Peer-to-Peer Networking Over UART

For Microcontrollers and Embedded Applications

By Bowen Liu

Abstract

In this report, we will be adding multi-way and peer-to-peer communications to the standard UART protocol, by overlaying a link layer on top the UART physical interface. The project aims to add enough network primitives to allow the end user freely implement more advanced overlay networks and networked applications in an embedded environment. This includes adding structured communication, ad-hoc peer-to-peer capabilities, and routing between multiple devices. The project will also support dynamic membership management to make the network suitable for wireless UART links (such as using Bluetooth SPP). Ideally, the network stack described in this project would be generalized to have a minimal amount of hardware dependence, so users can easily integrate it into any devices, or even adapt it to another end-to-end protocol with ease.

Introduction

Microcontrollers are ubiquitous, and are found in virtually every electronic devices and gadgets. They are tiny integrated circuits, with a built in low-power microprocessor along with various peripherals, all encapsulated within one chip.

While high end microcontrollers often have built in network controllers (eg: Ethernet or Bluetooth), integrating them into an existing design isn't often easy or cheap. Many additional components must be added to the physical circuit to use the controllers, and logically interfacing with the network controllers from the programmer's perspective is another daunting task. Furthermore, electronic hobbyist often cannot afford to use high end microcontrollers in their projects. Lower end microcontrollers, such as the ATMEL ATMEGA2560 [1] used in this project, do not have any sophisticated I/O protocols (ie, CANBUS) or dedicated network controllers.

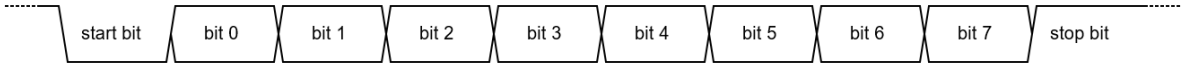
There are millions of different microcontrollers on the market, all with unique specifications and configurations designed to serve specific roles. However, virtually all microcontrollers have one I/O protocol in common, and that is UART.

UART

UART stands for **Universal Asynchronous Receiver/Transmitter**. Essentially, it's the same logical specification that has existed since the late 1960s, used by the computer's ***serial port***. The serial port was considered as a staple for I/O in the earlier days, connecting every peripheral ranging from mouse, keyboards, to even dial-up modems. The serial port is no longer considered relevant for consumer computers, but its popularity remains greatly among embedded systems.

The UART protocol is asynchronous, meaning that a sender can transmit data over a link whenever is needed. The receiver can also check for any received bytes whenever it's desired. In most microcontroller implementations, the UART peripheral is full-duplex, meaning it can send and receive bytes at the same time. Both of these traits makes it very suitable for constructing a network.

In addition, the operating principles of the UART protocol is also very simple. A link only requires two wires, one for transmitting (TX), and one for receiving (RX). UART is a raw interface and do not require any handshakes. Messages and data are sent/received as raw bytes across the link, transmitted one bit at a time. The simplicity of UART is the key to its popularity in embedded devices. [2]



Design Considerations

A network building on top of a UART port will not be able to achieve high network performances, mainly due to the nature of the protocol itself, and the limitations of typical microcontrollers. We must consider these major issues while designing and implementing our network protocol.

1) **Microcontrollers have small amount of system memory available.**
   The ATMEGA2560 Microcontroller we're using only has 8KB of system memory in total. This means that we must limit the size of our messages transmitted throughout the network, and limit the number of message a peer is expected to process in a given timeframe. In addition, we must also be "smart" about memory management to maximize the amount of system memory usable by the user's application.
2) **UART link speed is very slow.**
   Devices utilizing UART commonly uses a link speed of 9600bps, or 1.2KB/s. While the ATMEGA2560 microcontroller supports UART link speeds up to 115200bps (14.4KB/s), transmitting a 1MB file would still take 70 seconds if all other overheads are disregarded. Therefore, we should not expect the user to use this network protocol to perform *bulk transfers* frequently.
3) **The network is only intended for small number of peers, forming a local area network**.
   In addition to having slow link speeds and low system memory, microcontrollers are often very slow compared to proper microprocessors. The ATMEGA2560 uses an unpipelined 8-bit AVR processor core, running at 16-MHz. If we perform nothing but 32-bit operations in our code, the "effective clock rate" of this microcontroller would be less than 4Mhz.
   Therefore, it's fair to assume the processing delay of the network will be very high (resulting in high latency), thus preventing the network from scaling very well with all factors considered.

Network Principles

There is one major problem with using UART as a networking foundation for this project. UART is an **end-to-end protocol**, meaning it's only intended to have two peers connected directly with each other. While multi-way implementations of the serial port exist (eg, RS-485), their implementations are electronically incompatible with the standard UART interface found on most microcontrollers. [3]

Because of so, the fundamental network model used by this project is to overlay a **packet switching network** on top of the UART physical interface. The network will be mainly consisted of two types of devices, switches and end-nodes.

Switches acts as a store-and-forward broadcasting medium, with the sole purpose to route frames between the end-nodes. It's simply another microcontroller have multiple UART ports, allowing an end-node to be connected, or daisy chains another switch to expand connectivity. When a switch receives a frame from one of its ports, it will look at the destination field of the frame, and determine which port the destination is reachable at. The switch will then forward the frame out onto the corresponding port.

