Web 3.0 Chord DHT Resource Clustering

KaiHsiang Chan* and Young Yoon

Department of Computer Engineering, Hongik University, Seoul, South Korea
E-mail: zkdltid.chan@g.hongik.ac.kr; young.yoon@hongik.ac.kr
**Corresponding Author*

Received 15 March 2024; Accepted 19 June 2024

Abstract

This study explores the impact and challenges of new user behaviors in the Web 3.0 environment on distributed networks. The traditional Chord algorithm allows nodes to freely join and leave the network by hashing their IP addresses, and publishing and storing resources through the same hash function. When the keys of the resources are unique, the resources will be evenly distributed across each node, thereby achieving load balancing. However, in cases where many identical resources are published, this method leads to specific nodes bearing too much load, causing performance bottlenecks and resource concentration issues. In Web 3.0, when the nodes use the resource's topic as the key to publish resources, as the topic's popularity increases, the number of nodes using the same key as the publishing node and the nodes with demand for the topic resources will also increase. In the traditional Chord algorithm, the same key will be managed by the same node. The node responsible for the key needs to save the routing information of all related nodes and cope with a large number of resource requests for it. To address these issues, this paper proposes a new variant of the Chord algorithm, which uses two different Chord rings for resource clustering: one based on the hash of resource names and the other based on the hash of IP addresses. This method allows us to allocate resources more effectively, ensuring each node bears a reasonable load share according to capacity. This paper will present the design principles of this method and validate its effectiveness

Journal of Web Engineering, Vol. 23_5, 699–716.
doi: 10.13052/jwe1540-9589.2355
© 2024 River Publishers

in improving resource distribution and reducing the problem of single-point overload through experiments.

Keywords: Chord, DHT, Web 3.0, resource clustering, load balancing.

1 Introduction

With the rise of Web 3.0, distributed systems and peer-to-peer networks have fundamentally changed how information is stored and retrieved. In the Web 3.0 environment, the rapid development of peer-to-peer (P2P) networks and decentralized systems has made distributed hash tables (DHT) a key technology for implementing efficient, scalable, and fault-tolerant key-value storage solutions. Early implementations of DHT, such as Chord [10], CAN [8], Kademlia [5], Pastry [9], and Tapestry [11], laid the groundwork for the research and application of DHT technology through their unique routing and data management strategies. Among them, with its simple yet efficient design, the Chord algorithm attracted particular attention for using consistent hashing to implement fast key-value lookups and dynamic node join/leave handling on a virtual ring structure.

However, the evolution of P2P networks in the Web 3.0 environment has introduced new challenges, such as routing efficiency, data consistency, system scalability, and load balancing issues when dealing with many duplicate resources. These challenges have prompted improvements to the Chord algorithm to meet the requirements of current decentralized applications.

This study addresses a key issue in the resource-sharing design of P2P networks in the Web 3.0 environment: in traditional DHT designs, each node is randomly assigned a hash interval to manage the data for that interval. Nodes can find the node corresponding to the resource key value through the hash function and store the resource information on that node. Other nodes retrieve the resource information by finding the node corresponding to that resource key value. When a key value is frequently accessed or published, the node responsible for that hash interval may face an unfairly high load. However, this design does not account for the imbalance in resource access frequency, which may overload some nodes while others remain idle.

To solve this problem, this paper proposes an improved variant of the Chord algorithm. This variant introduces resource clustering and multiple Chord ring structures to enhance the system's load balancing and efficiency. In this design, resources with the same key values are grouped into the same cluster, and each cluster operates on a separate Chord ring. Each resource

key value will be mapped to a specific cluster, which not only helps alleviate the problem of single nodes being overloaded by popular key values but also promotes fairness among nodes and overall system performance. Nodes are no longer randomly responsible for unrelated key values but join the relevant cluster based on the resources they wish to publish, thus maintaining the network of the entire cluster.

This study's main contribution is proposing a new resource management mechanism that effectively solves the practical challenges faced by P2P networks in the Web 3.0 environment and offers new insights for designing and optimizing future distributed networks.

Furthermore, through a series of performance comparison experiments, this study further demonstrates the efficiency advantages of the improved algorithm over the standard Chord algorithm under different workload conditions.

