

DESARROLLO DE UNA LIBRERÍA DE SERIALIZACIÓN Y
DESERIALIZACIÓN BINARIA BASADA EN REFLEXIÓN EN TIEMPO DE
COMPILACIÓN CON ISO/IEC 14882:2020 (C++20)

Este Jurado; una vez realizado el examen del presente trabajo ha evaluado su contenido con el resultado:

J U R A D O E X A M I N A D O R

Firma:

Firma:

Firma:

Nombre: Nombre: Nombre:

Realizado por

López Larrad, Adrián

López Larrad, Leonardo

Profesora guía

Ortega, Dinarle Milagro

Fecha

Junio, 2025



DESARROLLO DE UNA LIBRERÍA DE SERIALIZACIÓN Y DESERIALIZACIÓN BINARIA BASADA EN REFLEXIÓN EN TIEMPO DE COMPILACIÓN CON ISO/IEC 14882:2020 (C++20)

TRABAJO ESPECIAL DE GRADO
presentado ante la
UNIVERSIDAD CATÓLICA ANDRÉS BELLO
como parte de los requisitos para optar al título de
INGENIERO EN INFORMÁTICA

Realizado por	López Larrad, Adrián López Larrad, Leonardo
Profesora guía	Ortega, Dinarle Milagro
Fecha	Junio, 2025

Para nuestra mamá, Anabela.

Este logro es tuyo.

Queremos expresar nuestro más sincero agradecimiento a nuestra mamá, Anabela; a nuestra hermana, Alexandra; y a nuestros abuelos, Angelita y Ángel. Ustedes nos han dado todo.

Agradecemos también a nuestro padrino, Alexander, por su valiosa ayuda a lo largo de todos estos años académicos.

Deseamos agradecer y reconocer a nuestra profesora guía, Dinarle, quien nos brindó toda su experiencia, su apoyo y su optimismo durante nuestros proyectos finales en la carrera.

Finalmente, extendemos nuestro agradecimiento a nuestros profesores, compañeros y amigos en la Facultad de Ingeniería, quienes nos acompañaron y guiaron en nuestra formación académica y profesional.

Índice de Contenido

Introducción.....	1
Capítulo I	
El Problema.....	3
Planteamiento del Problema.....	3
Objetivos.....	7
Objetivo general.....	7
Objetivos específicos.....	7
Alcance.....	8
Limitaciones.....	12
Justificación.....	13
Capítulo II	
Marco Teórico.....	15
Bases Teóricas.....	15
Serialización y deserialización binaria.....	15
Reflexión computacional.....	16
Reflexión en tiempo de compilación.....	17
Reflexión en tiempo de ejecución.....	18
Lenguaje de programación C++.....	18
Tipos de datos.....	19
Fundamentales.....	19
Compuestos.....	20
Clases.....	20
Clases trivialmente copiables.....	21
Clases de diseño estándar.....	21
Clases agregadas (Tipos de datos agregados).....	22
Objetos.....	23
Representación de objeto.....	23
Representación de valor.....	24
Plantillas y Conceptos.....	24
Librería estándar.....	25
Contenedores.....	25
Tuplas.....	25
Objetos opcionales.....	26
Variantes.....	26
Punteros inteligentes.....	26
Evaluación de Expresiones Constantes.....	27
Alineación de estructuras de datos.....	28
Alineación explícita.....	29
Estructuras de datos empaquetadas.....	30
Ordenamiento de bytes.....	31

Formatos de archivos.....	32
Formatos de números enteros y reales.....	33
Enteros de ancho fijo.....	33
Complemento a dos.....	33
Codificación Zig-Zag.....	33
Enteros de longitud variable.....	34
Números reales (aritmética de coma flotante).....	35
Evaluación de sistemas informáticos.....	36
Eficiencia de desempeño.....	36
Compatibilidad.....	37
Mantenibilidad.....	37
Flexibilidad.....	37
Antecedentes.....	38
Librerías de serialización binaria.....	38
Protocol Buffers.....	38
FlatBuffers.....	39
MessagePack.....	40
Boost.Serialization.....	41
Diseño de serialización de Yu Qi.....	42
Capítulo III	
Marco Metodológico.....	44
Metodología Utilizada.....	44
Capítulo IV	
Desarrollo y Resultados.....	49
Formato de Archivo Binario.....	49
Análisis del formato de Yu Qi.....	49
Diseño del formato de archivo binario.....	50
Alineación de objetos.....	50
Ordenamiento de bytes.....	50
Codificación de objetos.....	51
Esquema de datos.....	51
Retrocompatibilidad.....	52
Estructura del archivo binario.....	52
Cabecera de archivo.....	52
Carga útil.....	53
Especificación técnica del formato de archivo binario.....	53
Codificación de objetos.....	53
Números enteros.....	53
Números reales.....	53
Booleanos.....	54
Números enteros de longitud variable.....	54
Caracteres.....	54

Estructuras.....	54
Tuplas.....	55
Contenedores.....	55
Punteros propietarios.....	55
Objetos opcionales.....	55
Variantes.....	55
Objetos compatibles.....	55
Codificación del esquema de datos.....	56
Cabecera.....	57
Palabra mágica.....	57
Opciones de formato.....	57
Tabla de hashes.....	59
Carga útil.....	59
Módulo de Reflexión en Tiempo de Compilación.....	59
Mecanismos de reflexión.....	59
Conteo de variables miembros de una clase.....	60
Visita a variables miembros.....	62
Acceso a variables miembros por índice de posición.....	63
Comparación de igualdad a nivel de variables miembros.....	65
Enfoque de reflexión sobre la alineación de objetos.....	66
Módulo de Serialización y Deserialización Binaria.....	68
Diseño de la interfaz de serialización.....	68
Tipos de datos serializables.....	72
Esquema de datos.....	72
Procesos de codificación.....	73
Interfaz de lectura y escritura de datos.....	75
Ordenamiento de bytes.....	76
Codificación definida por el usuario.....	76
Manejo de errores.....	78
Modos de serialización.....	79
Retrocompatibilidad.....	79
Serialización en tiempo de compilación.....	80
Módulo de Enteros de Longitud Variable.....	83
Codificaciones de enteros de longitud variable.....	83
Clase genérica `teg::varint`	85
Pruebas de Software.....	86
Herramientas de testing.....	86
Entorno de pruebas (fixtures).....	87
Casos de prueba y aserciones.....	87
Diseño, construcción y ejecución de las pruebas de software.....	89
Evaluación de Rendimiento.....	91
Diseño de los experimentos.....	91
Métricas de rendimiento.....	91

Tasa de procesamiento de datos.....	91
Tiempo total de serialización y deserialización.....	91
Ratio de compresión.....	91
Uso de la memoria principal.....	92
Uso del procesador.....	92
Herramientas de benchmarking y profiling.....	92
Google Benchmark.....	92
Windows Performance Recorder.....	93
Windows Performance Analyzer.....	93
Experimentos de evaluación de rendimiento.....	93
Experimento 1: Query e-commerce.....	93
Experimento 2: Modelo 3D.....	94
Librerías a medir y evaluar.....	95
Datos de prueba.....	96
Informe de rendimiento.....	96
Experimento 1: Query e-commerce.....	97
Experimento 2: Modelo 3D.....	101
Discusión de los resultados.....	107
Capítulo V	
Conclusiones y Recomendaciones.....	110
Conclusiones.....	110
Recomendaciones.....	112
Referencias Bibliográficas.....	113
Apéndice A.	
Uso de la Librería.....	117
A. 1 Dependencias.....	117
A. 2 Configuración e Instalación.....	117
A. 3 Serialización y Deserialización de Objetos.....	118
A. 4 Codificación y Decodificación Explícita.....	120
A. 5 Configuración del Archivo Binario.....	121
Apéndice B.	
Interfaz de Programación.....	122
B. 1 Cabeceras.....	122
B. 2 Módulos.....	123
B. 2. 1 Módulo de Reflexión Estática.....	124
B. 2. 2 Módulo de Serialización de Objetos.....	126
B. 2. 3 Módulo de enteros de longitud variable.....	128
B. 2. 4 Módulo global.....	128

Apéndice C.	
Detalles de Implementación.....	130
C. 1 Tipos serializables.....	130
C. 1. 1 Tipos de datos built-in.....	130
C. 1. 2 Contenedores.....	131
C. 1. 3 Tuplas.....	133
C. 1. 4 Tipos opcionales.....	133
C. 1. 5 Variantes.....	134
C. 1. 6 Punteros propietarios.....	134
C. 1. 6 Tipos compatibles.....	135
C. 1. 7 Codificación definida por el usuario.....	135
C. 2 Funciones de Reflexión en Tiempo de Compilación.....	136
C. 2. 1 Conteo de variables miembros en clases agregadas.....	136
C. 2. 2 Visita a variables miembros en clases agregadas.....	137
C. 2. 3 Conversión de una clase agregada a una tupla.....	138
C. 2. 4 Acceso por índice a variables miembros de una clase agregada.....	138
C. 2. 5 Comparación de igualdad miembro a miembro en clases agregadas.....	139
C. 2. 5 Detección de estructuras empaquetadas.....	140
C. 4 Interfaz de números enteros de longitud variable.....	141
C. 5 Cadena de caracteres en tiempo de compilación.....	143
C. 6 Ordenamiento de bytes.....	145
Apéndice D.	
Resultados de las Pruebas de Rendimiento.....	146
D. 1 Experimento 1: Página E-commerce.....	146
D. 1. 1 Tiempos de ejecución.....	146
D. 1. 2 Tasa de procesamiento de datos.....	147
D. 1. 3 Tamaño de los archivos binarios.....	148
D. 1. 4 Utilización de la memoria principal.....	148
D. 1. 5 Utilización del procesador.....	149
D. 2 Experimento 2: Modelo 3D.....	150
D. 2. 1 Tiempos de ejecución.....	150
D. 2. 2 Tasa de procesamiento de datos.....	151
D. 2. 3 Tamaño de los archivos binarios.....	152
D. 2. 4 Utilización de la memoria principal.....	153
D. 2. 5 Utilización del procesador.....	154
Apéndice E.	
Futuro del Lenguaje C++.....	155
Anexos.....	156
Contrato de Cesión de Derechos de Reproducción y Comunicación de Tesis de Pregrado.....	156

Índice de Tablas

Tabla 1. Plan de trabajo por semana empleado para el desarrollo de la investigación.	45
Tabla 2. Codificación de objetos fundamentales del formato de archivo binario “Teg”.	52
Tabla 3. Codificación a formato de texto de los esquemas de datos en la librería “Teg”.	54
Tabla 4. Opciones del formato de archivo binario “Teg” desarrollado en la investigación.	56
Tabla 5. Descripción general de las técnicas de compresión de enteros orientadas a bytes. Autoría de Lemire, Kurz, y Rupp (2018).	81
Tabla 6. Repertorio de pruebas unitarias y de integración de la librería “Teg”.	87
Tabla 7. Configuración de las librerías de serialización a evaluar en las pruebas de rendimiento.	93
Tabla 8. Comparación de características funcionales de las librerías de serialización binaria evaluadas en la investigación.	105
Tabla 9. Cabeceras de la librería de serialización y deserialización binaria Teg.	120
Tabla 10. Módulos de la librería de serialización y deserialización binaria Teg.	121
Tabla 11. Conceptos, clases y funciones definidas en el módulo de reflexión en tiempo de compilación.	122
Tabla 12. Conceptos, clases y funciones definidas y exportadas en el módulo de serialización y deserialización binaria de objetos.	124
Tabla 13. Conceptos, clases y funciones definidas y exportadas en el Módulo de Enteros de Longitud Variable.	126
Tabla 14. Conceptos, clases y funciones definidas y exportadas en el módulo global.	126
Tabla 15. Tiempos de serialización (ms) requeridos por las librerías de serialización en función del tamaño de los datos de prueba procesados (MiB) en el experimento nº 1.	144
Tabla 16. Tiempos de deserialización (ms) requeridos por las librerías de serialización en función del tamaño de los datos de prueba procesados (MiB) en el experimento nº 1.	144
Tabla 17. Tasa de serialización (GiB/s) de las librerías de serialización en función del tamaño de los datos de prueba procesados (MiB) en el experimento nº 1.	145
Tabla 18. Tasa de deserialización (GiB/s) de las librerías de serialización en función del tamaño de los datos de prueba procesados (MiB) en el experimento nº 1.	145
Tabla 19. Tamaño de los archivos binarios (MiB) creados por las librerías de serialización como resultado de los procesos de serialización en el experimento nº 1.	146

