CS534'21 Implementation of OLSRv2 for ESP32 Platform

Ruichun Ma
ruichun2019@gmail.com

Abstract

For the CS534 course project, we implement the OLSRv2 protocol for ESP32 platform according to multiple RFC documents. This implementation enables the ESP32 devices to form a mobile ad-hoc network (or a mesh network) and compute the shortest routing paths to reach other nodes.

Though not fully complying with RFC specifications, our implementation has most of the key features of OLSRv2 protocol, such as neighborhood discovery, multipoint relay selection and routing calculation based on link metrics. It consists of around 3000 lines of C code.

We test its correctness and robustness through several different testing deployment. We also discuss the limitations and possible improvements for the current implementation.

1. Introduction

Mobile ad-hoc networks provide several benefits compared with networks that relies on fixed infrastructure. It is a decentralised network that devices can move freely without depending on the central infrastructure. It provides possibilities for applications in different areas such as environment monitoring, rescue operatiosna and tactical edge[7]. Its main advantage is that it can often be more robust than centralised networks. This comes from its way to deliver information through multiple hops.

OLSRv2 is a routing protocol optimized for mobile ad-hoc networks. Its key idea is to use multipoint relaying to reduce flooding and routing overhead. It is a proactive link state routing protocol. Although there are multiple implementations of OLSRv2 on Linux, we have not seen any attempt to use OLSRv2 for low-cost IoT devices. Such low-cost IoT devices can benefit from OLSRv2 in several ways. They can span over a larger area by forming mesh like network topology while still maintaining robustness. They can work in a peer-to-peer fashion over multiple hops without the need of a central infrastructure, which makes deployment much easier.

In this report, we present our implementation of OLSRv2 for ESP32 platform, a popular series of IoT chips. Developers around the world use ESP32 chips for all kinds of IoT applications. ESP32 devices features a full networking stack built on top of WiFi and Bluetooth protocols. Although it has an official Mesh protocol available, this protocol is limited by its design principle. It relies the AP-STA pairs between devices. So the network still require a root node and must form a tree topology. Thus, the support of OLSRv2 in this community is missing.

We follow a bottom-up design principle to implement OLSRv2 according to multiple RFC documents. We describe the implementation details in later sections. To finish this project in time, we make reasonable simplifications while still achieves the key features of OLSRv2, such as neighborhood discovery, multipoint relay selection and routing calculation based on link metrics. We test this implementation with several different deployment and examine nodes' debug logs. The results shows that our implementation can form a mobile ad-hoc network correctly and is robust to mobility. We also discuss the missing parts of our design to fully comply with OLSRv2 specification.

2. Background

2.1. Mobile Ad-hoc Networks

A (wireless) mobile ad-hoc network is a decentralized type of wireless network. It does not rely on existing infrastructure, such as access points to manage the network. Each node work independently and equally. Each node participates in routing by forwarding data for other nodes, the decision of routing path depends on the routing algorithm in use. It has been widely used for applications, such as sensor networks, rescue operations.

There are several challenges to form a mobile ad-hoc network. First, all nodes are mobile, so the topology can be very dynamic. The networking algorithm must have a high adaptability. Second, there are no central controller, so the network must be able to operate in a distributed manner. Third, given the nature of wireless links, we need to consider how to efficiently utilize the channel, such as how to broadcast a message to all the nodes. We also need to consider that the quality of wireless links are dynamic and asymmetric.

2.2. OLSRv2 protocol

OLSRv2[4] is an optimization of the classic link state routing protocol, developed for Mobile Ad-hoc Networks. It is a proactive protocol that exchanges topology information with other routers in the network regularly. Compared to its predecessor OLSRv1, the main enhancement is that it uses a link metric rather than hop count in the selection of shortest paths. OLSRv2 uses and extends the MANET Neighborhood Discovery Protocol[3] and the Generalized MANET Packet/Message Format[5] and the Multi-Value Time specified in [6].

The key concept to reduce the protocol overhead is multipoint relays(MPR). Each node in the network selects two set of MPRs, flooding MPRs and routing MPRs. Each set is a subset of its neighbor nodes that cover all of its symmetrically conntected twp-hop nodes. The algorithm use these two sets to achieve flooding reduction and topology reduction, respectively.

The protocol achieves flooding reduction by controlling the traffic flooded through the network hop-by-hop. Each node only needs to forward a packet when it is first received directly from one of its MPR selectors, i.e. a node that selects it as a flooding MPR. This mechanism reduces the number of transmissions required to distribute link state information across the network.