The structure of this paper is as follows: Section 2 will introduce related work to establish the theoretical foundation of this study; Section 3 will detail our algorithm design and clustering strategy; Section 4 will present the experimental design and result analysis; finally, Section 5 will discuss the results and conclude.

2 Related Work

First, we need to understand the Chord algorithm's fundamental principles and operating mechanisms, as well as the main challenges encountered in the current Web 3.0 environment.

2.1 Chord Algorithm

2.1.1 Node identifier

Each node is assigned a unique identifier (ID) in the Chord network. This ID is generated by applying a consistent hashing function to the node's IP address or another unique attribute [4]. This ensures that the distribution of node IDs is uniform, reducing the hotspot issue.

2.1.2 Hash space

Chord utilizes a circular hash space, ranging from 0 to $2^m - 1$, where m is the number of bits in the hash key. This circular space is known as the Chord ring. Both nodes and key-value pairs are mapped onto this space.

2.1.3 Node join

When a new node wishes to join the Chord network, it finds its position in the Chord ring based on its ID. Each node maintains the key-value pairs between itself and the next node (the first node in the clockwise direction in the hash space). Adding a new node can lead to the redistribution of key-value pairs.

2.1.4 Stabilization algorithm

Chord implements a set of stabilization algorithms to manage nodes joining and leaving. It ensures network stability by regularly sending health check messages to its successor and updating information about its successor. This algorithm ensures that the system can quickly reach a new stable state, even in the face of dynamic node changes.

2.1.5 Key-value pair storage

Each key-value pair is also mapped onto the Chord ring using the same consistent hashing function. A key-value pair is stored on the first node in the clockwise direction whose ID is greater than or equal to the key's hash value.

2.1.6 Lookup process

When looking up a key, the query starts at any node and is passed along the Chord ring clockwise until it reaches the node responsible for the key. Chord optimizes the lookup process with a structure known as a `finger table`, significantly reducing the number of hops in the network.

2.1.7 Load balancing

Assuming that keys are unique, the use of consistent hashing means that each node roughly handles an equal number of key-value pairs, thereby achieving load balancing.

2.2 Challenges in the Chord Algorithm for Resource Management

Assuming resources are hashed based on their content, we can ensure the uniqueness of key-value pairs. However, this method relies on search engines similar to BitTorrent [1, 7], allowing users to locate nodes storing the relevant resources. While this approach ensures even storage of resources across nodes, locating specific resources becomes challenging without maintaining

a central search engine. This paper aims to address user behaviors in Web 3.0, in particular, how to effectively search for required resources in the absence of search engines. Direct hashing based on resource names inevitably leads to many identical key-value pairs, challenging the load balance achieved by consistent hashing in the Chord algorithm. An excessive number of identical key-value pairs can severely burden the node responsible for that hash value, especially considering the risk of stress testing when a large volume of requests occurs. We propose a method to cluster identical resource names to address this issue to improve load distribution.

2.3 Optimization of the Chord Algorithm in Previous Research

The academic community has proposed various strategies for optimizing the Chord algorithm to enhance system performance and scalability. One strategy introduces super nodes acting as gateways [2], which help locate and connect other sub-nodes, thus improving the overall network's availability. This strategy also involves data sharding across multiple sub-nodes to increase download speeds. However, this design often requires super nodes to maintain an extensive list of sub-nodes, which poses an additional burden when a large and unpredictable number of sub-nodes join. Super-nodes need to record extensive information to assist in user localization.

Another paper [3] mentions an embedded ring method as an optimization to address the issue of joining nodes with identical hash values, reducing the frequency of conflicts when multiple nodes join. The paper describes how nodes within the entire circle collectively store blockchain blocks, with block identifiers (IDs) generated by cryptographic hash functions. This ensures that blocks can be evenly distributed and stored across nodes in the cluster. While the embedded ring structure is similar to our design, it mainly focuses on optimizing the conflict time when joining the network and blockchain applications, differing from our strategy to address uneven data access.

This research aims to implement a more decentralized optimization approach, allowing every node in the network to maintain its performance while being robust to dynamic changes in other nodes. Specifically, we hope that nodes can operate independently of the joining or leaving of other nodes and remain unaffected in their performance even in the face of suddenly increased queries for popular keywords based on current events. Such design considerations aim to provide a more scalable and performance-stable solution for distributed networks in the Web 3.0 environment.