Tabla 20. Utilización de la memoria principal (bytes privados) de las librerías de serialización binaria en los procesos de serialización del experimento nº 1.	146
Tabla 21. Utilización de la memoria principal (bytes privados) de las librerías de serialización binaria en los procesos de deserialización del experimento nº 1.	147
Tabla 22. Utilización de la CPU (porcentaje de todos los núcleos) de las librerías de serialización binaria en los procesos de serialización en el experimento nº 1.	147
Tabla 23. Utilización de la CPU (porcentaje de todos los núcleos) de las librerías de serialización binaria en los procesos de deserialización en el experimento nº 1.	148
Tabla 24. Tiempos de serialización (ms) requeridos por las librerías de serialización en función del tamaño de los datos de prueba procesados (MiB) en el experimento nº 2.	148
Tabla 25. Tiempos de deserialización (ms) requeridos por las librerías de serialización en función del tamaño de los datos de prueba procesados (MiB) en el experimento nº 2.	149
Tabla 26. Tasa de serialización (GiB/s) de las librerías de serialización en función del tamaño de los datos de prueba procesados (MiB) en el experimento nº 2.	149
Tabla 27. Tasa de deserialización (GiB/s) de las librerías de serialización en función del tamaño de los datos de prueba procesados (MiB) en el experimento nº 2.	150
Tabla 28. Tamaño de los archivos binarios creados por las librerías de serialización binaria como resultado de los procesos de serialización en el experimento nº 2.	150
Tabla 29. Utilización de la memoria principal (bytes privados) de las librerías de serialización binaria en los procesos de serialización del experimento nº 2.	151
Tabla 30. Utilización de la memoria principal (bytes privados) de las librerías de serialización binaria en los procesos de deserialización del experimento nº 2.	151
Tabla 31. Utilización de la CPU (porcentaje de todos los núcleos) de las librerías de serialización binaria en los procesos de serialización en el experimento nº 2.	152
Tabla 32. Utilización de la CPU (porcentaje de todos los núcleos) de las librerías de deserialización binaria en los procesos de serialización en el experimento nº 2.	152

Índice de Figuras

Figura 1. Modelo de proceso incremental.	42
Figura 2. Calendario del proyecto siguiendo el modelo incremental de Pressman.	44
Figura 3. Ejemplo del conteo de variables miembros de una clase agregada. Instrucciones ensamblador generadas de la función `teg::members_count()`.	59
Figura 4. Ejemplo de utilización de la función genérica de visita aplicada a las variables miembros de una clase agregada.	60
Figura 5. Demostración de la función `teg::tie_members` que ata las variables miembro de una clase agregada a una tupla de referencias.	62
Figura 6. Demostración de la función `teg::get_member` que permite el acceso a variables miembros de una clase agregada mediante sus índices de posición.	62
Figura 7. Ejemplo de la comparación de igualdad a nivel de variables miembros de una clase agregada.	64
Figura 8. Ejemplo de detección de estructuras empaquetadas utilizando el Concepto `teg::concepts::packed_layout`.	65
Figura 9. Diagrama de secuencia UML del proceso de serialización binaria de objetos.	66
Figura 10. Diagrama de secuencia UML del proceso de deserialización binaria de objetos.	67
Figura 11. Diagrama de secuencia UML del proceso de codificación binaria de objetos.	68
Figura 12 Diagrama de secuencia UML del proceso de decodificación binaria de objetos.	69
Figura 13. Demostración de la codificación del esquema de datos a un formato de texto y su digestión MD5.	71
Figura 14. Diagrama de clases UML de los codificadores y decodificadores binarios y las interfaces de lectura y escritura implementadas en la librería “Teg”.	72
Figura 15. Ejemplo de definición de una clase de escritura personalizada para utilizar la interfaz de codificación binaria.	73
Figura 16. Ejemplo de codificación explícita definida por el usuario en una clase que no es agregada.	76
Figura 17. Manejo de errores en la librería. El recuadro superior ilustra el uso de códigos de error, mientras que el recuadro inferior demuestra el uso de excepciones.	77
Figura 18. Retrocompatibilidad entre dos versiones de esquemas de datos.	78
Figura 19. Instrucciones de lenguaje ensamblador generadas en la serialización en tiempo de compilación de un número entero fijo de 32-bits.	79
Figura 20. Diagrama de flujo del algoritmo de codificación y decodificación de números enteros sin signos ULEB-128.	82
Figura 21. Ejemplo de la codificación ZigZag y codificación ULEB-128 en un número entero con signo de 64 bits.	83

Figura 22. Macros de aserciones definidos en la librería de testing desarrollada para el proyecto de investigación.	86
Figura 23. Ejemplo de aserciones en tiempo de compilación, herramienta especializada de la librería de testing del proyecto.	86
Figura 24. Esquema de los datos de prueba del experimento #1.	92
Figura 25. Esquema de los datos de prueba del experimento #2.	92
Figura 26. Generación de datos de prueba en función de un tamaño en bytes.	94
Figura 27. Tasa de serialización de objetos (GiB/s) en función del tamaño de los datos de prueba (MiB) en el experimento #1.	95
Figura 28. Tasa de deserialización de objetos (GiB/s) en función del tamaño de los datos de prueba (MiB) en el experimento #1.	96
Figura 29. Tiempo total de serialización y deserialización (ms) de las librerías en función del tamaño de los datos de prueba (MiB) en el experimento #1.	97
Figura 30. Tamaño de los archivos binarios (MiB) generados en el proceso de serialización de objetos en función del tamaño de los datos prueba (MiB) en el experimento #1.	97
Figura 31. Uso de la memoria principal (MiB) en el proceso de serialización de objetos en el experimento #1.	98
Figura 32. Uso de la memoria principal (MiB) en el proceso de deserialización de objetos en el experimento #1.	98
Figura 33. Tasa de serialización de objetos (GiB/s) en función del tamaño de los datos de prueba (MiB) en el experimento #2.	99
Figura 34. Tasa de deserialización de objetos (GiB/s) en función del tamaño de los datos de prueba (MiB) en el experimento #2.	100
Figura 35. Tiempo total de serialización y deserialización (ms) requerido por las librerías de serialización binaria en función del tamaño de los datos de prueba (MiB) en el experimento #2.	100
Figura 36. Tamaño de los archivos binarios (MiB) generados en el proceso de serialización de objetos en función del tamaño de los datos prueba (MiB) en el experimento #2.	101
Figura 37. Ahorro de espacio en los archivos binarios generados por las librerías en comparación al tamaño de los datos procesados en el experimento #2.	102
Figura 38. Uso de la memoria principal (MiB) durante la serialización de objetos en el experimento #2.	103
Figura 39. Uso de la memoria principal (MiB) durante la deserialización de objetos en el experimento #2.	103
Figura 40. Utilización del CPU (porcentaje de todos los núcleos) en los procesos de serialización y deserialización (cuadro superior e inferior respectivamente) de 1 MiB de datos de prueba en el experimento #2.	104
Figura 41. Comandos de configuración e instalación de la librería Teg.	115

Figura 42. Ejemplo de uso de la librería: serialización de una factura.	116
Figura 43. Ejemplo de uso de la librería: deserialización de una factura.	117
Figura 44. Ejemplo de uso de la librería: codificación explícita definida por el usuario de una clase no agregada.	118
Figura 45. Ejemplo de uso de la librería: configuración del archivo binario.	119
Figura 46. Implementación de los Conceptos que modelan tipos de datos fundamentales, clases, arreglos y las propiedades de estos tipos.	128
Figura 47. Implementación del Concepto de contenedor `teg::conceptos::container` según los requerimientos especificados por el estándar del lenguaje ISO/IEC 14882:2020.	129
Figura 48. Implementación de Conceptos de contenedores específicos según las propiedades inherentes de estos como `teg::conceptos::contiguos_container` y `teg::conceptos::fixed_size_container`.	130
Figura 49. Implementación de los Conceptos para verificar tuplas (e interfaces que se comportan como tuplas).	131
Figura 50. Implementación del Concepto `teg::concepts::optional` usado para detectar tipos opcionales.	131
Figura 51. Implementación del Concepto `teg::concepts::variant` usado para comprobar que un tipo es exactamente la clase estándar `std::variant`.	132
Figura 52. Implementación de los Conceptos de punteros `teg::concepts::unique_ptr`, `teg::concepts::shared_ptr` y `teg::concepts::owning_ptr`.	132
Figura 53. Implementación del Concepto `teg::concepts::compatible` usado para detectar clases de retrocompatibilidad en el formato de archivo binario.	133
Figura 54. Implementación del Concepto `teg::concepts::compatible` usado para detectar clases de retrocompatibilidad en el formato de archivo binario.	133
Figura 55. Implementación de la función genérica de reflexión `teg::members_count`.	134
Figura 56. Implementación de la función genérica de reflexión `teg::visit_members`.	135
Figura 57. Implementación de la función genérica de reflexión `teg::tie_members`.	136
Figura 58. Implementación de la función genérica de reflexión `teg::get_member`.	136
Figura 59. Implementación de la función genérica de reflexión `teg::memberwise_equal`.	137
Figura 60. Implementación del Concepto `teg::packet_layout` utilizado para identificar estructuras de datos empaquetadas.	138
Figura 61. Clase genérica `teg::varint`, interfaz utilizada para serializar números de longitud variable.	139
Figura 62. Codificación de la clase genérica `teg::varint`.	140
Figura 63. Decodificación de la clase genérica `teg::varint`.	140
Figura 64. Clase genérica base `teg::basic_fixed_string`, interfaz utilizada para implementar las cadenas de caracteres en tiempo de compilación.	142

Figura 65. Alias de la interfaz de cadenas de caracteres en tiempo de compilación utilizadas para definir el tamaño de los caracteres en bits. Por defecto se utiliza `teg::fixed_string`.	142
Figura 66. Conceptos utilizados para determinar cuándo se debe realizar un cambio de ordenamiento de bytes en los procesos de codificación y decodificación.	143
Figura 67. Implementación de la función genérica de reflexión `teg::visit_members` en C++26.	153

Sinopsis

La serialización es el proceso que convierte datos estructurados hacia un formato persistente para su almacenamiento o transmisión; la deserialización los restaura a su forma original. En C++, la ausencia de un mecanismo de reflexión estándar ha limitado las capacidades de las librerías de serialización, porque obliga a los desarrolladores a implementar soluciones ad-hoc basadas en generación de código, macros de preprocesador y código de *boilerplate* repetitivo y propenso a errores, incrementando la complejidad, reduciendo la mantenibilidad y comprometiendo la portabilidad del software. No obstante, características introducidas en el estándar ISO/IEC 14882:2020 permiten la implementación de un novedoso enfoque de reflexión en tiempo de compilación, creando oportunidades de mejora del software desarrollado en este lenguaje. La presente investigación tuvo como objetivo desarrollar una librería de serialización binaria basada en dicho enfoque. Para ello, se utilizó una metodología de software incremental, comenzando con el diseño de una especificación de formato binario portable, seguido de tres incrementos: Módulo de Reflexión Estática, Módulo de Serialización de Objetos y Módulo de Números Enteros de Longitud Variable. Finalmente, se realizaron pruebas de rendimiento donde se evidenció una alta eficiencia de desempeño por parte de la librería Teg, producto del presente trabajo, en términos de tasa de procesamiento de datos, uso de memoria, carga del procesador y compresión de archivos. En un procesador Intel de 3.4 GHz, Teg reportó una tasa de procesamiento máxima de 21 GiB/s, siendo hasta 20 y 100 veces más rápida que las librerías más utilizadas en la industria, como Protocol Buffers, MessagePack y Boost.Serialization.

Palabras clave: serialización binaria, deserialización binaria, reflexión en tiempo de compilación, lenguaje de programación C++20, ISO/IEC 14882:2020.

