UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Servicio de compilación de programas en C en modo terminal para el entorno Native client de Google Chrome

Trabajo fin de grado

Grado en Ingeniería Informática

Autor: Emilio Rodríguez Revert
Tutor: Salvador España Boquera

Curso 2016-2017

Resumen

El lenguaje de programación C es un lenguaje de alta relevancia en el campo de la ingeniería que ha servido como precursor de otros lenguajes actuales (véase C++, Java, C#, JavaScript, Objective-C,…). Esto hace de C un buen lenguaje para iniciarse en la programación y la prueba es que es utilizado en asignaturas de primer curso de varios grados de la UPV. Sin embargo existe una barrera de entrada para nuevos usuarios al necesitar de una serie de herramientas dependientes de cada entorno para poder compilar y ejecutar los programas. Este trabajo rompe con estas barreras al proporcionar un entorno web de edición, compilación y ejecución de programas. A diferencia de otras plataformas similares, nuestra propuesta ejecuta los programas escritos en C directamente sobre el navegador en lugar de utilizar un servidor para su ejecución (redireccionando la entrada/salida a la web). Esto es novedoso, incrementa notablemente la seguridad del servidor y reduce los tiempos de respuesta percibidos por el usuario.

Palabras clave: Lenguaje de programación C; Google Native Client; Aplicación web; Contenedores Docker

Resum

El llenguatge de programació C és un llenguatge d'alta rellevància en el camp de l'enginyeria que ha servit com a precursor d'altres llenguatges actuals (com C++, Java, C#, JavaScript, Objective-C,…). Açò fa de C un bon llenguatge per a iniciar-se en la programació i la prova és que és emprat en assignatures de primer curs en diversos graus de la UPV. No obstant això, hi ha una barrera d'entrada per a nous usuaris al necessitar una sèrie de ferramentes dependents de cada entorn per a poder compilar i executar els programes. Aquest treball trenca amb aquestes barreres al proporcionar un entorn web d'edició, compilació i execució de programes que, a diferència d'altres plataformes semblants, executa els programes escrits en C directament sobre el navegador en compte d'utilitzar un servidor per a la seua execució (redireccionant l'entrada/eixida a la web). Açò és una novetat que a més incrementa notablement la seguretat i redueix el temps de resposta percibit per l'usuari.

Paraules clau: Llenguatge de programació C; Google Native Client; Aplicació web; Contenedors Docker

Abstract

The C programming language is a highly relevant language in the field of engineering that has served as a precursor to other current languages (see C ++, Java, C#, JavaScript, Objective-C,…). This makes of C a good language to start programming and, indeed, it is used nowadays as the language of choice to learn programmin in several engineering grades at UPV. However, there is an entry barrier for new users since a series of tools, dependent on each environment, are requried to compile and run the programs. This work breaks down these barriers by providing a web environment for editing, compiling and executing programs that, unlike other similar platforms, can run programs written in C on the browser instead of using a server for execution (by redirecting the input/output to the web). This is not only new but it also increases the security of the server and reduces response times perceived by the user.

Key words: C programming language; Google Native Client; Web application; Docker containers

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1

Introducción

Motivación

Desarrollar programas resulta imprescindible para adquirir las habilidades necesarias en las asignaturas de programación. En la UPV se utiliza el lenguaje C en la asignatura "Informática" en diferentes grados (Ingeniería Mecánica, Eléctrica, etc.) de la Escuela Técnica Superior de Ingeniería del Diseño (ETSID).

A pesar de la existencia de diversas herramientas para programar en C, la instalación de las mismas puede suponer una barrera para trabajar en casa o, en general, fuera del laboratorio.

Durante muchos cursos, en la ETSID se ha utilizado el entorno Dev-C++ [16] aunque existen otras alternativas como Code::Blocks [5] (multiplataforma) o Xcode [21] (para macOS), etc. En general, este tipo de herramientas pueden dar problemas tanto en los laboratorios docentes como cuando los alumnos lo tienen que instalar en su casa:

- En los laboratorios: a veces hay cambios de aula a laboratorios donde no está instalado el software de la asignatura, especialmente en los exámenes. También existe un problema de interferencia con el antivirus que causa problemas de ejecución. Por ejemplo: no deja generar ejecutable, no deja ejecutarlo, o tarda mucho (más de un minuto) en compilar…
- Para los alumnos: si no utilizan Windows han de recurrir a otras herramientas (ej: Xcode para macOS), normalmente difíciles de instalar para un usuario nuevo y que, además, suponen problemas a la hora de seguir las prácticas debido a las diferencias que pueden presentar con el programa que se está usando en los laboratorios.

En este curso académico 2016-2017 se ha realizado una prueba en la que, en un grupo piloto, se ha sustituido el entorno Dev-C++ por un entorno vía web llamado Codeboard [3] que presenta varias ventajas:

- No es necesario instalar ningún programa (el navegador web suele venir de serie).
- Al trabajar "en la nube" el alumno siempre tiene acceso a sus programas sin necesidad de acceder a su cuenta de la universidad.
- Permite recopilar estadísticas.
- Permite crear proyectos partiendo de plantillas existentes directamente.
- Aunque no ha sido nuestro caso, es una mejor opción para cursos tipo MOOC ya que centra toda la interacción del usuario en el navegador. De hecho, este entorno es utilizado, por ejemplo, en algunos cursos de la plataforma edX [8].

A pesar de estas ventajas, utilizar Codeboard no está exento de problemas. Si bien es cierto que actualmente han surgido entornos accesibles vía web, estos necesitan ejecutar los programas en un servidor propio ya que, en principio, el lenguaje C no se ejecuta en los navegadores web. Ejecutar código de los usuarios en un servidor presenta algunos problemas o inconvenientes:

- Resulta peligroso o vulnerable para la entidad que mantiene el servidor, ya que se da al usuario total libertad para implementar y ejecutar programas en un entorno ajeno. Esto se vuelve muy difícil de controlar aún recurriendo a ejecutar estos en un sandbox y por ello, entre otros motivos, se requiere estar registrado para trabajar.
- En momentos de carga los usuarios pueden percibir una ejecución lenta de sus programas debido a las altas latencias.
- Es complicado escalar la aplicación de forma adecuada ya que unas pocas ejecuciones pueden disparar el uso de CPU del servidor en intervalos de tiempo indefinidos, lo que causa picos irregulares e inestabilidad.
- La complejidad de la implementación, ya que se han de tener en cuenta diversos factores de seguridad (evitar código malicioso, ejecutar en sandbox…, etc.) y sincronizar las sesiones de los usuarios con las ejecuciones de sus proyectos conectando la entrada/salida de estos en tiempo real con el navegador.

Los problemas de latencia y de carga se pueden resolver, en parte, instalando el software en servidores de la propia institución en lugar de recurrir a los servidores globales de Codeboard. Esto parece posible teniendo en cuenta que hay una versión de Codeboard disponible públicamente en `github`. Sin embargo, en la práctica no es tan fácil debido a que, por una parte, el repositorio está bastante desactualizado (tiene más de un año de antigüedad a pesar de que sabemos que se realizan actualizaciones recientemente) y, por otra parte, intentos frustrados de instalarlo durante más de una semana nos han hecho pensar que falta documentación esencial para este proceso. Estas dificultades nos han motivado a añadir, como objetivo secundario de este trabajo, la facilidad de despliegue de la herramienta desarrollada de una manera lo más sencilla posible.

El objetivo de este trabajo es mejorar los entornos de desarrollo vía web para tener sus ventajas sin los inconvenientes derivados de la ejecución de los programas en el propio servidor.

En este trabajo presentamos una herramienta para que los estudiantes puedan probar (compilar y ejecutar) los programas que desarrollan con la particularidad de que, al conseguir ejecutar los programas en el propio navegador, se reducen drásticamente los problemas de seguridad al tiempo que se mejoran los tiempos de respuesta percibidos por el usuario y la escalabilidad.

Esta herramienta, accesible con un navegador web adecuado, permite la prueba de cualquier programa sencillo en C sin necesidad tan siquiera de registrarse o identificarse, facilitando todavía más el desarrollo de las prácticas.

Objetivos

El objetivo de este proyecto es desarrollar un servicio que permita a los usuarios, a través de un navegador,[1] escribir, compilar y ejecutar programas escritos en lenguaje C de forma nativa sobre el hardware de su máquina y sin tener que instalar ningún software adicional. Rompiendo así con la principal barrera de entrada al aprendizaje de la programación que

[1] Como veremos más adelante, se tratará específicamente de Google Chrome.

supone el instalar toda una serie de herramientas diferentes en cada máquina o sistema operativo para poder escribir, compilar y ejecutar los programas escritos.

Como ya hemos mencionado, un objetivo secundario es que el servidor asociado a esta herramienta resulte lo más sencillo de instalar para cualquier docente sin necesitar grandes conocimientos técnicos.

Para conseguir el objetivo principal hemos dividido el trabajo en otros tres objetivos o hitos clave:

- Implementar una Interfaz de Programación de Aplicaciones (API) que se encargará de proveer a la aplicación web de las funciones y eventos en tiempo real del servidor además de mantener una cola de trabajos pendientes de compilación y persistir los datos en el servidor.
- Implementar un proceso trabajador o *worker* que pueda desplegarse de forma escalable para adaptar la aplicación a los distintos niveles de carga y que se encargará de vigilar constantemente la cola de trabajos del servidor. De esta manera, en el momento surja un nuevo trabajo se encargará de realizar la compilación, guardar el binario en la base de datos y comunicar al servidor que el trabajo ha finalizado y su resultado. Así, mediante un evento, se consigue lanzar la ejecución en el navegador del usuario.
- Implementar una aplicación web que sirva para demostrar la funcionalidad proveída por los anteriores objetivos en la que un usuario podrá escribir un fragmento de código C, compilarlo y ejecutarlo de forma nativa directamente en su navegador y bajo su hardware.

Para abordar el objetivo secundario hemos decidido utilizar contenedores software tal y como se explicará en el capítulo 5.

Estructura del documento

El actual documento se ha estructurado en 6 capítulos como se describe a continuación:

Capítulo 2: Contexto previo

Debido a que este trabajo intenta romper con la metodología utilizada hasta ahora para desarrollar entornos integrados de desarrollo, se ha dedicado este capítulo a analizar las aplicaciones disponibles actualmente al tiempo que damos algunos indicios sobre el método propuesto que será visto con detalle en los siguientes capítulos.

Capítulo 3: Diseño

Antes de la fase de implementación es necesario realizar un diseño. En este diseño se analiza cómo será la herramienta deseada y cómo se va a desarrollar. En este capítulo se ofrece tanto un vistazo global al proyecto, aportando los diagramas que muestran el flujo completo de la aplicación, como un examen más detallado a nivel de las tres partes fundamentales: API, Worker y Web.

Capítulo 4: Implementación

En esta fase se explica cómo se ha realizado la implementación del sistema basándonos en el diseño propuesto anteriormente. Nos vamos a centrar en cada una de las partes que componen esta aplicación distribuida. Se explicarán las tecnologías utilizadas y el porqué de las mismas.

Capítulo 5: Despliegue

En este capítulo se muestra detalladamente cómo realizar el despliegue de la aplicación final. Uno de los objetivos perseguidos es conseguir que cualquier usuario pueda montar un servicio de compilación de programas C sin apenas conocimientos ejecutando un único comando. Mostraremos el sistema utilizado para realizar este despliegue de una forma sencilla y el porqué de la elección del mismo.

Capítulo 6: Conclusiones

Llegados a este punto, el capítulo final del trabajo lo dedicaremos al análisis de las conclusiones a las que se ha llegado durante la realización de este proyecto. También abordaremos otros aspectos como las aportaciones técnicas y académicas, así como los posibles trabajos futuros.

CAPÍTULO 2
Contexto previo

Introducción

Actualmente existen múltiples herramientas que nos permiten escribir, compilar y ejecutar programas escritos en el lenguaje de programación C. Pueden existir diversos criterios de clasificación. Así, es posible distinguir entre el uso separado de editores y compiladores (ej: `Vim`/`Emacs`/… por un lado y `gcc`/`clang`/… por otro) y entornos de desarrollo integrado (IDE). Si nos centramos en estos últimos podemos dividirlos en dos grupos:

- Herramientas de escritorio, que se instalan y ejecutan en el propio puesto de trabajo.
- Herramientas en la nube, a las que se accede, normalmente, vía web.

Veamos con más detalle cada una de ellas desde la perspectiva del uso docente que ha motivado este trabajo.

Herramientas de escritorio

Son las que se han venido usando desde hace muchos cursos en las asignaturas "Informática" en diversos grados de Ingeniería de la ETSID, en la UPV. Estas herramientas están formadas o dependen de dos partes fundamentales:

- Interfaz de usuario: que permite a los usuarios crear proyectos y trabajar el código de estos.
- Toolkit de compilación: es el set de dependencias que utiliza la herramienta para compilar los proyectos creados con esta para el entorno de la máquina en la que se compilan.

Esta segunda parte, el toolkit de compilación, es distinto dependiendo de cada entorno (sistema operativo, arquitectura del procesador…, etc.). Así, por ejemplo, Dev-C++ parece utilizar internamente Cygwin [18] para compilar. Esta distinción resulta en parte irrelevante para el usuario final y le queda normalmente oculta. Lo relevante es más bien la dificultad de la instalación, que depende en gran medida de si se dispone de un sistema sencillo de instalación. Incluso cuando hay un programa "instalador", como en el caso del Dev-C++ [16], pueden surgir complicaciones para nuevos usuarios que tratan de iniciarse en el mundo de la programación. Así, por ejemplo, está la interferencia con el antivirus mencionada en el capítulo anterior.

