1c1fc027a0d10de6bcba8a60c47
  Pastry Prefix (8 digits): ebbf4258
```



Ορίζει τις παραμέτρους του DHT, όπως τον χώρο των 160 bits και το συνολικό μέγεθος του δακτυλίου αναγνωριστικών.

Μετατρέπει τίτλους ταινιών σε δεκαδικά και δεκαεξαδικά IDs, παρέχοντας παράλληλα τα αντίστοιχα προθέματα για το πρωτόκολλο Pastry.

Παράγει μοναδικά αναγνωριστικά για διευθύνσεις δικτύου κόμβων, προσομοιώνοντας την τοποθέτησή τους στον κατανεμημένο πίνακα.

Υπολογίζει τις αποστάσεις μεταξύ των αναγνωριστικών στον δακτύλιο Chord, τόσο στη δεξιόστροφη όσο και στην αριστερόστροφη κατεύθυνση.

Πραγματοποιεί ελέγχους εμβέλειας (range checking) για να διαπιστώσει αν ένα κλειδί ανήκει σε συγκεκριμένο διάστημα μεταξύ δύο κόμβων.

Chord: Θεωρία και Υλοποίηση

Η δομή του Δακτυλίου

Ο **Chord** βλέπει το δίκτυο σαν έναν κύκλο (δακτύλιο) από IDs από 0 έως 2^{160}-1. Κάθε κόμβος μπαίνει σε μια θέση στον κύκλο και είναι υπεύθυνος για όλα τα κλειδιά που βρίσκονται μεταξύ αυτού και του προηγούμενου κόμβου του.

Ο Finger Table

Κάθε κόμβος ChordNode διατηρεί μια λίστα από m_bits (160) άλλους κόμβους. Αυτοί οι κόμβοι είναι οι "σταθμοί" που μας επιτρέπουν να πηδάμε στο μισό της απόστασης που απομένει μέχρι το στόχο.

Στον κώδικα (αρχείο `chord_node.py`):

```
self.finger_table: List['ChordNode'] = [self] * self.m_bits
```

Αλγόριθμος Αναζήτησης (find_successor)

Όταν ψάχνουμε μια ταινία, ρωτάμε τον κόμβο: "Ποιος είναι ο επόμενος για αυτό το ID;".

```
def find_successor(self, id: int) -> 'ChordNode':
    if self.hasher.in_range(id, self.id, self.successor.id, inclusive_start=False, inclusive_end=True):
        return self.successor
    else:
        n0 = self.closest_preceding_node(id)
        if n0 is self:
            return self.successor

        return n0.find_successor(id)
```

Συντήρηση και Join

Επειδή οι κόμβοι μπορεί να μπουν ή να βγουν από το δίκτυο, έχουμε τον μηχανισμό **Stabilization** που "φτιάχνει" τον δακτύλιο συνεχώς ρωτώντας τον successor "ποιος είναι ο predecessor σου;".

Pastry: Θεωρία και Υλοποίηση

Δρομολόγηση βάσει Προθέματος (Prefix Routing)

Ο **Pastry** δουλεύει διαφορετικά. Το ID του κάθε κόμβου βλέπεται σαν μια σειρά από ψηφία στο 16-αδικό σύστημα. Η δρομολόγηση γίνεται προσπαθώντας σε κάθε βήμα να βρούμε έναν κόμβο που να έχει μεγαλύτερο κοινό πρόθεμα (prefix) με το κλειδί που ψάχνουμε.

Routing Table και Leaf Set

Ο Pastry χρησιμοποιεί δύο βασικές δομές:

- **Leaf Set**: Οι πιο κοντινοί κόμβοι αριθμητικά (μικρότεροι και μεγαλύτεροι).

- **Routing Table**: Ένας πίνακας με κόμβους που μοιράζονται κοινά προθέματα.

```
self.leaf_smaller: List['PastryNode'] = []
self.leaf_larger: List['PastryNode'] = []
self.neighborhood_set: List['PastryNode'] = []
self.num_rows = self.m_bits // self.b
self.routing_table: Dict[int, Dict[int, 'PastryNode']] = {}
```

Ο αλγόριθμος route

Ο αλγόριθμος ελέγχει πρώτα αν το ID είναι μέσα στο εύρος του **Leaf Set**. Αν όχι, πάει στο **Routing Table**.

```
def route(self, key_id: int, hops: int = 0, visited: Optional[set] = None) -> tuple['PastryNode', int]:
    if visited is None:
        visited = set()
    if self.id in visited or hops > 100:
        return (self, hops)
    visited.add(self.id)
    key_hex = self.hasher.get_hex_id(key_id, digits=self.m_bits//4)
    shared_prefix = self._shared_prefix_length(key_hex)
    if self.is_in_leaf_set_range(key_id):
        closest = self.find_closest_in_leaf_set(key_id)
        if closest is self:
            return (self, hops)
```

```
        if closest.id not in visited:
            return closest.route(key_id, hops + 1, visited)
        return (self, hops)
    if shared_prefix < self.num_rows:
        next_digit = int(key_hex[shared_prefix], 16)
        if shared_prefix in self.routing_table and next_digit in self.routing_table[shared_prefix]:
            next_node = self.routing_table[shared_prefix][next_digit]
            if next_node.id not in visited:
                return next_node.route(key_id, hops + 1, visited)
        for digit in range(self.base):
            if shared_prefix in self.routing_table and digit in self.routing_table[shared_prefix]:
                candidate = self.routing_table[shared_prefix][digit]
                if candidate.id not in visited:
                    candidate_key_prefix = self._shared_prefix_length_between(candidate.hex_id, key_hex)
                    if candidate_key_prefix > shared_prefix:
                        return candidate.route(key_id, hops + 1, visited)
    for row_idx in range(shared_prefix + 1, self.num_rows):
        if row_idx in self.routing_table:
            for digit, node in self.routing_table[row_idx].items():
                if node.id not in visited:
                    node_key_prefix = self._shared_prefix_length_between(node.hex_id, key_hex)
                    if node_key_prefix > shared_prefix:
                        return node.route(key_id, hops + 1, visited)
    closest_leaf = self.find_closest_in_leaf_set(key_id)
    if closest_leaf is not self and closest_leaf.id not in visited:
        return closest_leaf.route(key_id, hops + 1, visited)
    return (self, hops)
```

Είσοδος στο Δίκτυο (Join)

Όταν ένας κόμβος μπαίνει στο δίκτυο του Pastry, "ανακαλύπτει" όλους τους κόμβους που βρίσκονται στη διαδρομή προς το ID του και ενημερώνει τους δικούς του πίνακες αλλά και τους πίνακες των άλλων.

```
def join(self, introducer: Optional['PastryNode'] = None):
    for node in all_nodes:
        self.add_node(node)
        node.add_node(self)
```

Τοπικό indexing με B+ Trees

Αφού το δίκτυο (Chord ή Pastry) βρει ποιος κόμβος είναι υπεύθυνος για μια ταινία, ο κόμβος αυτός πρέπει να ψάξει γρήγορα στα δικά του δεδομένα. Όπως ζητάει η εκφώνηση, κάθε peer είναι εξοπλισμένος με ένα **B+ Tree** για να διαχειρίζεται τοπικά τις ταινίες του.

Γιατί B+ Tree;

Επιλέξαμε το B+ Tree γιατί προσφέρει:

Ταχύτητα: Αναζήτηση, εισαγωγή και διαγραφή σε χρόνο O(\log n).

Range Queries: Είναι ιδανικό για να βρίσκουμε εύκολα ένα εύρος τιμών (π.χ. ταινίες με βαθμολογία από 8 έως 10), κάτι που μια απλή Hash Table δεν μπορεί να κάνει αποδοτικά.

Υλοποίηση στον Κόμβο

Στον κώδικα μας (αρχείο `bplus_tree.py`), το δέντρο αρχικοποιείται με μια συγκεκριμένη τάξη (order=10).