Introducción

La serialización es el proceso mediante el cual datos estructurados o estados de objetos se convierten a un formato persistente para su almacenamiento o transmisión, mientras que la deserialización es el proceso inverso, que permite restaurar dichos datos a su forma original (Cassey, 2022). En los últimos años, la búsqueda de la optimización de estos procesos ha cobrado especial relevancia con la proliferación de los Sistemas Distribuidos, la Computación Paralela, los Sistemas Embebidos, el Internet de las Cosas (IoT) y la Inteligencia Artificial (IA), ámbitos en los que la eficiencia en la transferencia y almacenamiento de datos es crítica para el desempeño global (Carrera Castillo, Rosales y Torres Blanco, 2018).

La publicación del estándar ISO/IEC 14882:2020 (2020) ha abierto la oportunidad de implementación de un mecanismo de reflexión en tiempo de compilación en el lenguaje de programación C++, facilitando así, la técnica de introspección del código fuente para que éste sea tratado como un tipo de dato, requerida para la implementación de librerías de serialización (Tauro, Ganesan, Mishra y Bhagwat, 2012). Antes de la aparición de este estándar, desarrollar una librería de serialización en C++ obligaba a utilizar soluciones externas que afectan negativamente la mantenibilidad, reusabilidad y portabilidad del software (Qi, 2022). La incorporación de un mecanismo de reflexión con prácticas estándar abre nuevas posibilidades para implementar librerías más eficientes y robustas.

En este contexto, la presente investigación tiene como objetivo desarrollar una librería de serialización y deserialización binaria aplicando un nuevo enfoque basado en la reflexión de tiempo de compilación con C++20, que promueva características de calidad, en concordancia con el estándar ISO/IEC 25010:2023 (2023), en particular, la eficiencia de desempeño, en cuanto a la tasa de procesamiento y uso de recursos; la compatibilidad, mediante la interoperabilidad entre plataformas y la retrocompatibilidad de formato; la mantenibilidad, lograda a través de una arquitectura modular y estrategias de diseño que fomentan la reusabilidad; y la flexibilidad, haciendo especial énfasis en la adaptabilidad del software en diferentes entornos, y escalabilidad según la carga de trabajo.

En cuanto al aspecto metodológico, se adopta la metodología de desarrollo de software incremental, planteada por Pressman (2010). Se parte del diseño de una especificación de formato de archivo binario portable, para posteriormente implementar tres

módulos de manera incremental: Módulo de Reflexión Estática, Módulo de Serialización Objetos, y Módulo de Enteros de Longitud Variable. Por último, se planifican y se ejecutan pruebas de rendimiento, con el fin de comparar la solución propuesta con respecto a las librerías de serialización más populares en la industria del software.

El presente trabajo se estructura en cinco capítulos: el primero aborda el problema de investigación, presentando la solución propuesta, así como los objetivos, el alcance y las limitaciones del estudio. En el segundo capítulo, se examina la base conceptual que sustenta la investigación, recopilando y analizando antecedentes relevantes, además de profundizar en las teorías, conceptos y el estado del arte que se relacionan con el tema en cuestión. En el tercer capítulo, se detalla el marco metodológico, justificando las metodologías empleadas y las fases del desarrollo de la investigación. El cuarto capítulo describe el proceso de desarrollo y se presentan los resultados obtenidos. Finalmente, el quinto capítulo expone las conclusiones de la investigación y las recomendaciones para futuros trabajos en áreas relacionadas.

Capítulo I

El Problema

El presente capítulo aborda la identificación y descripción del problema central que se investiga. Se propone una solución a dicho problema y se puntuiza la investigación a través de sus objetivos, alcances y limitaciones. Finalmente, se justifica la formulación del estudio y desarrollo según su aporte, impacto y contribución.

Planteamiento del Problema

La serialización es el proceso de conversión de datos estructurados o estados de objetos a un formato persistente para su almacenamiento o transmisión (Cassey, 2022). A su vez, el proceso inverso es conocido como deserialización y consiste en reconstruir los datos a su estado original. Estas transformaciones, fundamentales para manipular, guardar y enviar datos, son especialmente relevantes en campos de la Informática como la Inteligencia Artificial, los Sistemas Distribuidos, la Computación en la Nube, los Sistemas de Gestión de Bases de Datos, la Interoperabilidad de Aplicaciones, la Computación Paralela y de Alto Rendimiento, y el Internet de las Cosas.

Márton y Porkolab (2010) explican que el lenguaje de programación C++ carece de una librería de serialización estándar debido a que en su diseño no existe un sistema de reflexión. La reflexión permite a un lenguaje manipular información sobre sí mismo (Demers y Malenfant, 1995), y es fundamental para la implementación de librerías de serialización en varios lenguajes, como Java y C# (Tauro, Ganesan, Mishra y Bhagwat, 2012). El lenguaje C++ no proporciona un mecanismo de serialización abstracto y de alto nivel pero soporta la codificación de tipos de datos fundamentales en formatos binarios (Casey, 2022). Por esto, una alternativa que utilizan algunos desarrolladores de software consiste en escribir manualmente funciones de codificación y decodificación para cada estructura o clase a ser transformada, pero esta práctica desemboca en código repetitivo, tedioso y propenso a errores (Soukup y Macháček, 2014).

Algunos programadores optan por utilizar librerías de serialización desarrolladas en C++ como Protocol Buffers, MessagePack y Boost.Serialization. Sin embargo, Qi (2022) critica estas librerías por no ser soluciones modernas ni efectivas, debido a que, con la

finalidad de lograr un cierto grado de introspección e intercesión¹ hacen uso intensivo de macros, dependen de la generación de código a través de compiladores externos con características no estándar o incurren en código intrusivo. Estas prácticas conllevan a problemas como dificultades para probar y depurar el código, violación del principio de encapsulación, altos niveles de acoplamiento y dependencia entre componentes, y limitaciones en la mantenibilidad, escalabilidad y reutilización del código.

Las versiones estándar ISO/IEC 14882:2017 e ISO/IEC 14882:2020, conocidas informalmente como C++17 y C++20 respectivamente, incorporan soporte y funcionalidades en el lenguaje para la evaluación de estructuras de control y expresiones en tiempo de compilación. Dichas facilidades abren la oportunidad de implementar una librería de reflexión estática² de manera estándar (Qi, 2017). La reflexión en tiempo de compilación permite que el compilador maneje la obtención de información de los tipos de datos, facilitando la creación de funciones genéricas de serialización y deserialización con métodos estándar, sin necesidad de código generado ni intrusivo. Qi (2022) también demuestra que se puede prescindir de la serialización de metadatos o esquemas, lo cual afecta positivamente el rendimiento, reduce el tamaño de los archivos binarios, y mejora la velocidad de serialización y deserialización de objetos.

Con la creciente complejidad de los sistemas informáticos de comunicación y almacenamiento, como los Sistemas Distribuidos y el Internet de las Cosas, optimizar los procesos de serialización y deserialización se ha vuelto una necesidad (Carrera Castillo, Rosales y Torres Blanco, 2018; Casey, 2022). El trabajo de Qi (2022) introduce un enfoque innovador que puede ser estudiado y aplicado para dar solución a estas necesidades. Sin embargo, este enfoque aún requiere de más investigación para su integración en entornos de producción. En este sentido, existen dos aspectos clave que no se consideran en el diseño original y que son prioritarios: el primero es la portabilidad, que garantiza la consistencia de los archivos binarios y los algoritmos de serialización entre diferentes arquitecturas; y el segundo es la codificación eficiente de números enteros, con la posibilidad de explorar codificaciones de enteros de longitud variable.

¹ La introspección e intercesión son las dos cualidades que conforman la reflexión. La primera es la facultad de obtener información del código fuente, y la segunda, la capacidad de modificar dicha información (Demers y Malenfant, 1995).

² La reflexión en tiempo de compilación y reflexión en tiempo de ejecución son comúnmente conocidas como reflexión estática y reflexión dinámica respectivamente.

En el modelo propuesto por Qi (2022) el compilador define el tamaño, alineación y orden de bytes de los datos basándose en la arquitectura para la cual se está compilando, lo que resulta en archivos binarios dependientes de dicha arquitectura. Este diseño puede ser utilizado para almacenar datos de manera local pero no para transmitir ni almacenar información entre diferentes plataformas; no es portable. Las arquitecturas de computadoras varían en cómo organizan los bytes y alinean los datos en memoria (Tanenbaum y Austin, 2013). Una librería de serialización necesita gestionar internamente estas diferencias para prevenir inconsistencias al interpretar archivos serializados en distintas plataformas (Chiu, 2004). Por ejemplo, Protocol Buffers y FlatBuffers son librerías de serialización que garantizan la portabilidad entre sistemas; la primera mediante la utilización de un esquema³ de codificación, mientras que la segunda logra esto a través de macros⁴.

Surge también el desafío de gestionar eficientemente la representación de números enteros. Librerías de serialización como Protocol Buffers utilizan codificaciones de enteros de longitud variable para optimizar el tamaño ocupado de los archivos binarios (Creager, 2021), esto aprovecha la serialización de enteros que son pequeños y, por lo tanto, que pueden ser representados con una menor cantidad de bits de la que normalmente necesitan. Esta técnica permite una reducción significativa en el espacio de almacenamiento y mejora la eficiencia en la transmisión de datos. Existe la oportunidad de analizar y adaptar mecanismos de codificación y decodificación de enteros de longitud variable con el enfoque de reflexión en tiempo de compilación para ser integrados al formato de archivo binario y a los algoritmos de serialización y deserialización.

Por el planteamiento anterior, se propone desarrollar una librería de serialización y deserialización binaria basada en reflexión en tiempo de compilación con ISO/IEC 14882:2020 (C++20). Se partirá del diseño inicial de Qi (2022) y se investigará e implementará una solución con prácticas estándar y modernas para la configuración de alineación de datos, orden de bytes y codificación de enteros de longitud variable. Estas funcionalidades serán desarrolladas bajo el emergente enfoque de evaluación de expresiones en tiempo de compilación y permitirán crear un formato de archivo binario portable entre sistemas de distintas arquitecturas. Con base en dicho formato se desarrollará un módulo de reflexión estática, un módulo de serialización y deserialización de objetos, y un módulo de codificación de enteros de longitud variable. Cada módulo contará con sus respectivas

³ Esquema de Protocol Buffers: <https://protobuf.dev/programming-guides/encoding/#cheat-sheet>

⁴ Especificación interna de FlatBuffers: https://flatbuffers.dev/flatbuffers_internals.html

pruebas unitarias y pruebas de integración. Finalmente, la librería de serialización será medida y evaluada con respecto a las librerías más utilizadas en la industria en términos de tamaños de archivos binarios generados y tiempo de serialización y deserialización de objetos.

Objetivos

Objetivo general.

Desarrollar una librería de serialización y deserialización binaria basada en reflexión en tiempo de compilación con ISO/IEC 14882:2020 (C++20).

Objetivos específicos.

1. Diseñar un formato de archivo binario portable con soporte de alineación de datos y orden de bytes.
2. Desarrollar un módulo de reflexión en tiempo de compilación para la introspección y generación de metadatos de tipos de datos agregados.
3. Desarrollar un módulo de serialización y deserialización binaria de objetos para tipos de datos agregados.
4. Desarrollar un módulo de codificación y decodificación binaria de enteros de longitud variable.
5. Construir un conjunto de pruebas unitarias y pruebas de integración para la aserción de las funcionalidades de la librería.
6. Evaluar el rendimiento de la librería en términos de eficiencia en el tamaño de archivos binarios generados, tiempo de serialización y deserialización de objetos, y uso de la CPU y la memoria RAM durante los procesos de serialización y deserialización.

Alcance

1. Diseñar un formato de archivo binario portable con soporte de alineación de datos y orden de bytes.

Se creará una especificación para un formato de archivo binario portable basado en el diseño de Qi (2022). Este formato facilitará la serialización y deserialización de objetos en C++ de manera eficiente y con prácticas estandarizadas. Se añadirá soporte para la

configuración y manejo de alineación de datos y el ordenamiento de byte, asegurando la independencia y portabilidad entre diferentes arquitecturas de computadoras.

