Navegación con Nav2 y patrullaje mediante FSM

Teoría y herramientas para la práctica (ROS 2 + C++)

Francisco Martín Rico y Rodrigo Pérez Rodríguez

Qué hay que hacer en la práctica

- Objetivo: implementar un sistema de **patrullaje autónomo** que visite secuencialmente waypoints mediante una **FSM**.
- Usar **Nav2** como capacidad de navegación (no lo implementas tú, lo utilizas como una caja negra).
- Separar claramente:
  - **Capacidad**: "Ir a una pose objetivo en el mapa" (Nav2).
  - **Misión**: "Visitar repetidamente una lista de poses" (tu FSM).
- Implementa una FSM en C++ que:
  - envíe objetivos a Nav2 mediante **actions**,
  - gestione feedback, resultados y fallos,
  - coordine la secuencia de waypoints.

Nota sobre la orquestación en esta práctica

- En este proyecto, la misión de patrullaje es tan simple que **no requiere orquestación compleja**.
- La FSM de la misión simplemente invoca la capacidad de navegación (Nav2) para ir a cada waypoint.
- No hay coordinación entre múltiples capacidades ni tareas concurrentes: la lógica es secuencial y directa.
- Esto permite centrarse en la separación **capacidad–misión** y en el uso de **actions** como interfaz estándar.
- En sistemas más complejos, la misión podría orquestar varias capacidades (navegación, manipulación, percepción, etc.) y gestionar prioridades o concurrencia.

Objetivo de esta mini-teoría

- Entender **patrones arquitectónicos** para implementar FSM en ROS 2:
  - Patrón básico (switch-case).
  - Patrón orientado a objetos (con `on_entry`, `on_do`, `on_exit`).
  - Descomposición ortogonal (múltiples FSM concurrentes).
- Ver cómo usar **actions** en ROS 2 como cliente: enviar goals, recibir feedback, gestionar resultados.

Arquitectura de la práctica: capacidad vs misión

- **Capacidad**: funcionalidad reutilizable de bajo nivel (ej. navegación).
- **Misión**: tarea de alto nivel que usa capacidades (ej. patrullaje).
- Nav2 proporciona la capacidad "navegar a una pose".
- La FSM implementa la misión "patrullar waypoints".
- La interfaz entre ambos es una *action* estándar: `NavigateToPose`.

Separación clara:

- FSM no sabe cómo navegar (usa Nav2).
- Nav2 no sabe qué waypoints visitar.
- Interfaz estándar = reutilizable.

Patrón básico: switch-case (FSM simple)

- Para FSM simples (menos de 10 estados), switch-case es directo.
- Estructura: `enum class State`, atributo `current_state_`, método `control_cycle()`.
- Callbacks de sensores **solo actualizan memoria** (no toman decisiones).
- El timer ejecuta la FSM periódicamente.

Patrón básico: switch-case (FSM simple) – Código

BasicFSMRobot.hpp (extracto)

```
class BasicFSMRobot : public rclcpp::Node
{
public:
  enum class State {
    IDLE, MOVING,
    OBSTACLE_DETECTED, STOPPED
  };

private:
  State current_state_;
  rclcpp::TimerBase::SharedPtr timer_;
  // ...
};
```

Patrón básico: ciclo de control

- El método `control_cycle()` se ejecuta periódicamente (ej. 10 Hz).
- Cada `case` ejecuta acciones y evalúa transiciones.
- Limitación: no hay `on_entry` ni `on_exit`, por lo que las acciones se repiten en cada ciclo.
- Válido para prototipos rápidos, no para sistemas complejos (es suficiente para esta práctica).

Patrón básico: ciclo de control – Código

control_cycle() (extracto)

```
1  void BasicFSMRobot::control_cycle()
2  {
3    switch (current_state_) {
4      case State::IDLE:
5        // acciones en IDLE
6        if (start_pressed_) {
7          current_state_ = State::MOVING;
8        }
9        break;
10     case State::MOVING:
11       publish_velocity(0.5, 0.0);
12       if (min_distance_ < threshold_) {
13         current_state_ = State::STOPPED;
14       }
15       break;
16     // ...
17   }
18 }
```

Patrón orientado a objetos: interfaz `State`

- Para FSM complejas, encapsular cada estado en una clase separada.
- Interfaz base `State` con tres métodos:
  - `on_entry()`: ejecutado *una vez* al entrar.
  - `on_do()`: ejecutado en cada ciclo.
  - `on_exit()`: ejecutado *una vez* al salir.
- Cada estado concreto hereda de `State`.