Además, aunque existen IDEs específicamente pensados para el uso docente (como es el caso de BLUEj [9] para Java), habitualmente se trata de herramientas de programación profesional que disponen de un gran conjunto de características o funcionalidades y que

están orientadas al desarrollo de proyectos de cualquier envergadura, lo que redunda en mayor dificultad para los alumnos de primeros cursos que, normalmente, no necesitan recurrir a proyectos ni hacer uso de estas características.

Herramientas en la nube

Debido al gran avance de las tecnologías web existen, actualmente, múltiples entornos de edición de código online [1]. Habitualmente están orientados a la facilidad de uso y eliminan una de las principales barreras de entrada para nuevos usuarios que supone la instalación de todos los programas necesarios para empezar a programar.

Estos entornos resultan especialmente idóneos cuando el lenguaje de programación es JavaScript, debido a que es el único que puede ejecutarse de forma directa en todos los navegadores actuales.

Por otro lado, si el lenguaje objetivo es C, su implementación es costosa y complicada ya que como, en principio, el código C no puede ejecutarse en navegadores, se recurre a realizar la compilación y ejecución en servidores propios. Esto provoca serios problemas de seguridad, rendimiento y escalabilidad al tener que controlar en todo momento qué se esta ejecutando y redirigir en tiempo real la entrada/salida de los programas hacia el navegador del usuario que los está ejecutando.

Análisis de algunas herramientas existentes

En este punto daremos un vistazo a las herramientas que se han estado usando hasta ahora dividiéndolas en las dos categorías vistas en el punto anterior. Con esto pretendemos analizar los precedentes de cara a mejorar nuestro diseño y a que se entienda con mayor claridad las ventajas que perseguimos.

Herramientas de escritorio

- Dev-C++: es un sencillo entorno de desarrollo integrado para windows para los lenguajes de programación C y C++ [16]. La Tabla 2.1 resume las características de este entorno. Una captura del mismo puede observarse en la Figura 2.1.

Tabla 2.1: Características de Dev-C++

Plataforma	Lenguajes	Compiladores
Windows	C C++	Basados en GCC

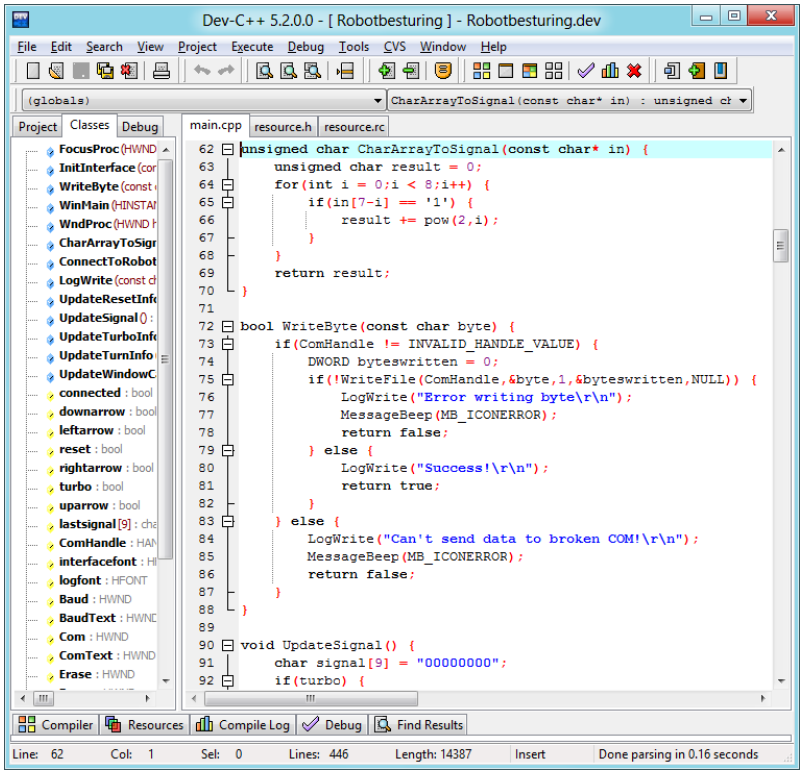


Figura 2.1: Captura de Dev-C++

- Code::Blocks: es un entorno de desarrollo integrado multiplataforma para escritorio [5]. Dispone de una gran variedad de compiladores para los lenguajes C, C++ y Fortran. Sin duda alguna se trata de un IDE sencillo (véase la Figura 2.2) pero que de cara a nuevos usuarios puede dar problemas debido a la gran variedad de compiladores que ofrece (ver Tabla 2.2).

Tabla 2.2: Características de Code::Blocks

Plataforma	Lenguajes	Compiladores
Linux	C	GCC (MingW / GNU GCC)
Mac	C++	MSVC++
Windows (wxWidgets)	Fortran	clang
		Digital Mars
		Borland C++ 5.5
		Open Watcom
		...

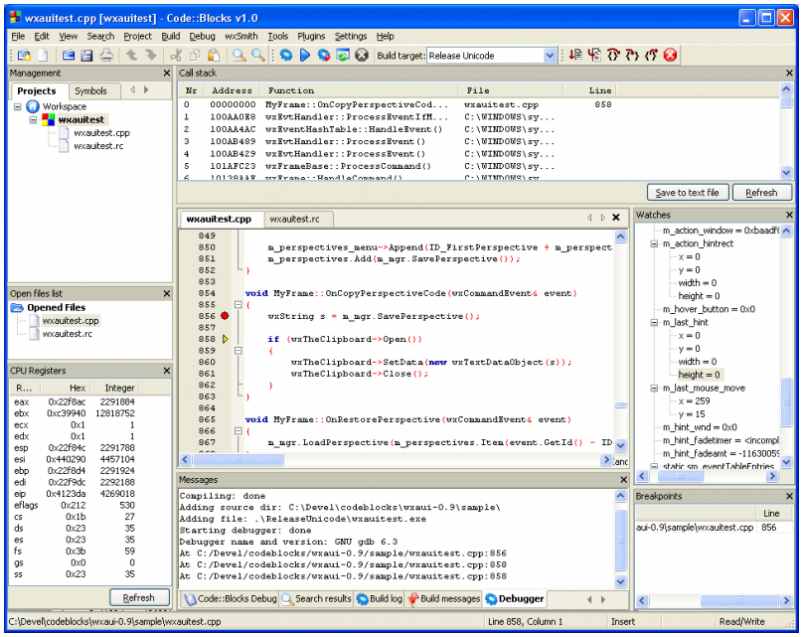


Figura 2.2: Captura de Code::Blocks

- Xcode: se trata de un entorno de desarrollo integrado para macOS propietario de Apple destinado al desarrollo de todo tipo de aplicaciones orientadas a ser usadas en sus dispositivos [21]. Esto hace de Xcode un programa muy complejo y con una grandísima cantidad de funciones que lo hacen poco adecuado para nuevos usuarios. La Tabla 2.3 resume las características de este entorno. Una captura del mismo puede observarse en la Figura 2.3.

Tabla 2.3: Características de Xcode

Plataforma	Lenguajes	Compiladores
Mac	C C++ Swift Objective-C Objective-C++ Java AppleScript	GCC

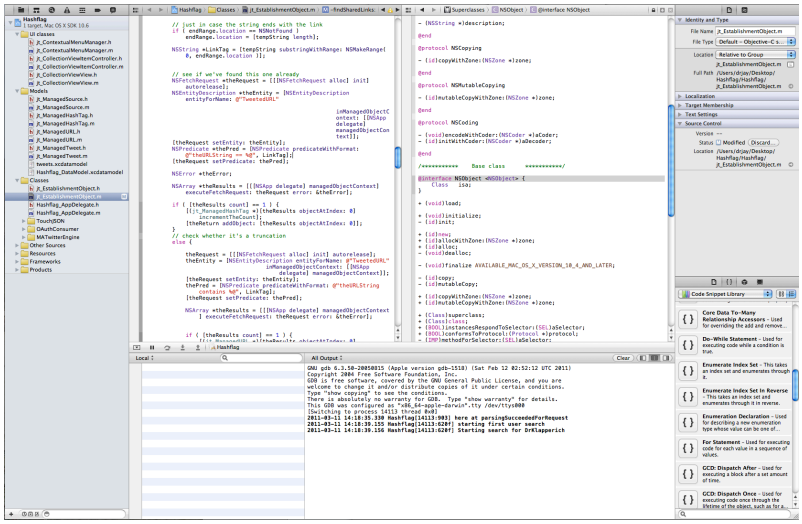


Figura 2.3: Captura de Xcode

Herramientas en la nube

- Codeboard: es un entorno de desarrollo integrado en la web orientado a la enseñanza de la programación (véase la Figura 2.4), por lo que resulta una herramienta ideal para nuevos usuarios. Soporta muchos lenguajes de programación (ver Tabla 2.4). Uno de los problemas que presenta es que no es capaz de ejecutar el código de forma nativa como el resto de aplicaciones de escritorio que hemos visto, por lo que en ocasiones puede dar una sensación de lentitud en los programas, dependiendo de la red y de la carga global de sus servidores.

Tabla 2.4: Características de Codeboard

Plataforma	Lenguajes	Compiladores
Web	C C++ Eiffel Haskell Java Python	En servidor propio

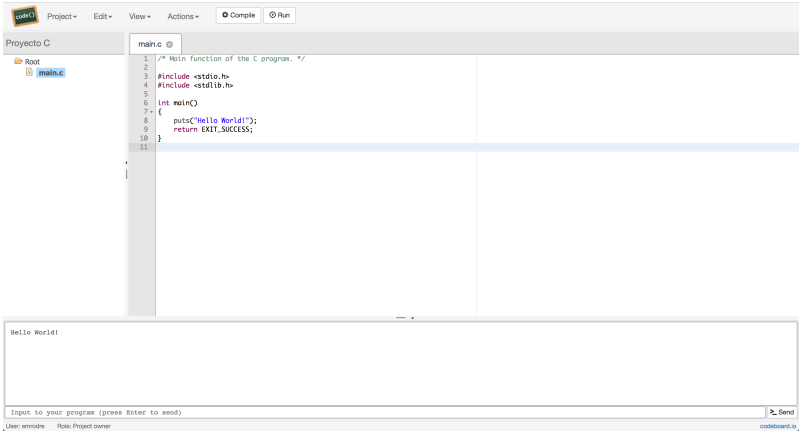


Figura 2.4: Captura de Codeboard

Estado del arte de la tecnología requerida

Una vez analizados los distintos tipos de aplicaciones y vistos los defectos de cada una de ellas individualmente, queda claro que ninguna de las herramientas cumple con nuestros requisitos mínimos:

- Aplicación web, accesible desde cualquier sistema
- No necesidad de instalar software adicional
- Ejecución del código en el lado cliente

Esto nos lleva investigar sobre las tecnologías disponibles actualmente para permitir la ejecución de los programas del usuario en el lado cliente (en el propio navegador). En la actualidad existen tres alternativas para conseguirlo:

Figura 2.5: Logo de Emscripten

- **Emscripten:** [23] es un compilador que convierte bitcode LLVM [11] (que puede ser generado a partir de código C o C++ utilizando los compiladores `llvm-gcc` o `clang`) a JavaScript. De esta forma el código resultante puede ser ejecutado de forma nativa en la mayoría de navegadores actuales, sin embargo, no se realiza una ejecución puramente nativa del código C en el navegador. Sin embargo, presenta algunos problemas para realizar la gestión de entrada/salida de los programas ya que no permite a estos lanzar eventos de lectura y escritura sino que se tiene que estar consultando constantemente para ver si algo ha cambiado y, en particular, impide emular de forma correcta un entorno de tipo terminal como el utilizado habitualmente en los cursos de introducción a la programación. Debido a que en este proyecto se pretende implementar una terminal en el navegador, se ha descartado esta tecnología.

Figura 2.6: Logo de WebAssembly

- **WebAssembly:** [20] es un formato de código binario portable (bytecode) que permite la ejecución de este en el navegador. Se trata de un lenguaje de bajo nivel al que se puede compilar desde lenguajes como por ejemplo C y C++. Esto hace de WebAssembly una tecnología ideal para realizar este proyecto, aunque desgraciada-

mente en la actualidad sigue en fase de desarrollo por lo que se ha descartado y se sugerirá como un posible trabajo futuro al final de este documento.

Figura 2.7: Logo de Chrome Native Client

- **Native Client:** es un *sandbox* integrado en el navegador Google Chrome que permite la ejecución de código C o C++ de forma nativa en el navegador, independientemente del sistema operativo del usuario, de forma segura [22]. Permite acceso a funciones de bajo nivel como la aceleración 3D, el uso del *local storage*, captura del ratón... etc. Cabe destacar que ha servido como precursor de WebAssembly y está destinado a ser reemplazado por este. Sin embargo, debido a la madurez actual del proyecto, se ha elegido esta tecnología para implementar nuestra aplicación. A pesar de tener como limitación la obligatoriedad de utilizar este navegador en lugar de otros (Firefox, Opera, Safari,...) pensamos que esto no resulta un grave problema puesto que Chrome está disponible prácticamente todas las plataformas. De hecho, es posible ejecutar la herramienta desarrollada en ordenadores con el operativo ChromeOS (ej. Chromebooks y Chromeboxes).

