P2P Networking Over UART

For Microcontrollers and Embedded Applications

By: Bowen Liu

Background

Problem Background

- Microcontrollers are ubiquitous, and it's the heart of all electronic gadgets and devices.
- Communication between multiple microcontrollers aren't always easy.
- Additional communication peripherals (eg: WiFi modules) are often needed and may add significant amount of design complexity and cost.
- These peripherals are often "overkill", since a lot of projects only require to send periodic messages to each other in a closed network.
- Instead of adding additional components for communication, why don't we just use something that's already present in all microcontrollers?

UART

- In early days, computer used the dial-up modems that connects to the serial port to access the internet!
- Virtually all microcontrollers today have built-in serial ports accessible by the application processor
- Also known as **UART** (Universal Asynchronous Receiver/Transmitter)
- **Asynchronous**: Send and receive data whenever it's needed.
- **Full-duplex**: Physical interface can receive and transmit at the same time.
- Async and full-duplex: Great for networking!
- **Easy to interface**: 1 wire for transmit (TX), 1 wire for receive (RX). Cheap physical medium!
- UART is a raw physical interface: Messages are transmitted as individual raw bytes, sequentially one bit at a time.

End-to-End Problem with UART

- The common UART interface only allows two end devices to be connected together. Multi-way communication not possible.
- Solution: Use a **packet switching** network model by overlaying a link layer on top of UART
- Network will be consisted of switches and end nodes
- **Switches** acts as store-and-forward broadcasting medium. It's simply a microcontroller that have multiple UART ports.
- Each of its UART port can connect an end node, or to another UART switch.
- Switches are transparent during normal data transfer. An end node only needs to tag its ID and the destination end node's ID to deliver a message.
- All switch and end nodes keeps track of routing tables to remember which end nodes are in the network, and which are reachable at each link.

Project Abstract

Project Abstract

- Provides a network stack to allow multiple microcontrollers to communicate with each other using only UART, an end-to-end protocol.
- The network stack aims to add structured communication, peer-to-peer and ad-hoc capabilities
- Supports dynamic memberships; suitable for wireless links
    - Eg: Microcontrollers interconnected wirelessly over UART, using Bluetooth SPP
- Ideally, generalize the network stack so most of its backbone capabilities can be easily adapted to other end-to-end protocols
    - P2P Network over LPT port maybe?

Design Considerations

- Microcontrollers have **small amount of system memory** available.
    - A node cannot buffer large number of unprocessed packets; packets with large payloads.
    - Memory management must be strict and tightly controlled
- UART link speed is very **slow**.
    - Devices commonly use 9600bps (1.2KB/s) for UART communications.
    - Our microcontrollers support up to 115200 bps (14.4KB/s).
    - A 1MB file would take at least 70 seconds to transmit at this speed
- Only **small number of peers** are present to form a **local area network**
    - The network forms a closed circuit to allow a group of application-specific microcontrollers to communicate
    - Microcontrollers often have **poor processing speed**. This plus the limited amount of memory prevents a network from scaling well.

Network Topology

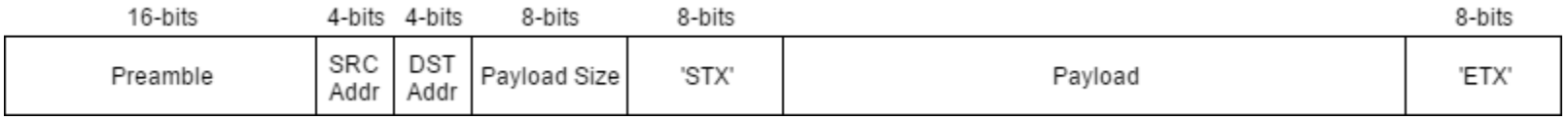
- Topologies supported:
  - **Star:** Network with only 1 switch; all nodes have same distance between each other
  - **Spanning Tree:** Network with multiple switches daisy-chained; nodes have different distances between each other
  - **End-to-End**: The two ends of a link are end nodes, forming a direct connection.
- Network Assumptions:
  - End nodes only have 1 link, and switches have > 1 links.
  - Only one path exists between any two end nodes.
  - The network must not contain cyclic paths between the switches

