Robot Camarero con Behavior Trees

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

Francisco Martín Rico y Rodrigo Pérez Rodríguez

Nota sobre la arquitectura en esta práctica

- En este proyecto, la misión del robot camarero es lo **suficientemente simple** para modelarse directamente con un Behavior Tree.
- **No se requiere FSM** para coordinar la misión: el BT expresa naturalmente la secuencia de pasos.
- El BT invoca directamente las capacidades necesarias:
  - Navegación (Nav2).
  - Diálogo (TTS, STT, extracción de información).
  - Acciones simuladas (recoger y entregar pedido).
- Esto permite centrarse en la separación **misión–capacidad** y en el diseño de nodos BT **reutilizables**.
- En sistemas más complejos, se podría combinar FSM (capa de coordinación) con BT (capa de tareas).

Qué hay que hacer en la práctica

- Objetivo: implementar un **robot camarero** que atiende pedidos mediante un **Behavior Tree**.
- Usar **Nav2** como capacidad de navegación (no lo implementas tú, lo utilizas como una caja negra).
- Integrar capacidades de diálogo usando un cliente que hace uso del paquete `simple_hri`.
- Separar claramente:
  - **Misión (BT)**: orquesta el flujo completo del servicio.
  - **Capacidades**: navegación, diálogo, acciones simuladas.
- Implementa un BT que:
  - detecte presencia de cliente,
  - tome el pedido mediante diálogo (bebida y comida),
  - navegue a la cocina y simule recoger el pedido,
  - regrese al cliente y simule entregar el pedido,
  - vuelva al punto de espera inicial.

Objetivo de esta mini-teoría

- Entender **patrones arquitectónicos** para implementar Behavior Trees en ROS 2:
  - composición jerárquica de comportamientos,
  - nodos de control (Sequence, Fallback, Decorator),
  - nodos hoja (Condition, Action).
- Aprender a **separar estructura (XML) de implementación (C++)**:
  - definición del árbol en XML,
  - implementación de nodos personalizados en C++.
- Ver ejemplos concretos de:
  - nodos con puertos de entrada/salida,
  - integración con ROS 2 (topics, actions, HRI),
  - programa principal que carga y ejecuta el árbol.

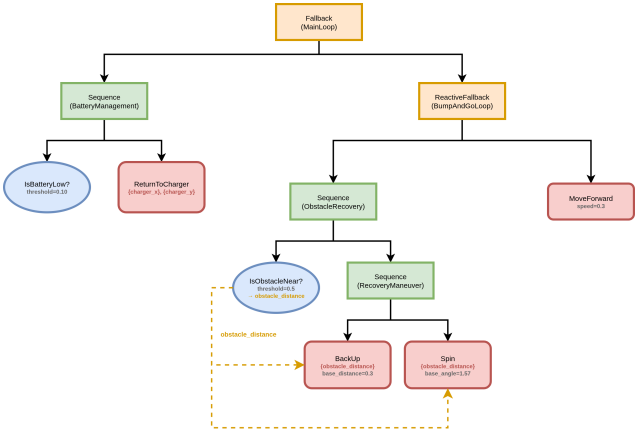
Arquitectura de dos niveles: BT + Capacidades

- **Nivel 1 - BT (misión)**: orquesta el flujo completo del servicio.
  - Define QUÉ hacer y en QUÉ orden.
  - No contiene lógica de navegación o diálogo.
  - Invoca capacidades mediante nodos hoja.
- **Nivel 2 - Capacidades**: implementan acciones atómicas.
  - Navegación: Nav2 (NavToPose action).
  - Diálogo: TTS, STT, extracción de información.
  - Acciones simuladas: temporizadores con anuncios.
- Objetivo: **separación clara** entre misión y capacidades.
- El BT define la estrategia; las capacidades ejecutan las acciones.

Ejemplo de referencia: Bump-and-Go

- Comportamiento clásico de exploración reactiva:
  - Avanza continuamente hasta detectar obstáculo.
  - Retrocede y gira para evitarlo.
  - Repite indefinidamente.
- Gestión de batería:
  - Si batería baja $\rightarrow$ interrumpe exploración.
  - Navega autónomamente a base de recarga.

