## Diseño lógico

En estas secciones se describe la estructura y funcionamiento del software incluído en los microcontroladores del robot.

### Estructura general

Todos los componentes lógicos del robot se modelan como módulos independientes. Existe dos tipos de módulos, los módulos de control y los módulos de dispositivo.

Un módulo de control actúa como un concentrador de mensajes, distribuyendo los mensajes del protocolo entre los distintos módulos de dispositivo. Los módulos implementan la relación cliente/servidor entre los componentes del robot. El módulo de control actúa de punto de acceso WiFi y servidor de cara a las comunicaciones TCP/IP. Cada módulo de dispositivo actúa como estación WiFi y cliente de cara a las comunicaciones TCP/IP.

Cada módulo, incluso el de control, puede disponer de cualquier número de dispositivos físicos, que denominaremos elementos, directamente controlables mediante un protocolo sencillo.

### Protocolo

El protocolo entre módulos permite comunicar mediante paso de mensajes todos los componentes, sin necesidad de tener un mecanismo de direccionamiento sofisticado.

Es un protocolo sin estado que se compone de mensajes textuales completamente homogéneos con el siguiente formato:

[Orden] [id\_ModuloOrig] [id\_ModuloDestino] ([id\_Elemento]+ ([Datos]))

La orden puede ser una de las siguientes letras:

|  |  |
| --- | --- |
| W | Escribir un valor determinado en el elemento id\_Elemento (pueden ser varios) del módulo id\_ModuloDestino |
| R | Leer el valor asociado al elemento id\_Elemento (pueden ser varios) conectado al módulo id\_ModuloDestino. |
| N | Notifica el resultado de una lectura al dispositivo que la pidió (id\_ModuloDestino). |
| L | Lista de los elementos conectados a un módulo. Esta orden es usada solamente para depuración. |

El protocolo permite identificar al módulo originario sin importar que otro módulo actúe de intermediario. Además, todos los mensajes son idempotentes, lo que simplifica notablemente la gestión de errores.

Algunos ejemplos de mensajes se muestran a continuación:

|  |  |
| --- | --- |
| W pc mod1 motor1 138 | Escribe en el elemento motor1 del módulo mod1 el valor 138. Lo pide el módulo pc. |
| R pc mod2 temp | Lee el elemento temp del módulo mod2. Lo pide el módulo pc. |
| N mod2 pc temp 23 | Notificación del valor 23 del elemento temp del módulo mod2. Lo notifica al módulo pc, presumiblemente porque antes hubo una lectura desde ese origen. |
| L pc mod1 | Lista los elementos disponibles en el módulo mod1. Lo pide el módulo pc. |

En un mismo mensaje se pueden indicar varios elementos del mismo módulo, separados por comas.

#### Modelo cliente servidor

Todo el proyecto utiliza comunicaciones TCP. Se asume que el módulo de control es un servidor TCP concurrente y los módulos de dispositivo son clientes TCP. Sin embargo, desde el punto de vista de la interacción, el sistema sigue un modelo peer to peer, en el que cualquiera puede tomar la iniciativa de la comunicación. Así mismo, la interacción se produce siempre mediante paso de mensajes para evitar problemas de fallo parcial. No existe garantía de orden en la entrega de mensajes, aunque esto se puede conseguir con un protocolo de mayor nivel si fuera necesario.

La implementación de cliente y servidor se realiza en los archivos client.hh y server.hh y sus respectivos client.cpp y server.cpp. Ambos se implementan de forma similar, con una máquina de estados que intenta conexiones persistentes.

Por ejemplo, el cliente tiene la siguiente estructura:

template <class Protocol>

class ROMEOClient {

public:

ROMEOClient() : \_state(State::Disconnected) { }

void run();

private:

// ...

enum class State {

Disconnected,

Waiting,

Associated,

Connected,

};

WiFiClient \_client;

State \_state;

LEDControl \_led;

};

El método run se ejecuta repetidamente y es el responsable de realizar las transiciones de estado cuando se cumplen las condiciones. Normalmente el módulo pasará de forma secuencial entre los estados

1. Disconnected. El módulo no está conectado ni asociado a ningún punto de acceso.
2. Waiting. El módulo está intentando asociarse a la red WiFi del módulo de control.
3. Associated. El módulo está asociado a la red WiFi e intenta conectarse al módulo de control usando TCP.
4. Connected. El módulo está conectado usando TCP con el módulo de control.

El servidor implementa la máquina de estados complementaria, con una estructura similar:

template <class Protocol>

class ROMEOServer {

public:

ROMEOServer() : \_state(State::MissingAP), \_server(80) { pinMode(2, OUTPUT); }

void run();

// ...

private:

enum class State {

MissingAP,

SoftAP,

Listening,

};

WiFiServer \_server;

WiFiClient \_clients[4];

State \_state;

LEDControl \_led;

};

Al igual que en el caso del cliente, el método run es el responsable de realizar las transiciones de estado. Normalmente seguirá la siguiente secuencia:

1. MissingAP. El módulo de control activa el punto de acceso. Aún no hay constancia de que haya tenido éxito.
2. SoftAP. El punto de acceso está disponible. Ahora activa el servidor TCP.
3. Listening. El servidor TCP está escuchando. Si hay algún nuevo cliente establece la conexión y se mantiene en el estado Listening.