El formato de archivo binario tendrá soporte para representar tipos de datos fundamentales: enteros, números de punto flotante y caracteres; tipos compuestos: clases, estructuras, arreglos, y contenedores genéricos como vectores, pilas, colas, conjuntos y mapas *hash*; también tiposopcionales usando `std::optional` y variantes mediante `std::variant`. No se permitirá la serialización de punteros crudos ni punteros nulos, pero sí la serialización de punteros inteligentes a través de `std::unique_ptr` y `std::shared_ptr`. No se prevé soporte para estructuras de datos con referencias circulares como listas enlazadas y grafos cíclicos. La compatibilidad con versiones anteriores del formato será gestionada mediante una clase genérica, aunque no se soportará la compatibilidad con versiones posteriores. Dado que el formato de archivo binario no contendrá esquema (o bien tendrá un esquema mínimo), se implementará un mecanismo de validación de los tipos de datos el cual se situará en una cabecera anterior a la carga útil.

2. Desarrollar un módulo de reflexión en tiempo de compilación para la introspección y generación de metadatos de tipos de datos agregados.

Se desarrollará un módulo de reflexión estática utilizando las últimas funcionalidades de evaluación de expresiones en tiempo de compilación incorporadas en C++20 implementando la capacidad de introspección de clases. Este módulo cumplirá con dos propósitos principales: la obtención de metadatos y la aplicación de una función genérica de visita a todas las variables miembros públicas de una clase de forma recursiva. Únicamente se podrá reflejar tipos de datos agregados, debido a que los tipos de datos no-agregados contienen miembros privados o protegidos y es imposible su inspección sin incurrir en código intrusivo.

La funcionalidad de manejo de alineación de datos será diseñada e implementada a través de un mecanismo de expresiones constantes en tiempo de compilación compatible e integrado con el módulo de reflexión. Serán consideradas las dos facilidades actuales disponibles en el lenguaje para ajustar la alineación de estructuras: el especificador de alineación `alignas` y la directiva de preprocesador `#pragma pack`.

Se emplearán Plantillas y Conceptos para establecer las parametrizaciones y restricciones genéricas de la interfaz de programación con la finalidad de proporcionar una usabilidad clara y robusta.

3. Desarrollar un módulo de serialización y deserialización binaria de objetos para tipos de datos agregados.

Se desarrollará un módulo de serialización y deserialización binaria bajo el paradigma de programación orientada a objetos, programación genérica y programación modular. La funcionalidad principal será convertir estructuras y estados de objeto en memoria a formatos de archivos binarios para su posterior manipulación, transmisión o almacenamiento. De manera inversa, serán proporcionadas funciones para restaurar los datos hacia la memoria desde archivos binarios válidos.

Este módulo interpretará el formato de archivo binario diseñado y dependerá directamente del módulo de reflexión en tiempo de compilación. Solo será posible procesar tipos de datos agregados debido a la misma restricción presente en el módulo de reflexión. Se implementarán funciones de validación del formato de archivo binario. Los errores de validación y de restricciones de tipos incorrectos, como la detección de tipos no-agregados o referencias circulares, serán manejados mediante dos mecanismos estándar: excepciones y códigos de error. Esto a razón de dar soporte de los mecanismos de manejo de errores más comunes e incluso los únicos disponibles en sistemas con recursos restringidos (Bonifácio et. al., 2015; Stroustrup, 2019).

La funcionalidad de manejo de orden de bytes será diseñada e implementada con un enfoque de evaluación de expresiones en tiempo de compilación. Serán proporcionados funciones de detección y de conversión entre *little-endian* y *big-endian*. El módulo de serialización inspeccionará los tipos agregados tomando en consideración el manejo de alineación de datos disponible en el módulo de reflexión estática. De esta manera se garantizará portabilidad entre distintas arquitecturas de computadoras.

4. Desarrollar un módulo de codificación y decodificación binaria de enteros de longitud variable.

Se analizará los métodos de codificación de enteros de longitud variable recopilados por Lemire, Kurz y Rupp (2018) y Creager (2021) para ser aplicados e integrados a la librería

de serialización siguiendo un enfoque basado en reflexión en tiempo de compilación. Posteriormente, se desarrollará un módulo donde estarán contenidas las clases y funciones de codificación y decodificación de enteros de longitud variable según los métodos seleccionados. Las clases y funciones definidas en el módulo podrán ser exportadas y usadas en nuevas clases definidas por el usuario para así habilitar la codificación de enteros de manera explícita.

5. Construir un conjunto de pruebas unitarias y pruebas de integración para la aserción de las funcionalidades de la librería.

Se construirá un conjunto de pruebas unitarias y pruebas de integración para la aserción de las funcionalidades de la librería según sus tres módulos: módulo de reflexión estática, módulo de serialización y deserialización de objetos, y módulo de codificación de enteros de longitud variable. Se implementarán pruebas de inspección de metadatos de clases, funciones de visita en tiempo de compilación, funciones de serialización y deserialización de clases, pruebas de manejo y configuración de alineación de datos y ordenamiento de bytes, y funciones de codificación y decodificación de enteros de longitud variable.

6. Evaluar el rendimiento de la librería en términos de eficiencia en el tamaño de archivos binarios generados, tiempo de serialización y deserialización de objetos, y uso de la CPU y la memoria RAM durante los procesos de serialización y deserialización.

Se construirá un conjunto de pruebas de rendimiento para la evaluación de la librería según los siguientes criterios: tamaño de los archivos binarios generados, tiempo de serialización de objetos y tiempo de deserialización de objetos, uso de la CPU y uso de la memoria RAM durante los procesos de serialización y deserialización. Estas mismas pruebas se construirán, medirán y compararán con las librerías de serialización binaria más utilizadas en la industria según Biswal y Almallah (2019) y Qi (2022), que son Protocol Buffers, FlatBuffers, MessagePack y Boost.Serialization.

Para las pruebas de rendimiento se seleccionarán diversas estructuras de datos complejas con la finalidad de lograr una evaluación exhaustiva de los procesos de serialización y deserialización. Se analizarán los resultados obtenidos y se escribirá un informe de rendimiento detallado con los resultados donde también estarán documentadas las configuraciones específicas de cada prueba y sus respectivas ejecuciones.

Limitaciones

- La librería será desarrollada en C++20 haciendo uso de funcionalidades incorporadas en el estándar ISO/IEC 14882:2020. La compatibilidad con versiones anteriores del lenguaje C++ será limitada o inviable.
- La librería será desarrollada empleando el paradigma de programación orientada a objetos, programación genérica y programación modular.
- Las herramientas de *testing* utilizadas para las pruebas unitarias y pruebas de integración serán desarrolladas utilizando el lenguaje C++ con funciones de la librería estándar; no se utilizarán librerías de terceros.
- Para la compilación de la librería, compilación de las pruebas unitarias y de integración, y compilación de las pruebas de rendimiento se utilizará el compilador MSVC⁵ en su versión 19.38.33135. La arquitectura objetivo de los archivos binarios generados será Windows de 64 bits (x86-64).
- Para la compilación y ejecución de las pruebas unitarias, pruebas de integración y pruebas de rendimiento se utilizará una computadora con un procesador Intel i5-4690S 3.4 GHz de arquitectura x86-64 y memoria RAM 32 GB *dual-channel* DDR3 1600 MHz, y sistema operativo Windows 11 Pro.
- Todas las pruebas de rendimiento serán ejecutadas en un solo hilo; no habrá paralelización.

Justificación

La serialización y deserialización son procesos fundamentales y la búsqueda de su optimización ha cobrado especial importancia al proliferar los Sistemas Distribuidos, la Computación Paralela y la Computación de Alto Rendimiento (Casey, 2022), también por el rápido desarrollo y surgimiento de los Sistemas Embebidos y el Internet de las Cosas (Biswal y Almallah, 2018), y son procesos cruciales en la Inteligencia Artificial (Fu, 2023). La serialización de objetos es un elemento clave a optimizar para reducir los tiempos de transferencia, la saturación de las redes, mejorar el procesamiento de datos y ahorrar recursos al almacenar información (Carrera Castillo, Rosales y Torres Blanco, 2018). Cada milisegundo ahorrado en los procesos de serialización puede traducirse en mejoras exponenciales en la eficiencia operativa de millones de aplicaciones.

⁵ Microsoft Visual C++ es un compilador desarrollado por Microsoft que posee soporte completo de las características estándar incorporadas en C++17 y C++20 requeridas para la librería de serialización.

El lenguaje de programación C++ es utilizado por millones de programadores en prácticamente todos los dominios de aplicación (Stroustrup, 2013). Catalogado como un lenguaje de sistemas, es principalmente empleado en el desarrollo de Redes y Comunicaciones, el Internet de las Cosas y la Computación de Alto Rendimiento (Li, Meng, Li y Cai, 2021), también en Sistemas Embebidos y la Computación Gráfica (Kazakova, 2024). Según Kazakova (2024), el uso de C++11 y C++14 ha migrado hacia nuevas versiones del lenguaje en los últimos siete años, habiendo una adopción fuerte hacia C++17 pero aún mayor hacia C++20, existiendo una adopción significativa del 29% en 2023 de esta última versión estándar.

Las nuevas funcionalidades añadidas en C++17 y C++20 permiten el desarrollo de soporte para la reflexión en tiempo de compilación con prácticas modernas (Qi, 2017), un mecanismo del cuál se ha tenido un alcance prácticamente nulo desde la primera versión estándar del lenguaje (Márton y Porkolab, 2010). Un enfoque de serialización y deserialización de objetos basado en reflexión en tiempo de compilación es capaz de generar un formato binario de datos más compacto y más rápido de procesar comparado con las librerías más utilizadas en la industria como Protocol Buffers, MessagePack y Boost.Serialization (Qi, 2022). Sin embargo, se deben considerar aspectos clave adicionales para asegurar que una librería implementada con dicho enfoque sea efectiva y práctica en entornos de producción. La presente investigación se centrará en dos de esos aspectos faltantes: portabilidad y codificación eficiente de números enteros.

Primero, la portabilidad es esencial para garantizar que los archivos binarios funcionen de manera consistente en diversas arquitecturas de hardware. Esto implica que los datos serializados deben ser interpretables y utilizables sin errores o incompatibilidades, independientemente de la plataforma en la que se encuentren. Segundo, la eficiencia en la codificación de números enteros es crucial para mejorar el rendimiento y reducir el tamaño de los archivos. Específicamente, se debe considerar el uso de técnicas avanzadas como la codificación de enteros de longitud variable, que permite representar números pequeños con menos bits, optimizando así el espacio de almacenamiento y la velocidad de procesamiento.

El desarrollo de una librería de serialización y deserialización binaria basada en un enfoque de reflexión en tiempo de compilación con ISO/IEC 14882:2020 (C++20) representará una solución moderna, portable y eficiente para la serialización de objetos. Esta librería brindará beneficios significativos a diversos beneficiarios y en diversos niveles. Los

desarrolladores que utilicen C++20 se beneficiarán de una herramienta que simplificará y optimizará los procesos de serialización, reduciendo el tiempo y el esfuerzo necesario para escribir y mantener código de serialización.

En el ámbito tecnológico, las empresas de tecnología podrán incorporar esta librería en sus aplicaciones, logrando una mejora en la eficiencia operativa de sus sistemas de comunicación y almacenamiento, lo que se traducirá en mayor rendimiento y menor consumo de recursos. A nivel económico, la optimización en el uso de recursos computacionales y el manejo eficiente de datos permitirán a las empresas y organizaciones reducir costos asociados con la infraestructura de datos y el ancho de banda necesario para la transmisión de información. Instituciones educativas y de investigación también se verán beneficiadas, ya que esta librería facilitará la investigación de algoritmos y formatos más eficientes para la serialización de datos, promoviendo el avance del conocimiento en el área.

Además, los usuarios finales de aplicaciones de comunicación y almacenamiento experimentarán mejoras en la velocidad y eficiencia de las aplicaciones que utilizan esta librería, proporcionando una mejor experiencia de usuario. Finalmente, la adopción de esta librería incentivará el uso de herramientas modernas y estandarizadas, fomentando la mejora continua en los procesos de desarrollo de software y la implementación de prácticas más eficientes y efectivas.

Capítulo II

Marco Teórico

En este capítulo se aborda la base conceptual que sustenta el estudio; se recopilan y analizan los antecedentes, y se profundiza en las teorías, conceptos y el estado del arte relacionados a la presente investigación.

Bases Teóricas

Serialización y deserialización binaria.

