Demo: Skip Graph Middleware Implementation

Yahya Hassanzadeh-Nazarabadi*†, Nazir Nayal†, Shadi Sameh Hamdan†, Ali Utkan Şahin†, Öznur Özkasap†, and Alptekin Küpçü †
DapperLabs, Vancouver, Canada*
Department of Computer Engineering, Koç University, İstanbul, Turkey†
{yhassanzadeh13, nnayal17, shamdan17, asahin17, oozkasap, akupcu}@ku.edu.tr

***Abstract*—Skip Graphs are Distributed Hash Table (DHT)-based data structures that are immensely utilized as routing overlays in Peer-to-Peer (P2P) applications. In this demo paper, we present the software architecture of our open-source implementation of Skip Graph middleware in Java. We also present a demo scenario on configuration and constructing an overlay of Skip Graph processes in a fully decentralized manner. Our implementation is capable of hosting data objects at the Skip Graph processes and serving as a P2P data storage platform as well. Our middleware implementation provides an open-source platform to support Skip Graph-based applications on top of it.**

***Index Terms*—Skip Graph, Overlay, P2P, Distributed Hash Table, Java, Implementation.**

I. Introduction

Skip Graphs [1] are a distributed key-value store of objects. They represent each object by a node, maintain it on distinct Skip Graph processes, and make it accessible to other processes' queries. Following this behavior, they are regarded as a type of Distributed Hash Table (DHT) class of distributed data structures. In this distributed systems realization of Skip Graphs, a node is either a process (i.e., peer) or a data object, and is identified with a network address and two distinct keys: a name ID and a numerical ID. Having n nodes in a Skip Graph overlay, each nodes needs to only maintain a *lookup table* of $O(\log n)$ many other nodes. With solely relying on its lookup table, each node can find the (IP) address of other nodes in a fully decentralized way by searching for their name ID [2] or numerical ID [1]. Both search protocols are done with a message complexity of $O(\log n)$. Due to its scalability, logarithmic message complexity, and load balancing, Skip Graphs support an extensive domain of applications including Peer-to-Peer (P2P) cloud storage [3], [4], [5], sensor networks [6], and blockchains systems [7], [8].

Despite the aforementioned examples of the research and development domains on Skip Graphs, there is no reliable and open-source implementation of Skip Graphs as distributed systems middleware. To provide an open-source platform that is capable of hosting Skip Graph-based applications on top of it, in this demo paper we present the first open-source implementation of the Skip Graph middleware in Java [9] with the following list of features:

Decentralization: Running independent instances of our Skip Graph middleware builds up a P2P Skip Graph overlay in a fully decentralized manner. By the decentralization, we mean the setup in which no specific process is in charge of bootstrapping others.

Process-level Operations: Our implementation supports the basic Skip Graph operations including joining and leaving the overlay as well as searching for other Skip Graph processes based on either their name ID or numerical ID.

Distributed Data Objects: Our implementation of Skip Graph supports the notion of distributed data objects. Distributed data objects are data objects (e.g., files) that are distributed across the processes of Skip Graph overlay [10]. This allows Skip Graph processes to represent the data objects they own as Skip Graph nodes. In this implementation, the numerical ID of data object nodes is assigned as the hash value of their corresponding data objects using a collision-resistant hash function. Representing numerical IDs as hash values provides a one-to-one mapping between the numerical ID space and data objects. In this way, each data object is uniquely identified by its numerical ID. The approach for name IDs is different, and the name IDs can be assigned by the developer, without any uniqueness constraint. This allows exposing attributes in the name IDs and provides decentralized attribute-based query processing over the data objects that are maintained on a Skip Graph. We discuss this idea in more detail in [7]. The address of a data object node is the same as its owner process. Hence, any search query for the identifiers of a data object is routed to and answered by its corresponding owner process. By supporting distributed data objects, our Skip Graph implementation can be utilized as other DHTs in their applications that involve maintaining distributed data objects, e.g., Chord [11]. Moreover, this supports applications of Skip Graphs that rely on the storage functionality of data objects as key-value pairs.