```
self.data = BPlusTree(order=10)
```

Range Queries και Φιλτράρισμα

```
def range_query(self, start_key, end_key) -> List[Any]:
    results = []
    node = self._find_leaf(start_key)

    while node:
        for i, key in enumerate(node.keys):
            if start_key <= key <= end_key:
                results.append(node.values[i])
            elif key > end_key:
                return results
        node = node.next_leaf
    return results
```

Με αυτή τη δομή, ο κάθε κόμβος δεν είναι απλά μια αποθήκη, αλλά ένα μικρό "database engine" που μπορεί να απαντήσει σε ερωτήσεις όπως: *"Δώσε μου τις ταινίες που έχεις με popularity πάνω από 50"*.

Δικτυακή Επικοινωνία (Sockets & JSON)

Για να λειτουργήσει το DHT ως κατανεμημένο σύστημα, έπρεπε οι κόμβοι να μπορούν να μιλάνε μεταξύ τους. Χρησιμοποιήσαμε **TCP Sockets** και ένα πρωτόκολλο ανταλλαγής μηνυμάτων σε μορφή **JSON**.

Δομή Μηνύματος (Message Protocol)

Κάθε επικοινωνία στο δίκτυο γίνεται μέσω ενός αντικειμένου Message. Στον κώδικα μας (αρχείο message_protocol.py), ορίσαμε διάφορους τύπους μηνυμάτων όπως FIND_SUCCESSOR, NOTIFY, LOOKUP, κλπ.

Ένα τυπικό μήνυμα περιέχει:

msg_type: Τι θέλουμε να κάνουμε.

sender_address: Ποιος στέλνει το μήνυμα.

payload: Τα δεδομένα του αιτήματος (π.χ. το key_id που ψάχνουμε).

```
class Message:
    def to_json(self) -> str:
        return json.dumps(self.to_dict())

    @classmethod
    def from_bytes(cls, data: bytes) -> 'Message':
        length = int.from_bytes(data[:4], byteorder='big')
        json_str = data[4:4+length].decode('utf-8')
        return cls.from_json(json_str)
```

Ο Network Client/Server

Ο κάθε κόμβος τρέχει έναν Server που "ακούει" σε μια θύρα (port) και έναν Client που στέλνει αιτήματα σε άλλους κόμβους. Όταν ένας κόμβος θέλει να ρωτήσει κάτι, ανοίγει μια σύνδεση, στέλνει το JSON μήνυμα και περιμένει την απάντηση.

Στο αρχείο `network_node_tcp.py`, βλέπουμε πώς γίνεται η διαχείριση των αιτημάτων:

```
def _handle_connection(self, conn, addr):
    data = conn.recv(self.buffer_size)
```

```
    message = Message.from_bytes(data)

    result = self.process_message(message)

    response = create_response(message, result)
    conn.sendall(response.to_bytes())
```

Αυτό το σύστημα μας επιτρέπει να "τρέχουμε" το DHT σε πολλούς διαφορετικούς υπολογιστές, αρκεί να ξέρουν τις IP διευθύνσεις και τα Ports των άλλων κόμβων.

Αξιολόγηση Απόδοσης Δικτυακού Επιπέδου

Με το benchmark_tcp_network.py πραγματοποιείται μια προσομοίωση του δικτύου DHT χρησιμοποιώντας πραγματικές συνδέσεις TCP αντί για απλές κλήσεις συναρτήσεων στη μνήμη. Συγκεκριμένα, αρχικοποιεί πολλαπλούς ανεξάρτητους κόμβους σε διαφορετικές θύρες (ports 15000-16003) και μετρά τον πραγματικό χρόνο που απαιτείται για την ανταλλαγή μηνυμάτων, τον συγχρονισμό και τη διαχείριση δεδομένων (Insert/Lookup). Μέσω αυτής της διαδικασίας αξιολογούμε τη συμπεριφορά του συστήματος σε συνθήκες πραγματικού δικτύου, καταγράφοντας τον συνολικό αριθμό των δικτυακών μηνυμάτων, τη μέση καθυστέρηση (latency) του πρωτοκόλλου και τη συνολική διακίνηση δεδομένων (throughput) σε λειτουργίες ανά δευτερόλεπτο. Με αυτόν τον τρόπο, αποδεικνύεται η ικανότητα των υλοποιήσεων Chord και Pastry να λειτουργούν αξιόπιστα πάνω από ένα πραγματικό δικτυακό επίπεδο.

```
======================================================================
BENCHMARKING CHORD (TCP/Network Context)
======================================================================
[*] Starting 4 nodes on ports 15000-15003...
[*] Forming DHT network (using network addresses)...
[*] Waiting for network stabilization...
[*] Executing 20 operations via TCP requests...
[*] Collecting and aggregating network metrics...

[+] Results for chord:
    - Network Messages: 147
    - Avg TCP Latency:  2.39 ms
    - Avg Op Latency:   0.99 ms
    - Total Throughput: 15.91 ops/sec

======================================================================
BENCHMARKING PASTRY (TCP/Network Context)
======================================================================
[*] Starting 4 nodes on ports 16000-16003...
[*] Forming DHT network (using network addresses)...
[*] Waiting for network stabilization...
[*] Executing 20 operations via TCP requests...
[*] Collecting and aggregating network metrics...

[+] Results for pastry:
    - Network Messages: 993
    - Avg TCP Latency:  22.35 ms
    - Avg Op Latency:   19.09 ms
    - Total Throughput: 86.70 ops/sec
```

Παράλληλα Ερωτήματα για Κ Ταινίες

Ένα από τα ειδικά ζητήματα της εργασίας ήταν η δυνατότητα να βρίσκουμε ταυτόχρονα τη δημοτικότητα (popularity) για **Κ διαφορετικές ταινίες** (π.χ. K=10), με τα ονόματα των ταινιών να δίνονται από τον χρήστη.

Γιατί παράλληλα;

Αν κάναμε τις αναζητήσεις μία-μία (σειριακά), η συνολική καθυστέρηση θα ήταν το άθροισμα των χρόνων κάθε αναζήτησης. Χρησιμοποιώντας **Multi-threading**, μπορούμε να στέλνουμε όλα τα αιτήματα ταυτόχρονα στο δίκτυο και να περιμένουμε τις απαντήσεις παράλληλα.

Υλοποίηση με ThreadPoolExecutor

Για να το πετύχουμε αυτό, φτιάξαμε την κλάση ParallelLookupExecutor (αρχείο parallel_lookup_executor.py). Αυτή χρησιμοποιεί ένα "pool" από threads για να διαχειριστεί τις αναζητήσεις.

