
Locutor automático de concursos de programación



Trabajo de Fin de Grado
Curso 2021–2022

Autores

Mario Blanco Domínguez y Javier del Río López

Directores

Pedro Pablo Gómez Martín
Raquel Hervás Ballesteros

Grado en Ingeniería del Software
Facultad de Informática
Universidad Complutense de Madrid

Locutor automático de concursos de programación

Trabajo de Fin de Grado en Ingeniería del Software
Departamento de Ingeniería del software e Inteligencia Artificial

Autores

Mario Blanco Domínguez y Javier del Río López

Directores

Pedro Pablo Gómez Martín
Raquel Hervás Ballesteros

Grado en Ingeniería del Software
Facultad de Informática
Universidad Complutense de Madrid

30 de mayo de 2022

Autorización de difusión

Los abajo firmantes, matriculados en el Grado en Ingeniería del Software de la Facultad de Informática, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores el presente Trabajo Fin de Grado: “Locutor automático de concursos de programación”, realizado durante el curso académico 21/22 bajo la dirección de Pedro Pablo Gómez Martín y Raquel Hervás Ballesteros en el Departamento de Ingeniería del Software e Inteligencia Artificial, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Mario Blanco Domínguez y Javier del Río López

30 de mayo de 2022

Agradecimientos

En primer lugar, nos gustaría agradecer a Raquel y a Pedro Pablo, directores de este TFG, todo el apoyo ofrecido durante el desarrollo de este proyecto, así como todos los consejos brindados.

En segundo lugar, queremos agradecer a toda la organización de los concursos de programación celebrados en la Facultad de Informática de la Universidad Complutense de Madrid. Nos han dado una gran oportunidad permitiéndonos locutar un concurso de programación y hemos podido probar nuestro sistema en dos de ellos. Además, hemos pasado muy buenos momentos junto al resto de voluntarios durante los preparativos de estos eventos.

Por último, también agradecemos a nuestros seres cercanos todo el ánimo que nos han dado durante esta experiencia.

Resumen

En la década de 1970 nació el primer concurso de programación universitario. Hoy en día este concurso se conoce como *International Collegiate Programming Contest* o ICPC, y se ha expandido a la comunidad universitaria a nivel global. ICPC sentó precedentes sobre los concursos de programación. Desde entonces la programación competitiva ha estado en auge, con diversos concursos celebrándose en distintas partes del mundo. La propia Facultad de Informática de la UCM ha albergado muchas de estas competiciones desde hace años, y se prevé que será la sede de otras tantas en el futuro. La retransmisión de estos concursos de programación suele ir acompañada de una locución de los hechos sucedidos, y esta es generalmente realizada por locutores humanos. No obstante, este tipo de retransmisiones no quedan al alcance de muchos concursos de programación que se celebran a menor escala y que cuentan con menos recursos. Por ello, la creación de un locutor automático de concursos de programación permitiría enriquecer la retransmisión e informar a los espectadores de lo que está ocurriendo en el concurso sin suponer un extra considerable para la organización.

Por ello, hemos desarrollado un locutor automático de concursos de programación. Su comportamiento se basa en la recopilación de información estática y dinámica de concursos en tiempo real, y su posterior procesamiento para generar comentarios interesantes y mantener la emoción. La arquitectura de este locutor busca ser extensible, dando la posibilidad de que se añadan nuevos tipos de mensajes fácilmente, ampliando la funcionalidad del programa. Además, ofrece la posibilidad de conectar distintos clientes, ya sea para enriquecer una retransmisión en streaming, publicar tweets periódicos, alimentar bots de Telegram o incluso ayudar a un locutor humano en su labor.

Para comprobar la idoneidad de nuestro locutor se realizó una evaluación real en un concurso de programación en directo. El día 28 de abril de 2022 tuvo lugar el concurso Nacional ProgramaMe 2022. En dicho evento, nuestro sistema se encargó de alimentar un bot de Twitter mediante un cliente

que se encargaba de narrar el desempeño de los concursantes de un instituto concreto. Adicionalmente, se lanzó otro cliente que se encargaba de informar a los espectadores de la retransmisión de YouTube de los sucesos relevantes que ocurrieron durante el concurso. Los resultados obtenidos fueron positivos, y nuestro locutor automático consiguió amenizar una retransmisión de un concurso que no contaba con un locutor humano.

Palabras clave

- Concurso de programación
- Locutor automático
- ICPC
- Programación competitiva
- API REST

Abstract

In the 1970s, the first university programming contest was born. Today this contest is known as the International Collegiate Programming Contest or ICPC, and it has expanded to the university community globally. ICPC set precedents for programming competitions. Since then, competitive programming has been booming, with various competitions held in different parts of the world. Our Faculty itself has hosted many of these competitions for years, and it is expected to host a lot more in the future. The broadcasting of these programming contests is usually accompanied by a locution of the events that take place, and this is usually done by human announcers. However, these types of broadcasts are not in the scope of many smaller-scale programming competitions which have fewer resources. Therefore, the creation of an automatic announcer of programming contests would make it possible to enrich the broadcast and inform the viewers of what is happening in the contest without requiring significant extra effort from the organization.

For this reason, we have developed an automatic caster of programming contests. The behavior of the automatic caster is based on the collection of static and dynamic information from programming contests in real time, and its subsequent processing to generate interesting comments and maintain excitement. The architecture of this speaker seeks to be extensible, giving the possibility of adding new message types easily, expanding the functionality of the program. In addition, it offers the possibility of connecting different clients, either to enrich a streaming broadcast, post periodic tweets, feed Telegram bots or even help a human announcer in their labor.

To check the suitability of our announcer an evaluation was carried out in a real live programming contest. On April 28, 2022 the ProgramaMe 2022 National Competition took place. In that event, our system was in charge of feeding a Twitter bot through a client which was in charge of narrating the performance of the contestants of a specific institute. In addition, another client was launched that was responsible for informing viewers of the YouTube broadcast about the relevant events that took place during the

contest. The results obtained were positive, and our automatic announcer managed to liven up a broadcast of a contest that did not have a human announcer.

Keywords

- Programming contest
- Automatic caster
- ICPC
- Competitive programming
- REST API

Índice general

1. Introducción	1
1.1. Motivación	1
1.2. Objetivos	2
1.3. Plan de trabajo	3
1.4. Estructura del documento	3
1. Introduction	5
1.1. Motivation	5
1.2. Objectives	6
1.3. Work plan	6
1.4. Document structure	7
2. Estado de la Cuestión	9
2.1. Concursos de programación	9
2.2. Infraestructura de los concursos estilo ICPC	11
2.2.1. Contest API	11
2.2.2. CCS y CDS	12
2.2.3. DOMjudge	13
2.3. Locuciones de concursos y herramientas	15
2.3.1. AutoAnalyst	17
2.3.2. MyICPC	18
2.4. Locutores automáticos en otros ámbitos	21
2.4.1. Desarrollo de un locutor en videojuegos de deportes	22
2.4.2. Locutores automáticos controlados mediante aprendizaje automático	24
Conclusiones	26
3. Diseñando un locutor automático de concursos de programación	29

3.1.	Locuciones en concursos de programación	29
3.2.	Nuestra experiencia como locutores	32
3.3.	Diseño de un locutor basado en puntos de vista	34
3.4.	Arquitectura dirigida por eventos	36
3.5.	Flujo de datos completo	37
3.6.	Tecnologías utilizadas y entorno de trabajo	38
4.	Implementación del sistema	41
4.1.	Entrada del sistema	41
4.1.1.	<i>Feed</i> de Eventos del CDS	43
4.2.	Puntos de vista	46
4.2.1.	Modelo	46
4.2.2.	Comportamiento de los puntos de vista	48
4.3.	Salida de datos	49
4.3.1.	Servidor de eventos interesantes	51
4.3.2.	Formato de salida	52
4.4.	Implementación completa	53
4.4.1.	Puntos de Vista implementados	54
4.4.2.	Clientes del locutor	55
4.4.3.	Ejemplo de funcionamiento	57
4.5.	Extensibilidad del sistema	59
4.5.1.	Nuevos puntos de vista	59
4.5.2.	Nuevas salidas de datos	60
4.5.3.	Nuevos clientes del locutor	61
5.	Evaluación	63
5.1.	Evaluación preliminar	63
5.2.	Evaluación final sobre un concurso real	65
6.	Conclusiones y Trabajo Futuro	69
6.1.	Conclusiones	69
6.2.	Trabajo futuro	71
6.	Conclusions and Future Work	75
6.1.	Conclusions	75
6.2.	Future Work	77
7.	Trabajo Individual	81
7.1.	Javier del Río López	81
7.2.	Mario Blanco Domínguez	83
A.	Configuración y Despliegado	87

A.1. Configuración inicial	87
A.2. Desplegado	90
Bibliografía	93

Índice de figuras

2.1.	Diagrama representativo de la infraestructura típica de un concurso de programación que utiliza servidores CCS y CDS.	14
2.2.	Ejemplo de panel de concursante/equipo con vista general de los problemas, últimos envíos y clarificaciones.	15
2.3.	Ejemplo de panel de concursante/equipo con vista de la clasificación general.	16
2.4.	Panel principal de administración de DOMjudge.	17
2.5.	Ejemplo de panel de administrador/juez con vista de las aclaraciones de todos los equipos.	18
2.6.	Ejemplo de panel de administrador/juez con vista de los últimos envíos de todos los equipos.	19
2.7.	Esquema del funcionamiento de AutoAnalyst.	20
2.8.	<i>Timeline</i> de noticias de MyICPC.	21
2.9.	Galería multimedia de MyICPC.	22
2.10.	Retos lanzados a los espectadores en MyICPC.	23
2.11.	Encuestas a los espectadores en MyICPC.	24
2.12.	Esquema de un proceso NLP mediante transformadores.	25
2.13.	Esquema de detección de imágenes mediante transformadores.	26
2.14.	Esquema que combina NLP y detección de imágenes para la retransmisión de eventos en video.	28
3.1.	Diagrama que representa el flujo de datos de la aplicación, los elementos en gris no forman parte de la misma.	40
4.1.	Diagrama de clase de los Puntos de Vista.	47
4.2.	Diagrama de clase de la emisión de eventos.	50
4.3.	Ejemplo de una de las entradas del log que se generó como resultado de la ejecución del programa durante un concurso.	51
4.4.	Demostración del rótulo generado por el <i>player</i> .	55

5.1.	Muestras de algunos de los mensajes que se mostraron durante la retransmisión del concurso ProgramaMe Nacional 2022. . .	66
5.2.	Muestras de algunos de los <i>tweets</i> que se publicaron durante la retransmisión del concurso ProgramaMe Nacional 2022. . .	67
A.1.	Ejemplo del posible contenido del archivo de configuración de variables de entorno <i>.env</i>	88
A.2.	Ejemplo del posible contenido del archivo de configuración <i>FeedInteresante.config.json</i>	90
A.3.	Muestra de los contenedores Docker que se encuentran en funcionamiento.	91
A.4.	Muestra de una shell Unix en el contenedor Docker.	91

Capítulo 1

Introducción

En este capítulo se presenta la introducción al Trabajo de Fin de Grado presentado en esta memoria. En primer lugar, en la sección 1.1 se explicará la motivación que nos ha llevado a realizar este trabajo. Por otro lado, en la sección 1.2, se comentarán los objetivos a cumplir.

1.1. Motivación

En la década de 1970 nació el primer concurso de programación universitario. Celebrado en la Universidad A&M de Texas, el concurso se llamó *First Annual Texas Collegiate Programming Championship*. En 1977, el concurso evolucionó a lo que hoy en día se conoce como *International Collegiate Programming Contest* o ICPC, y se ha expandido a la comunidad universitaria a nivel global. Es uno de los concursos de programación más importantes actualmente.

ICPC sentó precedentes sobre los concursos de programación. Desde entonces la programación competitiva ha estado en auge, con diversos concursos celebrándose en distintas partes del mundo. Aunque no tan prominente, la escena de la programación competitiva tiene también cierta presencia en España. De hecho, la Facultad de Informática de la Universidad Complutense de Madrid ha albergado muchas de estas competiciones desde hace años, y se prevé que será la sede de otras tantas en el futuro.

Los concursos de programación más importantes, sobre todo a nivel mundial, son competiciones que se suelen celebrar en grandes lugares de eventos (como polideportivos o salas de congresos), acostumbran a estar patrocinados, y son habitualmente retransmitidos. También es muy común, debido a la dificultad de congregarse a tanta gente en un mismo lugar y a la situación sanitaria vivida estos últimos años, que se celebren diversos concursos de forma no presencial, y algunos de estos concursos también suelen retransmitirse.

En ambos casos, la retransmisión suele ir acompañada de una locución de los hechos durante el concurso, generalmente realizada por locutores humanos.

Sin embargo, este tipo de retransmisión no queda al alcance de muchos concursos de programación que se celebran a menor escala y que cuentan con menos recursos para la organización de los mismos. Es por eso que la creación de un locutor automático de concursos de programación puede hacer este tipo de concursos más interesante. Contar con este sistema de locución permitirá amenizar la retransmisión e informar a los espectadores de lo que está ocurriendo en el concurso sin suponer un esfuerzo extra considerable para la organización.

1.2. Objetivos

El objetivo principal de este TFG es crear un locutor automático de concursos de programación a partir de información estática y dinámica del concurso. Su comportamiento se basará en la recopilación de información de concursos en tiempo real, y su posterior procesamiento para generar noticias y mantener la emoción durante la retransmisión.

El locutor automático debe ofrecer ayuda a los organizadores de los concursos para que el evento pueda enriquecerse con un mínimo esfuerzo. Se busca que el locutor pueda adaptarse a las diversas situaciones que pueden surgir en estos concursos.

Para conseguir este objetivo general, se plantean los siguientes objetivos específicos:

- El sistema será capaz de recopilar toda la información de lo que está sucediendo en un concurso en tiempo real, decidiendo en cada momento qué información es interesante de cara a la retransmisión.
- La salida del sistema se debe poder utilizar en distintos medios; ya sea para enriquecer una retransmisión en streaming, para publicar *tweets* periódicos, para alimentar bots de Telegram o incluso para ayudar a un locutor humano en su labor.
- El software debe ser extensible, tanto a nivel de ampliar la funcionalidad del locutor con nuevos mensajes, como a la hora de crear nuevos clientes que utilicen el locutor automático.
- Dado el gran número de concursos que se organizan en la Facultad de Informática de la UCM, se probará el sistema desarrollado en uno de ellos para obtener conclusiones acerca del funcionamiento del sistema construido, e intentar enriquecer su retransmisión de cara al público.

1.3. Plan de trabajo

Como plan de trabajo, se ha establecido que se van a realizar reuniones aproximadamente cada dos semanas para tener un seguimiento continuo del estado del proyecto. Esto garantizará que se trabaje de forma constante a lo largo de este curso, y puedan plantearse metas a corto plazo. Estas metas se irán ajustando según avanza el proyecto.

En primer lugar investigaremos acerca de los concursos de programación y su contexto. Lo que requerirá analizar grabaciones de concursos pasados para hacernos una idea de como podría funcionar un locutor automático e investigar acerca de los sistemas que hacen funcionar estos concursos para poder experimentar con ellos.

Después de este análisis previo de lo que concierne a los concursos, se procederá a implementar un locutor automático. Este desarrollo se realizará ayudándose de concursos simulados mediante el mismo software que se utiliza para los concursos reales.

Como colofón de este proyecto se pretende probar en un concurso real el sistema que se desarrolle.

1.4. Estructura del documento

El presente documento está compuesto por siete capítulos. El capítulo actual es el capítulo de introducción, cuyo fin es dar a conocer la motivación que lleva al desarrollo del proyecto y los objetivos de este. A continuación se presentan el resto de capítulos que componen la memoria:

- **Capítulo 2.** Corresponde al estado de la cuestión. Este capítulo se centra en explicar el dominio en el que se desarrolla este proyecto, los concursos de programación.
- **Capítulo 3.** En este capítulo se abordan las decisiones de diseño de alto nivel que se han tomado para el desarrollo de este locutor automático.
- **Capítulo 4.** Capítulo que describe la implementación del sistema. Se precisan detalles técnicos a bajo nivel, describiendo cada uno de los componentes que forman el sistema desarrollado.
- **Capítulo 5.** Este capítulo recoge los resultados de la evaluación a la que se ha sometido el sistema.
- **Capítulo 6.** A lo largo de este capítulo se exponen las conclusiones obtenidas junto al trabajo futuro al que podría verse sometido este proyecto.

- **Capítulo 7.** Este capítulo se centra en la explicación del trabajo individual de cada uno de los integrantes de este TFG.

Adicionalmente, se incluye el apéndice A que describe la configuración y despliegue del sistema.

Chapter 1

Introduction

This chapter presents the introduction to the Final Degree Project presented in this report. First of all, section 1.1 will explain the motivation that has led us to carry out this work. In section 1.2, the objectives to be met will be discussed.

1.1. Motivation

In the 1970s, the first university programming contest was born. Held at Texas A&M University, the contest was called the First Annual Texas Collegiate Programming Championship. In 1977, the contest evolved into what is now known as the International Collegiate Programming Contest or ICPC, and has expanded to the university community on a global level. It is currently one of the most important programming contests.

ICPC set precedents for programming contests. Since then, competitive programming has been booming, with several contests being held in various different parts around the world. Although not so prominent, the competitive programming scene also has some presence in Spain. In fact, the Faculty of Informatics of the Complutense University of Madrid has hosted many of these contests for years, and it is expected to host many more in the future.

The most important programming contests, especially worldwide, are contests that are usually held in large event venues (such as sports centers or congress halls), are usually sponsored, and are also usually broadcasted. It is also very common, due to the difficulty of gathering so many people in the same place and the health situation experienced in recent years, that some contests are held remotely, and some of these contests are also usually broadcasted. In both cases, the broadcast is usually accompanied by a voice-over of the events during the contest, usually carried out by human announcers.

However, this type of broadcasting is not within the reach of many programming contests that are held on a smaller scale and that generally have fewer resources to spend. That's why the creation of an automatic caster of programming contests can make such contests more interesting. Having this system will make it possible to liven up the broadcast and inform viewers of what is happening in the contest without requiring significant extra effort from the organization.

1.2. Objectives

The main objective of this Final Degree Project is to create an automatic caster of programming contests from static and dynamic contest information. The system's behavior will be based on the collection of information from contests in real time, and its subsequent processing to generate news and maintain excitement during the broadcast.

The automatic caster must offer help to the organizers of the contests so that the event can be enriched with minimal effort. It is sought that the system can adapt to the various situations that may arise in these contests.

To achieve this general objective, the following specific objectives are proposed:

- The system will be able to collect all the information about what is happening in a contest in real time, deciding at each moment what information is interesting for the broadcast.
- The output of the system must be able to be used in different media; either to enrich a streaming broadcast, publish periodic tweets, feed Telegram bots or even help a human caster in their work.
- The software must be extensible, regarding both the expansion of functionality of the announcer with new messages, and the creation of new clients that use the automatic caster.
- Given the large number of contests that are organized at the Faculty of Informatics of the UCM, the developed system will be tested in one of them to obtain conclusions about its performance, and try to enrich the broadcast to the public.

1.3. Work plan

As a work plan, it has been established that meetings will be held approximately every two weeks to continuously monitor the status of the project. This will ensure that work is carried out constantly throughout the whole

project duration, and that short-term goals can be set. These goals will be adjusted as the project progresses.

Firstly, we will research programming competitions and their context. This will require analysing recordings of past contests to get an idea of how an automated caster might work, and researching the systems that make these contests work in order to experiment with them.

After this preliminary analysis of what is involved in programming contests, we will proceed to implement an automatic caster. This development will be carried out with the help of simulated contests using the same software that is used for real ones.

As culmination of this project, the aim is to test the developed system in a real contest.

1.4. Document structure

This document is composed of seven chapters. The current chapter is the introductory chapter, whose purpose is to make known the motivation that leads to the development of the project and its objectives. Below are the rest of the chapters that make up the memory:

- **Chapter 2.** Corresponds to the state of the matter. This chapter focuses on explaining the domain in which this project is developed, programming contests.
- **Chapter 3.** This chapter discusses the high-level design decisions that have been made for the development of this automatic caster.
- **Chapter 4.** Chapter describing the implementation of the system. Lots of technical details are provided, describing each of the components that make up the developed system.
- **Chapter 5.** This chapter contains the results of the evaluation to which the system has been subjected.
- **Chapter 6.** Throughout this chapter, the conclusions obtained are presented together with the future proposed for this project.
- **Chapter 7.** This chapter focuses on the explanation of the individual work of each of the members of this Final Degree Project.

Additionally, the appendix A that describes the configuration and deployment of the system is included.

Capítulo 2

Estado de la Cuestión

Si se quiere empezar a desarrollar un locutor automático de concursos de programación, lo ideal es estudiar como se locutan actualmente los concursos de programación, para luego poder trasladarlo a un sistema automático.

El hecho de que estos concursos no sean muy comunes, ni exista una gran cantidad de entidades narrándolos, hace que sea complicado encontrar información sobre el proceso exacto que se utiliza para locutarlos. Sin embargo se pueden observar ciertos patrones y arquitecturas comunes entre muchos de estos concursos, así como en sistemas similares.

2.1. Concursos de programación

Un concurso de programación es una competición en la que los participantes intentan resolver el mayor número de problemas en el menor tiempo posible. Durante transcurso del mismo, los concursantes realizan envíos de un algoritmo que creen que resuelve el problema en un lenguaje de programación (Java, C++, Python...) para obtener un veredicto. Los problemas que se plantean pueden ser de muchas categorías distintas, y suelen tener asociadas restricciones de tiempo y/o memoria para descartar soluciones poco eficientes.

Los concursos de programación se celebran en ámbitos muy diversos. En los tipos de concursos que se tratan en este trabajo existen algunas características comunes. Una de estas características es que el marcador suele congelarse antes del final del concurso, lo que implica que a partir del momento de congelación no se puede saber si los participantes han resuelto más problemas o ha cambiado su posición en el ranking; congelar el marcador se hace principalmente para no desvelar antes de tiempo quién ha ganado el concurso, y en su lugar desvelarlo en una ceremonia de clausura posterior. Cabe destacar los siguientes concursos:

- A nivel internacional, el concurso más conocido es el ***International Collegiate Programming Contest***¹ (ICPC). En esta competición concursan equipos formados por 3 integrantes para ver cuál es capaz de resolver el mayor número de problemas, y en caso de empate, qué equipo lo ha hecho en menos tiempo. Esta competición tiene varias fases de clasificación. Cada equipo representa a una universidad, y un equipo puede participar si se clasifica en la fase regional correspondiente, donde se enfrentarán a universidades próximas. España, por ejemplo, pertenece a la región *Southwestern Europe Regional Contest*² (SWERC), donde se enfrenta a universidades de Portugal, Francia, Israel, Italia y Suiza. Este concurso es también considerado el *original*, a raíz del cual han surgido muchos otros, incluidos algunos de los explicados a continuación.
- La **Olimpiada Informática Española**³ (OIE) es un concurso en el que participan alumnos procedentes de institutos de toda España (secundaria, bachillerato y grado medio). Se concursa de forma individual y se utiliza un sistema de puntuación por subtareas. Cada problema planteado en el concurso se divide en varias subtareas, donde cada una aporta una puntuación distinta en función de la dificultad de la misma. Para poder participar en la OIE se celebran una serie de olimpiadas regionales previas⁴, así como un concurso clasificatorio abierto⁵.
- **AdaByron**⁶ es un concurso universitario de programación promovido por distintas universidades españolas. El formato de AdaByron suele consistir en una serie de concursos regionales, celebrados en distintas comunidades autónomas, que sirven como clasificatorios para la final. En ocasiones también se celebra un concurso online multisede, y cuando es posible, se celebra una final a nivel nacional con los mejores equipos de las competiciones regionales. Este formato se ha visto alterado de distintas formas en los últimos años por motivo de la pandemia de la COVID-19. AdaByron sigue las mismas reglas que el ICPC, de hecho ICPC sirvió de principal inspiración para su creación.
- **ProgramaMe**⁷ es un concurso que sigue las mismas reglas y formato que AdaByron, con la diferencia de que los integrantes de los equipos proceden de Ciclos Formativos de Formación Profesional. Este concurso también se inspiró y sigue las mismas reglas que el SWERC.

¹<https://icpc.global/regionals/abouticpc>

²<https://swerc.eu/>

³<https://olimpiada-informatica.org>

⁴<https://olimpiada-informatica.org/regionales>

⁵<https://olimpiada-informatica.org/registro>

⁶<https://ada-byron.es/>

⁷<https://programame.com/quees.php>

- **Las 12 Uvas**⁸ es un concurso online organizado desde la Facultad de Informática de la Universidad Complutense de Madrid, y tiene lugar todos los años el 31 de diciembre. El concurso es individual, consta de 12 problemas de dificultad variable y se compite por ser quien resuelva el mayor número de problemas, o quien más rápido lo haga en caso de empate. El concurso está dividido en distintas categorías y está abierto a todo el mundo.

