Documentation for Rafbook — A Fault-Tolerant and Topologically Optimized Distributed System Demo

Ognjen Prica

June 11, 2024

Abstract

This paper presents a demonstration of a distributed system featuring an adaptive, fault-tolerant virtual file system and a friendship management system between processes. The system is built upon the Chord distributed hash table (DHT) architecture, utilizing TCP/IP for communication. The system can tolerate up to two simultaneous node failures and employs a hybrid failure detection mechanism combining a buddy system and a heartbeat system. Public files are replicated across multiple nodes for fault tolerance, with the initial backup stored at the location determined by the Chord hash and additional replicas stored on a configurable number of successor nodes. The Suzuki-Kasami algorithm is used to achieve mutual exclusion in the system, ensuring that only one process can access the critical section at a time, it's main use being the handling of concurrently joining nodes. The communication protocol includes various message types for pinging, file management, friend management, and token handling in the Suzuki-Kasami algorithm.

Contents

1 Introduction

This document introduces a demonstration of a distributed system, featuring an adaptive, fault-tolerant virtual file system and a friendship management system between processes. The system is built upon the Chord DHT architecture, utilizing TCP/IP for communication. The following sections will provide a comprehensive overview, user guide, and technical documentation to help you explore and understand the system.

2 System Overview

This section provides a high-level view of the distributed system's architecture, components, and communication mechanisms.

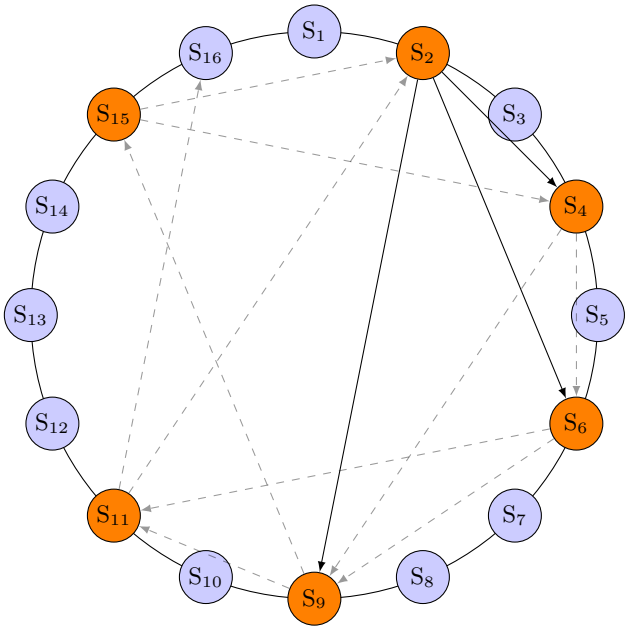

Figure 1: Chord DHT Diagram with 16 Nodes (6 Active)

1. **Servent (Node):** A process representing a user in the distributed system. Each servent interacts with the system through a command-line interface (CLI), enabling file management, friend interactions, and other operations. Servents are organized into a structured overlay network based on the Chord DHT.
2. **Virtual File:** A file stored within the distributed system. Files can be classified as:
   - **Public:** Accessible to all servents in the network. Public files are replicated across multiple servents for fault tolerance.
   - **Private:** Accessible only to the owner servent and their friends. Private files are **not** replicated.

3. **Friendship:** A bidirectional relationship between two servents. Friendships are established through a request-acceptance mechanism and allow servents to share and access each other's private files.

4. **Communication Protocol:** The system employs the **TCP/IP** protocol for reliable communication between servents, utilizing Java's serialization for object transmission.

5. **Algorithms:**

   - **Suzuki-Kasami Algorithm:** This algorithm facilitates the concurrent joining of multiple servents to the network. It ensures a smooth and controlled process, preventing conflicts and maintaining system stability during periods of growth.
   - **Failure Detection Mechanism:** A hybrid mechanism combining a **Buddy system** and a **Heartbeat system** to detect and handle servent failures. Each servent monitors two "buddy" servents (its predecessor and first successor). Token-holding servents (Suzuki-Kasami) send periodic heartbeats, "piggybacked" on messages sent for the Buddy system. The system can tolerate up to two simultaneous servent failures.
   - **File Replication Mechanism:** Public files are replicated for fault tolerance. The initial backup file is stored at the location determined by the Chord hash, and additional replicas are stored on a configurable number of successor servents, enhancing data availability and durability.