```
def lookup_popularity(self, titles: List[str]):
    results = {}
    with ThreadPoolExecutor(max_workers=self.max_workers) as executor:
        future_map = {executor.submit(self._lookup_single, title):
title for title in titles}

        for future in as_completed(future_map):
            title, popularity, elapsed, error = future.result()
            results[title] = popularity
    return results
```

Πλεονεκτήματα

Μειώνεται ο χρόνος αναμονής: Ο συνολικός χρόνος είναι σχεδόν ίσος με τον χρόνο της πιο αργής μεμονωμένης αναζήτησης, αντί για το άθροισμα όλων.

Καλύτερη αξιοποίηση του δικτύου: Το δίκτυο δουλεύει παράλληλα, στέλνοντας μηνύματα σε διαφορετικούς κόμβους την ίδια στιγμή.

Αυτή η λειτουργία είναι πολύ χρήσιμη για εφαρμογές όπως η πρόβλεψη τάσεων ή η δημιουργία λιστών με "trending" ταινίες, όπου χρειαζόμαστε μαζικά δεδομένα γρήγορα.

Δεδομένα και Αντοχή σε Σφάλματα (Replication)

Για να δοκιμάσουμε το σύστημα σε ρεαλιστικές συνθήκες, χρησιμοποιήσαμε ένα σύνολο δεδομένων με ταινίες και υλοποιήσαμε μηχανισμούς για να μην χάνονται τα δεδομένα αν κάποιος κόμβος "πέσει".

Το Dataset (TMDB Movies)

Φορτώσαμε δεδομένα από το **TMDB dataset**, το οποίο περιλαμβάνει τίτλους, βαθμολογίες και δημοτικότητα.

Κλειδί (Key): Ο τίτλος της ταινίας (αφού περάσει από SHA-1).

Τιμή (Value): Ένα λεξικό (dictionary) με όλα τα στοιχεία της ταινίας.

Μηχανισμός Replication

Σε ένα πραγματικό δίκτυο, οι κόμβοι μπορεί να αποσυνδεθούν ξαφνικά. Αν μια ταινία ήταν αποθηκευμένη μόνο σε έναν κόμβο, θα χανόταν. Για να το αποφύγουμε αυτό, υλοποιήσαμε το **Replication**: κάθε ταινία αποθηκεύεται στον "υπεύθυνο" κόμβο αλλά και στους **3 επόμενους γείτονές του**.

... στον Chord

Χρησιμοποιούμε τη **Successor List**. Κάθε κόμβος στέλνει αντίγραφα των δεδομένων του στους επόμενους κόμβους του δακτυλίου.

```
def replicate_data(self):
    for successor in self.successor_list:
        for key, value in self.data.items():
            successor.replicas[key] = value
```

... στον Pastry

Χρησιμοποιούμε το **Leaf Set**. Τα αντίγραφα αποθηκεύονται στους αριθμητικά κοντινότερους κόμβους.