Network Stack

- **Physical Layer**
  - Essentially the hardware UART implementation.
  - Not much work is done here for the project, except for setting control registers and interrupts
- **Link Layer** (Project Focus)
  - Frame Synchronization: frames are received several bytes at a time instead of a complete frame. Must assemble a complete raw frame before they can be processed.
  - Provide queueing service for sending and receiving frames; makes transmission asynchronous.
  - Frame Routing
  - Membership Management
- **Transport Layer** (Not implemented)
  - Large packet fragmentation
  - Reliable streams
- **Application Layer**
  - Defined by the user. Free to use any services provided by any network layer directly

Frame Format

- **Frames** are referred to as the link-layer packets.
- **Preamble** denotes the start of a new frame's header.
- Two values of Preamble are defined.
  - 0x81CD is used for message frames (used by user's application)
  - 0x81C3 is used for control frames (used for routing).
  - 0x81CD = 'SOH' + M 0x81C3 = 'SOH' + C
  - Both characters had their 8th bit set to 1 to balance the preamble's hamming weight.
- **SRC** and **DST** denotes the node ID of the frame's sender and intended receiver.
  - ID = 0 indicates the frame is sent from/to the link layer. Used by control frames only.
  - ID = 15 indicates a broadcasting frame. All end nodes will receive this frame.
  - 4-bits => 14 end nodes in the network. Additional can be supported by expanding the ID width
- **SIZE** field indicates the length of the payload in bytes. This could be anywhere between 0 to 255 bytes.
- The payload is preceded with a **'STX'** character, and proceeded with an **'ETX'** character. These two characters delimits the payload, and is used to check if the payload of an expected size was received.
- In a message frame, the **payload** itself typically contains raw data either sent by a user's application, or from a higher network layer.
- In control frames, the payload is often used to contain *secondary header fields* needed by specific control functions.



Routing and Membership Management

Routing

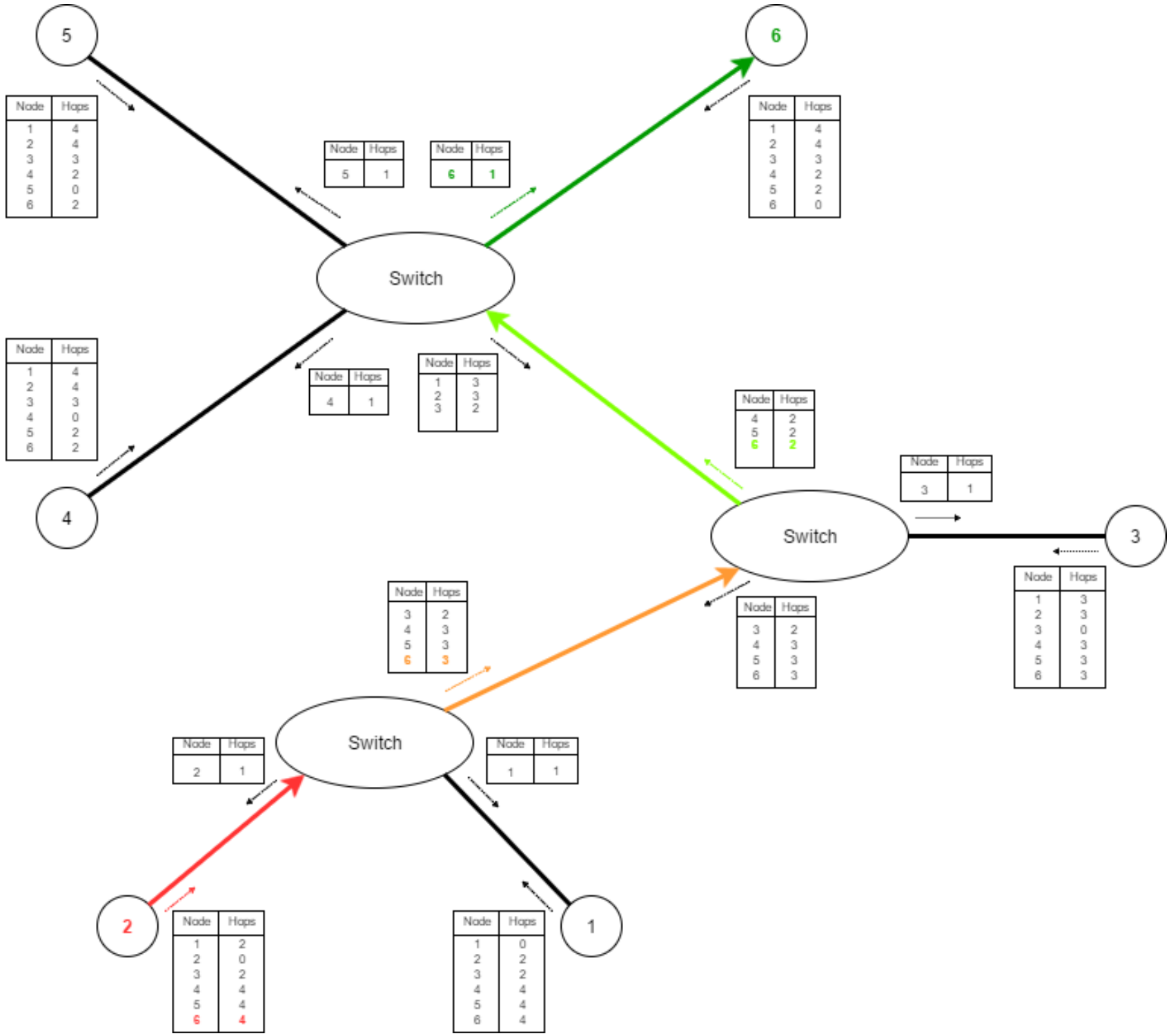
- All nodes and switches keeps track of a **routing table**. This table contains a list of all end-nodes in the network, and number of hops away from the current node.
- Hop count of 0 means invalid node or self. Only switches can have nodes with hop count of 1, indicating a leaf end-node connected to the switch.
- Routing is handled exclusively by switches. A frame is delivered from the sender to a receiver by propagating the messages through the switches sequentially.
- At each switch, the switch check which link the destination is reachable at.
- If unreachable, the frame is dropped.
- Worst case routing performance: **O(n) hops**, when the entire network is joined by a linear path, made of daisy-chained switches.

