Guía de Arquitectura: Red, micro-ROS y Localización 2D para un Proyecto Multi-Robot

Generado por Asistente de IA

November 6, 2025

Contents

1. Visión General del Proyecto

Este documento sirve como una guía técnica completa para el diseño e implementación de la infraestructura de red, el sistema de comunicación y el módulo de localización para un proyecto de robótica móvil basado en ROS 2. El objetivo es establecer una arquitectura robusta y escalable que permita la comunicación entre **múltiples robots** (ESP32 con micro-ROS) y un sistema de control y visualización centralizado, utilizando cámaras y AprilTags para la localización en un plano 2D.

2. Topología de Red para un Ecosistema ROS 2 Robusto

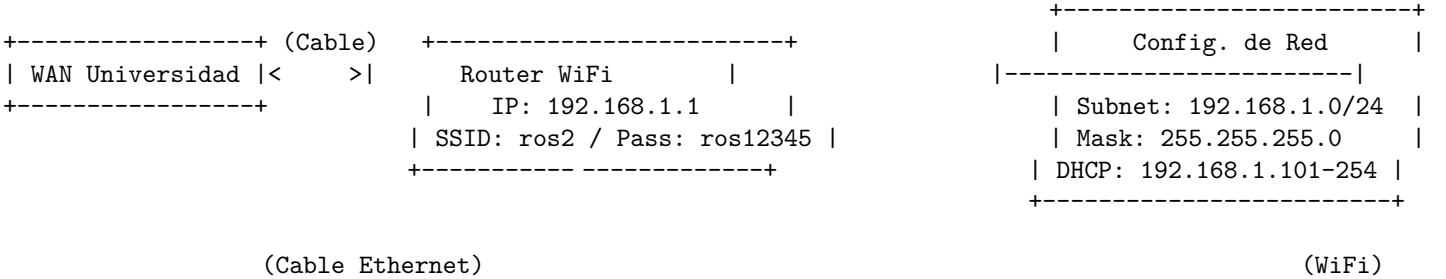
Para que ROS 2 y micro-ROS funcionen de manera fiable, es imprescindible una red local bien configurada donde todos los nodos puedan descubrirse y comunicarse entre sí. La topología recomendada es una **topología en estrella centralizada**, gestionada por un único router WiFi.

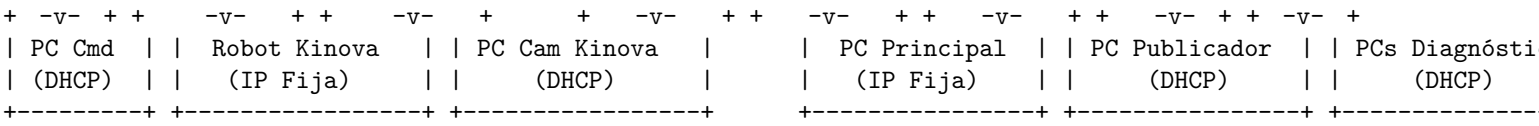
2.1. Componentes del Sistema

Componente	Rol Principal	Conexión	IP Asignada
WAN Universidad	Provee acceso a internet (actualizaciones, etc.).	Cable Ethernet	N/A (Externa)
Router WiFi	Cerebro de la red. Gestiona DHCP, DNS, NAT y tráfico.	Conectado a la WAN	`192.168.1.1`
Robot Kinova	Ejecuta sus propios nodos ROS 2 para control y estado.	Cable Ethernet	**Fija:** `192.168.1.10`
PC Comando Kinova	Envía comandos de alto nivel al robot Kinova.	Cable Ethernet	DHCP
PC Cámara Kinova	Procesa el stream de vídeo de la cámara del Kinova.	Cable Ethernet	DHCP
PC Principal	Ejecuta el `micro-ros-agent` para los robots ESP32.	WiFi	**Fija:** `192.168.1.100`
PC Publicador	Procesa imágenes de cámaras para localización (AprilTags).	WiFi	DHCP
PCs Diagnóstico (x2)	Ejecutan herramientas de introspección como RQT.	WiFi	DHCP
Robots (ESP32) (x2)	Clientes micro-ROS, ejecutan control de bajo nivel.	WiFi	DHCP
Cámara Aérea	Captura de vídeo del área de trabajo.	WiFi	DHCP
Celular (Tether)	Configuración y monitoreo del router.	WiFi 6	DHCP
Cámara Cableada	Captura de vídeo alternativa o complementaria.	USB al PC Publicador	N/A