Patrón orientado a objetos: interfaz `State` – Código

fsm_oop_example.hpp (extracto)

```
1  class State
2  {
3  public:
4    virtual void on_entry() {}
5    virtual void on_do() {}
6    virtual void on_exit() {}
7    virtual State* check_transitions() {
8      return nullptr;
9    }
10   virtual std::string get_name() const = 0;
11 };
```

¿Qué significa `virtual`?

En C++, la palabra clave `virtual` indica que el método puede ser redefinido (sobrescrito) por las clases derivadas. Esto permite el **polimorfismo**: al llamar a un método virtual a través de un puntero o referencia a la clase base, se ejecuta la versión correspondiente a la clase derivada en tiempo de ejecución.

Patrón orientado a objetos: `StateMachine`

- La clase `StateMachine` orquesta los cambios de estado.
- Método `step()`:
  1. Ejecutar `on_do()` del estado actual.
  2. Evaluar transiciones con `check_transitions()`.
  3. Si hay transición: `on_exit()`, crear nuevo estado, `on_entry()`.
- Garantiza que `on_entry` y `on_exit` se ejecutan exactamente una vez por transición.

Patrón orientado a objetos: StateMachine – Código

StateMachine (extracto)

```
1  class StateMachine
2  {
3    State* current_state_;
4  public:
5    void step() {
6      current_state_->on_do();
7      State* next = current_state_->check_transitions();
8      if (next != nullptr) {
9        current_state_->on_exit();
10       delete current_state_;
11       current_state_ = next;
12       current_state_->on_entry();
13     }
14   }
15 };
```

Patrón OOP: ejemplo de estado concreto (`IdleState`)

- Mantiene un puntero al **contexto**, que es el nodo de ROS 2 que ejecuta la aplicación del robot (todos los estados lo necesitan).
- `on_entry()`: detiene motores *una sola vez*.
- `on_do()`: no hace nada (el robot está parado).
- `check_transitions()`: verifica condiciones y retorna el nuevo estado si procede.

Patrón OOP: IdleState – Código (1/2)

IdleState: constructor y on_entry

```
1  class IdleState : public State
2  {
3    OOPFSMRobot* robot_;
4  public:
5    explicit IdleState(OOPFSMRobot* r)
6      : robot_(r) {}
7
8    void on_entry() override {
9      robot_->publish_velocity(0.0, 0.0);
10   }
11   // ...
12 };
```

¿Qué significa override?

En C++, la palabra clave override se coloca al final de la declaración de un método para indicar explícitamente que está sobrescribiendo (redefiniendo) un método virtual de la clase base. El compilador verifica que realmente exista un método virtual con la misma firma en la clase base, ayudando a evitar errores sutiles por diferencias en el nombre o los parámetros.

Patrón OOP: IdleState – Código (2/2)

IdleState: check_transitions **y** get_name

```
1  class IdleState : public State
2  {
3    // ...
4    State* check_transitions() override {
5      if (robot_->is_start_pressed()) {
6        return new MovingState(robot_);
7      }
8      return nullptr;
9    }
10
11   std::string get_name() const override
12     { return "IDLE"; }
13 };
```

Patrón OOP: `MovingState` y `on_exit`

- `on_do()`: publica velocidad en cada ciclo (acción continua).
- `check_transitions()`: verifica si hay obstáculo.
- `on_exit()` es *crítico*: detiene motores antes de cambiar de estado, evitando que el robot siga moviéndose durante la transición.

Patrón OOP: MovingState y on_exit – Código

MovingState (extracto)

```
class MovingState : public State
{
  OOPFSMRobot* robot_;
public:
  void on_do() override {
    robot_->publish_velocity(0.5, 0.0);
  }

  State* check_transitions() override {
    if (robot_->get_min_distance()
        < robot_->get_obstacle_threshold()) {
      return new StoppedState(robot_);
    }
    return nullptr;
  }
  // ...
```

Patrón OOP: MovingState y on_exit – Código (cont.)