End-nodes are devices running user defined applications as its main focus. It does not need to worry much about tasks directly related to networking. Whenever the user's application requires a message to be sent to another end-node, the program encapsulates the message into a frame, and tag it with its own ID and the destination's ID. The frame is then sent out the link, and the end-node will forget about this frame and return to its usual tasks. The end-nodes also do not require to handle incoming frames immediately. It can periodically check its receive queue for any missed messages, or it can be event driven.

Network Topology

The current version of the project makes the following assumption about the network:

1) End-nodes only have 1 link, and switches have more than 1 links.
2) Only one path exists between any two nodes.
3) The network does not contain cyclic paths between the switches.

These assumptions are made to simplify the project for time's being. In later version of the network, we will plan to support end-nodes with multiple links, allowing end-nodes to form direct connections with each other (to reduce latency) in addition to a globally routable path. Physical cyclic links between switches will also be allowed (for redundancy), but they are only enabled upon detection of a link failure.

Based on these assumptions, the current project supports three network topologies using the switch-and-node model:

1) **Star**: A star topology consists of a network with a single central switch, and a number of end-nodes connecting directly to it. The distance (in hops) between any two nodes are uniform and minimal.
2) **Spanning Tree**: A spanning tree topology consists of a network having multiple switches daisy-chained together, and a number of nodes connected to different switches. The end-nodes often have different distances among each other.
3) **End-to-End**: Two end-nodes can form a direct connection with each other, without needing messages to be routed by a switch.

The star topology typically provides the fastest latency in a multi-peer network, but the number of peers supported in the network may be limited by the physical number of UART links available in the central switch. In contrast, the spanning tree topology allows a much larger number of end-nodes to join the network, at the cost of increasing the logical distance between two nodes.

Direct connections are not considered much at the current stage of the project. It's intended to be used with a later implementation of the routing algorithm that allows end points to have multiple links. Another potential application is its use in a dynamic ad-hoc network over a wireless link, where an end-node may "move around" the network and connect to different neighbors, depending on its physical location. Direct connections are not recommended for static networks permanently having only two nodes as the network protocol incurs additional amount of processing overhead. The user should consider writing application-specific communication code directly over the raw UART link instead.

Network Stack

The bulk of the project focuses on overlaying a link layer on the physical UART interface. However, the final version of the project is expected to contain multiple network layers, each using services offered by the previous layer.

Physical Layer

The physical layer at this point is essentially the hardware UART implementation within a microcontroller. Currently, the network protocol interfaces with the low-level UART ports on an ATMEGA2560 microcontroller directly using the Arduino library for simplicity.

One of the major design goal mentioned earlier is to make the bulk of the network protocol portable and hardware independent. In later versions of the network protocol, we will remove all Arduino code and replace them with stubs instead. These stubs are meant to be used as part of interrupt service routines for common I/O related interrupts, such as the arrival of a new data block, or completing the transmission of a data block. It's up to the end users to incorporate them into platform-specific code needed by their projects.

Link Layer

The service provided at the lowest level of the link layer is **frame synchronization**. This part of the link layer works directly with the interrupt-driven raw I/O functions provided in the physical layer.

Because UART is asynchronous and unstructured, we do not know when data is expected. When data do start to arrive, we do not know the size of the stream we're currently expecting. Data arrives a few bytes at a time, instead of a complete frame sent by the user.

The frame synchronization service provides a raw receive buffer for all links. Whenever the service detects the hardware UART interface having new bytes ready for reading, the new bytes are moved from the UART's hardware buffer and appended to the end of the raw receive buffer located in the system's main memory. The service will check if a known frame preamble is found, if the buffer is not known to contain a valid frame yet.

If a valid preamble is detected somewhere in the current raw receive buffer, the location of the preamble, along with all subsequent received bytes currently in the buffer, are shifted to the beginning of the buffer. The buffer's status is marked as VALID, indicating it's currently receiving a frame and it must not be flushed.

If the buffer is marked valid and contains more than 4 bytes of data (expected size of the frame header), the service will check if the header's format is valid by matching a 'STX' character at the 5^{th} byte. If valid, the first 4 bytes of the raw read buffer are parsed to extract the frame information, notably the SIZE field, to know how many bytes to expect for the payload that's currently in flight. Once the number of bytes specified by the SIZE field has been received (after the header), the completeness of the frame is validated by matching a 'ETX' character at the end of the expected payload.

The raw receive buffer of a link will be flushed with all bytes discarded if:

1) The buffer is over 50% full and no valid preamble has been matched.
2) The number of bytes pending in the hardware buffer exceeds the amount free space in the software raw receive buffer (and it's not marked valid).
3) The 'STX' or 'ETX' were not found at the expected location, after detecting the start of a frame.

The link layer made communications asynchronous at the software level, by providing a **queueing service** for both sending and receiving frames. Once a complete and valid frame has been received in a raw receive buffer, the frame synchronization service concludes its task by unmarshaling the raw bytes into a properly structured FRAME, and appends it to the tail of the link's receive queue. The raw receive buffer is then flushed, allowing bytes from further transmissions to be buffered.