2.2. Infraestructura de los concursos estilo ICPC

En los concursos de programación, los algoritmos propuestos por los participantes son recibidos por un servidor, denominado juez automático, que se encarga de ejecutarlos contra muchos casos de prueba. Además, comprueba que la solución enviada es suficientemente buena, es decir, no usa ni memoria adicional ni tarda más tiempo del estipulado, comparándola con la solución de los jueces del concurso.

Los jueces automáticos no se utilizan solo para concursos, existen multitud de plataformas y jueces automáticos como “¡Acepta el reto!”⁹, Online Judge¹⁰ (antes conocido como el juez de la Universidad de Valladolid o juez de la UVA), Codeforces¹¹, la plataforma de programación competitiva de Google¹², AtCoder¹³, HackerEarth¹⁴, HackerRank¹⁵, LeetCode¹⁶, etc.

Muchas de estas plataformas también se utilizan para concursos, pero el funcionamiento de los jueces automáticos que utilizan es opaco y en ocasiones secreto. Existe sin embargo un estándar definido por la ICPC Foundation denominado Contest API que provee un estándar para sistemas CDS y CCS, definidos más adelante. Estos sistemas se utilizan en todos los concursos de ICPC, AdaByron y ProgramaMe, por lo que lo hace un estándar muy atractivo a estudiar para el objetivo de este TFG.

2.2.1. Contest API

Con el propósito de tener una API útil y estandarizada para herramientas que trabajen con concursos estilo ICPC, desde la ICPC Foundation¹⁷ especificaron Contest API¹⁸.

⁸<https://las12uvas.es>

⁹<https://www.aceptaelreto.com>

¹⁰<https://onlinejudge.org/index.php>

¹¹<http://codeforces.com/>

¹²<https://codingcompetitions.withgoogle.com/>

¹³<https://atcoder.jp/>

¹⁴<https://www.hackerearth.com/>

¹⁵<https://www.hackerrank.com/>

¹⁶<https://leetcode.com/>

¹⁷<http://icpc.foundation/>

¹⁸https://ccs-specs.icpc.io/2021-11/contest_api

Esta API nos ofrece toda la información sobre el concurso que podríamos esperar, desde la información estática como información de los equipos, hasta información dinámica como el estado del concurso, la tabla de clasificación y los veredictos para saber el resultado de los envíos al juez. Concretamente, incluye distintos métodos para obtener información del concurso, así como la puntuación, problemas, veredictos, etc. También incluye un feed de eventos que documenta cambios en el estado del concurso y otros datos de interés según estos ocurran. Tanto el CCS como el CDS explicados posteriormente implementan el Contest API.

El Contest API especifica una interfaz REST en conjunto a un *feed* de eventos NDJSON¹⁹. Los distintos *endpoints* del Contest API pueden agruparse en 3 categorías:

- Configuración: estos objetos son información estática ya especificada al principio del concurso. Salvo por alguna peculiaridad de un concurso en concreto, esta información no debería modificarse a lo largo del concurso. Estos *endpoints* son: `contests`, `judgement-types`, `languages`, `problems`, `groups`, `organizations`, `teams` y `team-members`.
- Datos en tiempo real: esta información se irá generando durante el concurso, a modo de eventos ocurridos a lo largo de este. Nos encontramos los siguientes *endpoints*: `state`, `submissions`, `judgements`, `runs`, `clarifications`, `awards` y `commentary`.
- Información agregada: consiste en agrupar la información de las dos categorías anteriores para obtener información del concurso. Por definición, esta información es de solo lectura. En esta categoría se encuentran los *endpoints* `scoreboard` y `event-feed`.

2.2.2. CCS y CDS

El **Contest Control System** (CCS) es una manera de denominar a un sistema de control de concursos estilo ICPC. Tiene una especificación muy estricta de los requisitos que tiene que tener un programa para que pueda ser considerado CCS²⁰. Entre estos requisitos se encuentran licencias, características referentes a la persistencia de datos, tipos de cuentas, autenticación segura, comunicación encriptada, capacidad de funcionar sin acceso a internet, marcas de tiempo para las distintas operaciones que tengan lugar, etc.

A efectos prácticos, un CCS es un software capaz de realizar automáticamente las distintas operaciones de un concurso de programación. Estas operaciones incluyen: ejecutar envíos de los equipos, juzgar los envíos, facilitar un sistema de clarificaciones entre concursantes y jueces, cálculo del

¹⁹https://ccs-specs.icpc.io/2021-11/contest_api#event-feed

²⁰https://ccs-specs.icpc.io/2021-11/ccs_system_requirements

ranking, generar una representación externa de los resultados del concurso y la propia configuración del CCS.

Un sistema CCS implementa el Contest API (explicado en la sección 2.2.1), algunos ejemplos de implementaciones CCS son DOMjudge, explicado más adelante y Programming Contest Control System²¹.

El **Contest Data Server**²² (CDS), se define como un punto de encuentro para obtener toda la información relacionada con un concurso de programación. Provee una serie de *endpoints* REST para acceder a una variedad de servicios relacionados con el concurso mediante peticiones HTTP estándar. Como el CDS implementa el Contest API, algunos de estos *endpoints* son los mismos que en el CCS. Un CDS permite la asignación de roles a usuarios, y cada usuario con un rol puede acceder a determinados servicios del CDS que se ofrecen o no en función de los servicios que se puedan ofrecer de la red del concurso. Entre estos servicios que el CDS agrega se encuentran: video-reacciones de un equipo en el momento de recibir un veredicto por parte del juez, una copia del directorio `/home` de cualquier equipo, un *feed* RSS del concurso (distinto formato al *feed* de eventos estándar), imágenes de cada equipo y la institución que representan, etc.

Un CCS puede utilizarse por sí solo, pero un CDS debe utilizarse en conjunto a un CCS del que pueda extraer información a través de sus *endpoints*. En una infraestructura típica, representada por la figura 2.1, el cliente se conectaría al CDS para obtener toda la información que necesite, el CDS ofrece ciertos servicios propios y el resto de información la obtiene del CCS. El CCS se encarga de gestionar el concurso en sí, y tiene a su disposición una serie de servidores (al menos uno) que utiliza para ejecutar el código que envían los participantes del concurso. Los clientes no necesitan conocer detalles de la arquitectura más allá de la localización del CDS y los servicios que ofrece.

Es importante destacar que este esquema puede variar en base a las peculiaridades de cada implementación, pero de cara a locutar un concurso, se sabe que toda la información puede obtenerse a través del CDS utilizando la especificación del Contest API.

2.2.3. DOMjudge

DOMjudge²³ es un sistema automatizado para gestionar y sostener concursos de programación que se ajusta a la definición de CCS. Ofrece una interfaz de usuario tanto a jueces como a participantes, se ha utilizado en muchos concursos y es de código libre.

²¹<https://pc2ccs.github.io/>

²²<https://tools.icpc.global/cds/>

²³<https://www.domjudge.org/>

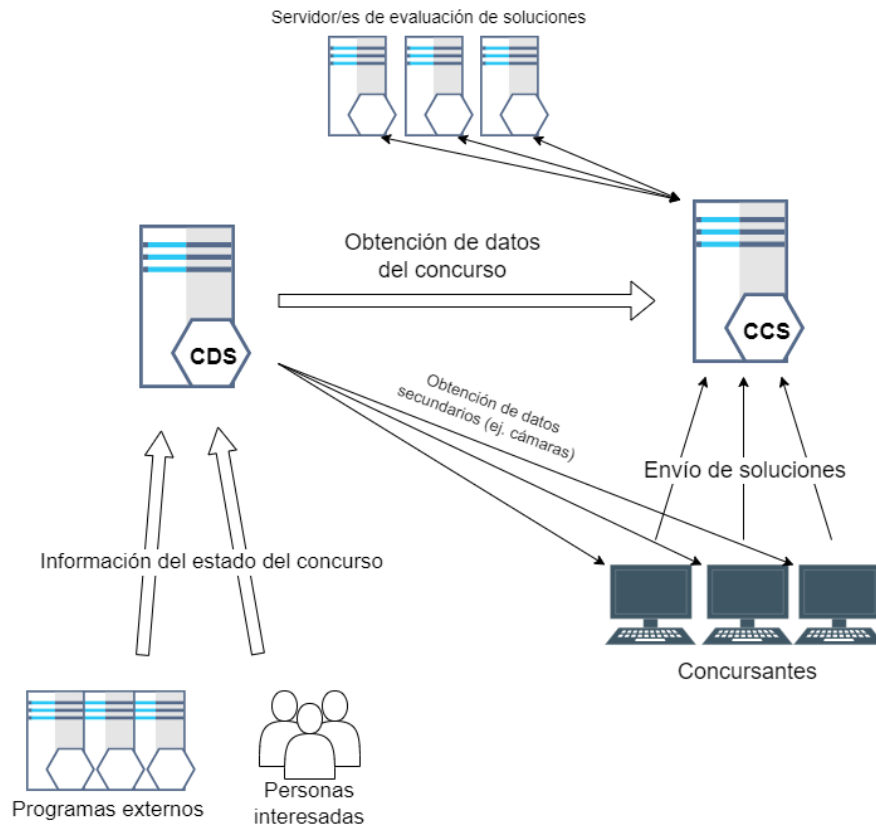


Figura 2.1: Diagrama representativo de la infraestructura típica de un concurso de programación que utiliza servidores CCS y CDS.

DOMjudge dispone de un manual²⁴ donde se detallan desde los requisitos del sistema, pasando por el proceso de instalación, hasta los pasos a seguir para hacer tu propio concurso.

Esta herramienta ofrece distintas funcionalidades e interfaces dependiendo de cuál sea el rol del usuario en el concurso. Por ejemplo, si se trata de un concursante o un equipo, estos pueden tener acceso a un panel que incluye el estado general de todos los problemas, un resumen con los últimos envíos que se han hecho, y un panel de aclaraciones para comunicarse con los jueces si es necesario. Un ejemplo de este panel puede verse en la figura 2.2. Un participante también puede ver la clasificación general del concurso como se muestra en la figura 2.3.

En el caso de los administradores o jueces del concurso, se tiene un conjunto de funcionalidades completamente distinto. En el panel principal que se muestra en la figura 2.4 se pueden observar algunas de las muchas op-

²⁴<https://www.domjudge.org/docs/manual/7.3/index.html>

DOMjudge

Home

Problemset

Print

Scoreboard

Submit

Logout

contest over

RANK	TEAM	SCORE	A	B	C	D	E	F	G	H	I	J	K
5	 Technische Universität München	9 835	188 1 try	49 1 try	74 2 tries		161 1 try	5 tries	125 1 try	23 1 try	12 1 try	147 2 tries	16 1 try

Submissions

time	problem	lang	result
14:59	F	CPP	WRONG-ANSWER
14:38	F	CPP	WRONG-ANSWER
14:25	F	CPP	WRONG-ANSWER
14:11	F	CPP	WRONG-ANSWER
13:43	F	CPP	WRONG-ANSWER
13:08	A	CPP	CORRECT
12:41	E	CPP	CORRECT
12:27	J	CPP	CORRECT
12:11	J	CPP	WRONG-ANSWER
12:05	G	CPP	CORRECT

Clarifications

time	from	to	subject	text
14:55	Jury	All	General issue	When the contest ends at 15:00, please take everything with you: keyboard, tcr's...
13:23	Jury	All	General issue	If you want some more food, go downstairs and take it. First come, first serve. ...
13:21	Jury	All	problem J	Yes. In every round, every better will place a bet.
11:50	Jury	All	General issue	Dear teams, Lunch is being served starting 12:00, it is located where the din...
11:28	Jury	All	General issue	Whenever you want to eat the food on your tables, please take it just outside th...
10:45	Jury	All	problem F	You can have multiple items in your inventory at the same time. You can equip

Figura 2.2: Ejemplo de panel de concursante/equipo con vista general de los problemas, últimos envíos y clarificaciones.

ciones de las que disponen los administradores, pueden verse como vistas completas de lo que para los equipos son vistas parciales. Por ejemplo, se pueden visualizar todas las aclaraciones de todos los equipos en un mismo lugar, como se muestra en la figura 2.5, o los últimos envíos junto a la batería de pruebas del problema (y donde ha fallado si es el caso) como se muestra en la figura 2.6.

Estas interfaces se ofrecen como servicio web junto al Contest API, propio de un sistema CCS.

2.3. Locuciones de concursos y herramientas

Aunque muchos concursos de programación no se locutan por distintos motivos, existe una serie de concursos, en su mayoría del ICPC, que sí cuentan con una retransmisión y locución durante el transcurso de la competición. Algunas de estas locuciones se analizan en la Sección 3.1.

La locución de un acto, sea o no un concurso de programación, es irremediablemente subjetiva; depende fundamentalmente de a quien vaya dirigida y de quien lo locute. Hay sucesos que pueden interesar en mayor o menor medida según si el destinatario es un experto, un familiar o alguien afín a un participante en concreto.

En algunas retransmisiones de concursos de programación, se pueden observar las herramientas y métodos que se utilizan para llevar a cabo una

RANK	TEAM	SCORE	A	B	C	D	E	F	G	H	I	J
1	Treenity University of Cambridge	11 1323	170 1 try	103 1 try	56 2 tries	289 3 tries	146 1 try	215 2 tries	77 2 tries	40 1 try	12 1 try	91 1 try
2	Los Patrons University of Oxford	10 1145	217 2 tries	38 1 try	85 2 tries	1 try	176 1 try	268 2 tries	157 2 tries	12 1 try	18 1 try	72 1 try
3	Double Cycle Cover University of Tartu	10 1470	267 1 try	63 1 try	214 1 try		104 1 try	183 1 try	134 2 tries	35 1 try	12 1 try	280 8 tries
4	Triniceratops University of Cambridge	9 788	177 1 try	59 1 try	31 1 try	1 try	159 1 try	5 tries	197 2 tries	6 1 try	10 1 try	93 2 tries
5	TUMbling Technische Universität München	9 835	188 1 try	49 1 try	74 2 tries		161 1 try	5 tries	125 1 try	23 1 try	12 1 try	147 2 tries
6	2 Brits and a Dutchman University of Oxford	9 1021	297 2 tries	147 1 try	83 2 tries		229 1 try		112 1 try	18 1 try	10 1 try	60 2 tries
7	Q++ Leiden University	9 1185	267 1 try	123 1 try	77 1 try		205 1 try		135 2 tries	39 2 tries	13 1 try	236 2 tries
8	<(OvO)> Saarland University	9 1226	293 4 tries	50 1 try	166 1 try		101 3 tries		152 1 try	25 2 tries	8 1 try	209 5 tries
9	Oxford Ji-geiko University of Oxford	9 1498	295 2 tries	74 2 tries	135 1 try	1 try	281 1 try	1 try	210 3 tries	26 1 try	16 2 tries	180 8 tries
10	=_=>+<<.:(-0)+(-+0000-00~0) Technische Universität München	8 915		138 1 try	60 1 try		263 3 tries		154 1 try	26 2 tries	23 1 try	117 3 tries
	Let's party! ---		235	72	173				110	24	12	179

Figura 2.3: Ejemplo de panel de concursante/equipo con vista de la clasificación general.

retransmisión o locución del concurso. Los concursos de programación actualmente son narrados por locutores humanos, utilizando varias de las herramientas disponibles de las ICPC tools. Estas herramientas se encargan de mostrar información que luego los locutores se encargan de transmitir de una forma interesante para el espectador.

Es importante hacer notar que la labor de estos programas no va más allá de presentar la información que se obtiene de una forma amigable y fácil de interpretar, pero los encargados de generar contenido interesante, ya sea imagen, audio, video o cualquier combinación de estos elementos, son los locutores que interpretan, filtran y generan contenido interesante que en última instancia se transmite al espectador. Es necesario, por tanto, que exista un equipo encargado de gestionar este aspecto del concurso, si este quiere ser retransmitido de una forma similar a la observada. Lo que puede no ser siempre viable por distintos motivos: disponibilidad de personal cualificado, estándar de calidad de la retransmisión deseado, etc.

En las finales de la ICPC, por ejemplo, disponen de una serie de comentaristas y analistas, así como distintas cámaras de las que se van mostrando imágenes intermitentemente. Juntando esto con una serie de programas que muestran la información del concurso en tiempo real, se consigue retransmitir el concurso con audio y video a una audiencia sin resultar pesado, repetitivo o monótono.

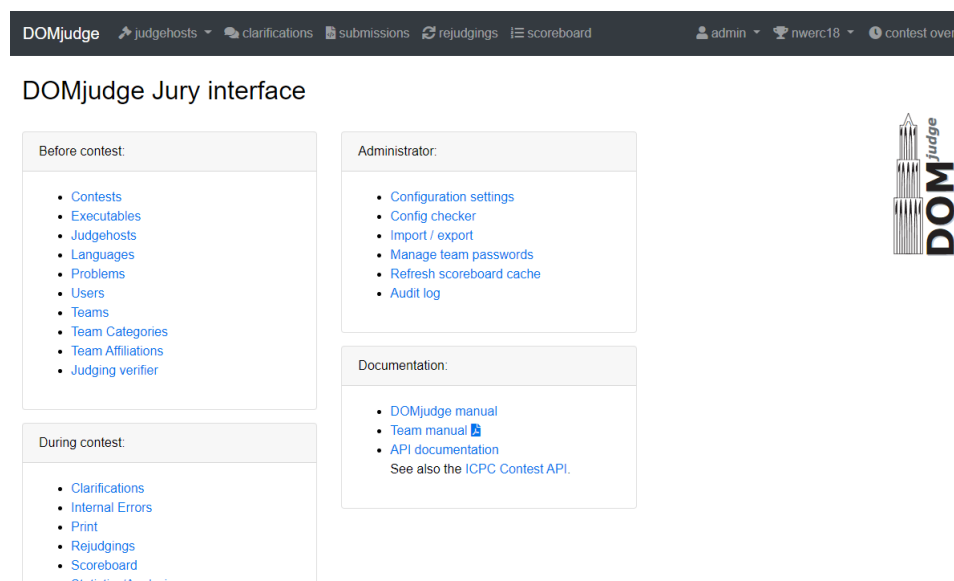


Figura 2.4: Panel principal de administración de DOMjudge.

2.3.1. AutoAnalyst

Las retransmisiones de concursos de programación tienen un problema que, generalmente, otras competiciones no suelen tener. Cuando un espectador se sienta a ver una grabación de uno de estos concursos tal cual, verá a muchas personas sentadas enfrente de un ordenador, y no sabrá qué está ocurriendo realmente. Para afrontar esto, ICPC creó un equipo de analistas con el objetivo de dinamizar la retransmisión de sus concursos. Su director, Fredrik Heintz, comentó en una entrevista²⁵ el propósito de este equipo: hacer de las finales del ICPC un deporte. El reto es hacer que la audiencia entienda qué ocurre en las mentes de los participantes y sus ordenadores.

El trabajo de este equipo es analizar estas competiciones para generar contenido interesante de cara a los espectadores. Este trabajo es, en parte, manual, pero también disponen del apoyo de herramientas automáticas. Para ello, se crearon las iCat²⁶ (*ICPC Contest Analysis Tool*), un grupo de herramientas que les facilita el análisis de los concursos en tiempo real.

Gracias a las iCat, los analistas pueden hacer un análisis semi-automático de los concursos. Como contaron en la presentación del proyecto AutoAnalyst²⁷, del cual hablaremos más detalladamente a lo largo de esta sección, el equipo de analistas cuentan con el *feed* de eventos del concurso, el acceso al juez para ver los intentos de resolución de los problemas, acceso a los ar-

²⁵<https://youtu.be/iIGKrmtJDFs?t=13>

²⁶<https://clics.ecs.baylor.edu/index.php?title=ICAT>

²⁷https://youtu.be/X_vmfkcS-Ds?t=158

ID	time	from	to	subject	text	state
186	14:46	We failed maths	Jury	problem C	The distance between points must be 10^{-3} On the example outputs, the distance ...	✓
184	14:37	FAU send percent conFAUndedl	Jury	problem J	Does Julia choose the same team as the majority of all bettors with the most poi...	✓
182	14:30	Snackoverflow	Jury	problem G	When the ball would hit a block and fall into the hole at the same time, does it...	✓
180	14:13	double wis(Key pp)	Jury	problem B	What is causing our compiler issue?	✓
178	14:08	Prim's prime squad	Jury	problem F	Can the shortcut item of a level be the item to be obtained in that level?	✓
177	13:35	Double Cycle Cover	Jury	general	No need, a volunteer came and fixed it.	✓
175	13:24	Double Cycle Cover	Jury	general	Our laptop is not charging and will run out of battery soon.	✓
171	13:19	<(@v@)>	Jury	problem E	Which formats do the integers in the input have?	✓
170	13:19	Keukentafelwiskundigen	Jury	general	Now everyone has gotten some chance at getting lunch, can we get more than two p...	✓

Figura 2.5: Ejemplo de panel de administrador/juez con vista de las aclaraciones de todos los equipos.

chivos de los participantes, junto a la vista de sus pantallas y webcams, y la información estática del concurso. AutoAnalyst es la herramienta de las iCat que se encarga de analizar los concursos de programación en tiempo real.

AutoAnalyst²⁸ es un proyecto desarrollado por el equipo de analistas de ICPC. Su propósito es analizar concursos estilo ICPC para generar comentarios y noticias interesantes. Analiza los eventos importantes ocurridos, hace cálculos para saber en qué tiempo debería un equipo resolver un problema para escalar en el ranking y es capaz de prever que problemas están intentando resolver los participantes, resolviendo así el problema que comentábamos anteriormente sobre qué ocurre en las mentes y ordenadores de los participantes. Este proyecto está más centrado en realizar análisis de lo que está ocurriendo en concursos de programación que en las locuciones en sí.

Esta herramienta no tiene documentada su API²⁹ por el momento. Actualmente está en desarrollo. Su esquema de funcionamiento puede apreciarse en la figura 2.7.

2.3.2. MyICPC

Por otra parte, existe una aplicación web llamada MyICPC³⁰ diseñada por el equipo programadores de ICPC para seguir un concurso en tiempo real

²⁸<https://github.com/icpc-live/autoanalyst#readme>

²⁹https://clics.ecs.baylor.edu/index.php?title=AutoAnalyst_API

³⁰<https://icpc.global/community/myicpc>

DOMjudge

- judgehosts
- clarifications
- submissions**
- rejudgings
- scoreboard

admin nweric18 contest overview

Submissions

Show newest unverified unjudged all

Filter

ID	time	team	problem	lang	result	verified	by	test results
s9675	20:00	Gregor Behnke	J	CPP	TOO-LATE (RUN-ERROR)	claimed	gregor	T x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
s9674	16:45	Alexander Dietsch	B	CPP	TOO-LATE (TIMELIMIT)	claimed	epi	x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
s9673	15:01	Reunion2.0	J	CPP	TOO-LATE (TIMELIMIT)	claimed	unclaim	x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
s9672	15:00	Om de hoek	J	PY2	TOO-LATE (WRONG-ANSWER)	claimed	unclaim	w x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
s9671	15:00	<{0v0}>	J	CPP	TOO-LATE (TIMELIMIT)	claimed	unclaim	x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
s9670	15:00	deeply biconected	J	CPP	TOO-LATE (WRONG-ANSWER)	claimed	jury	x x x x x x x x x x x x x x x x w x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x

Figura 2.6: Ejemplo de panel de administrador/juez con vista de los últimos envíos de todos los equipos.

y mantener más entretenidos a los espectadores. Esto lo consigue gracias a incluir, aparte del *scoreboard* de la competición, un *feed* de noticias de qué está ocurriendo en las redes sociales (relacionado con la competición), y una sección de juegos y encuestas para los espectadores.

Para que esta herramienta pueda funcionar durante el concurso, los organizadores tienen que configurar la misma. En esta aplicación destaca el apartado “*timeline*”, donde se mezclan notificaciones estilo *feed* de eventos (equipo X ha superado el problema Y), con información relevante publicada en redes sociales (puede configurarse por ejemplo para que muestre los *tweets* con ciertos *hashtags* o *tweets* de cuentas de los organizadores). La riqueza de los eventos reside en la inclusión de la información de las redes sociales, ya que las notificaciones que muestra MyICPC se limitan a las resoluciones de los problemas. Podemos ver un ejemplo de cómo luce este apartado de noticias en la figura 2.8. Adicionalmente, podríamos acceder al contenido multimedia postado en las redes sociales, en el apartado galería, como podemos ver en la figura 2.9.