Routing Example

Sending a frame from Node 2 to Node 6, in a spanning tree network with 3 switches and 6 nodes:

1. Node 2 finds Node 6 in the routing table for its only link. Node 2 tags the frame's *source* field with "2" and *destination* field with "6", and transmits it out the link.
2. The Switch 1 receives the frame from Node 2. Switch 1 finds Node 6 to be reachable from its second link, and forwards the frame.
3. Switch 2 receives the frame from Switch 1. Switch 2 finds Node 6 to be reachable from its first link, and forwards the frame.
4. Switch 3 receives the frame from Switch 2. Switch 3 finds Node 6 to be reachable from its third link (from the left), and forwards the frame.
5. Node 6 receives the frame forwarded by Switch 3, and sees that the frame originated from Note 2.

Note that none of the switches or nodes knows the exact path a frame took to get to the destination, or its current location. It only knows the source and intended destination of the frame.


5
6
4
3
2
1
Switch
Switch
Switch
Node Hops
1 4
2 4
3 3
4 2
5 0
6 2
Node Hops
5 1
Node Hops
6 1
Node Hops
1 4
2 4
3 3
4 0
5 2
6 2
Node Hops
4 1
Node Hops
1 3
2 3
3 2
Node Hops
4 2
5 2
6 2
Node Hops
3 1
Node Hops
1 3
2 3
3 0
4 3
5 3
6 3
Node Hops
3 2
4 3
5 3
6 3
Node Hops
3 2
4 3
5 3
6 3
Node Hops
2 1
Node Hops
1 1
Node Hops
1 2
2 0
3 2
4 4
5 4
6 4
Node Hops
1 0
2 2
3 2
4 4
5 4
6 4

Membership Management

- The network protocol supports dynamic membership. Nodes can join and leave the network at any given time.
- Routing tables of all nodes and switches are updated dynamically based on joining/leaving events.
- A series of *control messages* has been created to facilitate these tasks.
- When a new node initializes, it first sends a **HELLO message** to the other end of the link. The other end of the link will reply if it's a SWITCH or a NODE.
- If both ends are NODE, the two nodes will mutually configure their network to operate in *direct connection mode.*
- Otherwise, the new node sends a **JOIN message** to the switch.