The user can check the receive queue at any time for newly arrived frames, and is not required to handle a new frame immediately. This allows the user's application to perform productive tasks, instead of needing to wait for the incoming transmission to complete before able to parse the message.

The link layer also offers similar sending service at the lowest level, but its functionality is not as detailed. When the user wants a frame to be sent, the link layer will marshal the structured FRAME into a sequence of raw bytes. This raw frame is then added to the end of the link's sending queue. The user can submit a frame to the send queue at any time, but the link layer will only transmit frames periodically to prevent communications from degrading the user's application too much.

At a higher level of the link layer, the link layer also provides routing services for the frames sent between end-nodes, as well as membership management allowing nodes to join and leave the network at any time. These services will be discussed in detail in a later section.

Transport Layer

The Transport layer was originally planned in the initial project proposal, but was later abandoned for due to lack of time. The primary intention of the transport layer is to offers stream transfers, where the user can send out large blocks of data after being fragmented into sequences of smaller frames. The transport layer will also offer TCP-style reliable streams, where the sender must negotiate a connection with a receiver before data transfer can be initiated. Furthermore, the receiver is expected to acknowledge every chunk of data sent, so a stream of data is guaranteed to be delivered correctly.

The transport layer is also intended to introduce flow and error control, to dynamically adjust transmission rates preventing links from being congested from too many messages being sent. It is not known if this function will be implemented in the transport layer, or the link layer.

Application Layer

The application layer is where the users implement their own applications. Services from any of the lower layers may be used directly by the applications.

At this time, it is not known if streaming services provided by the transport layer is needed over a UART network, due to its low link speed. If unneeded, users can implement their application to interface with the link layer directly. The network layers in the projects do not have forward dependency (ie, link layer can operate independently without the transport layer, but not vice versa).

To send a link-layer frame, the user application simply calls the “send frame” function defined in the link layer libraries. Receiving frames is handled semi-automatically by the link layer through the use of callback functions.

In the current implementation, the user’s application is automatically “signaled” when a message frame (from the link layer) has arrived at the current end-node (or rather, has pending message frames in the receive queue upon periodic checking). The user must create a message parsing function (that takes a FRAME as an input) in its application to handle the application-specific message frames. This function is called automatically when the link layer finds new messages available in the receive queue.

The link layer has also defined a number of control frames used for routing updates and membership management (to be discussed later). These control frames are processed by the link layer automatically and do not require the user application’s intervention. If a P2P application require to update its behavior based on routing/membership changes within the network, the user application can be notified for receiving relevant control frames by setting handlers for the corresponding control frames. The link layer will first process the control frame based on the protocol specifications (eg, update the local routing table), and then call the user’s handler afterwards. If a user handler is not defined for a specific type of control frame, the signal is ignored by the application.

Routing

Routing Table

All end-nodes and switches in the network maintains a **routing table** for each of its link. For switches, one routing table is maintained for each of its links, serving as individual lists of end-nodes reachable at each of the links. End-nodes only maintain one single routing table, since they only have one links. This sole routing table is essentially a list of all active nodes in the network.

The routing table contains one entry for every possible node ID in the network. Currently, each entry in the routing table simply records how many hops away a node is, from the current node. For example, if the 5th entry of a node's routing table states 3 hops, that means Node 5 is a valid node, and a frame will be delivered to Node 5 from the current node in 3 hops.

A routing table entry with a hop count of zero indicates the node is either invalid/not present in the network, or the entry is for the current node (must be further validated by checking the "ID" field set in the link configuration struct). A switch can find all of its leaf end-nodes by finding entries with hop counts of 1 in its routing tables.

The routing tables is assumed to contain a consistent list of **valid entries** across all devices in the network, at the very least. That means all end-nodes that are alive in the network (**valid/active/live nodes**) must have a nonzero hop count in its corresponding entry in all routing tables, of all other devices in the network. Additionally, entry 0 (link control) and the entry with the highest ID (broadcast) are never used for any purposes.

End-nodes maintain routing tables solely for the purpose of knowing the IDs of all active end-nodes in the network. In addition, the hop count in the routing table is never used explicitly by the link layer code for end-nodes, and it's only there to potentially aid the user's application in finding closest neighbors to coordinate tasks with.

It's important to mention that the routing table is not preconfigured statically. Upon a device initializes, its routing table is blank with every entry's hop count set to 0. The routing table is filled dynamically either by requesting a current copy from the neighboring switch, or have individual entries modified based on join/leave events throughout the network. The details will be discussed in the Membership Management section.

