Banana Tree Protocol implemented on ESP32

André Silva
up201808899@fe.up.pt

Carla Jorge
up201505033@fe.up.pt

Nuno Nogueira
up201808905@fe.up.pt

***Abstract*—Tree based architectures suffer from overhead and high latency between connections if they are not changed within protocol execution. Banana Tree Protocol is a tree application protocol that has the same architecture problem. The implementation in ESP32 devices aims to verify the BTP characteristics, times and response applied on embedded systems. This system relies on FreeRTOS to achieve a real time communication system, which includes tasks to send and receive packets, as well as to control what to do with them. The implemented system main functionalities are sending a 'ping' to another node, change the state of the ESP32 built-in LED and the reconfiguration of the tree. It is then studied all the results to check this protocol efficiency. All this work was accomplished for the context of Distributed Systems course unit.**

***Index Terms*—Banana Tree Protocol, ESP32, UDP, overlay network, reconfiguration**

I. Problem Definition

Banana Tree Protocol (BTP) [1] relies on an overlay network application that can be used to send multicast packets in a tree architecture, that may overhead the parent of a node in the tree and cause a higher latency on the network. To solve this problem, it is necessary to cause a reconfiguration within the network.

The ESP32 is a low-cost and low-power device system on a chip with integrated Wi-Fi [2] which allows BTP to be implemented. Each ESP32 is named as a **node** in the network. These devices are connected within a overlay network with a tree architecture as requirement of the BTP and they all run the same program. To solve the previous stated problem, the ESP32 devices must be able to reconfigure themselves within the network and avoid latency between them.

The system should allow a few nodes in the network, however only 3 and 4 nodes were tested at the same time.

II. Solution Description

A. Real Time Operating System

In order to achieve UDP [3] real time communication between the nodes, we used *FreeRTOS* [4] to implement the *BTP*. The ESP32 micro-controller is specially used for implementing *FreeRTOS* based software. One important requirement to achieve a real time communication with UDP is to read and send packets with parallelism. This is solved by recurring to the FreeRTOS task handler.

1) Implemented tasks:

- **Read task -** periodic task with 10ms period which checks for newly received UDP packets. This is the first task to be created and executed. Takes maximum of 210μs to execute.
- **Control task -** sporadic task that handles the command input. This task calls a task to check the serial port for commands and suspends it self while it doesn't get any command. If it receives any command and it that requires sending a UDP packet, it invokes a task to send the packet and goes back to the waiting for serial command state. Takes maximum of 190μs to execute.
- **Check serial port task -** periodic task with 20ms period that checks for commands from the serial port. It is invoked by the control task and resumes its execution if it receives a valid command. Takes maximum of 14μs to execute.
- **Send task -** sporadic task that creates the UDP packet and sends it. It is invoked either by the read task and/or control task. It only runs once per invocation. Takes maximum of 160μs to execute.

2) Gantt charts: In order to check the task schedulability, we made 3 Gantt charts (with milliseconds timescale) which provide a timeline of what would be the worst case scenarios for the tasks execution.

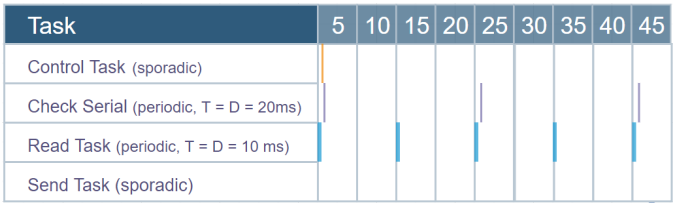

Fig. 1. Gantt chart for normal execution.

The Gantt chart in Figure 1 shows the tasks execution timeline in a normal situation, which means it is waiting for a serial command or a received packet. The CPU resource utilization is very low and the schedulability is achieved without any problem.

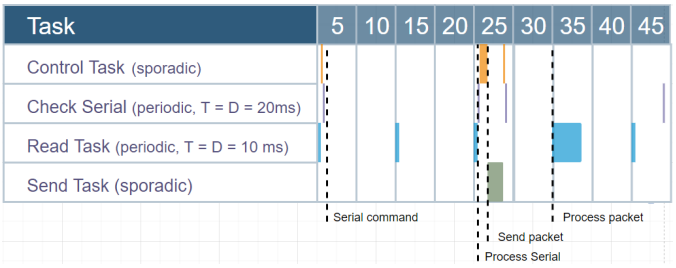

Fig. 2. Gantt chart for receiving serial command (worst case scenario).

The Gantt chart in Figure 2 presents a worst case scenario of receiving a serial command, in the instant the task *Check Serial* terminates its execution. The next time this task executes, it resumes the *Control* task execution and eliminates it self. The *Control* task will then process which command it received and will take an action accordingly, in this case send a packet. It then receives a response packet that is processed the next time *Read* task is executed. Then it returns to the normal execution. The CPU resource utilization is still very low and achieves schedulability easily.