Layered Architecture: In our implementation, the Skip Graph middleware is decoupled into two layers: the Skip Graph overlay and the underlying network. The Skip Graph overlay is responsible for performing Skip Graph protocols. The underlying network provides end-to-end communication among Skip Graph processes, and enables them to function properly. The layered architecture supports the independency of the Skip Graph protocols functionality from the underlying network protocol. This provides a modular design where a vast variety of underlying network protocols can be employed to support Skip Graph overlay protocols, e.g., Java Remote Method Invocation (RMI) [12], gRPC [13], Distributed Shared Memory (DSM) [14], Transmission Control Protocol (TCP), and User Datagram Protocol (UDP) [15].

Logging: Our implementation provides logging function-

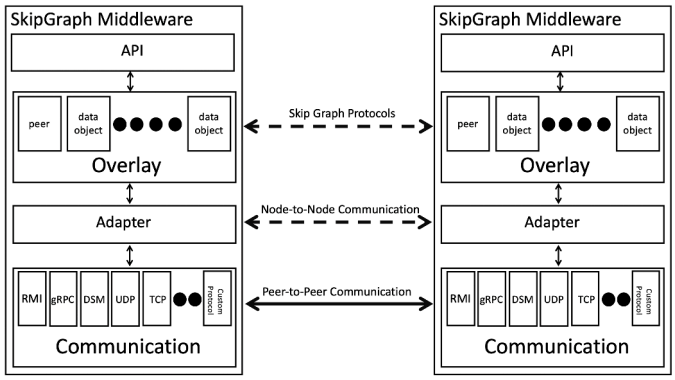

Figure 1: An overview of the layered architecture of the Skip Graph middleware in our implementation. The solid horizontal arrow corresponds to a real peer-to-peer communication via the underlying network. The dashed horizontal arrows correspond to the realization of services through the underlying network.

ality for Skip Graph middleware using Apache Log4j [16], which is a Java-based logging framework. In our implementation, we utilize several levels of Apache Log4j logging to cover different purposes. By default, we enable info-level logging for all high-level operations occurring in the Skip Graph process, e.g., joining and leaving the overlay, putting and getting data objects, and initiating and routing search queries. To facilitate debugging, we enable debug-level logging to scrutinize the intermediate steps the process takes to perform an operation, e.g., updating a lookup table entry. Likewise, we enable error-level logging to report the errors occurring during the Skip Graph protocol execution, e.g., duplicate visit of a process on the same search query. This logging feature establishes the performance monitoring and diagnostic features especially in deployments over the cloud computing platforms. For example, in deploying each Skip Graph process as a Compute Engine instance in Google Cloud Platform (GCP), the StackDriver module of GCP is configured to collect the individual logs of the processes, aggregate them all, and provide digest reports on their performance accordingly in the StackDriver console. It also enables performing arbitrary queries on the logs to diagnose a distributed error (e.g., distributed deadlock) on the overlay of Skip Graph processes.

II. Software Architecture

Figure 1 represents the software architecture of Skip Graph middleware instances as well as their interactions. The Skip Graph software architecture in our implementation follows a layered approach. Layers of each Skip Graph middleware instance from top to bottom are: *API*, *Overlay*, *Adapter*, and *Communication*.

The Abstract Programming Interface (API) layer is the front-end layer of our Skip Graph middleware implementation. It represents a Skip Graph process to the user. The representation is done by exposing a set of functions, e.g., the search for numerical ID query. The API layer is responsible to take the function calls from the user, process them by interacting with the Overlay layer, and return the result to the user.

The Overlay layer maintains the set of local Skip Graph nodes that this instance of middleware owns. By the instance of middleware, we mean the corresponding Skip Graph process of the middleware. The set of local nodes consists of a peer that runs the process, as well as all the data objects that the peer hosts. The Overlay layer is responsible for representing these local nodes in the Skip Graph overlay, and making them accessible and retrievable by other processes of the system. The Overlay layer also takes the function calls from the API layer, and process them either locally, or by interacting with the Adapter layer.