  La tecnología Native Client (NaCl) de Google permite a su navegador Google Chrome pasar de un enfoque tradicional de JavaScript + HTML + CSS a un enfoque NaCl + JavaScript + HTML + CSS (Figura 2.8) ofreciendo la posibilidad de ejecutar código C en el navegador.

  La versión Portable Native Client (PNaCl) [6] hace uso de LLVM para permitir independizar esta tecnología del tipo concreto de procesador, permitiendo utilizar el mismo código tanto en plataformas basadas en x86 como en ARM.

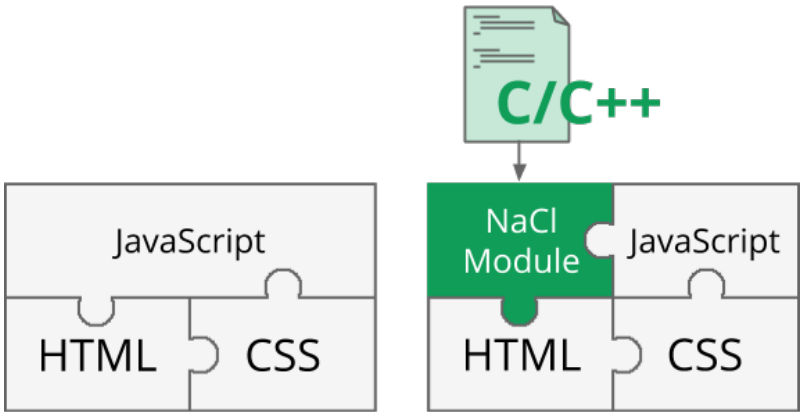


Figura 2.8: Enfoque tecnológico de Chrome con Native Client

Conclusión

Durante el análisis de las anteriores aplicaciones hemos dado una pincelada a los inconvenientes que surgen de estas. Esto nos ha permitido localizar el requerimiento fundamental para cumplir nuestro objetivo: ofrecer un entorno de desarrollo online que permita ejecutar el código del usuario en el propio navegador. Tras un estudio del estado del arte hemos localizado las tecnologías adecuadas para resolverlo, lo cual nos permitirá desarrollar el sistema planteado en los siguientes capítulos dando así solución a los problemas vistos.

CAPÍTULO 3

Diseño

Introducción

Como hemos visto en el capítulo anterior, la ejecución de código C de forma nativa en el navegador requiere un cambio de paradigma.

Actualmente las aplicaciones como Codeboard realizan la ejecución siguiendo el flujo ilustrado en la Figura 3.1.

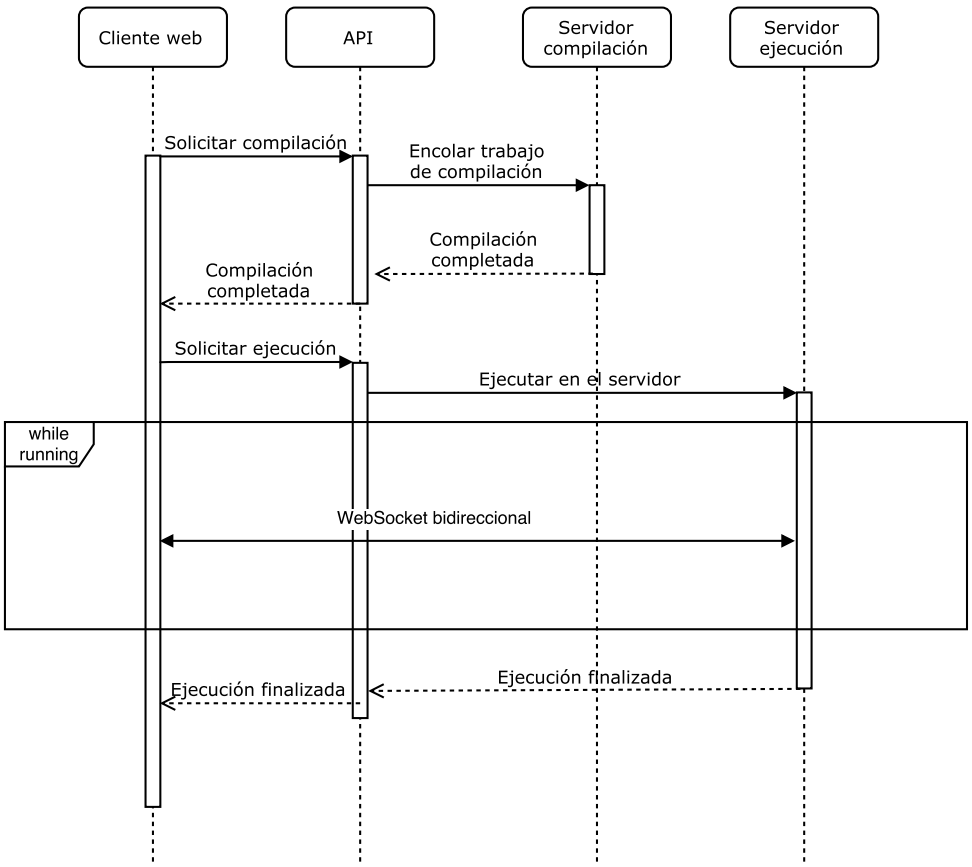


Figura 3.1: Diagrama de flujo de Codeboard

1. Cuando el usuario pulsa sobre “compilar”, la aplicación web envía a la API una solicitud de compilación.

2. La API mete el trabajo en la cola de compilación y, en cuanto el servidor de compilación ha realizado y guardado el trabajo, la API responde al cliente con el id de esta.
3. Una vez terminada la compilación, el cliente ya puede solicitar la ejecución. Al hacer clic sobre el botón "ejecutar" el cliente manda una petición a la API que iniciará la ejecución en un servidor propio.
4. Debido a que la ejecución no se realiza de forma nativa sino en un servidor, mientras el programa se esté ejecutando será necesario mantener una comunicación bidireccional mediante WebSocket [17] entre el cliente web y el servidor que ejecuta el programa para sincronizar la entrada/salida de este.
5. En cuanto la ejecución ha finalizado se cierra el WebSocket y el cliente sabe que la ejecución ha finalizado.

En nuestra aplicación prescindiremos del servidor de ejecución al realizar la misma de forma nativa en el navegador. Esto simplifica notablemente el diagrama de flujo ya que no se ha de mantener una comunicación bidireccional entre el servidor y el cliente quedando de la manera mostrada en la Figura 3.2.

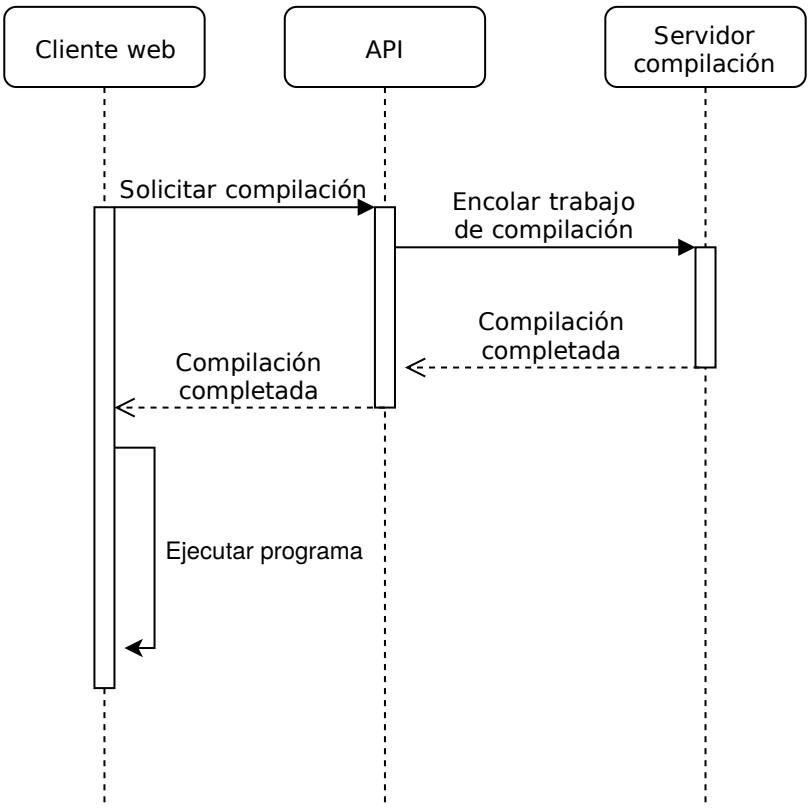


Figura 3.2: Diagrama de flujo del proyecto

1. Cuando el usuario pulsa sobre "ejecutar", la aplicación web envía a la API una solicitud de compilación.
2. La API mete el trabajo en la cola de compilación y en cuanto el servidor de compilación ha realizado y guardado el trabajo la API responde al cliente enviando el binario a ejecutar.
3. Una vez terminada la compilación y recibido el binario el cliente ya puede iniciar la ejecución de forma nativa gracias a la tecnología Native Client.

Stack

Debido a los requerimientos de escalabilidad y alta disponibilidad de este tipo de aplicaciones se ha optado por seguir una arquitectura de microservicios (Figura 3.3).

La arquitectura de microservicios[1] [7] es un patrón de desarrollo de software que consiste en dividir una aplicación en servicios pequeños, que se ejecutan en su propio proceso, se comunican entre sí y cada uno se encarga de implementar una parte de la funcionalidad completa de la aplicación.

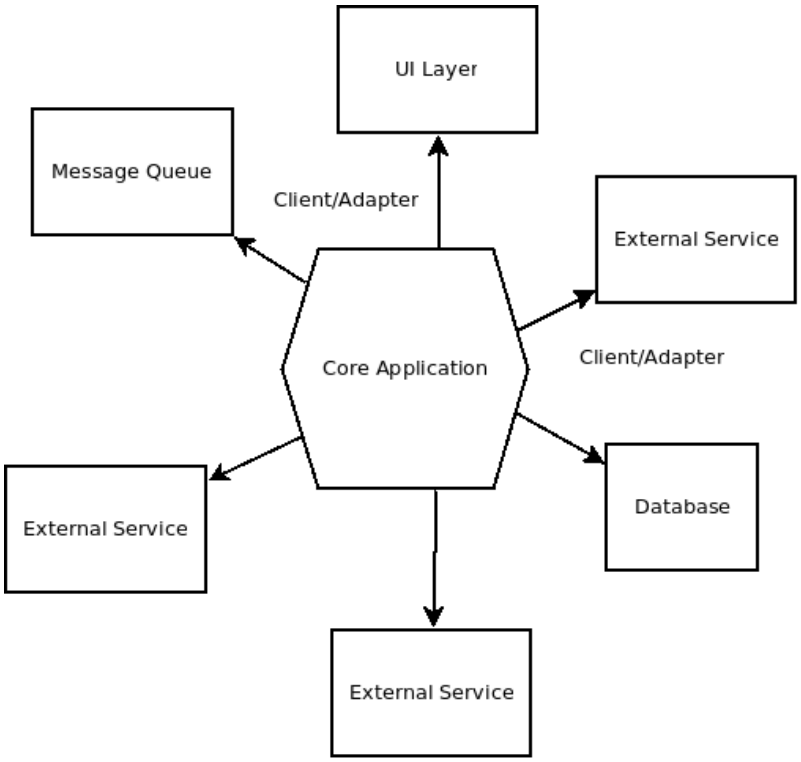


Figura 3.3: Arquitectura de microservicios

Los servicios en los que se ha dividido la aplicación son tres:

- La API: encargada de gestionar la persistencia de la aplicación, la cola de trabajos y los eventos en tiempo real.
- El worker: encargado de procesar las compilaciones y guardar los binarios en la base de datos.
- La aplicación web: que provee la funcionalidad de la aplicación al usuario y se nutre de la API.

La API

Este servicio se considera el núcleo de la aplicación y ofrece tres servicios principales descritos a continuación.

[1] https://en.wikipedia.org/wiki/Microservices

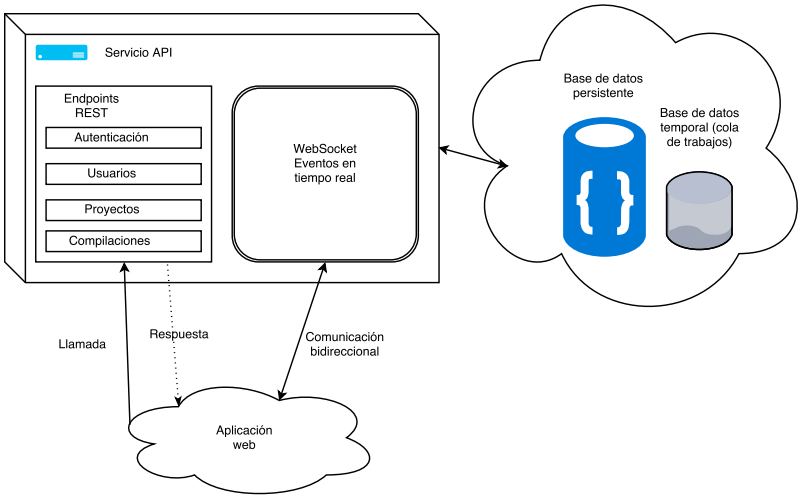


Figura 3.4: Diagrama de la API

En primer lugar ofrece una serie de *endpoints* o rutas REST[2] para interactuar con los datos y la capa de persistencia de la aplicación:

- **Usuarios:** se podrán crear, modificar y eliminar usuarios que podrán crear sus proyectos y persistirlos en la aplicación.
- **Proyectos:** cada usuario podrá tener varios proyectos, formados por una serie de ficheros de código que podrán ser compilados y ejecutados.
- **Compilaciones:** almacena los ficheros binarios para cada fichero de cada proyecto destinados a ser ejecutados por el cliente o aplicación web.