Según Casey (2022), la serialización binaria se refiere al proceso de convertir un objeto de datos dado o una estructura de datos en una secuencia de bytes para que pueda ser almacenada en disco o memoria, o ser transmitida a un sistema diferente. Esta versión serializada debe preservar la estructura y el estado del objeto de datos para que pueda ser reconstruido correctamente más tarde, potencialmente por un sistema diferente al de su origen, en un proceso conocido como deserialización. Tauro et. al. (2012) también definen la serialización como el proceso de convertir una estructura de datos o el estado de un objeto en un formato almacenable, y la deserialización como la resurrección de los datos almacenados en el mismo o en otro entorno informático.

Serializar tipos de datos fundamentales o primitivos resulta trivial, pero serializar estructuras con referencias, sobre todo referencias cíclicas, resulta mucho más complejo (Casey, 2022). Por este motivo, las librerías de serialización utilizan mecanismos de reflexión para inspeccionar objetos y poder condicionar de manera genérica la conversión de todos los miembros de una clase, y su vez los propios elementos contenidos en estos miembros de manera recursiva (Tauro, et al., 2012).

Para Soukup y Macháček (2014) la serialización es una técnica para implementar objetos que son persistentes pero no de manera automática. En este sentido, la finalidad que se le dé a los datos serializados es independiente del proceso. Una práctica habitual consiste en la escritura de datos serializados en un medio de almacenamiento, como podría ser un disco o una base de datos, con el fin de conservar un objeto o estructura para su utilización futura. En circunstancias similares, resulta conveniente el almacenamiento temporal en memoria de objetos o

estructuras en su forma serializada, funcionando este método como un sistema de caché para aquellos elementos que se prevea reutilizar en breve. Otra aplicación extendida de la serialización es la transmisión de datos serializados a través de redes, facilitando el intercambio de información en el envío de mensajes entre dos sistemas. Tal enfoque se adopta ampliamente en diversas áreas, incluidas las interacciones cliente-servidor de las aplicaciones web y la comunicación entre nodos en el marco de la computación distribuida en clústeres.

Reflexión computacional.

Según Márton y Porkolab (2010), la reflexión es la capacidad de un programa para inspeccionar y modificar su propia estructura; se hace referencia a la reflexión como la meta-information asociada con estructuras de programación como tipos y funciones. Para Demers y Malenfant (1995), la reflexión es la capacidad integral que tiene un programa de observar o cambiar su propio código, así como todos los aspectos de su lenguaje de programación: sintaxis, semántica e implementación.

Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession. Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called reification. (Demers y Malenfant, 1995, sección 2, párr. 5)

Como explican Demers y Malenfant, la reflexión se compone de dos aspectos: la introspección y la intercesión. La introspección es la capacidad de un programa para observar y razonar sobre su propio estado, mientras que la intercesión es la capacidad de un programa para modificar su estado de ejecución o alterar su propia interpretación o significado.

Existe una gran cantidad de tareas de computación cuya programación suele ser más apta para máquinas que para humanos puesto que representan la escritura de código repetitivo, tedioso, y propenso a errores (Soukup y Macháček, 2014); por ejemplo, la serialización y deserialización de datos en formatos basados en texto y

formatos binarios, el mapeo relacional de objetos (*ORM*), la creación de objetos simulados para pruebas (*mocks*), el soporte para programación de scripting, soporte para las llamadas a procedimientos remotos (*RPC*) e invocación de métodos remotos (*RMI*), la inspección y manipulación de objetos mediante interfaces gráficas de usuario, y la implementación de patrones de diseño de manera semiautomática o automática.

La Informática consiste en el tratamiento automático de la información por medio de programas de computadoras, y, en sí mismo, los programadores desarrollan software para automatizar procesos utilizando y operando sobre datos. Siguiendo este orden de ideas, el código fuente de un programa puede ser tratado como un dato, y por ende, este puede ser manipulado de manera automática. A esto último se le conoce como metaprogramación (Lilis y Savidis, 2019); programas que programan otros programas.

Reflexión en tiempo de compilación.

Márton y Porkolab (2010) definen la reflexión en tiempo de compilación como el mecanismo por el cual se obtiene información que es interna del compilador y que puede ser modificada para alterar el Árbol de Sintaxis Abstracta (del inglés, *Abstract Syntax Tree, AST*). La reflexión estática ocurre al momento en que se invoca el compilador para compilar código fuente. Este mecanismo no presenta sobrecargas en tiempo de ejecución, ni necesita de la inclusión de metadatos al código objeto⁶, además, se pueden seguir aplicando todas las fases de optimización del compilador, pues la intercesión ocurre en fases tempranas y es transparente hacia los demás procesos.

La reflexión estática presenta ventajas en el ámbito de seguridad, rendimiento en tiempo de ejecución y tamaño del código objeto, pero es limitada con respecto a la reflexión dinámica, pues no existe forma de inspeccionar ni interceder los objetos en tiempo real. Cuando los procesos de análisis sintáctico y análisis lexicográfico han culminado el compilador tiene acceso completo al AST, y de este es capaz de obtener tablas con todos los

⁶ Es el código que resulta de la compilación de código fuente y este es contenido en librerías compartidas o archivos ejecutables.

identificadores, tipos de datos, clases, interfaces, y, en definitiva, a todas las entidades declaradas en el código fuente, y es capaz de inspeccionar sus nombres, atributos, especificadores y calificadores.

Reflexión en tiempo de ejecución.

Este mecanismo ocurre durante la ejecución de un programa y funciona de manera que cada objeto en memoria está asociado a una metadata donde está almacenada la información de sus variables y métodos miembros (Márton y Porkolab, 2010). Este proceso requiere que los metadatos formen parte del archivo ejecutable del programa, lo que incrementa su tamaño pero otorga la capacidad de examinar y manipular clases, interfaces, campos y métodos en tiempo real.

La reflexión en tiempo de ejecución tiene algunas desventajas en comparación con la reflexión en tiempo de compilación, el tamaño del ejecutable será mayor incluso si no todos los objetos en tiempo de ejecución son reflejados, y el programa se ejecutará más lentamente (Márton y Porkolab, 2010). Esto es debido a que la reflexión en tiempo de ejecución implica tipos de datos que se resuelven dinámicamente y ciertas optimizaciones que suelen realizar los compiladores o intérpretes no son aplicadas. En consecuencia, las operaciones reflexivas tienen un rendimiento más lento que sus contrapartes no reflexivas y deberían evitarse en secciones de código que se llaman frecuentemente en aplicaciones sensibles al rendimiento.

Lenguaje de programación C++.

C++ es un lenguaje de programación multiparadigma y de propósito general creado por Bjarne Stroustrup a principios de la década de los ochenta en los Laboratorios Bell. Inicialmente conocido como “C con Clases”, fue diseñado para proporcionar las facilidades del lenguaje Simula⁷ para la organización de programas junto con la eficiencia y flexibilidad del lenguaje C⁸ para la programación de sistemas (Stroustrup, 2007). Está estandarizado por la Organización Internacional para la

⁷ Desarrollado en los sesenta por Ole Johan Dahl y Kristen Nygaard, Simula fue el primer lenguaje de programación en incluir clases, objetos, subclases, herencia y procedimientos virtuales.

⁸ C es un lenguaje de programación imperativo, procedural, y de propósito general creado a principios de los setenta por Dennis Ritchie en los Laboratorios Bell.

Estandarización (ISO) a través del comité técnico ISO/IEC JTC1/SC22/WG21, también conocido como Working Group 21 (WG21) o el C++ Standards Committee. Este grupo tiene la responsabilidad de desarrollar y preservar las características esenciales del lenguaje y de la librería estándar, ambas formalmente identificadas como ISO/IEC 14882.

Tipos de datos.

Según Stroustrup (2013), cada entidad en un programa de C++ está asociada con un tipo de dato. Los tipos de datos determinan qué operaciones se pueden aplicar a las entidades y cómo dichas operaciones han de ser interpretadas. En el lenguaje se clasifican los tipos de datos en dos divisiones: tipos fundamentales y tipos compuestos (ISO/IEC 14882, 2020, p. 68). A grandes rasgos, los tipos fundamentales son los bloques básicos con los que se construyen tipos compuestos y son la unidad mínima por la cual se acceden y procesan los datos. Es posible también dar la interpretación de que los tipos compuestos son todos aquellos tipos que no son tipos fundamentales.

Fundamentales. Conocidos como tipos *built-in*, Stroustrup (2013) señala que C++ tiene un conjunto de tipos fundamentales que corresponden a las unidades básicas de almacenamiento más comunes de una computadora. Dentro los tipos fundamentales se incluyen los booleanos, números enteros, números de punto flotante (decimales), caracteres y el tipo especial `void`. Los números enteros se dividen en dos categorías: con signo y sin signo, permitiendo representar tanto números positivos como negativos, y únicamente positivos, respectivamente. El tamaño ocupado en memoria por los tipos fundamentales no está definido por el estándar ISO/IEC 14882 y son dependientes de la implementación del compilador y de la arquitectura objetivo; por ejemplo, en plataformas de 16 bits es común encontrar que el tipo fundamental `int` tenga un tamaño de 2 bytes, mientras que en arquitecturas de 32 y 64 bits `int` suele tener un tamaño de 4 bytes.

Compuestos. Un tipo compuesto es un tipo que se define en términos de otro tipo (Lippman, Lajoi y Moo, 2013). El estándar ISO/IEC 14882 (2020) especifica que los siguientes son tipos compuestos: los arreglos, las funciones, los punteros, las referencias, las clases, las estructuras, las uniones, y las enumeraciones (pp. 72-73). Los tipos compuestos permiten la construcción de estructuras de datos complejas, la definición de funciones con parámetros y tipos de retorno especializados, y la creación de capas de abstracción. El tamaño ocupado en memoria de los tipos compuestos depende principalmente del tamaño de los tipos de datos contenidos y de la alineación de memoria, ambos factores establecidos por el compilador según la arquitectura objetivo.

Clases.

Según Stroustrup (2013), las clases en C++ son una herramienta para crear nuevos tipos que pueden utilizarse tan cómodamente como los tipos built-in. Lippman, Lajoi y Moo (2013) agregan que una clase define un tipo de dato junto con las operaciones relacionadas a dicho tipo. Stroustrup (2013) explica que la idea fundamental de definir nuevos tipos de datos a través de clases tiene como finalidad separar los detalles de implementación (como por ejemplo, la estructura de los datos utilizada para almacenar un objeto en memoria) de las propiedades esenciales para el uso correcto de estos tipos (tal como su interfaz pública, que muestra la lista de funciones disponibles para una clase). A través de las clases se puede expresar niveles o capas de abstracción que las estructuras de datos convencionales no poseen, y, además, formar relaciones jerárquicas y paramétricas con otras clases a través de la herencia, el polimorfismo y la programación genérica; estas abstracciones permiten programar código modular, genérico y reutilizable. En sí mismo, a nivel de representación de objeto en memoria, una clase contiene una estructura de datos. No obstante, existe una versatilidad en el lenguaje, tanto en sintaxis como en semántica, que al declarar y definir clases permite que

ideas abstractas y conceptos concretos del mundo real sean expresados en un campo discreto y lógico.

. El estándar ISO/IEC 14882 utiliza el término clase para referirse a tres entidades del lenguaje por igual: clases, estructuras y uniones (ISO/IEC 14882, 2020, p. 247). Dependiendo de su definición, una clase tendrá ciertas propiedades; estas propiedades permiten reconocer cómo está conformada dicha clase con respecto a sus variables y funciones miembros, sus clases bases y sus especificadores de acceso. En C++20 podemos clasificar las clases según tres propiedades esenciales: clases trivialmente copiables, clases de diseño estándar y clases agregadas.

Clases trivialmente copiables. Una clase es trivialmente copiable cuando sus operaciones de copia (constructores y operadores de asignación de copia o semánticas de movimiento) están establecidas por defecto ya sea de manera implícita o bien por el usuario a través de la palabra clave `default`. Las clases con esta propiedad son copiadas de forma eficiente, segura y automática, ya que, sin necesidad de código adicional, se copia la representación de objeto desde una región de memoria a otra byte por byte.