3 Method

This study aims to improve the Chord algorithm to address specific challenges in the Web 3.0 environment, particularly regarding resource management and node dynamics. We propose a series of optimization strategies and conduct related simulation tests to evaluate the effectiveness of these improvements.

3.1 Two Type Chord Ring

We designed two different types of `chord rings`: `Resource Ring` and `IP Ring`. This design provides better scalability for resource management and node organization.

3.1.1 IP ring (IPR)

Like the traditional Chord algorithm, this type of chord ring uses the node's IP address as input for hashing to generate a `node ID`. This `node ID` determines the node's position on the `IP ring` for organizing nodes. Each cluster based on specific resources will maintain an independent `IP ring`. However, unlike traditional DHT applications, the nodes on the IP ring in this paper do not need to store the resource information published by other nodes; they only need to manage the resources published by themselves and internal cluster information, such as finger table, successors, etc.

3.1.2 Resource ring (RR)

For resource lookup, we require a `resource key`, which can be any value uniquely identifying the resource, similar to keywords in a search engine. This chord ring hashes this resource key to transform it into a `cluster ID`, which represents the position of that resource cluster on the `resource ring`.

This method allows related resources to be effectively aggregated and managed. However, each `cluster ID` requires a corresponding node to maintain it. In our design, we refer to these nodes as leader nodes. Referencing Figure 1, we can see that each cluster on the `resource ring` has an `address`, which is the IP of the cluster's leader node. The leader node is responsible for maintaining important information of the `resource ring`, such as `finger table, successor list`, etc. The `IPR-successor-list` is a new structure we have added, which stores the `successor list` corresponding to the successor node on the `IP ring`. We will discuss this issue further in the next section.

A super node could maintain this information in traditional distributed systems since the leader node will bear more load. When the leader node

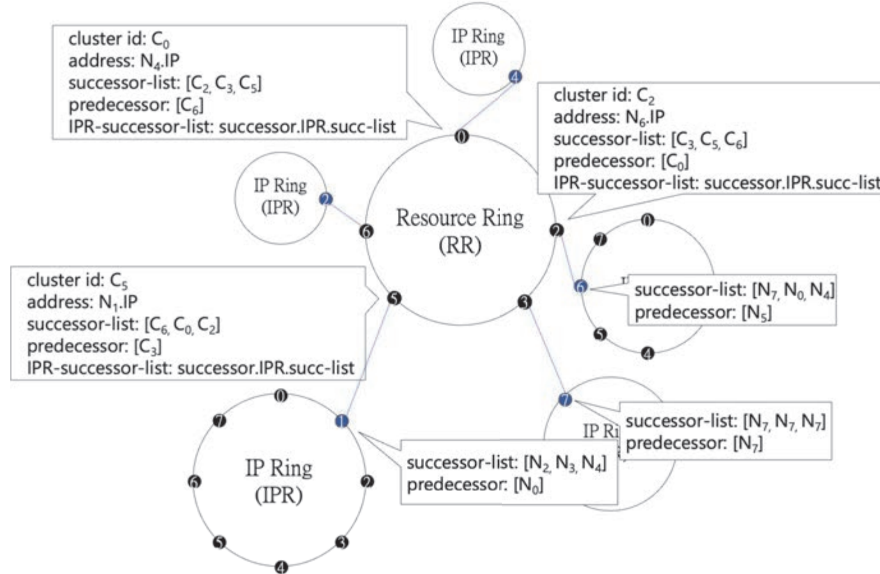


Figure 1 The figure illustrates our model architecture, where each cluster ID represents a cluster, and each IP ring represents the organization of nodes within a cluster. It is observable that each cluster possesses an address, which is the IP of the leader node of that cluster.

is overloaded, a leader node election mechanism, such as `Raft` [6] or `Paxos` [12], could elect a new leader node. However, this method is unsuitable for situations with many sub-nodes due to the massive communication overhead and the need for a super-node to maintain all sub-node information, which would significantly burden the super-node. We will compare the differences with our method through experiments later.

Utilizing the characteristics of the `Chord algorithm`, suppose we need to query if a resource key exists on the `resource ring`; we only need to hash this resource key. Referencing Figure 1, through the `find successor` function, we can locate the `successor` node corresponding to this key.

