elektron ámbar

Original Article

Revista elektron, Vol. 6, No. 1, pp. 8-19 (2022)
https://doi.org/10.37537/rev.elektron.6.1.143.2022

Infraestructura para el desarrollo de laboratorios remotos

Remote Laboratories Development Framework

Marco Miretti†[1], Emanuel Bernardi†[2]

†*AC&ES - Research Group, Universidad Tecnológica Nacional Facultad Regional San Francisco*
San Francisco, Córdoba, Argentina
[1]mmiretti@sanfrancisco.utn.edu.ar
[2]ebernardi@sanfrancisco.utn.edu.ar

Recibido: 27/02/22; Aceptado: 14/05/22

***Resumen*—El presente trabajo se fundamenta en la enseñanza a través de experiencias remotas. El mismo plantea el desarrollo de una infraestructura de aprendizaje a distancia mediante la construcción de laboratorios remotos de forma versátil, escalable y asequible, al apoyarse en la democratización de la tecnología.**

Si bien la propuesta es aplicable a un sinnúmero de ramas de la ciencia, con el fin de proveer un prototipo funcional, éste se implementó para la enseñanza de sistemas de control. En consecuencia, éste consiste en la construcción de un sistema dinámico, típico del área. Además, cuenta con una interfaz de programación de aplicación (API) que permite a los estudiantes desarrollar sus propios algoritmos de control, utilizando lenguajes de alto nivel, como son *Python* y *GNU Octave*, prescindiendo de competencias afines a los sistemas embebidos y a las redes de comunicación.

Palabras clave: educación; experimentación remota; sistemas de control.

***Abstract*—The present work is based on teaching through remote experiences. It proposes the development of a distance learning infrastructure by the construction of remote laboratories in a versatile, scalable and affordable way, by relying on the technology democratization.**

Although the proposal is applicable to countless areas of science, in order to provide a functional prototype, it was implemented for teaching control systems. Consequently, this consists of the construction of a dynamic system, typical of the area. In addition, it has an application programming interface (API) that allows students to develop their own control algorithms, using high-level languages, such as *Python* and *GNU Octave*, regardless of related skills to embedded systems and communication networks.

Keywords: education; remote experimentation; control systems.

I. Introducción

La experimentación como herramienta de aprendizaje, es en sí un método sumamente enriquecedor, que al ser complementada con conceptos teóricos posibilita la generación de resultados sobresalientes. Este aspecto es particularmente relevante en áreas de formación ingenieril, donde realizar ensayos prácticos constituye una parte íntegra del estudio [1]. Entonces, a la hora de percibir un fenómeno, u observar el comportamiento de un sistema, la práctica resulta mandatoria. Adicionalmente, el acceso remoto al material educativo ha demostrado ser de gran utilidad durante el brote de COVID-19.

En base a los fundamentos mencionados, a través de este trabajo se plantea el desarrollo de una infraestructura de laboratorios remotos para la enseñanza en la ingeniería, con la aplicación particular a los sistemas de control. A modo de justificar el desarrollo, se construyó un sistema dinámico prototipo que permite el ensayo y evaluación de las técnicas de control instruidas (tales como PID[1], MPC[2], entre otras).

Así, el propósito de esta metodología de enseñanza es permitir que el estudiante se instruya de forma autónoma y descentralizada, posibilitando el acceso a los experimentos de forma remota e ininterrumpida. Particularmente, con la implementación de este sistema el estudiante tiene a su alcance la capacidad de aplicar múltiples técnicas de control [2]–[4], siendo prescindible la experiencia en la programación de bajo nivel, i.e. programación de sistemas embebidos, configuración de las comunicaciones, etc. Para ello, el desarrollo contempla el diseño de librerías intuitivas y simples de utilizar, implementadas en los lenguajes GNU Octave y Python, de modo que para su utilización solo serán necesarios conocimientos mínimos de programación. Se eligieron estos lenguajes debido a que los mismos poseen extensos paquetes dedicados al control de procesos. Esto es, mientras que GNU Octave resulta ampliamente popular entre los investigadores de las áreas afines, Python posee una gran tracción por parte de la comunidad desarrolladora de software libre. Además, para el caso de GNU Octave existen proyectos para llevar la accesibilidad un paso adelante, como por ejemplo el desarrollo de una interfaz gráfica de usuario GUI[3] para la enseñanza de control de procesos [5].

En la Figura 1 se plasma la infraestructura básica planteada, como así también su alcance. La misma demuestra como los estudiantes acceden a experimentos de forma remota a través de internet, utilizando como intermediario un servidor remoto, en este caso implementado sobre una *Raspberry Pi* (aunque puede funcionar en cualquier computador que ofrezca una interfaz WiFi y un sistema operativo basado en GNU Linux). Los experimentos remotos se comunican con dicho servidor mediante plataformas embebidas

[1]Del inglés, Proportional-Integral-Derivative
[2]Del inglés, Model Predictive Control.
[3]Del inglés, Graphical User Interface

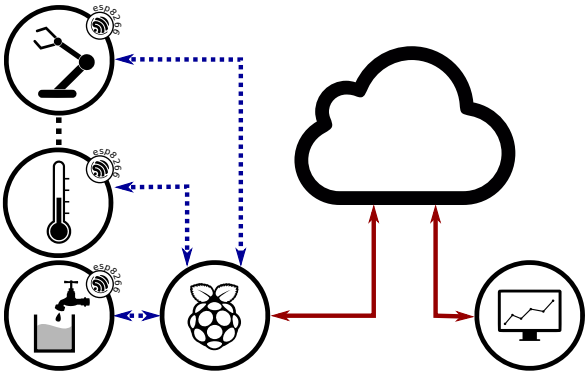

Fig. 1. Mapa del sistema completo

ESP8266, permitiendo el comando de actuadores y la lectura de sensores.

De esta forma sus usuarios, los estudiantes, tienen acceso continuo a la experimentación, sobre fenómenos físicos auténticos. Permitiéndoles aplicar sus conocimientos sobre infinidad de sistemas.

II. Estado del arte

Los laboratorios remotos no son nuevos, desde la invención de internet en los 70' se han propuesto y desarrollado incontables soluciones con una gran variedad de enfoques. Así también trabajos como [6] y [7] han relevado el estado del arte de dichas tecnologías, además de analizar sus puntos comunes, y diferencias más notables. Dicho estudio sirve como un punto de referencia para este análisis.

Es de destacar, que los laboratorios encontrados con mayor frecuencia se corresponden a competencias relevantes a la Ingeniería Electrónica (microelectrónica, robótica, control, sistemas embebidos) [8]. La arquitectura hallada en éstos trabajos se basa en el paradigma cliente-servidor, con un laboratorio actuando como servidor en el extremo remoto, y una aplicación cliente en el extremo del usuario.

Dentro de dicho paradigma existen gran variedad de arquitecturas, incluyendo una basada en servicios web *REST*[4], obteniendo buenos resultados [9]. Otra, como la que propone [10] se encuentra orientada a servicios, favoreciendo la abstracción y, por lo tanto, la modularización de los mismos. En tanto, [11] plantea la compatibilidad de los laboratorios remotos con los *dispositivos inteligentes*, con el fin obtener un mayor alcance.

A. Construcción y topologías

Según analiza [12], los experimentos a realizar en laboratorios remotos se dividen en dos grupos:

- Experimentos "únicos en su clase" o difíciles de replicar debido a su costo y complejidad.
- Experimentos de bajo costo, fácilmente replicables.

Luego, [13] caracteriza los laboratorios remotos dividiendo su topología en dos tipos principales:

- Aplicaciones web.
- Aplicaciones dedicadas de control remoto.

Las aplicaciones web, suelen ser estándares y requerir menos dependencias del lado del cliente. Una aplicación dedicada, si bien a veces provee una mejor interfaz de usuario, viene aparejada de dependencias de software e incompatibilidades.

Del lado del cliente, propone dos enfoques:

- Aplicaciones intrusivas, con permisos de acceso. (accesos por *SSH*[5])
- Aplicaciones no-intrusivas, tales como accesos a través del navegador.

Las primeras permiten una interacción mas fluida y mejor experiencia de usuario, teniendo en muchos casos acceso total a la información de sensores y actuadores, aunque acompañados de riesgos de seguridad. Las aplicaciones no-intrusivas solucionan muchos de estos riesgos, con la desventaja de ser menos interactivas.

Por último, analizando la comunicación servidor-experimento se hallan dos soluciones:

- Software propietario, tal como LabView o MATLAB/Simulink.
- Software libre, comúnmente de propósito general como *C*, *C++* o *Python*.