The protocol achieves topology reduction by controlling the link information shared across the network. A node only declare/flood link state information for their routing MPR selectors, if any. Nodes that are not selected as routing MPR do not send any link state information. In other word, routing algorithm only consider links between routing MPR and MPR selectors. This reduces the number of links distributed in the network.

The use of MPRs allows reduction in the size of link state messages and reduction in the amount of link state information maintained in each router.

2.3. ESP32 Micro-Controllers

ESP32 is a series of low-cost, low-power micro-controllers with integrated WiFi and Bluetooth support. It has dual-core version with up to 240MHz clock. ESP32 is created and developed by Espressif Systems, a Shanghai-based Chinese company, and is manufactured by TSMC using their 40 nm process. It is a successor to the popular ESP8266 micro-controller. It has been widely used for IoT applications around the world.

To develop a firmware/program for ESP32 devices, we use the official ESP-IDF(IoT development framework)[1]. It features the FreeRTOS Operating system support and various networking protocols, such as HTTP, mDNS, lwIP and WiFi protocol, covering the full networking stack. The FreeRTOS provides the ability for users to create multiple tasks, so that ESP32 can perform regular IoT tasks together with our OLSRv2 task.

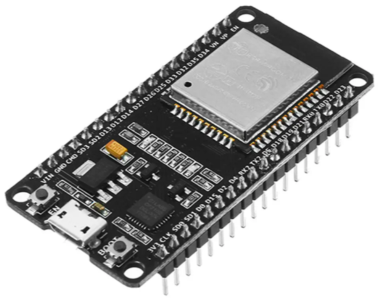
Figure 1. ESP32 Development board

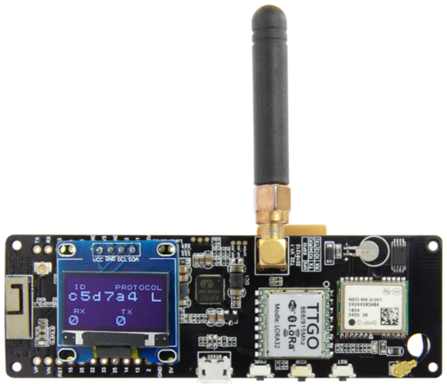
Figure 2. ESP32 device with LORA module

The devices we use for this project are several ESP32 development boards, as shown in Figure 1. They consist of a ESP32 module and minimal peripherals, such as one WiFi antenna and micro-usb port. Although we developed the project targeting WiFi, it is feasible to use our code for other wireless communication protocols, such as LORA. As shown in Figure 2, ESP32 devices can have LORA links by adding a LORA module. Thus, our code potentially can also be used for long range applications.

2.4. ESP-MESH Protocol

ESP-IDF provides official support for mesh networking through a self-developed protocol, ESP-MESH[2]. In this section, we briefly describe the design of this protocol to show that it does not establish a mobile ad-hoc network, thus, does not overlap with the goal of our project.

The ESP-MESH builds on top of the unique feature of ESP32 that it can behave as an access point and a client at the same time. The feature enable us to connect multiple AP-STA networks together. The resulting topology is a tree, instead of a mesh, as shown in Figure 3. The protocol will

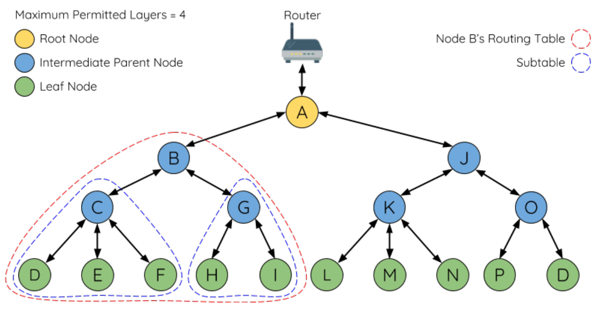

Figure 3. ESP-MESH protocol routing example

first elect a root node to work as a gateway and also the root of the tree. For other nodes in the network, they connect to the children nodes as an AP and connect to the parent nodes as a station. Figure 3 also shows the routing table for such a network. The routing algorithm depends on the tree topology, so each node will either send the packets upstream or downstream.

This mechanism is ideal if all the traffic are sent to or coming from the gateway, as it is for a lot of IoT applications. It, however, does not provide a short routing path for each two nodes in the network. And the network relies on the root node to control the network as a central entity. So this protocol does not conflict with what is presented in this report.