Se ofrecerá también un *endpoint* de autenticación que permita registrar usuarios y hacer login, obteniendo así un token de seguridad que tendrá que ser enviado en cada petición al servicio API para ejecutar métodos para los que es necesario haber iniciado sesión, como el crear y guardar proyectos.

En segundo lugar ofrecerá la capacidad de establecer una conexión bidireccional vía WebSocket que permitirá el envío y recepción de eventos en tiempo real entre la API y el cliente web. Esto permitirá lanzar compilaciones, ver el estado de estas en tiempo real y obtener el binario de forma inmediata en cuanto termine la compilación.

Por último esta API implementará un servicio de cola de trabajos que permitirá almacenar los ficheros a compilar en una base de datos temporal que permitirá a los procesos trabajadores o worker ser escalables para adaptar la aplicación a distintos niveles de carga.

El worker

El worker es la parte de la aplicación encargada de compilar los fragmentos de código escritos por los usuarios y generar ficheros binarios que puedan ser ejecutados por el cliente web.

Este servicio es completamente escalable y pueden desplegarse tantos procesos como se desee.

[2] https://en.wikipedia.org/wiki/Representational_state_transfer

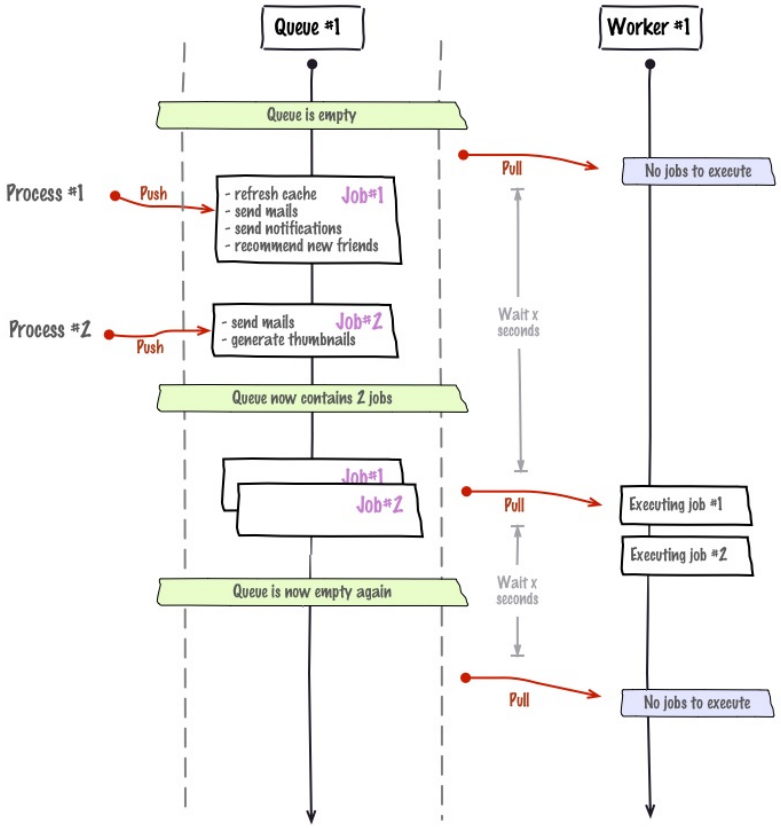


Figura 3.5: Ejemplo de worker

El funcionamiento es el siguiente:

1. El worker monitoriza la cola de trabajos implementada por la API.
2. Si el worker está libre coge un trabajo de la cola, lo bloquea para que otro worker no realice el mismo trabajo y lo compila.
3. Al terminal el trabajo se guarda el resultado de este, es decir, el fichero binario en la base de datos y guarda el trabajo como finalizado con el id del build almacenado en base de datos.
4. El worker vuelve a estado libre y sigue monitorizando la cola de trabajos a la espera de nuevas compilaciones.

La aplicación web

La aplicación web consiste en una interfaz que proporciona una funcionalidad básica para interactuar con el servicio de compilación y ejecución de código.

Permitirá a los usuarios escribir código C, mandar la compilación al servidor y visualizar el resultado de la ejecución en un terminal integrado en el portal.

Para realizar estas tareas el cliente web utilizará el servicio REST y WebSocket implementado por la API y para ejecutar el código en el navegador, le aspecto principal de este cliente, utilizará la librería de Native Client de Chrome.

Cabe destacar que obviamente este cliente se verá limitado a ser usado en el navegador de Google. Como hemos mencionado anteriormente, creemos que esto no supone un gran problema.

CAPÍTULO 4

Implementación

En este capítulo analizaremos a nivel técnico cómo se ha implementado cada una de las partes que componen este proyecto, dando detalles sobre las distintas tecnologías utilizadas en los desarrollos y el por qué de las mismas. Además se mostrarán también en las secciones correspondientes la estructura y los resultados de estos desarrollos en forma de capturas de pantalla de la aplicación resultante.

Stack

Para hacer realidad este proyecto se ha utilizado una gran variedad de tecnologías que permiten el desarrollo y la integración entre las distintas partes de la aplicación de forma sencilla. Algunas de estas se han utilizado de forma general a lo largo de las distintas partes del proyecto y se presentan a continuación:

Figura 4.1: Logo de Node.js

- **Node.js** es un entorno de ejecución multiplataforma que permite la ejecución de código JavaScript en el servidor [19] *NodeJS* [15]]. Está basado en el motor V8 de Google[1] y fue creado con la intención de ser útil en la creación de programas con alto uso de I/O altamente escalables por los que resulta ideal para el desarrollo de aplicaciones web y APIs.

  Los programas escritos en Node.js se ejecutan en un único hilo de ejecución pero, debido a su naturaleza asíncrona, es posible ejecutar concurrentemente múltiples tareas sin incurrir en costos asociados al cambio de contexto.

[1] https://en.wikipedia.org/wiki/Chrome_V8

Además, Node.js dispone de una enorme comunidad que provee a este de multitud de paquetes disponibles de forma sencilla mediante su gestor de paquetes Node Package Manager (NPM).

Para el desarrollo de este proyecto se ha elegido Node.js debido a que nos permite, utilizando una base de datos como MongoDB que veremos a continuación, un desarrollo homogéneo entre el código del cliente y el servidor al realizar todo el proyecto en un mismo lenguaje, JavaScript.

Figura 4.2: Logo de Express

- **Express** es un framework web para Node.js. Está diseñado para construir aplicaciones web y APIs y, debido a su popularidad, se ha convertido en el standard *de facto* en Node.js.

Figura 4.3: Logo de Socket.IO

- **Socket.IO** es una librería JavaScript que proporciona funciones para realizar aplicaciones en tiempo real utilizando el protocolo WebSocket, que permite la comunicación bidireccional entre el cliente web y el servidor. Por esto, resulta ideal para comunicar nuestro cliente web y la API.

Figura 4.4: Logo de MongoDB

- **MongoDB** es un sistema de base de datos NoSQL [**chodorow2013mongodb**], es decir, no relacional y que no estructura los datos en tablas como estamos acostumbrados a ver en las bases da datos tradicionales SQL. MongoDB utiliza documentos JSON (JavaScript Object Notation) que permiten tener un esquema dinámico para guardar documentos, con lo que aporta gran flexibilidad a la hora de trabajar y realizad modificaciones sobre el conjunto de datos existente.

  Además de proveer de una gran cantidad de características, MongoDB es una base de datos lista para su uso en producción orientada a mantener grandes volúmenes de datos.

Para este proyecto se ha elegido esta base de datos debido a la gran flexibilidad y facilidad de uso que provee junto a Node.js ya que MongoDB permite realizar consultas utilizando JavaScript, por lo que podemos desarrollar todo el proyecto en este lenguaje.

Figura 4.5: Logo de Mongoose

- **Mongoose** es una librería para Node.js que permite gestionar fácilmente una base de datos MongoDB mediante el uso de modelos, automatizando la verificación y validación de los documentos y proveyendo de los métodos de consulta necesarios para nuestra aplicación [13].

Figura 4.6: Logo de Redis

- **Redis** es un motor de base de datos clave/valor en memoria [14], lo que lo hace increíblemente rápido y además es compatible con JavaScript y Node.js por lo que en este proyecto resulta ideal para implementar una cola de trabajos que funcione de forma rápida y ágil.
- **Kue** es una cola de trabajos para Node.js que hace uso de la base de datos Redis y permite encolar trabajos, controlar sus prioridades e implementar un número ilimitado de workers que puedan escuchar los eventos de esta, realizar los trabajos y mandar resultados de vuelta [10].

Figura 4.7: Logo de Chrome Native Client

- **Native Client SDK** es el eje central en torno al cual gira todo este proyecto ya que nos aporta el conjunto de librerías o bibliotecas necesarias para implementar el servidor de compilación de código C que generará los binarios que serán ejecutados en el navegador Google Chrome del usuario de la aplicación.

  Este paquete contiene las toolchains necesarias para compilar para Native Client (NaCl) y Portable Native Client (PNaCl) siendo el segundo el que utilizaremos en este proyecto debido a su capacidad de generar, como dice su nombre, código portable, es

decir, que pueda ejecutarse en la máquina de cualquier usuario, independientemente de su arquitectura.

Figura 4.8: Logo de AngularJS

- **AngularJS** es un framework JavaScript creado y mantenido por Google que permite crear aplicaciones web de una sola página que proveen del rendimiento de aplicaciones de escritorio al evitar tiempos de espera entre cambio de páginas.

  Además permite la organización de la aplicación utilizando el modelo modelo vista controlador (MVC) simplificando el desarrollo y las pruebas.

  Se ha elegido para el desarrollo de este proyecto debido al rendimiento que ofrece y a su facilidad de integración con servicios web o APIs.

La API

El desarrollo de la API se ha realizado utilizando Node.js como entorno de ejecución, express como framework web para proveer de los endpoints REST, MongoDB como base de datos, Mongoose para implementar los modelos de esta, Kue como cola de trabajos, Redis como base de datos para la cola de trabajos y Socket.IO para sincronizar los eventos en tiempo real que emite la cola de trabajos con el cliente web.

A continuación se muestra un fragmento del entrypoint de la aplicación, el fichero **src/app.js**:

```
// Inicializar cola de trabajos
const queue = kue.createQueue({
  redis: {
    host: config.redis.host,
    port: config.redis.port,
  },
});
kue.app.listen(config.kue.port);

// Inicializar eventos en tiempo real
io.on('connection', (socket) => {
  socketio(socket, queue);
});

// Inicializar endpoints REST
server.listen(config.port, () => {
  console.log(`App listening on port ${config.port}`);
});
```

En este fichero se han inicializado las partes principales de la API, los endpoints REST y los eventos REALTIME. A continuación detallaremos la implementación de estos por separado.

REST

Se han implementado una serie de rutas que proveen a la aplicación de las funciones necesarias para manejar los datos de esta. Además se ha implementado una capa de seguridad para evitar el acceso a las rutas seguras por parte de usuarios no autorizados.