B. Ejemplos de aplicación

El campo de aplicación de los laboratorios remotos es muy extenso, y tal como se mencionó previamente, su uso es frecuente en las competencias afines a la Ingeniería Electrónica. De esta forma, existen muchos tipos de laboratorios:

- Para el área de la microelectrónica y el diseño de microcontroladores, basados en FPGAs[6] [14].
- Para experiencias afines a la física y óptica [15].
- En el área de la electrónica de potencia, bancos de prueba remotos en donde el usuario puede elegir la estructura de control y obtener resultados gráficos en tiempo real [16].
- En el campo de los sistemas embebidos, produciendo laboratorios sencillos y fáciles de replicar como por ejemplo el *"lab-in-a-box"* (laboratorio en una caja) presentado por [17].
- En la enseñanza de los sistemas de control, cuyos conceptos teóricos requieren de la práctica para que el estudiante logre asimilarlos [6], existe una gran variedad de soluciones y estudios dedicados a los mismos [18]. La gran mayoría de estos experimentos remotos solo permiten al estudiante elegir, a partir un grupo preestablecido, una estrategia de control y regular sus parámetros [19], [20], sin embargo hay unos pocos enfoques como el ACT[7] que no solo permiten esto, sino también dan la posibilidad al alumno de diseñar sus propios algoritmos y probarlos en una gran variedad de sistemas dinámicos (levitador magnético, sistema de tanques de agua, motores y simulador de helicóptero) [21].

III. Propuesta

Para comprender este desarrollo es fundamental entender que el mismo no busca crear un experimento específico,

[4]Del inglés, *Representational State Transfer*.

[5]Del inglés, *Secure Shell*.

[6]Del inglés, Field-Programmable Gate Array

[7]Del inglés, Automatic Control Telelab

del área de control o explicar como dicho experimento es utilizado por los docentes y estudiantes para enriquecer el aprendizaje. El proyecto se desarrolló para presentar una solución a preguntas de mayor envergadura: ¿Cómo puede cualquier desarrollador[8] construir una plataforma de experimentación remota? ¿Qué guía y cuales herramientas debe tener a su alcance?

El resultado, como el título indica es una **infraestructura** para la creación de laboratorios remotos.

Con el fin de refinar dicha infraestructura, se construyó un experimento remoto de ejemplo, que sirvió como base para detectar las necesidades de un laboratorio remoto, ayudando a formar una metodología de desarrollo replicable. Este experimento sirve también de prueba de concepto, poniendo en práctica las bondades de la infraestructura propuesta.

IV. Descripción de los componentes

En la Figura 2 se aprecian los diferentes componentes involucrados. Se hace énfasis en el vasto contenido de *software* y *firmware* en los mismos, uno de los pilares más importantes en este trabajo.

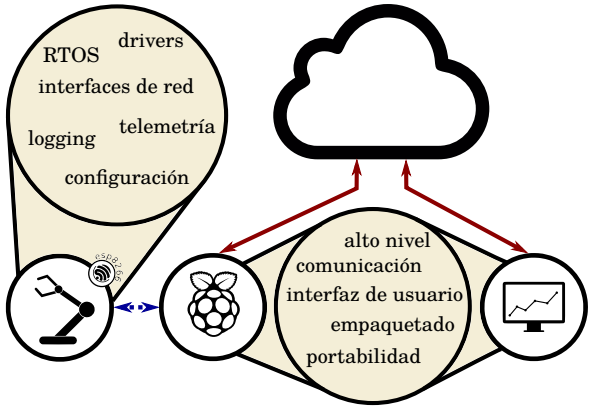

Fig. 2. Mapa de componentes.

A la izquierda está representado el experimento, que a través de una conexión local se comunica con el servidor intermedio. Este último se comunica mediante una red de área amplia (internet) con el usuario final, representado como la burbuja de la derecha.

El laboratorio remoto no es únicamente el experimento, sino también el servidor intermedio, la conexión con el usuario final y el software que se ejecuta en cada uno de los dispositivos.

A. Experimento

Detalladamente, el experimento consiste en un sistema embebido basado en la plataforma *ESP8266*, el cual es el encargado de comandar sensores y actuadores. Luego, la naturaleza del sistema dinámico variará según con que se quiera experimentar, algunos ejemplos son:

- Resistencias y capacitores formado configuraciones de filtros, con entradas comandadas por señales PWM y medidos con puertos analógicos, para prácticas en el área de Señales y Sistemas.
- Otro sistema embebido de interés, para una experiencia en el área de Técnicas Digitales.
- Un sistema termodinámico, como una resistencia calefactora y un sensor de temperatura LM-35, utilizando las interfaces PWM y 1-Wire, para la experimentación en Sistemas de Control.
- Un péndulo aeropropulsado, conectado mediante la interfaz PWM para su comando, y midiendo su ángulo a través de un encoder, utilizando la interfaz GPIO con interrupciones.

Tanto para estos ejemplos, como para muchos otros posibles, es necesario que la infraestructura cumpla con ciertas condiciones, respecto al sistema embebido y su firmware. Esto es:

- Debe tener la capacidad de interactuar sus interfaces para comandar las entradas y leer las salidas de los elementos conectados.
- Debe posibilitar la exposición de la información recibida desde los dispositivos conectados, para que la misma llegue al usuario final. Como así también utilizar la información indicada por el usuario para comandar los actuadores indicados.
- La comunicación con el cliente debe ser lo suficientemente ágil para satisfacer las necesidades del sistema dinámico.
- Debe soportar un sistema operativo de tiempo real, con capacidad de crear y ejecutar tareas.
- Debe tener una interfaz clara de comando y configuración.
- Debe generar *logs*, a ser consultados por el usuario, para entender su funcionamiento, como también por el programador del experimento en la etapa de desarrollo.

Para cumplir con estos requisitos se programó un *Firmware* basado en *FreeRTOS*, permitiendo la creación de múltiples tareas. Para ello, posee una librería de *drivers* para comunicar el dispositivo embebido con sus sensores y actuadores a través de interfaces GPIO, SPI, I2C, PWM, 1-Wire y con libertad suficiente para otras específicas como; interfaz de encoder angular incremental, interfaz de comandos y PWM para controladores de motores sin escobillas.

Además, tiene la capacidad de comunicación HTTP y Websockets para configuraciones y telemetría respectivamente, pudiendo así ser comandado y leído por el usuario final con la confianza y velocidad necesarias.

Finalmente, el embebido tiene la capacidad de *logging* a través de la interfaz UART.

B. Cliente-servidor intermedio

En esta infraestructura es necesario un dispositivo intermedio que se encargue de recolectar información de los experimentos presentes en su red y transmitirla a los usuarios correspondientes. Para este propósito se utilizó una plataforma Raspberry PI 4B, aunque el software necesario para que cumpla su propósito puede ser instalado en cualquier computador con un sistema operativo basado en linux y una interfaz de red WiFi.

Sus requisitos mínimos para cualquier laboratorio remoto son:

- Capacidad de comunicarse con el dispositivo embebido.

[8]Para estos fines se considera desarrollador a un docente, estudiante, programador o grupo multidisciplinario, con nociones básicas de programación.

- Memoria y capacidad de procesamiento suficiente para almacenar, preprocesar y retransmitir la información.
- Posibilidad de comunicarse con el usuario final.

El software que se creó a partir de estos requerimientos permite que un programa escrito en los lenguajes *Python* u *Octave*, provisto por el usuario final, se ejecute en el servidor intermedio y se comunique con el embebido a través de *HTTP* y *Websockets*. Los paquetes de software multiplataforma que se generaron con este objetivo son *Nyquist* y *Octave Websockets*, publicados en el gestor de paquetes *pip* para *Python* y en el índice oficial de paquetes de *Octave*, respectivamente.

Además, para este computador se desarrolló un software que permite abrir una conexión *SSH* o de *Jupyter Notebook* con el usuario final, permitiendo ser invocado por este último a través de una plataforma de mensajería como *Discord*. De esta forma, el usuario final podrá comandar el sistema embebido como si fuese el servidor intermedio.

V. Elección de *hardware*

Debido a que en cada experimento se requiere implementar un sistema de bajo costo, con recursos computacionales entre bajos y moderados, y conexión WLAN, se optó por utilizar un sistema embebido basado en el microprocesador ESP8266. Dicha plataforma cuenta con soporte físico *Wi-Fi* y *stack TCP/IP*, a un costo mínimo.

Así, desde el punto de vista de los elementos de hardware, éstos se componen en una gran variedad de *breakouts*. Particularmente, se optó por el dispositivo *WeMos-D1 mini*, ya que su tamaño resulta ventajoso al momento de diseñar experimentos modulares, y la cantidad de pines disponibles se considera suficiente para estas aplicaciones.