Estructura del árbol bump-and-go: visualización



Estructura del árbol bump-and-go (XML) – Parte 1/2

Raíz del árbol y gestión de batería:

```xml
<root main_tree_to_execute="BumpAndGo">
  <BehaviorTree ID="BumpAndGo">
    <Fallback name="MainLoop">

      <!-- Prioridad 1: batería baja -->
      <Sequence name="BatteryManagement">
        <IsBatteryLow threshold="0.10" />
        <ReturnToCharger
          charger_x="{charger_x}"
          charger_y="{charger_y}" />
      </Sequence>

      <!-- Rama 2: exploración continua (ver siguiente slide) -->
      ...

    </Fallback>
  </BehaviorTree>
</root>
```

Estructura del árbol bump-and-go (XML) – Parte 2/2

Bucle de exploración bump-and-go:

```xml
<!-- Prioridad 2: exploración continua -->
<ReactiveFallback name="BumpAndGoLoop">
  <!-- Obstáculo: maniobra de recuperación -->
  <Sequence name="ObstacleRecovery">
    <IsObstacleNear threshold="0.5"
      obstacle_distance="{obstacle_dist}" />
    <Sequence name="RecoveryManeuver">
      <BackUp obstacle_distance="{obstacle_dist}"
        base_distance="0.3" />
      <Spin obstacle_distance="{obstacle_dist}"
        base_angle="1.57" />
    </Sequence>
  </Sequence>
  <!-- Si no hay obstáculo: avanzar -->
  <MoveForward speed="0.3" />
</ReactiveFallback>
```

Flujo de ejecución del bump-and-go

1. **Comprobación de batería (máxima prioridad)**:
   - Si `IsBatteryLow` → ejecuta `ReturnToCharger`.
   - Interrumpe cualquier actividad en curso.
2. **Bucle de exploración**:
   - `ReactiveFallback` crea un bucle continuo.
   - Reevalúa constantemente condiciones reactivas.
3. **Reacción a obstáculos**:
   - Si `IsObstacleNear` → ejecuta `BackUp` + `Spin`.
   - Distancia del obstáculo fluye por blackboard (`obstacle_dist`).
4. **Avance continuo**:
   - Si no hay obstáculo → `MoveForward`.
   - El bucle continúa indefinidamente.

Herramienta: Groot2 para diseño visual de BTs

- **Groot2**: editor gráfico para Behavior Trees disponible en https://www.behaviortree.dev/groot/
- Permite diseñar, visualizar y depurar árboles de comportamiento de forma **visual e intuitiva**.
- Características principales:
  - Interfaz tipo drag-and-drop para construir BTs sin escribir XML manualmente.
  - Visualización en tiempo real de la ejecución del árbol (qué nodos están activos).
  - Compatible con archivos XML de BehaviorTree.CPP (carga y guarda en el mismo formato).
  - Validación automática de puertos y conexiones.

Comunicación entre nodos: blackboard y puertos

- **Blackboard**: memoria compartida donde los nodos leen/escriben datos.
- **Puertos**: interfaz tipada para acceder al blackboard.
  - **InputPort**: el nodo lee un valor del blackboard.
  - **OutputPort**: el nodo escribe un valor al blackboard.
- Ejemplo:
  - `IsObstacleNear` *produce* `obstacle_distance` (OutputPort).
  - `BackUp` y `Spin` *consumen* `obstacle_distance` (InputPort).
- Ventajas:
  - Comunicación explícita y tipada.
  - Sin acoplamiento directo entre nodos.
  - Facilita reutilización.

Nodo personalizado: IsObstacleNear (estructura) – 1/2

```
1  class IsObstacleNearCondition : public BT::ConditionNode
2  {
3  public:
4    IsObstacleNearCondition(const std::string& name,
5                            const BT::NodeConfiguration& config);
6    static BT::PortsList providedPorts() {
7      return {
8        BT::InputPort<double>("threshold", 0.5),
9        BT::OutputPort<double>("obstacle_distance")
10     };
11   }
12   BT::NodeStatus tick() override;
```