A trivially copyable class is a class: (1.1) — that has at least one eligible copy constructor, move constructor, copy assignment operator, or move assignment operator (11.4.3, 11.4.4.2, 11.4.5), (1.2) — where each eligible copy constructor, move constructor, copy assignment operator, and move assignment operator is trivial, and (1.3) — that has a trivial, non-deleted destructor (11.4.6). (ISO/IEC 14882, 2020, p. 248)

Clases de diseño estándar. Una clase es de diseño estándar cuando la disposición de sus miembros en la memoria está bien definida y es predecible según la especificación del lenguaje. En C++, esto significa que la clase no utiliza características avanzadas del lenguaje que permitirían al compilador elegir libremente la representación en memoria, como es el caso de clases base virtuales, funciones virtuales

o miembros con diferentes controles de acceso. Las clases de diseño estándar son por naturaleza portables entre diferentes lenguajes y plataformas y pueden verse como una estructura de datos opaca.

A class S is a standard-layout class if it: (3.1) — has no non-static data members of type non-standard-layout class (or array of such types) or reference, (3.2) — has no virtual functions (11.7.2) and no virtual base classes (11.7.1), (3.3) — has the same access control (11.9) for all non-static data members, (3.4) — has no non-standard-layout base classes, (3.5) — has at most one base class subobject of any given type, (3.6) — has all non-static data members and bit-fields in the class and its base classes first declared in the same class [...]. (ISO/IEC 14882, 2020, p. 248)

Al igual que las clases trivialmente copiables, las instancias de objetos de clases de diseño estándar ocupan una región de memoria contigua. ISO/IEC 14882 (2020) “An object of trivially copyable or standard-layout type (6.8) shall occupy contiguous bytes of storage.” (p. 56). Esto permite copiar los bytes subyacentes de un objeto de manera segura, por ejemplo, a través de `std::memcpy` o bien haciendo una reinterpretación hacia un arreglo de bytes.

Tipos de datos agregados. Un tipo de dato agregado es un arreglo o una clase que no posee constructores definidos por el usuario, todos sus miembros son públicos y no tiene funciones virtuales. Al satisfacer estos criterios, los agregados representan una estructura de datos predecible en memoria, pues no se les adiciona un puntero hacia una tabla virtual ni poseen miembros inaccesibles.

An aggregate is an array or a class (Clause 11) with no user-declared or inherited constructors (11.4.4), no private or protected direct non-static data members (11.9), no virtual functions (11.7.2), and no virtual, private, or protected base classes (11.7.1). (ISO/IEC 14882, 2020, pp. 192-193)

Objetos.

El estándar ISO/IEC 14882 (2020) define un objeto como una entidad que ocupa una región de almacenamiento y cuya vida y características están definidas por su tipo de dato. Es decir, es un conjunto de datos en memoria que el programa puede manipular. Los objetos se pueden crear mediante definiciones, expresiones `new`, operaciones implícitas, cambios en miembros activos de una unión, o cuando se crean objetos temporales. Los objetos pueden tener nombres, una duración de almacenamiento y tipos que determinan cómo se interpretan sus valores. Pueden contener otros objetos (subobjetos) y algunos pueden cambiar su tipo durante la ejecución del programa (polimorfismo). Todos los objetos poseen una representación de objeto y una representación de valor, en este aspecto el estándar define:

The object representation of an object of type T is the sequence of N unsigned char objects taken up by the object of type T, where N equals sizeof(T). The value representation of an object of type T is the set of bits that participate in representing a value of type T. Bits in the object representation that are not part of the value representation are padding bits. For trivially copyable types, the value representation is a set of bits in the object representation that determines a value, which is one discrete element of an implementation-defined set of values. (ISO/IEC 14882, 2020, pp. 68-69)

Representación de objeto. Se refiere a la secuencia de bytes que ocupa un objeto (el estándar utiliza el tipo `unsigned char` para describir un byte). La representación de objeto hace referencia a cómo se almacena físicamente un objeto en memoria, incluyendo todos los bits que lo componen, sean útiles para el valor o no. Un ejemplo de esto son los tipos booleanos, cuya representación en memoria es un byte completo pero su representación de valor es un único bit.

Representación de valor. La representación de valor de un objeto de tipo “T” es el conjunto de bits que participan en representar un valor del tipo “T”. En otras palabras, se refiere a aquellos bits específicos dentro de la representación de objeto que realmente se utilizan para determinar el valor del objeto. Los bits que no forman parte de la representación de valor se denominan bits de relleno.

Plantillas y Conceptos.

Las Plantillas son una característica del lenguaje que permite la programación genérica al conceder la capacidad de parametrizar clases y funciones con tipos genéricos (Stroustrup, 2013). Dado que C++ es un lenguaje de tipado estático, cada entidad está asociada con un tipo determinado, y este tipo puede ser especificado directamente por el programador o inferido por el compilador. Numerosas estructuras de datos y algoritmos mantienen una forma consistente independientemente del tipo de datos sobre el que operan, por lo cual es beneficioso trabajar con tipos genéricos para facilitar la reutilización del código y mejorar su mantenibilidad. La utilización de Plantillas ayuda a crear librerías más flexibles y da pie a mejores abstracciones en el lenguaje.

Los Conceptos forman parte del lenguaje desde la versión de C++20 (Sutton, 2017) y son una descripción de las operaciones admitidas para un tipo de dato. Es posible pensar en los Conceptos como la definición de un conjunto de restricciones aplicadas en los parámetros de Plantillas evaluados en tiempo de compilación de clases y funciones genéricas. Los Conceptos se utilizan principalmente para introducir la verificación de tipos en la programación genérica y simplificar los diagnósticos del compilador para instancias de Plantillas fallidas. C++20 introduce los Conceptos y, con ellos, una librería específica de Conceptos estándar (Carter y Niebler, 2018). Estos se organizan en cuatro categorías principales: esenciales del lenguaje, que abarcan tipos fundamentales, herencia y constructores; de comparación, que determinan si un tipo se puede convertir a booleano o puede ser comparado; de objetos, que evalúan si un tipo puede ser copiado, movido o intercambiado; y de llamadas, que verifican si un tipo es invocable, como en el caso de las funciones. Con

los recursos que ofrece la librería estándar, es posible diseñar nuevos Conceptos que faciliten el manejo eficiente y avanzado de los diferentes tipos de datos, tanto fundamentales como compuestos, y diferenciar entre tipos de datos agregados y no-agregados.

Librería estándar.

El estándar ISO/IEC 14882 (2020) establece una colección de clases y funciones contenidas en una librería de código conocida como la librería estándar. Entre sus componentes principales se encuentran los contenedores, los algoritmos de búsqueda, ordenamiento y transformaciones de datos, y los iteradores, definidos en la librería estándar de plantillas; las clases de flujos de datos y las funciones de entrada/salida, definidos en las cabeceras de entrada/salida; las facilidades de hilos para la programación concurrente y paralela, definidas en las cabeceras multihilo; y varias funciones y clases utilitarias, definidas en las cabeceras de utilidades.

Contenedores. Los contenedores son clases genéricas que almacenan una colección de objetos (elementos). Los contenedores se encargan de gestionar el alojamiento y desalojamiento en memoria de sus elementos y de proveer accesos a éstos, ya sea con referencias directas o con iteradores. Entre los contenedores definidos por la librería estándar se encuentran arreglos estáticos y arreglos dinámicos (vectores), listas enlazadas simples y listas doblemente enlazadas, pilas, colas y colas de prioridad, conjuntos, y mapas hash.

Tuplas. Según Stroustrup (2013), una tupla es una secuencia de N elementos con tipos arbitrarios (donde N es un número natural). El estándar ISO/IEC 14882 (2020) define una tupla como una colección de tamaño fijo de valores heterogéneos. En la cabecera `<tuple>` se especifican las clases de tuplas `std::tuple` y `std::pair` Jabot (2022) describe que algunos objetos, bajo ciertos requerimientos, pueden ser también tratados como tuplas; por ejemplo, un arreglo de tamaño fijo es una tupla donde todos los tipos de sus elementos son idénticos.

Objetos opcionales. Un objeto opcional maneja un valor contenido de manera opcional; un valor que puede o no estar presente. El estándar ISO/IEC 14882 (2020) especifica que un objeto opcional es un objeto que contiene el almacenamiento de otro objeto y gestiona la vida útil de este objeto contenido, si es que existe. Comúnmente los objetosopcionales son instanciados desde clases genéricas, y estas clases suelen tener un diseño de estructura de datos de tal manera que siempre tienen espacio para un booleano o para un booleano y el tipo de dato genérico contenido. En la librería estándar, desde C++17, se incorporó la cabecera `<optional>` junto con la clase `std::optional`, proveyendo una interfaz explícita y optimizada para el manejo de estos tipos de datos.

Variantes. Naumann (2016) define que una variante es una unión segura de tipos (también llamada unión discriminada o unión con etiqueta). Una clase variante es una clase genérica que contiene una lista paramétrica de tipos llamados “alternativas”; una instancia de esta clase, en cualquier momento dado, contiene un único valor de una de sus alternativas. A diferencia de una unión, las variantes llevan un registro mediante un índice de cuál alternativa está activa, lo que permite operar con el valor contenido de estos objetos de manera segura. La librería estándar especifica la clase genérica `std::variant` en la cabecera `<variant>` y el estándar define:

A variant object holds and manages the lifetime of a value. If the variant holds a value, that value’s type has to be one of the template argument types given to variant. These template arguments are called alternatives. (ISO/IEC 14882, 2020, p. 593).

Punteros inteligentes. Para Stroustrup (2013), los punteros inteligentes son objetos que gestionan automáticamente la vida útil de un objeto al que apuntan; estos se utilizan para prevenir errores como fugas de memoria y representan una práctica segura en comparación con los punteros crudos. El estándar ISO/IEC 14882 (2020) define los punteros

inteligentes como objetos que poseen y manejan otro objeto a través de un puntero interno, y que se encargan de desalojar el objeto contenido una vez que la propia vida útil del puntero inteligente llega a su fin.

En la librería estándar, en la cabecera `<memory>` se especifican tres punteros inteligentes: `std::unique_ptr`, `std::shared_ptr` y `std::weak_ptr`. El primero, un puntero único, es utilizado cuando la propiedad del objeto subyacente debe ser exclusiva. El segundo, un puntero compartido, define una propiedad del objeto subyacente compartida. Y, el tercero, un puntero débil, hace referencia a un objeto manejado por otro puntero compartido, pero este no toma ninguna propiedad de dicho objeto. Los punteros únicos y compartidos también suelen ser llamados “punteros propietarios”.

Evaluación de Expresiones Constantes.

El cómputo de expresiones en tiempo de compilación existe con el objetivo de mejorar y optimizar el código máquina, aportando herramientas adicionales al programador para instruir al compilador y habilitar optimizaciones de código (Fertig, 2021). Normalmente el compilador tiene como objetivo compilar código fuente para construir código objeto en forma de ejecutables y librerías compartidas, y así, las instrucciones definidas en un programa pueden ser ejecutadas en tiempo de ejecución. Sin embargo, en algunos lenguajes de programación y bajo ciertas condiciones, es posible que un compilador esté en capacidad de llamar funciones en tiempo de compilación. Estas herramientas permiten demostrar la intención del programador para crear código que es más predecible, más fácil de leer y más fácil de mantener. La importancia de la computación en tiempo de compilación radica en realidad en el producto final: el código objeto, destinado a ser ejecutado en tiempo de ejecución.

A partir de C++11 se incluyó soporte inicial para la evaluación de expresiones constantes y literales definidos por el usuario (Dos Reis, Stroustrup y Maurer, 2007), y en siguientes versiones del estándar se fueron incorporando mejoras, extensiones y características nuevas para la

computación en tiempo de compilación. Solo es posible evaluar y ejecutar en tiempo de compilación funciones puras (Stroustrup, 2013). Una función pura cumple con dos propiedades esenciales: la primera es determinismo, ya que el resultado de una función pura es un literal constante y es dependiente exclusivamente del conjunto de sus valores de entrada; la segunda es la inexistencia de efectos secundarios, lo que significa que en su ejecución no se altera ningún estado global, no se mutan referencias, ni tampoco se invocan funcionalidades de entrada/salida.

Dos Reis, Stroustrup y Maurer (2007) describen que la intención de las expresiones constantes es la de mejorar la seguridad de tipos y la portabilidad de código que requiere evaluación estática, mejorar el soporte para la programación de sistemas, la creación de librerías de código y la programación genérica, y eliminar inconvenientes en el lenguaje en áreas donde anteriormente se había utilizado metaprogramación de plantillas.