VI. Firmware

Por otro lado, en lo que concierne al firmware existen numerosos y amplios *kits* de desarrollo, que proporcionan drivers y librerías que resuelven los aspectos de más bajo nivel.

A. Entorno de desarrollo y RTOS

Con entorno de desarrollo nos referimos a aquellos *drivers* y bibliotecas de software que resuelven las secciones de bajo nivel. El *framework* elegido es el *esp-open-rtos*, un entorno basado en el sistema operativo de tiempo real *FreeRTOS*, escrito en lenguaje *C* y acompañado por complementos extras, los que contemplan desde el manejo de la interfaz *PWM* hasta servidores *HTTP*.

Es de destacar la existencia de numerosos *frameworks* para este microprocesador. Siendo otras alternativas disponibles:

- *ESP866_RTOS_SDK*: también basado en *FreeRTOS*, con soporte oficial de *espressif* y documentación sobresaliente. Se descartó por parecer poco versatil, en caso de incorporar funcionalidades no contempladas. Un ejemplo claro de esto es el requisito de comunicación por *Websockets*.
- *esp-idf*: soportado oficialmente por *espressif*, extremadamente amplio y versátil. Documentado minuciosamente. Se descartó ya que fue deprecado para el chip *ESP8266* en etapas tempranas de su desarrollo, soportando únicamente microprocesadores de la gama *ESP32*.

Por último, para favorecer la replicabilidad, y facilitar la creación de experimentos por parte de nuevos desarrolladores, se generaron imágenes de *Docker*, una de ellas con las herramientas de compilación necesarias y las bibliotecas de software ya incluidas, otra con la capacidad de grabar mediante UART binarios ya compilados en un microprocesador *ESP8266*. Todo esto, junto con las instrucciones de uso se encuentran en la documentación del proyecto, almacenado en GitLab [22].

B. Redes y comunicación

En esta sección se describe fundamentalmente la comunicación entre el experimento y la computadora de alto nivel que actúa como servidor remoto. En tanto, la conexión entre el usuario y dicho servidor será omitida ya que no resulta relevante en este contexto.

Con el objetivo de comunicar tanto comandos, configuraciones, como señales de control y medición, entre el experimento y el servidor se utilizaron dos protocolos.

- HTTP: para comunicar todo tipo de señales de configuración, cuya latencia no es necesariamente un factor determinante, pero si lo son los códigos de error reportados luego de aplicar las mismas.
- Websockets: para enviar señales de comando a los actuadores y recibir información de los sensores. Este tipo de señales requieren una latencia reducida.

Así, para la gestión de configuraciones se optó por un patrón de mensajería *request-response* y se creó una API-REST[9] [23], con el objetivo de mejorar la escalabilidad, simplicidad, versatilidad y portabilidad. Además, se generó una especificación OpenAPI [24].

Por el contrario, en el caso de las señales transmitidas a través de *Websockets*, se eligió un patrón del estilo *publish-subscribe*, donde cada dispositivo envía paquetes de telemetría encapsulados con formato *JSON*. La latencia observada entre fuente y destino para un paquete con la información de un par de sensores o actuadores es de 10 ms aproximadamente, lo que se considera satisfactorio para el común de los experimentos contemplados por la propuesta. El patrón de comunicación y su secuencia de inicialización se observan en la Figura 3.

Tal como se aprecia en el diagrama de secuencia de la Figura 3, la comunicación es iniciada por el cliente (Nodo Central), abriendo un *Websocket*. Al hacer esto, el lazo principal del firmware presente en el módulo ESP crea un nuevo hilo de ejecución para la rutina seleccionada según la dirección del *socket* abierto. Así, dicha rutina o *task*, queda ligada a las comunicaciones correspondientes.

La secuencia que se muestra se corresponde con el caso típico de un sistema de control comandado por el sistema embebido. En este, el algoritmo de control se comunica con el sistema para asignar valores a sus actuadores y recibe información de los sensores, esto último solo en caso de tratarse de un control a lazo cerrado.

Es de destacar que con este patrón la comunicación no se bloquea la lectura de sensores o el comando de

[9] Del inglés, Representational State Transfer

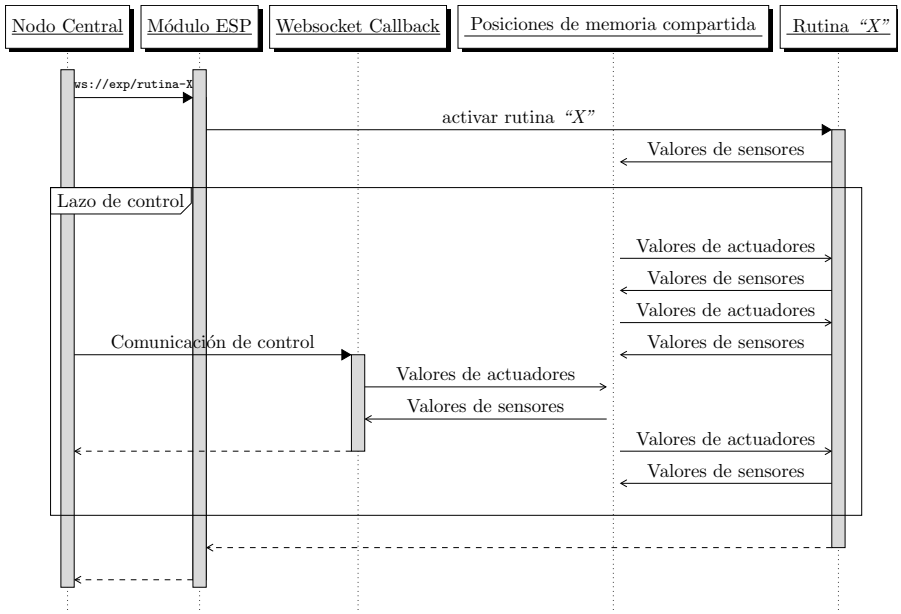

Fig. 3. Diagrama de secuencia para la comunicación del experimento.

actuadores, y viceversa. Esto se debe al uso de una memoria compartida. La tarea denominada "X" se encarga de leer los sensores y almacenar la información adquirida de éstos en la memoria compartida. De la misma forma, tomará los valores correspondientes a los actuadores desde dicha fuente de almacenamiento y accionará los elementos correspondientes a través de sus respectivos drivers (ver sección VI-H) es importante remarcar que es posible que existan varias tareas similares a "X", que utilizarán la información de esta memoria compartida para diferentes propósitos, como controlar los valores máximos comandados, detectar cuando el experimento está fuera de rango, proporcionar un *logging* constante (ver sección VI-G y sección VI-F).

Mientras la tarea "X" se está ejecutando, las llamadas de *Websockets* entre el nodo central y el dispositivo embebido se ejecutan en pseudo-paralelo[10]

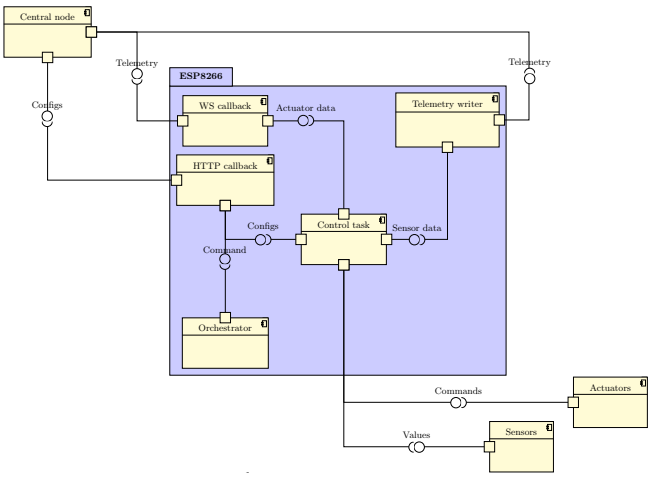

Fig. 4. Diagrama UML de componentes.

[10]No pueden ejecutarse realmente en paralelo debido a que el dispositivo embebido tiene un solo núcleo, sin embargo el *RTOS* orquesta los tiempos de ejecución de cada tarea, para que desde el punto de vista práctico se pueda asumir que éstas se ejecutan de forma paralela.