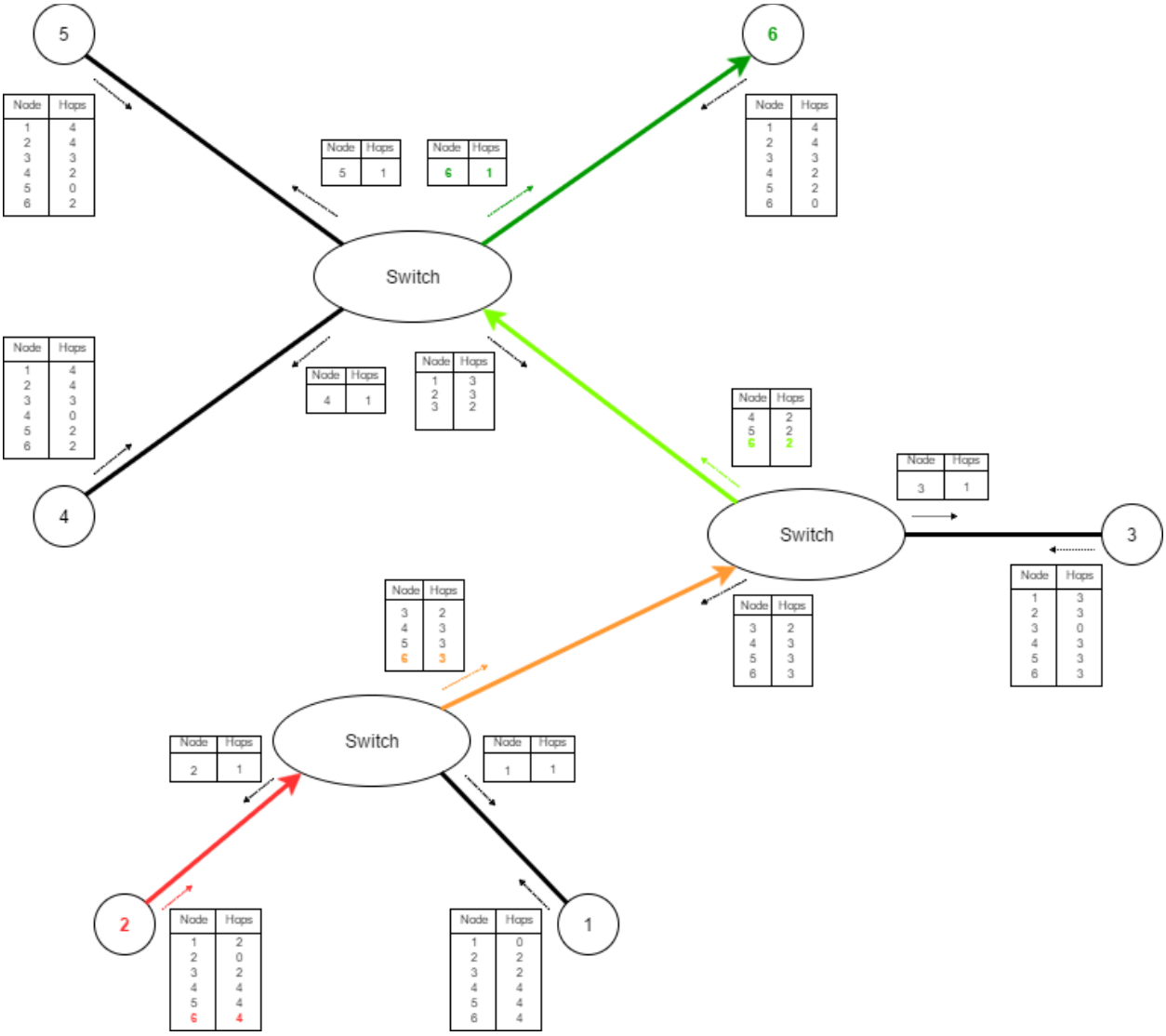
Frame Routing

As previously mentioned, frame routing is handled exclusively by the switches in the network. When an end-node *a* wants to send a frame to end-node *b*, node *a* fills the source and destination of the frame's header as *a* and *b* respectively, and transmits the frame out its only link. Neither end-nodes will know the exact route it took for the frame to be delivered, or the frame's current location.

Whenever a switch receives a frame on any of its links, it will first examine the purpose of the frame. If the frame has a "message" preamble, and its destination field is nonzero, the switch will attempt to forward this frame towards its destination. The switch starts by looking through each of its link's routing tables. When a valid entry is found in one of its link's routing table for the frame's destination, the frame will be forwarded out the corresponding link without any modifications.

In a spanning tree network, it's likely for the frame to arrive at another switch immediately after. This process is repeated by each of the subsequent switches encountered by the frame, and each switch will forward the frame one hop closer towards the destination. Eventually, the frame arrives at a switch having the frame's destination as a leaf end-node, and this switch will forward the frame to the intended end-node. The worst-case routing performance using this method is O(*n*) hops, when the entire network is consisted of a linear route formed by *n* switches, with the source and destination nodes located on polar opposite ends of the linear route.

Below is an example of a frame routing from Node 2 to Node 6. The example shows a spanning tree network having 3 switches and 6 end-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 a switch will drop the received message frame silently if it's unable to find a valid entry in any of its links' routing table for the frame's destination. On the other hand, if the switch receives a frame having a "control" preamble, and the frame's destination field is set to 0, the switch will consume the frame by parsing its payload for additional control parameters, and reacting to it appropriately. A list of control frames, and how they are handled will be discussed in the following section.