By comparing the `cluster ID` of this `successor` node, we can determine whether this `resource key` exists in this cluster:

$$\text{clusterID} = \text{hash}(\text{resourceKey}) \tag{1}$$

$$\text{successor} = \text{findSuccessor}(\text{clusterID}) \tag{2}$$

$$\text{resourceExist} = \begin{cases} \text{successor}, & \text{if clusterID} = \text{successor.clusterID}, \\ \text{null} \end{cases} \tag{3}$$

Through the return value of `resourceExist`, we can perform different operations based on the client's needs, which include:

- Joining a cluster
- Creating a cluster
- Retrieving resources.

3.1.3 Joining/creating a cluster

For clients intending to share resources, the return value of `resourceExist` helps determine whether the `key` exists on the `resource ring`. If `resourceExist` returns `null`, it indicates that the `key` does not exist on the `resource ring`. The client will then create a new `IP ring` and become the leader node of this cluster, joining the `resource ring`. If `resourceExist` returns a `successor`, it signifies that the key exists on the `resource ring`. The client will join the cluster's `IP ring` through the `successor` node.

Each node can join or create more than one and different clusters based on its needs. However, as the number of joined clusters increases, the node will need to maintain information for different clusters. Although this will impact the node's performance, it allows it to control its performance impact based on its choices rather than being affected by changes in the entire network.

3.1.4 Finding resources

For clients looking to find resources, the return value of `resourceExist` determines if the `key` exists on the `resource ring`. Then, by obtaining the `successor` node, which is the current leader node of that cluster, the client can retrieve information about the resource.

3.1.5 Leader Node Change

3.1.5.1 *IPR-successor-list*

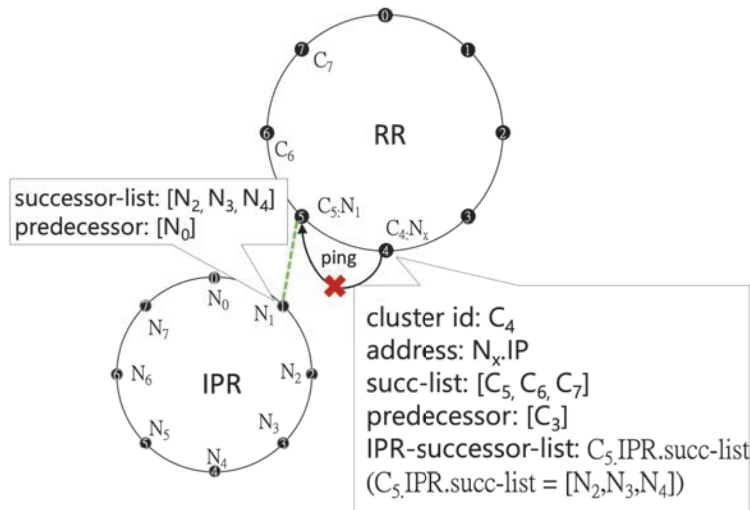
In our dual-ring design, besides maintaining the `successor list` of the `resource ring`, as mentioned earlier, we added a `successor list` for the `IP ring` called `IPR-successor-list`. It will be updated with the `Chord algorithm's stabilization algorithm`, where the `resource ring's leader node` requests the `successor` to update its `successor list` on the `IP ring`. This `IPR-successor-list` will be crucial when changing the leader node during node exits and polling.

3.1.6 Leader node exit

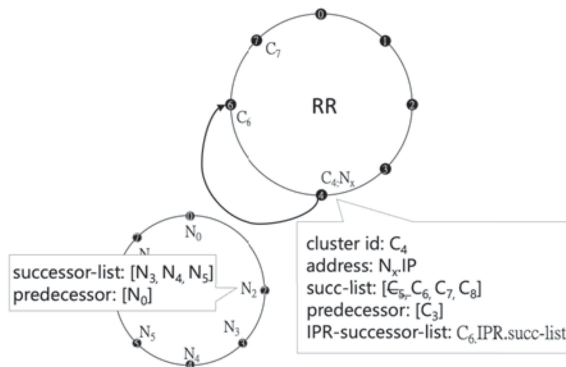
In the typical Chord algorithm, the `stabilization algorithm` regularly checks the node's `successor` to ensure network stability. If a node finds its

`successor` unresponsive, it will update its `successor` using the `successor list`. However, due to our dual-ring design, we cannot use this method for updating the `resource ring's successor`, as the `successor` would be identified as the next cluster's leader node, leading to the loss of the leader node and the cluster being split and inaccessible.