En la Figura 4 se aprecian los componentes del sistema embebido y sus interfaces. Como se explicó, el embebido se comunica con el nodo central de dos formas; mediante preguntas y respuestas para configuraciones y comandos *one-shot*, por otro lado, de forma publicar-subscribir para transmitir información a alta velocidad para el control en línea. En tanto, con los sensores y actuadores, se comunica mediante las interfaces disponibles (GPIO, SPI, I2C), según cada especificación de los dispositivos que se conecten.

Internamente el dispositivo embebido comunica sus tareas de tres formas, ya sea mediante una memoria compartida, a través de *task-messages* (provistos por el framework *FreeRTOS*) o bien utilizando rutinas de interrupción.

C. Estructura de la telemetría

En casi cualquier dispositivo, es importante que la telemetría tenga las siguientes características:

- Parseabilidad.
- Escalabilidad.
- Observabilidad.

La parseabilidad se refiere a la capacidad de la telemetría para ser procesada y almacenada en variables dentro de la memoria de programa. Por otro lado, una comunicación escalable es versátil si es posible agregar, modificar o eliminar variables en el futuro, sin romper la compatibilidad del protocolo. Finalmente, la observabilidad consiste en la capacidad de "espiar" cualquier canal de comunicación y leer lo que se está transmitiendo. En caso de nuevos experimentos este último punto resulta muy útil durante la etapa de desarrollo y reparación de "bugs".

Con el fin de cumplir estas condiciones, se eligió el formato de intercambio de datos JSON[11]. A continuación se muestra la composición de la misma, proveniente del experimento:

```
{"encoder": {"angle_deg": 30.255, "ts_ms": 1633306309135,
     "code": 0}}
```

[11]Del inglés, JavaScript Object Notation

De la misma forma, se muestra la telemetría emitida desde el nodo central:

```
{"propeller": {"pwm_duty": 10.3, "ts_ms": 1633306309147,
    "code": 0}}
```

En este ejemplo, el dispositivo embebido decodifica el JSON extrayendo los valores necesarios para comandar el actuador (un propulsor cuyo driver recibe instrucciones según el duty de PWM). Luego utiliza los datos leídos en el sensor para generar la línea de "encoder" que es emitida por *Websockets*.

Con el objetivo de parsear los JSONs se generó una librería a partir de un detokenizador primitivo llamado jsmn [25]. El mismo es extremadamente limitado, pero posee un gran potencial; ya que no tiene dependencias, es simple, portable, y no requiere colocación de memoria dinámica. Extendiendo esta utilidad se logró una robusta y veloz librería, apta para el caso de uso planteado.

D. Interfaz RESTful

En el experimento, las transiciones de estado se disparan a través de comandos HTTP, utilizando el estilo de arquitectura REST. De esta forma, cada uno de los estados o elementos de la configuración presentes en el sistema embebido están emparejados a un recurso HTTP, cuyas transiciones se encuentran bien definidas, produciendo así una API.

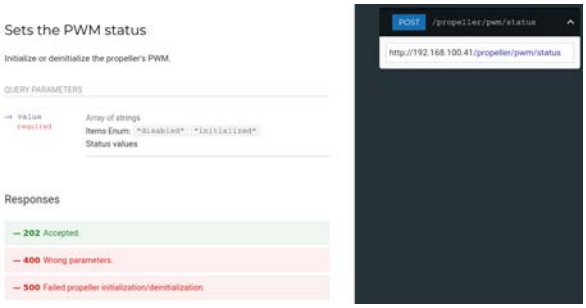

Fig. 5. Documentación OpenAPI del verbo POST para el recurso `propeller/pwm/status`.

En la Figura 5 se aprecia la especificación OpenAPI en forma de documentación HTML para los verbos POST de un recurso.

A continuación, se observa una interacción entre un usuario y el experimento, en donde éste solicita que el propulsor se inicialice y el experimento responde que su requisito fue aceptado.

```
> curl "http://192.168.100.41/propeller/pwm/status?verb=
    POST&value=initialized"
202 Accepted
```

De forma similar, el usuario puede no solo modificar el estado de los recursos sino también consultarlo, típicamente con el recurso `GET`. Esto se muestra a continuación.

```
> curl "http://192.168.100.41/propeller/pwm/status?verb=
    GET"
initialized
```

En dicho caso, el experimento responde con "initialized" para indicar que el recurso ya se encuentra inicializado. Por supuesto, dicho estado era conocido, ya que el último comando enviado había confirmado la correcta inicialización en su respuesta.

De esta forma, el proyecto aprovecha las bondades del protocolo HTTP. El cual permite realizar pedidos, ejecutar porciones de código en el servidor embebido y dar una respuesta adecuada en el código de error, por lo cual se lo consideró una opción sumamente robusta y apropiada para realizar configuraciones o enviar comandos singulares, tal como la inicialización de un actuador. Mientras que para la información de control, que no requiere una respuesta o código de error, la información se envía por Websockets, una alternativa mucho mas veloz, aunque menos robusta en cuanto a confirmaciones.

E. Performance de las comunicaciones

Como ya se ha mencionado, la solución elegida para las comunicaciones se trata de un sistema híbrido, utilizando los protocolos *HTTP* y *Websockets*. El primero, es utilizado para la configuración y comandos, mientras que el protocolo *Websockets* se utiliza para la emisión de telemetría en forma periódica, en cada uno de los *hosts*.

A continuación se observa un paquete *HTTP* de configuración luego de ser adquirido, utilizando el software *tshark*:

```
1.722 COMPUTADORA -> EXPERIMENTO TCP 54588 -> 80 [SYN]
1.741 EXPERIMENTO -> COMPUTADORA TCP 74 80 -> 54588 [SYN,
    ACK]
1.741 COMPUTADORA -> EXPERIMENTO TCP 66 54588 -> 80 [ACK]
1.742 COMPUTADORA -> EXPERIMENTO HTTP 170 GET /logger/
    level?value=LOG_INFO&verb=POST
1.761 EXPERIMENTO -> COMPUTADORA HTTP 185 202 Accepted  (
    text/plain)
1.762 COMPUTADORA -> EXPERIMENTO TCP 66 54588 -> 80 [FIN,
    ACK]
1.765 EXPERIMENTO -> COMPUTADORA TCP 66 80 -> 54588 [ACK]
```

La primer columna indica el tiempo transcurrido en segundos desde que se inició la adquisición. De inicio a fin transcurren unos 43 milisegundos. Allí se aprecia como la comunicación es orquestada utilizando el protocolo *HTTP*, en donde numerosas verificaciones suceden entre *host* y *cliente*, aumentando así la confianza. De esta forma, el tiempo transcurrido resulta más que satisfactorio para propósitos de configuración o comandos aislados.

Por otro lado, si consideramos que el computador intermediario tiene que consultar el valor de un sensor, procesar la información y enviar el valor para que el embebido accione los actuadores. El período mínimo para el sistema de control es:

$$T_{min} = 2t_{HTTP} + t_{proc},$$

siendo t_{HTTP} la latencia ocasionada por la comunicación, y t_{proc} el tiempo de procesamiento. Es decir:

$$T_{min} > 2t_{HTTP}$$

En un promedio realizado con 100 muestras, de *endpoints* distintos $t_{proc} = 39$ ms, lo que significa una frecuencia de muestreo máxima de 12.8 Hz. Demasiado lenta para algunos sistemas de dinámicas muy veloces.

La alternativa que se eligió para solucionar este inconveniente, fue la de emitir los datos de sensores y actuadores de forma constante y no solicitada, en forma de telemetría. Es decir, se optó por un patrón *publish-subscribe* en lugar de *request-response*

Dicha telemetría se envía utilizando el protocolo *Websockets*, que a diferencia de *HTTP* solo abre el puerto *TCP* al inicio de cada comunicación, evitando una gran cantidad de mensajes de sincronía y *acknowledgement* luego del primer paquete.

El período con el cual dicha telemetría es emitida es configurable, en la adquisición que se muestra a continuación, se visualiza al experimento emitiendo la información de sus sensores a aproximadamente 20 Hz, mientras que el computador intermediario toma esta información, la procesa y envía los comandos para los actuadores antes que llegue el paquete siguiente, cerrando así un lazo de control a 20 Hz. El patrón de las comunicaciones de telemtría se ven a continuación:

```
3.172 COMPUTADORA -> EXPERIMENTO WebSocket 90 WebSocket
    Text [FIN] [MASKED]
3.196 EXPERIMENTO -> COMPUTADORA WebSocket 226 WebSocket
    Text [FIN]
3.223 COMPUTADORA -> EXPERIMENTO WebSocket 90 WebSocket
    Text [FIN] [MASKED]
3.240 EXPERIMENTO -> COMPUTADORA WebSocket 186 WebSocket
    Text [FIN]
3.275 COMPUTADORA -> EXPERIMENTO WebSocket 90 WebSocket
    Text [FIN] [MASKED]
3.301 EXPERIMENTO -> COMPUTADORA WebSocket 146 WebSocket
    Text [FIN]
```

Es de destacar, que esta frecuencia ya incluye al tiempo de procesamiento. Esto se debe a que el mismo se realiza de manera no-bloqueante respecto de la comunicación de naturaleza asincrónica.