Broadcast

In addition to routing messages between two end-nodes, the network protocol also supports broadcasting. If an end-node wishes to send a frame to be received by all other end-nodes, it simply needs to fill the destination field of the frame to the broadcasting address (currently 15, the highest numerical value supported by the 4-bit ID fields).

When a switch receives a frame with the broadcasting address as its destination, the switch will send a copy of the frame out all of its links, except for the link that received the broadcast frame. This is based on the realistic assumption that all other end-nodes reachable from the received link has already received (or will be receiving) a copy of the frame that was broadcasted by the previous switches in the broadcasting path.

When a switch forwards a broadcasted frame to an end-node, the frame's destination ID will remain unchanged as the broadcast address. An end-node receiving a broadcasted frame will treat the frame as any other frames directly destined to itself, and will not forward it any further.

Membership Management

Another major design goal considered for the network stack was the support of dynamic membership, allowing any nodes to join and leave the network at any given time. The network protocol would immediately signal the user application when there are routing updates, so the user application can reconfigure itself to adapt to the new network state. Adding supports for dynamic membership allows the network protocol to be potentially used with end-nodes connecting over wireless links.

Network Join

Several control frames have been defined solely for the sake of providing routing updates, and are the backbone of the dynamic membership system. When a new end-node initializes, it sends a **HELLO** control frame out its link to probe the other end of the link. This message contains the new node's ID in the frame's *source* field, and an additional field telling the other end of the link it's an end-node. If a receiver exists on the other end of the link, it will also reply back a HELLO.

If the replied HELLO indicates the other end of the link is a switch, the new node will configure itself to operate in *P2P mode*, essentially everything we've described so far. If the other end of the link replied that it's another end-node, the new node will configure the itself to operate in *direct connection mode* to operate in an End-to-end fashion. Currently, the direct connection mode is not implemented. Its operating principle is expected to be a reduced subset of the regular P2P mode.

Once the new node has probed the other end of the link (assuming it's a switch), the new frame will send a **JOIN** control frame to the switch by setting its own ID as the source, and the destination as zero. The join message contains an additional "hop count" field in its payload, and its value is set to 1 by the new node when it sends the JOIN to the switch. A JOIN frame's source ID and its additional hop count field is used to update the routing table of the recipient.

When a JOIN control frame is received by any end-node or switch, the receiver will mark entry (corresponding to the frame's source ID) in its receiving link's routing table as valid, by copying the value in the hop count field into the routing table entry. In addition, switches that received the JOIN message will increment the hop count field in the JOIN frame by 1, and broadcasts the frame to all of its other links. Eventually, all end-nodes and switches will receive this JOIN message, and every existing device will have their routing table updated to contain their distances away from the new node.

However, the new node itself still has an empty routing table, and is not aware of anyone else present in the network. The switch that first received the new node's JOIN message (ie, the switch that's connected directly to the new node) addresses this by sending a copy of its own updated routing table to the new node (using a **RTBLE** frame), with the hop count of every valid node in the network incremented by 1. The new node copies every entry in the RTBLE frame into its own routing table, except for the entry corresponding to the new node's own ID (which will remain as 0). The new node,

along with every other end-nodes and switches, now have am up-to-date and consistent (having correct hop counts between all valid nodes) routing table with each other.

Network Leave

An end-node in a network can choose to leave the network voluntarily by sending out a **LEAVE** control frame out its link. The LEAVE control frame contains an additional field in the payload allowing the leaving node to specify an 8-bit integer as its reason to leave. The reason definitions, as well as how they are handled, is dependent on the user's application. The network protocol only defines a reason of "1" as a generic planned leave, while a reason of "0" indicates the end-nodes has died unexpectedly (discussed later).

When an end-node or switch receives a **LEAVE** control frame, the process is very similar to how JOIN frames are handled. The only difference is the receiver will mark the entry (corresponding to the LEAVE frame's source ID) in its receiving link's routing table to 0, indicating this node is no longer valid in the network. A switch receiving a LEAVE frame will also forward it out all its other links, without modifying the frame.

Active Monitoring

In the scenario described above, a node leaves the network gracefully by letting every other device in the network know about the event ahead of time. However, an end-node can fail unexpectedly and quietly leave the network without being known by any other devices. This problem is a lot more prevalent with wireless links.

To overcome failing end-nodes, all switches in the network actively monitors the liveness of all end-nodes that are connected directly to them. Each switch will maintain an **inactivity counter** for each of its end-nodes to count the ticks elapsed since any frames were send from a corresponding end-node. The inactivity counters are incremented at a fix frequency, with each period called a **tick**.

An inactivity counter will be reset to zero if the switch processes a frame sent from the corresponding end-node. In contrary, the inactivity counter will reach an **inactivity threshold** if no messages have been sent by the corresponding end-node, after a predefined number of ticks elapsed. In this case, the switch will send a HELLO message to probe the corresponding end-node to ensure it's still alive. If the end-node replies to the HELLO message, the counter is reset.

If the corresponding end-node did not reply to the first HELLO message, the switch will resent the HELLO message 2 more times (once on each of the two subsequent ticks). If the end-node still has not replied to any of the HELLO messages, this end-node is declared dead. The switch will remove this node from its routing table by marking its hop count to zero, and then broadcasts a LEAVE message for the dead end-node (by filling its node ID as the frame source) with a leave reason of "Unexpected Leave".