3. Code File Description

We publish the code of this project on Github. Readers can access it here, `https://github.com/Rui-Chun/ESP32-OLSRv2-Mesh`. We put the instructions to run the code in the README file. We only briefly describe the content of each file here.

- **./main/espnow_olsr_main.c** : The entry point of the firmware. It creates a dedicated task for OLSRv2 and dispatch event to `olsr_handlers`.
- **./main/libs/olsr_handlers.c** : It contains several handler functions for different OLSR related event.
- **./main/libs/info_base.c** : It stores all necessary information for OLSRv2 protocol as a list of entries. Each node in the network has its own entry with corresponding information.
- **./main/libs/routing_set.c** : It contains routing related functions that can be called be olsr handlers to compute routing paths.
- **./main/libs/rfc5444.c** : It defines the packet and message format, including how to generate, how to parse and how they are stored in the memory. It also provide various helper functions to process packet and message data structures.

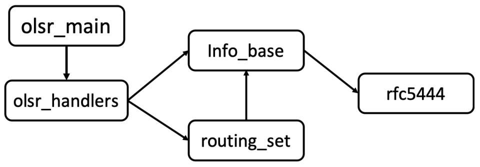

Figure 4. Overview of Code Blocks

Figure 4 shows the dependence of the major blocks of our implementation. The `olsr_main` hosts an event loop and a timer. It depends on the `olsr_handlers` to process and handle the event. The handler functions call the functions in `info_base` and `routing_set` to either update the local information or compute routing paths with current information. In order to update local information or to generate HELLO/TC messages, the proragm relies on the `rfc5444` block to provide packet and message related data structures.

4. Bottom-Up Implementation

4.1. ESPNOW Protocol

The first requirement to form a mobile ad-hoc network is to establish a peer-to-peer link. One of WiFi's operation modes is ad-hoc mode, which enables the device to establish peer-to-peer ad-hoc link. Most of the WiFi devices, however, do not support this mode due to rare usage of this feature. Unfortunately, ESP32 also does not support WiFi ad-hoc mode. Given AP and STA modes, we can probably only achieve the same thing as ESP-MESH protocol described above.

After some efforts, we find a workaround for this issue. ESP32 actually does support a connectionless, peer-to-peer communication protocol for WiFi, defined by Espressif. The underlying approach is to use the vendor-specific action frame allowed by WiFi protocol and then transmitted from one Wi-Fi device to another without connection. It is a mac layer protocol, so we implement OLSRv2 using mac address to identify every node.

One practical issue of ESPNOW is that it only supports up to 250 bytes of data to be transmitted. This is too short a frame for most of the applicaitons. To solve this, we implement a basic frame aggregation feature on top of ESPNOW. When transmitting a lerge packet, it will be sent out as a series of frames. The receiver side check and assemble the frames as the whole packet.

4.2. RFC5444 Packet

The next step is to define the packet format to be transmited through ESPNOW protocol. As stated in OLSRv2

RFC7181[4], the packet format uses the definitions in Generalized Mobile Ad Hoc Network (MANET) Packet/Message Format, RFC5444. This format, however, is very flexible and extensible. It does not define the specific layout of a packet or message, but rather defines all the possible components of a message or a packet. For example, a packet can contain one or multiple messages with different types. The receiver can not make any assumption about the order of the message layout. This makes it very complex to implement a generator and parser for RFC5444 packets. So we make some simplifications and add some assumptions about the layout ordering.

Here are part of the code in rfc5444.h, which defines the related data structures. We use this as an example to show how packets and messages are defines in our implementation. For RFC5444 packet structure, it contains a simple packet header and pointers to message structures. The order between possible messages are fixed, while not every messages have to exist in the packet. Same principle applies to the HELLO message. The message structure has a message header and multiple pointers to tlv blocks and a address block. The pointers are set as NULL if not data is present in these fields.

```
typedef struct hello_msg_t {
    msg_header_t header;
    // pointers to blocks
    tlv_block_t* msg_tlv_block_ptr;
    addr_block_t* addr_block_ptr;
    tlv_block_t* addr_tlv_block_ptr;
} hello_msg_t;

typedef struct rfc5444_pkt_t {
    // packet header
    uint8_t version;
    uint8_t pkt_flags;
    uint16_t pkt_len;
    hello_msg_t* hello_msg_ptr;
    tc_msg_t* tc_msg_ptr;
} rfc5444_pkt_t;
```

When sending a packet, We first generate a packet structure by dynamically allocating memory and setting all the fields. Then the program goes through the packet structure and copy all the data into the packet buffer. We can not allocate a continuous chunk of memory to hold the packet because the program can not know the exact layout of the packet ahead of time. To receiving a packet, we also create a rfc5444 packet structure to represent the packet.