Tabla 4.1: (Endpoints REST de la API)

Endpoint	Funciones	Segura
/auth/login	Iniciar sesión y obtener token de seguridad	No
/auth/signup	Crear nuevo usuario y obtener token de seguridad	No
/users/:id?	Crear, modificar y consultar datos de usuario	Sí
/projects/:id?	Crear, modificar y consultar datos de proyecto	Sí
/builds/:id?	Crear, modificar y consultar compilación	No
/builds/:id/:file	Obtener fichero binario de una compilación	No

Para realizar la implementación de las rutas REST se ha seguido el flujo:

Router ⟷ Controlador ⟷ BLL (Capa de negocio)

A continuación se muestra la implementación para la ruta de proyectos:

- Router: se encarga de capturar las peticiones http recibidas para los distintos métodos y redirigir la petición hacia la función o método adecuado en el controlador, que se encargará de realizar las acciones necesarias y devolver una respuesta.

  router.js

  ```
  router.use('/projects', jwt.verify, projectsRouter);
  ```

  projects.router.js

  ```
  router.post('/', controller.create);
  router.get('/:id?', controller.get);
  router.put('/:id', controller.update);
  router.delete('/:id', controller.delete);
  ```

- Controlador: provee de las funciones necesarias para realizar las acciones solicitadas a la API, obtiene los datos de la petición, realiza las acciones necesaria a través de la BLL y una vez terminada la acción, envía una respuesta al cliente.

  projects.controller.js

  ```
  exports.create = (req, res, next) => {
    co(function* () {
      const project = req.body;
      const createdProject = yield bll.create(project);
      res.send(createdProject);
    }).catch(next);
  };

  exports.get = (req, res, next) => {
    co(function* () {
      const id = req.params.id;
      const userId = req.query.userId;
  ```

```
    let projectOrProjects;
    if (id) {
      projectOrProjects = yield bll.getById(id);
    } else if (userId) {
      projectOrProjects = yield bll.getByUserId(userId);
    } else {
      projectOrProjects = yield bll.get();
    }

    res.send(projectOrProjects);
  }).catch(next);
};

exports.update = (req, res, next) => {
  co(function* () {
    const id = req.params.id;
    const project = req.body;
    const updatedProject = yield bll.update(id, project);
    res.send(updatedProject);
  }).catch(next);
};

exports.delete = (req, res, next) => {
  co(function* () {
    const id = req.params.id;
    const deletedProject = yield bll.delete(id);
    res.send(deletedProject);
  }).catch(next);
};
```

- Bll: se encarga de implementar la lógica de negocio y las llamadas a base de datos para conseguir abstraer al controlador de las funciones que son mas específicas del tipo de implementación que de la acción que se está realizando.

  projects.bll.js

```
exports.create = function (project) {
  const createdProject = ProjectModel.create(project);
  return createdProject;
};

exports.get = function () {
  const projects = ProjectModel.find({}).lean().exec();
  return projects;
};

exports.getById = function (id) {
  const project = ProjectModel.findOne({ _id: id }).lean().exec();
  return project;
};

exports.update = function (id, project) {
  const updatedProject = ProjectModel.update({ _id: id }, project);
```

```
  return updatedProject;
};

exports.delete = function (id) {
  const deletedProject = ProjectModel.remove({ _id: id }).lean().exec();
  return deletedProject;
};

exports.getByUserId = function (userId) {
  const projects = ProjectModel.find({ owner: new ObjectId(userId) })
    .lean().exec();
  return projects;
};
```

Como vemos, para los métodos de inserción en base de datos, la Bll hace uso de los métodos que provee el modelo definido utilizando la librería Mongoose. A continuación se muestra el modelo de documento utilizado para almacenar los proyectos en la base de datos MongoDB:

projects.bll.js

```
const projectSchema = new Schema({
  owner: {
    type: Schema.Types.ObjectId,
    ref: 'User',
    required: true,
  },
  users: [
    {
      type: Schema.Types.ObjectId,
      ref: 'User',
    },
  ],
  name: {
    type: String,
    required: true,
  },
  files: [
    {
      name: String,
      content: String,
    },
  ],
});
```

En cuanto a la seguridad de las rutas, si nos fijamos en el fichero **router.js** visto arriba, podemos ver que la ruta de proyectos implementa el middleware `jwt.verify`, con esto conseguimos verificar que al realizar la petición el usuario ha mandado el token de seguridad en la cabecera de la petición de la siguiente manera:

```
Authorization: Bearer EL_TOKEN_AQUÍ
```

Esta medida de seguridad se ha implementado mediante el uso de JSON Web Token (JWT), una tecnología que sigue el estándar RFC 7519.

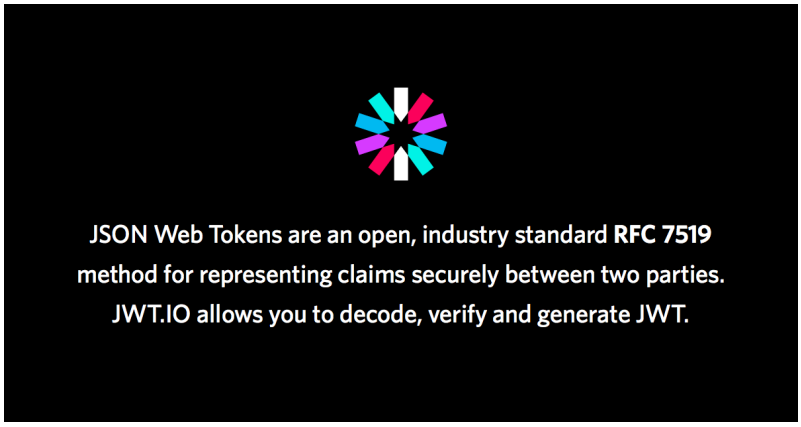


Figura 4.9: JSON Web Token

REALTIME

Para gestionar los eventos en tiempo real entre la cola de trabajos y el cliente se ha utilizado la librería Socket.IO mencionada anteriormente, con ello conseguimos sincronizar los eventos de la cola de trabajos con la API y de esta con el cliente que estará escuchando mediante el cliente de Socket.IO.

socketio/index.js

```
module.exports = function (socket, queue) {
  // Al recibir un evento 'make' se añade un trabajo a la cola de trabajos y se
  // espera a recibir una serie de eventos que devolverá esta
  socket.on('make', (msg) => {
    const job = queue.create('make', msg).save((err) => {
      if (err) {
        throw new Error('Error al crear trabajo');
      }
    });

    // Cuando el trabajo se ha encolado se notifica al cliente
    job.on('enqueue', () => {
      socket.emit('make-enqueue');
    // Cuando la compilación se inicia se notifica al cliente
    }).on('start', () => {
      socket.emit('make-start');
    // Cuando la compilación se completa se notifica al cliente y se envían los
    // datos de esta para que el cliente web pueda descargar y ejecutar el binario
    }).on('complete', (result) => {
      socket.emit('make-complete', result);
    // Cuando la compilación falla avisa al cliente y se manda el resultado de
    // la compilación que contiene el mensaje de error
    }).on('failed', (errorMessage) => {
      socket.emit('make-failed', errorMessage);
    });
  });
};
```

El worker

El worker se ha implementado utilizando Node.js como entorno de ejecución, el cliente de Kue para obtener los trabajos de la cola, el toolkit Native Client SDK para realizar la compilación del código C a ficheros binarios y Mongoose para guardar estos en la base de datos MongoDB.

A continuación se muestra un fragmento del entrypoint de la aplicación, el fichero **app.js**:

```
// Se inicializa la conexión a la base de datos MongoDB
mongoose.connect(`mongodb://${config.mongodb.host}`
  + `:${config.mongodb.port}/${config.mongodb.db}`);

// Se inicializa la conexión a la cola de trabajos
const queue = kue.createQueue({
  redis: {
    host: config.redis.host,
    port: config.redis.port,
  },
});

// Cuando en la cola de trabajos entra un evento del
// tipo 'make' se inicia el trabajo
queue.process('make', jobs.make);
```

En cuanto se lanza el evento 'make' se procede a la ejecución del trabajo que define el fichero **jobs/make.js**:

```
// Generar el makefile que se ejecutará para la compilación
function* createMakefile(destPath, sources) {
  let makeFile = yield fs.readFileAsync(path.join(__dirname,
    '../templates/Makefile'));
  makeFile = makeFile.toString();
  makeFile = makeFile.replace('{{NACL_SDK_ROOT}}',
    path.join(config.nacl_sdk.path, config.nacl_sdk.pepper));
  makeFile = makeFile.replace('{{SOURCES}}', sources);
  yield fs.writeFileAsync(path.join(destPath, 'Makefile'), makeFile);
}

// Guardar el fichero binario en la base de datos
function* save(sourcePath, make) {
  const nmf = yield fs.readFileAsync(path.join(sourcePath,
    'pnacl/Release/bin.nmf'));
  const pexe = yield fs.readFileAsync(path.join(sourcePath,
    'pnacl/Release/bin.pexe'));
  const build = new Build({
    nmf,
    pexe,
    make,
  });
  return yield build.save();
}

// Lanzar la compilación y devolver el resultado de esta
```

```
module.exports = (job, done) => {
  co(function* () {
    const dir = yield tmp.dir({ unsafeCleanup: true });
    yield createMakefile(dir.path, job.data.fileName);
    yield fs.writeFileAsync(path.join(dir.path, job.data.fileName),
      job.data.fileContent);
    const makeResult = yield exec('make', { cwd: dir.path,
      timeout: config.jobs.make.timeout });

    const result = {
      make: {
        stdout: makeResult.stdout,
        stderr: makeResult.stderr,
      },
    };
    const saveResult = yield save(dir.path, result.make);
    result.buildId = saveResult._id;
    done(null, result);
    dir.cleanup();
  }).catch((err) => {
    done(err);
  });
};
```

Antes de iniciar la compilación se generará el fichero `Makefile` correspondiente en base al siguiente esquema:

```
VALID_TOOLCHAINS := pnacl glibc clang-newlib linux

NACL_SDK_ROOT = {{NACL_SDK_ROOT}}

TARGET = bin

include $(NACL_SDK_ROOT)/tools/common.mk

DEPS = ppapi_simple nacl_io
LIBS = ppapi_simple nacl_io ppapi pthread

CFLAGS = -Wall
SOURCES = {{SOURCES}}

$(foreach dep,$(DEPS),$(eval $(call DEPEND_RULE,$(dep))))
$(foreach src,$(SOURCES),$(eval $(call COMPILE_RULE,$(src),$(CFLAGS))))

ifneq (,$(or $(findstring pnacl,$(TOOLCHAIN)),$(findstring Release,$(CONFIG))))
$(eval $(call LINK_RULE,$(TARGET)_unstripped,$(SOURCES),$(LIBS),$(DEPS)))
$(eval $(call STRIP_RULE,$(TARGET),$(TARGET)_unstripped))
else
$(eval $(call LINK_RULE,$(TARGET),$(SOURCES),$(LIBS),$(DEPS)))
endif

$(eval $(call NMF_RULE,$(TARGET),))
```

Una vez generado el `Makefile` se ejecuta la compilación y una vez terminada, para guardar en la base de datos los ficheros binarios compilados se ha definido el siguiente modelo de Mongoose:

models/build.js:

```
const buildSchema = new Schema({
  nmf: Buffer,
  pexe: Buffer,
  make: {
    stdout: String,
    stderr: String,
  },
});
```

Como se puede apreciar se guardan en formato binario o buffer los ficheros nmf y pexe necesarios para la ejecución, además de los resultados stdout y stderr de la compilación.

La aplicación web

La aplicación web se ha implementado utilizando el framework de Google AngularJS que permite el desarrollo de aplicaciones single page haciendo uso de la API. Se ha utilizado también el cliente de Socket.IO para realizar la comunicación de eventos en tiempo real con la API.

Al igual que en las partes anteriores, y debido a la extensión de este desarrollo, en este documento se mostrará una pequeña parte de este consistente en el flujo seguido por el cliente desde que el usuario inicia la ejecución de un programa hasta que este empieza a hacerlo.

En primer lugar vemos el entrypoint de la aplicación, que nos lleva al editor:

app.js:

```
app.config(function ($stateProvider, $urlRouterProvider,
  jwtOptionsProvider, $httpProvider) {
  // Define app states
  var states = [
    {
      name: 'login',
      url: '/login',
      templateUrl: 'views/login.html'
    },
    {
      name: 'editor',
      url: '/editor?project',
      templateUrl: 'views/editor.html'
    },
    {
      name: 'projects',
      url: '/projects',
      templateUrl: 'views/projects.html'
    },
    {
      name: 'newproject',
```

```
      url: '/projects/new',
      templateUrl: 'views/new-project.html'
    }
  ];

  // Load states
  for (var i = 0, length = states.length; i < length; i++) {
    var state = states[i];
    $stateProvider.state(state);
  }

  // Redirect to '/' if not in state
  $urlRouterProvider.otherwise('/editor');

  // JWT
  jwtOptionsProvider.config({
    tokenGetter: ['loginService', function (loginService) {
      return loginService.getToken();
    }],
    whiteListedDomains: ['localhost']
  });

  $httpProvider.interceptors.push('jwtInterceptor');
});
```

Al estar en la pantalla del editor AngularJS cargará el controlador de este, que entre otras muchas funciones provee de la función que manda el código escrito por el usuario a compilar vía Socket.IO.

editor/editor.controller.js:

```
$scope.run = function (index) {
  if ($scope.disableRun) {
    return;
  }

  var file = $scope.openFiles[index];

  var msg = {
    title: file.name,
    fileName: file.name,
    fileContent: file.content
  };

  socket.emit('make', msg);
};
```

Una vez solicitada la compilación el trabajo se inserta en la cola de trabajos implementada por la API, posteriormente el worker realizará el trabajo de compilación y guardara los resultados, devolviendo el resultado del trabajo que será capturado por el la siguiente función del controlador que se encargará de lanzar la ejecución del fichero compilado utilizando el módulo Native Client de Chrome.

editor/editor.controller.js:

```
socket.on('make-complete', function (result) {
  common.createNaClModule('bin', 'pnacl', config.api + '/builds/'
  + result.buildId, 0, 0, {
    PS_STDOUT: 'dev/tty',
    PS_STDIN: 'dev/tty',
    PS_TTY_PREFIX: 'tty:',
    PS_EXIT_MESSAGE: 'exit'
  });
  update('Compilación terminada');
});
```

A continuación se muestra una captura de pantalla de la aplicación ejecutando un código de ejemplo en el proyecto Demo:

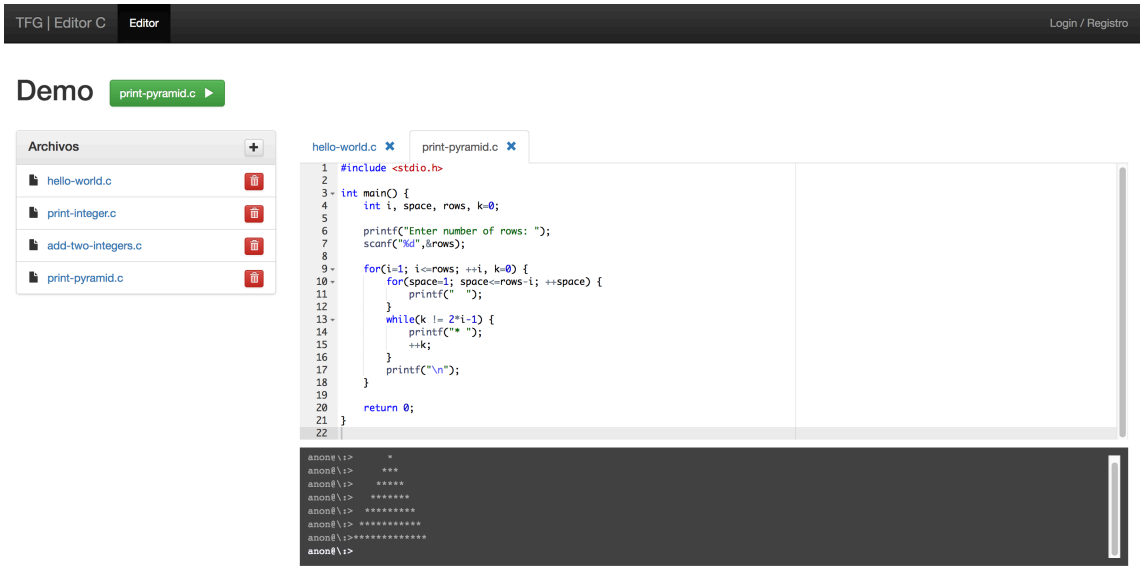


Figura 4.10: Captura del editor

CAPÍTULO 5
Despliegue

Introducción

Debido a la arquitectura orientada a microservicios que se ha implementado, la aplicación consta de varios módulos que deben ser capaces de mantener una comunicación entre ellos y con las bases de datos, además de ser fácilmente escalables y distribuidos.