En C++20 existen los especificadores `constexpr` y `consteval` (ISO/IEC 14882, 2020, pp. 162-164). Ambos están diseñados para indicarle al compilador que una expresión puede o debe ser evaluada en tiempo de compilación; `constexpr` ofrece la posibilidad de evaluación en tiempo de compilación pero permite la ejecución en tiempo de ejecución si es necesario, mientras que una función `consteval` es estrictamente evaluada en tiempo de compilación. También existe soporte para la estructura de control condicional en tiempo de compilación `if constexpr` (Maurer, 2016), que habilita la evaluación de condiciones estáticas y permite la selección de diferentes ramas de ejecución según un contexto constante.

Alineación de estructuras de datos.

Según Tanenbaum y Austin (2013), una palabra (del inglés, *word*) es la unidad de datos de tamaño estándar con la cual una arquitectura de computadora está diseñada para manejar y procesar datos de manera eficiente; por ejemplo, en arquitecturas de 32 bits una palabra contiene 4 bytes. El subsistema de memoria en un procesador moderno suele estar restringido a acceder a la memoria del computador con la granularidad y alineación de su tamaño de palabra.

An object doesn't just need enough storage to hold its representation. In addition, on some machine architectures, the bytes used to hold it must have proper alignment for the hardware to access it efficiently (or in extreme cases to access it at all). [...] Of course, this is all very implementation specific, and for most programmers completely implicit. [...] Where alignment most often becomes visible is in object layouts: sometimes structs contain “holes” to improve alignment. (Stroustrup, 2013, p. 151)

Los "huecos" de los que hace referencia Stroustrup son bytes de relleno (del inglés, *padding bytes*) que se insertan en las estructuras de datos para alinear los campos de datos en la memoria. En C++, todos los tipos de datos fundamentales tienen un tamaño y alineación específicos que dependen de la arquitectura de la máquina objetivo. Cuando se definen tipos compuestos (estructuras, clases y uniones) el compilador inserta bytes de relleno para asegurar que los campos de datos estén alineados correctamente. Por ejemplo, en una arquitectura de 32 bits, un entero de 4 bytes debe estar alineado en una dirección múltiplo de 4; si se define una estructura con un entero de 4 bytes seguido de un carácter de 1 byte, el compilador insertará 3 bytes de relleno entre ambos campos para alinear el carácter en una dirección múltiplo de 4; en este caso, el tamaño de la estructura será de 8 bytes en lugar de 5 bytes.

Aunque en un principio pueda parecer que los bytes de relleno son un desperdicio de espacio, el procesador accede a la memoria de manera más eficiente cuando los datos están alineados, provocando un enorme impacto positivo en el rendimiento de un programa, y por lo cual siempre se prefiere pagar el costo de estos "huecos". En situaciones donde la optimización del uso de memoria es crucial, como en los sistemas embebidos, se pueden emplear herramientas específicas del lenguaje y de los compiladores para controlar la alineación de datos.

Alineación explícita.

En algunos lenguajes de programación es posible alinear las estructuras de datos de manera explícita, esto suele ser con la finalidad de cumplir con requerimientos externos inherentes a la máquina en la cual se trabaja. En C++ existen dos mecanismos que forman parte del estándar ISO/IEC 14882 y que son utilizados para obtener y establecer explícitamente la alineación de

memoria de los tipos de datos: `alignof` y `alignas`. El operador `alignof` retorna la alineación en bytes de un tipo o objeto en memoria, y, por otra parte, el especificador `alignas` solicita al compilador el cambio de la alineación de un tipo según un argumento expresado en tamaño de bytes.

Object types have alignment requirements (6.8.1, 6.8.2) which place restrictions on the addresses at which an object of that type may be allocated. An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type; stricter alignment can be requested using the alignment specifier (9.12.2). (ISO/IEC 14882, 2020, p. 64)

Es importante notar que el estándar ISO/IEC 14882 especifica que una alineación explícita puede ser solicitada, más no garantiza que se cumpla el cambio de alineación. Sea o no efectivo este cambio, el operador `alignof` devolverá el valor real de alineación de los objetos en memoria.

Estructuras de datos empaquetadas.

Es posible empaquetar una estructura de datos estableciendo la alineación de todos sus miembros agregados en un límite de bytes especificado. Si el número de dicho límite es menor a la alineación natural de algún miembro se colocarán todos los miembros uno tras otro directamente en la memoria, evitando el relleno de bytes y reduciendo el tamaño ocupado por la estructura.

El empaquetado de estructuras no es una práctica parte del estándar ISO/IEC 14882 de C++, y, por ende, es dependiente de la implementación del compilador utilizado. Sin embargo, los compiladores MSVC⁹, GCC¹⁰ y Clang¹¹ han adoptado la directiva de preprocesador `#pragma pack` que instruye al compilador para empaquetar los miembros de una clase con una alineación particular. Los compiladores que no soportan esta directiva suelen

⁹ MSVC: <https://learn.microsoft.com/en-us/cpp/preprocessor/pack?view=msvc-170>

¹⁰ GCC: <https://gcc.gnu.org/onlinedocs/gcc/Structure-Layout-Pragmas.html>

¹¹ Clang: <https://releases.llvm.org/3.4/tools/clang/docs/UsersManual.html#microsoft-extensions>

tener implementaciones del atributo `__attribute__((packed))`, que cumple una funcionalidad idéntica. La técnica de empaquetado es mantenida a razón de limitaciones en el especificador de alineación `alignas`; la más notable, es que el argumento pasado a “alignas” debe ser una potencia de dos y no puede ser menor que `alignof(T)` para cualquier tipo `T`, lo que significa que es posible aumentar la alineación de un tipo de dato pero no disminuirla.

Ordenamiento de bytes.

Según Tanenbaum y Austin (2013), los bytes contenidos en una palabra pueden numerarse de izquierda a derecha o de derecha a izquierda. La forma en la que se ordenan los bytes se conoce como *endianness*, el formato *big-endian* almacena el byte más significativo en la dirección de memoria más baja, y, por el contrario, el formato *little-endian* invierte este orden colocando el byte menos significativo en la dirección de memoria más baja. Chiu (2004) indica que diferentes arquitecturas de máquinas pueden utilizar distintos ordenamientos para los bytes que conforman una unidad de varios bytes, como, por ejemplo, un número de punto flotante de 64 bits. Por lo tanto, para que una máquina lea un número cuyo orden de bytes difiere del de la máquina que originalmente lo escribió será mandatorio reorganizar dichos bytes.

Fertig (2021) señala que el tráfico de red se rige por el "orden de bytes de red", que es big-endian. No obstante, la mayoría de las computadoras actuales, como las que usan procesadores Intel y ARM, son little-endian por defecto. Por lo tanto, para transferir datos exitosamente entre un host little-endian y uno big-endian, el primero debe realizar una conversión invirtiendo el orden de sus bytes.

Es posible detectar si una máquina utiliza big-endian o little-endian en tiempo de ejecución a través de la conversión de punteros de distintos tamaños, y, por otra parte, también es posible detectarlo en tiempo de compilación, pero para esto la comunicación con el compilador es requerida. A partir de C++20 existe soporte en la librería estándar del lenguaje para obtener el orden de bytes de la arquitectura objetivo a compilar utilizando `std::endian` (Hinnant, 2017; Fertig, 2021).

Formatos de archivos.

Biswal y Almallah (2018) consideran tres categorías para etiquetar los formatos de archivos: con esquema y sin esquema, según la codificación empleada, y si requieren del uso de una librería. Los formatos con esquema predefinido utilizan un Lenguaje de Descripción de Interfaz (del inglés, *Interface Description Language, IDL*) para dictar cómo deben interpretarse los datos, asegurando que la información siga una estructura específica. En cambio, los formatos sin esquema ofrecen mayor flexibilidad en la organización de los datos, aunque esto puede incrementar su volumen al necesitar incluir detalles sobre la estructura de los mismos junto con sus valores. Desde la perspectiva de la codificación, existen formatos basados en texto, que convierten los datos a caracteres legibles por personas usando codificaciones como ASCII o UTF-8, y formatos binarios, que son interpretados directamente por las máquinas. El procesamiento de los formatos binarios suele requerir herramientas especializadas contenidas en librerías de código diseñadas para el manejo de secuencias de bytes, mientras que los formatos de texto se pueden gestionar fácilmente con las funciones estándar de edición de texto y caracteres presentes en la mayoría de los lenguajes de programación.

Según Chiu (2004), los serializadores binarios utilizan formatos no textuales y representan los números en una forma más cercana a la requerida por el hardware de cómputo. Los formatos binarios representan los datos como una secuencia de bytes, y estos, frecuentemente, están destinados a ser interpretados como valores diferentes a caracteres de texto. Al poseer esta naturaleza, los archivos binarios no están hechos para la lectura y edición directa por humanos sino por computadoras. Consecuentemente, por el tipo de codificación que utilizan, son más eficientes en términos de espacio y, sobre todo, en tiempo de procesamiento.

Los formatos basados en texto, como XML y JSON, utilizan codificaciones de caracteres (comúnmente ASCII, UTF-8 o Latin1) para crear representaciones de datos legibles para humanos. Estos formatos permiten la edición de los datos en cualquier editor de texto, aunque son menos eficientes en espacio y tiempo de procesamiento comparados con las alternativas binarias (Casey, 2022). El costo de los formatos textuales a menudo no se debe tanto a la verbosidad inherente del texto sino a la necesidad de conversión de una representación decimal a una binaria (Chiu, 2004).

Formatos de números enteros y reales.

Enteros de ancho fijo.

Los números enteros de ancho fijo tienen un tamaño de bits específico, como por ejemplo 8, 16, 32 y 64 bits. El número de bits define el rango de valores que pueden representar y estos pueden ser con signo o sin signo. Los enteros positivos se representan directamente en binario, mientras que los enteros negativos utilizan el complemento a dos para su representación (Morris, 2003).

Las instrucciones de procesador utilizan enteros de ancho fijo que son alojados directamente en la memoria caché y en los registros, siendo la forma más eficiente de realizar las operaciones aritméticas y lógicas.

Complemento a dos. Morris (2003) define el complemento a dos como un método para representar números enteros con signo en sistemas binarios. Se utiliza comúnmente en informática y electrónica para facilitar operaciones aritméticas. La técnica permite la representación de números enteros negativos mediante la inversión de todos los bits de un número positivo (complemento a uno) y la adición de 1 al resultado.

Codificación Zig-Zag. Fitzgerald (2015) explica que la codificación Zig-Zag es una técnica utilizada para intercalar números positivos y negativos de manera que los números negativos con un valor pequeño se almacenen en una cantidad menor de bytes, a diferencia de la representación de complemento a dos donde un número negativo se representa marcando el bit más significativo, y por ende, utilizando todo el ancho de bits. Esta codificación se utiliza para almacenar enteros fijos con signo, y consiste simplemente en convertir los números negativos a una representación de números positivos de la siguiente forma:

$$z_p = 2 \cdot p$$

$$z_n = 2 \cdot |n| - 1$$

donde z_p es la codificación de un entero positivo p y z_n la codificación de un entero negativo n .

A nivel de instrucciones de máquina, esta codificación sólo representa dos desplazamientos de bits y un operador xor, por lo cual se considera muy eficiente. Específicamente:

$$z = (n \ll 1) \oplus (n \gg (b - 1))$$

donde z es el número codificado y b es el ancho de bits (por ejemplo, 32 bits).

Enteros de longitud variable.

Los enteros de longitud variable representan una manera de codificar números enteros en un formato que puede ocupar menor espacio del que normalmente necesitan (Creager, 2021). Por defecto, las computadoras utilizan enteros de longitud fija por razones de eficiencia del hardware, sin embargo, al transmitir o almacenar números enteros, es posible codificar estos números en una secuencia de bytes variable para ahorrar espacio de almacenamiento o ancho de banda (Lemire, Kurz y Rupp, 2018). La utilización de este tipo de representación se sustenta bajo la premisa de que la mayoría de los números enteros no suelen estar distribuidos uniformemente y que es más común encontrar números con valores menores a 2^{64} y 2^{32} (Creager, 2021).