4.3. Event Loop and Timer

We create a FreeRTOS task to handle all OLSRv2 related events through an event loop. All the sending and receiving events are put in a queue for the event to process. We create the event queue also use FreeRTOS support, which is safe to access from different tasks. So our OLSRv2 implementation can interact safely with the other tasks running on the same device.

We also use a timer to periodically schedule an event to support time related events, such as regularly sending HELLO message, deleting timeout local information. The timer event contains a local timestamp for the event handler as a reference. Global time synchronization in the network can be supported, but not implemented yet. Thus, we do not use RFC5497 as specified in OLSRv2. This is one of the limitation of current implementation.

4.4. Neighbor Discovery

OLSRv2 uses HELLO message defined in RFC6130 to discover its neighborhood nodes. Each node regularly sends out HELLO message according to a configurable time interval. And every node listens to the channel to process the HELLO messages from neighbor nodes. In our implementation, we use a simpler format of HELLO message but it carries similar information.

In our implementation, we keep an entry for every node in the information base locally. We define three types of entries, i.e. neighbor entry, two-hop entry, and remote entry. We also maintain a list of pointers for each type of entry so that the program can index them quickly.

The code below shows the data structure of a neighboring node entry. It contains all the information for a certain neighboring node. If a node hears a HELLO message, first, it checks that whether this node has been recorded. If not, it creates a new entry for this node and copy all the information from the HELLO message into the entry. This information is only valid for a certain period of time after the receiving, if no new HELLO message arrives to fresh the valid_time field, this entry will be deleted after tiemout.

```
typedef struct neighbor_entry_t {
    uint8_t entry_type;
    uint8_t peer_id; // node id
    uint32_t msg_seq_num;
    uint32_t valid_until;
    link_status_t link_status;
    uint8_t link_metric;
    uint8_t in_link_metric;
    uint8_t is_mpr_willing;
    flooding_mpr_status_t flooding_status;
    routing_mpr_status_t routing_status;
    link_info_t link_info;
    routing_info_t routing_info;
} neighbor_entry_t;
```

The program also records local processing results for a node in the entry, such as MPR selection, bidirectional link metrics, and routing paths. The node shares some of them in the messages sending out, but some fields are for local use only, such as routing decision.

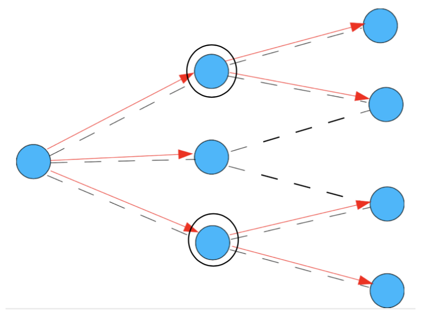

Figure 5. Example of MPR selection: nodes marked with black circles are flooding MPRs

4.5. MPR Selection

From the HELLO messages of neighboring nodes, we have its link information, from which we knows the information of two-hop nodes. Based on that, we select the flooding and routing MPRs. We use the example algorithm described in the Appendix B. of RDC8171. The algorithm aims to find a minimal subset of neighboring nodes that cover all the symmetrically connected twp-hop nodes with a minimal link metric as well.

One thing to note that this algorithm is that is can be used for both flooding and routing MPR selection. But it should use out-going link metrics for flooding MPRs and use in-coming metrics for routing MPRs. This corresponds to the usage of flooding MPRs and routing MPRs. For flooding MPRs, they forward out-going packets for their MPR selectors; for routing MPRs, they broadcast the link information for their MPR selectors so that other nodes can find the shortest in-coming routing paths. An example is given in Figure 5

In our implementation, we support the out-going and in-coming metric storage and processing. However, we have not implemented any link metric calculation algorithms. So we use the hop number as the linke metric, but we wrote the code with explicit support for bidirectional link metrics.

4.6. Routing Calculation

After selecting the MPRs, the nodes are able to generate TC(topology control) messages and flood them through the network.

The TC messages contain the link information of links between a routing MPR and its MPR selectors. Nodes not selected as routing MPRs do not need to generate TC messages, which means the topology of the network is reduced. This topology reduction does not affect the calculation of shortest path because routing MPR selection algorithm has identified the shorted paths within two hops.