6. **Chord Distributed Hash Table (DHT):** The foundation of the system's architecture. Chord DHT provides a structured overlay network, enabling efficient routing, data lookup, and load balancing among servents. It maps virtual files and servents to unique identifiers, ensuring that data is distributed and quickly accessible throughout the network.

3 User Guide

This section provides step-by-step instructions on how to set-up, run, and interact with the distributed system using the command-line interface (CLI).

3.1 Bootstrap Server and System Configuration

- **Bootstrap Server:** A special servent designated to facilitate the joining of new servents to the network. New servents connect to the bootstrap server, specified by the bootstrap.ip and bootstrap.port properties in their configuration files, to obtain information about existing servents in the network, enabling them to integrate into the Chord DHT.

- **Configuration Files:** Both the bootstrap server and the servents must have their configuration files. They specify their data and provide information to other servents in the system on how to connect to them.

1. **Bootstrap Configuration File:** Bootstrap is configured by its configuration file ('bootstrap.properties') and any output from the bootstrap is logged in the 'bootstrap.err.txt' and 'bootstrap.out.txt' files.
   - `ip`: The IP address of the bootstrap server. (e.g. ip=localhost)
   - `port`: The port number on which the bootstrap server listens for incoming connections. (e.g. port=2000)
   - `chord_size`: The size of the Chord ring (number of possible IDs). (e.g. chord_size=64)
2. **Servent Configuration File:** Each servent is configured using a local configuration file named servent.properties. This file contains the following key-value pairs:
   - `root`: The root directory where the servent's files are stored. (e.g. root=root/s0)
   - `ip`: The IP address of the servent. (e.g. ip=localhost)
   - `port`: The port number on which the servent listens for incoming connections. (e.g. port=1000)
   - `bootstrap.ip`: The IP address of the bootstrap server. (e.g. bootstrap.ip=localhost)
   - `bootstrap.port`: The port number on which the bootstrap server listens. (e.g. bootstrap.port=2000)
   - `chord_size`: The size of the Chord ring (number of possible IDs). (e.g. chord_size=64)
   - `weak_failure_consistency`: Time in milliseconds before a servent is considered potentially failed (suspicious) if no response is received from it. (e.g. weak_failure_consistency=4000)
   - `strong_failure_consistency`: Time in milliseconds before a servent is declared dead if no response is received from it after the weak failure threshold. (e.g. strong_failure_consistency=10000)

- **Example Bootstrap Configuration File** ('bootstrap.properties'):

```
bootstrap.ip=localhost
bootstrap.port=2000
chord_size=64
```

- **Example Servent Configuration File** ('servent0.properties'):

```
root=root/s0
ip=localhost
port=1000
bootstrap.ip=localhost
bootstrap.port=2000
chord_size=64
weak_failure_consistency=4000
strong_failure_consistency=10000
```

3.2 Command Line Interface (CLI)

Once the servents are running, you can interact with the system using the CLI. Here are the available commands:

3.2.1 Available Commands

1. `add <filename> <public/private>`: Adds a file to the network. Specify the filepath in relation to the rood directory and whether it should be public or private.
2. `remove <filename>`: Removes a file from the network. This operation deletes the original file and all copies of the file across servents.
3. `open_file <filename>`: Opens a file locally if it's available on the current servent.
4. `view_files <ip_address>:<port>`: Requests files from another servent (based on that servent's ip address and port). Specify the servent address of the servent you want to request the files from.
5. `add_friend <ip_address>:<port>`: Sends a friend request to another servent.
6. `accept_friend <ip_address>:<port>`: Accepts a friend request from another servent.
7. `list_requests`: Displays a list of pending friend requests.
8. `list_friends`: Displays a list of your current friends.

3.2.2 Examples

- To add a public file named file.txt:
  - `add_file file.txt public`
- To remove a file named image.jpg:
  - `remove_file image.jpg`
- To send a friend request to servent with address localhost:1040:
  - `add_friend localhost:1040`

4 Technical Documentation

4.1 Servent Joining Process

The process of a new servent joining the distributed system is orchestrated by the bootstrap server and involves several steps to ensure the servent is correctly integrated into the Chord DHT and receives the necessary data and configuration information.