MovingState (extracto, continuación)

```
  void on_exit() override {
    robot_->publish_velocity(0.0, 0.0);
  }
};
```

Integración con ROS 2: `OOPFSMRobot`

- El nodo `OOPFSMRobot` encapsula la FSM y la infraestructura ROS 2.
- Constructor: configura suscripciones, publicadores, timer.
- `control_cycle()`: ejecuta `fsm_->step()` periódicamente.
- Callbacks de sensores: solo actualizan variables (`min_distance_`).
- Métodos públicos: interfaz controlada para los estados.

OOPFSMRobot (extracto)

```
class OOPFSMRobot : public rclcpp::Node
{
  StateMachine* fsm_;
  double min_distance_;
  // ...
public:
  OOPFSMRobot() : Node("oop_fsm_robot") {
    fsm_ = new StateMachine(
      new IdleState(this), get_logger());
    timer_ = create_wall_timer(
      100ms, [this]() { fsm_->step(); });
    // subs, pubs...
  }
  double get_min_distance() const
    { return min_distance_; }
  void publish_velocity(double l, double a);
};
```

Descomposición ortogonal: problema y solución

- **Problema**: En sistemas reales, el robot gestiona múltiples preocupaciones simultáneas (navegación, batería, luces, etc.).
- Una FSM plana provoca explosión combinatoria: 3 estados de navegación $\times$ 3 estados de batería $=$ 9 estados combinados.
- **Solución (Harel)**: Descomponer en **regiones ortogonales**: múltiples FSM independientes que se ejecutan concurrentemente.
- Crecimiento lineal: $3 + 3 = 6$ estados (no $3 \times 3 = 9$).

Ejemplo práctico:

- Región 1 (navegación): IDLE, MOVING, STOPPED.
- Región 2 (batería): OK, LOW, CRITICAL.
- Ambas FSM se ejecutan en paralelo en cada ciclo.

Descomposición ortogonal: `GenericStateMachine<T>`

- Clase genérica `GenericStateMachine<T>` mediante plantillas C++.
- Cada región mantiene su propia máquina de estados independiente.
- El tipo `T` puede ser `NavState`, `BatteryState`, etc.
- Permite reutilizar la lógica de la FSM para diferentes tipos de estados.

Descomposición ortogonal: GenericStateMachine<T> – Código

GenericStateMachine (extracto)

```
1  template<typename T>
2  class GenericStateMachine
3  {
4    T* current_state_;
5    std::string region_name_;
6  public:
7    GenericStateMachine(
8      T* initial, rclcpp::Logger logger,
9      std::string name)
10     : current_state_(initial)
11     , region_name_(name) {}
12   // ...
```

¿Qué significa `template<typename T>`?

En C++, `template<typename T>` permite que una clase funcione con distintos tipos. Así, `GenericStateMachine` sirve para cualquier conjunto de estados (ej. `NavState`, `BatteryState`), y puedes tener varias FSM paralelas, cada una con su propio tipo de estado.

Descomposición ortogonal: GenericStateMachine<T> – Código (cont.)

GenericStateMachine (extracto, continuación)

```
  void step() {
    current_state_->on_do();
    T* next = current_state_->check_transitions();
    if (next != nullptr) { /* ... */ }
  }
};
```

Descomposición ortogonal: regiones de navegación y batería

- Cada región tiene su propia jerarquía de estados.
- **Región navegación**: `NavState` (base) → `NavIdleState`, `NavMovingState`, `NavStoppedState`.
- **Región batería**: `BatteryState` (base) → `BatteryOKState`, `BatteryLowState`, `BatteryCriticalState`.
- Las transiciones de cada FSM se basan exclusivamente en su dominio (navegación en obstáculos, batería en porcentaje).
- Cada región define su propia interfaz base con los métodos estándar del patrón OOP.