2.2. Diagrama de la Arquitectura de Red

El siguiente diagrama ilustra cómo todos los componentes se interconectan a través del router central, asegurando que todos residan en la misma subred (ej. `192.168.1.xxx`).





2.3. Configuración del Hardware de Red

2.3.1. Router WiFi / Punto de Acceso

- **Función:** Crear la red WiFi (`ros`), actuar como servidor DHCP para la subred `192.168.1.0/24`, y gestionar el tráfico entre la red local y la WAN.
- **Configuración Crítica:**
  1. **SSID y Contraseña:** Configurar el nombre de red (SSID) a `ros2` y la contraseña a `ros12345` para facilitar la conexión durante el desarrollo.
  2. **Aislamiento de Clientes (AP Isolation): DESACTIVAR** esta función. Si está activada, los dispositivos WiFi no podrán comunicarse entre sí, rompiendo la comunicación de ROS 2.
  3. **Reserva de DHCP / IPs Fijas:** Es crucial que los servidores clave tengan direcciones predecibles.
     - **PC Principal:** Asignar la IP fija `192.168.1.100` mediante reserva de DHCP. Los clientes micro-ROS (ESP32) apuntarán a esta dirección.
     - **Robot Kinova:** Configurar la IP fija `192.168.1.10` directamente en la interfaz del robot. Se recomienda también crear una reserva en el router para evitar conflictos.

2.3.2. Adaptadores de Red de los PCs

- **PC Principal (Agente):** Debe tener la IP fija configurada a través de la reserva DHCP del router.
- **Resto de Dispositivos:** Todos los demás componentes (PCs, cámaras, ESP32) están configurados para obtener su dirección IP automáticamente del servidor DHCP del router. Esto simplifica la gestión de la red, ya que no es necesario configurar manualmente cada nuevo dispositivo.

2.4. Configuración de Variables de Ambiente de ROS 2

Para que los nodos de ROS 2 en diferentes computadores se descubran y comuniquen correctamente, es fundamental que compartan la misma configuración de "dominio".

- `ROS_DOMAIN_ID`: Esta variable segmenta la red. Solo los nodos que tengan el mismo `ROS_DOMAIN_ID` podrán verse entre sí. Se utilizará el valor por defecto `0`. Esto es simple, pero significa que si hay otros sistemas ROS 2 en la misma red física (ej. la red de la universidad), podrían interferir. Para este proyecto aislado, es aceptable. **Todos los PCs de tu proyecto deben usar el mismo ID.**
- `ROS_LOCALHOST_ONLY`: Si esta variable está configurada en `1`, la comunicación de ROS 2 se restringirá únicamente al computador local (`localhost`), impidiendo la comunicación por la red. Asegúrate de que esta variable no esté configurada o esté explícitamente en `0`.

Para aplicar esta configuración de forma permanente en un sistema Linux (Ubuntu), edita el archivo `.bashrc` en tu directorio de usuario:

```
# Abrir el archivo de configuración de la terminal
gedit ~/.bashrc
```

Añade las siguientes líneas al final del archivo:

```
# Configuración de ROS 2 para la red del proyecto
export ROS_DOMAIN_ID=0
export ROS_LOCALHOST_ONLY=0
```

Guarda el archivo, cierra la terminal y abre una nueva para que los cambios surtan efecto. Repite este proceso en **todos los PCs** que participarán en la red ROS 2.