Figure 2 shows this scenario: suppose the `leader node` of our C_4 cluster discovers that its `successor` (i.e., C_5) is inoperative. If it updates its `successor` based on the `successor list` of the `resource Rring`,

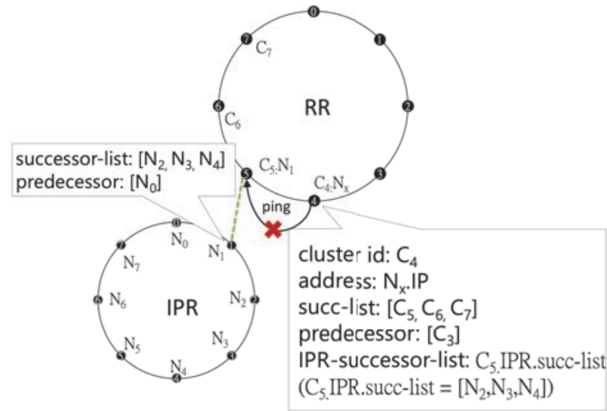


(a) ping failed

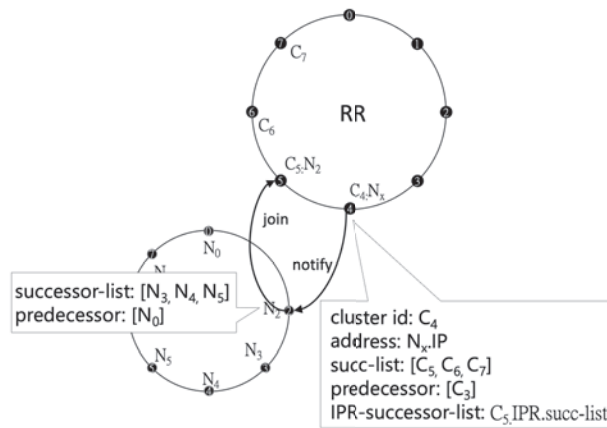


(b) successor update failed

Figure 2 Leader node failure with successor list.



(a) ping failed



(b) successor update success

Figure 3 Leader node failure with IPR successor list.

as illustrated in Figure 2(b), the `successor` would be identified as the C_6 cluster. Consequently, the C_5 cluster, having lost its `leader node`, would become fragmented and could not be accessed correctly.

Therefore, through the `IPR-successor-list`, our leader node can notify the next node in this list, and the notified node will become the new leader node of the cluster, ensuring the entire cluster's stability, as shown in Figure 3.

3.1.7 Leader node polling

The leader node plays a crucial role in the network, especially in resource retrieval. To distribute traffic evenly across all nodes in the cluster and achieve load balancing, after implementing the leader node exit mechanism by setting parameters, we only need the leader node to return an exit message during the `resource ring's predecessor ping` at specific intervals. The `predecessor` node can find a new leader node using the `IPR-successor-list`. Additionally, when the leader node experiences high traffic, it can autonomously change the leader node in the same way.

3.1.8 Handling non-leader node exits

When a non-leader node plans to exit, we use the stability strategy from the standard Chord algorithm to handle it.

4 Experiment

This section evaluates the effectiveness of our solution on balancing the load among Web 3.0 Chord nodes.

First, we simulated scenarios in which both the traditional Chord and the Web 3.0 Chord algorithms handle the publication of resources with a large number of identical resource keys. This allowed us to observe the overhead on different nodes in both algorithms (the leader and non-leader nodes in the Web 3.0 Chord IP ring and the responsible and successor nodes in the traditional Chord). Finally, we performed a cross-comparison of the results from both algorithms.

Subsequently, we periodically queried the same resource to verify if it effectively allowed all nodes in the IP ring to receive requests, thereby evenly distributing the network request load.

Our tests were conducted on a MacBook M1 Pro, which features an Apple M1 Pro chip with a 10-core CPU and 16GB of unified memory. The Chord algorithm was implemented using Golang. Each node was run as a separate process on this system.