Nodo personalizado: IsObstacleNear (estructura) – 2/2

```
1  class IsObstacleNearCondition : public BT::ConditionNode
2  {
3    // ...continuación...
4  private:
5    rclcpp::Node::SharedPtr node_; // obtenido del blackboard
6    rclcpp::Subscription<sensor_msgs::msg::LaserScan>::SharedPtr
7      laser_sub_;
8    sensor_msgs::msg::LaserScan::SharedPtr last_scan_;
9  };
```

Nodo personalizado: IsObstacleNear (tick)

```cpp
BT::NodeStatus IsObstacleNearCondition::tick() {
  double threshold;
  getInput("threshold", threshold);

  if (!last_scan_) return BT::NodeStatus::FAILURE;

  float min_distance = std::numeric_limits<float>::max();
  for (const auto& range : last_scan_->ranges) {
    if (std::isfinite(range) && range < min_distance) {
      min_distance = range;
    }
  }

  setOutput("obstacle_distance", static_cast<double>(min_distance));

  return (min_distance < threshold)
    ? BT::NodeStatus::SUCCESS
    : BT::NodeStatus::FAILURE;
}
```

IsObstacleNear: lógica del nodo

- **Constructor**: obtiene el nodo de ROS 2 del blackboard con `blackboard->get("node", node_)`.
- **Suscripción continua**: subscribe a `/scan` en el constructor.
- **Puerto de entrada**: lee `threshold` del blackboard (default 0.5m).
- **Cálculo**: busca la distancia mínima en el array `ranges`.
- **Puerto de salida**: escribe `obstacle_distance` al blackboard con `setOutput()`.
- **Retorno**: SUCCESS si hay obstáculo dentro del umbral, FAILURE en caso contrario.
- Permite que otros nodos (BackUp, Spin) adapten su comportamiento según la distancia del obstáculo.

Nodo personalizado: BackUp (StatefulActionNode) – 1/2

```
1  class BackUpAction : public BT::StatefulActionNode
2  {
3  public:
4    BackUpAction(const std::string& name,
5                 const BT::NodeConfiguration& config);
6    static BT::PortsList providedPorts() {
7      return {
8        BT::InputPort<double>("obstacle_distance"),
9        BT::InputPort<double>("base_distance", 0.3)
10     };
11   }
```

Nodo personalizado: BackUp (StatefulActionNode) – 2/2

```
1  class BackUpAction : public BT::StatefulActionNode
2  {
3    // ...continuación...
4    BT::NodeStatus onStart() override;
5    BT::NodeStatus onRunning() override;
6    void onHalted() override;
7  private:
8    rclcpp::Node::SharedPtr node_; // obtenido del blackboard
9    rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr cmd_vel_pub_;
10   double distance_; // distancia a retroceder calculada
11   rclcpp::Time start_time_;
12   rclcpp::Duration duration_;
13 };
```

Nodo personalizado: BackUp (onStart)

```
1  BT::NodeStatus BackUpAction::onStart() {
2    double obstacle_dist, base_dist;
3    getInput("obstacle_distance", obstacle_dist);
4    getInput("base_distance", base_dist);
5    // Margen de seguridad: +20cm extra
6    distance_ = std::max(base_dist, obstacle_dist + 0.2);
7    start_time_ = node_->now();
8    duration_ = rclcpp::Duration::from_seconds(distance_ / 0.2);
9    return BT::NodeStatus::RUNNING;
10 }
```

Nodo personalizado: BackUp (onRunning)

```
BT::NodeStatus BackUpAction::onRunning() {
  auto elapsed = node_->now() - start_time_;
  if (elapsed < duration_) {
    geometry_msgs::msg::Twist cmd;
    cmd.linear.x = -0.2;
    cmd_vel_pub_->publish(cmd);
    return BT::NodeStatus::RUNNING;
  }
  geometry_msgs::msg::Twist cmd;
  cmd_vel_pub_->publish(cmd);
  return BT::NodeStatus::SUCCESS;
}
void BackUpAction::onHalted() {
  geometry_msgs::msg::Twist cmd;
  cmd_vel_pub_->publish(cmd);
}
```

ActionNode vs StatefulActionNode

ActionNode (simple):

- Solo implementa `tick()`: se ejecuta completamente en una sola llamada
- Retorna SUCCESS, FAILURE o RUNNING inmediatamente
- Útil para acciones síncronas o muy cortas
- Ejemplo: leer un sensor, comprobar una condición simple

StatefulActionNode (con estado):

- Ciclo de vida: `onStart()`, `onRunning()`, `onHalted()`
- Permite acciones de **larga duración** que retornan RUNNING durante múltiples ticks
- `onStart()`: inicialización (se llama una vez)
- `onRunning()`: continúa ejecución (se llama repetidamente mientras está RUNNING)
- `onHalted()`: limpieza si el nodo es interrumpido (preemption)
- Útil para: navegación, movimientos temporales, diálogos, etc.