When the network is operating under the normal *P2P Mode*, end-nodes themselves do not perform any liveness monitoring tasks at this time. This ultimately makes a dangerous assumption that switches in the network must not fail or leave the network, especially if a switch still have live end-nodes connected to it.

Failing Switches

There aren't any recovery mechanisms implemented for failing switches currently. If a switch fails and does not come back online, the network is effectively partitioned. This scenario is unrecoverable under the current network topologies, as no cyclic paths are permitted between two switches. Therefore, it's not possible for two network partitions to find an alternative path to reconnect to each other again, since no redundant paths exists in the first place.

Once cyclic paths are supported by the routing algorithm, we can also start letting switches probe neighboring switches to ensure they're alive. If a switch finds one of its neighboring switches as dead, the detecting switch will attempt to find an alternative path to reconnect the partitioned network.

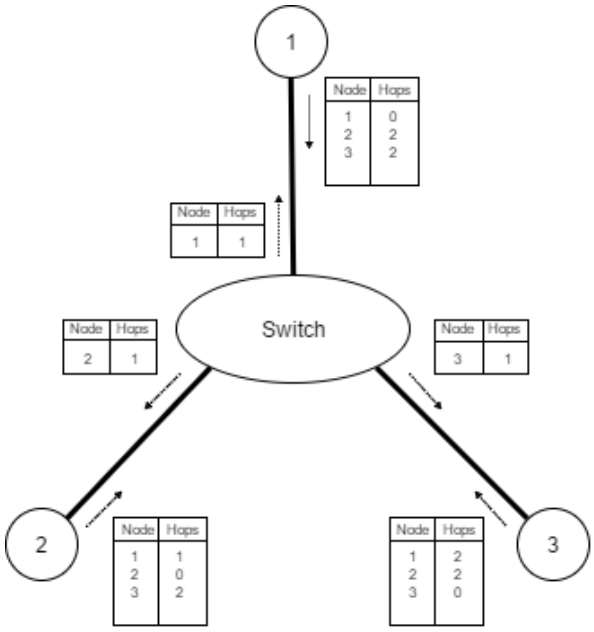
If a failed switch managed to reboot itself and come back online, the rebooted switch will have a blank routing table thinking there are no active nodes in the network. As a result, this switch will drop all frames it receives, which also

effectively partitioned the network. This failure scenario is recoverable under our current network topologies, but no recovery mechanism has been implemented yet.

One possible approach is to extend the HELLO message by adding additional fields to indicate if the sender of the message has already joined the network (as well as its ID if applicable). In addition, we will also let switches send out HELLO messages upon initialization as well (currently, only end-nodes sends out HELLO messages upon initialization. Switches simply remains quiet upon initialization). If the switch receives a HELLO message replied from an end-node claiming it has already joined the network, the switch will simply add the end-node's ID into its routing table. If the switch receives a HELLO replied from another switch on one of its links, the current switch will request the routing table from the sender and use its entries to update its own routing table for that link. At the end of this process, the recovering switch will have a consistent routing table with the rest of the network.

Demo Application

To demonstrate the capability and potentials of the current network stack, we've built a simple UART network using four Arduino Mega 2560 boards. The network is connected under a star topology, with three end-nodes and one switch.



All three end-nodes are equipped with a servo motor and a LED each. In addition, Node 1 is also equipped with a PlayStation 3 Joystick. The end-nodes are running two demo applications concurrently, as described below, and are implemented in the application-layer (ie, the demos do not interact with the lower level network interfaces directly).

Remote Servo Controller

This demo application demonstrates the use of *message broadcasting* in a *master-slave* application model. Node 1 is predefined to be the master, while node 2 and 3 are the slave. The master uses its joystick positions to tell the slaves (and itself) where to move its servo, and how bright its LEDs should be. At all time, the servo position and LED brightness are synchronized across all nodes.

The X-axis of the joystick is used to control the servo position, and the Y-axis controls the brightness of the LEDs that are currently on. The master reads the value of the analog-digital-converter (ADC) connected to the joystick's X-axis pin (ranged from 0 to 1023) and normalize it to a degree between 0 to 180. The ADC value of the Y-axis pin is normalized to a number between 0 to 255, which are the ranges supported by the digital-analog-converter (DAC) used to change the brightness of the LEDs.

If the master detects a change in either joystick axis, it broadcasts an application-defined “MVSV” command out onto the network. This command sent out as regular message frames, by concatenating the opcode “!MVSV”, the current servo position, and the current LED brightness together as the payload.

All slave nodes has defined their message frame parsers to look for the “!MVSV” opcode at the beginning of every message payload. If found, the subsequent bytes are extracted as the servo position and LED brightness, and are used to update the corresponding peripherals.

Peer-to-Peer LED Cycler

This demo application will make the LEDs on each board toggle one after another, forming a coordinated blinking cycle changing sequentially across all boards. This is an example of a fully *decentralized* P2P application model, using end-to-end messaging instead of broadcasting. Every node in the network runs the exact same code for this application, with no predefined master/slaves. In theory, this demo supports an unlimited number of peers participating, as long as the network itself can support it.