```python
def replicate_data(self):
    leaf_set = self.get_leaf_set()
    for node in leaf_set:
        for key, value in self.data.items():
            node.replicas[key] = value
```

Ανάκτηση Δεδομένων (Recovery)

Αν ένας κόμβος καταλάβει ότι ο προκάτοχός του (predecessor) έφυγε, μπορεί να ανακτήσει τα δεδομένα από τα δικά του αντίγραφα (`replicas`) και να γίνει αυτός ο νέος υπεύθυνος, διασφαλίζοντας ότι το δίκτυο παραμένει πλήρες.

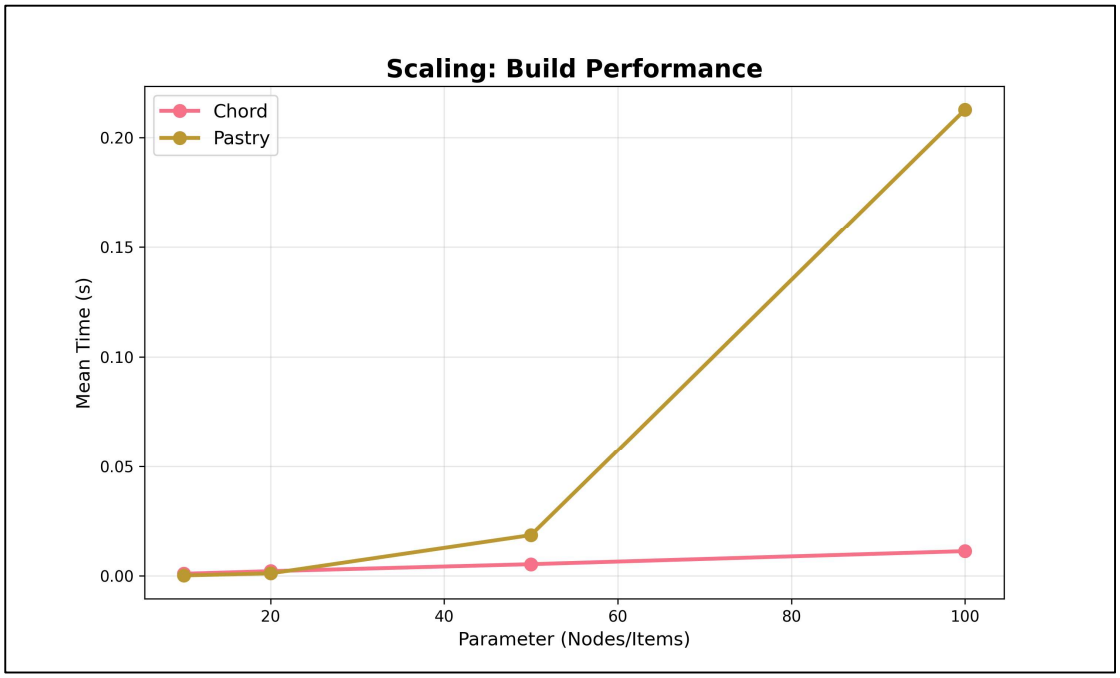
Αυτή η προσέγγιση εγγυάται ότι ακόμα και αν "χάσουμε" μερικούς κόμβους ταυτόχρονα, οι ταινίες μας θα είναι ακόμα διαθέσιμες κάπου αλλού στο δίκτυο.

Πειράματα και Αποτελέσματα

Στο τελευταίο στάδιο, τρέξαμε μια σειρά από πειράματα για να δούμε πώς αποδίδουν τα δύο συστήματα στην πράξη. Χρησιμοποιήσαμε το movie dataset και μετρήσαμε τον αριθμό των **hops** (βημάτων) για χιλιάδες αναζητήσεις.

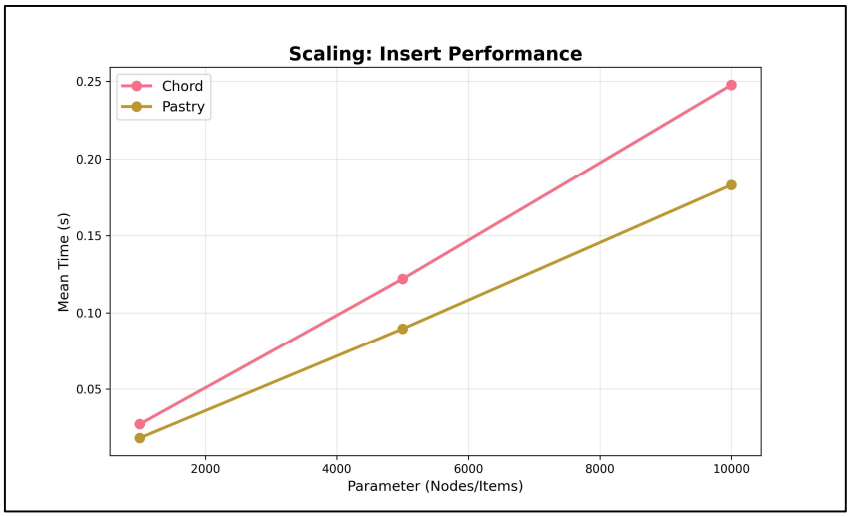
Scaling: Build Performance

Το διάγραμμα αυτό συγκρίνει τον μέσο χρόνο που απαιτείται για την κατασκευή του δικτύου (Build Performance) μεταξύ των πρωτοκόλλων Chord και Pastry καθώς αυξάνεται ο αριθμός των κόμβων και των αντικειμένων. Παρατηρούμε ότι ενώ το Pastry είναι ταχύτερο για μικρό πλήθος δεδομένων, ο χρόνος του αυξάνεται εκθετικά μετά τους 20 κόμβους, καθιστώντας το Chord πολύ πιο σταθερό και αποδοτικό στις μεγάλες κλίμακες.



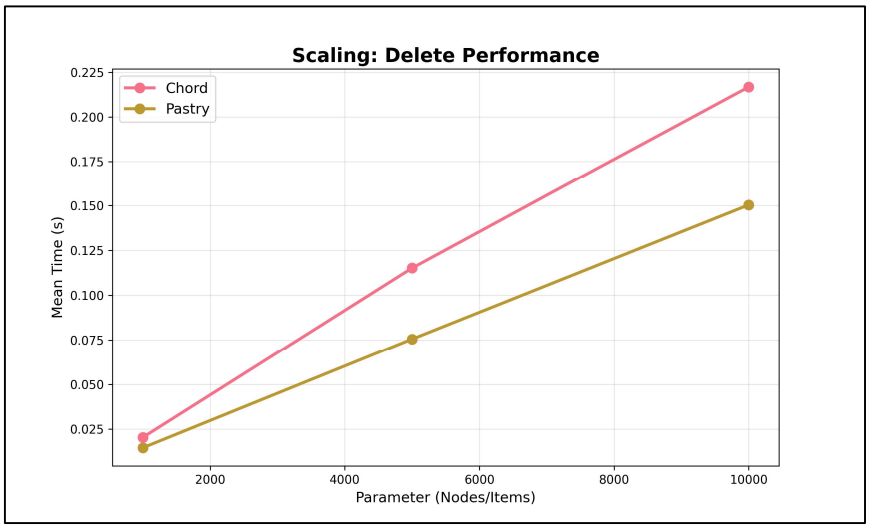
Scaling: Insert Performance

Το διάγραμμα δείχνει τον μέσο χρόνο που χρειάζεται για την εισαγωγή νέων δεδομένων στα πρωτόκολλα Chord και Pastry καθώς αυξάνεται ο όγκος τους. Το Pastry εμφανίζεται σταθερά πιο γρήγορο από το Chord στην εισαγωγή στοιχείων, διατηρώντας χαμηλότερο μέσο χρόνο καθώς το φορτίο αυξάνεται από τις 100 στις 10000 εγγραφές.



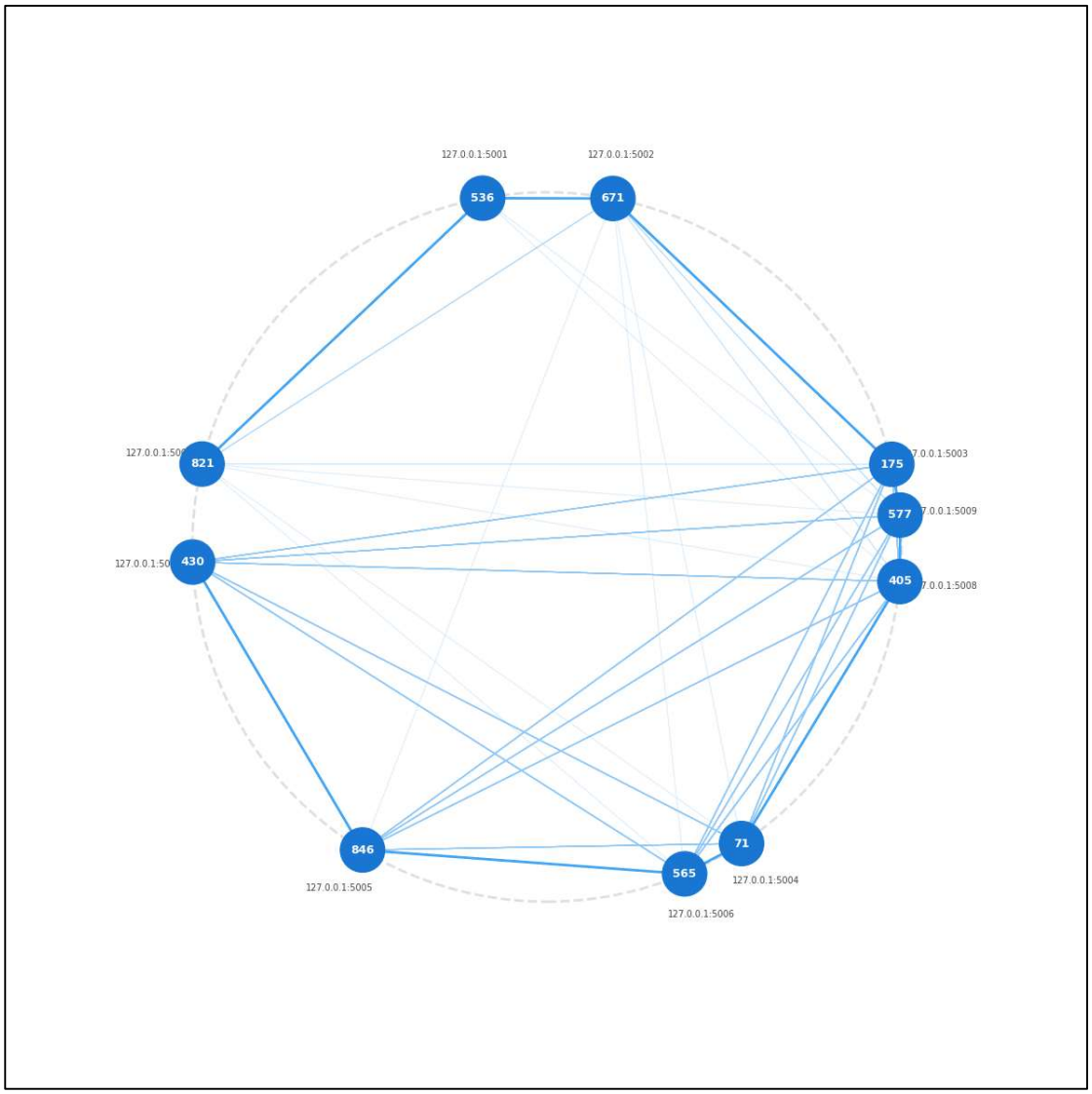
Scaling: Delete Performance