2.5. Recomendaciones sobre el Firewall

El sistema de comunicación de ROS 2 (DDS) utiliza varios puertos UDP para el descubrimiento y el intercambio de datos. Los firewalls del sistema operativo a menudo bloquean este tipo de tráfico por defecto, lo que puede impedir que los nodos se vean entre sí.

ADVERTENCIA DE SEGURIDAD: Deshabilitar el firewall solo debe hacerse en una **red local controlada y de confianza**, como la red de pruebas dedicada para este proyecto. **NUNCA** desactives el firewall en un PC conectado directamente a una red pública o no segura.

Para fines de desarrollo y depuración en un entorno controlado, la forma más sencilla de descartar problemas de red es deshabilitar temporalmente el firewall.

En Ubuntu/Linux (usando `ufw`):

```
# Comprobar el estado actual del firewall
sudo ufw status

# Deshabilitar el firewall
sudo ufw disable

# Para volver a habilitarlo después de las pruebas
# sudo ufw enable
```

En Windows: Busca "Firewall de Windows Defender" en el menú de inicio y desactívalo para las redes "Privadas".

3. Comunicación Multi-Robot con un Único Agente micro-ROS

El `micro-ros-agent` está diseñado para ser un servidor de alto rendimiento que puede gestionar múltiples clientes (ESP32) de forma simultánea. No se requiere un agente por robot.

3.1. Rol del Agente como Proxy Inteligente

El agente no reenvía indiscriminadamente todo el tráfico de la red a los microcontroladores. Actúa como un guardián (gatekeeper):

1. Cada ESP32, al iniciarse, se suscribe a los tópicos que le interesan (ej. `/robot_A/cmd_vel`).
2. Informa al agente de estas suscripciones.
3. El agente escucha en la red ROS 2 principal y solo reenvía a una ESP32 específica los mensajes de los tópicos a los que esta se ha suscrito.

Esto protege a los microcontroladores de ser inundados con datos irrelevantes (ej. imágenes de alta resolución, nubes de puntos LiDAR).

3.2. Configuración del Agente y los Clientes (ESP32)

3.2.1. Ejecución del Agente

En el **PC Principal** (`192.168.1.100`), ejecutar:

```
ros2 run micro_ros_agent micro_ros_agent udp4 --port 8888
```

3.2.2. Configuración del Firmware de las ESP32

Todas las ESP32 se configuran para apuntar a la IP fija del agente. La diferenciación se logra mediante **namespaces**.

Código Base en cada ESP32:

```
// --- Configuración de Red (Igual en todas las ESP32) ---
const char* ssid = "ros2";
const char* password = "ros12345";
const char* agent_ip = "192.168.1.100"; // IP fija del PC con el agente
const uint16_t agent_port = 8888;

// --- Inicializacion del transporte micro-ROS ---
void setup()
{
  // Conecta el WiFi y establece el enlace UDP con el agente
  set_microros_wifi_transports(ssid, password, agent_ip, agent_port);

  // Verifica que el agente responda antes de crear nodos o publicar
  if (!rmw_uros_ping_agent(1000, 5)) {
    // Implementar reintentos o modo seguro si no hay respuesta
    return;
  }

  init_micro_ros_nodes(); // Funcion del usuario donde se crean los nodos y publishers
}
```

