PL3: WebSocket Chat Federation

The soft deadline for this programming lab is July 26. The hard deadline is August 31.

| **Submission and group work:** The requirements for the submission and group work are as for PL2.

Your task is to extend your implementation of the *dnChat* server to allow connections to and from other *dnChat* servers to create a *network* of servers. The clients (from PL1) shall observe no functional difference between being connected to (a) a single server with a set of directly connected users as in PL2 and (b) a server that is part of a network of servers with the same set of users connected to different servers. A connection from one server to another is initiated when one server reads a line from standard input starting with `connect`, followed by a space, the other server's host name or IPv4 address, and optionally another space followed by a port number. The communication *between servers* shall follow the basic principles of the *dnChat* protocol as specified in the previous labs, including the use of the WebSocket protocol and the same default port number, but shall use only the following set of messages:

`SRVR`
Only valid as the first message sent over a server-to-server connection by the server that initiated the connection. Must be sent by that server before sending any other messages. The number must be zero. (This message allows a server to tell whether a new connection is from a client or from another server.)

`ARRV`
As in the previous labs, but with the addition of a fourth line that contains the *hop count*, i.e. the minimum number of *other* servers that messages for the specified user, when handed to this server, must pass through before reaching their destination. (That is, the hop count is zero for users on clients connected directly to this server, one for users on clients connected to a server that has a direct connection to this server, and so on. When the hop count from a server changes, it may send new `ARRV` messages.)

`LEFT`
As in the previous labs. Indicates that the specified user is no longer reachable via this server.

`SEND`
As in server-initiated communication in the previous labs. If the second line is `*`, the message shall be broadcast throughout the server network using controlled flooding. Otherwise, is shall be sent (unicast) over the shortest path (in terms of number of intermediate servers) to the specified user's client.

`ACKN`
As in server-initiated communication in the previous labs. Shall be unicast like a `SEND` message (see above).

Servers do not send `OKAY` or `FAIL` messages to each other. When an invalid message (i.e. a malformed one, one with an invalid number, ...) is received, the server-to-server connection shall be closed immediately.

Hints and details: Test your implementation with servers connected in cyclic and acyclic topologies of at least four servers. Check that broadcast and unicast messages as well as acknowledgments for these messages are delivered correctly, and that the list of connected users is always correct on all clients (modulo network delays). To achieve the latter, you need to handle the case where a server disconnects from another server without first sending `LEFT` messages. When this happens, in-transit (chat) messages can get lost, but you do not need to implement reliable data transfer mechanisms to handle this case—it is acceptable that the message just gets lost since chat messages are acknowledged end-to-end anyway. A well-behaved server, however, will send `LEFT` messages for all users it knows about and wait a second or two before disconnecting from another server. Your implementation shall be well-behaved. Think about what type of routing algorithm you have to implement, and what information you consequently have to store in addition to your current forwarding table mapping user numbers to connections. You can also assume in server-to-server communication that numbers are uniquely generated, such that, for example, there cannot be two chat messages with the same number in the network at the same time.