Το διάγραμμα συγκρίνει τον μέσο χρόνο που απαιτείται για τη διαγραφή δεδομένων μεταξύ των πρωτοκόλλων Chord και Pastry καθώς αυξάνεται ο αριθμός των στοιχείων. Το Pastry είναι σταθερά ταχύτερο από το Chord, διατηρώντας χαμηλότερο χρόνο επεξεργασίας σε όλο το εύρος της κλιμάκωσης από τις 100 έως τις 1000 διαγραφές.



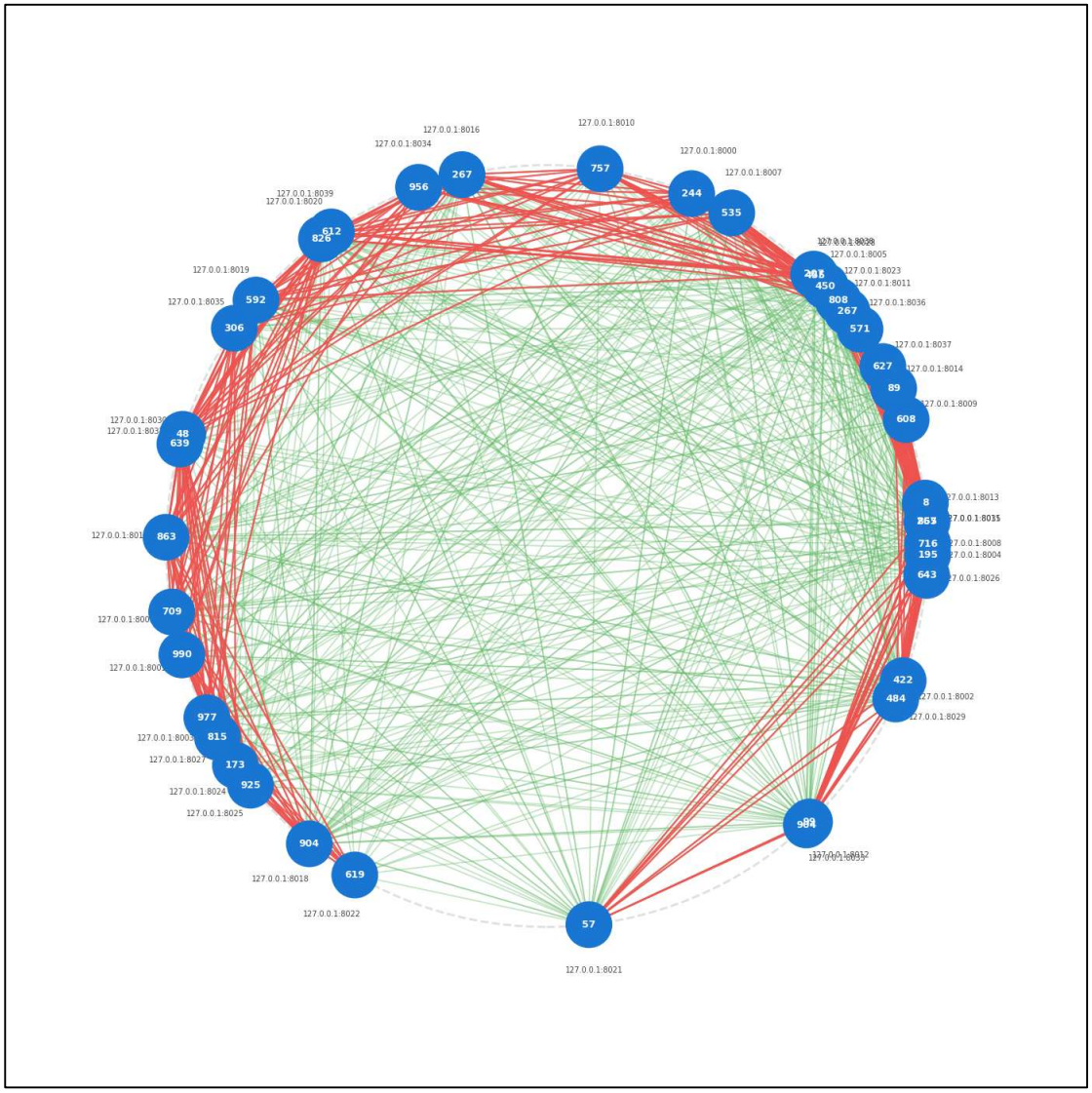
Chord Network Topology

Η εικόνα απεικονίζει τη δομή ενός δικτύου Chord σε μορφή δακτυλίου, όπου οι κόμβοι είναι τοποθετημένοι κυκλικά βάσει του αναγνωριστικού τους (ID). Οι μπλε γραμμές αναπαριστούν τις συνδέσεις του πίνακα δακτύλων (finger table) ενός συγκεκριμένου κόμβου, δείχνοντας πώς αυτός συνδέεται με απομακρυσμένους κόμβους για να επιταχύνει τη δρομολόγηση των αναζητήσεων στο δίκτυο.



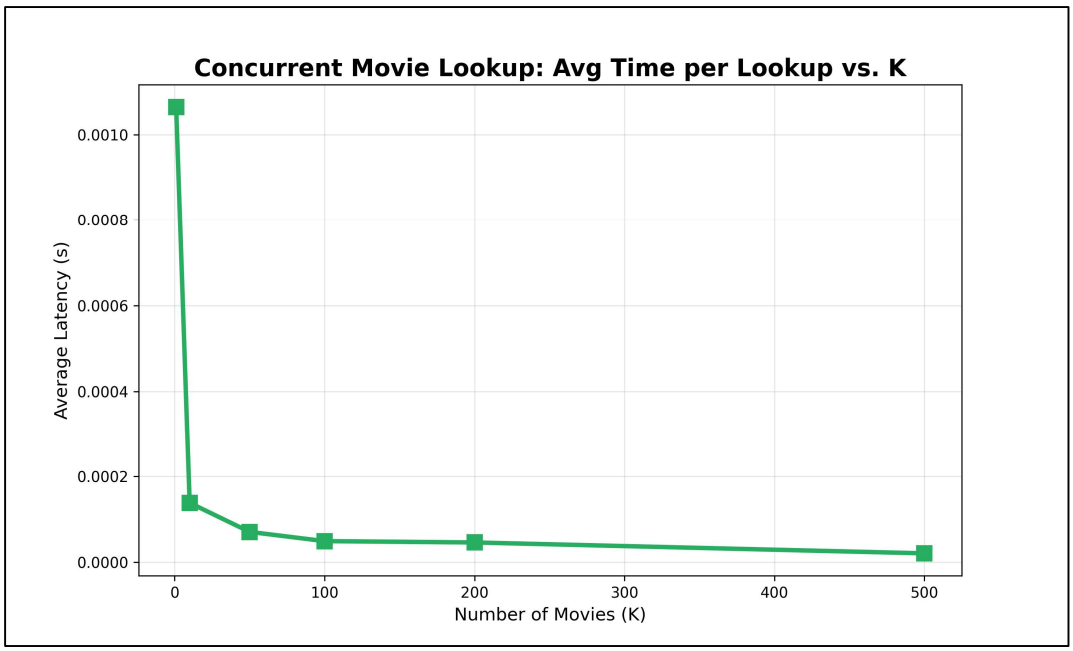
Pastry Network Topology

Η εικόνα απεικονίζει την τοπολογία ενός δικτύου Pastry DHT αποτελούμενου από 40 κόμβους διατεταγμένους στον δακτύλιο αναγνωριστικών, όπου οι κόκκινες γραμμές αναπαριστούν το Leaf Set συνδέοντας κάθε κόμβο με τους πλησιέστερους αριθμητικά γείτονές του. Παράλληλα, οι πράσινες γραμμές δείχνουν τις συνδέσεις του Routing Table που επιτρέπουν τη γρήγορη δρομολόγηση σε μακρινές περιοχές του δικτύου μέσω μεγάλων αλμάτων. Το πλούσιο αυτό πλέγμα συνδέσεων φανερώνει τον τρόπο με τον οποίο το πρωτόκολλο επιτυγχάνει αποδοτική αναζήτηση δεδομένων με ελάχιστα ενδιάμεσα βήματα. Η συγκεκριμένη οπτικοποίηση επιβεβαιώνει τη σύνθετη δομή του Pastry, η οποία συνδυάζει την τοπική γνώση της γειτονιάς με τη δρομολόγηση βάσει κοινών προθεμάτων. Με αυτόν τον τρόπο διασφαλίζεται η ισορροπία του φορτίου και η εξαιρετική κλιμακωσιμότητα του συστήματος ακόμη και σε μεγαλύτερα δίκτυα.



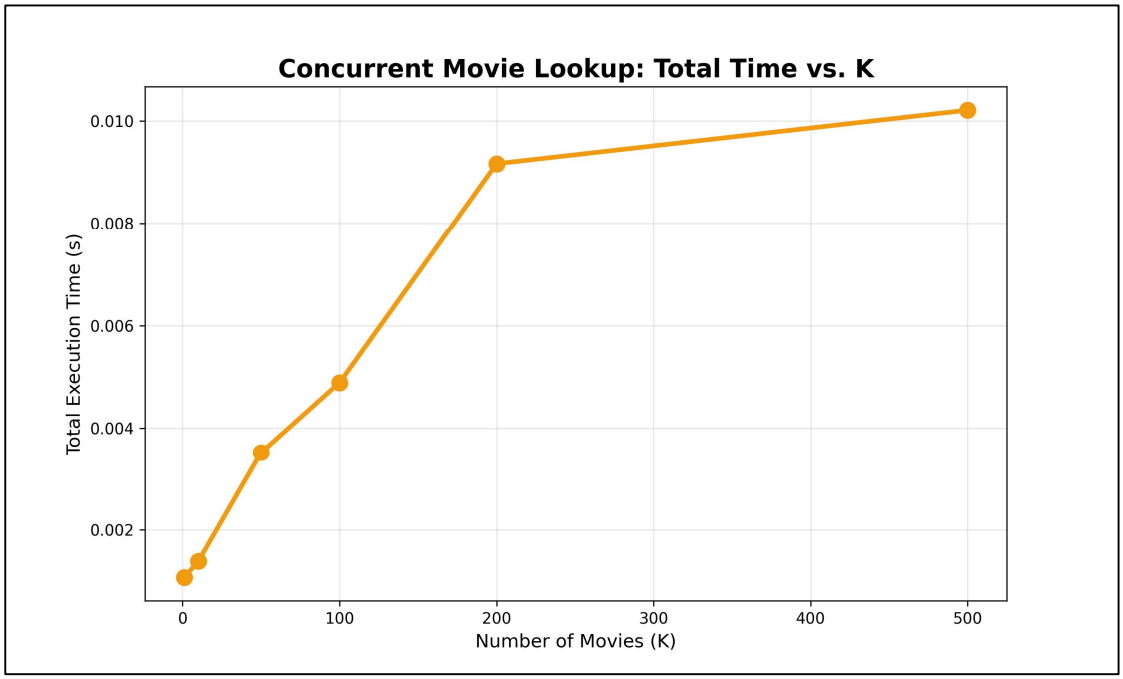
Chord Concurrent Movie Lookup: Avg Time per Lookup vs. K