In this demo, the **predecessor**/**successor** of a node (with ID ***i***) is another node with an ID smaller/larger than *i*, while also having the ID closest to *i.*

If node *i* currently has the smallest/largest ID in the network, then its predecessor/successor will be the node with the largest/smallest ID.

This demo can be considered as a *distributed finite state machine (DFSM)*. Each node waits for a toggle command (“!TLED” opcode) coming from its predecessor. Similar to the remote servo controller demo, the command is simply encapsulated into a message frame’s payload. Each end-node’s message frame parser is further extended to try match this opcode, in addition to the “!MVSV” command (for slaves only) in the previous demo.

When the toggle command is received by a node from its predecessor, the receiving node will toggle its LED, and sends out a new toggle command to its successor. This effectively creates a cycle where every node tells the subsequent node to toggle its LED.

Any nodes can join or leave this *DFSM* at any time without causing a deadlock to the LED cycle. If a node was signaled for routing updates (receiving a JOIN, LEAVE, or RTBLE control frames), the node will reset its LED status to off, and recalculate its new predecessor and successor to receive/send the toggle command to/from.

Further Work

A number of proposed features and improvements has been mentioned throughout this report. In general, fault tolerance should be the next major design goal to be worked on, as most of the network protocol operates under the assumption that no faults occur during transmission (ie, frame corruption). In the meantime, there are two unrelated features that are currently being worked on, and will be most likely featured in the next revisions of the network protocol.

Virtual End-nodes

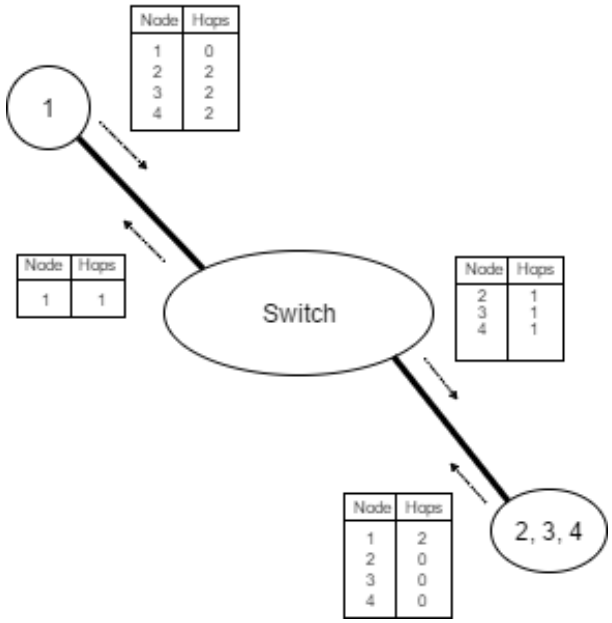
The concept of virtual end-nodes is to allow one physical microcontroller to “host” multiple logical end-nodes over the same physical link. As a reminder, the network protocol does not differentiate between physical and logical end-nodes in the network. All end-nodes are considered logical.

This feature was accidentally discovered while testing an earlier version of membership management, where one physical end-node can be recognized as multiple logical end-nodes in the network by sending multiple JOIN messages with different IDs.

Recall that every time an end-node/switch receives a JOIN message, the receiver marks its routing table entry (of the receiving link) for the sender as valid, by setting hop count with the value found in the frame’s payload. The switch connecting directly by the nodes’ shared physical link will record a hop count of 1 (into the receiving link’s routing table

entries) for all of the source IDs of the different JOIN messages, indicating they are all a leaf end-node reachable at that link. Therefore, all frames destined for any of these logical end-nodes will be routable from all end-nodes, and are delivered the same physical link.

Below is an example of a network with two physical end-nodes. Node 1 is hosted on its own dedicated microcontroller, while node 2, 3, 4 are hosted on a shared microcontroller and all communicating over the same link. Node 1 can send end-to-end messages to any of node 2, 3, 4 and they will all be routed to the same physical link (and vice versa). Note that if any two nodes of 2, 3, 4 sends a message to another, the message will be looped back to the same physical link by the switch.



Theoretically, the immediate switch to a physical end-node can detect logical nodes from a single physical link, since the initial HELLO message sent out by the physical end-node claims that it's not a switch, yet multiple JOIN messages originated from this link. However, all the other switches or end-nodes cannot differentiate this, since they only know these end-nodes are at least 2 hops away (could be multiple physical end-nodes connected to the same switch).

One practical application for using logical end-nodes would be the separation of application-level control messages with data messages. Every device can instantiate one logical end-node and use it to transmit out-of-band control messages, and instantiate a second logical end-node to transmit user messages. This separation provides an additional layer of security to prevent potential exploits letting user messages to be accidentally parsed as control messages instead.

At this time, the virtual end-node feature is not supported by the current routing protocol, but will be fixed for the next revision.

Dynamic ID Assignment

In the current membership management implementation, nodes wishing to join the network must have a predefined ID programmed into its code already. This is fine for static memberships where the end-nodes are hard wired and only a fixed number of them are joining the network at all time, but such scheme is limiting for dynamic memberships. The goal here is to allow an end-node without an ID to join the network, by letting the connected switch assign an ID to the new node.