- MongoDB: se trata de la base de datos no-SQL mencionada en el capítulo anterior y que utilizaremos para la persistencia de datos [4].
- Redis: es la base de datos en memoria especializada en tablas clave/valor también mencionada en el capítulo anterior. La utilizaremos para el almacenamiento temporal de trabajos [14].
- La API: para la gestión de la aplicación.
- El/los worker/s: para la compilación de trabajos.
- El cliente: para el acceso por parte del cliente a interactuar con la aplicación.

En concreto, deben poder producirse los siguientes casos para un correcto funcionamiento del conjunto:

- La API debe poder comunicarse tanto con la base de datos MongoDB para almacenar, consultar y modificar datos como con la base de datos REDIS para sincronizar la cola de trabajos. Además esta debe exponer al cliente tanto sus endpoints o rutas REST vía http como a través del protocolo WebSocket para poder enviar y recibir eventos en tiempo real ya que será el cliente el que conecte con esta y también expondrá la interfaz de usuario de Kue, que nos permitirá gestionar la cola de trabajos de forma visual.
- El worker debe ser capaz de comunicar con las bases de datos MongoDB y Redis, utilizando la primera para almacenar los ficheros binarios generados por este y la segunda para mantenerse sincronizado con la cola de trabajos. Es importante destacar el requisito de escalabilidad que presenta este servicio, ya que deben poder desplegarse con facilidad múltiples instancias de este para adaptarse a la carga de la aplicación en cualquier situación.
- El cliente web debe estar accesible mediante el protocolo http y debe poder tener acceso a la API, con esto bastará para poder hacer uso de todas las funcionalidades de la aplicación.

Debido a todos estos requisitos se ha optado por realizar el despliegue a través de una herramienta externa que permite el despliegue de estos servicios utilizando un sistema de virtualización conocido como "contenedores". Esta tecnología permite encapsular los servicios en paquetes independientes que contienen todas las dependencias necesarias y

pueden ser ejecutados en cualquier sistema con un solo comando. En concreto, para la implementación de este método se ha escogido la plataforma Docker[1] [12] debido a su madurez y buena aceptación por la comunidad, lo que lo hace ideal para desplegar en entornos de producción de forma sencilla y segura.

Docker

Figura 5.1: El logo de Docker

Se trata de una herramienta de virtualización a nivel de sistema operativo "*Operating-system-level virtualization*" basada en contenedores.

La virtualización a nivel de sistema operativo[2] es un método de virtualización que permite, ejecutar múltiples sistemas operativos de forma nativa y aislada compartiendo únicamente entre ellos el kernel del sistema host. Esto resulta en un rendimiento de las aplicaciones muy cercano al nativo (llegando incluso a ser difícil detectar cuando una aplicación se esta ejecutando en un contenedor), a diferencia de otros métodos de virtualización en los que este se ve afectado.

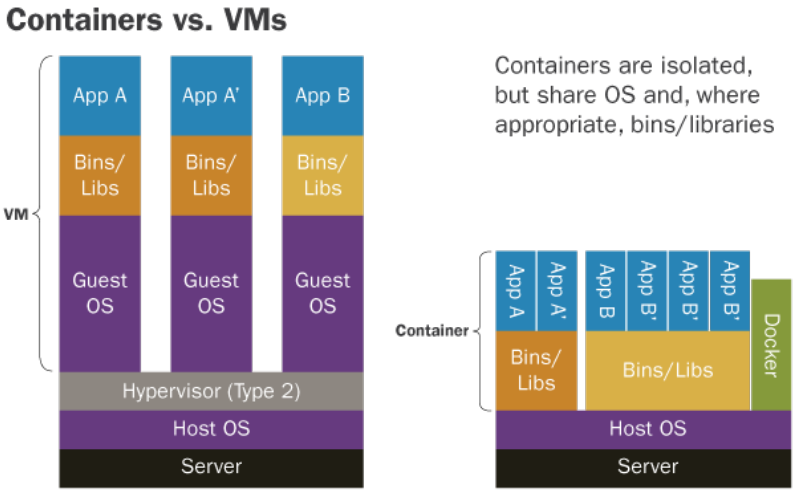


Figura 5.2: Contenedores vs Máquinas virtuales

Separar aplicaciones en contenedores que solamente comparten el kernel nos permite aprovechar una serie de ventajas que se traducen en una mayor seguridad y facilidad de uso:

- Se limitan o eliminan las comunicaciones no deseadas entre los distintos programas.
- Debido a que los contenedores son interoperables se elimina la dependencia de hardware, pudiendo ejecutar estos en cualquier sistema compatible con Docker.
- Se permite una mejor gestión de recursos ya que se puede limitar el contenedor en sí, no teniendo que limitar todos los programas que este contiene.

[1] https://www.docker.com
[2] https://en.wikipedia.org/wiki/Operating-system-level_virtualization

El soporte de Linux para contenedores consigue aislar casi totalmente las aplicaciones que se ejecutan en estos del sistema operativo host. De esta forma cada contenedor dispone de su propio sistema operativo, su propio árbol de procesos, redes, usuarios y sistema de archivos, compartiendo únicamente las funciones del kernel para gestión y limitación de recursos (CPU, memoria, I/O y red).

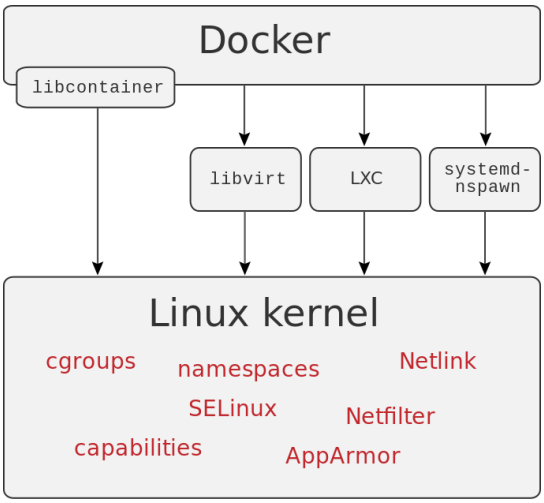


Figura 5.3: Acceso al kernel en Docker

Para comenzar a trabajar con Docker debemos entender primero su flujo de funcionamiento que consiste en tres componentes principales:

- Dockerfile: es un documento de texto que contiene todos los comandos que un usuario debería ejecutar en la terminal para preparar el entorno de ejecución del programa, es decir, instalar las dependencias y la aplicación.

Ejemplo:

```
# Usar la imgen de node como base (node ya instalado)
FROM node:boron

# Crear directorio de la aplicación
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# Instalar dependencias
COPY package.json /usr/src/app/
RUN npm install

# Copiar fuentes
COPY . /usr/src/app

# Exponer el puerto 8080
EXPOSE 8080

# Iniciar la aplicación
CMD [ "npm", "start" ]
```

Utilizando este documento, Docker es capaz de generar una imagen con todo lo necesario para ejecutar la aplicación en un contenedor.

- Imágenes: es un fichero inmutable que contiene todos los datos necesarios para desplegar un contenedor con la aplicación, es decir, un *snapshot* de esta. Las imágenes de Docker están formadas por capas, siendo cada capa uno de los comandos del Dockerfile de manera que, al modificar este y regenerar la imagen, sólo se generarán las capas afectadas.
- Contenedores: un contenedor es una instancia de una imagen, es decir, una copia de esta que sí puede ser ejecutada. En Docker cada contenedor representa un proceso de aplicación y cabe destacar que, por cada imagen, podemos desplegar múltiples contenedores, por lo que es ideal para la escalabilidad.

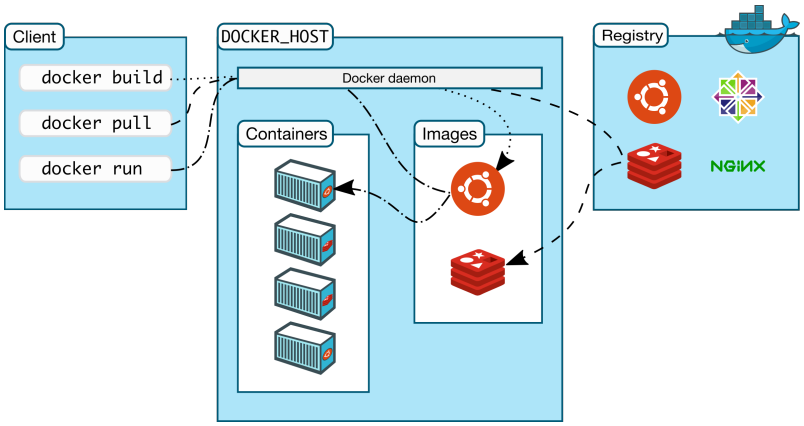


Figura 5.4: Arquitectura de despliegue con Docker

A continuación se muestran los Dockerfile (con anotaciones sobre qué es lo que hacen) utilizados en el proyecto para generar las imágenes de los distintos servicios:

La API

```
# Partimos de la imagen de Node Boron que contiene Debian Jessie y Node 6
FROM node:boron

# Creamos el directorio en el que instalaremos la app
RUN mkdir -p /usr/src/app

# Generamos las claves publicas y privadas que servirán para la autenticación
# vía token
RUN mkdir -p /usr/src/app/keys
WORKDIR /usr/src/app/keys
RUN ssh-keygen -t rsa -f key
RUN ssh-keygen -e -m pem -f key.pub > key.pub.pem

# Instalamos las dependencias y copiamos el código de la aplicación
WORKDIR /usr/src/app
COPY package.json /usr/src/app
RUN npm install
COPY . /usr/src/app

# Abrimos los puertos 3030 y 3031 para poder acceder a la API y a la interfaz
# de la cola de trabajos Kue
```

```
EXPOSE 3030
EXPOSE 3031

# Lanzamos la ejecución de la app como proceso principal del contenedor
CMD ["npm", "start"]
```

El worker

```
# Partimos de la imagen de Node Boron que contiene Debian Jessie y Node 6
FROM node:boron

# Instalamos las dependencias de las herramientas de desarrollo de
# Chrome Native Client
RUN apt-get update
RUN apt-get install -y python make wget unzip libc6-i386 bzip2

# Creamos el directorio en el que instalaremos la app y el SDK de Native Client
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# Instalamos y actualizamos el SDK y la API de Native Client
RUN wget https://storage.googleapis.com/nativeclient-mirror/nacl/nacl_sdk/nacl_sdk.zip
RUN unzip nacl_sdk.zip
RUN rm nacl_sdk.zip
WORKDIR /usr/src/app/nacl_sdk
RUN ./naclsdk update pepper_49

# Instalamos las dependencias y copiamos el código de la aplicación
RUN mkdir -p /usr/src/app/worker
WORKDIR /usr/src/app/worker
COPY package.json /usr/src/app/worker
RUN npm install
COPY . /usr/src/app/worker

# Lanzamos la ejecución de la app como proceso principal del contenedor
CMD ["npm", "run", "production"]
```

La aplicación web

```
# Partimos de la imagen de Node Boron que contiene Debian Jessie y Node 6
FROM node:boron

# Install el servidor web Nginx que servirá la aplicación web estática
# al cliente vía http
RUN apt-get update
RUN apt-get install -y nginx

# Creamos el directorio en el que instalaremos la app
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
```

```
# Hacemos el build de la app
COPY package.json /usr/src/app
RUN npm install
COPY . /usr/src/app
RUN /usr/src/app/node_modules/.bin/gulp

# Copiamos el build de la app listo para ser desplegado a la carpeta
# pública del servidor web
RUN cp -R /usr/src/app/dist/* /var/www/html

# Abrimos el puerto 80 para poder acceder al servidor web vía http con
# el navegador
EXPOSE 80

# Iniciamos el servidor web y con ello la aplicación como proceso principal
# del contenedor
CMD ["nginx", "-g", "daemon off;"]
```

Docker Compose

Compose es una herramienta de Docker que permite desplegar aplicaciones multi-contenedor de forma sencilla utilizando un fichero de configuración (habitualmente en formato YAML [2]) que permite configurar los contenedores y las relaciones entre ellos.

