a. Your keepalive message implementation.

For my keepalive message implementation, I followed basically all the guidelines in the writeup (the example where keepalive interval is 3 s, i.e. 3 seconds in between each keepalive, and 3 missed keepalives means assume inactive). So in order to do this, I have a neighbor class that for each neighbor stores whether that neighbor is_active flag and 1) timestamp of when last keepalive, 2) how many keepalive misses.

When an instance of my content_server.py starts running, it'll spawn off keepalive_sender thread, keepalive_checker thread, receiver_thread for ensuring keepalives are being checked for all neighbors, and keepalives are being sent to all neighbors as specified with time above. For keepalive_checker, check if need to change the timestamp of when last keepalive and increment the keepalive misses count as necessary. In receiver_thread, spawn off another thread for processing a keepalive message, and for handle_keepalive, mark that neighbor as active/update timestamp of when last keepalive/reset keepalive misses.

b. Your link-state advertisement implementation.

For LSA, I followed basically all the guidelines in the writeup (send LSA periodically i.e. every 3s to all neighbors, send LSA when new neighbor added, send LSA when neighbor detected as inactive). So in order to have this, I have a ContentServer class that stores the current network_map as dict: name -> dict(name -> metric), node_names as dict: uuid -> name, lsa_sequence as dict: uuid -> last seq #, own_sequence = 0, lsa_timestamp as dict: uuid -> last lsa time, lsa_timeout = 10. That way, when I also spawn off lsa_sender thread at start of program, I can cleanup_stale_lsas (based upon their last lsa time and lsa_timeout) as well as send lsa to all neighbors. For the lsa message itself that is sent to all neighbors (not just active ones), it contains the self.uuid, self.name, self.own_sequence, and the neighbors info, where neighbors info ONLY contains info on active neighbors. (So lsa message is sent to all neighbors but info about the neighbors inside the lsa message only is about active neighbors)

Then for receiving/handling lsa's, the receiver_thread from before spawns off another thread for processing a lsa message. Then it will check the lsa's sequence_num to discard lsa message/update own server info/forward to other neighbors as necessary.

I have helper functions such as trigger_lsa_broadcast which is a wrapper for send_lsa_to_all. Send_lsa_to_all (where LSA message is actually crafted and send_message done to all neighbors).

c. Libraries used (optional; name, version, homepage-URL; if available).

Libs used: all python native stuff: **sys, json, threading, socket, time**

d. Extra capabilities you implemented (optional, anything which was not specified in this doc).

Extra capabilities: realized that locking is probably not needed, but used threading.RLock() to create 2 locks, one for the network_map and neighbors. This was implemented because of the many threads that spawn off modifying and working with the network_map / node_names / lsa_sequence, it would be good to protect access those just in case different threads handling stuff about LSA / keepalive might cause concurrency issues.

Same argument for neighbors lock, because want to ensure self.neighbors is being changed by one thread at a time.