BackUp: lógica del nodo

- **StatefulActionNode**: implementa ciclo de vida completo para acción de larga duración.
- **Constructor**: obtiene el nodo de ROS 2 del blackboard con `blackboard->get("node", node_)`.
- **Puertos de entrada**: lee `obstacle_distance` y `base_distance`.
- **Cálculo dinámico**: ajusta distancia de retroceso según proximidad del obstáculo.
- **Control temporal**: retrocede a velocidad constante durante tiempo calculado.
- **onHalted**: detiene robot por seguridad si el nodo es interrumpido.
- Patrón típico para acciones de larga duración.

Programa principal: registro de nodos

```cpp
int main(int argc, char** argv) {
  rclcpp::init(argc, argv);
  auto node = std::make_shared<rclcpp::Node>("bumpandgo_bt");
  BT::BehaviorTreeFactory factory;

  // Registrar nodos personalizados
  register_bt_nodes(factory);

  // Leer parametros ROS
  node->declare_parameter("charger_x", 0.0);
  node->declare_parameter("charger_y", 0.0);
  double charger_x = node->get_parameter("charger_x").as_double();
  double charger_y = node->get_parameter("charger_y").as_double();

  // Crear blackboard y poner recursos ANTES de crear el arbol
  auto blackboard = BT::Blackboard::create();
  blackboard->set("node", node);
  blackboard->set("charger_x", charger_x);
  blackboard->set("charger_y", charger_y);
```

Programa principal: carga del árbol y ejecución

```
1  // Obtener path al archivo XML
2  auto pkg = ament_index_cpp::get_package_share_directory(
3    "bt_examples");
4  auto xml_file = pkg + "/config/bumpandgo_tree.xml";
5
6  // Cargar arbol desde XML con la blackboard que contiene los recursos
7  auto tree = factory.createTreeFromFile(xml_file, blackboard);
8
9  // Logger para depuracion
10 BT::StdCoutLogger logger(tree);
11
12 // Bucle de ejecucion
13 rclcpp::Rate rate(10);
14 auto status = BT::NodeStatus::RUNNING;
15 while (rclcpp::ok() &&
16        status == BT::NodeStatus::RUNNING) {
17   rclcpp::spin_some(node);
18   status = tree.tickOnce();
19   rate.sleep();
20 }
```

Programa principal: puntos clave

- **BehaviorTreeFactory**: registro de tipos de nodos personalizados.
- **Blackboard previa**: se crea y se inicializa ANTES de cargar el árbol.
- **Recursos compartidos**: el nodo de ROS se pone en el blackboard con `blackboard->set("node", node)`.
- **Carga desde XML**: `createTreeFromFile(path, blackboard)` parsea el XML, instancia nodos y les pasa la blackboard.
- **Bucle híbrido**:
  - `rclcpp::spin_some()`: procesa callbacks ROS 2 (sensores).
  - `tree.tickOnce()`: ejecuta un tick del árbol.
  - Permite reactividad a cambios sensoriales con control del flujo.

Separación XML vs C++

- **XML**: define la estructura del árbol.
  - Jerarquía de nodos de control.
  - Conexión de puertos.
  - Parámetros configurables.
- **C++**: implementa la lógica de los nodos.
  - Interacción con ROS 2 (topics, actions, services).
  - Algoritmos y cálculos.
  - Gestión de estado.
- Ventajas:
  - Iterar rápidamente en el diseño sin recompilar.
  - Reutilizar nodos en diferentes árboles.
  - Separación de responsabilidades.