Luego de las respectivas pruebas se concluyó que esta estrategia funciona sin problema alguno para lazos de hasta 100 Hz, lo que permite cubrir la gran mayoría de los sistemas físicos de interés para un laboratorio remoto.

F. Logging y debug

El *logging* es una parte importante durante el proceso de desarrollo, es por esto que el proyecto cuenta con una librería para direccionar mensajes de depuración por la interfaz *UART*, o *Bluetooth* utilizando el módulo *HC-08*.

Esta biblioteca emite mensajes con contexto de hora, nivel de alerta y línea de código en la que fue generado. A continuación se muestra un ejemplo de los logs generados por el experimento:

```
22:25:15 DEBUG lib/driver/src/encoder.c:72 GPIOs de
    econder seteados como entrada
22:25:15 DEBUG tasks/src/send_telemetry.c:48 encoder
    inicializado
22:25:17 WARNING callbacks/src/propeller_cgi.c:86
    inicializando propulsor ...
22:25:17 INFO callbacks/src/propeller_cgi.c:93 propulsor
    inicializado
```

Naturalmente, direccionar estos mensajes a través de la interfaz *UART* consume recursos del procesador. Es por esto que es posible configurarse mediante variables de precompilador, permitiendo así imprimir únicamente los mensajes con nivel mayor o igual al seleccionado. Los niveles posibles, de menor a mayor gravedad son; *trace, debug, info, warning, error* y *fatal*.

G. Datos y memoria embebida

Las tareas relacionadas con la lectura de sensores y publicación de los mismos por telemetría comparten un mismo recurso, encargado de almacenar el valor leído de los sensores, algo similar sucede con los valores destinados a los actuadores. Para compartir esta información entre los distintos hilos de ejecución se utiliza el método de comunicación entre procesos denominado *shared memory*[12].

En la Figura 6, se visualiza un mapa de memoria RAM, donde se designa una posición fija de la memoria correspondiente a variables inicializadas para este propósito.

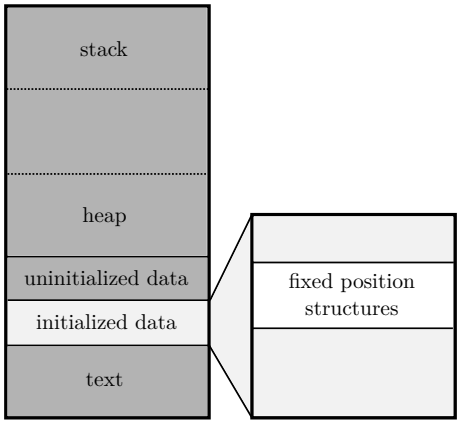

Fig. 6. Mapa de la memoria RAM.

Con el fin de garantizar atomicidad de lectura y escritura los hilos utilizan *mutexes*[13]. Los mismos evitan que más de un proceso modifique el recurso simultáneamente. Esto es, solo pueden utilizarlo una vez que los demás procesos lo hayan liberado. Esto resulta fundamental para evitar corrupciones de memoria.

En las versiones mas recientes de *FreeRTOS* se encuentra disponible la biblioteca `atomic.h`, la cual promete realizar esta tarea utilizando menor cantidad de recursos.

H. Drivers

Para el sistema piloto que se describe en la sección IX fue necesario escribir bibliotecas que permitan comandar los sensores y actuadores del mismo. Éstas, agregan un importante valor a la infraestructura de laboratorios remotos, estableciendo una base para generar comunicaciones a sensores y actuadores de nuevos experimentos.

De esta forma, se plantea a nuevos desarrolladores una programación objetiva de la comunicación, utilizando estructuras que emulen objetos y atributos en lenguajes de alto nivel, proveyendo funciones para interactuar con los mismos.

A continuación se bosqueja un ejemplo que involucra un objeto con sus atributos y métodos, en este caso para un encoder angular incremental.

```
/**
 * \brief   A quadrature encoder structure.
 */
typedef struct EncoderObjectType {
    uint8_t pin_a;      /**< \brief The "a" pin of the
        encoder. */
    uint8_t pin_b;      /**< \brief The "b" pin of the
        encoder. */
    size_t  last_state; /**< \brief A latch of the last
        encoder state. */
    uint16_t value;     /**< \brief The output value. */
} EncoderObjectType;

uint16_t get_encoder_value() {
```

[12]Del inglés, memoria compartida
[13]Del inglés *exclusión mutua*

```
    return encoder.value;
}

void set_encoder_value(uint16_t value) {
    encoder.value = value;
}

void quadrature_encoder_init(uint8_t pin_a, uint8_t pin_b
    ) {
    /* initialization routine */
}

void encoder_intr_handler(uint8_t gpio_num) {
    /* interrupt handler */
}
```

VII. Software de alto nivel

El software considerado de *alto nivel*, comprende a las aplicaciones y bibliotecas que son ejecutadas en sistemas complejos, para este proyecto consideramos de alto nivel a todo software destinado a los computadores no embebidos.

A. Nyquist

1) Propósito: Los comandos para interactuar con el experimento, expuestos hasta el momento, resultan a simple vista crípticos, difíciles de recordar, y poco amigables para un usuario con conocimientos limitados sobre los protocolos *HTTP* y *Websockets*. Como ya se mencionó, el objetivo principal de esta infraestructura es romper con la barrera que representan los conocimientos sobre programación y redes, para que los experimentos sean utilizados por estudiantes o investigadores con mínimos conocimientos de *scripting*, ya sea en *Octave* o bien *Python*.

Para tal fin se programó el paquete *Nyquist* [26], un modesto traductor multiplataforma que permite enviar o recibir comandos y telemetría de laboratorios remotos que se adapten a la infraestructura de comunicación propuesta, abstrayéndose de las dificultades que presentan los diversos protocolos, *handshakes*, asincronísimos, etc.

2) Instalación: El paquete *Nyquist* se encuentra publicado en el repositorio oficial de Python *pip*, y su instalación es tan simple como ejecutar el siguiente comando:

```
pip install nyquist
```

3) Utilización: Con el fin de demostrar lo simple e intuitiva que resulta la programación de un experimento, se adjunta el código necesario para comandar una prueba sobre el prototipo desarrollado, el péndulo aero-propulsado de la sección IX. Si bien este programa resulta intuitivo para un programador familiarizado con *Python*, se escribió la documentación de la biblioteca *Nyquist* [26], la cual documenta de forma exhaustiva cada uno de los puntos necesarios para experimentar con los laboratorios.

```
from nyquist.lab import System
from nyquist.control import Experiment


# we create an inherited class from Experiment
class MyExperiment(Experiment):

    # the three fundamental functions should be defined
    def before_the_loop(self):
        # we will use aero to comunicate with the system
        self.aero = System("aeropendulum")
        self.aero.propeller.pwm.status.post("initialized"
            )
        self.aero.logger.level.post("LOG_INFO")
        self.aero.propeller.pwm.duty.post(self.duty_low)

    def in_the_loop(self):
        angle = self.aero.sensors.encoder.angle.get()
        if angle < self.setpoint_deg:
            self.aero.propeller.pwm.duty.post(self.
                duty_high)
        else:
            self.aero.propeller.pwm.duty.post(self.
                duty_low)

    # the after script will be executed even on failure
    def after_the_loop(self):
        self.aero.propeller.pwm.duty.post(0)
        self.aero.propeller.pwm.status.post("disabled")


# instance experiment
exp = MyExperiment()

# set constants
exp.setpoint_deg = 90
exp.duty_high = 10
exp.duty_low = 5

# and run!
exp.run()
```

4) Extensión: Si se desea agregar un nuevo experimento, es suficiente con generar una lista de los recursos *HTTP* y *Websockets* que dicho experimento tiene disponible. Al hacerlo, la biblioteca *Nyquist* utilizará esta información para crear los atajos necesarios que faciliten la comunicación con el nuevo laboratorio. Este punto resulta vital, ya que el desarrollador de un nuevo sistema dinámico no necesita conocimientos de *Python*, paquetería, o comunicaciones en absoluto.