4.1 Memory Usage Evaluation

In traditional applications of the Chord algorithm, when a node publishes a key-value pair, this pair is stored on the node's successor. We will compare our design with the traditional Chord algorithm to prove that it effectively

reduces the single-point overload issue. In this experiment, we assume there are 2000 nodes possessing a resource called A.

In the traditional Chord algorithm, through the hash function, we can identify a corresponding successor node, which will be responsible for assisting in storing the routing information of the node owning the A resource. In our Web 3.0 Chord design, we similarly use A as the resource key to join the cluster.

In the traditional Chord algorithm, we select a random node within the ring, as well as the successor node responsible for storing the A resource, as our observation subjects and observe the memory usage of these two nodes. Figure 5 illustrates the memory cost difference between the successor storing the A resource and a random node within the ring. Since the successor node will store a large amount of information, it will incur increased memory costs, while the memory cost of a random node within the ring will remain at a lower level.

Next, we observe the leader node in our Web 3.0 Chord design compared to any leader node on the resource ring of a sample cluster. Figure 4 shows the memory costs for the Web 3.0 Chord. Even as the number of nodes in the cluster increases, the memory costs for the leader node and any sample leader node on the `resource ring` remain similar. However, due to the dual-ring

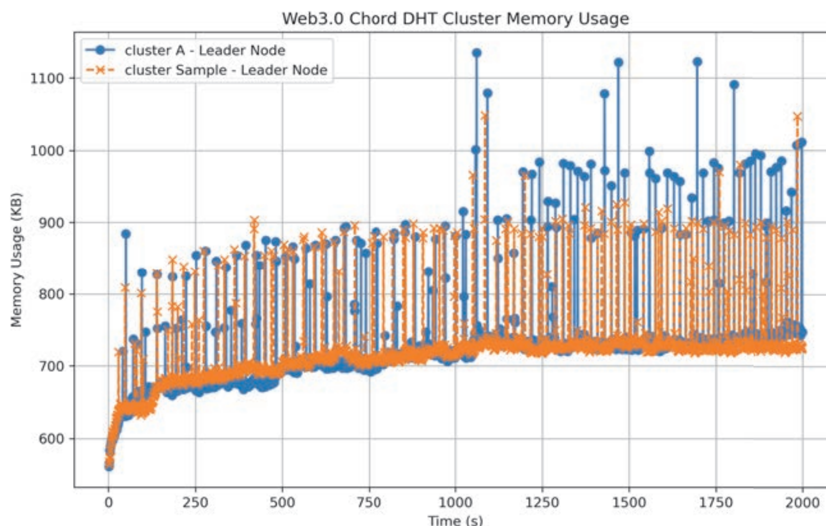


Figure 4 Web 3.0 Chord memory usage (2000 nodes in cluster(A) vs. 1 node in cluster(sample)).

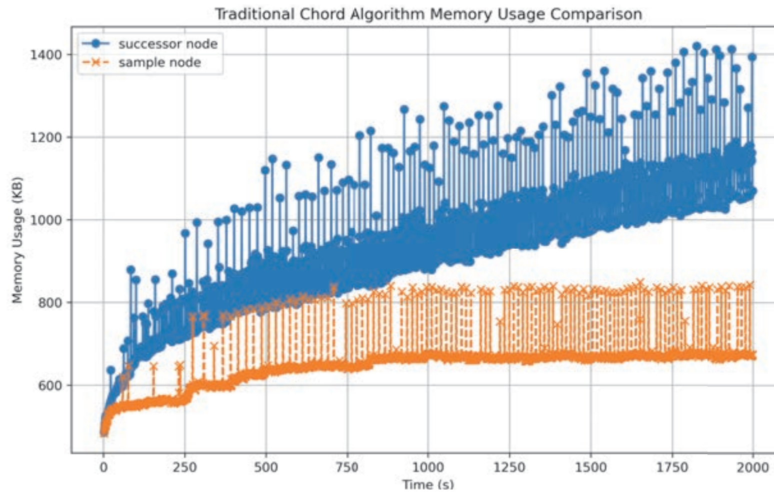


Figure 5 Traditional Chord algorithm memory usage (2000 resource in successor node vs. 0 resource in sample node).