Interfaces base (extracto)

```
// Región navegación
class NavState {
  virtual void on_entry() {}
  virtual void on_do() {}
  virtual void on_exit() {}
  virtual NavState* check_transitions()
    { return nullptr; }
};
```

Descomposición ortogonal: interfaz `BatteryState`

Interfaz BatteryState (extracto)

```
1  // Región batería
2  class BatteryState {
3    virtual void on_entry() {}
4    virtual void on_do() {}
5    virtual void on_exit() {}
6    virtual BatteryState* check_transitions()
7      { return nullptr; }
8  };
```

Descomposición ortogonal: `OrthogonalRobot`

- El nodo `OrthogonalRobot` integra ambas regiones.
- Dos FSM independientes: `nav_fsm_` y `battery_fsm_`.
- `control_cycle()`: ejecuta `step()` de *ambas* FSM en cada iteración.
- Ambas máquinas operan sobre el mismo contexto (el robot), pero gestionan aspectos ortogonales.
- Resuelve la explosión combinatoria: crecimiento lineal.

Descomposición ortogonal: `OrthogonalRobot` – Código (1/2)

OrthogonalRobot: estructura y constructor

```
class OrthogonalRobot : public rclcpp::Node
{
  GenericStateMachine<NavState>* nav_fsm_;
  GenericStateMachine<BatteryState>* battery_fsm_;
public:
  OrthogonalRobot() : Node("orthogonal_robot") {
    nav_fsm_ = new GenericStateMachine<NavState>(
      new NavIdleState(this), get_logger(), "NAV");
    battery_fsm_ = new GenericStateMachine<BatteryState>(
      new BatteryOKState(this), get_logger(), "BAT");
    timer_ = create_wall_timer(
      100ms, [this]() { control_cycle(); });
  }
  // ...
};
```

Descomposición ortogonal: `OrthogonalRobot` – Código (2/2)

OrthogonalRobot: `control_cycle`

```
class OrthogonalRobot : public rclcpp::Node
{
  // ...
private:
  void control_cycle() {
    battery_fsm_->step(); // Primero: actualizar restricciones
    nav_fsm_->step(); // Segundo: generar comandos
  }
};
```

Interacción entre regiones ortogonales

- Las regiones son *ortogonales* (independientes), pero pueden coordinarse mediante **estado compartido**.
- Ejemplo: variable `top_speed_` compartida entre ambas regiones.
- Región de batería modula `top_speed_`: 0.3 (normal), 0.15 (baja), 0.0 (crítica).
- Región de navegación publica velocidad usando `get_top_speed()`.
- Mantiene ortogonalidad: batería no conoce estados de navegación; navegación no consulta nivel de batería.

Interacción entre regiones ortogonales – Código (1/2)

BatteryCriticalState: `on_entry` **y** `on_do`

```
1  class BatteryCriticalState : public BatteryState
2  {
3    OrthogonalRobot* robot_;
4  public:
5    void on_entry() override {
6      RCLCPP_WARN(robot_->get_logger(),
7        "BATTERY CRITICAL! Stopping robot.");
8      robot_->set_top_speed(0.0); // Imponer velocidad máxima = 0
9    }
10
11   void on_do() override {
12     // Mantener restricción de velocidad cero
13     robot_->set_top_speed(0.0);
14   }
15   // ...
16 };
```

Interacción entre regiones ortogonales – Código (2/2)

BatteryCriticalState: `check_transitions`

```
class BatteryCriticalState : public BatteryState
{
  // ...
  BatteryState* check_transitions() override {
    if (robot_->get_battery_level() >
        robot_->get_critical_threshold()) {
      return new BatteryLowState(robot_);
    }
    return nullptr;
  }
};
```

Acciones (actions) en ROS 2: concepto

- Las **actions** son un patrón de comunicación para tareas de larga duración con feedback.
- Compuesto por tres mensajes:
  - **Goal**: objetivo a alcanzar (ej. pose destino).
  - **Feedback**: progreso periódico (ej. distancia restante).
  - **Result**: resultado final (éxito, fallo, cancelación).
- Ventajas sobre servicios:
  - Permiten cancelación.
  - Proporcionan feedback continuo.
  - No bloquean al cliente.
- En esta práctica: `NavigateToPose` (Nav2) es una action.