4.1.1 Initial Contact:

The new servent initiates the joining process by sending an informal "Hail" message to the bootstrap server. This message contains the new servent's IP address and listening port.

4.1.2 Bootstrap Server Response:

The bootstrap server responds with one of the following:

- If the new servent is the first to join the network, the bootstrap server sends back the string "first\n".
- If other servents are already in the network, the bootstrap server selects a random active servent and sends its IP address and port to the new servent.

4.1.3 NewNodeMessage:

The new servent sends a `NewNodeMessage` to the specified existing servent. This message does not contain any additional parameters beyond the basic message structure.

4.1.4 Collision Check and WelcomeMessage:

The existing servent that received the `NewNodeMessage` checks if the new servent's Chord ID collides with any existing servent's ID.

- If there's a collision, a `SorryMessage` is sent to the new servent, indicating that it cannot join.
- If there's no collision, the existing servent prepares and sends a `WelcomeMessage` to the new servent containing:
  - A map of key-value pairs that the new servent should store.
  - A map of file paths to `FileInfo` objects representing the files the new servent should store.

4.1.5 Chord State Update and Data Transfer:

The new servent receives the `WelcomeMessage` and uses it to initialize its Chord state and file storage. It sends an informal "New" message to the Bootstrap server to confirm that there was no collision. It then sends an `UpdateMessage` to its successor in the Chord ring, triggering a chain of updates to propagate the new servent's information throughout the network.

4.1.6 UnlockMessage:

After successfully integrating into the network, the new servent sends an `UnlockMessage` to the original sender of the `NewNodeMessage`, signaling the completion of the joining process and releasing the critical section held by the servent tasked with adding the new node during the joining process.

4.1.7 Key Classes and Handlers

- **ServentInitializer:** This class handles the initial communication with the bootstrap server and the sending of the `NewNodeMessage`.
- **BootstrapServer:** This class manages the list of active servents and responds to "Hail" and "New" messages from new servents.

- **NewNodeHandler:** This message handler processes `NewNodeMessage` messages, checks for collisions, and sends `WelcomeMessage` or forwards the `NewNodeMessage` as needed (to find the servent which should be responsible for adding this new servent).
- **WelcomeHandler:** This message handler processes `WelcomeMessage` messages, initializes the new servent's state, and triggers the `UpdateMessage` propagation.
- **UpdateHandler:** This message handler processes `UpdateMessage` messages, updates the local Chord state, and forwards the message to the next servent in the ring. The message propagation ends once the new servent receives the update message (i.e. when the update message goes through the entire Chord ring).

4.2 Suzuki-Kasami Algorithm

The Suzuki-Kasami algorithm is a fair token-based algorithm used to achieve mutual exclusion in the distributed system, ensuring that only one servent can access the critical section (e.g. insert a new servent into the system) at a time. The token is sent via special message that grants the holder permission to enter the critical section.

4.2.1 Token Structure:

The token consists of:

- **Q**: A queue (ConcurrentLinkedQueue) storing the Chord IDs of servents that have requested the token.
- **LN**: A map (ConcurrentHashMap) where `LN[j]` stores the sequence number of the most recently executed request by servent S_j.

4.2.2 Message Handling:

The Suzuki-Kasami algorithm relies on message handlers to process token requests, replies, and unlock notifications. These handlers are:

- **TokenRequestHandler:**
  - Receives `TokenRequestMessage` from other servents.
  - Updates the local `RN` array with the received sequence number.
  - If the servent holds the idle token and the request is valid (next in line), it sends the token to the requesting servent using a `TokenReplyMessage`.
- **TokenReplyHandler:**
  - Receives `TokenReplyMessage` from other servents.
  - If the message is for the current servent, it sets the local token and enters the critical section.

- Otherwise, it forwards the message to the next servent in the request chain.

- **UnlockHandler:**
  - Receives `UnlockMessage` when a servent has completed its integration into the network.
  - Signals the token-holding servent (itself) that it can exit the critical section.