Το διάγραμμα απεικονίζει τη μέση καθυστέρηση ανά αναζήτηση για ταυτόχρονες ανακτήσεις ταινιών σε σχέση με το πλήθος τους (K). Παρατηρούμε μια απότομη μείωση του μέσου χρόνου καθώς αυξάνεται ο αριθμός των ταινιών, γεγονός που αποδεικνύει τη βελτίωση της αποδοτικότητας του συστήματος μέσω του παραλληλισμού. Μετά τις 100 ταινίες, η καθυστέρηση σταθεροποιείται σε εξαιρετικά χαμηλά επίπεδα, υποδηλώνοντας τη βέλτιστη αξιοποίηση των πόρων του δικτύου.



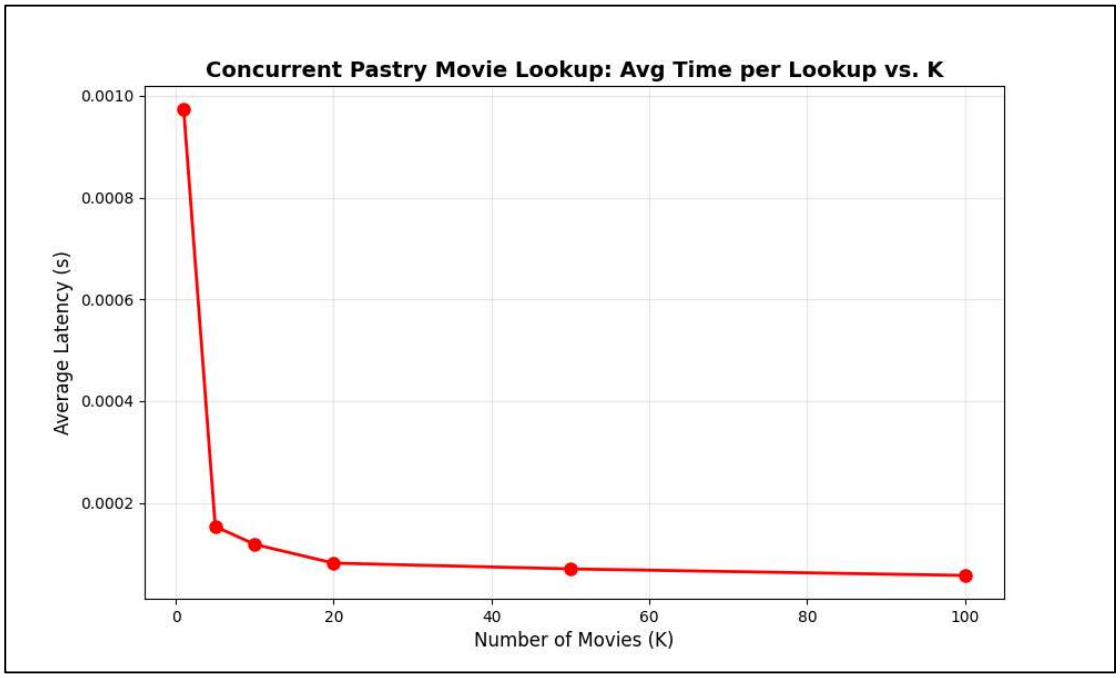
Chord Concurrent Movie Lookup: Avg Time per Lookup vs. K

Το διάγραμμα αυτό δείχνει τη μέση καθυστέρηση (latency) ανά αναζήτηση ταινίας σε συνθήκες ταυτόχρονης εκτέλεσης πολλαπλών ερωτημάτων. Παρατηρούμε ότι ο μέσος χρόνος ανά αναζήτηση μειώνεται δραστικά καθώς αυξάνεται ο αριθμός των ταινιών (K), γεγονός που υποδηλώνει ότι το σύστημα διαχειρίζεται πολύ πιο αποδοτικά τον φόρτο μέσω παραλληλισμού, σταθεροποιώντας την απόδοσή του σε πολύ χαμηλά επίπεδα μετά τις 100 ταινίες.



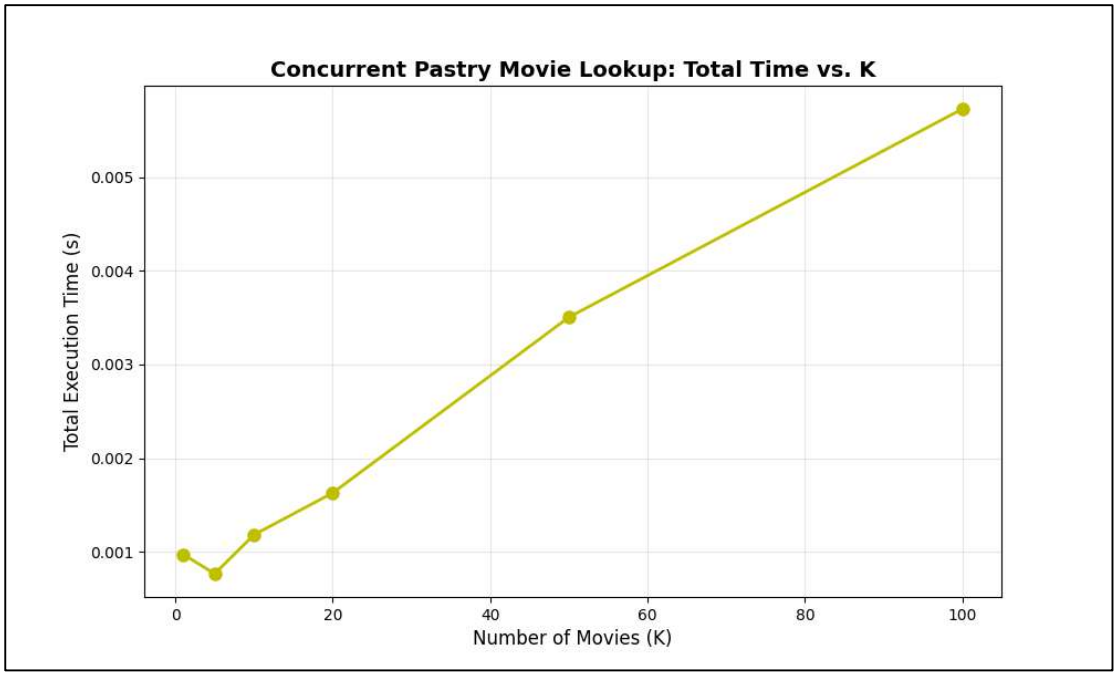
Pastry Concurrent Movie Lookup: Avg Time per Lookup vs. K

Το διάγραμμα απεικονίζει τη δραματική μείωση του μέσου χρόνου απόκρισης ανά αναζήτηση καθώς αυξάνεται το πλήθος των ταυτόχρονων ερωτημάτων (K) στο δίκτυο Pastry. Ξεκινώντας από υψηλότερη καθυστέρηση για μεμονωμένα αιτήματα, η απόδοση βελτιώνεται ραγδαία λόγω της αποτελεσματικής διαχείρισης πόρων μέσω της παραλληλίας. Η καμπύλη σταθεροποιείται γρήγορα σε πολύ χαμηλά επίπεδα (~0.1ms) μετά τις 20 ταυτόχρονες αναζητήσεις, υποδεικνύοντας ότι το σύστημα φτάνει στη βέλτιστη λειτουργία του. Αυτή η συμπεριφορά επιβεβαιώνει την ικανότητα του Pastry να κλιμακώνεται ομαλά και να εξυπηρετεί μεγάλο όγκο κίνησης χωρίς να καταρρέει.



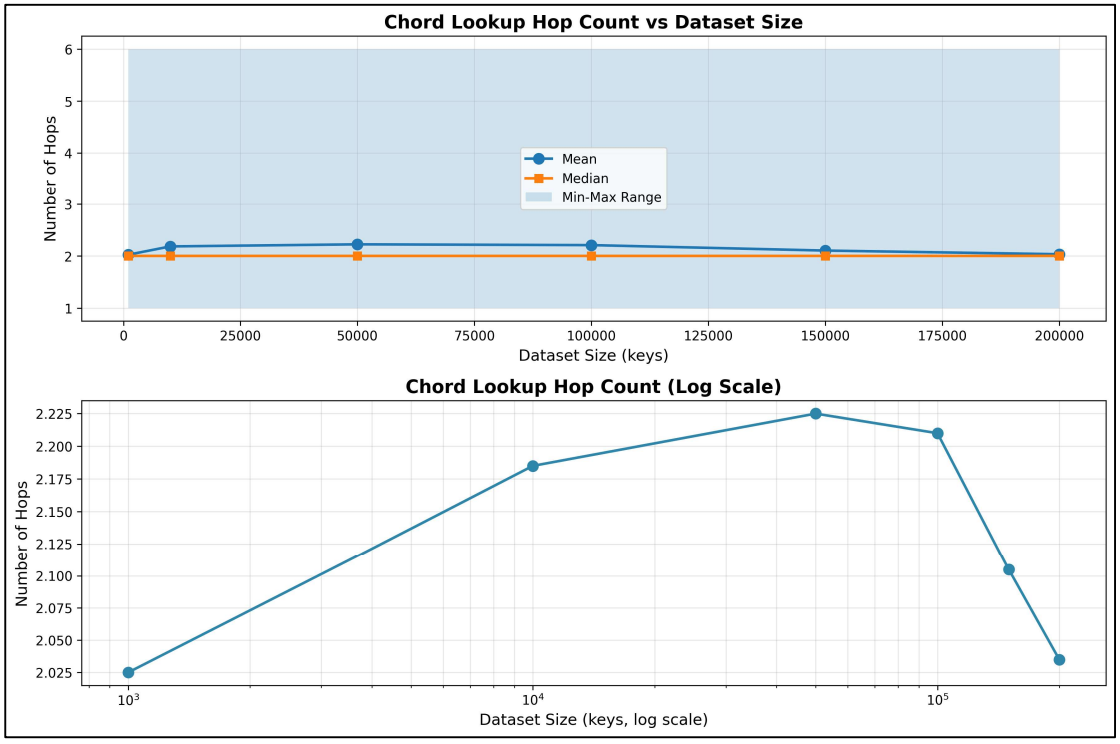
Pastry Concurrent Movie Lookup: Avg Time per Lookup vs. K