En MyICPC también se incluyen encuestas (figura 2.11), diferentes vistas del marcador del concurso e incluso pequeños retos / ejercicios para optar a aparecer en el ranking del mismo u obtener algún premio. Por ejemplo, en la figura 2.10 vemos un reto que se lanzó a lo largo de un concurso, que consistía en crear un programa que escribiese por pantalla una frase un número determinado de veces, y mandar un *Vine* (antigua plataforma de

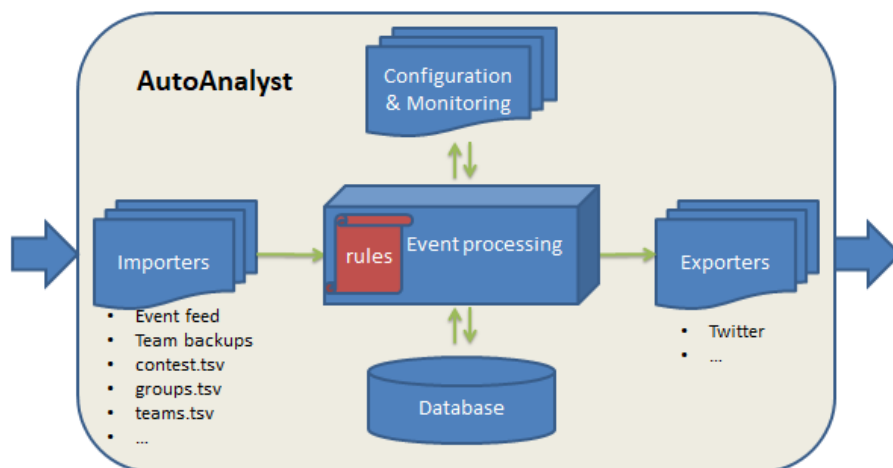


Figura 2.7: Esquema del funcionamiento de AutoAnalyst.

videos de corta duración) mostrándolo. Todas estas funcionalidades buscan mantener al público entretenido a lo largo de la competición.

Para comprobar el interés de la audiencia usando esta aplicación, se realizó el estudio Cerny y Donahoo (2018) durante las finales de 2014, 2015 y 2016. Este estudio concluye que el uso de aplicaciones de “segunda pantalla” mantienen más entretenidos e informados a los espectadores de retransmisiones de este estilo.

Como podemos ver en el estudio anterior, la mayoría de usuarios (en torno al 75 % de los usuarios) de MyICPC usaron un ordenador para conectarse a la aplicación, y esto es normal, ya que casi todos los espectadores ven las finales en la comodidad de sus casas, donde pueden seguir el concurso mientras hacen otras tareas. Nos parece muy interesante el posible uso de MyICPC para los espectadores que se encuentren físicamente en el lugar de la competición. Pensamos que sería algo complicado y quizá aburrido seguir el concurso presencialmente.

Por ello, cabe destacar el trabajo de fin de máster de Filip Rysavy (Rysavý (2018)), estudiante checo de Ingeniería del Software, que portó esta aplicación a dispositivos Android. No obstante, su objetivo estaba más enfocado en superar lo que denomina la “Great Firewall of China”, alegando que durante las finales de aquel año (2018), disputadas en Beijing, no se

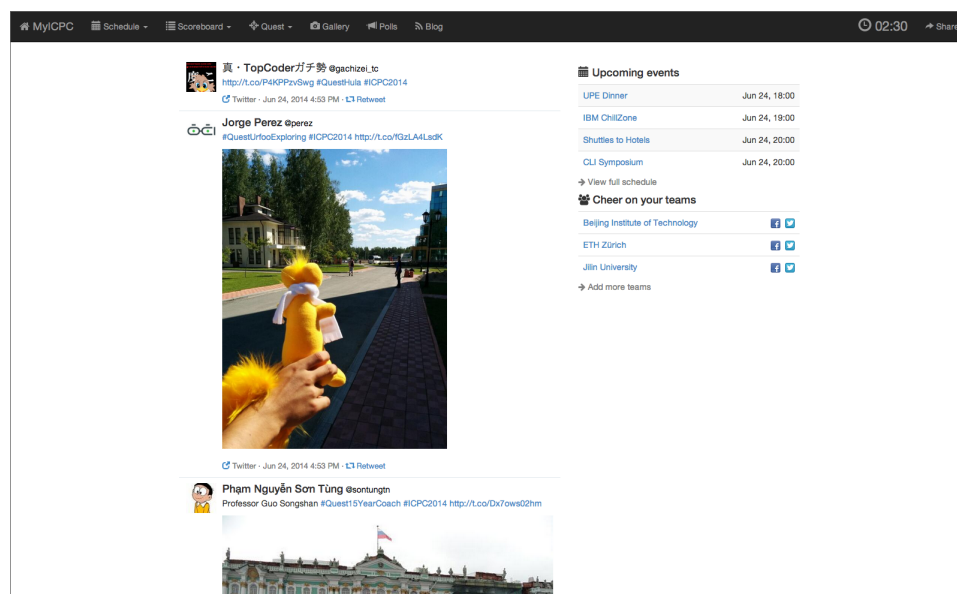


Figura 2.8: *Timeline* de noticias de MyICPC.

podría usar esta aplicación, al hacer uso de datos enviados por Twitter (bloqueado por China). Esta aplicación haría más sencillo conectarse a MyICPC desde móviles, ya que no tendrían que hacer uso de una VPN para poder usar la aplicación.

Esta aplicación es muy interesante, ya que, aunque no sea una aplicación que permita la narración de un concurso, está enfocada en mejorar la retransmisión de este, haciendo que el espectador se enganche.

2.4. Locutores automáticos en otros ámbitos

Aunque actualmente no existen locutores automáticos capaces de narrar un concurso de programación, sí podemos encontrar su implementación en otros ámbitos.

Un ejemplo muy popular es el locutor que incorporan los juegos de EA Sports. Todos estos juegos incluyen un narrador que enriquece de manera significativa la experiencia del jugador mediante diversos comentarios, ya sean narrando lo que sucede en la partida o con comentarios más elaborados relacionados con otros aspectos, emulando así una retransmisión real. Encontramos juegos de diversos deportes, siendo fútbol uno de los más populares, pasando por otros deportes como fútbol americano, hockey, artes marciales e incluso competiciones de automóviles (Fórmula 1).

El primer FIFA, videojuego de fútbol, que incorporó un narrador en los

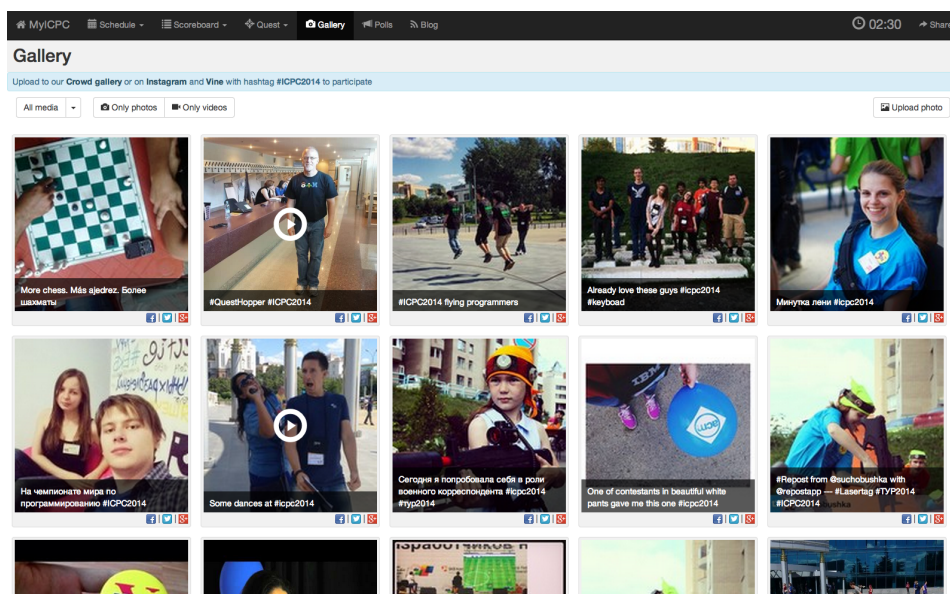


Figura 2.9: Galería multimedia de MyICPC.

partidos, fue FIFA 97³¹. Como podemos apreciar, han pasado muchos años desde que implementaron este sistema por primera vez, y EA Sports, desde el principio, ha tratado de perfeccionar este sistema, ya sea haciendo mejoras para que la voz del locutor se escuche menos robótica y más realista, y mejorando la detección de eventos *in-game* para que la retransmisión en sí sea más jugosa y no se limite a mencionar los nombres de los jugadores cuando tocan balón y a cantar goles. En la siguiente sección se presentan técnicas que utilizaron los desarrolladores de este sistema de locución automático.

2.4.1. Desarrollo de un locutor en videojuegos de deportes

Ernest W. Adams³² trabajó en Electronic Arts durante varios años, participando en el desarrollo de la saga de videojuegos Madden NFL de fútbol americano. Además, publicó muchos artículos relacionados con el desarrollo de videojuegos en la web Game Developer³³, anteriormente conocida como “Gamasutra”. Adams trabajó seis años en la producción de audio y video de esta serie de videojuegos de fútbol americano, y nos presenta una pequeña guía de cómo se hizo en Adams (2009).

En primer lugar, Adams insta a decidir el alcance del locutor a desarrollar. Por un lado, podría limitarse a hacer los comentarios “jugada a jugada”,

³¹https://en.wikipedia.org/wiki/FIFA_97#Commentary

³²https://en.wikipedia.org/wiki/Ernest_W._Adams

³³<https://www.gamedeveloper.com>

The screenshot shows the MyICPC website interface. At the top, there's a navigation bar with links like 'MyICPC', 'Schedule', 'Scoreboard', 'Quest', 'Gallery', 'Pols', and 'Blog'. The main content area is titled 'Quest' and includes instructions for participating. Below this, there's a 'Challenges' section with a 'Welcome Program' challenge that has 30 points and a 24:04:49 time limit. To the right, there's a 'Contestant leaderboard - TOP 5' table. Below the challenge, there's a 'What's happening now' section featuring a tweet from @gachizai_30 about a house challenge. At the bottom right, there's a 'Non-contestant leaderboard - TOP 5' table and a 'Vote' section.

Participant	Points
Ikumi Hide (@gachizai_30)	220
Sean Wentzel (@h4tguj)	50
Sanjeev Shenoy (@sanjeevshenoy)	40
Johannes Kaphammer (@icpcQuestAcc14)	40
Osamu Koga (@osa_1)	30

Participant	Points
Borja Sotomayor (@borjasotomayor)	340
Mikael Renström (@renstrom_icpc)	220
Niyaz Nigmatulin (@niyaznigmatul)	220
Matina Kalkounis (@matinakounis)	190
Louisa Bengut (@louis_bengut)	120

Figura 2.10: Retos lanzados a los espectadores en MyICPC.

que en su caso vendría ser que transcribir el *feed* de eventos de la partida (faltas, puntos y poco más), o programar también lo que él denomina los “comentarios coloridos”. Un ejemplo de un comentario de este estilo podría ser: *el equipo “x” está teniendo mala suerte hoy*, ya que no podemos obtener esta información directamente con los eventos ocurridos, sino que necesitamos un análisis adicional, el cual puede parecer fácil para un locutor humano, pero para uno automático es una tarea más compleja. Estos comentarios añadirán mucha riqueza a la narración y hacen que no sean tan monótonas.

Hemos hecho un análisis de los consejos que pensamos que son aplicables a nuestro proyecto, como por ejemplo la grabación de las voces, o si la narración es estilo radio o estilo televisión (en nuestro caso, no tiene sentido tratar de asemejarse a uno de estos estilos de narración, porque en un concurso de programación no hay tantos eventos como en una competición deportiva).

Este artículo es muy interesante, y podemos asimilarlo en nuestro proyecto en muchos aspectos. Ver el análisis de un experto que se enfrenta a una tarea parecida a la nuestra es enriquecedor. Para empezar, nos ha dado una idea de como estructurar los análisis que hagamos a comentarios reales de locutores en nuestro ámbito, separándolos por categorías. Cabe destacar que en su proyecto generar comentarios variados (tratándose de simular una retransmisión real) es crucial, y tiene más relevancia que en el nuestro por numerosos motivos. El primero, y más evidente, es que su proyecto se trata de un videojuego, y si este se limita a decir solo los eventos que ocurren podría ser aburrido y, sobre todo, poco realista. Por experiencia propia, estos locutores automáticos en videojuegos tienen comentarios interesantes, y no están tan alejados de una locución real. De hecho, a veces sorprenden, lo cual

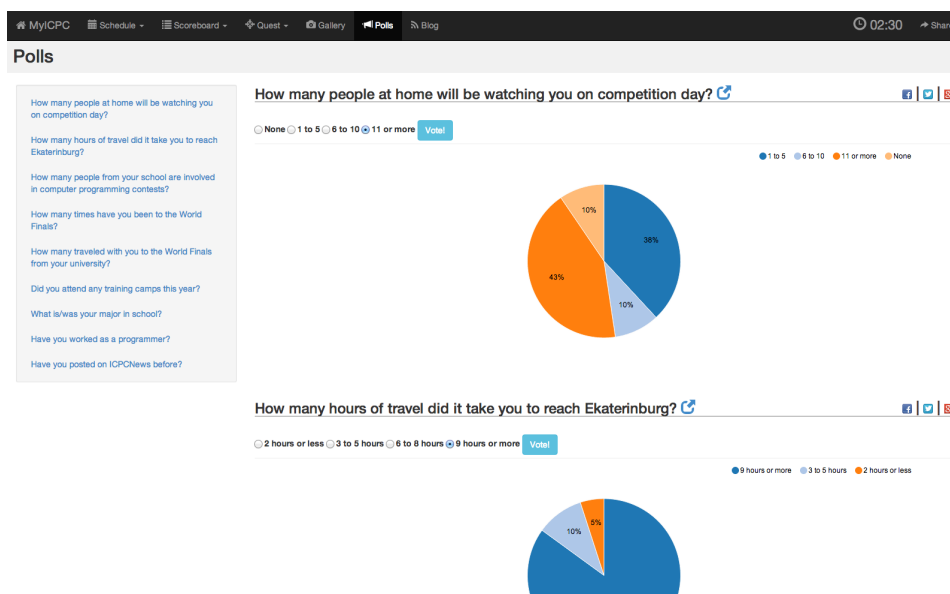


Figura 2.11: Encuestas a los espectadores en MyICPC.

causa impresión en el jugador. El otro motivo es que una locución auditiva debe generar más contenido que una locución escrita, que, en un principio, es el alcance deseado en este TFG. Por último, en una competición de programación no hay tanta acción como podría existir en una competición deportiva. Esto lo hemos podido comprobar gracias a la visualización de algunas retransmisiones en diferido de estos concursos. El locutor, al contrario de lo que podría ocurrir en retransmisiones deportivas, no habla en todo momento. Gracias a este artículo tenemos la visión de un experto en la materia, que se ha tenido que enfrentar con el problema de crear un locutor. Desde un principio, nuestro objetivo era “hacer bonito” el *feed* de eventos ocurridos en el concurso, por lo que tendremos que generar comentarios interesantes partiendo de eventos, en principio, poco interesantes.

2.4.2. Locutores automáticos controlados mediante aprendizaje automático

Es posible que el título de este TFG incite a pensar en un locutor que, mediante técnicas de aprendizaje automático, sea capaz de extraer los eventos desde la propia retransmisión y generar comentarios utilizando también estas técnicas, en vez de rellenar unas plantillas de mensaje con los datos del evento ocurrido. La tarea de desarrollar un locutor utilizando esta rama de la inteligencia artificial es ardua. Prueba de ello son los pocos proyectos que lo están implementando. A continuación se detallan algunos de ellos.

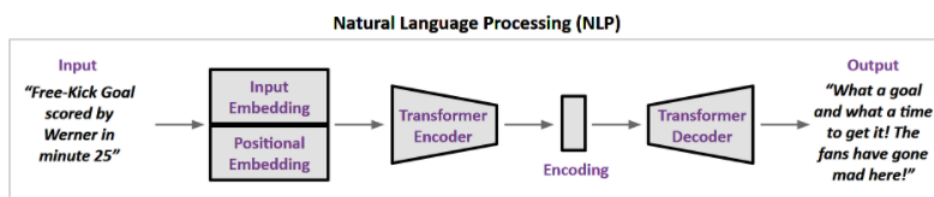


Figura 2.12: Esquema de un proceso NLP mediante transformadores.

La universidad Kaist de Corea organiza una competición de fútbol vía inteligencia artificial. La competición se basa en desarrollar inteligencias artificiales que compitan entre sí, pero tienen un apartado de competición para desarrollar locutores automáticos³⁴. Sin embargo, no hemos encontrado ningún locutor automático en acción. Una competición totalmente automatizada, incluyendo a los participantes y al narrador, sería muy interesante de ver, pero no desde el lado del espectador (ya que estas competiciones perderían emoción), sino como demostración de la potencia del *machine learning*.

Adicionalmente, encontramos un artículo interesante³⁵ que trabaja en lo mismo que la competición citada anteriormente. Este artículo, publicado por Deep Gaming AI (equipo de data scientists de la India), nos presentan una hipótesis de funcionamiento de un locutor que utiliza el aprendizaje automático para poder narrar competiciones deportivas. La cuestión es que ya existen algoritmos de IA basados en detección de imágenes, como todos sabemos, y también existen algoritmos IA para procesar lenguaje natural (NLP). Sin embargo, estos funcionan de una forma muy diferente el uno del otro, por lo que combinarlos es una tarea complicada. Muchas tareas de NLP siguen un modelo de aprendizaje automático basado en transformadores, que funcionan como se muestra en la figura 2.12.

El autor de este post comenta el artículo científico Carion et al. (2020), en el que muestran buenos resultados aplicando transformadores para algoritmos de detección de imágenes. Como se muestra en la figura 2.13, han utilizado el mismo modelo basado en transformadores para la detección de imágenes.

Esto es un gran avance para los locutores automáticos desarrollados, ya que, sabiendo que podría utilizarse el modelo de transformadores para detección de imágenes, podríamos entrenar una red neuronal para comentar partidos combinada con la red del procesamiento de lenguaje. La figura 2.14 muestra como se podría utilizar el codificador del modelo de detección de imágenes con el decodificador NLP.

³⁴http://aiworldcup.org/ai_commentator

³⁵<https://medium.com/deepgamingai/ai-based-video-game-commentary-with-end-to-end-transformers-a4729e34582c>

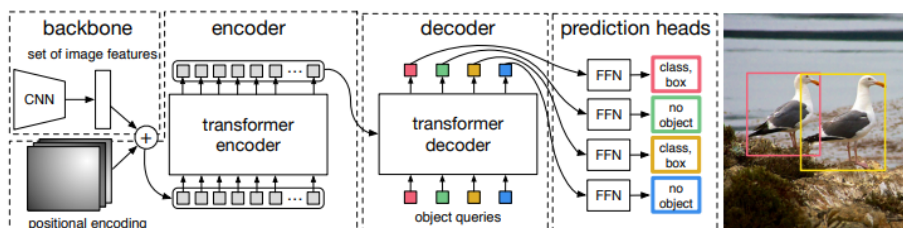


Figura 2.13: Esquema de detección de imágenes mediante transformadores.

Conclusiones

A lo largo de este capítulo hemos visto todas las herramientas disponibles enfocadas en la locución de concursos de programación. Desde un principio éramos conscientes de que ver una competición de programación, incluso con narradores y una producción con varias cámaras, puede llegar a ser algo aburrido. Esto se debe a que, en estos concursos, no hay mucha acción realmente. Un equipo puede tardar un tiempo muy largo en resolver un problema, y cuando lo consigue, tampoco puede celebrarlo muy efusivamente para no molestar al resto de participantes. Entre eventos, queda un espacio vacío, difícil de rellenar. Por ello, las retransmisiones deben mantener a los espectadores interesados en las mismas.

La celebración de los concursos de programación suele ir acompañada de una retransmisión. Sin embargo, organizar un *streaming* de la competición entretenido conlleva un esfuerzo extra para los organizadores. En las grabaciones que hemos podido ver de concursos hemos observado que es muy común que uno o varios locutores humanos amenicen la retransmisión narrando los sucesos de la competición. Otras grabaciones se limitan a retransmitir el ranking de los participantes. En este caso, el seguimiento del concurso puede ser complicado para los espectadores. Por ello, la creación de un sistema de locución automático permitiría solventar este problema sin suponer un gran esfuerzo para aquellas retransmisiones de menor escala. Contando con que la Facultad de Informática de la Universidad Complutense de Madrid ha albergado múltiples concursos de programación en los últimos años y prevé ser la sede de otros tantos en el futuro, el desarrollo de este locutor automático podría sentar las bases para la emisión en directo de este tipo de eventos, en las cuales un locutor automático se encargaría de la locución de los sucesos, y así reducir la carga de trabajo para los organizadores, que podrían dedicar más tiempo a otros aspectos que avivan la retransmisión, como entrevistas con los participantes, explicación de problemas o mostrar las cámaras de los participantes.

En este capítulo hicimos un repaso a la herramienta AutoAnalyst, en la

subsección 2.3.1. Este proyecto, desarrollado por el equipo de analistas de ICPC, promete hacer un análisis en tiempo real de concursos estilo ICPC. Es capaz de analizar cómo están rindiendo los participantes, ya que vimos que podía estimar en qué problemas están trabajando los equipos, y realizar cálculos sobre cuándo un equipo debe resolver un problema para escalar en la clasificación. Este TFG podría haber sido una extensión de este sistema, pero esta idea fue descartada porque actualmente se encuentra en fase de desarrollo, y no dispone de documentación suficiente para poder plantearse extender su funcionalidad. Además, el análisis que produce este sistema está más centrado en facilitar el trabajo de los analistas del concurso que en las locuciones en sí. Cabe destacar que este análisis podría ser muy interesante para locutores humanos, ya que podrían dar estadísticas difíciles de calcular a simple vista.

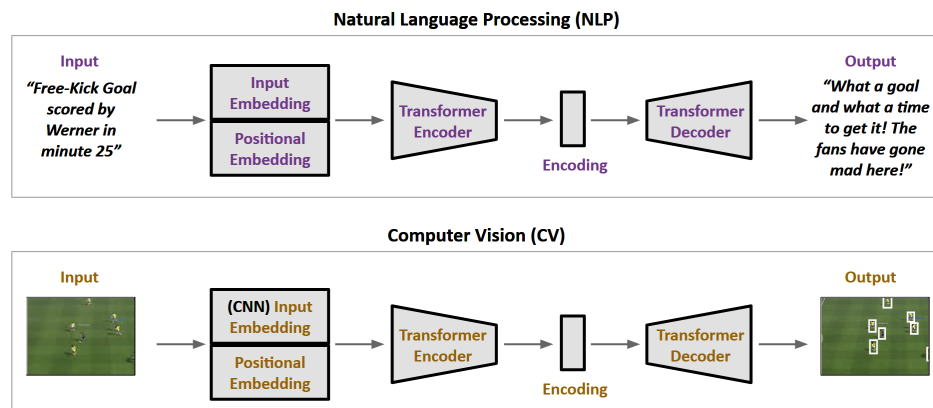
La solución que ofrece MyICPC al problema de mantener al público entretenido, tal y como se mostró en la subsección 2.3.2 resulta muy interesante. MyICPC facilita el seguimiento del concurso en tiempo real mediante contenido interactivo. La relación con el proyecto expuesto en este TFG es clara, ya que el objetivo es el mismo. No obstante, la solución ofrecida es muy diferente a la planteada en nuestro proyecto al no estar orientada a la locución de concursos de programación.

Ambas herramientas sirven como inspiración para nuestro proyecto al compartir los mismos objetivos. De AutoAnalyst nos ha gustado la idea de aportar comentarios que serían complicados de extraer a simple vista, incluso por un locutor humano. MyICPC nos ha mostrado que para mantener al público entretenido e informado es esencial capturar los sucesos de la competición y mostrarlos de una forma amigable y sencilla. La funcionalidad relacionada con ofrecer contenido interactivo a los participantes es muy interesante, pero escapa del alcance de este TFG.

Por otro lado, la investigación realizada acerca de locuciones automáticas en otros ámbitos (subsección 2.4.1) nos ha dado una idea de como estructurar los análisis que hagamos a comentarios reales de locutores en nuestro ámbito, separándolos por categorías. Aparte de pensar en situaciones que podrían ocurrir en concursos de programación, es crucial que se hagan análisis a situaciones reales para poder comprender mejor la realidad de los concursos.

Para finalizar, hemos pasado por el desarrollo de locutores mediante técnicas de Aprendizaje Automático, el cual es un campo por explorar. Nuestro objetivo está muy lejos de estos trabajos. Lo que nosotros buscamos es generar un *feed* de eventos mucho más interesantes para los concursos de programación, que podría aplicarse en varios casos, como generar *tweets* para una cuenta de Twitter, pasando por enriquecer una locución de narradores reales o crear un bot de Telegram que informa del estado del concurso. No obstante, resulta interesante ver como se podría llegar a sustituir un locutor humano gracias a las tecnologías de Aprendizaje Automático.

Transformer Model



CV+NLP Transformer Model

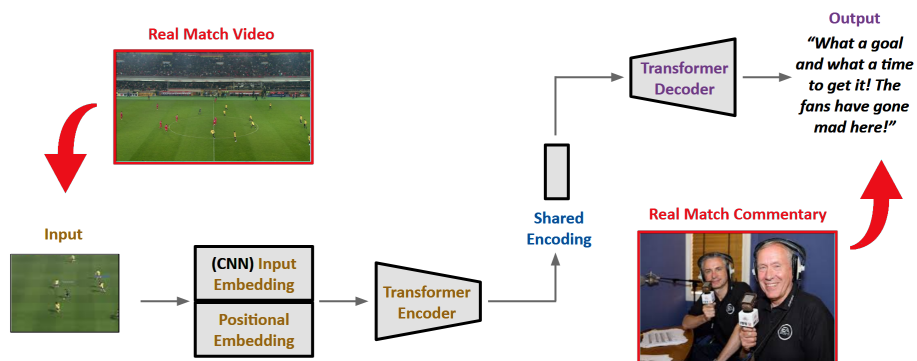


Figura 2.14: Esquema que combina NLP y detección de imágenes para la retransmisión de eventos en video.