B. El paquete Websockets para GNU Octave

Dado que el principal objetivo de este proyecto es que los usuarios, ya sea estudiantes o investigadores, experimenten libremente sin requerir amplios conocimientos de programación, la infraestructura no solo es soportada en el lenguaje *Python*, sino también en *GNU Octave*. Este último se ha convertido en uno de los lenguajes más utilizados en el área de sistemas de control, y es idóneo para muchos estudiantes e investigadores.

Si bien *GNU Octave* tiene la capacidad de realizar comunicaciones *HTTP*, y posee una biblioteca para establecer comunicaciones por *Sockets*, carecía de soporte para el protocolo *Websockets*, elegido como uno de los medios principales de comunicación para la infraestructura.

Por este motivo se creó el paquete *Octave websockets* [27], resolviendo la característica faltante para la implementación de la infraestructura. Es de destacar que dicho paquete no solamente sirve al proyecto, sino a todos los usuarios de *GNU Octave* que deseen establecer comunicaciones *Websockets* dentro de sus algoritmos. Dicha biblioteca se publicó en el índice oficial de paquetes de *GNU Octave* y forma parte de los 86 paquetes de *Octave* al día de la fecha.

1) Utilización: Para instalar *Octave Websockets*, es necesario ejecutar el siguiente comando dentro de *GNU Octave*:

```
pkg install "https://github.com/gnu-octave/octave-
    websockets/archive/v0.1.0.tar.gz"
```

Un ejemplo de uso muy sencillo se puede ver a continuación:

```
host = "192.168.0.10"
uri = "ws://propeller/pwm/duty"
port = 80
```

```
% conectarse
ws = ws_connect (host, uri, port)

mode = "text"
% enviar datos
ws_send (ws, data, mode)

% recibir respuesta
resp = ws_receive (ws)
```

C. Conexión entre el usuario y el servidor intermedio

En las secciones anteriores, si bien se resuelven los problemas de conectividad entre el experimento remoto y el servidor intermedio, aún queda pendiente demostrar como un usuario se conecta a dicho servidor intermedio. Para esto se desarrollaron varias soluciones con posibilidades de aplicación muy diferentes entre sí, basadas en las herramientas *Gitlab Runners* y *Remote Mole* (esta última creada específicamente para este proyecto).

Como se explicó en la sección II, dentro de los laboratorios remotos existen diferentes enfoques en cuanto a la forma de conexión que establece el cliente (usuario) con el servidor (experimento); intrusivos y no-intrusivos. Cada uno de estos, posee ventajas y desventajas marcadas, que dependerán del caso de uso. Un investigador que está probando sus nuevos algoritmos en un laboratorio remoto, requerirá un acceso intrusivo al mismo, que le permita observar cuantas variables quiera, instalar paquetes, modificar entornos, entre otros. Por otra parte, para un grupo de estudiantes será mas apropiado el acceso no-intrusivo que los aísle de forma segura de la plataforma, proveyendo también una adecuada orquestación para administrar el acceso.

1) Remote Mole: Esta herramienta [28] permite a un usuario acceder de forma "intrusiva" al experimento. La aplicación posibilita la creación remota de túneles *SSH*, *Jupyter Notebooks* y manejo de instrumentos remotos, tales como un osciloscopio, todo a través de una interfaz de mensajería, utilizando la plataforma *Discord*.

Los túneles *SSH* facilitan el acceso al dispositivo, como si se tratase de una red local. Además, por tratarse de un protocolo abierto y sumamente popular existe un sinnúmero de programas, sobre cualquier plataforma, para que el usuario acceda de manera remota, proporcionando usuario y contraseña. Si el usuario se encuentra en la misma red de área local, o si posee una IP pública, crear un tunnel utilizando la herramienta podría no ser necesario, sin embargo esas no son condiciones que podamos garantizar. En la Figura 7 se ve como el usuario pide al *Bot* que cree un túnel.

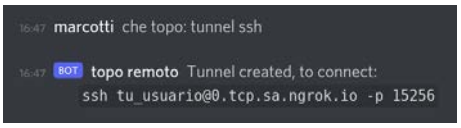

Fig. 7. Interacción de creación de túnel SSH.

Luego, para acceder, simplemente se utiliza el comando provisto, seguido de la contraseña del usuario:

```
marco ~ > ssh marco@0.tcp.sa.ngrok.io -p 15256
marco@0.tcp.sa.ngrok.io's password:
Linux raspberrypi 5.10.63-v7l+ #1459 SMP Wed Oct 6
     16:41:57 BST 2021 armv7l

The programs included with the Debian GNU/Linux system
     are free software;
the exact distribution terms for each program are
     described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to
     the extent
permitted by applicable law.
Last login: Sun Jul  4 15:58:24 2021 from ::1
marco@raspberrypi:~ $
```

Un usuario podrá también valerse de una interfaz mas amigable e intuitiva, como son los *Jupyter Notebooks*. Con la misma herramienta, *Remote Mole*, podrá direccionar el puerto dedicado para estos *Notebooks* hacia una dirección pública. Esto es muy útil, incluso dentro de la misma red local. En la Figura 8 se observa el comando de invocación del túnel.

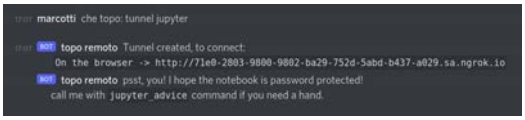

Fig. 8. Interacción de creación de túnel Jupyter.

Al ingresar en el enlace indicado, es necesario ingresar las credenciales del usuario, luego se debe seleccionar el archivo sobre el cual trabajar (Figura 9) y finalmente desarrollar código sobre el mismo (Figura 10).

Fig. 9. Visualizador de archivos.

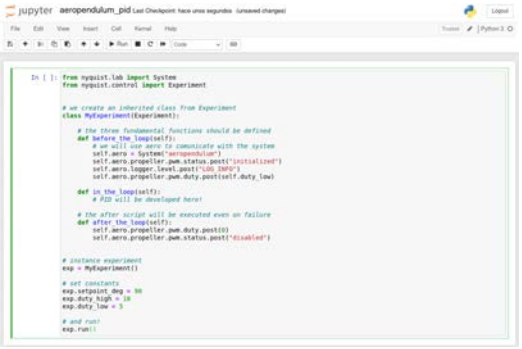

Fig. 10. Desarrollo con acceso a un experimento.

2) Herramientas LXI: Por último, *Remote Mole* también permite acceder a instrumentación remota a través del protocolo *LXI*. De este modo, el usuario está en condiciones de conectarse a un instrumento, ya sea una fuente de alimentación, un osciloscopio o un generador de funciones, modificar sus configuraciones, pedir datos o realizar capturas de pantalla. En las Figuras 11, 12 y 13 se aprecia como usuarios se conectan a un osciloscopio, envían comandos de configuración y piden información de su pantalla.

En consecuencia, dicha herramienta resulta una característica prometedora, ya que no solamente permite acceder a los experimentos sino también a instrumentos de medición, a partir de estándares conocidos. Utilizando esto como base, la cantidad de experiencias prácticas, relacionadas a

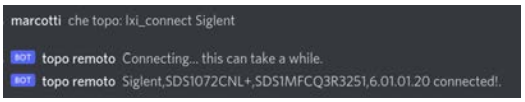

Fig. 11. Establecimiento de conexión con osciloscopio Siglent.

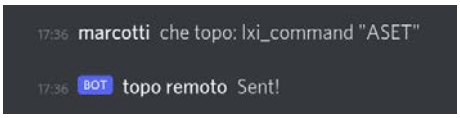

Fig. 12. Comando de auto-set enviado al a osciloscopio.

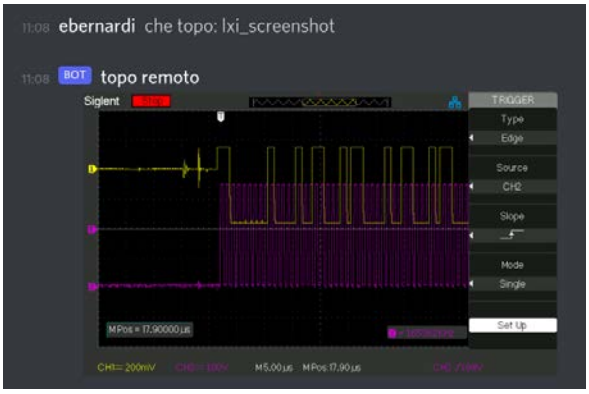

Fig. 13. Pedido de captura de pantalla al osciloscopio.

la electrónica, que se pueden llevar a cabo son incontables. Esto es, electrónica analógica, señales y sistemas, dispositivos electrónicos, electrónica digital, programación, etc.