JOIN Message

- The JOIN message contains a hop count in its payload to indicate how many times the message has been forwarded.
- When a switch or end node receives a JOIN message, it adds the message's source ID to its routing table, along with the hop count in the payload.
- When a switch receives a JOIN, it will additionally increment the hop count in the payload, and forward the JOIN message out of its links, except for the link that received the JOIN.
- If a switch's end node sent the JOIN message, the switch will also send an complete routing table to the new node.
- Once all nodes and switches has received the JOIN message, every node/switch (including the newly joined node) will have an updated and consistent routing table.

LEAVE message

- A node may also leave the network voluntarily by sending a **LEAVE message** to the other end of its link.
- The process is the same as JOIN, except all recipients of a LEAVE message marks the source ID's hop count to 0 in its routing table; indicating the node is no longer valid in the network.

Other Control Messages

- **REQRT**: A node can request another node's (or the switch on the other end of the link) routing table by sending a **req**uest **r**outing **t**able message. (pic?)
- **RTBLE**: This message is used to send a complete routing table to a node. The switch uses this message to pass the latest routing table to a newly joined node. This message is also used to reply REQRT messages.

Note that the handler for all control messages can be overridden by the user, so application specific actions can be added.

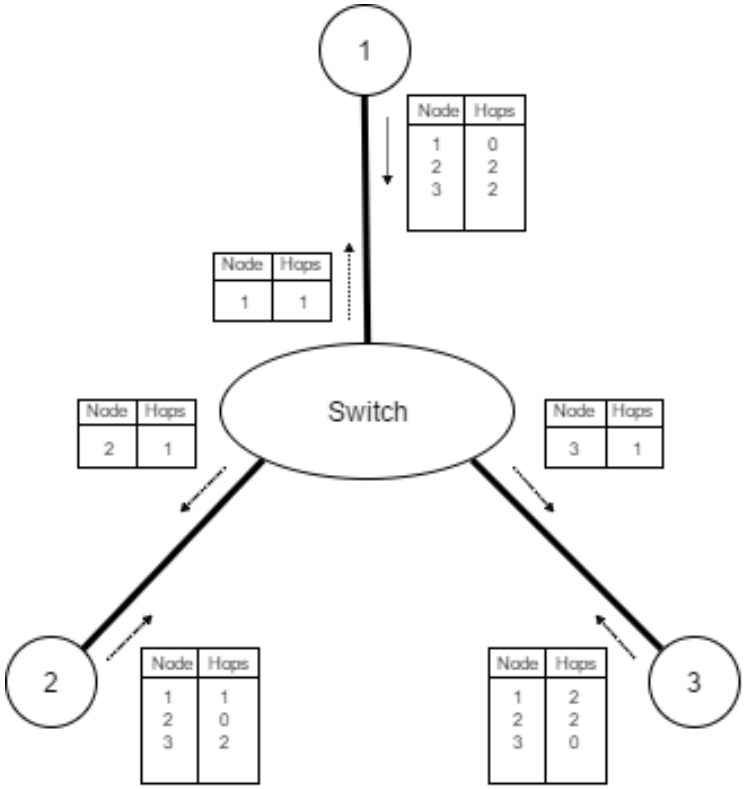
Active Monitoring

- An end node can fail and quietly leave the network without being known.
- All switches in the network monitors the liveness of all end nodes that are connected directly to them, by setting a counter for each.
- The counter for each end node is automatically incremented at a fixed frequency, and the counter is reset to zero whenever the switch processes a frame sent out by the corresponding end node.
- An end node's counter will reach a threshold if there have been no sending activities for a while. In this case, the switch sends out a HELLO message to the corresponding end node.
- If the end node replies back to the HELLO message, the counter is reset.
- The switch will resend the HELLO message up to 3 times in subsequent ticks, if the end node did not reply.
- If all 3 HELLO are missed, the switch broadcasts a LEAVE message for the end node, with the reason of "Unexpected Leave".
- If a network consists of two end nodes forming a *direct connection mode*, the two nodes will also perform this periodic monitoring routine with each other.
- In normal P2P mode, the end nodes themselves do not perform any liveness monitoring tasks.
- This ultimately assumes that the switches must never fail or leave the network, especially if it still has live end nodes.
- A rebooted switch will have a blank routing table, and will drop all incoming frames. This effectively partitioned the network.
- Currently, no recovery mechanism has been implemented for failing switches.
- At this time, switches also do not probe other neighboring switches for liveness in a spanning tree topology. This is planned for the future.