NavigationClient: adaptador de la capacidad de navegación

- **NavigationClient** encapsula toda la comunicación con Nav2.
- Propósito: ofrecer una **interfaz simple** para consumir la capacidad de navegación.
- La aplicación (FSM, tarea, misión) NO necesita conocer detalles de actions, callbacks, etc.
- Patrón: **composición** (la app contiene un `shared_ptr` al cliente).

Ventaja arquitectónica: separación clara entre CÓMO se comunica con Nav2 (cliente) y QUÉ hace la aplicación (lógica de la tarea).

NavigationClient: interfaz pública (métodos disponibles)

Métodos para verificar y activar la capacidad:

- `wait_for_action_server(timeout)` → verifica si Nav2 está listo.
- `create_pose_stamped(x, y, yaw)` → construye objetivos de navegación.
- `send_goal(target_pose)` → solicita navegación (asíncrono).

Métodos para consultar estado (no bloqueantes):

- `is_goal_active()` → ¿hay navegación en progreso?
- `is_goal_done()` → ¿terminó la navegación?
- `was_goal_successful()` → ¿fue exitosa?
- `get_feedback()` → obtiene progreso (distancia, tiempo).

Método para cancelar:

- `cancel_goal()` → aborta navegación en progreso.

Uso desde aplicación: `SimpleNavigationApp`

Patrón de uso:

1. Crear el cliente como **componente** (composición, no herencia).
2. Usar un **timer periódico** para implementar el flujo de control.
3. **Consultar estado** mediante métodos del cliente (no bloquear).
4. Progresar por **fases** claramente definidas.

Fases del flujo:

- Fase 1: Verificar disponibilidad de la capacidad.
- Fase 2: Activar la capacidad (enviar objetivo).
- Fase 3: Monitorizar progreso.
- Fase 4: Procesar resultado.

Ejemplo: SimpleNavigationApp – Estructura

```cpp
class SimpleNavigationApp : public rclcpp::Node
{
public:
  SimpleNavigationApp() : Node("simple_navigation_app") {
    // 1. Crear cliente (COMPOSICIÓN)
    // Pasar 'this' al constructor: el cliente usa este nodo
    nav_client_ = std::make_shared<NavigationClient>(this);
    // 2. Construir objetivo
    target_pose_ = nav_client_->create_pose_stamped(6.0, -2.0, 0.0);
    // 3. Timer periódico (NO BLOQUEANTE)
    timer_ = create_wall_timer(std::chrono::milliseconds(500),
      std::bind(&SimpleNavigationApp::control_cycle, this));
  }

private:
  std::shared_ptr<NavigationClient> nav_client_;
  geometry_msgs::msg::PoseStamped target_pose_;
  rclcpp::TimerBase::SharedPtr timer_;
  bool server_ready_ = false;
  bool goal_sent_ = false;
```

Ejemplo: SimpleNavigationApp – FASE 1 y 2

```
1  void control_cycle() {
2    // FASE 1: Verificar disponibilidad (wait_for_action_server)
3    if (!server_ready_) {
4      if (nav_client_->wait_for_action_server(std::chrono::seconds(1))) {
5        RCLCPP_INFO(get_logger(), "Servidor disponible");
6        server_ready_ = true;
7      }
8      return;
9    }
10
11   // FASE 2: Activar capacidad (send_goal)
12   if (!goal_sent_) {
13     RCLCPP_INFO(get_logger(), "Enviando objetivo...");
14     nav_client_->send_goal(target_pose_);
15     goal_sent_ = true;
16     return;
17   }
18   // ... continúa en siguiente slide
```

Ejemplo: SimpleNavigationApp – FASE 3 y 4

```cpp
  // FASE 3: Monitorizar (is_goal_done, get_feedback)
  if (!nav_client_->is_goal_done()) {
    auto feedback = nav_client_->get_feedback();
    if (feedback) {
      RCLCPP_INFO(get_logger(), "Distancia: %.2f m",
                  feedback->distance_remaining);
    }
    return;
  }

  // FASE 4: Procesar resultado (was_goal_successful)
  if (nav_client_->was_goal_successful()) {
    RCLCPP_INFO(get_logger(), "Navegación exitosa");
  } else {
    RCLCPP_WARN(get_logger(), "Navegación fallida");
  }

  timer_->cancel();
}
```

Preguntas