#### Rutado

Los módulos de dispositivo son nodos finales. Solo entregan los mensajes que van destinados a ellos. Sin embargo, los módulos de control actúan como concentradores (hubs) de mensajes, retransmitiendo todos los mensajes que no van destinados a ellos a todos los módulos de dispositivo conectados. Esto permite que el rutado pueda implementarse sin mantener ningún tipo de estado, lo que dota al sistema de una aceptable tolerancia a fallos.

#### Paso de mensajes

Los mensajes de los que consta el protocolo constituyen fundamentalmente transferencias de estado completas. Las escrituras cambian directamente el estado de un elemento por el valor que incorpora el mensaje. Es decir, se trata de un protocolo idempotente.

El transporte de los mensajes se realiza mediante TCP, pero eso no permite concluir que estamos en un entorno ausente de fallos. Las pérdidas de la conexión WiFi o de la conexión TCP pueden ocurrir en cualquier momento. El sistema está preparado para reconectar inmediatamente, pero pueden producirse pérdidas y duplicados de mensajes.

La idempotencia garantiza que no es necesario filtrar duplicados, pero las pérdidas de mensajes no están contempladas en el protocolo. En caso de ser necesario deberá implementarse en un protocolo de mayor nivel.

### Controlador

Para ilustrar la implementación veremos con cierto detalle cómo se implementa el módulo controlador. Puede resumirse en las siguientes líneas:

ROMEODevice device("control", "M12M32");

struct ControlProto {

static void intro(WiFiClient& client, WiFiServer& server);

static void run(WiFiClient& client, WiFiServer& server);

};

using ROMEOModule = ROMEOServer<ControlProto>;

ROMEOModule module;

La primera línea identifica al módulo de control como un posible destino de mensajes, con el nombre control, y con los elementos asociados que se muestran en el argumento M12M32. La cadena de configuración determina que dispone de un motor conectado entre los pines 1 y 2 y otro motor conectado entre los pines 3 y 2.

A continuación, se define una clase que implementa el protocolo de paso de mensajes y se instancia la plantilla ROMEOServer empleando este protocolo. La plantilla ROMEOServer implementa el rol de servidor en la comunicación TCP/IP. Por un lado, se configura el módulo como punto de acceso WiFi y por otro atiende conexiones concurrentes de todos los clientes (ver plantilla ROMEOClient).

El protocolo ControlProto solo tiene dos métodos.

* El método intro se invoca una sola vez, en el momento del establecimiento de la conexión. Está pensado para implementar algún tipo de saludo, que impida un posible interbloqueo de TCP en presencia de fallos. Actualmente el saludo puede ser cualquier línea de texto enviada por el cliente, incluso una línea vacía.
* El método run se invoca cada vez que hay datos disponibles. Actualmente lo único que hace es replicar el mensaje en todas las conexiones abiertas y redirigirlo además al módulo de dispositivo (ROMEODevice).

Un módulo ROMEODevice es simplemente un posible destino de mensajes. Aglutina un conjunto de elementos de distinto tipo, derivados de la clase común Element. Un elemento es capaz de interpretar los mensajes destinados a él (métodos read, write y notify). Como ejemplos, en este proyecto se proporciona un actuador (MotorElement) y un sensor (DistanceSensorElement). Se remite al lector interesado en los detalles de implementación al archivo device.hh[[1]](#endnote-1) del repositorio GitHub.

### Dispositivos

El módulo de dispositivo es muy similar al módulo de control, salvo que implementa el rol de cliente en lugar de servidor y el protocolo no replica los mensajes, sino que los consume directamente:

ROMEODevice device("com", "D12D78");

struct ClientProto {

static void intro(WiFiClient& client);

static void run(WiFiClient& client);

};

using ROMEOModule = ROMEOClient<ClientProto>;

ROMEOModule module;

En este caso el módulo dispositivo es un posible destino con nombre com (módulo de comunicaciones) y configurado con un sensor de distancia entre los pines 1 y 2 y otro sensor de distancia entre los pines 7 y 8.

El protocolo de cliente es análogo al del control, salvo por:

* El saludo (intro) es enviado en lugar de recibido.
* El procesamiento de mensajes simplemente redirige el mensaje al dispositivo (ROMEODevice).

La implementación de los módulos está desacoplada en las diferentes funciones, de manera que sea sencillo construir módulos más complejos.

### Utilidades

Para la comunicación con el usuario se emplea un pequeño módulo de manejo de leds que permite generar fácilmente parpadeos de distintos periodos.

class LEDControl {

public:

LEDControl(): \_count(0) { pinMode(2, OUTPUT); }

static void on() { digitalWrite(2, LOW); }

static void off() { digitalWrite(2, HIGH); }

static void toggle() { digitalWrite(2, !digitalRead(2)); }

void blink(short n) { if (++\_count % n == 0 ) toggle(); }

private:

short \_count;

};

1. <https://github.com/SaulOrtegaVaz/ROMEO/blob/master/src/SimpleRemote/src/device.hh> [↑](#endnote-ref-1)