4.2.3 Requesting the Critical Section:

1. **Token Absence:** If a servent S_i wants to enter the critical section but does not hold the token, it:
   - Increments its sequence number $RN_i[i]$ by 1.
   - Broadcasts a `TokenRequestMessage(i, sn)` to all other servents, where `sn` is the updated sequence number. This message is processed by the `TokenRequestHandler` on receiving servents.
2. **Receiving Request:** When a servent S_j receives a `TokenRequestMessage(i, sn)`, it:
   - Updates its own $RN_j[i]$ to the maximum of $RN_j[i]$ and `sn`.
   - If S_j holds the idle token and $RN_j[i] = LN[i] + 1$ (meaning S_i is the next in line), it sends the token to S_i using a `TokenReplyMessage`.

4.2.4 Executing the Critical Section:

Servent S_i enters the critical section upon receiving the token. While its executing the critical section, it will hold the token.

4.2.5 Releasing the Critical Section:

1. **Updating LN:** After executing the critical section, S_i updates $LN[i]$ to $RN_i[i]$.
2. **Updating Token Queue:** For each servent S_j not in the queue, if $RN_i[j] = LN[j] + 1$, S_i adds S_j to the token queue Q.
3. **Token Transfer:** If the queue Q is not empty, S_i removes the head of the queue (next servent ID) and sends the token to that servent using a `TokenReplyMessage`.

4.2.6 Key Mechanisms

- **Request Numbers (RN):** Each servent maintains an array (RN) to track the highest sequence number received in a request from each other servent. This helps distinguish outdated requests.
- **Last Numbers (LN):** The token's LN array allows servents to determine if another servent has an outstanding request for the critical section.
- **Token Queue (Q):** The token's queue ensures fairness by maintaining the order of requests.

4.2.7 Code Implementation Notes:

- **SuzukiKasamiMutex:** This class manages the token, request/reply messages, and the critical section logic.
- **Token:** Represents the token object, holding the queue Q and the array LN.
- **TokenRequestMessage, TokenReplyMessage, UnlockMessage:** Used for communication between servents during token requests, transfers, and unlock notifications.
- **TokenRequestHandler, TokenReplyHandler, UnlockHandler:** Responsible for processing the respective message types.

4.3 Failure Detection

The distributed system employs a robust failure detection mechanism to identify and handle servent failures, ensuring the system's continuous operation and data integrity. The mechanism relies on a combination of buddy monitoring, heartbeats, and message-based failure confirmation and recovery, facilitated by the **FailureDetector** class and various message handlers.

4.3.1 Buddy System

Each servent maintains a list of its two buddies: its *predecessor* and *successor* in the Chord ring. These buddies are responsible for monitoring the servent's liveness. The `FailureDetector` class manages the buddy list and tracks the last response times from each buddy. The core of this Buddy system implementation are Ping and Pong messages and their handlers. Simply said, a servent sends a `PingMessage` to its buddies and they respond with a `PongMessage` if they are alive, and the response time is then updated. After that the servent checks whether its buddies responded before any failure consistency threshold. If the weak threshold is exceeded then the servent asks another servent to confirm whether its buddy is alive. If the strong threshold is exceeded then the servent declares that its buddy is dead and initiates the recovery process;

4.3.2 Heartbeat System

The token-holding servent sends periodic "heartbeats" piggybacked on Ping messages sent to its buddies. The heartbeats indicate whether the servent currently holds the token, which is crucial for selecting a new token holder if the current one fails. The `PingMessage` requests are sent at 1000ms intervals.

4.3.3 Failure Confirmation

If a servent suspects a buddy has failed (after the weak failure consistency threshold of 4000ms), it sends a `CheckSusMessage` to its other buddy. The receiving buddy then sends a `UAliveMessage` to the suspicious servent. If the suspected servent is still alive, it responds with an `AmAliveMessage`, which is relayed back to the original sender via an `IsAliveMessage`. This allows the original sender to update the last response time from the suspicious servent. If the

suspicious servent fails to respond within the strong failure consistency threshold (10000ms), the original sender declares it dead and initiates the recovery process.