The Adapter layer is the broker [15] between the Overlay and Communication layers, and handles all the access calls between these two layers. Both the Overlay and Communication layers register themselves to the Adapter layer and then they can invoke each other through that layer. The benefits of having an adapter layer are two-fold. First, it provides a transparent node-to-node communication service to the Overlay layer. In other words, handling the (local) queries about the local nodes at an Overlay layer instance is done in the same way as the remote queries about nodes owned by another Overlay instance. Without the Adapter layer, the Overlay layer should be self-aware of all the nodes that it manages at the local process. It should handle the queries about the local nodes locally while handling the queries about the remote nodes via the underlying network. However, with the Adapter layer in place, all queries are going through the Adapter layer, and then depending on the query being about a local or remote node, it is bounced back to the local Overlay instance or is forwarded to the underlying network, respectively. The second key benefit of having the Adapter layer in place is that it provides interface independence interactions among the Overlay and Communication network. In other words, neither of these two layers should be aware of each other as well as each other's interface. Rather, they both are sufficient to comply with the interface of the Adapter layer. As explained later, this enables the utilization of a vast variety of underlying network protocols at the Communication layer without the need to change anything at the Overlay layer.

The Communication layer is responsible for providing peer-to-peer communication in the underlying network. It takes a query for a remote Skip Graph process from the Adapter layer and sends it to the target process via the underlying network. Similarly, it takes the incoming traffic for the Skip Graph process and passes that to the Adapter layer. The Communication layer in our implementation of Skip Graph supports a large array of network protocols including Java Remote Method Invocation (RMI) [12], gRPC [13], Distributed Shared Memory (DSM) [14], Transmission Control Protocol (TCP), and User Datagram Protocol (UDP). The Communication layer is also capable of being configured by the custom protocols that are developed by the user.

III. Sample Demo Scenario

Listing 1 represents a sample Skip Graph middleware instance configuration. In our implementation, a Skip Graph process is configured by specifying its features in `config.yaml` file that is located at the root of the source code [9]. In this Listing, `config.version` represents the version of the configuration file, which defines how the parameters in this file should be interpreted. Depending on the version, configuring

```
config:
    version: 1.0

network:
    protocol:   RMI
    port:  1099

process:
    version: 1.0
    name_id:
        protocol: self-assigned
        value: 0b00110101
    numerical_id:
        protocol: randomized
    storage:
        address: /usr/local/skipgraph
```

Listing 1: A sample config.yaml file for Skip Graph middleware

some parameters may be mandatory or optional. The optional parameters are set to a default value by the Skip Graph process. However, skipping mandatory configuration parameters results in the Skip Graph process terminating with an error indicating there are not enough parameters to configure, and listing up the missing ones.

`network.protocol` defines the protocol that the Communication layer of Skip Graph is operating on. The current implementation supports `RMI`, `DSM`, `gRPC`, `TCP`, and `UDP`. The developer can also develop any arbitrary custom network protocol and configure it as detailed in [9]. `network.port` defines the port number that the Skip Graph process exposes for `network.protocol`.