HRI: nodos de diálogo con HRIClient

- Para el robot camarero necesitamos capacidades de diálogo:
  - **TTS (Text-to-Speech)**: robot habla al cliente.
  - **STT (Speech-to-Text)**: robot escucha al cliente.
  - **Extracción de información**: analizar pedido con LLM.
- **HRIClient**: clase que abstrae la comunicación con servicios de diálogo.
- Nodos BT para HRI:
  - `SayTextClientAction`: usa HRIClient para TTS.
  - `ListenTextClientAction`: usa HRIClient para STT.
  - `ExtractInfoClientAction`: usa HRIClient para extracción.

SayTextClientAction: estructura

```
1  class SayTextClientAction : public BT::StatefulActionNode
2  {
3  public:
4    SayTextClientAction(const std::string & name,
5                        const BT::NodeConfig & config);
6    static BT::PortsList providedPorts() {
7      return { BT::InputPort<std::string>("text") };
8    }
9    BT::NodeStatus onStart() override;
10   BT::NodeStatus onRunning() override;
11   void onHalted() override;
12 private:
13   std::shared_ptr<HRIClient> hri_client_; // obtenido del blackboard
14   std::chrono::steady_clock::time_point start_time_;
15 };
```

SayTextClientAction: onStart – 1/2

```cpp
BT::NodeStatus SayTextClientAction::onStart() {
  std::string text;
  if (!getInput("text", text)) {
    return BT::NodeStatus::FAILURE;
  }
  text = formatText(text);
  hri_client_->start_speaking(text);
  start_time_ = std::chrono::steady_clock::now();
  return BT::NodeStatus::RUNNING;
}
```

SayTextClientAction: onRunning – 2/2

```
1  BT::NodeStatus SayTextClientAction::onRunning() {
2    auto elapsed = std::chrono::steady_clock::now()
3      - start_time_;
4    if (elapsed > std::chrono::seconds(30)) {
5      return BT::NodeStatus::FAILURE;
6    }
7    if (hri_client_->is_speaking_done()) {
8      return hri_client_->get_speaking_result()
9        ? BT::NodeStatus::SUCCESS
10       : BT::NodeStatus::FAILURE;
11   }
12   return BT::NodeStatus::RUNNING;
13 }
```

SayTextClientAction: lógica

- **formatText()**: expande variables del blackboard.
  - Ejemplo: "Hola {name}"→ "Hola Juan"si `name="Juan"`.
- **Patrón asíncrono**: `onStart()` inicia TTS, `onRunning()` monitoriza.
- **Timeout defensivo**: 30 segundos máximo.
- **HRIClient abstrae complejidad**: el nodo no gestiona servicios ROS directamente.
- Permite integrar diálogo natural en el BT.

ListenTextClientAction: estructura – 1/2

```
1  class ListenTextClientAction : public BT::StatefulActionNode
2  {
3  public:
4    ListenTextClientAction(const std::string & name,
5                           const BT::NodeConfig & config);
6
7    static BT::PortsList providedPorts() {
8      return { BT::OutputPort<std::string>(
9        "recognized_text", "Recognized text") };
10   }
```

ListenTextClientAction: estructura – 2/2

```
1  class ListenTextClientAction : public BT::StatefulActionNode
2  {
3    // ...continuación...
4    BT::NodeStatus onStart() override;
5    BT::NodeStatus onRunning() override;
6    void onHalted() override;
7
8  private:
9    std::shared_ptr<HRIClient> hri_client_; // obtenido del blackboard
10   std::chrono::steady_clock::time_point start_time_;
11 };
```

ListenTextClientAction: onStart – 1/2

```
1 BT::NodeStatus ListenTextClientAction::onStart() {
2   hri_client_->start_listen();
3   start_time_ = std::chrono::steady_clock::now();
4   return BT::NodeStatus::RUNNING;
5 }
```

ListenTextClientAction: onRunning – 2/2

```
BT::NodeStatus ListenTextClientAction::onRunning() {
  auto elapsed = std::chrono::steady_clock::now() - start_time_;
  if (elapsed > std::chrono::seconds(60)) {
    return BT::NodeStatus::FAILURE;
  }
  if (hri_client_->is_listen_done()) {
    std::string recognized_text = hri_client_->get_listened_text();
    if (recognized_text.empty()) {
      return BT::NodeStatus::FAILURE;
    }
    setOutput("recognized_text", recognized_text);
    return BT::NodeStatus::SUCCESS;
  }
  return BT::NodeStatus::RUNNING;
}
```

ListenTextClientAction: lógica

- **Puerto de salida**: escribe texto reconocido al blackboard.
- **Timeout mayor**: 60 segundos (el cliente puede tardar en hablar).
- **Validación**: verifica que el texto no esté vacío.
- Permite capturar pedidos del cliente de forma natural.