4.3.4 Recovery Process

When a servent is declared dead, the FailureDetector class of the servent whose buddy has died:

1. **Notifies Bootstrap Server:** Informs the bootstrap server about the dead servent.
2. **Restructure Message:** Broadcasts a `RestructureSystemMessage` to its successors in the Chord ring, notifying them of the failed servent. This message is handled by the `RestructureSystemHandler`.
3. **Chord Restructuring:** The `RestructureSystemHandler` removes the failed servent from the Chord ring and updates its own successor/predecessor lists. The system's Chord state is then updated to reflect the new topology.
4. **Token Holder Recovery (If Applicable):** If the failed servent was the token holder, its buddies compete to become the new token holder. They declare themselves as token holders by sending a `NewTokenHolderMessage` to the network. `The NewTokenHolderHandler` resolves any conflicts (based on timestamps) and ensures a single new token holder is selected. In an edge-case where successors of a node which declared itself as a new token holder are dead, there is a mechanism such that when receiving a `NewTokenHolderMessage`, if the timestamp is less than the receiving process then send a direct `YouWereSlowMessage` to the servent which also declared itself as the token holder.

4.3.5 Failure Threshold and Limitations

The system can tolerate up to two simultaneous servent failures. If more than two servents fail concurrently, the system's behavior becomes unpredictable, and it **may not** be able to recover the token, depending on which servents failed.

4.3.6 FailureDetector Class

The FailureDetector class is the core component responsible for:

- Managing the buddy list and tracking last response times.
- Sending `PingMessage` to buddies.
- Handling failure confirmation (`CheckSusMessage`, `UAliveMessage`, `AmAliveMessage`, `IsAliveMessage`).
- Initiating the recovery process when a servent is declared dead.
- Updating the system's state after restructuring.

4.3.7 Code Implementation (Message Handlers):

- **PingHandler:** Sends `PongMessage` to the sender and since `PingMessage` acts as a heartbeat for token-holding servents, it also sets the id of the token-holding servent if the sender is indeed the token-holding servent.
- **PongHandler:** Handles `PongMessage` replies, updates last response times, and for faster restructuring, it triggers restructuring if it hasn't began already based on the information provided by the sender.
- **CheckSusHandler:** Sends `UAliveMessage` to the suspicious servent.
- **UAliveHandler:** Handles `UAliveMessage` and responds with `AmAliveMessage`.
- **AmAliveHandler:** Handles `AmAliveMessage` and forwards an `IsAliveMessage` to the original sender.
- **IsAliveHandler:** Handles `IsAliveMessage` and updates last response times for the suspicious servent.
- **NewTokenHolderHandler:** Handles `NewTokenHolderMessage`, resolving conflicts and determining the new token holder.
- **YouWereSlowHandler:** Handles `YouWereSlowMessage`, telling the servent that it was slower than the sender when declaring itself as the new token holder.
- **RestructureSystemHandler:** Handles `RestructureSystemMessage`, removing the failed servent and updating the Chord ring.

4.3.8 Additional Notes:

- The system does not explicitly recover data from failed servents. Any data stored exclusively on a failed servent is considered lost.
- The failure detection mechanism ensures high availability and fault tolerance by quickly identifying and reacting to servent failures.
- The use of heartbeats and buddy monitoring provides redundancy and helps prevent false positives in failure detection.

4.4 Data Replication

To ensure data availability and durability in the face of potential servent failures, the distributed system implements a replication mechanism for public files. This mechanism ensures that multiple copies of each public file are maintained across different servents in the network, leveraging the Chord DHT for efficient storage and retrieval.

4.4.1 Replication Factor

The replication factor, currently set to 3, determines the number of replicas created for each public file. It represents a trade-off between data redundancy and storage overhead. A higher replication factor enhances data availability and resilience to failures but consumes more storage resources.

4.4.2 Replica Placement and Backup Process

When a *public* file is added to the system using the `add_file` command in the CLI, the file is saved on the servent which executed the command and the following process is triggered by the `FileManager` and `BackupFileHandler` classes:

1. **File Hashing:** The system calculates the Chord hash of the file's path using the `FileManager.fileHas(String filePath)` method. This hash value determines the primary storage location of the initial backup file on the Chord ring.

2. **Primary Storage:** The initial file backup is stored on the servent whose ID is closest to the calculated hash value. If the owner of the file is already the closest servent, **the backup is skipped**.

3. **Backup Initiation:** The `FileManager`'s backupFile method initiates the backup process by sending a `BackupFileMessage` to the successor servent of the primary storage location. This message contains the file information and the file hash.