Το παραπάνω διάγραμμα απεικονίζει τη γραμμική σχέση μεταξύ του συνολικού χρόνου που απαιτείται για την ολοκλήρωση των ερωτημάτων και του αριθμού των ταινιών (K) που αναζητούνται ταυτόχρονα. Παρατηρούμε ότι ο χρόνος αυξάνεται σταθερά όσο μεγαλώνει το K, κάτι που είναι αναμενόμενο καθώς περισσότερα αιτήματα απαιτούν περισσότερους πόρους επεξεργασίας. Ωστόσο, η απόλυτη τιμή του χρόνου παραμένει εξαιρετικά χαμηλή (κλάσματα του δευτερολέπτου ακόμα και για 100 ταινίες), αποδεικνύοντας την υψηλή απόδοση του συστήματος.



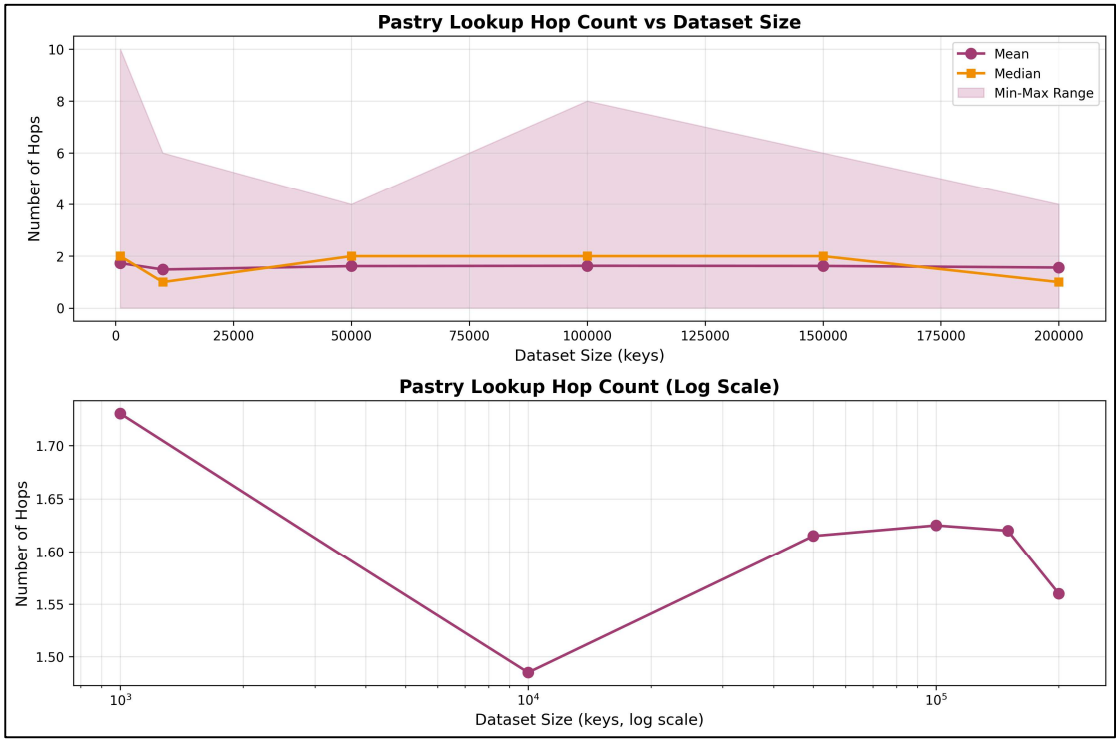
Ανάλυση Αποδοτικότητας Δρομολόγησης Chord DHT

Τα διαγράμματα παρουσιάζουν την επίδοση του αλγορίθμου Chord ως προς τους μεσολαβητές κόμβους (hops) για την εύρεση κλειδιών σε datasets διαφορετικού μεγέθους. Στο πρώτο γράφημα (γραμμική κλίμακα), ο μέσος αριθμός hops (μπλε γραμμή) παραμένει εξαιρετικά σταθερός γύρω στα 2 hops, με σχεδόν αμετάβλητη διάμεσο, αποδεικνύοντας την πολύ καλή κατανομή φορτίου. Η περιοχή Min-Max δείχνει το αναμενόμενο εύρος διακύμανσης. Στο δεύτερο γράφημα (λογαριθμική κλίμακα), παρατηρούμε μια μικρή αύξηση μέχρι τα 150.000 κλειδιά, ακολουθούμενη από μια ενδιαφέρουσα μείωση, που πιθανόν οφείλεται στην καλύτερη πλήρωση του δακτυλίου και την πιο αποτελεσματική χρήση των finger tables. Γενικά, το Chord επιδεικνύει σταθερότητα και προβλεψιμότητα, διαχειριζόμενο αποτελεσματικά την αύξηση των δεδομένων χωρίς σημαντική επιβάρυνση στην απόδοση αναζήτησης..



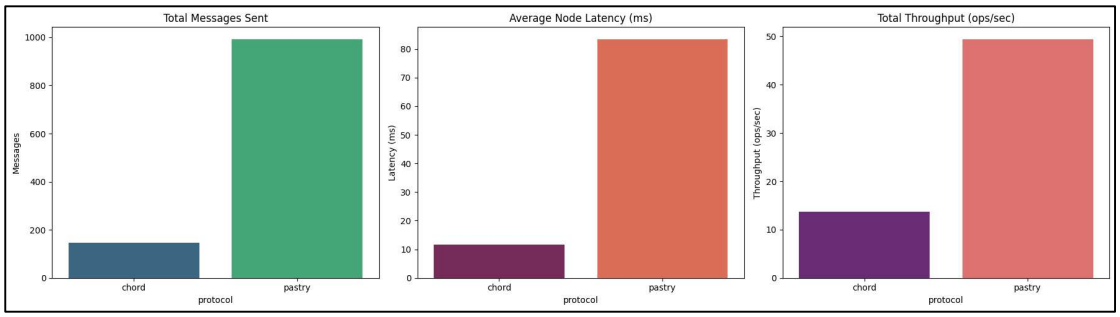
Ανάλυση Αποδοτικότητας Δρομολόγησης Pastry DHT

Τα διαγράμματα απεικονίζουν την υψηλή αποδοτικότητα του πρωτοκόλλου Pastry ως προς τον αριθμό των ενδιάμεσων κόμβων (hops) που απαιτούνται για την εύρεση δεδομένων. Το πρώτο γράφημα δείχνει ότι ο μέσος όρος των hops (purple line) παραμένει εντυπωσιακά σταθερός και χαμηλός (~1.5-1.6 hops) ακόμα και όταν το μέγεθος του dataset εκτοξεύεται από 1.000 σε 200.000 εγγραφές. Η ζώνη (Min-Max Range) υποδεικνύει ότι ενώ υπάρχουν