3) Gitlab Runners: La tecnología de *Gitlab Runners*, o simplemente *runners* utilizada ampliamente en la industria del software para resolver los problemas de integración y despliegue continuo resulta una solución propicia para el uso no supervisado de los experimentos, ya que dichos *runners* permiten ejecutar rutinas de código previamente escritas por un estudiante y subidas a un repositorio *git* hosteado en *Gitlab*. Además, de ser necesario es posible compilar el código fuente, revisar y simular el mismo para verificar que no existan peligros para el experimento, antes de ejecutarlo en la plataforma, todo de forma automática.

El proceso es muy sencillo, únicamente se debe:

- Escribir el código del experimento en uno de los proyectos generados para estos experimentos.
- Subirlo al repositorio remoto de *git*.

VIII. Entorno de trabajo

A. Contenerización

En algunas ocasiones durante este trabajo se mencionó la *replicabilidad*. Esta característica indica que tan posible es reproducir lo que se hizo en este proceso, mientras menos esfuerzo requiera, más replicable es.

Con el objetivo de reducir este esfuerzo, todas las herramientas de desarrollo se encuentran contenerizadas. Lo que permite ejecutar algo similar a una maquina virtual, aunque ligero y ágil, para compilar, testear y ejecutar código. Así, con este esfuerzo ya no es necesario instalar programas en la computadora personal, únicamente se debe descargar una imagen que genere un contenedor para dichas dependencias, y trabajar en el mismo.

Todos los problemas de dependencias se resuelven con tres comandos:

```
docker pull marcotti/esp-open-rtos
docker pull marcotti/esptool-minimal
docker pull marcotti/remote-mole-dev
```

B. Integración continua

Todas las herramientas generadas poseen *pipelines* de integración continua. Esto es, grupos de pasos automatizados que compilan, testean y despliegan el código cada vez que se sube, y periódicamente, para asegurar que el desarrollo esté siempre funcionando y listo para utilizarse. En la Figura 14 se aprecian los pasos ejecutados con cada actualización de *remote-control-lab*.

Fig. 14. *Pipeline* de *remote-control-lab*.

C. Documentación del proyecto

Dentro de la integración continua que se comentó en el punto anterior existe un paso para la generación automática de documentación, en donde se compilan y publican las páginas web que describen la utilización de esta infraestructura, de forma automática [22], [24], [26], [28].

IX. Sistema piloto: péndulo aero-propulsado

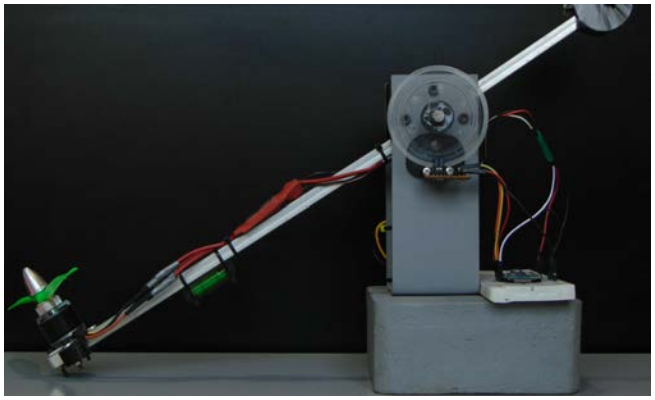

Fig. 15. Fotografía del sistema construido.

Como ya se detalló en reiteradas ocasiones a lo largo del proyecto, se construyó un sistema piloto para demostrar el potencial y versatilidad del *framework* creado. El mismo, que se muestra en la Figura 15, trata de un péndulo propulsado a través de una hélice de *drone*, con un motor sin escobillas.

El principio de operación de este sistema se basa en establecer un ángulo respecto al vector de gravedad (set-point), para que en base a la manipulación del empuje de la hélice se compense la desviación entre el ángulo actual y el de set-point. Es de destacar que este sistema es no lineal SISO[14].

Entre sus posibilidades, este sistema permite realizar pruebas con filtros de Kalman aplicando fusión de sensores,

[14]Del inglés, Simple-Input Simple-Output

ya que posee un acelerómetro/giróscopo, junto a un *encoder* incremental, acoplados en el eje de rotación. Esto, se ha realizado previamente en otros trabajos [29] y su aplicación en este contexto resulta relevante.

Específicamente, el péndulo aero-propulsado es un sistema de dinámica rápida, por lo que pone a prueba los límites establecidos por la latencia en las comunicaciones. A modo de ejemplo, el experimento de observar la respuesta a un cambio de set-point del tipo escalón fue programado con unas pocas líneas de código, utilizando la biblioteca *nyquist*. En la Figura 16 se muestra el resultado de la ejecución de dicho algoritmo.

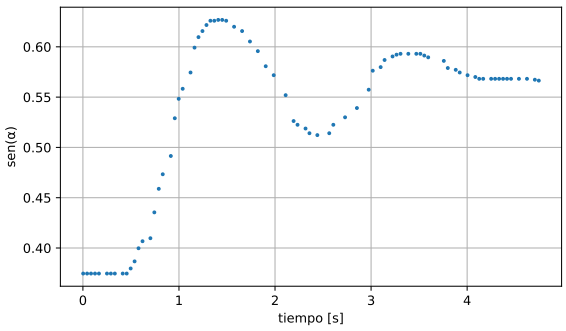

Fig. 16. Respuesta al escalón.

La respuesta temporal obtenida deja en evidencia que la tasa de actualización de datos alcanzada es suficiente para este tipo de aplicaciones. Esto es, un lazo de control puede ejecutarse a una frecuencia menor a 100 Hz sin problema alguno, utilizando los sensores y actuadores del experimento.

Si bien es un experimento *único en su clase*, se priorizó una construcción de bajo costo, modular y fácilmente replicable. Para este fin se aprovechó la existencia de módulos de desarrollo provistos en masa, posibles gracias a la corriente de democratización del hardware [30]. Además, se evitó, en la medida de lo posible, construir hardware *ad-hoc*, debido a que esto afecta negativamente a la replicabilidad del experimento. Finalmente, también se confeccionaron y detallaron planos mecánicos.

En la Figura 17 se ve la respuesta del experimento a diferentes tipos de algoritmos de control, entre ellos P, PI, PD y múltiples sintonizaciones de controladores PID, todos estos posicionando al péndulo para el mismo *set-point*.

El código necesario para realizar esta tarea resulta increíblemente sencillo, gracias a la abstracción que provee el proyecto, permitiendo concentrarse realmente en el algoritmo de control, sin prestar atención a la construcción y comunicación con el experimento.

La programación del mismo consta de las siguientes elementales líneas:

```
class MyExperiment(Experiment):

    def before_the_loop(self):
        self.aero = System("aeropendulum")
        self.aero.propeller.pwm.status.post("initialized"
            )
        self.aero.telemetry.period.post(20)
        self.aero.logger.level.post("LOG_INFO")
        self.data = []  # {'time': %f, 'angle': %f, '
            setpoint': %f}
```

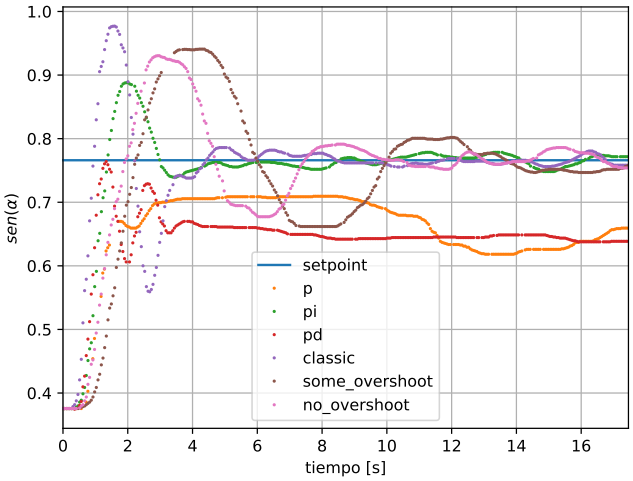

Fig. 17. Comparación de controladores en el experimento.

```
        pid_coeffs = ziegler_nichols(Ku=40, Tu=1.45,
            control_type='no overshoot')

        self.pid = PIDController(
            pid_coeffs['Kp'],
            pid_coeffs['Ki'],
            pid_coeffs['Kd'],
            20,
            0.05
        )
        self.linerizer = PendulumLinearizer(
            41.67880775502065,
            4.570736581055918,
        )
        self.aero.sensors.encoder.angle.get()
        self.start_ts =  time.monotonic()

    def in_the_loop(self):
        angle_deg = self.aero.sensors.encoder.angle.get()
        if angle_deg is None:
            return

        angle_deg = avoid_negative_angles(angle_deg)
        angle_rad = np.deg2rad(angle_deg)

        spent = time.monotonic() - self.start_ts


        pid_duty = self.pid.update(self.setpoint_rad -
            angle_rad)
        compensator_duty = self.linerizer.compensate(
            angle_rad)
        duty = pid_duty + compensator_duty
        duty = safety_check(duty, self.max_duty,
            SAFE_DUTY)
        self.aero.propeller.pwm.duty.post(duty)

        self.data.append(
            {
                'time': spent,
                'sin_angle': np.sin(angle_rad),
                'duty': duty,
                'sin_setpoint': np.sin(self.setpoint_rad)
                    ,
            }
        )

    def after_the_loop(self):
        self.aero.propeller.pwm.duty.post(0)
        self.aero.propeller.pwm.status.post("disabled")
        self.aero.propeller.pwm.status.post("initialized"
            )
```