4. **Backup Propagation:** The `BackupFileHandler` in each servent receives the `BackupFileMessage`. If the servent's ID is closest to the file hash, and it doesn't already have a copy, it stores the file locally. The `backupId` is incremented, and the message is forwarded to the next successor servent if the replication factor hasn't been reached. This process continues until the desired number of replicas is created.

4.4.3 Data Consistency

The current implementation of the data replication mechanism prioritizes availability over strong consistency. And as a result, there are no explicit measures in place to ensure strict consistency among replicas. In the event of concurrent updates or failures, temporary inconsistencies may arise between replicas. However, the system is designed to be eventually consistent, meaning that replicas will eventually converge to a consistent state as updates propagate throughout the network. Given the demonstrative nature of this system and its focus on showcasing the core concepts of distributed file storage and fault tolerance, the absence of a strong consistency mechanism is a deliberate design choice. It allows for a simpler implementation while still highlighting the challenges and trade-offs involved in achieving data consistency in a distributed environment.

4.4.4 Additional Considerations:

- The replication factor is currently fixed at 3. Future enhancements could allow this to be configurable to adapt to different storage and availability requirements.

- The system could potentially implement a more sophisticated data consistency mechanism if strong consistency is a critical requirement.

5 Communication Protocol

The distributed system relies on a well-defined communication protocol to facilitate inclusion of new servents into the network, interactions between servents, manage the Suzuki-Kasami token, perform failure detection, and replicate data. The following message types are used for these purposes:

5.1 Servent Joining Messages

1. **Hail Message (Informal):** Sent by a new servent to the bootstrap server to initiate the joining process.
   - `String`: IP address of the new servent.
   - `int`: Port of the new servent.
2. **NewNodeMessage:** Sent by the servent that wants to join to the servent it got from the bootstrap server to inform other servents about the new servent joining the network.
3. **WelcomeMessage:** Sent by a servent to the joining servent, providing information about the new servent's position in the Chord ring and the data it should store.
   - `Map<Integer, Integer> values`: A map of key-value pairs that the new servent should store.
   - `Map<String, FileInfo> files`: A map of file paths to `FileInfo` objects, representing the files the new servent should store.
4. **New Message (Informal):** Sent by a new servent to the bootstrap server to confirm that there was no collisions.
   - `String`: IP address of the new servent.
   - `int`: Port of the new servent.
5. **UpdateMessage:** Sent by the joining servent to its successor, with a goal to propagate information about the new servent throughout the network, updating the Chord state of each servent and eventually returning to the joining servent.
   - `String text`: A comma-separated list of IP addresses and ports of servents in the network.
6. **SorryMessage:** Sent by a servent to a new servent if there is a collision (same Chord ID) with an existing servent, indicating that the new servent cannot join the network.

5.2 Failure Detection Messages

1. **PingMessage:** Sent periodically by a servent to its buddies (successor and predecessor in the Chord ring) to check their liveness.

- `boolean hasToken`: Indicates whether the sending servent currently holds the Suzuki-Kasami token. This serves as a heartbeat mechanism for the token holder.

2. **PongMessage:** Sent in response to a `PingMessage`, confirming the servent's liveness.
    - `List<String> deadServents`: A list of IP addresses and ports of servents detected as dead since the last ping. This information helps propagate knowledge of failed servents throughout the network.
3. **CheckSusMessage:** Sent to a buddy servent when a servent suspects another servent of failure but hasn't yet confirmed it as dead.
    - `String susIp`: IP address of the suspected failed servent.
    - `int susPort`: Port of the suspected failed servent.
4. **UAliveMessage:** Sent to a potentially failed servent to inquire about its status. This message is triggered by a `CheckSusMessage`.
    - `int concernedId`: Chord ID of the servent that initiated the failure check (`CheckSusMessage`) and is waiting for confirmation.
5. **AmAliveMessage:** Sent by a servent in response to a `UAliveMessage`, confirming that it is still active.
    - `int concernedId`: Chord ID of the servent that initiated the failure check (`CheckSusMessage`).
6. **IsAliveMessage:** Forwarded by the buddy servent (who received the `AmAliveMessage`) back to the original requester (`concernedId`), confirming that the suspected servent is indeed alive.
    - `String aliveIp`: IP address of the confirmed alive servent.
    - `int alivePort`: Port of the confirmed alive servent.
7. **NewTokenHolderMessage:** Sent by a servent to declare itself as the new token holder after the failure of the previous holder. This message is propagated to other servents to ensure they are aware of the new token holder.
    - `long declaredTime`: The timestamp when the servent declared itself as the new token holder, used for resolving conflicts if multiple servents attempt to become the new holder simultaneously.
8. **YouWereSlowMessage:** Sent by a servent which declared itself as a new token holder as a fail-safe if the successors of the servent are dead, in which case the `NewTokenHolderMessage` won't be able to reach the servent which also declared itself as the token holder and we could end up with multiple token-holding servents.
9. **Dead Message (Informal):** Sent to the bootstrap server so it can know which servents to remove from the list of active servents.
    - `String deadIpAddress`: IP address of the failed servent.