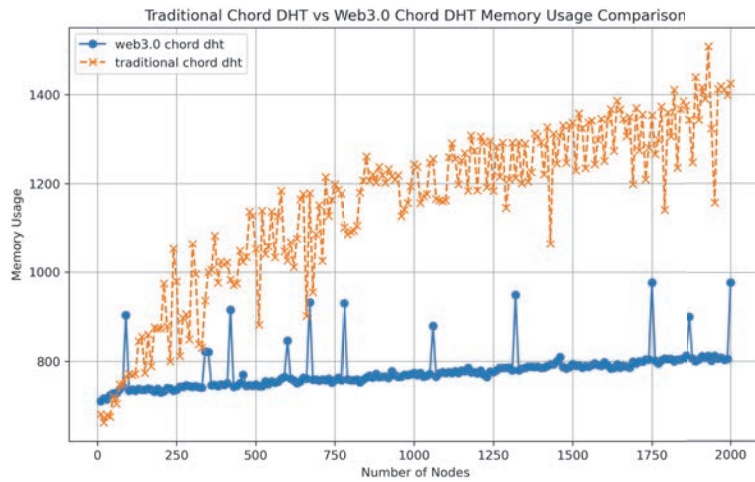


Figure 6 Traditional Chord algorithm vs. Web 3.0 Chord memory usage comparison.

structure, the basic memory cost of the leader node will be slightly higher than that of the traditional Chord algorithm before a large amount of data is added, as the leader node needs to maintain information about both rings.

Finally, through Figure 6, we can compare the memory cost differences between the traditional Chord algorithm and our dual-ring design.

Table 1 Node received request counts

Node id	Req. Cnt. (10 nodes)	Req. Cnt. (30 nodes)	Req. Cnt. (50 nodes)	Req. Cnt. (100 nodes)
$node_1$	62	19	16	9
$node_2$	64	20	12	8
$node_3$	60	16	8	5
...	...	...	...	...
$node_{10}$	60	16	8	4
$node_{30}$	–	16	8	5
$node_{50}$	–	–	8	5
$node_{100}$	–	–	–	5

Note: Req. Cnt. Stands for Request Count.

4.2 Evaluation on Query Request Balancing

In this study, we established a scenario to observe the number of query requests received by each node and its memory usage when different numbers of nodes (namely, 10, 30, 50, and 100) join a cluster. This experimental setup is designed to simulate the querying behavior of many users for the same specific resource in a distributed system and its impact on the system. To simulate a realistic situation, we assume this operation will result in a memory consumption of 300 KB per node. This memory consumption is intended to reflect the resource occupancy during the data loading and computation process to handle query requests. Moreover, we introduced a one second processing delay to simulate the computational delay and data exchange time during the query processing. Finally, we set up a polling of the leader node in the cluster every 30 seconds. Users will make query requests for resource A to any node on the entire `resource ring`.

As shown in Table 1, with the increase of nodes, the requests are evenly distributed among the nodes. For `node1`, in the case of 10 nodes, it received 62 requests, while in the case of 100 nodes, it received 9 requests.

Figure 7 shows the change in memory usage of a single node as the number of nodes increases. Through simulating memory consumption, it is clear that as the number of nodes within the cluster increases, the number of requests received decreases, and the memory cost of maintaining the Chord ring does not increase with the addition of nodes.