Naturalmente, fue necesario crear unas pequeñas bibliotecas para un controlador PID, y para linealizar el sistema (de naturaleza no-lineal). Si bien estas pueden ser también utilizadas por el usuario, la construcción de las mismas proveerá al mismo un mejor entendimiento en la imple-

mentación de la teoría de control.

X. Conclusión

A través de este trabajo se ha desarrollado un sistema escalable y replicable, que permite a docentes, estudiantes, profesionales y aficionados construir sus propios laboratorios remotos. Aportando así a una educación descentralizada y cada vez más colaborativa.

Esto es, mediante la construcción del sistema piloto, el *framework* desarrollado mostró ser versátil y a su vez intuitivo. Por lo que adaptarlo a las necesidades que imponía este experimento no representó un problema. Además, la plataforma propuesta logró suplir las necesidades de *performance* requeridas por un sistema de rápida dinámica. Es por ello, que al satisfacer tales necesidades, la gama de experiencias que pueden generarse utilizando esta interfaz es realmente amplia.

Por otro lado, luego de meses de pruebas internas y experimentación constante, las herramientas de educación remota resultaron ser extremadamente útiles. Las interfaces y el software generado fue puesto a prueba en el dictado de clases y en grupos de investigación, permitiendo a los usuarios manipular herramientas a distancia para comprender sistemas complejos o acceder a instrumentos de medición.

La *API* generada demostró ser transparente y eficaz, permitiendo con pocas líneas de código generar algoritmos de control como los ya mencionados, implementando capas de comunicación complejas en una sola línea.

Por último, la documentación web desarrollada resulta suficiente para que cualquier usuario, a través de la experimentación y manipulación de los ejemplos provistos, disfrute y democratice el aprendizaje.

Referencias

[1] P. C. Wankat and F. S. Oreovicz, *Teaching engineering*. Purdue University Press, 2015.

[2] K. Ogata, *Modern Control Engineering*, 5th ed. Pearson Education, 2010.

[3] F. Golnaraghi and B. C. Kuo, *Automatic Control Systems*, 9th ed. Wiley, 2009.

[4] E. J. Adam, *Instrumentación y Control de Procesos. Notas de Clase*, 3rd ed. Santa Fe: Ediciones UNL, 2018.

[5] E. S. Burgos and E. J. Adam, "Graphical user interface editor for octave applications," *Engineering Reports*, p. e12269, 2020.

[6] L. Gomes and S. Bogosyan, "Current trends in remote laboratories," *IEEE Transactions on industrial electronics*, vol. 56, no. 12, pp. 4744–4756, 2009.

[7] J. Sáenz, L. de la Torre, J. Chacón, and S. Dormido, "A study of strategies for developing online laboratories," *IEEE Transactions on Learning Technologies*, vol. 14, no. 6, pp. 777–787, 2021.

[8] C. Gravier, J. Fayolle, B. Bayard, M. Ates, and J. Lardon, "State of the art about remote laboratories paradigms-foundations of ongoing mutations," *International Journal of Online Engineering*, 2008.

[9] M. Ngolo, L. B. Palma, F. Coito, L. Gomes, and A. Costa, "Architecture for remote laboratories based on rest web services," in *2009 3rd IEEE International Conference on E-Learning in Industrial Electronics (ICELIE)*. IEEE, 2009, pp. 30–35.

[10] M. Tawfik, C. Salzmann, D. Gillet, D. Lowe, H. Saliah-Hassane, E. Sancristobal, and M. Castro, "Laboratory as a service (laas): a novel paradigm for developing and implementing modular remote laboratories," *International Journal of Online and Biomedical Engineering (iJOE)*, vol. 10, no. 4, pp. 13–21, 2014.

[11] C. Salzmann, S. Govaerts, W. Halimi, and D. Gillet, "The smart device specification for remote labs," in *Proceedings of 2015 12th International Conference on Remote Engineering and Virtual Instrumentation (REV)*. IEEE, 2015, pp. 199–208.

[12] L. Gomes, F. Coito, A. Costa, and L. B. Palma, "Teaching, learning, and remote laboratories," *Advances on remote laboratories and e-learning experiences*, p. 189, 2007.

[13] J. García-Zubía, D. López-de Ipiña, and P. Orduña, "Evolving towards better architectures for remote laboratories: a practical case," *International Journal of Online and Biomedical Engineering (iJOE)*, vol. 1, no. 2, 2005.

[14] L. S. Indrusiak, M. Glesner, and R. Reis, "On the evolution of remote laboratories for prototyping digital electronic systems," *IEEE Transactions on Industrial Electronics*, vol. 54, no. 6, pp. 3069–3077, 2007.

[15] L. De La Torre, L. T. Neustock, G. K. Herring, J. Chacon, F. J. G. Clemente, and L. Hesselink, "Automatic generation and easy deployment of digitized laboratories," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 12, pp. 7328–7337, 2020.

[16] F. J. Rodriguez, C. Giron, E. J. Bueno, A. Hernandez, S. Cobreces, and P. Martin, "Remote laboratory for experimentation with multilevel power converters," *IEEE Transactions on Industrial Electronics*, vol. 56, no. 7, pp. 2450–2463, 2009.

[17] H. Bähring, J. Keller, and W. Schiffmann, "Remote operation and control of computer engineering laboratory experiments," in *Proceedings of the 2006 workshop on Computer architecture education: held in conjunction with the 33rd International Symposium on Computer Architecture*, 2006, pp. 6–es.

[18] A. Leva and F. Donida, "Multifunctional remote laboratory for education in automatic control: The crautolab experience," *IEEE Transactions on Industrial Electronics*, vol. 55, no. 6, pp. 2376–2385, 2008.

[19] D. Hargreaves, "Student learning and assessment are inextricably linked," *European Journal of Engineering Education*, vol. 22, no. 4, pp. 401–409, 1997.

[20] M. Wu, J.-H. She, G.-X. Zeng, and Y. Ohyama, "Internet-based teaching and experiment system for control engineering course," *IEEE Transactions on Industrial Electronics*, vol. 55, no. 6, pp. 2386–2396, 2008.

[21] M. Casini, D. Prattichizzo, and A. Vicino, "The automatic control telelab," *IEEE Control Systems Magazine*, vol. 24, no. 3, pp. 36–44, 2004.

[22] M. Miretti, "Remote Control Lab documentation," https://marcomiretti.gitlab.io/remote-control-lab, 2021, accessed: 2021-07-11.

[23] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[24] M. Miretti, "Remote Labs - Control openapi specification," https://marcomiretti.gitlab.io/remote-control-lab/openapi.html, 2021, accessed: 2021-07-11.

[25] Zaitsev, Serge, "Jsmn: The most simple json parser for c in small systems," https://zserge.com/jsmn/, 2019, accessed: 2021-10-03.

[26] M. Miretti, "Nyquist rtd documentation," https://nyquist.readthedocs.io/en/latest, 2021, accessed: 2021-07-11.

[27] ——, "Octave websockets," https://gnu-octave.github.io/packages/websockets, 2020, accessed: 2021-10-31.

[28] ——, "Remote Mole rtd documentation," https://remote-mole.readthedocs.io/en/latest/, 2021, accessed: 2021-07-11.

[29] M. Miretti, F. Busano, E. Bernardi, H. Pipino, and G. Peretti, "Estimación híbrida de posición angular en dispositivos de captura de imágenes aéreas," *VIII Jornadas de Ciencia y Tecnología para Alumnos (CyTAL)*, 08 2018.

[30] L. Buechley, "Lilypad arduino: How an open source hardware kit is sparking new engineering and design communities," 2009.