Capítulo 3

Diseñando un locutor automático de concursos de programación

El objetivo de este TFG es crear un locutor automático de concursos de programación. En este capítulo describiremos las decisiones de diseño de alto nivel que se han tomado, así como los procesos de investigación que se han llevado a cabo para informar estas decisiones.

3.1. Locuciones en concursos de programación

Se considera que la función de un locutor en cualquier ámbito consiste en añadir una capa de entretenimiento a un evento ya existente. En muchas ocasiones es difícil concebir que un cierto tipo de evento no vaya de la mano de un locutor. Incluso hay gente que a los estadios de fútbol se lleva una radio, para tener alguien que le cuente lo que está viendo.

La función de un locutor puede incluir el aportar información del evento que puede no ser obvia a primera vista, expresar cierto tipo de emociones que el evento provoca (y de esta manera amplificarlas en el espectador), analizar situaciones hipotéticas y sus implicaciones, destacar sucesos que pueden resultar interesantes, crear o amplificar la tensión (sobre todo si se trata de competiciones), entretener al espectador durante momentos en los que no estén ocurriendo sucesos interesantes, etc.

Así, un locutor automático no debería limitarse a describir lo que ocurre en un momento dado de forma plana, sino que debería suplir el mayor número de funciones posible de las que ya hace un locutor humano. Es por tanto un requerimiento que el locutor automático comente sucesos interesantes. Esto no quiere decir que un locutor consista únicamente en un filtro que distinga los sucesos interesantes de los que no lo son, sino que también debería generar sucesos nuevos a partir de la información de la que dispone en cada momento.

Catalogar un suceso como interesante es obviamente subjetivo. Dentro de esta subjetividad, la idea es crear un locutor que agrade al mayor número de personas posible. Es útil por tanto fijarse en lo que funciona en la actualidad, que ya ha ido siendo perfeccionado con el paso del tiempo, y partir desde ahí.

Por ello, vamos a analizar locuciones de concursos reales pasados, con el objetivo de identificar los distintos tipos de comentarios que se realizan y poder clasificarlos.

Se han observado distintas locuciones de concursos pasados, sobre todo del ICPC explicado anteriormente en la sección 2.2. Entre las locuciones observadas se encuentran: Regional Madrid Ada Byron 2021¹, ICPC World Finals Moscow 2021², Southwestern Europe Regional Contest (SWERC) 2019-2020³, ICPC World Finals 2019⁴ y 2018 ICPC World Finals en su versión española⁵ e inglesa⁶.

La conclusión es que las locuciones varían mucho dependiendo del contexto. Sin embargo, existen ciertos comentarios o actitudes que son recurrentes en la mayoría de concursos. A continuación se encuentran, a alto nivel y clasificados, estos comentarios:

- Comentarios previos al inicio del concurso:
 - Se desea buena suerte a los participantes/equipos.
 - Se mencionan los distintos premios (si existen).
 - Se mencionan patrocinadores (si hay).
 - Se explica el funcionamiento del concurso.
- Comentarios contextuales:
 - Se empiezan a ver los primeros problemas resueltos.
 - Si un problema que se ha resuelto hace mucho tiempo no ha vuelto a ser resuelto.
 - Se comenta que un equipo debería ponerse con un problema específico, siendo este problema uno de los más resueltos que este equipo aún no tiene.
 - A un equipo le está costando resolver uno de los problemas más resueltos (tiene varios envíos erróneos).
 - Cuando un equipo ha resuelto muchos problemas al primer intento.

¹<https://www.youtube.com/watch?v=kEqoPwMIS50>

²<https://www.youtube.com/watch?v=5IrtE19r-UA>

³https://www.youtube.com/watch?v=tJ_Nrdgvs8I

⁴<https://www.youtube.com/watch?v=PFirT8mqvXo>

⁵<https://www.youtube.com/watch?v=1Ltega-Snyw>

⁶<https://www.youtube.com/watch?v=LpIT35y33WY>

- Casos especiales:
 - Resolver a la primera algo que no ha intentado nadie.
 - Casos especiales en los que el marcador sería distinto si no se hubiese realizado algún envío fallido.
- Comentarios a gran escala:
 - Qué porcentaje de los problemas han sido intentados/resueltos.
 - Qué porcentaje de los equipos han resuelto un número específico de problemas (varias veces a lo largo de los concursos).
 - Intentos y/o envíos correctos por lenguaje y/o problema.
 - Cuando Hay varios equipos empatados a un número específico de problemas.
- Repasos de la clasificación:
 - Repaso de los equipos en las primeras posiciones.
 - Repaso de los primeros equipos de cada categoría.
 - Repaso de los problemas.
 - Repaso de los premios (si hay) y quien se los ha llevado o llevaría si el concurso sigue su ritmo.
- Comentarios respecto a los premios (si los hay):
 - Un equipo se garantiza un premio al ser el primero en resolver un problema específico (solo si existe tal premio).
 - Se comenta si en un momento específico nadie cumple los requisitos necesarios para llevarse un premio en concreto.
 - Se comenta que si el marcador no cambia, un equipo específico se llevará un premio.
- Comentarios respecto al tiempo:
 - El concurso empieza.
 - Cuando han transcurrido o quedan un número específico de horas/minutos de concurso (por lo general números redondos).
 - Falta poco tiempo para que se congele el marcador.
 - El marcador se ha congelado.
 - El concurso acaba.
- Comentarios después de congelar el marcador. En este intervalo del concurso no hay tantos sucesos que locutar, y se suele transmitir menos información o más espaciada en el tiempo:

- Se menciona cuántos (o qué) problemas necesita un equipo para entrar en la clasificación.
- Tiempo restante de concurso, sobre todo cuanto más se acerca el final.
- Se comenta cuantos problemas distintos ha intentado un equipo.
- Se comenta si un equipo ha intentado el mismo problema muchas veces.

La elaboración de esta lista no exhaustiva de comentarios que los locutores suelen hacer en concursos de programación ha resultado en las siguientes conclusiones.

En primer lugar, muchos de estos comentarios son candidatos a poder implementarse en un locutor automático una vez estudiada la viabilidad de implementar cada uno. Para cada comentario, la información necesaria, la complejidad del comentario o la frecuencia de aparición son algunos de los factores que influyen en la viabilidad de implementación.

En segundo lugar, ha servido para darse cuenta de que los comentarios dependen de muchos factores. En algunas ocasiones, es difícil encontrar terreno común entre las locuciones de dos concursos distintos, a pesar de que, en teoría, están haciendo lo mismo. Esto se debe principalmente a que cada locutor se centra en aspectos distintos. A esto se le añade que los comentarios que se pueden hacer durante un concurso de programación seguramente también cambien con el tiempo. Esto conduce a la conclusión de que, con el objetivo de que no quede obsoleto en el futuro, el locutor automático debe construirse de manera que facilite cambios futuros, ampliaciones, etc.

3.2. Nuestra experiencia como locutores

El 19 de febrero de 2022 se celebró en la facultad la II Olimpiada Informática de Madrid (OIM)⁷. Tuvimos la oportunidad de cubrir el rol de locutores de la retransmisión del concurso que se iba a hacer en directo en YouTube⁸, lo que nos permitió vivir en primera persona la experiencia de locutar un concurso de programación.

En el concurso competían alrededor de 60 concursantes de 44 centros distintos. El evento se extendió a lo largo de todo el día, y la competición como tal tuvo una duración de 3 horas y media, y durante ese periodo es cuando tuvo lugar la colaboración en la retransmisión. El marcador se congelaba a 30 minutos del final, es decir, que a partir de las 3 horas de concurso se dejaron de ver actualizaciones en el marcador para poder desvelar los cambios en el acto de clausura.

⁷<https://olimpiadas.informatica.ucm.es/2022/>

⁸<https://www.youtube.com/watch?v=r80ah6fd578>

La OIM utiliza un sistema de puntuación basado en subtareas, como se explica en la sección 2.1. Por lo tanto, el concurso no es del mismo tipo de los que se pretenden locutar automáticamente en este trabajo. Sin embargo sí que tiene muchos aspectos en común con los sistemas basados en el modelo ICPC (también explicado en las secciones 2.1 y 2.2) o similares.

Los objetivos principales de esta prueba consistieron en:

- Concretar la información que se quiere transmitir en un concurso real, lo que incluye saber qué información puede ser de interés para el espectador, priorizar algunos comentarios sobre otros y descartar sucesos que pueden resultar irrelevantes o repetitivos.
- Estimar el tiempo entre los distintos comentarios que se hagan, o dicho de otra manera, estimar qué cantidad de información es excesiva y qué cantidad es escasa.
- Descubrir nuevas ideas sobre posibles comentarios acerca del concurso que no estén incluidas en la clasificación de la sección anterior.
- Obtener información general relevante para nuestro TFG que no se haya contemplado en los anteriores puntos.

El día anterior al concurso se realizó una reunión de coordinación donde se acordó que la retransmisión no tendría locución continua, sino que estaría compuesta por una serie de intervenciones nuestras intercaladas entre otro tipo de contenido, como entrevistas en las que no participábamos, videos preparados, cámaras de los participantes y el marcador del concurso.

Durante la retransmisión del concurso se realizaron un total de 7 intervenciones en el directo⁹. Además, en momentos donde no se estaba locutando o mostrando mensajes en pantalla, se aprovechó para analizar la situación del concurso, anotar los sucesos interesantes y planificar la siguiente intervención.

De cara a los objetivos que se plantearon, se obtuvieron las siguientes conclusiones:

- La misma información puede ser o no relevante en distintos momentos del concurso, dependiendo de las veces que haya ocurrido algo similar. Por ejemplo, la primera vez que alguien resuelva un problema es un suceso más interesante que las siguientes veces que otros equipos resuelvan ese mismo problema.
- Después de congelar el marcador hay muy poca información interesante que se pueda ofrecer al público que no incluya recapitular o ponerse en casos hipotéticos.

⁹<https://www.youtube.com/watch?v=r80ah6fd578>

- Se obtuvieron algunas ideas de nuevos comentarios como indicar si los primeros puestos iban muy igualados. Esta idea por ejemplo surgió a raíz de que, durante el concurso, los dos primeros puestos se encontraban muy a la par en puntos y se habían adelantado el uno al otro en múltiples ocasiones.

En cuanto al *feedback* sobre la retransmisión, fue mayoritariamente positivo. Algunos de los comentarios que se recibieron:

“He estado haciendo un repaso a las retransmisiones y han quedado fenomenal. ¡Menudo currazo los comentaristas! ¡Enhorabuena a todos los que habéis salido y a los que lo habéis hecho posible! ¡¡¡Nivelazo!!!”

“[Foto de un premio Óscar] Para todos los encargados de la retransmisión en directo. La he visto completa y está genial.”

En general en *feedback* fue muy positivo y distintos espectadores quedaron satisfechos con el resultado.

3.3. Diseño de un locutor basado en puntos de vista

Hemos hecho un análisis de locuciones de concursos, con las que hemos extraído comentarios habituales en este tipo de eventos. Lo aprendido lo hemos utilizado en nuestra propia locución, que nos ha hecho vivir en primera persona lo que es locutar un concurso, y cuáles son las fuentes de información para conseguir datos que transmitir.

Para poder crear un locutor automático tenemos que plantear una aproximación de alto nivel que nos permita generar, de forma modular, los comentarios a los sucesos más importantes que ocurren a lo largo del concurso, al igual que hacen los locutores humanos.

Un análisis en profundidad de los comentarios descritos en la sección 3.1 nos permitió identificar que, en muchas ocasiones, las frases dichas en la retransmisión eran sucesos asociados a entidades concretas del concurso. Por ejemplo había frases centradas en los equipos (“El equipo A ha resuelto su tercer problema”, “El equipo B se ha puesto en primera posición”, “El equipo C ha adelantado al equipo D”), frases centradas en los problemas (“El problema A ha sido resuelto por el $n\%$ de los equipos”, “El problema B está generando muchos envíos incorrectos”, ...) o frases relacionadas con el propio concurso (“Empieza el concurso”, “Los 10 primeros equipos tienen 3 problemas resueltos”, “Se congela el marcador”). Esto nos lleva a plantearnos lo que hemos denominado un “locutor basado en puntos de vista”.

Conceptualmente, un punto de vista viene a ser una entidad que se centra y opera en un dominio específico. Más concretamente, en un concurso de programación, se puede tener un punto de vista por cada equipo participante, problema, organización, etc. Dicho de otro modo, es una manera de observar el concurso desde un punto de vista específico para poder locutarlo con el

foco puesto en distintos sitios.

Así, la idea es trocear nuestro locutor en muchos locutores, donde cada uno está absolutamente volcado en fijarse qué es lo que ocurre alrededor de uno y solo uno de los elementos de la actividad que se retransmite. En nuestro contexto, habrá un elemento software que estará atento únicamente de lo que ocurre alrededor de, por ejemplo, cada uno de los equipos o cada uno de los problemas, para sacar a la superficie sucesos que le afecten, por muy intrascendentes que puedan parecer a otros elementos de la actividad. Esto genera una amalgama de comentarios de todo tipo, entre los que habrá luego que elegir cuáles son lo suficientemente importantes como para ser emitidos.

De cara al planteamiento del programa, la lógica se segmenta por cada punto de vista, lo que genera entidades especializadas en un ámbito concreto. Esto facilita la implementación al mismo tiempo que reduce la complejidad relativa que pueda tener el hecho de añadir comentarios nuevos al locutor, sin interferir con los ya existentes.

Como se menciona en la sección 2.3, las locuciones son subjetivas. Por eso es importante que el locutor automático se adapte y se pueda configurar según los intereses del espectador. La idea es que por un lado el locutor ofrezca una visión general del concurso y atienda los sucesos importantes, mientras que por otro lado pueda informar de forma personalizada sobre ciertos sucesos a los usuarios que puedan estar más interesados.

Con esto en mente, la finalidad de los puntos de vista es también producir comentarios parcialmente sesgados, lo que aporta carácter al locutor, y hace que no se perciba solo como un sistema totalmente automático. Por otro lado, permite a posibles usuarios del locutor ignorar comentarios que no les interesen por el motivo que sea. Esto es posible gracias a un sistema de etiquetado que funcionará en conjunto a los puntos de vista.

El locutor automático producirá entonces eventos interesantes, pero como un evento puede no parecerle interesante a todo el mundo, se utilizará el sistema de etiquetado para indicar que el evento es interesante si te interesa alguna de las etiquetas.

Un **evento interesante** se definirá entonces como un fragmento de salida del programa, que ha sido generado por un punto de vista, y está etiquetado en función de a quién pueda interesar. El programa generará eventos interesantes a lo largo del tiempo, durante el transcurso de un concurso de programación específico.

Para mostrar un ejemplo de funcionamiento a alto nivel, supongamos una situación en la que un equipo es el primero en resolver un problema, y lleva unos cuantos problemas resueltos en poco tiempo, lo que provoca un movimiento en la clasificación del concurso. Esta situación se puede interpretar de distintas maneras si se observa desde distintos puntos de vista:

- Desde el punto de vista del equipo, se podría decir que está en racha.
- Desde el punto de vista del problema resuelto se puede decir que es el primer equipo en resolverlo.
- Desde el punto de vista del marcador se puede decir que han ganado 3 puestos con la resolución de este problema.

3.4. Arquitectura dirigida por eventos

La arquitectura dirigida por eventos puede definirse, según Hugh Taylor (2008), como aquella que tiene la habilidad de detectar eventos y reaccionar inteligentemente a ellos. Esta arquitectura debe implementar una serie de funcionalidades para cumplir su objetivo. Estas funcionalidades son la capacidad de detectar eventos, mecanismos de transmisión para el intercambio de mensajes entre sus componentes, procesar la reacción al evento de entrada y mecanismos para iniciar la reacción al evento de entrada.

Las locuciones son, básicamente, una serie de comentarios surgidos en torno a los sucesos que tienen lugar durante una competición, los cuales son procesados y, posteriormente, narrados por un locutor. Por este motivo, una arquitectura dirigida por eventos es ideal para implementar nuestro sistema, que intenta reflejar el comportamiento de locutores humanos.

En este proyecto, podemos distinguir dos tipos de eventos: los eventos de entrada, los cuales vienen directamente del sistema CDS, y los eventos interesantes (explicados en la sección 3.3), que son los que constituyen la salida del programa.

Para ser más concretos, este sistema va a desarrollarse bajo el paradigma de la programación reactiva. La programación reactiva, o *Reactive Programming*, es un paradigma enfocado en el trabajo con flujos de datos, ya sean finitos o infinitos, de manera asíncrona, permitiendo que estos datos se propaguen generando cambios en la aplicación.

La programación reactiva está estrechamente relacionada con el patrón de diseño *Observer*. El patrón *Observer* es un patrón de diseño software en el cual un objeto llamado sujeto mantiene una lista de dependencias con otros, llamados observadores. Estos son notificados automáticamente cuando ocurren cambios. Este proyecto está basado en este patrón, ya que es la forma más lógica de mantener a las estructuras internas del programa siempre actualizadas con los eventos ocurridos en el concurso sin la necesidad de que cada uno de ellos acceda al *stream* de datos por su cuenta.

Utilizando programación asíncrona, se busca disminuir el uso ineficiente de recursos inactivos. Los nuevos datos se notifican a los objetos en vez de tener que solicitarlos. Por ello se invierte el diseño normal del procesamiento de entrada y salida. Este enfoque resulta en que los observadores solo trabajen cuando se realiza un cambio en los objetos observados, en vez de

estar constantemente pendientes de si han ocurrido cambios, reduciendo así la carga de trabajo.

Según el manifiesto reactivo¹⁰, los sistemas construidos como sistemas reactivos son más flexibles, con bajo acoplamiento y escalables. Son significativamente más tolerantes a fallos y cuando fallan responden con elegancia, y no con un desastre. Sus principales características, nombradas en dicho manifiesto, son:

- Responsivos: El sistema responde a tiempo en la medida de lo posible.
- Resilientes: El sistema permanece responsivo frente a fallos.
- Elásticos: El sistema se mantiene responsivo bajo variaciones en la carga de trabajo.
- Orientados a Mensajes: Los sistemas reactivos confían en el intercambio de mensajes asíncronos para establecer fronteras entre componentes, lo que asegura bajo acoplamiento y aislamiento.

3.5. Flujo de datos completo

El sistema está compuesto de varias partes, y todas funcionan simultáneamente, creando un flujo de datos constante. En esta sección se explica y detalla el flujo de datos de la aplicación.

Una descripción de todos los pasos que sigue el programa de forma repetida sería:

1. Llegada de un evento de la fuente de información del concurso, en este caso, el sistema CDS.
2. Conversión del evento de formato JSON a una estructura de datos interna.
3. Notificación a todos los puntos de vista del evento.
4. Cada punto de vista decide si ese evento es de relevancia o no para sus intereses.
5. Si un punto de vista decide que el evento es de relevancia, actualiza su estado interno, y si se cumplen ciertos requisitos, se genera un evento interesante (o evento de salida) desde el punto de vista de esa entidad.
6. El evento de salida se etiqueta en base a los datos que contenga.
7. Todos los eventos generados se reúnen y transmiten a distintas entidades:

¹⁰<https://www.reactivemanifesto.org>

- Una entidad que se encarga de mostrar los eventos interesantes por consola.
- Una entidad que se encarga de guardar todos los eventos generados durante la ejecución del programa en un log (único por ejecución).
- Un servicio web que emite en directo, a todos los clientes que le estén escuchando, los eventos interesantes.

Puede verse una representación del flujo de datos de la aplicación en la figura 3.1. Los elementos en gris no forman parte del programa como tal, pero deben existir para que el programa funcione correctamente.

En la siguiente sección se profundiza en el elemento del diagrama denominado “Clientes del locutor”.

3.6. Tecnologías utilizadas y entorno de trabajo

Las herramientas que se han escogido para este trabajo lo fueron en base a ciertas consideraciones, como son la arquitectura en sí, nuestra experiencia y las necesidades del proyecto exploradas a lo largo de este capítulo. Estas herramientas son:

- TypeScript: Elegir JavaScript ha sido una decisión consensuada, ya que es un lenguaje de programación que conocemos muy bien y que permite la implementación de funciones síncronas y asíncronas fácilmente, siendo una buena opción para implementar una arquitectura dirigida por eventos. El problema de JavaScript es que es débilmente tipado. En JavaScript no se comprueban muchos aspectos como que una propiedad a la que se intenta acceder exista o que un parámetro sea del tipo correcto, por lo que podíamos enfrentarnos a diversos errores en distintas áreas del programa. TypeScript es una forma de codificar en JavaScript como si fuese un lenguaje fuertemente tipado. En esencia, TypeScript añade una capa de compilación que en muchas ocasiones sirve para detectar errores de forma precoz. Por todos estos motivos, se decidió utilizar TypeScript como lenguaje principal para la implementación del locutor automático.
- RxJS: ReactiveX¹¹ es una API para programar de forma asíncrona el observado de *streams* de datos, siendo RxJS su implementación en JavaScript. Permite implementar el patrón Observer de una forma muy sencilla, que facilita las tareas de recibir los eventos del CDS o emitir eventos interesantes entre otras.

¹¹<https://reactivex.io>

El desarrollo de este locutor automático requiere de un entorno concreto. Para probar el sistema, se requiere tener acceso a un servidor CDS, el servidor CDS necesitaba acceso a un servidor CCS, y el servidor CCS (que en nuestro caso fue DOMjudge). Este último necesita acceso a un servicio específico, *judgehost* que se encarga de evaluar los envíos que se realizan al juez. DOMjudge y *judgehost* se instalaron de forma local, haciendo uso de máquinas virtuales. Adicionalmente se instalaron los servidores CDS, necesarios para probar el sistema, de distintas formas.

Para simular concursos con muchos envíos, se hizo uso de un script ya creado por parte de Marco Antonio Gómez. Este script se encargaba de simular envíos de un usuario aleatorio, a un problema aleatorio, con un veredicto aleatorio. Aunque nunca fuera a funcionar igual que un concurso real, este script hacía un trabajo lo suficientemente parecido como para que se pudieran probar el programa a gran escala, sin hacer muchos envíos manualmente.

Los problemas de prueba que se utilizaban en DOMjudge eran problemas triviales de resolver, ya que lo que interesaba eran los veredictos de los envíos y como afectaba eso al estado del concurso. De todos modos, para un problema de concurso trivial, se pueden forzar todo tipo de veredictos.

Todo el código del proyecto se ha ido implementando mediante el sistema de control de versiones git, concretamente haciendo uso de la plataforma GitHub¹². El repositorio en su totalidad puede encontrarse en la siguiente dirección <https://github.com/TFG-locutor/feed-interesante>.

Pincipalmente se ha utilizado VS Code¹³ como IDE. Y se ha utilizado el gestor de paquetes de node **npm**¹⁴ para manejar las dependencias con paquetes ya existentes y la instalación de TypeScript.

¹²<https://github.com/>

¹³<https://code.visualstudio.com/>

¹⁴<https://www.npmjs.com/>

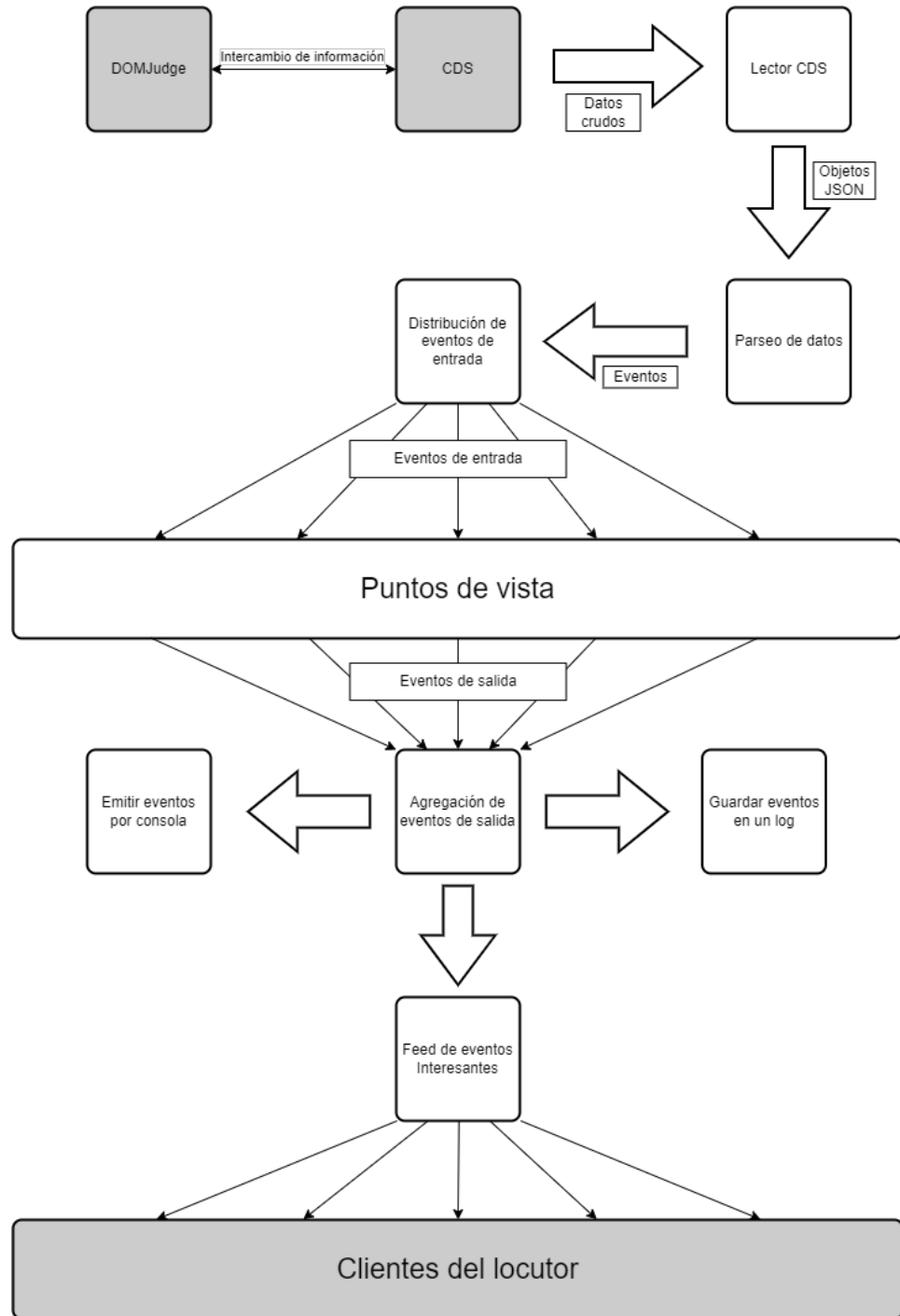


Figura 3.1: Diagrama que representa el flujo de datos de la aplicación, los elementos en gris no forman parte de la misma.