ExtractInfoClientAction: estructura – 1/2

```
1  class ExtractInfoClientAction : public BT::StatefulActionNode
2  {
3  public:
4    ExtractInfoClientAction(const std::string & name,
5                            const BT::NodeConfig & config);
6    static BT::PortsList providedPorts() {
7      return {
8        BT::InputPort<std::string>("interest"),
9        BT::InputPort<std::string>("full_text"),
10       BT::OutputPort<std::string>("extracted_info")
11     };
12   }
```

ExtractInfoClientAction: estructura – 2/2

```
1  class ExtractInfoClientAction : public BT::StatefulActionNode
2  {
3    // ...continuación...
4    BT::NodeStatus onStart() override;
5    BT::NodeStatus onRunning() override;
6    void onHalted() override;
7  private:
8    std::shared_ptr<HRIClient> hri_client_; // obtenido del blackboard
9    std::chrono::steady_clock::time_point start_time_;
10 };
```

ExtractInfoClientAction: lógica (onStart) – 1/2

```
1  BT::NodeStatus ExtractInfoClientAction::onStart() {
2    std::string interest, full_text;
3    if (!getInput("interest", interest) ||
4        !getInput("full_text", full_text)) {
5      return BT::NodeStatus::FAILURE;
6    }
7
8    hri_client_->start_extract(interest, full_text);
9    start_time_ = std::chrono::steady_clock::now();
10   return BT::NodeStatus::RUNNING;
11 }
```

ExtractInfoClientAction: lógica (onRunning) – 2/2

```cpp
BT::NodeStatus ExtractInfoClientAction::onRunning() {
  auto elapsed = std::chrono::steady_clock::now() - start_time_;
  if (elapsed > std::chrono::seconds(10)) {
    return BT::NodeStatus::FAILURE;
  }

  if (hri_client_->is_extract_done()) {
    std::string extracted = hri_client_->get_extracted_info();
    setOutput("extracted_info", extracted);
    return BT::NodeStatus::SUCCESS;
  }
  return BT::NodeStatus::RUNNING;
}
```

ExtractInfoClientAction: uso

- **Doble puerto de entrada**: lee `interest` y `full_text`.
- **Puerto de salida**: escribe información extraída.
- Ejemplo:
  - Input: interest="bebida", full_text="Quiero un café y una tostada"
  - Output: extracted_info=çafé"
- Usa LLM para analizar lenguaje natural.
- Permite procesar pedidos complejos.

Integrando todo: misión del robot camarero

1. Esperar cliente: `WaitForPerson` (Condition).
2. Saludar: `SayTextClient("Hola, ¿qué desea?")`.
3. Escuchar pedido: `ListenTextClient` → `recognized_text`.
4. Extraer bebida: `ExtractInfoClient` con interest=bebida, full_text=recognized_text.
5. Extraer comida: `ExtractInfoClient` con interest=comida, full_text=recognized_text.
6. Navegar a cocina: `NavigateTo` a pose cocina.
7. Simular recoger: `SimulateAction` recogiendo pedido.
8. Regresar a cliente: `NavigateTo` a pose cliente.
9. Entregar: `SimulateAction` entregando pedido.
10. Volver a espera: `NavigateTo` a pose home.

Importante: activar capacidades antes de ejecutar

Para que el robot camarero funcione correctamente, es necesario **activar las capacidades** que utiliza:

1. **Capacidades de navegación** (Nav2):
   - Lanzar stack de navegación Nav2
   - Proporciona: planificación de trayectorias, localización, control

2. **Capacidades de interacción** (HRI):
   - Lanzar nodo de interacción humano-robot
   - Comando: `ros2 launch simple_hri simple_hri_launch.py` (u otro launcher del paquete)
   - Proporciona: TTS (text-to-speech), STT (speech-to-text), extracción con LLM

Sin estas capacidades activas, los nodos del BT que las usan fallarán.

Preguntas