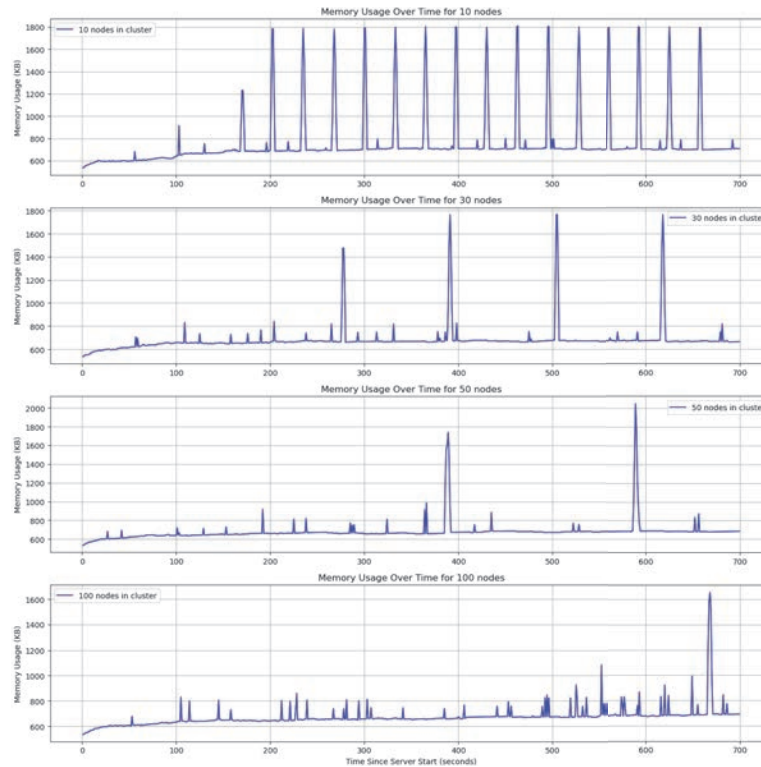


Figure 7 Web 3.0 Chord memory usage.

5 Conclusion

New user behaviors in the Web 3.0 environment inevitably lead to uneven resource distribution. For instance, with the advent of new technologies like artificial intelligence and blockchain, a vast amount of resources on the same topics will be published or queried, which can cause specific nodes to be overloaded when using traditional Chord algorithms. This paper proposes a new Chord DHT design aimed at solving the issue of load balancing in the Web 3.0 environment. While the leader node incurs some additional load to maintain two distinct Chord rings, our experiments have demonstrated that through a polling mechanism of the leader node, our design allows nodes to take turns sharing the load.

However, although our design has achieved significant results in balancing the load with a large volume of identical resources, there is still room for improvement in handling massive concurrent queries and during

the transition of leader nodes. Future research could further explore how to optimize the management strategy of leader nodes to enhance the system's performance and stability.

With ongoing optimization and improvement, we believe this design will provide strong technical support for building a more efficient and reliable infrastructure for Web 3.0.

Acknowledgments

This work was supported by 2024 Hongik University Innovation Support Program Fund and 2024 Hongik University Research Fund, by the MSIT (Ministry of Science and ICT), Korea under the ITRC (Information Technology Research Center) support program (RS-2023-00259099) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation, by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2023-00240211).

References

[1] Beth Cohen.Incentives build robustness in bit-torrent. 2003.

[2] Mirko D'Angelo and Mauro Caporuscio. Sa-chord: A self-adaptive p2p overlay network. In *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 118–123, 2018.

[3] Xie Jiagui, Li Zhiping, Gao Likun, and Nie Fanjie. Dht cluster node join improvement and load balancing. In *2021 IEEE International Conference on Electronic Technology, Communication and Information (ICETCI)*, pages 650–654, 2021.

[4] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. ACM STOC*, 02 2001.

[5] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, pages 53–65, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[6] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference*

on USENIX Annual Technical Conference, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[7] Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. The BitTorrent p2p file-sharing system: Measurements and analysis. In Miguel Castro and Robbert van Renesse, editors, *Peer-to-Peer Systems IV*, pages 205–216, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[8] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, Aug 2001.

[9] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Rachid Guerraoui, editor, *Middleware 2001*, pages 329–350, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

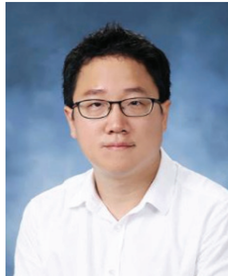
[10] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

[11] B.Y. Zhao, Ling Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

[12] Álvaro García-Pérez, Alexey Gotsman, Yuri Meshman, and Ilya Sergey. Paxos consensus, deconstructed and abstracted (extended version), 2018.

Biographies

KaiHsiang Chan is currently a master's student in computer engineering at Hongik University, having enrolled in 2022. His research focuses on the areas of Web 3.0 and distributed systems.



Young Yoon is an associate professor of computer engineering at Hongik University and the CTO of Neouly Incorporated. His research interests include distributed systems, middleware, cyber security, AI applications, and emerging Web 3.0 themes. Yoon earned his B.A. and M.Sc. in computer sciences at the University of Texas at Austin in 2003 and 2006, respectively. He earned his Ph.D. in computer engineering at the University of Toronto in 2013.