Capítulo 4

Implementación del sistema

En esencia, el sistema implementado actúa como un cliente que se conecta al servidor del concurso para recibir las notificaciones de los sucesos importantes (esencialmente envíos a los problemas) y que, a su vez, proporciona un servicio para que otros clientes se conecten para recibir eventos interesantes que pueden transformarse fácilmente en mensajes de locución. Para eso, vamos a utilizar Node como motor de ejecución del programa, y para el desarrollo utilizaremos TypeScript, un lenguaje que se transpila a JavaScript que es luego interpretado directamente por Node. Se va a utilizar el gestor de paquetes *npm* para gestionar los distintos paquetes que se utilizan en este programa.

Para comprender la implementación hay que diseccionar la forma en la que el programa consigue los sucesos del concurso que se va a locutar, lo que significa conocer los servicios web de los gestores del concurso. Además, hay que decidir cómo se cristaliza la implementación del diseño basado en puntos de vista que se describió en la sección 3.3 siguiendo la arquitectura basada en eventos (sección 3.4).

Todo el código asociado a este trabajo puede encontrarse en el siguiente repositorio de GitHub:

<https://github.com/TFG-locutor/feed-interesante>

4.1. Entrada del sistema

Como se trata de un locutor automático, la entrada del programa no puede depender del usuario. Toda la información que se puede necesitar se obtiene de manera automática, en este caso específico, haciendo uso del Contest API.

Como se menciona en la sección 2.2, el Contest API se ofrece tanto desde los programas CDS como CCS. Sin embargo, después de analizarlo y realizar

pruebas sobre ambos (utilizando DOMjudge como servidor CCS), se perciben diferencias entre las salidas de los dos programas. Las diferencias más significativas que se han encontrado son:

- El *feed* de eventos CCS incluye una marca de tiempo que indica el momento en el que ocurrió cada evento, en CDS no se incluye ninguna marca de tiempo.
- CDS emite eventos de tipo **state** que no se emiten en CCS, los eventos que se emiten de esta manera suelen ser muy muy importantes, lo que implica que en CCS hay que obtener esta información por otros medios.
- En algunos sistemas CCS se requiere autenticación para acceder al *feed* de eventos, en el CDS no se pide.
- El CCS en ocasiones emite eventos que no son necesariamente de conocimiento público, sin embargo el CDS no.

Estas diferencias, aunque se ajustan a la definición de Contest API, son lo suficientemente significativas como para que se requiera un programa especializado para cada una si se quiere poder leer de ambas. Además, desde el punto de vista de las herramientas alrededor de un concurso, el CDS es el servidor de datos al que se conectarían los locutores humanos para tener toda la información del concurso, así como la mayoría de las herramientas ya existentes mencionadas en el apartado 2.3. Por todos estos motivos, se ha decidido restringir el dominio de la aplicación para que solo pueda ser utilizada con servidores CDS.

El Contest API del CDS se utiliza para obtener toda la información que se necesite de dos maneras distintas: llamadas a los *endpoints* y lectura continua del *feed* de eventos.

Tanto al iniciar el programa como en momentos clave a lo largo de la ejecución de este, se hacen una serie de llamadas a distintos *endpoints* del CDS. Las distintas llamadas se describen a continuación:

- **problems:** *Endpoint* para conocer los problemas de un concurso. Si la llamada se hace antes de que el concurso empiece, la respuesta será vacía, y por lo tanto será necesario obtener estos datos a través del *event-feed* cuando empiece el concurso.
- **teams:** *Endpoint* para conocer los equipos que participan en un concurso.
- **organizations:** *Endpoint* para conocer las organizaciones representadas en un concurso. Las organizaciones están compuestas por uno o varios equipos participantes en el concurso.
- **scoreboard:** Es útil para obtener la ordenación actual de los equipos en el ranking y no tener que calcularla manualmente.

4.1.1. *Feed* de Eventos del CDS

El resto de datos que utiliza nuestra aplicación se obtienen del *feed* de eventos del Contest API.

El *feed* de eventos es esencialmente la enumeración, una a una, de todas las operaciones de creación, actualización o borrado que van ocurriendo a lo largo del tiempo en la mayoría de estructuras internas del CDS utilizadas para almacenar información del concurso, por lo general, las accesibles y/o modificables por el resto de *endpoints*.

Esto provoca que el *feed* de eventos contenga todos los eventos que han ocurrido a lo largo de la historia del concurso, incluyendo todos los eventos de configuración, creación de problemas, tipos de veredictos, creación de equipos, etc. Parte de esta información es redundante, ya que también puede obtenerse mediante llamadas a los *endpoints* anteriormente mencionados. Esta información repetida puede simplemente ignorarse.

La llamada al *feed* de eventos es continua. La respuesta consiste en un listado de todas las operaciones sobre el concurso relevantes hasta la fecha, y, una vez se ha llegado al momento actual del concurso, la petición no termina y devuelve los eventos según ocurran en el servidor. Este comportamiento hace que para leer del *feed* de eventos del Contest API sea necesario realizar una única petición, y no esperar a que termine, sino procesar de forma independiente cada paquete de datos que se reciba a lo largo del tiempo.

Los eventos que se reciben del *feed* tienen parte de su estructura común, y parte de su estructura variable en función del tipo de evento que se emita. El siguiente es un análisis de los distintos campos comunes presentes en cada evento:

- **id** (identificador): único para cada evento que se emite, puede ignorarse. La única información que puede aportar es el orden de emisión de los eventos al ser numérica secuencial. Sin embargo, esta información no es relevante, ya que se puede inferir de forma implícita debido a la estructuración del programa. Los eventos ocurren en el orden en el que se reciben.
- **op** (operación): contiene el tipo de operación que se hace sobre un objeto. Siempre suele ser “create” (creación de un nuevo objeto de cierto tipo) o “update” (actualización de un objeto que existía previamente)
- **type** (tipo): indica el tipo de objeto sobre el que se ha realizado la modificación. El valor de este campo determina la estructura del campo “data”.
- **data** (datos): contiene una estructura con toda la información relevante del objeto que se ha especificado en los otros campos. La estructura de este campo es variable en función del tipo de datos.

A continuación se muestra un análisis sobre lo que cada evento puede aportar a la aplicación en función de su tipo:

- El tipo **runs** hace referencia a la prueba de una solución enviada por un equipo a un problema, contra un único caso de prueba. La inmensa mayoría de eventos del *feed* suelen ser de creación de este tipo. Sin embargo, no es un evento que se tenga que tener en cuenta para este tipo de aplicación, ya que un veredicto se emite después de que todas las **runs** necesarias hayan ocurrido (un problema puede tener varios lotes de casos de prueba), y el veredicto final se comunica mediante un evento de un tipo distinto que sí interesaría tratar en este caso. De todos modos esta es información que no se ofrece al público u observadores del estado del concurso, porque puede dar pistas sobre la solución a un problema saber en qué lote de casos ha fallado exactamente. Por lo tanto, no es lógico procesar este tipo de eventos en el programa.
- El tipo **contests** hace referencia a los cambios que se hacen sobre un concurso en sí. Estos cambios no son muy comunes durante el transcurso de un concurso. Sin embargo, esto no quiere decir que no deban tratarse. Si se produce una actualización sobre los datos del concurso, puede ser de interés para las entidades del programa que calculen datos de forma relativa a datos del concurso. Por ejemplo, si el momento de congelado del marcador cambia durante el concurso por el motivo que sea, la información debe procesarse para que el locutor siga locutando de forma correcta y no tenga una representación interna del concurso que no refleja la realidad.
- El tipo **judgement-type** hace referencia a los tipos de veredictos que se pueden dar, pero no a los veredictos en sí. Lo más común es que se creen antes del inicio del concurso, y se mantengan igual. Puede asumirse que este tipo de datos no va a cambiar durante los concursos, y si lo hace, lo hará de forma excepcional. Este evento por lo tanto no se tiene que manejar internamente en el programa.
- El tipo **languages** hace referencia a los distintos lenguajes de programación en los que el juez admite las soluciones. Este tipo de datos es de naturaleza similar al anterior, y tampoco se debe manejarse internamente.
- El tipo **problem** hace referencia a los distintos problemas del concurso. No es común que se añadan o modifiquen problemas durante un concurso, pero conviene manejar este tipo de eventos para concursos dinámicos u otros casos no contemplados.
- El tipo **group** hace referencia a los distintos grupos que participan en un concurso. Este tipo de datos, al igual que los problemas, suele

configurarse previo inicio del concurso, y es extraño que se modifique durante el concurso. Pero debe soportarse para manejar eventos como el cambio de visibilidad de un grupo que, aunque poco comunes, ocurren de vez en cuando y afectan a la percepción del concurso.

- El tipo **organizations** hace referencia a las organizaciones a las que pertenecen los equipos. También se suele configurar antes del inicio del concurso y se utiliza una aproximación similar a la de los grupos.
- El tipo **teams** hace referencia a los distintos equipos que participan en un concurso. Suelen añadirse antes del concurso y es raro que se modifiquen mientras está en marcha. Al ser uno de los objetos más importantes, conviene manejar cualquier tipo de eventos de este tipo aunque no sean muy comunes.
- El tipo **state** es un tipo de evento que aparece pocas veces, pero para dar información que es muy interesante de cara a la locución, sucesos como pausas, reanudaciones, si el concurso ha terminado, etc. Este evento es esencial capturarlo y manejarlo adecuadamente.
- El tipo **submissions** hace referencia al intento de un equipo por resolver un problema específico. Este evento se emite en el momento en el que se hace el envío, no en el que se da el resultado. Por lo tanto, este tipo de eventos se va a seguir emitiendo cuando el marcador se encuentre congelado. Este evento es de los más importantes de capturar y comunicar a la mayoría de entidades del programa.
- El tipo **judgements** hace referencia al veredicto que emite el juez para un envío, una vez este se ha ejecutado contra los casos de prueba correspondientes. Este tipo de evento deja de emitirse después de que se congele el marcador, ya que solo se pueden saber los intentos de cada equipo, pero no los resultados. Es muy importante capturar y manejar este evento correctamente, pues se pueden considerar los eventos más esenciales de un concurso de programación, ya que este evento suele ser el detonante de muchos de los comentarios que realiza un locutor humano.
- El tipo **clarifications** hace referencia a las clarificaciones que ocurren entre los concursantes y los organizadores mediante el sistema de aclaraciones. Debido a que estas comunicaciones son, en la mayoría de los casos, de carácter privado, solo interesa comentar aquellas clarificaciones que estén dirigidas de los jueces a todos los participantes, porque se entiende entonces que son de interés general y pueden interesar a un locutor del concurso.
- El tipo **awards** hace referencia a los premios que se otorgan a los concursantes. Este tipo de evento no suele emitirse. Esto se debe a que por

lo general existe una ceremonia de entrega de premios que se celebra de forma ajena al juez, y por lo tanto los datos no llegan a procesarse nunca. De todas formas la competencia de este programa no incluye la locución de entregas de premios, sino el desarrollo del concurso en sí, por lo que este tipo de eventos puede ignorarse por completo.

4.2. Puntos de vista

Como se va a seguir un modelo basado en puntos de vista, el sistema no genera eventos interesantes mediante la aplicación constante de todas las comprobaciones sobre el estado del concurso, que habría que tener de forma interna. En su lugar se dispone de una serie de entidades donde cada cual gestiona su información y emite sus eventos. Esto resulta en un software más modular, así como más fácil de entender y modificar de cara al futuro.

El diseño basado en puntos de vista descrito en la sección 3.3 se materializa en la creación de distintas “entidades” o “puntos de vista”, donde cada una represente un foco de interés de cara a los usuarios finales del programa. Distintos ejemplos de puntos de vista pueden ser problemas, equipos, organizaciones, etc.

Cada entidad / punto de vista se encarga de gestionar la información que le interese e ignorar toda la que no sea de relevancia para ella. Por lo tanto no hace falta mantener una representación global y única del estado del concurso, lo que puede acarrear problemas si la estructura cambia en un futuro, si no que cada entidad será la encargada de mantener, de la forma que más convenga, solo la información que necesite.

Es también cada punto de vista quien se encarga de generar eventos interesantes que estarán relacionados consigo mismo. Por ejemplo, si el punto de vista “Problema H” generase un evento interesante como resultado de un cambio de su estado interno, se sabe que ese evento puede ser interesante para los usuarios de la aplicación interesados en el problema H en concreto, y se comunica a través del *feed* como tal mediante el sistema de etiquetado.

4.2.1. Modelo

Para trasladar la idea de puntos de vista a código se ha hecho uso de clases con la estructura mostrada en la figura 4.1. Se puede comprobar que existe una clase abstracta *PuntoDeVista* de la que heredan todos los puntos de vista. Cualquier punto de vista debe implementar al menos dos métodos: *filtrar*, que recibe un evento de entrada y decide si le interesa o no, y *actualizar*, donde un punto de vista recibe un evento que le interesa y actualiza su estructura interna. La lógica del método *actualizar* depende completamente del punto de vista. La única restricción que se impone es que cuando se quiera emitir un evento interesante se debe llamar al método *emitir* del padre,

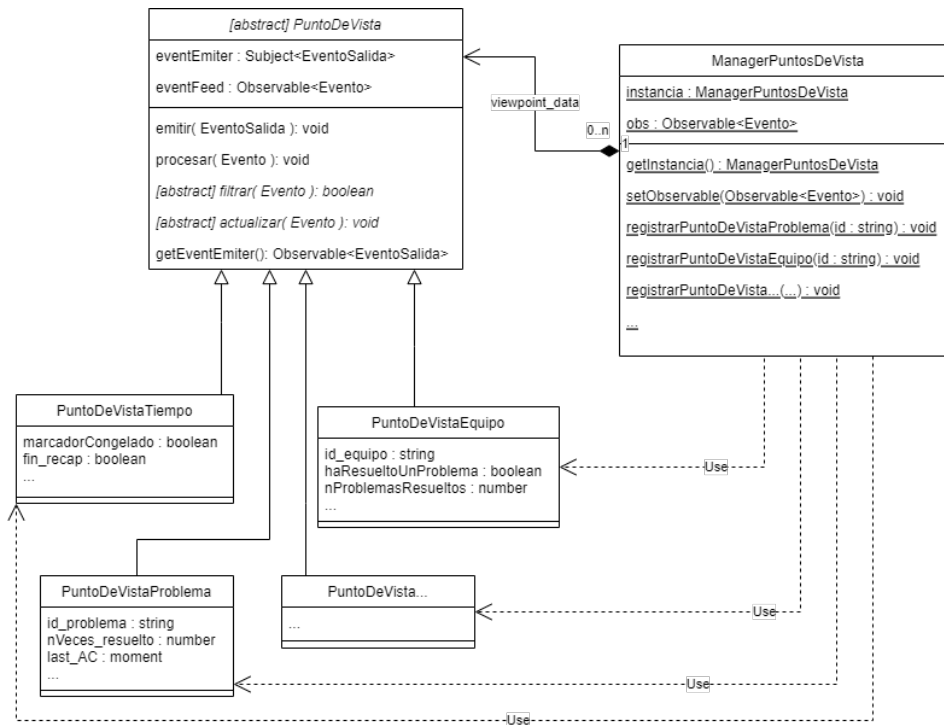


Figura 4.1: Diagrama de clase de los Puntos de Vista.

que se encarga de gestionarlo.

La clase *ManagerPuntosDeVista* implementa el patrón *Singleton*, y es la responsable de crear todos los puntos de vista, almacenarlos (en su atributo *viewpoint_data*), y engancharlos al flujo de eventos de entrada. Esta clase crea puntos de vista en el constructor, pero también tiene métodos para crear puntos de vista dinámicos, que no pueden crearse siempre de la misma manera porque varían en función del concurso, este puede ser el caso de los equipos, problemas u organizaciones.

En clases como *ManagerPuntosDeVista* es especialmente útil usar el patrón *Singleton*, ya que no tiene sentido que exista más de una instancia de esta clase en la aplicación. Esto también evita tener que pasar instancias de la clase entre muchos métodos de la aplicación, y que simplemente se llame cuando se necesita.

Los puntos suspensivos en las distintas partes del diagrama sirven para indicar que es un diagrama ampliable, concretamente:

- Cada punto de vista puede tener todos los atributos que necesite.
- Se pueden crear más puntos de vista siempre que hereden de la clase *PuntoDeVista*.

- Para añadir un punto de vista al *pipeline*: si es un tipo de punto de vista simple, basta con crearlo en el constructor de la clase *ManagerPuntosDeVista*. Si el tipo es dinámico, basta con implementar una función similar a las ya existentes en la clase *ManagerPuntosDeVista* y llamarla en el momento que se conozca la información.

4.2.2. Comportamiento de los puntos de vista

Para entender mejor el funcionamiento de los puntos de vista, se va a explicar su comportamiento en detalle.

Un punto de vista puede ser estático o dinámico. Nos referiremos a un punto de vista como **estático** cuando es el mismo para todos los concursos, es decir, el punto de vista existirá independientemente de la información específica de un concurso; un ejemplo de un punto de vista estático sería uno que solo se preocupe de eventos relacionados con el tiempo, como el momento de congelación. Por otro lado, nos referiremos a un punto de vista como **dinámico** cuando se instancia un número variable de veces en función de datos propios de cada concurso; un ejemplo de un punto de vista dinámico sería un punto de vista de un problema, este punto de vista se instanciaría una vez por cada problema que exista en el concurso que se quiere locutar, siendo cada instancia capaz de identificar el problema en el que poner el foco.

En el constructor de la clase *ManagerPuntosDeVista* se instancian los puntos de vista estáticos, ya que, debido a su naturaleza, no requieren ningún tipo de información externa. Los puntos de vista dinámicos, por otro lado, no pueden instanciarse en el constructor porque necesitan información propia del concurso, y no se dispone de esta información en el momento de construir la clase *ManagerPuntosDeVista*. Por este motivo cada tipo dinámico de punto de vista necesita un método personalizado dentro de *ManagerPuntosDeVista*. Este método será el encargado de instanciar ese tipo dinámico con la información relevante en cada caso, y puede llamarse cuando se disponga de la información necesaria.

Cada punto de vista puede contener todos los atributos que le hagan falta para almacenar información del concurso. De esta forma se abren infinitas posibilidades en cuanto a qué información almacenará y qué estructuras de datos utilizará para hacerlo. La funcionalidad de cada punto de vista está segmentada en distintos métodos con el objetivo de facilitar tanto su entendimiento como su programación o modificación.

Cuando un punto de vista recibe un evento, lo primero que se hace es ver si el evento es de relevancia para el punto de vista en cuestión. Esto se lleva a cabo mediante el método *filtrar*. Este método es el encargado de discriminar eventos que no aportarían nada a la visión del concurso que este punto de vista representa, es decir, eventos que no resultarían en la actualización de ninguna estructura de datos interna ni en la emisión de un evento interesante.

La parte de un punto de vista encargada de procesar la información de un evento es la función *actualizar*. Los eventos que aquí se procesan son exclusivamente los que no han sido discriminados por la función *filtrar*. Procesar la información implica dos cosas: por un lado se actualizan las estructuras de datos internas del punto de vista con la última información del concurso que ha proporcionado el evento; por otro lado cada punto de vista es responsable de valorar si el cambio de los datos ha sido suficientemente relevante como para emitir un evento interesante. Valorar si un cambio de datos es relevante se traduce en el código en forma de distintas condiciones que varían en complejidad. Estas valoraciones son realmente las que, de forma indirecta, determinan la frecuencia con la que se emitirán estos eventos interesantes durante un concurso, y por lo tanto, los primeros parámetros que hay que modificar para ajustar esta métrica.

Para emitir un evento interesante se hace uso del método *emitir*, que recibe el evento interesante a emitir. Antes de llamar a este método, debe construirse el objeto *EventoSalida* correspondiente, lo que implica tener acceso a toda la información que haga falta y etiquetarlo debidamente. Todo punto de vista es responsable de guardar la información que necesite incluir en los eventos interesantes que él mismo genere, de asignarle una prioridad, y de etiquetar cada uno con todas las etiquetas que hagan falta. Por ejemplo, si se genera un evento que involucra a un equipo y problema concretos, el nombre del problema y del equipo se encontrarán entre las etiquetas.

Cuando se emite un evento, este pasa al *pipeline* que lo conduce a la parte final del programa, que se explica a continuación.

4.3. Salida de datos

El resultado del trabajo realizado por todos los puntos de vista es un flujo continuo en el tiempo de eventos interesantes, tal y como se explica en la sección 4.2. Para gestionar estos eventos, y que puedan convertirse en la salida del locutor automático, se explica a continuación el modelo que se ha seguido, así como el funcionamiento de las distintas salidas de datos.

Un programa puede tener multitud de salidas de datos, y, en el caso del locutor automático, se ha diseñado e implementado con esta idea en mente. Se ha utilizado la estructura de clases mostrada en el diagrama de la figura 4.2 para poder implementarlo con distintas salidas aisladas de datos.

Como se puede ver en el diagrama, *EventHandler* es una clase abstracta que representa un flujo de salida de eventos interesantes del programa. Las distintas clases que heredan de *EventHandler* son implementaciones concretas de acciones que llevar a cabo con un evento interesante. Al ser clases independientes, los detalles de la implementación de cada una quedan aislados del resto de clases, lo que hace el código más modular. Como resultado, cada forma de tratar los datos puede especializarse en su tarea específica, y

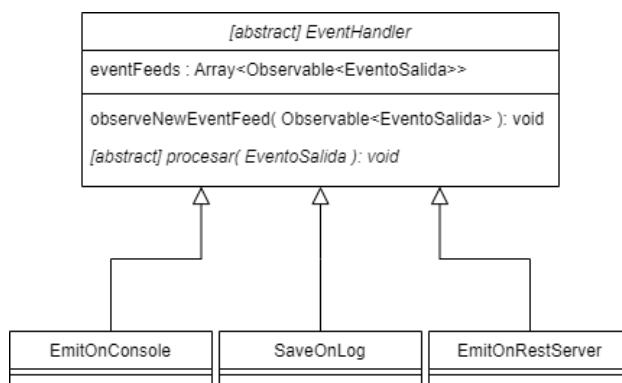


Figura 4.2: Diagrama de clase de la emisión de eventos.

no preocuparse de la forma en que estén tratando los datos el resto de clases que heredan de *EventHandler*.

Tanto el atributo *eventFeeds* como el método *observeNewEventFeed* se utilizan en la construcción de estas clases para poder añadirse al *pipeline* de RxJS al iniciar el programa. Esto significa que las distintas clases que hereden de *EventHandler* no tienen por qué preocuparse de la existencia de estos métodos, reduciendo su único requerimiento a implementar el método *procesar*.

El método *procesar* es un método que recibe un evento interesante ya generado y listo para ser distribuido. Cuando se recibe un evento interesante mediante este método, cada una de las clases que heredan de *EventHandler* puede tratarlo de la forma que más le convenga. En la versión actual del locutor automático se han implementado 3 formas de manejar los eventos interesantes.

La clase ***EmitOnConsole*** se encarga de escribir los eventos interesantes que reciba por consola. Su funcionamiento es simple, pero puede ser útil para hacer pruebas del sistema antes de desplegarlo o como la forma más directa de saber cuál es la salida del programa en todo momento.

