Project 2: Content Distribution

Design Document

– Oorjit Chowdhary (oorjitc@uw.edu), EE 419/565 - Spring 2025

Manual Grading Request: My code does not pass the Gradescope tests, but I have tested it locally and it works as expected. I would like to request manual grading for my code.

This document outlines the design for a distributed UDP Content Server system. Each server instance (node) maintains awareness of the network topology through peer-to-peer communication. The system utilizes keepalive messages to monitor the status of direct neighbors and Link-State Advertisements (LSAs) to propagate network connectivity information. This allows each node to build a complete network map and calculate shortest paths to other nodes using Dijkstra's algorithm.

The system can be composed of any arbitrary number of server instances, though the design has not been tested with more than 6 nodes. Each server instance runs the `ContentServer` class, which is capable of perforing the following concurrent operations: (1) sending and receiving KeepAlive messages to/from direct neighbors, (2) sending and receiving Link-State Advertisements (LSAs) to/from all nodes in the network, (3) building and updating a local representation of the entire network topology, and (4) responding to the `uuid`, `neighbors`, `map`, `rank`, `kill` commands from the client.

KeepAlive Implementation

KeepAlive messages are sent to direct neighbors at periodic intervals. Each server instance maintains a list of its direct neighbors and their last known and last seen statuses. If a neighbor does not respond within a specified timeout period, it is marked as dead. The system uses a heartbeat mechanism to ensure that all nodes are aware of the current state of their direct neighbors.

This is implemented using an independent `keepalive_loop` thread that sends KeepAlive messages as well as monitors for neighbor node timeouts. The receiving side of `KeepAlive`s is handled by another `receive_loop` thread, which is shared between `KeepAlive` and `LSA` messages.

Using the `keepalive_loop`, each node calls the `send_keepalive` method every 3 seconds, which sends a minimal JSON message containing the node's name and UUID to all its direct neighbors. The `receive_loop` thread listens for incoming messages and updates the last seen time and alive status of the sender in the `neighbors` dictionary. If it receives a `KeepAlive` message from unknown node, it simply ignored it.

For failure detection of nodes, the `keepalive_loop` thread compares the current time with the last seen time of each neighbor after sending `KeepAlive`s. If the

difference exceeds 9 seconds, the neighbor is marked as dead and removed from the local representation of the network topology. Following this, the thread also sends an LSA to all nodes in the network to inform them of the change in the network topology.

Link-State Advertisement Implementation

Link-State Advertisements allow each node to advertise its direct, alive neighbors and their associated metrics to all other nodes in the network. Each LSA is associated with a sequence number, which is incremented each time a new one is sent. This enables each node to build a complete, up-to-date map of the network and calculate the shortest paths to other nodes using Dijkstra's algorithm.

Similar to the `KeepAlive` messages, the `LSA` messages are sent using an independent `lsa_loop` thread. The `lsa_loop` thread sends LSAs to all nodes in the network every 5 seconds by calling the `send_lsa` method. In addition to the periodic 5 second LSAs, an LSA message is also sent when a direct neighbor's status changes, i.e. it comes alive or dies or when a new node joins the network. This ensures that all nodes are aware of the current state of the network and can update their local maps accordingly.

The `send_lsa` method constructs a JSON message containing the sender's UUID and name, the LSA's sequence number, and a dictionary of the sender's direct, alive neighbors with their associated distance metrics. At this point, the sender's local representation of the network topology is also updated to reflect the latest alive neighbors and their metrics. Finally, the LSA is sent to all nodes in the network using a helper `broadcast` method.

On the receiving side, the `receive_loop` thread listens for incoming LSA messages and updates the local representation of the network topology. It also checks the sequence number of the received LSA against its local representation. If the received LSA is more recent, it updates its sequence number tracker, updates the local representation of the network topology, and determines the original sender of the LSA as LSAs are also forwarded. Finally, it sends the LSA to all nodes except the original sender (to avoid loop), ensuring that all nodes in the network receive the most up-to-date information.

Libraries Used

- `socket`: For UDP communication between nodes.
- `threading`: For concurrent execution of the `keepalive_loop`, `lsa_loop`, and `receive_loop` threads.
- `json`: For encoding and decoding JSON messages.
- `time`: For managing timeouts and delays in the system.
- `argparse`: For parsing command-line arguments.
- `sys`: For handling system-level operations, like graceful shutdown.
- `logging`: For logging messages and errors to stdout.

Extra Capabilities

N/A.