- `int deadPort`: Port of the failed servent.

10. **RestructureSystemMessage:** Sent to notify other servents about a failed servent and trigger the restructuring of the Chord ring.
    - `String deadIpAddress`: IP address of the failed servent.
    - `int deadPort`: Port of the failed servent.

5.3 Suzuki-Kasami Mutex Messages

1. **TokenRequestMessage:** Broadcast by a servent to all other servents when it wants to enter the critical section but does not hold the token.
    - `int j`: Chord ID of the requesting servent.
    - `int n`: Sequence number of the request, used to differentiate between new and outdated requests.
2. **TokenReplyMessage:** Sent by the current token holder to grant the token to the next requesting servent in the queue.
    - `Token token`: The Suzuki-Kasami token object, containing the request queue ('Q') and the array of last executed request numbers ('LN').
    - `int requesterId`: Chord ID of the servent that is granted the token.
3. **UnlockMessage:** Sent by the new servent to the original sender of the `NewNodeMessage` to signal that it has completed its integration into the network, allowing the token-holding servent to exit the critical section.

5.4 File Management Messages

1. **BackupFileMessage:** Sent to initiate the backup process for a public file. It is propagated along the Chord ring until the desired replication factor is reached.
    - `FileInfo fileInfo`: Metadata and content of the file to be backed up (file name, type, content, visibility, owner ID, backup ID).
    - `int fileHash`: Chord hash of the file path, used to determine the storage locations of the original and replicas.
2. **AskRemoveFileMessage:** Sent to request the removal of a file and its backups from the network.
    - `String filePath`: Path of the file to be removed.
    - `int fileHash`: Chord hash of the file path.
    - `int backupId`: ID of the backup to be removed (goes up to the replication factor).
    - `int ownerId`: Chord ID of the file owner.
3. **AskRemoveOriginalFileMessage:** Sent to request the removal of the original file from its owner.

- `String filePath`: Path of the file to be removed.
- `int ownerId`: Chord ID of the file owner.
- `List<Integer> visited`: List of servents that have already processed this message (to prevent infinite loops).

4. **AskViewFilesMessage:** Sent by a servent to request a list of files from another servent.
   - `String ip`: IP address of the servent to which the request is being sent.
   - `int port`: Port of the servent to which the request is being sent.
   - `int ogKey`: Chord ID of the servent initiating the request.
5. **TellViewFilesMessage:** Sent in response to an `AskViewFilesMessage`, containing the requested list of files.
   - `int ogKey`: Chord ID of the servent that initiated the request.
   - `Map<String, FileInfo> fileMap`: A map of file paths to `FileInfo` objects, representing the files available on the responding servent.

5.5 Friend Management Messages

1. **AddFriendRequestMessage:** Sent by a servent to request friendship with another servent.
   - `int toBefriendHash`: Chord ID of the servent receiving the friend request.
2. **AddFriendResponseMessage:** Sent in response to an `AddFriendRequestMessage`, indicating acceptance or rejection of the friend request.
   - `int requesterHash`: Chord ID of the servent that initiated the friend request.

References

[1] Kshemkalyani, A. D., & Singhal, M. (2008). *Distributed Computing: Principles, Algorithms, and Systems* [pp. 336-339, 688-695]. Cambridge University Press.