κάποιες διακυμάνσεις, η πλειοψηφία των αναζητήσεων ολοκληρώνεται ταχύτατα. Το δεύτερο γράφημα (λογαριθμική κλίμακα) επιβεβαιώνει τη λογαριθμική πολυπλοκότητα O(log N) του αλγορίθμου, όπου η προσθήκη χιλιάδων νέων δεδομένων επιβαρύνει ελάχιστα το κόστος αναζήτησης. Συμπερασματικά, το Pastry αποδεικνύεται εξαιρετικά κλιμακώσιμο, διατηρώντας σταθερή απόδοση ανεξάρτητα από τον όγκο των αποθηκευμένων δεδομένων.



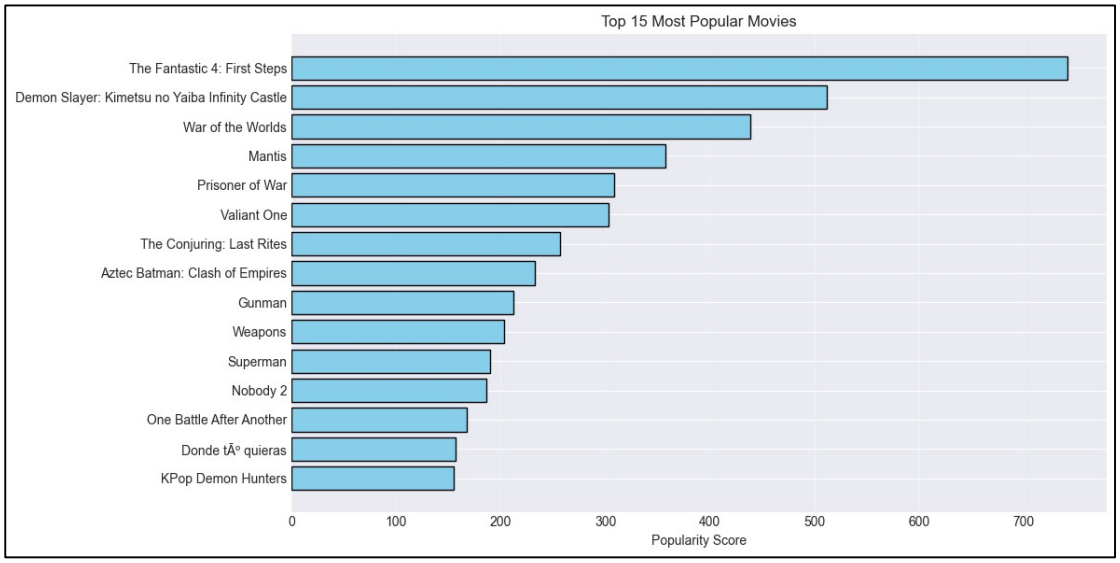
Συγκριτική Ανάλυση Απόδοσης Δικτύου TCP: Chord vs Pastry

Το παραπάνω διάγραμμα απεικονίζει τη συγκριτική επίδοση των πρωτοκόλλων Chord και Pastry σε πραγματικό δίκτυο TCP, εστιάζοντας στα μηνύματα, την καθυστέρηση και το ρυθμό εξυπηρέτησης. Παρατηρείται ότι το Pastry παράγει σημαντικά μεγαλύτερο όγκο μηνυμάτων για τη συντήρηση της τοπολογίας του, γεγονός που επιβαρύνει το δίκτυο και αυξάνει δραματικά τη μέση καθυστέρηση απόκρισης σε σχέση με το πιο αποδοτικό Chord.



Top 15 Most Popular Movies

Το διάγραμμα αυτό παρουσιάζει τις δεκαπέντε πιο δημοφιλείς ταινίες με βάση το σκορ δημοτικότητάς τους διατεταγμένες σε φθίνουσα σειρά. Η ταινία "The Fantastic 4: First Steps" κατέχει την κορυφή με το υψηλότερο σκορ, ενώ ακολουθούν οι υπόλοιποι τίτλοι με σταδιακά μειούμενη δημοτικότητα. Ο οριζόντιος άξονας απεικονίζει τη βαθμολογία δημοτικότητας, επιτρέποντας την εύκολη σύγκριση της απήχησης μεταξύ των πιο γνωστών κινηματογραφικών έργων του δείγματος.



Budget vs Revenue

Το διάγραμμα αυτό συσχετίζει τον προϋπολογισμό (Budget) των ταινιών με τα έσοδά τους (Revenue) σε λογαριθμική κλίμακα. Η διακεκομμένη κόκκινη γραμμή ορίζει το σημείο ισορροπίας (Break-even), όπου τα έσοδα ισούνται με τα έξοδα. Οι κουκκίδες που βρίσκονται πάνω από τη γραμμή αντιπροσωπεύουν κερδοφόρες παραγωγές, ενώ εκείνες κάτω από τη γραμμή δείχνουν ταινίες που δεν κατάφεραν να καλύψουν το κόστος παραγωγής τους.