La clase ***SaveOnLog*** es la encargada de guardar todos los eventos interesantes que se generen durante la ejecución del programa en un archivo *log*. Este *log* se crea al inicio del programa, con un nombre único que incluye el momento de ejecución. Luego se van escribiendo en el archivo todos los eventos interesantes que se van produciendo hasta que finalice el programa. Para implementar esta clase se ha hecho uso del paquete `log4js`¹. Un ejemplo de una de las entradas que esta clase genera se muestra en la figura 4.3.

La clase ***EmitOnRestServer*** se puede considerar el manejador de eventos más importante, y se explica a continuación.

¹<https://www.npmjs.com/package/log4js>

```
[2022-04-28T04:08:02.084] [DEBUG] default - EventoSalida {  
  id: 1,  
  message: 'Ha terminado el concurso',  
  prioridad: 4,  
  tags: [ 'time' ],  
  info: {},  
  time: '2022-04-28T04:08:02+02:00',  
  type: 'contest_end'  
}
```

Figura 4.3: Ejemplo de una de las entradas del log que se generó como resultado de la ejecución del programa durante un concurso.

4.3.1. Servidor de eventos interesantes

Aunque el modelo que utiliza la clase *EventHandler* ofrece la posibilidad de implementar virtualmente cualquier manejador de eventos, este acercamiento tiene un inconveniente. Si una persona quiere ampliar la funcionalidad del locutor automático, está “obligada” a entender su funcionamiento interno y a implementar al menos parte de esta funcionalidad en Node.

Ofrecer estos eventos interesantes como un servicio web soluciona este problema. Este acercamiento desacopla casi por completo a los clientes del locutor automático, y consigue que el locutor automático pueda percibirse como una caja negra que, una vez configurada, emitirá eventos interesantes durante la celebración de un concurso. De esta forma no hace falta que ningún cliente del locutor automático conozca el funcionamiento interno de este, no limita los lenguajes en los que los programas cliente pueden programarse, y se da servicio a un número variable de clientes desde un mismo locutor automático.

La clase *EmitOnRestServer* es precisamente la encargada de ofertar este servicio web. En el constructor, esta clase crea un servicio web que consiste en un único *endpoint* al que se accede mediante la ruta principal de este servicio web. Este *endpoint* es el encargado de ofrecer a los clientes los eventos interesantes que ocurran durante el concurso. Esto se hace mediante un sistema de *stream* de datos, similar al utilizado por el *feed* de eventos del Contest API.

Cuando un cliente se conecta al *endpoint* el servidor se encarga de comunicar al cliente todas las cabeceras necesarias, donde se incluye toda la información que hace falta para establecer este tipo de comunicación continua. Después, la conexión no se termina, en su lugar se deja abierta y se añade a una colección de conexiones abiertas. Finalmente se comunica al

cliente que ha empezado a escuchar del *feed* con éxito.

Cuando *EmitOnRestServer* recibe un evento interesante mediante el método *procesar*, lo que hace es comunicárselo a todas las conexiones que se encuentren activas actualmente, de esta manera todos los clientes que estén escuchando en ese momento recibirán la misma información. De forma periódica se comprueba si alguna de las conexiones activas se ha interrumpido para eliminarla de la colección. Cuando el concurso que se está locutando termina, o el programa se detiene, todas las conexiones activas se cierran.

Desde el punto de vista del cliente, este acercamiento solo supone tener que hacer una única petición al locutor automático para suscribirse al *feed* de eventos interesantes, y, cada vez que se reciban datos, tratarlos de la forma que se desee.

4.3.2. Formato de salida

Se puede pensar que la salida de nuestro sistema tenga que consistir en texto “plano” listo para ser locutado, es decir, que la salida sea similar a los comentarios que haría un locutor humano textualmente. Este acercamiento tiene algunos problemas. En primer lugar, se crea una limitación basada en el idioma, restringiendo así el uso del programa a usuarios que compartan el idioma por defecto. Por otro lado, si ya se crean las frases listas para locutar, se pierde capacidad creativa de cara a locutar un concurso de maneras específicas. También se limitan las locuciones a cosas que se podrían hacer con la “frase” en vez de con los “datos”. Sin embargo, este tipo de aproximación puede hacer el trabajo de los clientes del programa más sencillo en ocasiones, por lo que se ha decidido incluir entre los datos de salida un mensaje ya compuesto que puede ignorarse si se desea.

La salida del locutor automático cumple distintos requisitos. En primer lugar, dispone de toda la información que pueda hacer falta para locutar un fragmento del concurso, limitando así la carga de los usuarios del programa que no utilicen las frases ya generadas, a leer los datos y formatearlos adecuadamente. En segundo lugar, el idioma no es un impedimento, se puede identificar el tipo de evento y las entidades involucradas en el mismo para posteriormente formatear esa información como se desee, sin importar el idioma. Por último, se pueden diferenciar fácilmente las etiquetas asociadas a un evento de la salida.

La salida del programa se hace en formato **json**², que permite todo lo anteriormente mencionado, y que ya es utilizado en multitud de programas como un formato de texto sencillo para el intercambio de datos.

Como los eventos pueden ser muy variados, y por lo tanto tener estructuras distintas, se utiliza un acercamiento similar al del Contest API, donde todos los eventos tienen parte de la estructura común y parte de la estruc-

²<https://www.json.org/json-en.html>

tura variable en función del tipo del evento en concreto. Este tipo puede conocerse porque viene especificado en uno de los campos comunes.

A continuación se muestra un ejemplo de uno de los eventos que generó el programa durante su ejecución durante un concurso.

```
1 {
2   "id": 8,
3   "message": '¡El equipo BoSSH0 Team ha resuelto su
4     primer problema!, Cinquecento',
5   "prioridad": 2,
6   "tags": [ 'BoSSH0 Team', 'AC', 'Cinquecento', '
7     SalesianosLog' ],
8   "info": {
9     "team": 'BoSSH0 Team',
10    "organization": 'CEIP Salesianos Los Bosc0s',
11    "problem_label": 'C',
12    "problem_name": 'Cinquecento',
13    "first_problem": true,
14    "attempts": 1
15  },
16  "time": '2022-05-09T12:43:45+02:00',
17  "type": 'accepted_answer'
```

El programa cliente que recibe estos datos tiene varias opciones:

- Puede acceder al campo *message*, y locutar el texto tal cual.
- Puede ignorar el campo *message* utilizar el resto de campos para locutar una frase distinta, por ejemplo: “BoSSH0 team just solved problem C! It’s their first problem which was also solved at the first try!”
- Puede hacer algo completamente distinto con los datos que se le dan, no existen limitaciones en lo que se puede hacer.

De esta manera cada programa cliente es responsable de lo que locuta, y el *feed* de eventos interesantes procura aportar toda la información que los clientes puedan necesitar.

4.4. Implementación completa

Durante el proceso de desarrollo del locutor automático se han acabado implementando 6 puntos de vista del locutor distintos, y 2 clientes.

4.4.1. Puntos de Vista implementados

Los distintos puntos de vista que se han implementado siguiendo la estructura descrita en la sección 4.2 son los siguientes.

- El punto de vista del **tiempo** es un punto de vista estático que controla los distintos momentos que dependen del paso del tiempo en un concurso. Estos eventos son el inicio del concurso, el momento de congelación del marcador y el momento de finalización. Se generan eventos interesantes cuando llega la hora de cualquiera de estos eventos, en los momentos previos a los mismos (por ejemplo que queden 5 minutos para que acabe el concurso), o cuando se superan ciertos límites de tiempo establecidos como por ejemplo que ya haya transcurrido la mitad del concurso.
- El punto de vista *scoreboard* es un punto de vista estático que se encarga de detectar los cambios de posición en el ranking global, y de comparar los primeros puestos. Este punto de vista mantiene internamente una representación del ranking y gestiona la variación de las posiciones que sufren los equipos a lo largo del tiempo.
- El punto de vista de un **problema** es un punto de vista dinámico. Este punto de vista se instancia una vez por cada problema que tenga el concurso, y guarda información como cuántos equipos lo han resuelto, el último momento en el que se resolvió, los intentos que cada equipo ha hecho, etc. Este punto de vista se encarga de generar eventos interesantes como saber quién es el primer equipo en resolver un problema, o detectar quién está teniendo problemas para resolverlo.
- El punto de vista de un **equipo** es dinámico, y se instancia una vez por cada equipo que participe en el concurso. Este punto de vista gestiona información como los problemas que ha resuelto un equipo o los intentos de cada problema para generar eventos interesantes como la primera vez que un equipo resuelve un problema, o si ha resuelto muchos problemas a la primera.
- El punto de vista de un **grupo** es dinámico. Este punto de vista se instancia una vez por cada grupo de usuarios que participen en el concurso. Un grupo puede considerarse una división artificial de los equipos que participan en un concurso, y solo existirá si se ha configurado previamente en el servidor del concurso. Un ejemplo de un grupo podría ser “Participantes de primero”. El punto de vista de un grupo mantiene una representación interna del ranking que solo contiene a los equipos pertenecientes a ese grupo, y genera eventos interesantes como los cambios de posiciones que tiene lugar entre los miembros.



Esto es un evento interesante de prueba emitido por el *player*

Figura 4.4: Demostración del rótulo generado por el *player*.

- El punto de vista de una **organización** es también dinámico, y se instancia una vez por cada organización representada en un concurso. Su funcionamiento es muy similar al punto de vista anterior, pero en vez de por grupo, agrupa a los equipos participantes por organización.

4.4.2. Clientes del locutor

Los clientes pueden entenderse como programas independientes que utilizan el servicio que ofrece un locutor automático para obtener eventos interesantes, luego cada cliente trata estos eventos de la forma que desee.

Se han implementado dos clientes del locutor automático que utilizan el endpoint de eventos interesantes explicado en la subsección 4.3.1.

Uno de los clientes implementados es un *player*. Este cliente desarrollado como una aplicación web portátil, puede utilizarse como una fuente en programas de retransmisión como puede ser OBS³. El *player* consiste en una página web que se conecta al *feed* de eventos de un locutor automático. Cuando recibe un evento interesante se extrae la información de este para crear un mensaje que pasa a una cola de mensajes que quedan por mostrar. El cliente comprueba periódicamente si quedan eventos en la cola por mostrar, y si es el caso, va mostrando los mensajes pendientes por pantalla y eliminándolos de la cola.

El *player* dispone de un mecanismo para reconectarse automáticamente al locutor automático en caso de que la conexión se interrumpa. Esto puede ocurrir por varios motivos, como que haya un error de red o que el servidor de eventos interesantes deje de funcionar. En cualquier caso esta reconexión se hace “en las sombras”, de modo que no suponga una molestia para los potenciales espectadores. Un ejemplo de un rótulo mostrado por el *player* puede encontrarse en la figura 4.4

El otro cliente implementado es un **bot de twitter**. Este programa, desarrollado en Node utiliza el API de Twitter⁴ para poder publicar tweets desde una cuenta específica de forma automática.

Para su funcionamiento, este cliente se conecta al *endpoint* que ofrece el locutor automático, y va recibiendo distintos eventos interesantes a lo largo

³<https://obsproject.com/>

⁴<https://developer.twitter.com/en/docs/twitter-api>

del tiempo. Luego transforma los datos de estos eventos en mensajes que poder publicar, estos mensajes se pasan a la parte del cliente encargada de publicarlos. Para publicar un mensaje, hemos hecho uso de un paquete de Node⁵ que facilita la conexión con el API de Twitter.

En los dos clientes implementados es posible que no se deseen mostrar todos los eventos interesantes que se reciben. Para poder filtrar estos eventos, se puede hacer uso del sistema de prioridad y etiquetado. Su origen viene explicado en la subsección 4.2.2.

Cada evento interesante que emita el locutor automático vendrá acompañado tanto de una prioridad como de una lista de etiquetas para que se pueda determinar de forma rápida y eficiente si interesa tratar con cada evento.

La prioridad que el locutor asigna a un evento consiste en un número entero que va desde el 0 al 4 inclusive, siendo 0 la prioridad más baja y 4 la prioridad asignada a los eventos que más interés locutar. A continuación se muestra una idea general de que puede esperarse de cada prioridad:

- **0** es equivalente a una prioridad mínima. Son eventos que a casi nadie le interesan, en esta categoría entran eventos como los resultados de envíos sin información adicional.
- **1** es equivalente a una prioridad baja. Son eventos que pueden interesar a poca gente. Por ejemplo un veredicto que ha resultado en un pequeño cambio en la parte baja del marcador.
- **2** es equivalente a una prioridad media. Se asigna a eventos que interesan a más o menos gente en función del contexto, como por ejemplo cambios más relevantes en el *scoreboard*.
- **3** es equivalente a una prioridad alta. Estos eventos es probable que interesen a mucha gente. Un ejemplo de eventos con esta prioridad serían las primeras veces que se resuelve un problema o que un equipo se ha puesto en primera posición.
- **4** es equivalente a una prioridad máxima. Estos eventos le interesan a todo el mundo, suelen ser partes troncales del concurso y no ocurren muy a menudo. Es recomendable locutar siempre estos eventos. Algunos ejemplos son: el inicio del concurso, el primer problema resuelto o el momento de congelar el marcador.

Las etiquetas que acompañan a un evento interesante son en esencia indicadores de todas las áreas que toca. El atributo `tags`, incluido en los eventos emitidos, es una colección de cadenas de texto con estos indicadores. Por ejemplo un evento que afecte al equipo “SVO” (que es de la organización

⁵<https://www.npmjs.com/package/twitter-api-v2>

“UCM”) y al problema “Ordenando el armario”, tendrá incluidas entre sus etiquetas “SVO”, “UCM” y “Ordenando el armario”.

A efectos prácticos, un cliente que quiera filtrar los eventos interesantes solo tiene que incluir en el código del programa condiciones simples que miren la prioridad del evento y/o las etiquetas que tenga. Por ejemplo un cliente puede querer locutar todos los eventos de prioridad 2 o superior, mientras que otro querrá locutar todos los eventos de prioridad 1 o superior, pero solo si tienen una etiqueta específica. Este sistema minimiza el esfuerzo necesario para implementar nuevos clientes.

4.4.3. Ejemplo de funcionamiento

Para mostrar el funcionamiento del sistema entero, se va a describir toda la secuencia de acciones que se llevarían a cabo, así como los roles de cada parte del sistema, en dos situaciones que ocurren secuencialmente:

- En un hipotético concurso de programación (que utiliza un CDS) tiene lugar el siguiente suceso: el equipo A, que pertenece a la organización B, envía una solución al problema X y no lo consiguen resolver. Esto produce que el *feed* de eventos del CDS produzca un evento que refleje esta información parcialmente.
- El evento es capturado por nuestro sistema, y se transforma de un objeto JSON a un objeto con estructura dentro del locutor automático. El evento se enriquece con datos que se han ido capturando anteriormente como puede ser el nombre del problema o el nombre de la organización. Estos datos no han formado parte del evento que se ha leído del CDS necesariamente.
- Este evento pasa por los filtros de cada punto de vista (método *filtrar*) y es ignorado por muchos de ellos, sin embargo una serie de puntos de vista indican que este evento les interesa. Estos puntos de vista son: el del *scoreboard*, el del problema X, el del equipo A y el de la organización B.
- El evento luego se pasa al método *actualizar* de los puntos de vista que han indicado que estaban interesados, esta vez con la intención de que procesen su información, actualicen sus estructuras internas de datos, y emitan un evento interesante si lo consideran oportuno. Cada uno de los 4 puntos de vista procesa el evento a su manera:
 - El punto de vista del *scoreboard* registra este evento, como no se producen cambios en el marcador no genera ningún evento interesante.

- El punto de vista del equipo A guarda la información del intento fallido y tampoco genera ningún evento interesante.
 - El punto de vista de la organización B registra este evento, como tampoco se han producido cambios en su representación interna del marcador, no se genera ningún evento interesante.
 - El punto de vista del problema X guarda también la información del intento fallido y no genera ningún evento interesante.
- Unos minutos después, el mismo equipo A, envía otra solución al problema X, y esta vez lo consiguen resolver. El *feed* de eventos del CDS produce otro evento como consecuencia.
 - Este evento también es capturado por nuestro sistema. Lo transforma y enriquece de forma análoga al evento anterior.
 - Los mismos puntos de vista que antes indicaron que están interesados en el evento lo vuelven a hacer, y cada uno lo vuelve a procesar a su manera:
 - El punto de vista del *scoreboard* registra este evento, sin embargo el equipo no cambia de posición, por lo que no se genera ningún evento interesante.
 - El punto de vista del equipo A guarda la información del acierto, no es su primer problema resuelto, y por tanto no genera ningún evento interesante.
 - El punto de vista de la organización B registra este evento, como tampoco se han producido cambios en su representación interna del marcador, no se genera ningún evento interesante.
 - El punto de vista del problema X detecta que es la primera vez que un equipo resuelve el problema, y genera un evento interesante que transmite haciendo uso del método *emitir*.
 - El evento interesante generado contiene una frase lista ya montada “¡El equipo A es el primero en resolver el problema X!”. También contiene todos los datos que le puedan hacer falta a un locutor como pueden ser el nombre del problema, del equipo, la organización, etc. El evento tiene prioridad 3 y contiene etiquetas del equipo A, la organización B, y el problema X.
 - El evento interesante se distribuye entonces entre todas las salidas de datos del programa, cada una hace su función:
 - *EmitOnConsole* saca la información del evento interesante por consola.

- *SaveOnLog* guarda el evento interesante en el log que corresponda a esa ejecución del programa.
 - *EmitOnRestServer* comunica a todos sus clientes activos que ha tenido lugar un evento interesante.
- En este momento del concurso hay dos clientes escuchando el *feed* de eventos interesantes que reciben estos datos:
- Un cliente es un bot de Telegram interesado en locutar todo lo que tenga que ver con la organización C o sea muy importante. Este cliente captura los datos, y mediante una comprobación rápida verifica que el evento que le ha llegado no contiene la etiqueta 'C', también comprueba que la prioridad del evento es inferior a 4. Como consecuencia, este cliente ignora el evento.
 - Otro cliente es un *player* conectado a una retransmisión que está interesado en mostrar rótulos de todos los eventos que tengan prioridad 3 o superior. Este *player* captura el evento, mediante una comprobación rápida se verifica que cumple con la restricción de prioridad, y se genera un rótulo. Los espectadores de la retransmisión pueden ver un rótulo que les informa de que el equipo A es el primero en resolver el problema X.

4.5. Extensibilidad del sistema

Todo el sistema ha sido construido con la idea de ser extensible, y en general, altamente modular. Esto se traduce en dos cosas:

Por un lado, la lógica del locutor automático se encuentra segmentada en el código, lo que permite acciones como su hacer modificaciones de forma precisa, o ampliar con nuevas funcionalidades. Por otro lado simplifica los requisitos necesarios para introducir nuevas partes del sistema que sigan un modelo similar a las ya existentes. Lo que significa que se pueden añadir nuevos puntos de vista, nuevas salidas de datos o nuevos clientes del locutor sin modificar la estructura base del programa, y sin necesidad de entender su funcionamiento completo.

A continuación se muestran los requisitos necesarios para añadir ciertos nuevos tipos de partes al sistema.

4.5.1. Nuevos puntos de vista

Si se desea que el sistema utilice un nuevo punto de vista, se debe crear una nueva clase que herede de *PuntoDeVista*, y seguir los siguientes pasos:

- El constructor de la clase debe recibir un objeto de tipo *Observable<Evento>* que debe pasar como único argumento al constructor del padre. Este objeto representa el *pipeline* de eventos de entrada.
- La clase debe implementar un método *filtrar* que reciba un objeto de tipo *Evento*. Esta función deberá retornar un booleano que indique si el evento recibido es de interés o no para el punto de vista. La lógica que se implemente en este método dependerá por completo de los que se quiera hacer con el punto de vista en cuestión.
- La clase debe implementar un método *actualizar* que reciba un objeto de tipo *Evento*. Esta función será la encargada de actualizar sus estructuras de datos internas, y comprobar si los cambios supondrían la emisión de un evento interesante. La lógica de esta función será completamente distinta dependiendo del punto de vista. La función no devuelve nada.
- Dependiendo de si el punto de vista va a ser estático o dinámico (como se explica en la sección 4.2.2), la clase se instanciará de formas distintas, en ambos casos dentro de la clase *ManagerPuntosDeVista*:
 - Si el punto de vista es estático, puede instanciarse en el constructor de la clase *ManagerPuntosDeVista*, pasando entre los argumentos del constructor el atributo *obs*.
 - Si el punto de vista es dinámico, será necesario crear un nuevo método que reciba los datos necesarios para crear una instancia de la clase. El número y tipo de parámetros de esta función variará en función del punto de vista que se quiera implementar. Luego, esta función debe invocarse desde cualquier punto del programa en el que se disponga de los datos necesarios para hacerlo. Esto será posible gracias a que *ManagerPuntosDeVista* utiliza el patrón *singleton*.

Una vez cumplidos estos requisitos, el punto de vista estará ya integrado en el programa. El nuevo punto de vista creado puede ahora contener todos los atributos y lógica que se desee.

Para emitir un evento interesante hay que utilizar el método *emitir* de la clase padre, este método recibe un parámetro de tipo *EventoSalida* que debe construirse con todos los parámetros necesarios: mensaje ya construido, prioridad del evento, etiquetas del evento, información del evento necesaria para su locución, momento de emisión y un identificador de tipo único.

4.5.2. Nuevas salidas de datos

Si se desea que el programa utilice una salida de datos adicional, será necesario crear una nueva clase que herede de *EventHandler*. Esta clase debe

cumplir los siguientes requisitos:

- La clase debe implementar un método *procesar* que reciba un único argumento de tipo *EventoSalida*. En este método se recibirán todos los eventos interesantes que generen todos los puntos de vista durante la ejecución del programa, y se pueden tratar de la forma que más convenga.
- La clase debe instanciarse al inicio del programa junto al resto de salidas de datos. Para añadir la salida de datos correctamente al *pipeline* se debe seguir el mismo formato que se sigue para el resto, actualmente todas las salidas de datos se añaden al pipeline en el archivo `index.ts`.

4.5.3. Nuevos clientes del locutor

Los dos tipos de ampliaciones anteriores requerían modificar el código del locutor automático. Sin embargo, para crear un nuevo cliente del sistema, el locutor automático puede tratarse como una caja negra, sin que sea necesario saber como funciona, solo lo que emite.

Un cliente del locutor puede estar programado en cualquier lenguaje que permita hacer peticiones a servicios web, ya que para utilizar el *feed* de eventos interesantes solo es necesario conectarse al servidor en el que esté desplegado por el puerto correspondiente. Todas las instrucciones y detalles a tener en cuenta sobre el proceso de desplegado están expuestos en el apéndice A.

Por tanto, el único paso necesario para crear un cliente del locutor automático es que se haga una petición web al lugar donde esté desplegado el sistema en su ruta base. Después de haberse conectado exitosamente al *feed*, el cliente no tiene que esperar a que la petición termine (ya que nunca acabará de forma natural), en su lugar hay que procesar cada paquete de datos que llega a lo largo del tiempo de forma independiente.

Los datos que recibirá el cliente se encontrarán en formato JSON, como se explica en la subsección 4.3.2. Después de parsear los datos, se puede hacer con ellos lo que se quiera.

Capítulo 5

Evaluación

El proceso de evaluación del sistema se ha llevado a cabo mediante su prueba en concursos de programación reales, comprobando de esta manera la viabilidad de utilizar el locutor automático en los tipos de situaciones que estaba diseñado para manejar. Se realizaron dos evaluaciones distintas.

5.1. Evaluación preliminar

La evaluación preliminar del sistema tuvo lugar el día 7 de abril de 2022 y se realizó sobre el concurso AdaByron Multisede 2022¹. Este concurso se realizó online debido a la naturaleza del mismo, por lo que la prueba se pudo realizar desde los entornos de prueba que ya teníamos preparados. El concurso tenía una duración de 3 horas, y solo podían participar las provincias españolas que no tuvieran su propio concurso regional.

La prueba del programa fue privada, es decir, aunque los datos del concurso eran públicos, la salida del programa no se compartió. Esto es debido a que se querían probar distintas cuestiones antes de la siguiente evaluación, que sí que sería de carácter público.

En el momento de hacer esta prueba no estaba implementado aún el sistema de prioridades o etiquetado, lo que significaba que la salida del programa no podía filtrarse. Uno de los objetivos principales de esta prueba fue observar detenidamente toda la salida del programa para ver que eventos se emitían con demasiada frecuencia y como de interesante era cada evento en el momento en el que se generaba. De esta manera la frecuencia y prioridad de los eventos generados podría ajustarse de cara a la evaluación final.

Otro de los objetivos principales de esta prueba fue pensar en nuevas ideas eventos interesantes que nos gustaría incluir en el locutor automático. Observar de forma tan directa todo lo que ocurre en un concurso, mientras

¹<https://ada-byron.es/2022/reg/multisede/>

observamos nuestra locución automática, hace que se surjan ideas de que nos gustaría ver comentado en ese cada momento del concurso.