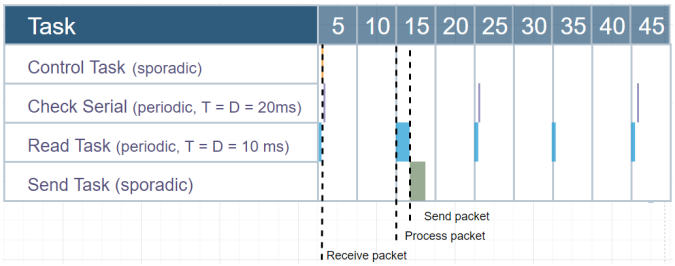

Fig. 3. Gantt chart for receiving a packet (worst case scenario).

The Gantt chart in Figure 2 reveals a worst case scenario of receiving a packet, in the instant the task *Read* terminates its execution. The next time this task executes, it reads the packet and acts accordingly. In this case, it sends a packet. It then returns to its normal execution. Once again, the schedulability is maintained.

B. Routing Algorithm

In the application implemented the routing algorithm used to established communication between the nodes has as its main functions:

- If the destination node is directly connected to the sender node, its a parent or a child, the communication is only established between these two nodes.
- If the destination node is unknown from the current node, it will forward the packet to every node with exception of the previous node.

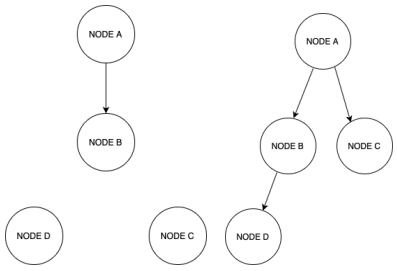

Fig. 4. Routing algorithm layout. First rule on the left, second rule on the right.

To implement the algorithm, it was create an UDP packet with 4 parameters as we can observe in figure 5.

The parameter *FROM* corresponds to the id destination node, the *TO* to the id of origin node, the *MESSAGE* to the

Fig. 5. Packet parameters

instruction to be performed and the *TARZAN* to the id of the last node that sent the packet.

C. Ping command

To check if there is communication between the nodes within the tree and if the tree is in its correct layout, it is implemented a ”Ping” functionality. In the figures II-C, we can observe the messages exchange between the nodes, when the node C pings the node D.

The node that wants to check if other node is in the network, sends a ’hello’ that is sent to the destination through the respectively parents and/or children. The node receiving a ’hello’ sends back a ’hello to you too’ working as an acknowledge from the received ’hello’.

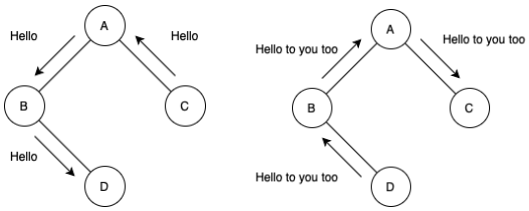

Fig. 6. ”Ping” command from node C to node D

D. LED command

The LED command functionality behavior is very similar to the ”ping”, with exception that there isn’t a acknowledge. Every time, a node receives the LED command message the state of the built-in LED of the ESP32 changes.

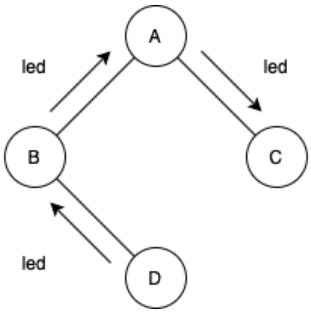

Fig. 7. Led command sequence

E. Tree reconfiguration

In this application is possible for the nodes at the ends of the tree to change its parent. To choose the new parent the node runs a simulated algorithm where the ”cost” of the connection to the other nodes in tree is calculated through a random function. If the ”cost” to a node is smaller than the current ”cost” of the connection to the parent, a reconfiguration

of the tree starts. After, the nodes sends a request to the new parent, who, then, sends a message accepting the new son.The node sends to the old parent a message informing that he is not his parent anymore. And another to the new parent notifying that he is already directly connected to him.

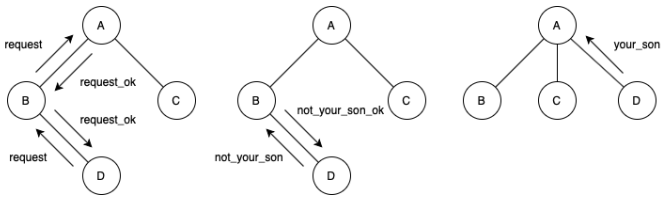

Fig. 8. Reconfiguration of node D

III. Results

The number of samples is small (10) and, since this is tested in a embedded environment, it cannot take any definitive assumption. But it is enough to have some application results and compare them to the expected behavior.