3.2.3. Suscripción a tópicos desde la ESP32

Dentro de `init_micro_ros_nodes()` (o la función equivalente en tu firmware) se deben crear los nodos, asignar namespaces y registrar los `subscriptions`. Cada ESP32, en este escenario, se suscribe a **dos tópicos**: `/aruco/pose2d` (pose estimada por visión) y `/move_to_pose/goal` (objetivo que debe alcanzar). Un ejemplo con el stack `rclc`, asumiendo que el primer tópico usa `geometry_msgs/msg/Pose2D` y el segundo `geometry_msgs/msg/PoseStamped`, podría verse así:

```
#include <rcl/rcl.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>
#include <geometry_msgs/msg/pose2_d.h>
#include <geometry_msgs/msg/pose_stamped.h>
```

```
static rcl_node_t node;
static rcl_subscription_t sub_pose_aruco;
static rcl_subscription_t sub_move_goal;
static geometry_msgs__msg__Pose2D pose_aruco_msg;
static geometry_msgs__msg__PoseStamped move_goal_msg;
static rclc_executor_t executor;

void aruco_pose_callback(const void * msg_in)
{
  const geometry_msgs__msg__Pose2D * msg = (const geometry_msgs__msg__Pose2D *)msg_in;
  // Actualiza la estimacion interna de pose con el dato de vision
}

void move_goal_callback(const void * msg_in)
{
  const geometry_msgs__msg__PoseStamped * msg = (const geometry_msgs__msg__PoseStamped *)msg_in;
  // Replanifica trayectorias o activa controladores hacia la meta solicitada
}

void init_micro_ros_nodes()
{
  rcl_allocator_t allocator = rcl_get_default_allocator();

  rclc_support_t support;
  rclc_support_init(&support, 0, NULL, &allocator);

  rcl_node_options_t node_ops = rcl_node_get_default_options();
  node_ops.namespace_ = ROBOT_ID; // Usa el namespace definido para cada robot
  rclc_node_init_default(&node, "motor_controller", ROBOT_ID, &support);

  rclc_subscription_init_default(
    &sub_pose_aruco,
    &node,
    ROSIDL_GET_MSG_TYPE_SUPPORT(geometry_msgs, msg, Pose2D),
    "aruco/pose2d" // El tópico final será /<ROBOT_ID>/aruco/pose2d
  );

  rclc_subscription_init_default(
    &sub_move_goal,
    &node,
    ROSIDL_GET_MSG_TYPE_SUPPORT(geometry_msgs, msg, PoseStamped),
    "move_to_pose/goal" // El tópico final será /<ROBOT_ID>/move_to_pose/goal
  );

  rclc_executor_init(&executor, &support.context, 2, &allocator);
  rclc_executor_add_subscription(&executor, &sub_pose_aruco, &pose_aruco_msg, &aruco_pose_callback, ON_NEW_DAT
  rclc_executor_add_subscription(&executor, &sub_move_goal, &move_goal_msg, &move_goal_callback, ON_NEW_DATA);
}
```

En el bucle principal (`loop()`), recuerda llamar periódicamente a `rclc_executor_spin_some(&executor, RCL_MS_TO_NS(10));` para procesar mensajes entrantes y ejecutar el callback.

Diferenciación con Namespaces:

En el firmware del Robot A:

```
#define ROBOT_ID "robot_A"
// El código crea el nodo "motor_controller" en el namespace "robot_A".
// Los tópicos serán: /robot_A/odom, /robot_A/cmd_vel
```

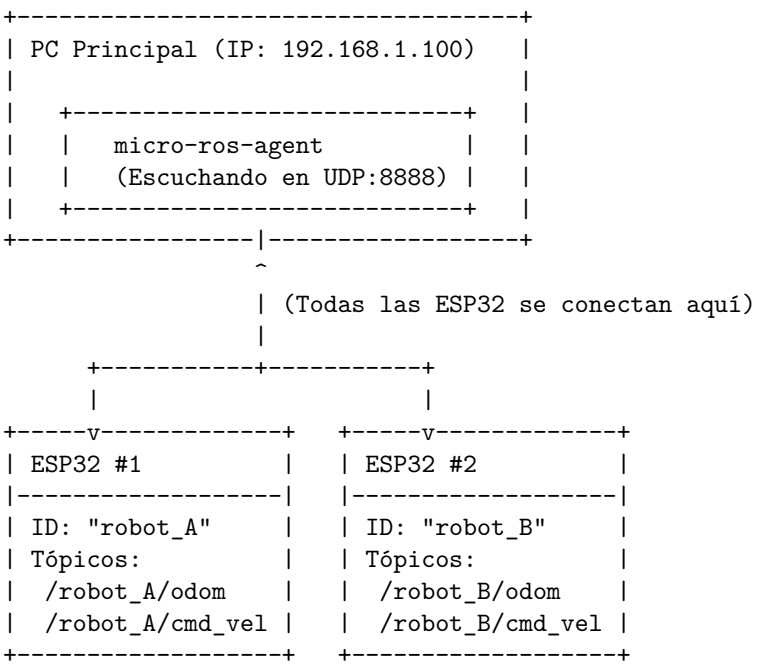
En el firmware del Robot B:

```
#define ROBOT_ID "robot_B"
// El código crea el nodo "motor_controller" en el namespace "robot_B".
// Los tópicos serán: /robot_B/odom, /robot_B/cmd_vel
```