The `process` attribute spans the configuration parameters of the process at the Skip Graph protocol level. `process.version` defines the version of the Skip Graph process software. Specifying the version enables remote processes to negotiate and acknowledge each others' versions before engaging in the Skip Graph protocol. This approach counters the protocol-level errors due to inconsistent versions at which Skip Graph processes are operating on. `process.name_id.protocol` represents the name ID assignment protocol. The current version supports the self-assigned and randomized modes. However, any customized name ID assignment protocol [17] can be developed and configured. The self-assigned approach of name IDs implies the name ID is defined by the configuration file, and requires the file to have a `process.name_id.value` field. The name ID value can be either in Binary (i.e., `0b`) or Hexadecimal format (i.e., `0x`). In both cases, the maximum length of name ID is 256 bits to be compatible with SHA3-256 format [18]. If the self-assigned name ID by the user is shorter than 256 bits, it is extended by zeros, i.e., zeros are added to its right to reach 256 bits in length. The randomized name ID protocol takes the SHA3-256 value of the IP address and port number of the process as its name ID. `process.numerical_id.protocol` represents the numerical ID assignment protocol. Similar to the name ID, it can be self-assigned, randomized, or a user defined protocol. `storage.address` is an optional field representing the local persistent storage path of this process on the disk. If no address is provided, the process establishes its persistent storage in its root directory. The persistent storage maintains the state of the Skip Graph process, and helps it recover swiftly from a failure, and continue from where it left off.

Once the configuration file is ready, the user executes the Skip Graph middleware as detailed in [9]. Upon startup, to join the Skip Graph overlay, the process asks for the address (i.e., IP and port number) of its introducer. The introducer is one arbitrary node of Skip Graph overlay that helps a new Skip Graph process joins the overlay [17]. After joining the overlay, the middleware provides command-line access for the user, which supports adding or removing a data object, and searching for a name ID or numerical ID. By inserting a data object in Skip Graph overlay, the process represents itself as the host of the object, and any search for the name ID or the numerical ID of that object is routed to this Skip Graph process. Also, a search for name ID or numerical ID that is initiated by a Skip Graph process is routed through the Skip Graph overlay and is replied to the initiator by all the processes that hold the target name or numerical ID, respectively. The result of each operation is displayed to the user in Log4j format.

REFERENCES

[1] J. Aspnes and G. Shah, "Skip graphs," *ACM TALG*, 2007.

[2] Y. Hassanzadeh-Nazarabadi, A. Küpçü, and Ö. Özkasap, "Laras: Locality aware replication algorithm for the skip graph," in *IEEE/IFIP NOMS*, 2016.

[3] ——, "Decentralized utility-and locality-aware replication for heterogeneous dht-based p2p cloud storage systems," *IEEE TPDS*, 2019.

[4] ——, "Decentralized and locality aware replication method for dht-based p2p storage systems," in *FGCS*. Elsevier.

[5] Y. Hassanzadeh-Nazarabadi and Ö. Özkasap, "Elats: Energy and locality aware aggregation tree for skip graph," in *IEEE BlackSeaCom*, 2017.

[6] P. Desnoyers, D. Ganesan, and P. Shenoy, "Tsar: a two tier sensor storage architecture using interval skip graphs," in *Proceedings of the 3rd international conference on Embedded networked sensor systems*, 2005.

[7] Y. Hassanzadeh-Nazarabadi, A. Küpçü, and Ö. Özkasap, "Lightchain: A dht-based blockchain for resource constrained environments," *arXiv preprint arXiv:1904.00375*, 2019.

[8] Y. Hassanzadeh-Nazarabadi, N. Nayal, S. Sameh Hamdan, Ö. Özkasap, and A. Küpçü, "A containerized proof-of-concept implementation of lightchain system," in *ICBC*. IEEE, 2020.

[9] "Skip graph node:https://github.com/yhassanzadeh13/skipgraphnode."

[10] W. Emmerich, *Engineering distributed objects*. John Wiley & Sons Software, 2000.

[11] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM*, 2001.

[12] E. Pitt and K. McNiff, *Java. rmi: The remote method invocation guide*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[13] grpc.io, "Introduction to grpc," https://grpc.io/docs/what-is-grpc/introduction/, May 2020.

[14] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," *Computer*, 1996.

[15] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[16] Apache, "Apache log4j 2," https://logging.apache.org/log4j, May 2020.

[17] Y. Hassanzadeh-Nazarabadi, A. Küpçü, and Ö. Özkasap, "Locality aware skip graph," in *IEEE ICDCSW, 2015*.

[18] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, "Keccak implementation overview, version 3.2," 2012.