Esta herramienta nos permite realizar las mismas operaciones que podemos realizar de manera individual sobre cualquier contenedor Docker pero de forma atómica sobre todos los contenedores que componen la aplicación.

Podremos de esta forma:

- Generar todas las imágenes de los servicios utilizando el comando `docker-compose build`
- Lanzar todos los contenedores usando `docker-compose start`
- Parar todos los contenedores utilizando `docker-compose stop`
- Ver los logs de estos `docker-compose logs`
- etc.

En este proyecto se ha implementado el siguiente fichero `'docker-compose.yml'` que nos permitirá realizar estas operaciones sobre los contenedores que forman la aplicación de forma sencilla:

```
version: '2'

services:
  mongo:
    image: mongo
    ports:
      - '27017:27017'

  redis:
    image: redis

  api:
```

```
    build:
      context: ./api
    depends_on:
      - mongo
      - redis
    links:
      - mongo
      - redis
    ports:
      - '3030:3030'
      - '3031:3031'

  worker:
    build:
      context: ./worker
    depends_on:
      - mongo
      - redis
    links:
      - mongo
      - redis

  web:
    build:
      context: ./web
    depends_on:
      - api
    ports:
      - '8080:80'
```

Esto nos permite lanzar el stack completo de la aplicación mediante un único comando:

```
$> cd tfg/
$> docker-compose up --scale worker=4 -d
```

Cabe destacar que, como podemos ver en el comando, el parámetro "`--scale worker=X`" indica el número de contenedores worker a desplegar, proporcionando así una escalabilidad muy sencilla y que se puede modificar en cualquier momento a la aplicación.

CAPÍTULO 6

Conclusiones

En este capítulo procedemos a realizar una evaluación del proyecto realizado, analizando lo que la aplicación puede aportar tanto a nivel técnico como a nivel académico. Además comentaremos algunos de los inconvenientes que han ido surgiendo al tratarse de una tecnología diferente a la habitual y los pros y contras de las soluciones que se han adoptado ante estos.

Aportaciones y resultados del trabajo realizado

Técnicas

Este proyecto sienta las bases para la construcción de entornos de desarrollo integrados online capaces de ejecutar código C de forma nativa en el navegador, por lo que a nivel técnico supone un avance respecto a el enfoque tradicional de ejecutarlo en el servidor.

Además se ha implementado una arquitectura de microservicios que da como resultado un proyecto escalable y de fácil mantenimiento. Por lo que no sería complicado ampliar la funcionalidad de este o mejorarla para ofrecer un entorno más completo.

Académicas

En cuanto a las aportaciones académicas, se ha conseguido con éxito el disponer de un entorno sencillo, que no depende de la instalación de ningún software adicional en el que el usuario puede realizar las prácticas en el laboratorio y seguir con ellas en su casa trabajando sobre el mismo entorno, independientemente de su sistema.

Publicación

Como resultado de este trabajo hemos enviado un pequeño artículo en el 25 Congreso Universitario de Innovación Educativa en las Enseñanzas Técnicas[1] (CUIEET) y que este trabajo ha sido aceptado. Adjuntamos una copia del artículo en un anexo de esta memoria.

[1] https://25cuieet.es

Limitaciones

Utilizar NaCL puede ser una limitación al estar únicamente soportado por el navegador Chrome. También es un problema que hay indicios de que va a quedar obsoleto (*deprecated*) con la aparición de WebAssembly. A este respecto, hemos de comentar que actualmente la tecnología WebAssembly no está suficientemente madura para realizar un proyecto como este.

Otra limitación de este proyecto es que han de utilizarse bibliotecas que se puedan compilar con NaCL: esto no supone un problema en asignaturas de introducción a la programación pero sí podría serlo en cursos más avanzados que hagan usos de otras bibliotecas.

Una de las mayores limitaciones, de cara a su utilización en la asignatura "Informática" de algunos grados de la ETSID, es no poder trabajar con ficheros en LocalStorage como si se tratase de un sistema de archivos normal. Poder trabajar con ficheros resulta imprescindible en esta asignatura. Esta limitación se trata brevemente en el siguiente apartado sobre trabajos futuros.

Trabajos futuros

Durante el desarrollo del proyecto han surgido ideas que no han podido ser implementadas, bien sea por la falta actual de tecnologías para llevarlas a cabo o por aumentar demasiado la extensión de este. Algunas de ellas se comentan a continuación:

- Con la aparición de WebAssembly, destinado a sustituir a la tecnología Native Client con la que ha sido implementado este trabajo, se propone como trabajo futuro el realizar una migración a esta tecnología. A fecha de presentación de este proyecto todavía no es posible llevarla a cabo debido a que se trata de una tecnología en fase de desarrollo que no dispone aún de la funcionalidad necesaria.
- Recopilar estadísticas: se plantea como trabajo el recopilar estadísticas de uso de la aplicación, número de compilaciones, compilaciones exitosas, fallidas, etc. Esto es lo que ya hace Codeboard y que tiene mucha utilidad de cara a que los profesores puedan realizar un seguimiento del trabajo de los alumnos (*learning analytics*).
- Permitir poder trabajar con ficheros en LocalStorage como si se tratase de un sistema de archivos normal puesto que en la aplicación implementada no es posible trabajar sobre ficheros, es decir, transparentemente al usuario de la aplicación para que pueda utilizar las librerías estándar de C, para ello se plantea hacer un *wrapper* de estas para conseguirlo.
- La posibilidad de crear una biblioteca que permita a los alumnos dibujar en un canvas para no estar limitados a programas en modo terminal sino que puedan utilizar algún tipo de librería para generar gráficos 2D y mostrarlos en el mismo navegador. Este tipo de mejoras no resultan complejas de implementar.

Bibliografía

[1] https://cssauthor.com/best-online-code-editors-for-developers. accedido: 2017-07-01.

[2] Oren Ben-Kiki, Clark Evans y Brian Ingerson. "YAML Ain't Markup Language (YAML™) Version 1.1". En: *yaml. org, Tech. Rep* (2005).

[3] Estler C. y Nordio M. *Codeboard*. https://codeboard.io. accedido: 2017-06-26.

[4] Kristina Chodorow. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. .ºReilly Media, Inc.", 2013.

[5] *Code::Blocks*. http://www.codeblocks.org. accedido: 2017-06-26.

[6] Alan Donovan y col. "PNaCl: Portable native client executables". En: *Google White Paper* (2010).

[7] Nicola Dragoni y col. "Microservices: yesterday, today, and tomorrow". En: *arXiv preprint arXiv:1606.04036* (2016).

[8] *edX*. https://www.edx.org. accedido: 2017-06-21.

[9] Michael Kölling y col. "The BlueJ system and its pedagogy". En: *Computer Science Education* 13.4 (2003), págs. 249-268.

[10] *Kue*. https://github.com/Automattic/kue. accedido: 2017-06-28.

[11] Chris Lattner y Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". En: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2004, pág. 75.

[12] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". En: *Linux Journal* 2014.239 (2014), pág. 2.

[13] *Mongoose*. http://mongoosejs.com. accedido: 2017-07-01.

[14] Jeremy Nelson. *Mastering Redis*. Packt Publishing Ltd, 2016.

[15] *NodeJS*. https://nodejs.org. accedido: 2017-06-20.

[16] Orwell. *Dev-C++*. http://orwelldevcpp.blogspot.com.es. accedido: 2017-06-26.

[17] Victoria Pimentel y Bradford G Nickerson. "Communicating and displaying real-time data with WebSocket". En: *IEEE Internet Computing* 16.4 (2012), págs. 45-53.

[18] J Racine y col. "The Cygwin tools: a GNU toolkit for Windows". En: *Journal of Applied Econometrics* 15.3 (2000), págs. 331-341.

[19] Stefan Tilkov y Steve Vinoski. "Node. js: Using JavaScript to build high-performance network programs". En: *IEEE Internet Computing* 14.6 (2010), págs. 80-83.

[20] *WebAssembly*. http://webassembly.org. accedido: 2017-06-28.

[21] *Xcode*. https://developer.apple.com/xcode. accedido: 2017-06-26.

[22] Bennet Yee y col. “Native client: A sandbox for portable, untrusted x86 native code”. En: *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE. 2009, págs. 79-93.

[23] Alon Zakai. “Emscripten: an LLVM-to-JavaScript compiler”. En: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM. 2011, págs. 301-312.

CAPÍTULO 7
Anexos

Congreso Universitario de Innovación
Educativa en las Enseñanzas Técnicas
Badajoz, 5-8 de septiembre de 2017

Facilitando la práctica de programación mediante un servicio de compilación y ejecución de programas C en el navegador

Emilio Rodríguez Revert[a] y Salvador España Boquera[a]
[a]Escuela Técnica Superior de Ingeniería Informática, Universitat Politècnica de València, Camino de Vera s/n 46022 Valencia. emrode@inf.upv.es

Abstract

Developing programs is essential to acquire the necessary skills in programming. The programming C language is used in several Engineering degrees in ETSID, at UPV.

In spite of the existence of different tools to program in C, their installation can be a barrier to work at home or, in general, outside the laboratory. Even with the existence of web-accessible environments (such as Codeboard), they need to run the programs on their own server since, in principle, the C language cannot be executed in web browsers. This is dangerous, which explains that users must be registered to use the tool.

In this work we introduce a tool to compile and execute C programs with the peculiarity that they are executed in the browser, drastically reducing the security problems while improving scalability.

This tool, accessible through a suitable web browser, allows testing of any simple program in C without requiring the user to be registered or identified, making it even easier to develop the student practices.

Keywords: *non-presential work, programming environment, compilation, computer security.*

Resumen

Desarrollar programas resulta imprescindible para adquirir las habilidades necesarias en las asignaturas de programación. En la UPV se utiliza el lenguaje C en la asignatura "Informática" en diferentes grados de ingeniería de la ETSID.

A pesar de la existencia de diversas herramientas para programar en C, la instalación de las mismas puede suponer una barrera para trabajar en casa o, en general, fuera del laboratorio. Si bien es cierto que actualmente han surgido entornos accesibles vía web (por ejemplo, Codeboard), éstos necesitan ejecutar los programas en un servidor propio ya que, en principio, el lenguaje C no se utiliza en los navegadores web. Ejecutar código de los usuarios en un servidor resulta peligroso y, por este motivo, entre otros, se requiere estar registrado para trabajar.

En este trabajo presentamos una herramienta para que los estudiantes puedan probar (compilar y ejecutar) los programas que desarrollan con la particularidad de que, al conseguir ejecutar los programas en el propio navegador, se reducen drásticamente los problemas de seguridad al tiempo que se mejora la escalabilidad.

Esta herramienta, accesible con un navegador web adecuado, permite la prueba de cualquier programa sencillo en C sin necesidad tan siquiera de registrarse o identificarse, facilitando todavía más el desarrollo de las prácticas.

***Palabras clave:**. trabajo no presencial, entorno de programación, compilación, seguridad.*

Introducción

La asignatura “Informática”, impartida en los distintos grados de Ingeniería en la Escuela Técnica Superior de Ingeniería del Diseño (ETSID) en la Universidad Politécnica de Valencia, utiliza el lenguaje de programación C para explicar los conceptos básicos de programación.

Se ha elegido C por la relevancia que tiene este lenguaje en la ingeniería. Se trata de un lenguaje de programación imperativo de propósito general y conviene resaltar que gran parte de los lenguajes de programación actuales (véase C++, Java, C#, JavaScript, Objective-C,…) han derivado directa o indirectamente de él, de modo que los conocimientos adquiridos por el alumno en esta asignatura se pueden trasladar y aprovechar al utilizar alguno de estos lenguajes posteriormente.

La experiencia nos muestra que el trabajo práctico resulta de fundamental importancia para aprender a programar, de modo que facilitar al alumno la posibilidad de practicar repercute directamente en la calidad de la enseñanza y en los resultados obtenidos.

Por este motivo, resulta importante que el entorno en el que realiza su trabajo sea accesible, sencillo y amigable. Es preferible la utilización de un entorno de desarrollo integrado (IDE) en lugar de recurrir separadamente a un editor, a un compilador, etc.

En los últimos años se ha estado utilizando el entorno Dev-C++ (Orwell) en esta asignatura y en otras similares, aunque este pasado curso se han realizado pruebas piloto con un entorno de desarrollo vía web (España, 2017). El entorno utilizado en estas pruebas se denomina Codeboard (Estler) (ver https://codeboard.io), aunque no es el único trabajo en esta línea (Stanciu, 2012).

En este trabajo pretendemos mostrar algunas ventajas e inconvenientes de los entornos de desarrollo vía web utilizados para la docencia de programación con el uso del lenguaje C. Tras discutir algunas ventajas e inconvenientes, mostramos una forma de resolver alguno de los problemas detectados y describimos, a continuación, una realización o implementación de una herramienta que pretende resolver varios de esos aspectos (latencia, carga y seguridad) mediante un servicio de compilación de programas en C. Esperamos que este trabajo ayude a una mayor adopción de este tipo de herramientas de desarrollo vía web por parte de todos aquellos que imparten cursos de programación utilizando el lenguaje C. También pensamos que puede extenderse fácilmente a otros lenguajes.