During flooding process, a node will only forward the messages form its flooding MPR selectors, which reduces the number of transmission in the shared channel. After flooding, all nodes should have the topology of the whole network. Then we use Dijkstra's algorithm to compute the shortest paths to every other node and then record the next hop in the corresponding node information entry.

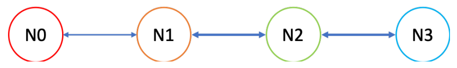

Figure 6. Topology of Experiment 1

4.7. Debug Printing

To show whether the implementation works correctly, the program provides detailed logs, including the calling sequence of functions and routing information. Every node also prints its local information about the network topology and routing paths. The ESP32 devices will send the debug logs to the serial port, so users can acquire debug logs by connecting a computer with the devices and read the serial port. We use this feature to perform several tests as shown in the next section.

5. Experiments

To demonstrate the functionality of our implementation, we perform several tests. We deploy several ESP32 devices and read the debug logs form some of the devices to show that a network has been established.

We reduce the transmitting power of every node to decrease the range of network, so that we can perform the tests indoor. But this leads to the issue that the wireless channel fluctuates a lot when human moves around the house. So it is very hard to get the whole picture of network topology.

5.1. Topology 1

In this test, we place four nodes in a line. Each node can only communicate with two nodes next to it, as shown in Figure 6.

You may access the debug output from this website, `https://github.com/Rui-Chun/ESP32-OLSRv2-Mesh/blob/main/test_logs.md`. These records are not shown here to give a better viewing experience.

To sum up the results, our implementation matches well with the deployment topology by labeling the type of every node correctly. It also finds out the correct MPRs and routing paths.

5.2. Topology 2

We also tested the implementation with a more complex deployment, as shown in Figure 7. But we only capture the links visible to N0 and N1, because the wireless links change dynamically. If we move to the node N2

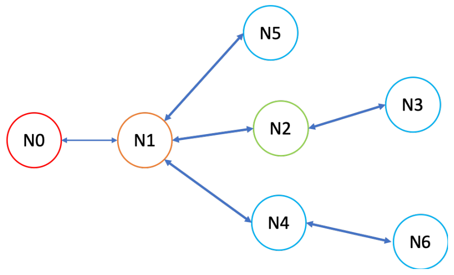

Figure 7. Topology of Experiment 2

and try to collect data, the existence of human body will affect the links. The output log of N0 and N1 are available here, `https://github.com/Rui-Chun/ESP32-OLSRv2-Mesh/blob/main/test_logs.md`.

5.3. Mobility

We also performed tests to understand how OLSRv2 behaves when devices move around. From our observations, the node gains the new topology information quickly. The exact time depends on the interval of HELLO/TC messages.

6. Discussions

6.1. Potential Improvements

For now, this implementation only allows ESP32 devices to form a mobile ad-hoc networka and compute the routing path. There are several improvements that can be done in the futuer to make this implementation more useful to all the users of ESP32 devices.

- Adding data message support. Our current implementation does not yet support data packets. It can be improved by adding a data message support.
- Integrate with other high level protocols. Our implementation is a mac layer protocol which can be integrated with high level protocol like lwip so that users can use application protocols like HTTP on top of OLSRv2 implementation.
- By using the feature of AP+STA, we can turn certain ESP32 nodes into gateways that connect with the Internet.

6.2. Limitations Compared with RFC7181

As discussed in previous sections, this implementation does not fully comply with RFC7181. Except several simplifications, we also skip several parts of OLSRv2 due to limited time.

- We do not support MPR willingness setting and broadcasting.
- We do not support multiple networking interfaces on one device that are all connected to the mobile ad-hoc netowrk.
- We do not support the declaration of gateway and attached networks.

References

[1] ESP-IDF(IoT development framework). `https://github.com/espressif/esp-idf`.

[2] ESP-MESH protocol. `https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-guides/mesh.html`.

[3] NHDP RFC6130. `https://datatracker.ietf.org/doc/html/rfc6130`.

[4] OLSRv2 RFC7181. `https://datatracker.ietf.org/doc/html/rfc7181`.

[5] RFC5444. `https://datatracker.ietf.org/doc/html/rfc5444`.

[6] RFC5497. `https://datatracker.ietf.org/doc/html/rfc5497`.

[7] T. R. Halford, K. M. Chugg, and A. Polydoros. Barrage relay networks: System protocol design. In *21st Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1133–1138, 2010.