Momentos antes del inicio del concurso, uno de los organizadores nos indicó donde estaba situado el CDS que se utilizaría para la competición, y se procedió a configurar el programa con estas propiedades. Se identificó un error con el método de autenticación que no se había tenido en cuenta. También se detectó un error derivado de la diferencia en el tamaño de los paquetes que se transmiten por internet, que era distinto al tamaño de los paquetes entre algunos servidores locales. Estos errores hacían que el programa no pudiera iniciarse, pero se solucionaron antes del inicio del concurso.

Una vez iniciado el concurso, y con varias instancias del programa funcionando, se detectó un error en la obtención de datos por el cual no se llegaba a obtener el nombre de los equipos participantes. Este error se solucionó más o menos a la media hora de concurso, lo que provocó que los eventos interesantes que generó el programa en ese intervalo de tiempo carecieran del nombre de equipo.

Durante el resto del concurso no se encontraron más errores graves que fuese necesario solucionar en el momento, y se procedió con los objetivos que se habían marcado para la prueba hasta que esta finalizó.

El resultado de la prueba fue, en general, bastante positivo. Una vez arreglados los errores iniciales se pudo mantener durante todo el concurso al menos una instancia del locutor ejecutándose en todo momento. Se verificó también que una instancia del sistema que se empezara a ejecutar en medio del concurso recreaba correctamente el estado del mismo, ya que generaba los mismos comentarios interesantes que las instancias que se empezaron a ejecutar antes de que el concurso comenzase. Se comprobó también que los *logs* generados por el programa contenían todos los eventos interesantes que el programa había emitido.

La salida de eventos se evaluó detenidamente. Algunos tipos de eventos como envíos simples constituían la mayor parte de la salida, mientras que otros se emitían menos de lo que se esperaría. Los eventos interesantes se ajustaron en frecuencia y prioridad en base a lo observado en preparación para la evaluación final.

También se recogieron nuevas ideas de eventos interesantes para locutar. Algunos ejemplos de estos tipos de eventos fueron recordar de forma periódica el podio, plantear casos hipotéticos o destacar rachas de equipos que resolviesen muchos problemas a la primera. De todas las ideas generadas, algunas pudieron ser implementadas antes de la evaluación final.

Una observación técnica muy importante que se realizó fue la existencia de las muchas diferencias que había entre los *feeds* de eventos del CCS y del CDS. Fue en este punto en el que se decidió restringir el dominio del programa para que solo pudiese funcionar con sistemas CDS. Esto provocó

tener que refactorizar parte del código antes de la evaluación final.

El resto de observaciones que se realizaron fueron de menor importancia, entre las que se encontraban *bugs* que repercutían mínimamente en la ejecución del programa, y eventos del CDS que no se habían contemplado durante el desarrollo. Todo esto se corrigió antes de la evaluación final.

5.2. Evaluación final sobre un concurso real

El día 28 de abril de 2022 tuvo lugar el concurso Nacional ProgramaMe 2022², y sobre este concurso tuvo lugar una demostración en vivo del funcionamiento de nuestro programa, ejerciendo como locutor automático del mismo.

El evento, organizado desde la Facultad Informática de la UCM, se retransmitió por YouTube³, y se celebraba a nivel nacional desde cada uno de los centros participantes, es decir, era completamente online. En el concurso participaban alrededor de 140 equipos de una 50 instituciones distintas.

El sistema había cambiado desde la primera prueba que se hizo, muchos errores estaban arreglados, se habían añadido locuciones nuevas, y se habían implementado partes importantes del programa que antes no lo estaban, como el servidor de eventos interesantes o el sistema de etiquetado. En este punto, el programa tenía implementadas todas las funcionalidades descritas en la sección 4.4.

Para este concurso, el objetivo fue que el locutor automático transmitiese información a los espectadores del concurso y personas interesadas. Para que esto fuese posible, hizo falta implementar clientes de nuestro propio locutor. En otras palabras, nuestro programa era esencialmente un servidor de eventos interesantes, y hacía falta un cliente simple que leyese estos eventos y los mostrase a los usuarios de una forma amigable. Los dos clientes que se mencionan en la subsección 4.4.2 se adaptaron a las características de este concurso.

El *player* se modificó para incluir un rótulo del logo del concurso, y se configuró para que mostrase por pantalla todos los eventos interesantes que recibiese con prioridad 2 o superior. Una vez terminado, la aplicación se subió a un servidor para que fuese accesible desde internet y se envió la dirección a los organizadores junto al código fuente, por si preferían tener la aplicación ejecutándose localmente.

Para utilizar el bot de Twitter se creó una cuenta⁴ que se encargaría de publicar las locuciones automáticamente, luego se configuró el bot con las credenciales necesarias para poder publicar tweets desde la cuenta. La idea

²<https://programame.com/2022/nac/>

³https://www.youtube.com/watch?v=ElxGx_5Rac4

⁴https://twitter.com/tu_locutor_fav

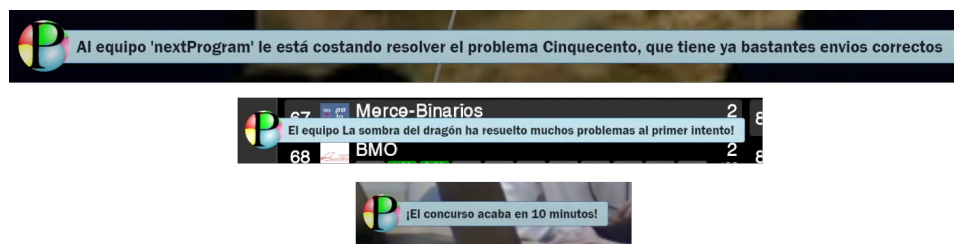


Figura 5.1: Muestras de algunos de los mensajes que se mostraron durante la retransmisión del concurso ProgramaMe Nacional 2022.

de este bot era que locutase los eventos interesantes que le ocurrieran a equipos pertenecientes a la organización IES Comercio⁵, e ignorase los eventos que no fuesen interesantes desde su punto de vista. Para implementar de filtrado de mensajes simplemente se hizo uso del sistema de etiquetado y prioridad. El bot publicaría cualquier evento con prioridad 4 y también cualquier evento con prioridad 2 o superior que incluyese “IES Comercio” entre sus etiquetas.

El servidor se desplegó en una máquina virtual proporcionada por los organizadores del concurso. Haciendo uso de `pm2`⁶ para que el servidor se volviese a ejecutar automáticamente en caso de fallo o reinicio inesperado del sistema.

El servidor se ejecutó y dejó funcionando antes de que empezara el concurso, y luego se revisó periódicamente que siguiera funcionando. Todo funcionó de forma remota. Tanto el *player* como el bot de Twitter funcionaron sin problemas durante todo el concurso.

En el caso del *player*, los organizadores lo introdujeron como otra fuente en la retransmisión, y durante el concurso mostró mensajes como los que se enseñan en la figura 5.1. El vídeo de la retransmisión entera, con todos los mensajes que se mostraron, puede visualizarse en la siguiente dirección: https://www.youtube.com/watch?v=ElxGx_5Rac4.

En el caso del bot de Twitter, realizó su función correctamente, y publicó los eventos interesantes que tenían que ver con la organización IES Comercio o se consideraban de prioridad máxima. Un ejemplo de algunos de los mensajes que se publicaron pueden observarse en la figura 5.2. Durante el concurso, publicamos manualmente algunos mensajes con fotos. Estos mensajes no se pudieron publicar desde el bot como se ideó originalmente por las limitaciones que tenía Twitter en el momento respecto a publicar fotos desde la API. Las fotos, sin embargo, las generamos automáticamente mediante un

⁵<https://iescomercio.com/>

⁶<https://pm2.keymetrics.io/>

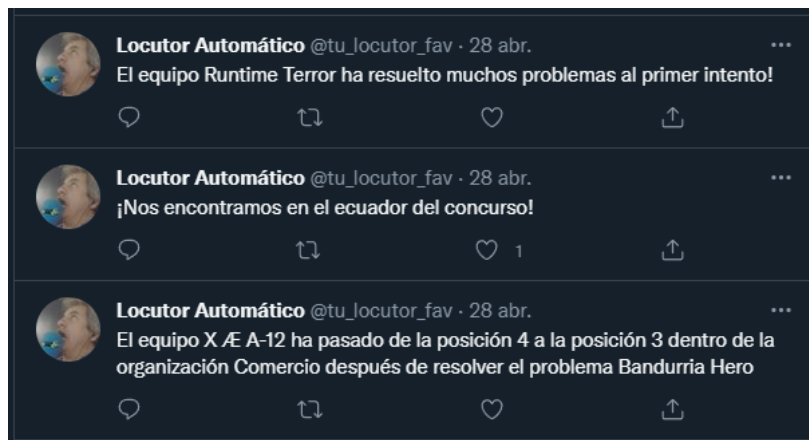


Figura 5.2: Muestras de algunos de los *tweets* que se publicaron durante la retransmisión del concurso ProgramaMe Nacional 2022.

script desarrollado para este propósito.

Todos los mensajes que se publicaron durante el concurso pueden observarse en la siguiente dirección: https://twitter.com/tu_locutor_fav.

Se recibió una gran cantidad de *feedback* informal. La mayoría de comentarios dijeron alguna variación de “el locutor da vida al concurso, es más interesante que ver un marcador constantemente”. Por este motivo se considera que el resultado de la demostración fue positivo, ya que crear un locutor que destaque sucesos **interesantes** era el principal propósito de este TFG. Muchos comentarios incluían sugerencias para ampliar o modificar ciertos aspectos del programa. Las ideas más repetidas fueron:

- Crear distintas plantillas para el mismo tipo de mensaje, evitando de esta manera que se repitan demasiado.
- No comentar sucesos en los que un equipo que va muy por detrás sube pocos puestos en el ranking.
- Estaría bien mencionar más a los centros, para dar contexto a la situación y visibilidad a las instituciones.
- A veces se echa de menos un “agregador de mensajes”, porque en ocasiones salen mensajes muy parecidos de forma seguida.
- Falta plantear casos hipotéticos de forma aleatoria. Por ejemplo, cuantos puestos adelantaría un equipo si consigue resolver el problema que está intentando.
- Hay muy pocos mensajes después de que se congele el marcador.

- En algunos momentos del concurso, como cuando había muchos envíos, los comentarios eran excesivos.

Algunos comentarios también incluían distintas ideas sobre nuevos tipos de mensajes interesantes.

Todo este *feedback* es muy útil porque, debido a la arquitectura altamente modular del programa, muchas de estas sugerencias pueden implementarse sin tener que reestructurar el programa.

En general creemos que las impresiones fueron buenas, y, al final del día, el locutor automático hizo todo lo que se había programado para que hiciese, sin fallos.

Capítulo 6

Conclusiones y Trabajo Futuro

En este capítulo se presentan, las conclusiones del trabajo realizado (sección 6.1), y se describen aspectos del sistema que podrían implementarse para mejorar y ampliar el locutor automático en el futuro (sección 6.2).

6.1. Conclusiones

Los concursos de programación son competiciones con una larga historia detrás. Año tras año, estos eventos han crecido en relevancia. Desde sus inicios, la tasa de participación ha crecido en más de un 2000 % según ICPC (2022). La retransmisión de estos concursos es la puerta de entrada del público general al evento, y cada vez son más los espectadores que disfrutan viendo el desarrollo de un concurso de estas características. Un obstáculo a solucionar es conseguir que esta retransmisión mantenga a los espectadores interesados en la misma, y esto se consigue mediante una buena planificación del directo. Se pueden planear entrevistas, explicaciones de problemas y mostrar contenido multimedia de lo que ha ocurrido y de lo que está ocurriendo en el concurso, pero el peso de la retransmisión se lo lleva la locución de los sucesos en el concurso. Por ello, hemos desarrollado un sistema capaz de extraer información relevante de los eventos de los concursos mediante su información estática y dinámica, y de generar noticias relevantes para los espectadores.

Nuestro locutor automático está pensado para mantener a los espectadores informados en todo momento de lo que ocurre en estos concursos y generar emoción cuando ocurren situaciones específicas, como por ejemplo, cuando un concursante está arrasando en la competición con muchos problemas resueltos al primer intento. Esto se ha logrado creando un sistema que, a través de una conexión con la API del concurso, en este caso el Contest API del servidor CDS, tal y como vimos en la sección 2.2, se obtiene la informa-

ción de los concursos, la cual es tratada desde distintos “puntos de vista”, tal y como se explica en la sección 3.3, y estos generan noticias interesantes. La relevancia de estos eventos de salida es variable, y puede ser importante si nos enfocamos en locutar a unos equipos concretos o no. Por ello, los eventos de salida se etiquetan según a qué entidades del concurso afectan directamente, y además se proporciona una relevancia clasificada del 0 al 4, siendo 0 una importancia muy baja y 4 una muy alta, que resulta ser interesante aunque tu locución concreta no esté centrada en aquellas etiquetas proporcionadas por el evento.

No obstante, este locutor no es capaz de contemplar todo lo que puede ocurrir en un concurso, y por lo tanto, no genera tantos eventos como podría hacer un locutor humano. Como veremos en la sección 6.2, se pueden implementar tantos mensajes como se deseen, por lo que este locutor podría convertirse en una herramienta aún más interesante.

Los eventos que produce este locutor son suficientes como para tener una idea de lo que está ocurriendo en un concurso, por lo que tal y como está implementado es una herramienta a tener en cuenta para organizar una retransmisión. Además, tal y como vimos en la segunda evaluación del locutor, explicada en la sección 5.2, donde probamos nuestro locutor en un concurso real, es posible configurar un cliente que se conecte a nuestro locutor y recibir mensajes relevantes en caso de querer centrarte, por ejemplo, en los equipos de una institución concreta.

Si bien al principio de este proyecto encontramos AutoAnalyst, una herramienta a la que dimos un repaso en la subsección 2.3.1, y que promete hacer un análisis en tiempo real de concursos estilo ICPC y crear un análisis interesante para los locutores, no pudimos comprender la funcionalidad alcanzada por su sistema debido a la escasa documentación de la que disponía. Este proyecto desarrollado por el equipo de analistas de ICPC es una herramienta más enfocada en hacer un análisis de cómo están rindiendo los participantes, ya que vimos que era capaz de estimar en qué problemas están trabajando los equipos, y realizar cálculos como cuándo un equipo debe resolver un problema para escalar en la clasificación. Este análisis está más centrado en los analistas del concurso que en las locuciones en sí. Cabe destacar que este análisis podría ser muy interesante para locutores humanos, ya que podrían dar estadísticas difíciles de calcular a simple vista. Autoanalyst, junto a MyICPC, han sido los dos servicios que más se asemejan al proyecto de este TFG. Ambas herramientas sirven como inspiración para nuestro proyecto al compartir los mismos objetivos. De AutoAnalyst nos ha gustado la idea de aportar comentarios que serían complicados de extraer a simple vista, incluso por un locutor humano. MyICPC nos ha mostrado que para mantener al público entretenido e informado es esencial capturar los sucesos de la competición y mostrarlos de una forma amigable y sencilla. La funcionalidad relacionada con ofrecer contenido interactivo a los participantes es

muy interesante, pero escapa del alcance de este TFG. Nuestro trabajo está más centrado en las locuciones en sí, y sin ser ni mucho menos un sistema perfecto para generar noticias relevantes, consigue realizar un buen trabajo en su tarea y es capaz de amenizar una retransmisión.

A lo largo de estos meses hemos trabajado junto a los tutores de este TFG en un proyecto tanto en la parte de investigación como en la parte de desarrollo. En la parte de investigación, hemos aprendido acerca del mundo de las retransmisiones en general. El trabajo de un locutor, como pudimos experimentar en la Olimpiada Informática de Madrid, es una tarea que requiere estar muy atento a los eventos que ocurren en un concurso, y también conlleva ser capaz de realizar comentarios interesantes para los espectadores. Un sistema como el nuestro puede ser de gran ayuda a los locutores humanos, y también hace que una retransmisión de un concurso no sea tan plana, teniendo siempre una fuente de noticias de eventos de relevancia variable, incluyendo desde simples veredictos a problemas, hasta análisis más complejos del estado del concurso.

Por otra parte, la parte de desarrollado ha sido una oportunidad perfecta para aplicar los conocimientos adquiridos durante el grado de Ingeniería del Software cursado por ambos. Gracias a cursar este grado, hemos podido gestionar el proyecto más grande al que nos hemos enfrentado hasta la fecha.

Para concluir, este proyecto nos ha aportado nuevos conocimientos. Entre ellos, destacan la programación reactiva, el tipado en JavaScript mediante TypeScript, la compresión de aplicaciones y su despliegado en Docker, el uso de LaTeX y muchos conceptos aprendidos, sobre todo el entorno de los concursos de programación en el cual nos hemos movido, incluyendo su funcionamiento e historia.

6.2. Trabajo futuro

Después de todo el proceso de desarrollo, se considera que se han cumplido los objetivos principales que se plantearon inicialmente para este TFG, siendo el más importante el ofrecer una alternativa a la locución humana en forma de un programa que locute de forma automática.

El locutor automático en su estado actual es funcional, y aunque puede utilizarse sin problema en concursos de programación estilo ICPC, puede mejorarse y ampliarse de distintas formas. La escalabilidad y modularidad son clave en este locutor automático, especialmente a la hora de asegurarse de que los cambios futuros planteados para el programa puedan realizarse de una forma menos costosa.

La principal forma de ampliar el locutor automático es mediante la introducción de nuevos tipos de eventos interesantes para que sean emitidos durante un concurso. La siguiente lista de eventos, propuestos como una po-

sible ampliación, se ha elaborado teniendo en cuenta el *feedback* obtenido de la prueba pública del sistema, descrita en la sección 5.2, y las ideas de eventos interesantes que no se han llegado a programar por falta de tiempo. Sin embargo, esta lista es orientativa, ya que el locutor puede ampliarse con muchos más eventos mientras se pueda obtener la información necesaria.

- Introducción de eventos interesantes que tengan lugar durante el momento del concurso en el que el marcador se encuentra congelado. En la versión actual del locutor no se emiten eventos interesantes cuando se ha congelado el marcador, más allá de cuanto tiempo queda de concurso, lo que hace este tramo menos interesante que el resto. La introducción de distintos eventos interesantes después de congelar el marcador solucionaría este problema, pero hay que trabajar con las limitaciones que esto supone, es decir, no saber los resultados de los envíos que se realizan durante este periodo. Un ejemplo de un evento interesante que puede emitirse durante el periodo de congelación sería un análisis del estado del marcador justo antes de congelarse.
- Se plantea también la introducción de eventos interesantes que planteen casos hipotéticos. Plantearse cuáles serían las consecuencias de que un equipo resuelva un problema que está intentando actualmente genera expectación entre las personas interesadas en el concurso, y por tanto, hace el concurso más interesante. También pueden plantearse casos hipotéticos en el pasado, por ejemplo, que habría ocurrido si un equipo hubiese tenido menos envíos erróneos, y si esto supondría que se encontraría más arriba en el ranking. Esta idea podría estar relacionada parcialmente con el punto anterior, ya que durante la congelación del marcador solo se puede especular. Por ejemplo, teniendo el marcador congelado y un envío por parte de un equipo, se puede emitir un evento interesante que explique qué ocurriría en caso de resolverlo.
- Un evento interesante encargado de mencionar que a un equipo que va bien en el concurso le falta por resolver uno de los problemas más fáciles. Para que un problema se identifique como uno de los más fáciles tendría que identificarse que es uno de los más resueltos.
- Eventos interesantes que aporten distintas estadísticas a gran escala, por ejemplo: porcentaje de problemas resueltos, porcentaje de equipos con un número específico de problemas resueltos, porcentaje de aciertos en función del lenguaje...

A continuación se listan algunos aspectos del sistema que pueden mejorarse respecto a la implementación actual:

- Elaborar diferentes plantillas de mensajes para un mismo tipo de evento interesante, evitando así que el locutor pueda parecer muy repeti-

tivo. Dicho de otra forma, que los mensajes asociados a los eventos interesantes no tengan un formato fijo, pero que todas las variaciones del mensaje transmitan la misma información. Por ejemplo “¡El equipo X ha resuelto su primer problema!, Y” y “Y ha sido el primer problema resuelto por el equipo X” son mensajes que transmiten la misma información y se originan del mismo evento interesante.

- Ajustar la frecuencia y prioridad de los mensajes en función de las dimensiones del concurso, ya que la cantidad de información que se desea transmitir varía en función de factores como el número de equipos, el número de problemas o la duración. Como se vio en el *feedback* en la sección 5.2, en ocasiones el sistema emitía eventos en exceso. Tener estos factores en cuenta para emitir más o menos eventos interesantes o variar la prioridad de los mensajes resultará en un locutor más dinámico, capaz de adaptarse a concursos de distinta escala.

Poniendo el foco en las características generales del sistema, se pueden realizar varias ampliaciones:

- Crear nuevos clientes del locutor. Una vez está creado el locutor de eventos interesantes, crear nuevos clientes es una tarea sencilla. Algunos ejemplos de nuevos clientes del programa serían: un bot de Telegram que anuncie como va un equipo en un concurso, un programa que presente la información obtenida de forma gráfica a un locutor humano (para ayudarlo en la tarea de locutar), o una aplicación de móvil que envíe una notificación cada vez que tu equipo favorito resuelve algún problema.
- Elaborar la documentación del formato del campo “info” en función del tipo de evento interesante para facilitar su uso. Tal y como se explica en la subsección 4.3.2, este campo sirve para que un cliente pueda crear un mensaje con formato propio. Como alternativa pueden utilizarse los mensajes ya creados por el propio locutor automático.
- Dar al usuario la opción de elegir el idioma de los mensajes que construye el *feed* interesante, reduciendo así el trabajo potencial que le supondría a los clientes del locutor que deseen otro idioma de locución.
- Permitir insertar a mano eventos en el locutor, ofreciendo así a los responsables de la ejecución del programa la posibilidad de insertar comentarios propios. Estos eventos “manuales” tendrían que tener un formato específico para poder identificarse como tal.

Chapter 6

Conclusions and Future Work

This chapter presents the conclusions of the work carried out (section 6.1), and describes aspects of the system that could be implemented to improve and expand the automatic caster in the future (section 6.2).

6.1. Conclusions

Programming contests are competitions with a long history behind them. Year after year, these events have grown in relevance. Since its inception, the participation rate has grown by more than 2000% as shown by ICPC (2022). The broadcasting of these contests is the general public's gateway to the event, lately more and more viewers enjoy watching the development of a contest of these characteristics. An obstacle to solve is to ensure that this broadcast keeps viewers interested in it, and this is achieved through a good planning of the broadcast. Interviews, explanations of problems and multimedia content of what has happened and is happening in the contest can be planned ahead, but the weight of the broadcast is carried by the casting of the events in the contest. Therefore, we have developed a system capable of extracting relevant information from the events of the contests through their static and dynamic information, and generate relevant events for the viewers.

Our automatic caster is designed to keep viewers informed at all times of what is happening in these contests and generate hype when specific situations occur, such as, for example, when a contestant is sweeping the competition with many problems solved at the first attempt. This has been achieved by creating a system that, through a connection with the API of the competition, in this case the CDS Contest API, as we saw in section 2.2, we get the information of the competitions, which is treated from different “points of view”, as explained in section 3.3, and these generate interesting

events. The relevance of these events is variable, and it can be important if we focus on covering specific teams or not. Therefore, the interesting events are labeled according to which entities of the contest they directly affect, and in addition a relevance indicator that goes from 0 to 4 is provided, with 0 being a very low importance and 4 a very high one, which turns out to be interesting even if your main focus is not centered around the labels provided by the event.

However, this automatic caster is not able to contemplate everything that can happen in a contest, and therefore, it does not generate as many events as a human announcer could. As we see in section 6.2, as many messages as desired can be implemented, so this announcer could become a tool even more interesting.

The events produced by this automatic caster are enough to have an idea of what is happening in a contest, which means in its current state it is a tool to take into account when organizing a broadcast for such contests. In addition, as we saw in the second assessment of the system, as explained in section 5.2, where we tested our caster in a real contest, it is possible to configure a client to connect to our host and receive relevant messages in case you want to focus, for example, in teams belonging to a particular institution.

At the beginning of this project we found AutoAnalyst, a tool that we review in section 2.3.1, and which promises to make a real-time analysis of ICPC style contests and create an interesting analysis for the speakers, we were not able to understand the functionality achieved by this system due to the limited documentation available. This project developed by the ICPC team of analysts is a tool whose main focus is doing an analysis of how the participants are performing, since we saw that it was able to estimate what problems the teams were working on, and perform calculations such as when a team should solve a problem to climb the ranking. This analysis is more focused on the analysts of the contest than on the voiceovers themselves. It should be noted that this analysis could be very interesting for human casters, since it can provide statistics which are difficult to calculate by hand. Autoanalyst, together with MyICPC, have been the two services that most resemble the project presented here. Both tools serve as inspiration for our project by sharing the same goals. From AutoAnalyst we liked the idea of providing comments that would be difficult to extract by hand, even by a human caster. MyICPC has shown us that in order to keep the public entertained and informed it is essential to capture the events of the competition and show them in a friendly and simple way. The functionality related to offering interactive content to participants is very interesting, but it is out of the scope of this project. Our work focuses more on the casting itself, and being far from a perfect system to generate relevant news, it manages to do a good job at its proposed task and is able to liven up a broadcast.