3.3. ¿Cuándo usar más de un micro-ros-agent?

- Un solo micro-ros-agent puede atender decenas de ESP32 siempre que compartan el mismo transporte (UDP en este caso) y apunten al mismo `agent_ip:agent_port`.
- Ejecutar dos agentes simultáneamente en el mismo PC sólo es necesario cuando se desea aislar redes o transportar protocolos distintos (por ejemplo, uno en UDP y otro en serial).
- No hay conflicto si cada agente escucha en una interfaz o puerto distinto, pero los microcontroladores deben configurarse explícitamente para el agente al que se conectarán.
- Ejecutar agentes duplicados en la misma IP/puerto generará colisión; el firmware de la ESP32 se conectará solo con el primero que responda.
- Para este proyecto, mantener un único agente simplifica la gestión de namespaces, QoS y monitoreo.

3.4. Resumen Gráfico de la Conexión Multi-ESP32



4. Localización 2D con Cámaras y AprilTags

El objetivo es obtener la pose 2D (`x`, `y`, `theta`) de cada robot en un mapa fijo, utilizando una cámara aérea que detecta AprilTags. La cámara puede tener una perspectiva o inclinación. La solución robusta a este problema en ROS es el sistema de transformadas **TF2**.

4.1. El Árbol de Transformadas (TF Tree)

TF2 mantiene una relación en tiempo real entre diferentes sistemas de coordenadas (`frames`).

- `map`: El frame de referencia global y fijo del mundo.
- `camera_link`: El frame de la cámara aérea.
- `robot_A/base_link`: El frame del Robot A.
- `robot_B/base_link`: El frame del Robot B.
- `tag_N`: Frames para cada AprilTag detectado.

4.1.1. Diagrama Visual del Árbol de Transformadas

El siguiente diagrama ilustra la relación entre los diferentes `frames` y cómo TF2 los utiliza para calcular la pose final del robot.

!Árbol de Transformadas

4.2. Flujo de Trabajo y Roles de los PCs

1. **PC Publicador:**
   - Ejecuta el nodo de la cámara para publicar imágenes en un tópico (ej. `/camera/image_raw`).
   - Ejecuta un nodo de detección de AprilTags (ej. `ros2_apriltag`) que se suscribe a `/camera/image_raw`.
   - Este nodo de detección publica las poses 3D de cada tag detectado con respecto a la cámara. Crucialmente, también publica las transformadas TF desde `camera_link` hasta `tag_N`.
2. **Nodo de Localización (puede correr en cualquier PC):**
   - Este nodo utiliza TF2 para calcular la pose de cada robot (`robot_A/base_link`) con respecto al mapa (`map`).
   - Proyecta esta pose 3D a una `Pose2D` (`x`, `y`, `theta`).
   - Publica la pose 2D en un tópico dedicado para cada robot (ej. `/robot_A/pose2d`).

4.3. Proyección de la Perspectiva a un Plano 2D

El "problema" de la perspectiva de la cámara se resuelve de forma elegante con TF2. No necesitas hacer cálculos de proyección manuales.

1. **Calibración de la Cámara:** Es fundamental calibrar la cámara para corregir la distorsión de la lente. El nodo `ros2_apriltag` utiliza esta información de calibración para calcular la pose 3D de los tags con alta precisión.
2. **Definición del Mapa con Tags Fijos:** Coloca uno o más AprilTags en el suelo en posiciones conocidas. Publica sus transformadas estáticas con respecto al frame `map`. Esto ancla el sistema de coordenadas de la cámara al mundo.