A. Ping times

In order to measure the ping delay, a ping is sent and the time is saved right when the transmission node receives a acknowledge to the 'ping'. The 'ping' command was measured 10 times giving times of execution between 647ms(m2L) and 880ms(M2L) for 2 layers layout and time between 1390ms(m3L) and 1568ms(M3L) for a 3 layers layout. As seen in the figure 9, the average time of the system 'ping' is 1095ms, basically 1 second.

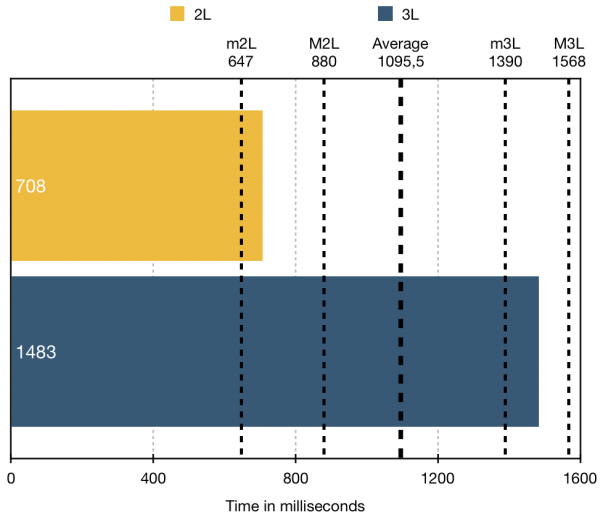

Fig. 9. Average time of 'ping' between 2 layers and 3 layers node configurations.

The standard deviation for 2 layers layout is 118,9ms and 89,3ms for 3 layers layout. These values proves that, due to the standard deviation for 3 layers being smaller than the 2 layers deviation, the system has no major consequence by adding more nodes to the system and keeps the same execution performance even if more nodes are added to the system.

B. Reconfiguration times

The reconfiguration was tested from 2 layers to 3 layers, 3 layers to 2 layers and for 3 layers to 3 layers. The behavior of the reconfiguration from 2 to 3 layers or from 3 to 2 layers was identical and it is considered as a one type reconfiguration.

A reconfiguration from 2 to 3 layers took $\approx 2,276s$ and from 3 to 2, $\approx 2,17s$. As expected, both reconfiguration had very identical execution times and may be considered as a one type reconfiguration.

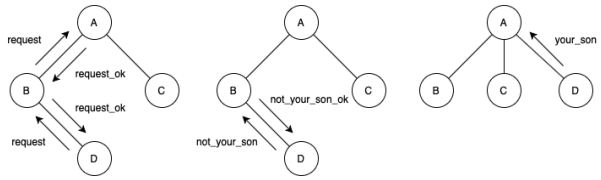

Fig. 10. Reconfiguration from a 3 to 2 layers layout.

A reconfiguration from 3 layers to 3 layers took $\approx 3,1s$. As expected, took more execution time, which may be explained by the higher number of nodes that had to process the initial message.

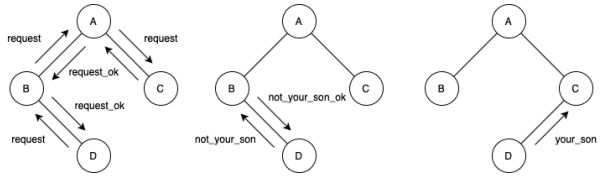

Fig. 11. Reconfiguration from a 3 to 3 layers layout.

IV. Critical Review

This project implementation was somehow challenging since there is no other information about this protocol other than what is in the original article and in a few slides from this course. Understanding the concept of this protocol and planning to add our own features to make our own version took quite some time of the project duration, but after all that, it was easy to implement. The reason for this easiness was due to the team members experience in embedded systems and real time operating systems.

It was learned to implement an overlay network, specially in an embedded environment, UDP communication protocol for both unicast and multicast and the knowledge in C++ was extended.

For future work, it would be interesting to implement a proper election process that calculates the strength of which connection (based on its latency) between two nodes instead of using pseudo-random values to simulate this behavior.

As for the team members contribution to this project, all the work that was done was well distributed between them, making a contribution percentage of 33% for each member.

REFERENCES

[1] Helder, D and Jamin, Sugih (2000) *Banana tree protocol, an end-host multicast protocol*

[2] Espressif API https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/index.html

[3] ArduinoWiFi https://www.arduino.cc/en/Reference/WiFi

[4] FreeRTOS https://www.freertos.org