We can implement dynamic ID assignment by further modifying the initial HELLO message send from new end-nodes out its link. Currently, the initial HELLO message (that's sent out when a node initializes) has the node's preprogrammed ID filled into the frame's *sources* field. We can denote a special case where if the source field of a HELLO is "0", this indicates the sending node does not have an ID yet. We'll call these messages "**anonymous HELLO**".

When a new node wants to send out an anonymous HELLO message, the new node must also attach a randomized identifier into an additional *message identifier* field in the payload. If the new node also receives anonymous HELLO in return, it will use this identifier to match if it's the intended recipient. A matching message identifier indicates the reply contains a new ID to be used, and a mismatching identifier indicates the other end of the link is also requesting an ID simultaneously (in the case of an end-to-end connection). As with the rest of this report, we will not consider the end-to-end scenario.

When the switch receives an anonymous HELLO, it will look into all of its routing tables and find an ID that's not currently being allocated to any valid nodes in the network. The switch will then append the chosen ID after the *message identifier* in the payload of the received anonymous HELLO frame, and sends the anonymous HELLO back to the link it came from. Note that the *source* field of the frame is still retained as zero.

When an end-nodes receives an anonymous HELLO back on its link, the receiving end-node looks into the *message identifier* field of the payload and matches the one it originally sent out. If it matches, then the new node will accept the ID provided by the switch from the payload, and send out a JOIN message with it. If an end-node receives an anonymous HELLO, and the network is not end-to-end, the end-node will drop this message. Only switches will handle ID assignments for its leaf end-nodes.

In summary, the below steps are needed for a new node to obtain an ID:

1) An end-node without an ID sends an anonymous HELLO (*src* field is 0) out its link, along with a randomized message identifier in its payload.
2) The switch receiving the anonymous HELLO checks all of its routing tables to find an unallocated ID, appends it after the frame's message identifier in the payload, and sends it back to the link it came from.
3) The receiver (new node) matches the message identifier in the reply's payload with the one it sent out. If matches, the new node will accept the new ID, and send a JOIN message out its link with it.

Conclusion

This report documents our attempt to expand the standard UART protocol by adding multi-way communications, ad-hoc and peer-to-peer capability by overlaying a link layer on top of its physical link. A packet switching networking model is used to overcome the end-to-end design of UART, and the network supports star and spanning tree topologies, as well as direct end-to-end connections. End nodes can communicate with each other in pairs, or can broadcast frames to allow all end nodes to receive a message. The network also permits dynamic memberships, allowing end nodes to join and leave at any time.

The purpose of this project is not to replace any other existing multi-way peripheral buses, such as CAN. Because UART is a protocol found on virtually all microcontrollers, this project aims to provide a framework of networking primitives to allow application-specific networks to be built by the user (for a localized network of embedded devices), needing only a minimal amount of additional hardware. The network protocol is suitable for electronic hobbyist to use budget microcontrollers create projects requiring communication across multiple devices, as well as potential industrial applications. The devices can communicate over wires described in this report, or can be used over a wireless UART link such as Bluetooth SPP mode [4].

While this project is considered far from complete (both in terms of implementation and design), the current work documented serves as a great proof-of-concept. In addition, a lot of further optimization is needed to improve the network protocol's throughput and latency. Fault tolerance will be the next major design goal to be worked on, as the network mostly operates under the assumption that no errors occur during transmission.

References

[1] “Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V.” [Online]. Available: http://www.atmel.com/Images/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf. [Accessed: 10-Apr-2017].

[2] “Universal asynchronous receiver/transmitter,” Wikipedia, 07-Apr-2017. [Online]. Available: https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter. [Accessed: 10-Apr-2017].

[3] “RS-485,” Wikipedia, 07-Apr-2017. [Online]. Available: https://en.wikipedia.org/wiki/RS-485. [Accessed: 10-Apr-2017].

[4] “Bluetooth Basics,” Bluetooth Basics. [Online]. Available: https://learn.sparkfun.com/tutorials/bluetooth-basics/bluetooth-profiles. [Accessed: 10-Apr-2017].

Evaluation Remarks

Please do not mark this section of the report.

I am terribly sorry for this lengthy report. I’ve put in a lot of thought and effort into the designs and implementation of the current network stack, and it was not possible for me to use less words to explain my design rationales at each stage.

In addition, I’ve tried my best to explain the workings of my network protocol as detailed as possible, so the reader wouldn’t need to look into the actual code to figure out what’s going on. The code is still somewhat disorganized at this point, and I would not recommend spending extensive amount of time looking at it. If any clarification is needed, please feel free to contact me.

I have included a most recent copy of the codebase for my project, and further updates can be found here: https://github.com/binexec/P2P-UART-Network .

In addition, I did not provide many references simply because I used very little to design the network protocol. All of the designs and ideas described in this report are original work produced from a lot of trial and error. Most of the references I’ve used are programming related, and I don’t think they’re relevant to this report.

Thank you for your time for reading through my report!