   En nuestro caso, usamos un único tag (ID=0) para definir el origen del mundo. Por lo tanto, la transformada entre `map` y `tag_0` es una transformada de identidad (cero traslación, cero rotación), haciendo que ambos frames sean coincidentes.

   ```
   # El Tag ID=0 define el origen del mapa. La transformada es de identidad.
   ros2 run tf2_ros static_transform_publisher --x 0 --y 0 --z 0 \
     --frame-id map --child-frame-id tag_0
   ```

3. **Definición de la Geometría del Robot:** Pega un AprilTag único (ej. ID=100) en la parte superior de cada robot y publica su posición fija con respecto al centro del robot (`base_link`).

   ```
   # El Tag ID=100 está 6cm por encima del centro del Robot A
   ros2 run tf2_ros static_transform_publisher --z 0.06 \
     --frame-id robot_A/base_link --child-frame-id tag_100
   ```

4. **Cálculo Automático de TF2:** Cuando el nodo de detección ve `tag_0` y `tag_100` al mismo tiempo, TF2 tiene una cadena de transformación completa para calcular la pose del robot en el mapa: `map` → `tag_0` → `camera_link` → `tag_100` → `robot_A/base_link`

4.4. Código del Nodo de Localización (Python)

Este nodo solicita la transformada calculada por TF2 y extrae la información 2D.

```
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Pose2D
import tf2_ros
from tf2_ros import TransformException
from tf_transformations import euler_from_quaternion
import math

class PoseProjector(Node):
    def __init__(self, robot_ns):
        super().__init__(f'{robot_ns}_pose_projector')
        self.robot_namespace = robot_ns
        self.target_frame = f'{self.robot_namespace}/base_link'
        self.reference_frame = 'map'

        self.tf_buffer = tf2_ros.Buffer()
        self.tf_listener = tf2_ros.TransformListener(self.tf_buffer, self)

        # Publicador para la pose 2D
        self.pose_publisher = self.create_publisher(
            Pose2D,
            f'/{self.robot_namespace}/pose2d',
            10
        )

        self.timer = self.create_timer(0.1, self.on_timer) # Intentar cada 100ms

    def on_timer(self):
        try:
            # Solicitar la transformada de 'map' a 'base_link' del robot
            t = self.tf_buffer.lookup_transform(
                self.reference_frame,
                self.target_frame,
                rclpy.time.Time())
        except TransformException as ex:
            self.get_logger().info(
                f'No se pudo obtener la transformada {self.target_frame}: {ex}')
            return

        # Proyección a 2D
        pose_2d_msg = Pose2D()

        # La posición X e Y son directamente de la traslación
        pose_2d_msg.x = t.transform.translation.x
        pose_2d_msg.y = t.transform.translation.y

        # La orientación 'theta' se extrae del cuaternión
        q = t.transform.rotation
        (roll, pitch, yaw) = euler_from_quaternion([q.x, q.y, q.z, q.w])

        # El yaw es el ángulo que nos interesa en un plano 2D
        pose_2d_msg.theta = yaw

        self.pose_publisher.publish(pose_2d_msg)
        self.get_logger().info(f'Publicando pose para {self.robot_namespace}: x={pose_2d_msg.x:.2f}, y={pose_2

def main(args=None):
```

```
    rclpy.init(args=args)
    # Crear un proyector para cada robot
    projector_A = PoseProjector('robot_A')
    projector_B = PoseProjector('robot_B')

    executor = rclpy.executors.MultiThreadedExecutor()
    executor.add_node(projector_A)
    executor.add_node(projector_B)

    executor.spin()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Al ejecutar este nodo, obtendrás un tópico `/robot_A/pose2d` con la localización precisa del robot en el mapa, independientemente de la inclinación o perspectiva de la cámara, siempre que esta pueda ver al menos un tag fijo del mapa y el tag del robot.