Demo Application

Network Demo

- Star topology with 3 end nodes and 1 switch
- All end nodes are equipped with a servo motor and a LED
- In addition, Node 1 is equipped with a PlayStation 3 Joystick
- Two concurrent application demos running among the boards



Remote Servo Controller

- Demonstrates the use of broadcasting in a master-slave application model
- The joystick's X-axis controls the position of the servo motors.
- The Y-axis controls the brightness of the LEDs that are currently on.
- Upon moving the joystick, Node 1 broadcasts a message to all other nodes in the network with the new servo position and/or LED brightness.
- All other nodes will move the servo's position and/or adjust the LED's brightness to the new set of values upon receiving the broadcasted message.

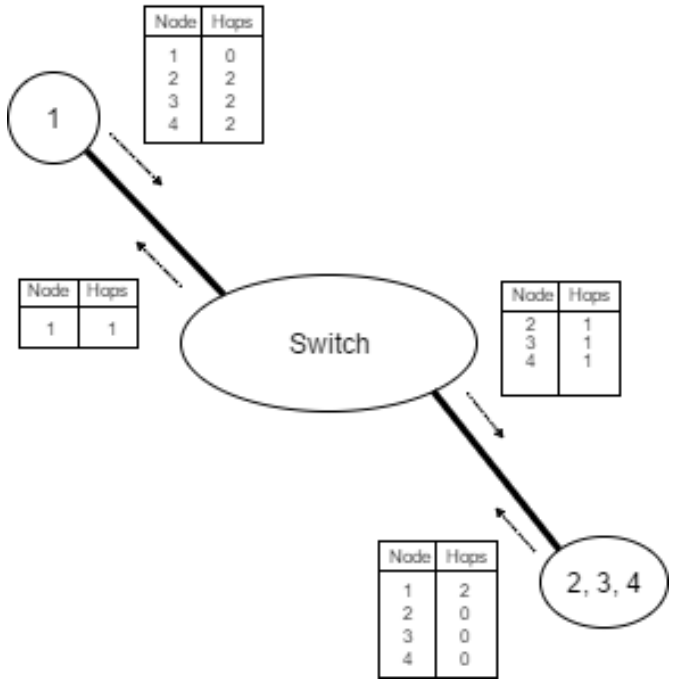
Peer-to-Peer LED Cycler

- Demonstrates a decentralized application model, with end-to-end messaging.
- Purely P2P, supports an "unlimited" number of peers participating
- This demo is essentially a distributed finite state machine (DFSM)
- An end node with ID *i* waits for a "toggle" message coming from node *i-1*.
- When received, the node toggles its LED state between ON/OFF, and sends a toggle message to node *i+1*.
- If no node with higher ID exists, the node will send a toggle message to the node with the lowest ID in the network, completely one complete cycle.
- Nodes can join and leave this DFSM anytime, without causing a deadlock
    - Done by overriding the JOIN and LEAVE message handlers

Work-In-Progress
Link-Layer Features

Virtual Links

- One physical microcontroller and be used to “host” multiple logical end nodes over the same physical link.
- By sending multiple JOIN message with different IDs, a physical end node can be recognized as multiple logical end nodes in the network
- All frames destined for these logical end nodes will be routed to the same the same physical link, arriving at the same physical microcontroller
- Reminder: The network protocol does not differentiate physical/logical nodes in the network protocol. All end nodes are considered logical.
- Example application: one device can use a logical end node to transmit out-of-band control messages, and a second logical end node to transmit data messages.



Dynamic ID Assignment

- At this time, nodes wishing to join the network must have a predefined GUID. Fine for static membership, but not for dynamic membership.
- Currently, the initial HELLO message has the sender's predefined ID filled into the frame's *source* field.
- We can denote a special case where if the source of a HELLO is "0", that means a node does not have an ID yet.
- These "anonymous" HELLO messages must also add a randomized *message identifier* field in the payload.
- When a switch receives an "anonymous" HELLO, it will look into its routing table and find an unused ID.
- The switch then copies the message identifier from the received HELLO, appends the chosen ID after the message identifier, and sends it back to the same link the anonymous HELLO came from.
- The receiver of the response will validate the message by matching the message identifier. If it matches the identifier it used to send the anonymous HELLO, it will accept the ID and send a JOIN message with it.
- Dynamic ID assignment is not yet implemented, and is be planned for the future.