Throughout these months we have worked together with the mentors of this Final Grade Project both in the research part and in the development part. In the research part, we have learned about the world of broadcasting in general. The work of an caster, as we were able to experience at the Madrid Informatics Olympiad, is a task that requires being very attentive to the events that occur in a contest, and also entails being able to make interesting comments for viewers. A system like ours can be of great help to human announcers, and also livens up the broadcast of a contest, always having a news source of events of variable relevance, wich include events ranging from simple verdicts to problems to complex analysis of the state of the contest.

Also, the process of development on its own has been a perfect opportunity to apply the knowledge acquired during the Software Engineering degree studied by both. Thanks to studying this degree, we have been able to manage the largest project we have faced to date.

To conclude, this project has provided us with new knowledge. Amongst it, reactive programming stands out, learning and using TypeScript, application compression and its deployment in Docker, the use of LaTeX and many other concepts learned, especially the environment of programming contests in which we have moved, including its history and how it operates.

6.2. Future Work

After the entire development process, we considered that the main objectives that were initially proposed for this Final Grade Project have been met, the most important one being to offer an alternative to human speech in the form of a program wich casts automatically.

The automatic caster in its current state is functional, and although it can be used without problem in ICPC-style programming contests, it can be improved and expanded in different ways. Scalability and modularity are key in this automatic caster, especially when it comes to making sure that future changes proposed for the program can be made in a less taxing way.

The main way to expand the automatic caster is by introducing new types of interesting events to be broadcast during a contest. The following list of events, proposed as a possible extension, has been developed taking into account the feedback obtained from the public test of the system, described in section 5.2, and the ideas of interesting events that have not been yet implemented mainly due to lack of time. However, this list is orientative, since the announcer can be expanded with many more events as long as the necessary information can be obtained.

- Introduction of interesting events that take place during the time of the contest in which the scoreboard is frozen. In the current version

of the automatic caster, interesting events are not broadcasted when the scoreboard has been frozen, except how much time is left, which makes this section less interesting than the rest. The introduction of different interesting events after freezing the scoreboard would solve this problem, but it is necessary to work with the limitations that this implies, that is, not knowing the results of the submissions that are made during this period. An example of an interesting event that can be emitted during the freezing period would be an analysis of the state of the scoreboard just before freezing.

- The introduction of interesting events that raise hypothetical cases is also proposed. Analyzing what the consequences would be for a team to solve a problem they are currently trying to solve generates expectation among people interested in the contest, and therefore, makes the contest more interesting. Hypothetical cases can also be raised in the past, for example, what would have happened if a team had fewer erroneous submissions, and if this would mean that they would be higher in the ranking. This idea could be partially related to the previous point, since during the freezing of the scoreboard you can only speculate. For example, having the scoreboard frozen and a submission by a team, an interesting event can be issued that explains what would happen in case of solving it.
- An interesting event in charge of mentioning that a team that is doing well in the contest has yet to solve one of the easiest problems. For a problem to be identified as one of the easiest, it would have to be firstly identified as one of the most solved.
- Interesting events that provide different statistics on a large scale, for example: percentage of problems solved, percentage of teams with a specific number of problems solved, percentage of correct submissions depending on the language...

Listed below are some aspects of the system that can be improved over the current implementation:

- Develop different message templates for the same type of interesting event, thus avoiding that the announcer may seem repetitive. In other words, making it so that the messages associated with the interesting events do not have a fixed format, but that all the variations of the message transmit the same information. For example “Team X has solved its first problem!”, and “Y has been the first problem solved by team X” are messages that convey the same information and originate from the same interesting event.

- Adjust the frequency and priority of the messages according to the dimensions of the contest, since the amount of information to be transmitted varies depending on factors such as the number of teams, the number of problems or the duration. As seen in the feedback in section 5.2, sometimes the system emitted events in excess. Taking these factors into account to broadcast more or less interesting events or vary the priority of the messages will result in a more dynamic automatic caster, able to adapt to contests of different scale.

Focusing on the general characteristics of the system, several extensions can be made:

- Create new clients of the automatic caster. Once the main program is created, creating new clients is a simple task. Some examples of new clients of the program would be: a Telegram bot that announces how a team is going in a contest, a program that presents the information obtained graphically to a human caster (to help them in their task), or a mobile application that sends a notification every time your favorite team solves a problem.
- Prepare the documentation of the format of the “info” field depending on the type of interesting event to facilitate its use. As explained in subsection 4.3.2, this field can be used by a client to create a message with its own format. As an alternative, messages already created by the automatic caster itself can be used.
- Give the user the ability to choose the language of the messages created by the interesting feed, thus reducing the potential work of the caster’s clients who want the pre-built messages in another language.
- Allow insertion of events in the announcer by hand, thus offering those responsible for the execution of the program the ability of inserting their own comments. These “manual” events would have to have a specific format in order to be identified as such.

Capítulo 7

Trabajo Individual

A continuación se expone el trabajo individual que ha realizado cada miembro del equipo en el proyecto.

7.1. Javier del Río López

Los primeros días del proyecto Mario y yo acudimos a una serie de reuniones con los tutores de este TFG. En estas reuniones se habló de la motivación del proyecto, de lo que se entendía como locutor, y cuáles iban a ser los objetivos de este TFG. Pedro Pablo ofreció una visión general de como están estructurados los concursos de programación, y esta estructura se estudiaría en profundidad más adelante.

Las tareas más específicas que realicé a lo largo de la duración del TFG fueron:

- Estudié junto a Mario el contexto histórico de los concursos de programación, su evolución a lo largo de los años, y su estado en la actualidad.
- Exploré, junto a Mario, el Contest API en profundidad, para ver qué utilidad podían tener los distintos *endpoints*, qué operaciones y permisos nos hacían falta, la forma correcta de obtener cada tipo de datos, el formato de las respuestas, etc.
- Investigué junto a Mario la infraestructura típica que tenía un concurso de programación estilo ICPC, informándonos de como interactuaban los distintos componentes del sistema, y cuáles eran relevantes para nuestro propósito.
- Surgió y se desarrolló la idea de los puntos de vista como eje central del locutor automático. Luego diseñé en colaboración con Mario la arquitectura a alto nivel del programa, pensando en como interactuarían

las distintas clases del locutor automático, donde encajarían los puntos de vista, y la mejor forma de programarlo para que fuese fácilmente extensible.

- Desplugué una serie de servidores de prueba para simular una infraestructura típica de concurso sobre la que poder hacer pruebas. Utilicé un sistema de *rollback* para poder reiniciar un concurso a un estado previo al inicio, y de esta forma probar el programa más eficientemente. Los servidores desplegados fueron *judgehost*, *DOMjudge* y *CDS*.
- Implementé junto a Mario un prototipo del programa que se encargaba de leer del feed de eventos del Contest API de forma continua. Este fue el punto de partida del código, ya que implementar esta parte del programa era esencial para poder hacer pruebas sobre el resto de sus componentes.
- Programé la parte de la aplicación responsable de recibir los distintos eventos del *event feed* y convertirlos en objetos tipados que fuesen manejables por TypeScript. Esto evitaba utilizar los eventos en su formato *json* original, que fue útil por dos motivos: se garantizaba que no se accedieran a propiedades que podían no existir, y se utilizaba un tipo común para tratar las marcas de tiempo y poder operar sobre ellas.
- Programé la parte de la aplicación responsable de cargar un archivo de configuración con los datos de un concurso. Para hacer esta parte utilicé el módulo **confinode**.
- Implementé la lógica principal de los puntos de vista, es decir su funcionamiento interno. Los puntos de vista que implementé fueron los referentes al *scoreboard*, los problemas, los equipos, los grupos, las organizaciones y el tiempo. Fui actualizando la lógica contenida en cada punto de vista conforme se iban implementando nuevos eventos interesantes. Este fue un proceso largo y continuo que se alargó hasta el final de la implementación del sistema, y se compaginó con otro tipo de actividades. No programé al principio todos los puntos de vista, sino que los fui añadiendo según iban haciendo falta.
- Desarrollé una funcionalidad que resolvía un fallo de comunicación en red. Esta funcionalidad se encargaba de reunir los distintos paquetes de datos que podían formar un solo evento, y evitar así que fuese imposible parsearlos. El problema se originó de asumir que cada evento se emitía en un paquete independiente.
- Implementé un sistema para detectar cambios en las fechas de inicio, congelado y fin del concurso que permitía manejar situaciones como que se adelantaran o retrasaran estas fechas durante la celebración de

un concurso sin necesidad de reiniciar el locutor automático, evitando así que fuese necesario reiniciar el programa para su correcto funcionamiento.

- Programé el módulo del programa responsable de levantar un servidor que maneja muchas peticiones continuas para transmitir constantemente los eventos interesantes que le llegasen, asegurándome de que las peticiones no caducaban con el paso del tiempo, de que se descartaban las conexiones cerradas, y de que se podían llegar a manejar un gran número de conexiones simultáneamente. Consiguiendo así simular el comportamiento que ya conocía del Contest API.
- Implementé el *player* utilizado en la retransmisión del concurso Nacional ProgramaMe 2022, que se encargaba de mostrar mensajes en directo. Utilicé tecnología web para crear un programa que poder añadir como fuente a programas de retransmisión en directo.
- Desplegué el locutor interesante y el *player* para la prueba pública del sistema sobre el concurso Nacional ProgramaMe 2022. Utilicé un servidor proporcionado por uno de los organizadores del concurso.

7.2. Mario Blanco Domínguez

Los primeros días del proyecto, tuvimos una reunión con los tutores del TFG que nos sirvió para entender cuál era la motivación de este proyecto. Además, Pedro Pablo nos hizo un repaso sobre la infraestructura de los concursos de programación estilo ICPC (DOMjudge y CDS) para ubicar nuestro sistema dentro de dicha infraestructura. También nos otorgó un par de máquinas virtuales, una que aloja el sistema CCS implementado mediante DOMjudge y otra con la función de ser el juez de envíos. Estas nos sirvieron de gran ayuda para probar nuestro locutor. Durante estos primeros días, nos aseguramos de que teníamos un sistema CCS montado en nuestras máquinas virtuales y nos dedicamos a explorar a fondo el sistema DOMjudge para familiarizarnos con él. Tanto Javier como yo ya conocíamos este sistema, debido a que se utiliza en asignaturas como FAL, EDA y TAIS para resolver problemas, pero nunca habíamos estado en la parte de administrador.

Tras esta primera toma de contacto, comenzamos a investigar sobre el ecosistema de los concursos de programación. Junto a Javier, estuve informándome sobre el contexto histórico de los concursos de programación, pasando por su historia, su funcionamiento y las diferentes herramientas que se utilizaban para gestionarlos. Personalmente, partía de un conocimiento menor sobre el funcionamiento de los concursos que mi compañero, debido a que él es aficionado a los concursos de programación. Aprendí conceptos como el marcador congelado, que cuando un equipo resuelve un problema se

entrega un globo a modo de recompensa y que existe un sistema interno de clarificaciones mediante el cual los equipos pueden solicitar ayuda a los jueces humanos del concurso tanto como canal de información sobre el evento en general (horarios de descanso y comidas, pedir un bolígrafo o más folios para escribir...).

Junto a Javier, estuvimos examinando el Contest API en profundidad con el objetivo de tener en mente todos los métodos que este ofrece, identificar aquellos *endpoints* más útiles para nuestro proyecto y para inspeccionar el formato del API.

Personalmente, me encargué de explorar qué existía actualmente en cuanto a locutores automáticos y humanos de concursos de programación. A pesar de no existir mucha información al respecto, ya que podríamos considerar que estas competiciones son de nicho, pude averiguar que diferentes personas han tratado de amenizar las retransmisiones de estos concursos de diferentes maneras. Esto me llevó a investigar acerca de locutores automáticos en otros ámbitos, con el objetivo de obtener posibles ideas para nuestro locutor, y para poder establecer un objetivo alcanzable en comparación a lo que se ha logrado hasta ahora.

Ambos estuvimos pensando en la arquitectura del programa, trazando en nuestras mentes qué componentes podrían formar parte de nuestro sistema y cómo interactuarían entre ellos.

A la hora de programar, se implementó en conjunto un pequeño prototipo de un programa capaz de leer del *feed* de eventos del Contest API de forma continua. Teniendo este prototipo, decidimos separarnos en el desarrollo. Personalmente, estuve trabajando en la gestión de eventos de entrada y de salida. Para ello, utilicé la librería RxJS, que permite implementar de forma sencilla el patrón de diseño Observer. Esto me permitió suscribir a una serie de observadores, en este caso, los puntos de vista del sistema, a un canal donde se envían los eventos recibidos del CDS, para que estos reaccionen según el tipo de evento que se produce.

Para la segunda prueba de concepto, configuré al completo un bot de Twitter que se encargaría de twittear los resultados de los equipos que representarían a una institución en el concurso. No era mi primera experiencia con un bot de Twitter, pero por desgracia toda la API había cambiado recientemente y no me servía lo que tenía hecho hasta ahora. También desarrollé un script que construiría dinámicamente una imagen para acompañar a los mensajes del locutor. Esta imagen se construiría a partir de las fotos alojadas en el servidor CDS y del mensaje en sí. Por desgracia, Twitter me otorgó el acceso al *endpoint* que me permitía incluir multimedia en los mensajes un día después de la celebración del concurso. Fue bastante molesto, pero como ya había creado este script decidimos utilizarlo manualmente para subir algún *tweet* con imagen¹.

¹https://twitter.com/tu_locutor_fav/status/1519608218716094464

También me encargué de proporcionar una imagen Docker que contenga nuestro sistema. Nunca había utilizado esta herramienta, y nos resultó muy interesante a ambos para ofrecer el despliegado de un programa. Docker utiliza máquinas virtuales Linux, y permite que los programas sean independientes de la plataforma en la que se ejecutan.

Apéndice A

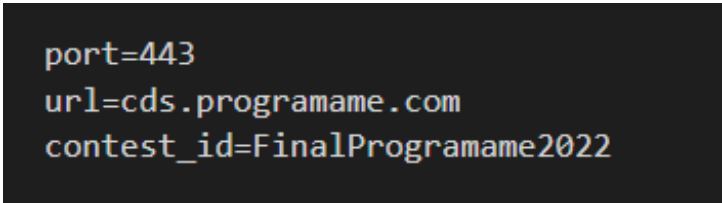
Configuración y Despliegado

Este apéndice tiene como objetivo facilitar el uso de nuestro sistema. En la sección A.1 se encuentra la configuración inicial del sistema para poder conectarse al servidor CDS. En la sección A.2 se detallan las opciones de despliegado de nuestro sistema, junto a los pasos a reproducir para cada una de ellas.

A.1. Configuración inicial

Para configurar los aspectos del locutor automático relacionados con el servidor del cual va a extraer la información del concurso, se ofrecen, a continuación, las propiedades que se pueden configurar mediante variables de entorno:

- **url (Requerido. String):** IP o nombre de dominio donde se encuentra desplegado el servidor CDS al que se desea conectar el locutor automático.
- **port (Requerido. Number):** Puerto en el que está escuchando el servidor CDS.
- **contest_id (Requerido. String):** ID del concurso que se desea escuchar.
- **https (Opcional. Boolean):** Indica si el servidor utiliza o requiere una conexión segura para poder acceder a los datos. Su valor por defecto es **true**.
- **allow_expired_tls (Opcional. Boolean):** Indica si el sistema debe ignorar la validez de los certificados TLS en servidores seguros. Esta propiedad es útil para probar el locutor en entornos controlados donde



```
port=443
url=cds.programame.com
contest_id=FinalProgramame2022
```

Figura A.1: Ejemplo del posible contenido del archivo de configuración de variables de entorno *.env*.

puede ser complicado conseguir un certificado válido, pero no es recomendable activarla en entornos de producción. La propiedad se ignora si se indica, mediante la propiedad anterior, que el servidor no requiere una conexión segura. Su valor por defecto es **false**.

- **api_user (Opcional. String)**: Si el servidor requiere autenticación, el programa utiliza este campo como usuario para autenticarse. Si se deja vacío, se intenta acceder de forma anónima.
- **api_password (Opcional. String)**: Si el servidor requiere autenticación, el programa utiliza este campo como contraseña para autenticarse. Este campo se ignora si el campo anterior está vacío.

El sistema, en caso de encontrar como mínimo las variables de entorno requeridas, utilizará este método como forma de configuración. Mediante estas variables de entorno, podremos configurar la conexión con el servidor CDS. Su razón principal de existir es facilitar la configuración del programa en entornos de producción, concretamente en la imagen desplegada en Docker que se explicará más adelante en la sección A.2. En entornos de desarrollo existen muchas formas de establecer las variables de entorno, dependiendo principalmente del sistema operativo en el cual se está desarrollando. La opción que encontramos más cómoda y más compatible entre diferentes entornos de programación es crear un fichero denominado *.env* en la carpeta raíz del proyecto, que contenga las variables de entorno que se deseen configurar. Podemos apreciar en la figura A.1 un ejemplo de este fichero que contiene las variables de entorno.

En entornos de desarrollo también es posible realizar esta configuración mediante un fichero denominado “FeedInteresante.config.json”, el cual se encuentra en la carpeta raíz del proyecto. La existencia de este fichero hace que el programa pueda configurarse del mismo modo que usando las variables de entorno.

Todas las propiedades son requeridas en este fichero de configuración. Su estructura es la siguiente:

- **cds**: Una colección de propiedades que indican al programa como conectarse a un CDS específico, esas propiedades son:
 - **url (String)**: IP o nombre de dominio donde se encuentra desplegado el servidor CDS al que se desea conectar el locutor automático.
 - **port (Number)**: Puerto en el que está escuchando el servidor CDS.
 - **https (Boolean)**: Indica si el servidor utiliza o requiere una conexión segura para poder acceder a los datos.
 - **allow_expired_tls (Boolean)**: Indica si el sistema debe ignorar la validez de los certificados TLS en servidores seguros. Esta propiedad es útil para probar el locutor en entornos controlados donde puede ser complicado conseguir un certificado válido, pero no es recomendable activarla en entornos de producción. La propiedad se ignora si se indica, mediante la propiedad anterior, que el servidor no requiere una conexión segura.
 - **contest_id (String)**: ID del concurso que se desea escuchar.
 - **api_user (String)**: Si el servidor requiere autenticación, el programa utiliza este campo como usuario para autenticarse. Si se deja vacío, se intenta acceder de forma anónima.
 - **api_password (String)**: Si el servidor requiere autenticación, el programa utiliza este campo como contraseña para autenticarse. Este campo se ignora si el campo anterior está vacío.
- **server_port (Number)**: Número que indica en qué puerto se pondrá a escuchar el servidor de eventos interesantes que lance el programa. Este servicio se explica más adelante.

Un ejemplo del archivo de configuración puede observarse en la figura A.2.

En caso de no existir las variables de entorno requeridas, y estando en un entorno de desarrollo (es decir, utilizando el código fuente en vez de utilizar la imagen Docker), se utilizará este método de configuración. Si el fichero no tiene una estructura válida, se comunica al usuario cuál es error. Si alguna propiedad reconocida tiene un tipo de dato distinto al esperado también se comunica al usuario.

Si el archivo y los datos tienen un formato correcto, se prueba a conectar con el servidor que ha especificado el usuario. Si la conexión falla, el programa termina y comunica al usuario que no ha sido posible conectarse.

```
{
  "cds" : {
    "url": "cds.programame.com",
    "port": 443,
    "https": true,
    "allow_expired_tls": false,
    "contest_id": "FinalProgramame2022",
    "api_user": "",
    "api_password": ""
  },
  "server_port": 1337
}
```

Figura A.2: Ejemplo del posible contenido del archivo de configuración *FeedInteresante.config.json*.

A.2. Despliegado

Para desplegar una instancia del locutor automático se pueden seguir dos procedimientos distintos.

Una forma de desplegar el programa es hacer uso de la imagen Docker¹ ofrecida. Esta imagen se encuentra en DockerHub², un repositorio de contenedores de Docker. Se encuentra en la siguiente dirección:

<https://hub.docker.com/r/bdmariobd/locutor-concursos>.

Los únicos requisitos necesarios para poder desplegar un contenedor con nuestro sistema utilizando esta imagen es tener instalado el cliente Docker. Para ejecutar la imagen, se deben seguir los siguientes pasos:

- Ejecutar el comando *pull* para descargar la imagen de DockerHub: `docker pull bdmariobd/locutor-concursos:latest`. Podemos comprobar mediante `docker image ls` que la imagen se ha descargado correctamente.
- Lanzar un contenedor con la imagen desplegada. En este paso, se deben incluir necesariamente las variables de entorno explicadas anteriormente en la sección A.1 requeridas para la correcta conexión con el servidor CDS. Para ello, se debe incluir un parámetro `-e` por cada variable de entorno deseada. Adicionalmente, es necesario incluir el parámetro `-p` para indicar el puerto que se utilizará para escuchar los eventos de salida. El sistema lanza el servidor de eventos interesantes en el puerto 1337, por lo que para remapear este puerto de la máquina virtual

¹<https://www.docker.com>

²<https://hub.docker.com/search?q=>

```
PS C:\Users\mario> docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
7d6e4536d949	bdmariobd/locutor-concursos	"node main.js"	5 minutes ago	Up 5 minutes	0.0.0.0:3000->1337/tcp	upbeat_bardeen

Figura A.3: Muestra de los contenedores Docker que se encuentran en funcionamiento.

```
PS C:\Users\mario> docker exec -t -i 7d6e4536d949 /bin/sh
/app # ls
logs          main.js       main.js.LICENSE.txt
/app # cd logs
```

Figura A.4: Muestra de una shell Unix en el contenedor Docker.

de Docker a nuestro sistema, se debe incluir el parámetro `-p puerto deseado:1337` en el comando de ejecución. Un ejemplo de comando de ejecución sería:

```
docker run -p 3000:1337 -e url='cds.programame.com' -e port='443' -e
contest_id='FinalProgramame2022' bdmariobd/locutor-concursos.
```

- Desplegar aquellos clientes deseados para escuchar al locutor. Adicionalmente, podemos ver el fichero de los logs del programa en la carpeta logs del proyecto. Para acceder a ella desde un contenedor Docker, debemos ejecutar primero el comando `docker ps` para ver los contenedores que tenemos actualmenete en funcionamiento, como puede verse en la figura A.3. Tras obtener el id del contenedor contenido en el campo *CONTAINER ID*, se procede ejecutando `docker exec -t -i id del contenedor /bin/sh`. En este punto, el contenedor se encuentra en un shell de Unix, por lo que se puede ejecutar comandos como `ls`, `cd/logs` y `nano nombre archivo` para acceder a los ficheros de logs, tal y como se muestra en la figura A.4.

Una forma alternativa de desplegar el locutor requiere tener instalado Node³. Al haber programado la aplicación en TypeScript, el código debe ser compilado a JavaScript. Por ello, si se desea desplegar el locutor mediante Node, pero sin modificar el código fuente, ya sea para implementar alguna característica descrita en la sección 6.2 o para introducir otros cambios, se debe descargar el código fuente de la aplicación contenido en el siguiente repositorio: <https://github.com/TFG-locutor/feed-interesante>. Tras introducir los cambios necesarios y configurar el archivo *FeedInteresante.config.json* o configurar las variables de entorno deseadas, se debe ejecutar el comando `npm install` para instalar las dependencias de la aplicación. Posteriormente, para ejecutar el locutor, se debe ejecutar el comando `npm start`, el cual compilará el código a JavaScript en la carpeta *build* y lanzará el sistema.

³<https://nodejs.org/>

En caso de desear desplegar el locutor mediante Node, pero sin modificar el código fuente, se ofrece en el apartado **Releases** del repositorio (<https://github.com/TFG-locutor/feed-interesante/releases>) con una versión del código previamente compilado a JavaScript. El proceso consiste en descargarse el código de cualquier *release*, editar el archivo *FeedInteresante.config.json* o configurar las variables de entorno necesarias, y lanzar el locutor ejecutando mediante el comando `node build/index.js`.

Bibliografía

ADAMS, E. How to Write Sports Commentary. 2009.

CARION, N., MASSA, F., SYNNAEVE, G., USUNIER, N., KIRILLOV, A. y ZAGORUYKO, S. End-to-end object detection with transformers. En *Computer Vision – ECCV 2020* (editado por A. Vedaldi, H. Bischof, T. Brox y J.-M. Frahm), páginas 213–229. Springer International Publishing, Cham, 2020. ISBN 978-3-030-58452-8.

CERNY, T. y DONAHOO, M. J. Second screen engagement of event spectators. *Advances in Human-Computer Interaction*, vol. 2018, página 3845123, 2018. ISSN 1687-5893.

HUGH TAYLOR, L. P. F. M., ANGELA YOCHER. *Event-Driven Architecture: How SOA Enables the Real-Time Enterprise*. Pearson Education, 2008. ISBN 978-0-321-32211-1.

ICPC. ICPC FACT SHEET. Informe técnico, Dhaka, Bangladesh, 2022.

RYSAVÝ, F. *Mobile application for MyICPC with context acquisition for IoT environments*. Masters Thesis, Czech Technical University in Prague, Prague, 2018.