Algunos pros y contras de los entornos de desarrollo online

Como se ha mencionado en la introducción, facilitar el desarrollo del trabajo práctico de los alumnos es vital en el aprendizaje de la programación y esto supone no solamente que el entorno sea sencillo y amigable sino también que se pueda utilizar tanto en el laboratorio como en casa.

Entre las principales ventajas de utilizar un entorno vía web, si lo comparamos con un IDE convencional como Dev-C++, podemos destacar la no necesidad de instalación, lo que repercute positivamente al eliminar una barrera de entrada inicial. Si este entorno se utiliza también en el aula conseguimos una mayor homogeneidad. El hecho de que las prácticas se guarden en la nube facilita que el alumno pueda continuar en casa lo que empezó en clase y viceversa. Aunque nuestro caso de estudio consiste en cursos presenciales, podemos observar que estas ventajas son incluso más significativas cuando consideramos otros escenarios como los cursos online. De hecho, el entorno Codeboard es utilizado en la actualidad en algunos MOOC como edX (https://www.edx.org).

A pesar de estas ventajas, uno de los principales inconvenientes del uso de los entornos de desarrollo utilizados desde un navegador web proviene del hecho de que estos navegadores no han sido desarrollados para utilizar el lenguaje de programación C. Esto significa que herramientas como Codeboard han de recurrir a compilar y a ejecutar el código del alumno en un servidor, y esto repercute en problemas de seguridad, de latencia y de escalabilidad.

Si bien los problemas de seguridad no afectan directamente el trabajo del alumno, es cierto que hacen que muchos centros prefieran evitar este tipo de soluciones a pesar de sus posibles ventajas. La alternativa a tener sus propios servidores es delegar en terceros a riesgo de perder la posibilidad de adaptar la herramienta. Esto supone renunciar a un mayor control sin olvidar que también hay un riesgo de problemas de sobrecarga de estos servidores, con los consiguientes retardos de compilación y ejecución percibidos negativamente por los alumnos.

En la práctica, esto supone que, en algunas ocasiones, el alumno sufre retrasos en la compilación y en la ejecución de los programas.

Además de problemas de latencia que observa el alumno y de los riesgos de instalación que se plantean los docentes, hay un detalle que puede que sea de menor importancia y algo subjetivo: cuando un alumno utiliza un entorno convencional instalado en su máquina tiene la posibilidad de ver el resultado de su trabajo en forma de un ejecutable, algo concreto que puede pasar a terceros si lo desea. Por el contrario, en un entorno online debe limitarse a ejecutar su programa desde el propio entorno.

Situación actual

Actualmente el tipo de plataformas de programación vía web que permiten utilizar el lenguaje C presenta dos problemas:

1. Debido a que el lenguaje C se trata de un lenguaje de programación compilado este sólo puede ser compilado en máquinas con las herramientas necesarias para ello.
2. La mayoría de los navegadores web actuales únicamente son capaces de ejecutar código JavaScript.

Estos dos puntos hacen que las plataformas de ejecución de código C que existen actualmente opten por una solución que sigue el siguiente flujo:

1. El navegador manda el código escrito por el usuario a un servidor que se encargará tanto de compilar como de ejecutar el programa.
2. El servidor manda el resultado de la ejecución al navegador para mostrarlo al usuario.

Este acercamiento presenta dos problemas. Por una parte es muy complicado controlar la seguridad debido a que se está permitiendo a cualquier usuario ejecutar código C en el servidor. A pesar de que normalmente se exige que los usuarios estén registrados, esto no es un gran impedimento para evitar problemas de seguridad. La solución convenional para tratar estos temas consiste en ejecutar el código de los alumnos dentro de un *sandbox* o entorno aislado donde los recursos y permisos están muy limitados. Por otro lado se sobrecarga el servidor al realizar tanto tareas de compilación como de ejecución ya que cualquier

usuario puede ejecutar un bucle infinito en el servidor y la única forma que tenemos de controlarlo es limitando el tiempo de ejecución por usuario. Además de la sobrecarga en CPU debida a la ejecución de los programas, estos no pueden ser interactivos a menos que se mantenga una conexión punto a punto y resulta complicado simular un entorno similar al ofrecido por una herramienta de desarrollo que ejecuta los programas en local. De hecho, una de las limitaciones del entorno Codeboard es que no ofrece una emulación de terminal de manera que la interacción resulta diferente a la que tendríamos con el mismo programa ejecutado en local.

Solución propuesta

De cara a resolver los problemas mencionados en la sección anterior, este trabajo propone recurrir a una de las tecnologías que han surgido recientemente para poder ejecutar programas compilados en el navegador. Si se consigue compilar los programas desarrollados por el alumno de manera que se puedan ejecutar directamente en su navegador conseguimos los siguientes beneficios:

- Eliminamos una de las principales vulnerabilidades asociadas a los entornos que permiten compilar y ejecutar en un servidor el código de los usuarios. Si bien esto no parece un beneficio directo para los alumnos, sí lo es al facilitar la adopción de este tipo de herramientas por parte de los docentes, con todos los beneficios que acarrea y que se han mencionado anteriormente.
- Al no tener que ejecutar el programa en remoto no hay que mantener una conexión permanente para enviar los eventos del usuario y responder con las operaciones de salida del programa. Esto simplifica enormemente el desarrollo de este tipo de herramientas.
- Una vez compilado el programa, el alumno puede ejecutarlo cuantas veces desee sin problemas de latencia ni retardos en la ejecución, ya que normalmente el ordenador donde trabaja no tiene más carga que el propio navegador web que está utilizando.

Con el fin de poder permitir la ejecución de los programas de los alumnos en el navegador, incluso si estos han sido escritos en C, lenguaje que los navegadores no soportan, y tras consultar las tecnologías disponibles, tenemos las siguientes opciones:

- Compilar el programa del alumno a Javascript. Para ello existen soluciones como "Emscripten" (http://emscripten.org) que permiten generar un programa en Javascript a partir del código LLVM (siglas de "Low Level Virtual Machine") obtenido al compilar de forma especial (por ejemplo, usando clang ver http://clang.llvm.org) el programa del alumno (escrito en C). La ventaja de este

sistema es que el programa del alumno podría ejecutarse, en principio, en cualquier navegador.

- Utilizar una tecnología desarrollada por Google para su navegador Chrome que permite ejecutar en el navegador código compilado. Esta tecnología, denominada "Native Client" (https://developer.chrome.com/native-client) permite ejecutar el programa del alumno cuando este utiliza el navegador Chrome con esta opción activada.
- Existen otras opciones como "WebAssembly" (http://webassembly.org) que es un nuevo estándar en desarrollo con unos objetivos similares a "Native Client". El principal problema es que, al ser un estándar muy reciente y en desarrollo, todavía no está soportado por la mayoría de los navegadores y las herramientas de desarrollo están todavía en un estado muy prematuro.

De estas tres alternativas hemos elegido "Native Client" debido a que, por un lado, Emscripten tiene limitaciones con la entrada/salida para emular un terminal y, por otra, WebAssembly no parece suficientemente maduro para ser utilizado en la actualidad. Podemos mencionar que "Native Client" es utilizado en la actualidad por Google y por terceros para crear todo tipo de aplicaciones web que justamente se apoyan en la posibilidad de compilar programas tanto en C como en otros lenguajes (C++, etc.), con lo que se trata de un producto maduro y contrastado. Hemos elegido una versión denominada "Portable Native Client" (abreviado PNaCl) que, al ser independiente del procesador del dispositivo del alumno, permite utilizar la solución propuesta tanto en ordenadores de sobremesa y portátiles como en dispositivos smartphone o tablets (que, en muchos casos, utilizan procesadores ARM en lugar de Intel x86).

Con vistas a facilitar la instalación de esta herramienta por parte de cualquier docente, incluso sin la ayuda de técnicos especializados, se ha utilizado un sistema de despliegue basado en contenedores muy conocido y estandarizado denominado Docker (www.docker.com) que nos permite, una vez instalado Docker, instalar la aplicación y ejecutarla en local con un solo comando. También se ha prestado especial atención a la escalabilidad permitiendo que se puedan lanzar varios procesos de compilación o *workers* de manera que resulta muy sencillo adaptar la herramienta a niveles de carga determinados.

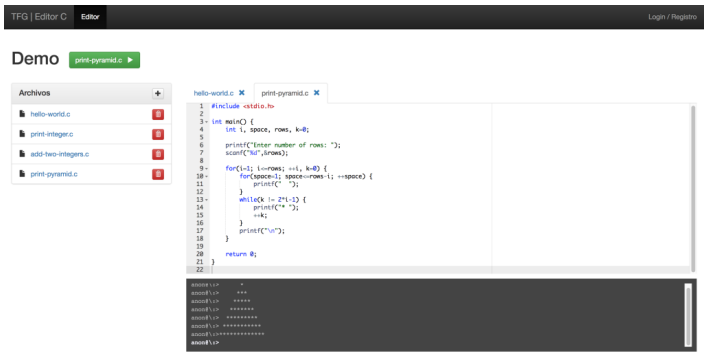
La implementación de este proyecto consta de 3 partes que pueden desplegarse y escalarse con Docker de forma sencilla:

1. Una cola de trabajos para la gestión de la aplicación.
2. El proceso encargado de la compilación o *worker* de compilación de código C (escalable, ya que pueden lanzarse varios de ellos). Estos procesos toman peticiones de la cola de trabajo y la procesan para devolver el resultado.

3. La aplicación web propiamente dicha, que hace uso de lo anterior enviando el código desarrollado por el alumno, cuando este quiera compilar, a la cola de trabajo para que sean procesados, posteriormente, por uno de los *workers*.

La aplicación permite que los alumnos se registren, creen nuevos proyectos y compilen y ejecuten código sobre estos. Dispone de un editor de código y, en caso de compilar el programa sin errores, permite la ejecución del mismo mostrando la salida en la terminal integrada, tal y como puede observarse en la Figura 1.

Figura 1: Ejemplo de ejecución del entorno desarrollado.



Una de las limitaciones de este tipo de entornos, que viene principalmente del uso de Native Client, es la dificultad para incluir de manera arbitraria cualquier tipo de biblioteca. Al utilizar un compilador específico, estamos limitados a las bibliotecas que actualmente soporta el mismo. Debemos mencionar que esta limitación no supone un impedimento de cara a los cursos de iniciación a la programación, como en nuestro caso, aunque sí podría serlo para cursos más avanzados o especializados que requieran alguna biblioteca no soportada.

Conclusiones

En este trabajo hemos mostrado las ventajas e inconvenientes de utilizar un entorno de desarrollo integrado vía web para la iniciación en la programación. A pesar de las ventajas que ofrecen estos entornos, hay elementos que pueden limitar la adopción de estas herramientas por parte de muchos docentes o que dificultan la utilización por parte de los estudiantes. Estas limitaciones vienen fundamentalmente del hecho de que los programas escritos por los alumnos tengan que ser no solamente compilados sino también ejecutados en un servidor. Una vez identificado el problema, hemos dado un primer paso en su resolución

desarrollando un entorno donde, en lugar de ejecutar el programa en el servidor, se envía compilado al cliente en un formato que éste puede utilizar y que permite solventar los problemas detectados.

Esta solución, que planeamos publicar en un entorno de acceso libre, permitirá a cualquier docente, con un mínimo esfuerzo técnico, consistente en instalar Docker, ofrecer a sus alumnos un entorno de programación que podrán utilizar en cualquier ordenador que disponga del navegador Chrome y de acceso a internet sin necesidad de instalar nada más. Al facilitar el trabajo y al reducir la barrera de entrada a las prácticas esperamos que esto se refleje en un mejor aprendizaje.

Como trabajos futuros se plantea la posibilidad de recopilar estadísticas de uso de la herramienta por alumno y por proyecto para permitir a los docentes el uso de técnicas de *learning analytics* (Blikstein, 2011) (Siemens, 2011) de cara a detectar deficiencias, adecuar las clases o mejorar los ejercicios propuestos.

Agradecimientos

Este trabajo ha sido parcialmente financiado por el Vicerrectorado de Estudios, Calidad y Acreditación mediante la subvención PIME 2016-B29, por el Instituto de Ciencias de la Educación y por la Escuela Técnica Superior de Ingeniería del Diseño, todos ellos de la Universidad Politécnica de Valencia.

Referencias

Blikstein P. (2011). *Using learning analytics to assess students' behavior in open-ended programming tasks*. Proceedings of the 1st international conference on learning analytics and knowledge. ACM, pp. 110-116.

España S., Hermida A., Guerrero D., Alvarruiz F. (2017) *El aula informática en casa del alumno desde el primer día* Actas del 25 Congreso Universitario de Innovación Educativa en las Enseñanzas Técnicas.

Estler C., Nordio M. https://codeboard.io, accedido 2017-06-26.

Orwell. http://orwelldevcpp.blogspot.com.es, accedido 2017-06-26.

Siemens G., Long P. (2011). *Penetrating the fog: Analytics in learning and education*. EDUCAUSE review, vol. 46, no. 5, p. 30.

Stanciu C.C., Pastor-Pellicer J., España Boquera S. (2012). *Catalyde: herramienta para enseñar programación*. V Jornada de Innovación Docente, Valencia, España.