Un entero de longitud variable utiliza más bytes para números grandes y menos bytes para los números pequeños, y, de esta forma, su longitud varía dependiendo del tamaño real del valor contenido. Esto permite ahorrar espacio en los casos en que efectivamente los valores numéricos sean pequeños. Por ejemplo, un entero de 64 bits que contiene un número menor a 256 estaría desperdiando los 56 bits superiores de una representación de ancho fijo. Por supuesto, codificar y decodificar números de longitud variable constituye la utilización de recursos de cómputo y tiempo de procesamiento, pero cuando se trabaja con estructuras que contienen mayormente datos enteros suele ser más rentable el espacio total ahorrado por la compresión de enteros, especialmente cuando estos datos serán almacenados o enviados a través de una banda ancha.

Según Fitzgerald (2015), los enteros de longitud variable son un compromiso: o bien tienen un prefijo de longitud explícita, o bien incluyen un centinela que señala el final del número. Existen múltiples codificaciones de enteros de longitud variable, Google utiliza una codificación propia en su librería Protobuf, conocida formalmente como VLQ¹² (del inglés, *Variable Length Quantity*). Esta codificación es una de las más famosas y funciona de forma que se utiliza el bit más significativo de cada byte como un marcador de continuación. Los 7 bits inferiores de cada byte codifican algunos de los bits del valor entero que se está codificando. Un valor de “1” para el bit de continuación significa que existen más bytes por seguir; un valor de “0” significa este es el último byte para un valor.

Números reales (aritmética de coma flotante).

Según Boldo, Jeannerod, Melquiond, y Muller (2023), los números de coma flotante son una representación numérica que se utiliza para aproximar números reales en computadoras. En esencia, las operaciones de coma flotante se realizan como si los resultados se calcularan primero con una precisión infinita y luego se redondeará al formato objetivo. Esto asegura que la aritmética de coma flotante cumple con un modelo estándar para analizar la precisión de los algoritmos que la utilizan.

El estándar IEEE-754, establecido en 1985, ha sido fundamental para estandarizar la aritmética de coma flotante, definiendo reglas estrictas pero útiles para su manejo. Con anterioridad a este estándar existían múltiples representaciones entre los diferentes fabricantes de procesadores, lo que significaba que para programar algoritmos portables se tenía tener en cuenta múltiples formatos y sus distintas conversiones entre ellos. Según Boldo et al. (2023), a día de hoy casi todas las unidades de coma flotante (del inglés, *FPU*, *Float-Point Unit*) en el mercado adoptan este estándar que especifica tres formatos básicos: “binary32”, “binary64” y “binary128” designados con la intención de ser utilizados para la aritmética de punto flotante de 32, 64 y 128 bits respectivamente, y un formato “binary16” expuesto solo para almacenamiento de números reales en 16 bits.

¹² Codificación VLQ utilizada en Protobuf: <https://protobuf.dev/programming-guides/encoding/#varints>

Evaluación de sistemas informáticos.

Según Lilja (2004), la evaluación de sistemas informáticos se concibe como un proceso integral que combina la medición, interpretación y comunicación de la “velocidad” o “tamaño” de un sistema informático. Este último término, “tamaño”, también se conoce en ocasiones como la “capacidad” del sistema. La obra de Lilja se centra en el rendimiento desde la perspectiva de eficiencia de desempeño, pero otros autores como Molero, Juiz y Rodeño (2004) añaden que la evaluación de sistemas está directamente relacionada con la valoración de ciertas características de calidad, como lo son la seguridad y la disponibilidad.

La calidad de un producto de software se define por su capacidad para satisfacer los requisitos de los usuarios y, de este modo, aportar valor. El estándar ISO/IEC 25010:2023 se especializa precisamente en este aspecto, estableciendo un modelo de calidad que categoriza las propiedades de un software en un conjunto de características y subcaracterísticas evaluables. Las siguientes son algunas de estas características:

Eficiencia de desempeño.

La eficiencia de desempeño según el ISO/IEC 25010 (2023) evalúa el rendimiento del software según su comportamiento temporal, utilización de recursos y capacidad. Si un producto de software cumple con esta característica de calidad, significa que realiza sus funciones con una velocidad de respuesta óptima y con un uso eficiente de recursos bajo determinadas condiciones y cargas de trabajo.

Diversos autores como Molero et al. (2004) hacen especial énfasis en la eficiencia de desempeño, a veces también llamada simplemente eficiencia o rendimiento. Lilja (2004) señala que al momento de evaluar un sistema existen tres técnicas fundamentales: medición, simulación y modelado analítico. Las mediciones reales suelen ofrecer los mejores y más creíbles resultados al no requerir simplificaciones, y dichas mediciones son utilizadas para obtener métricas de rendimiento. Existe una correspondencia entre estas métricas y las subcaracterísticas de calidad del ISO/IEC 25010. Más específicamente, las métricas de velocidad y tiempos de respuesta evalúan el comportamiento

temporal, mientras que otras métricas como el porcentaje de uso del CPU o el uso máximo de memoria RAM valoran la utilización de recursos.

Compatibilidad.

Dentro del estándar ISO/IEC 25010 (2023), la compatibilidad se refiere a la habilidad de un producto para funcionar en un entorno compartido con otros sistemas, pero su aspecto más crucial y dinámico es la interoperabilidad. Mientras que la coexistencia implica que un software puede compartir recursos con otro sin causar conflictos, la interoperabilidad va un paso más allá, definiendo la capacidad real de dos o más sistemas no solo para intercambiar datos, sino para comprender y utilizar activamente esa información intercambiada.

Mantenibilidad.

La mantenibilidad se define como la capacidad de un producto de software para ser modificado de manera efectiva y eficiente, ya sea para corregir errores, mejorar su rendimiento o adaptarlo a nuevas necesidades (ISO/IEC 25010, 2023). Esta cualidad se sustenta en subcaracterísticas clave como la modularidad, que asegura que un cambio en un componente no afecte negativamente a otros, y la reusabilidad, que permite que partes del software o activos completos puedan ser utilizados para construir nuevos sistemas o funcionalidades, optimizando así tiempo y recursos en el desarrollo y la actualización.

Flexibilidad.

La flexibilidad es la aptitud de un software para evolucionar frente a nuevas exigencias (ISO/IEC 25010, 2023). Esto se manifiesta en su adaptabilidad (o portabilidad) para funcionar correctamente sobre distinto hardware o software, y en su escalabilidad para manejar de forma eficiente un aumento o disminución en el volumen de trabajo, garantizando que el producto no se vuelva obsoleto ante cambios en su entorno o demanda.

Antecedentes

Librerías de serialización binaria.

Una librería de código es una colección de recursos utilizados en el desarrollo de software para producir un programa de computadora. A su vez, una librería de serialización binaria es una librería de código que contiene un repertorio de funcionalidades para la serialización y deserialización de datos en un formato binario.

Chiu (2004) señala que “a binary serializer implementation can define two orthogonal aspects: format and programming interface (API)” (sección 4, párr. 1). En este sentido, las librerías de serialización binaria necesitan un formato de archivo binario de convención para transmitir y almacenar información entre sistemas, y a su vez, deben proporcionar una interfaz de programación capaz de consumir este formato para serializar y deserializar los datos.

Protocol Buffers.

Conocido como Protobuf¹³, es una librería de serialización binaria de código abierto que es desarrollado y mantenido por Google. Diseñado para ser eficiente, extensible y neutro respecto al lenguaje de programación y a la plataforma objetivo, funciona mediante la definición de esquemas de datos contenidos en archivos “.proto” que luego son compilados para generar código fuente con definiciones de clases y métodos de serialización y deserialización especializados en un lenguaje de programación determinado.

Protocol Buffers se compone de varias partes: un lenguaje de definición de datos con el cual se describen mensajes (estructuras de datos) en archivos “.proto”, un compilador conocido encargado de generar código fuente, librerías en múltiples lenguajes para manejar los procesos de serialización y deserialización, y finalmente, un formato de datos binario con un estilo clave-valor (conocido como *wire format*) utilizado para almacenamiento y transmisión. El esquema contenido en los archivos “.proto” está diseñado para definir dos tipos de objetos: los mensajes y las enumeraciones. Un mensaje es un tipo de objeto agregado donde se definen

¹³ Página oficial de Protocol Buffers: <https://protobuf.dev/overview/>

todos los campos miembros, estos pueden ser otros mensajes o enumeraciones. Las enumeraciones son listas de valores. El compilador se invoca sobre archivos “.proto” para generar código en varios lenguajes de programación tales como C++, Java o Python (generando archivos “.h”, “.c”, “.java” y “.py”), con clases por cada tipo de mensaje descrito. Cada clase generada contiene métodos de acceso simples para cada campo y métodos para serializar y analizar toda la estructura en bytes.

En el formato binario de Protobuf cada campo de un mensaje se precede por una clave que combina el número de campo (definido en el archivo “.proto”) y el *wire type*, que indica la naturaleza del dato (como entero, cadena, mensaje anidado, etc.). Los números de campo y los wire types son esenciales para identificar y procesar cada parte del mensaje durante la serialización y deserialización. Los valores se codifican de formas específicas dependiendo de su tipo; por ejemplo, los enteros utilizan una codificación *varint* (enteros de longitud variable) que ajusta el tamaño del dato al mínimo necesario, mientras que las cadenas y bytes se preceden de su longitud, también codificada como varint, seguida por los datos mismos.

FlatBuffers.

FlatBuffers¹⁴, a veces llamado Flatbuf, es una librería de serialización multiplataforma también creada por Google originalmente destinada para el desarrollo de videojuegos y otras aplicaciones de rendimiento crítico. Utiliza un formato con esquema, neutro en cuanto a la plataforma y lenguaje. La documentación oficial afirma que Flatbuf ofrece una mejor eficiencia en memoria y velocidad en comparación con protobuf, debido a la utilización del concepto de un buffer binario plano que no requiere de desempaquetado ni de convertirse en otra representación de datos al momento de acceder los datos. Esto implica que el proceso de deserialización suele ser mucho más rápido.

El lenguaje de esquema de Flatbuf se basa en dos maneras de definir objetos, tablas y estructuras. Las tablas son la forma principal y menos restringida y pueden incluir campos de diferentes tipos de datos, estructuras y

¹⁴ Página oficial de FlatBuffers: <https://flatbuffers.dev/>

otras tablas anidadas. Las estructuras están más restringidas en términos de que solo pueden incluir valores escalares u otras estructuras y se utilizan para objetos más pequeños y, a su vez, se pueden acceder más rápidamente. En todos los casos, se necesita al final una definición de “tipo raíz” que declara cuál será la tabla raíz para los datos serializados en caso de que se usen múltiples tablas. La forma en que funciona Flatbuf es similar a Protobuf, una vez que se escribe el archivo de esquema ".fbs", el archivo se compila utilizando el compilador del lenguaje objetivo, la compilación genera los archivos de cabecera que incluyen clases de ayuda para acceder y construir datos serializados. Para leer el buffer de nuevo, solo se necesita obtener el puntero al objeto raíz para tener un acceso in situ a los campos.

MessagePack.

MessagePack¹⁵ es una librería y una especificación de código abierto de serialización binaria de objetos. Se fundamenta principalmente en dos conceptos: el sistema de tipos y los formatos. El sistema de tipos establece los tipos de datos admitidos, abarcando los principales tipos de datos con algunas limitaciones en el rango de valores enteros y el tamaño máximo de bytes de cadenas de texto. Este sistema también incorpora el concepto de mapa, que representa un par de objetos clave-valor. El concepto de formatos describe cómo se codifica cada tipo de dato en binario. Por ejemplo, se utiliza un único byte para codificar enteros pequeños, mientras que las cadenas requieren un byte adicional para su codificación, además de la codificación de las propias cadenas. MessagePack emplea una lógica de empaquetado y desempaquetado para referirse a los procesos de codificación/decodificación. Este mecanismo está diseñado para ser ligero y tener una huella de memoria pequeña; al ser un mecanismo sin esquema, su implementación resulta más sencilla en comparación con Protobuf y Flatbuf.

¹⁵ Página oficial de MessagePack: <https://msgpack.org/>

Boost.Serialization.

Boost.Serialization¹⁶ es una librería parte del conjunto de librerías Boost¹⁷ desarrollada en 2002 por Robert Ramey. Esta librería proporciona facilidades para serializar y deserializar objetos en C++ siendo capaz de manejar una amplia variedad de casos de uso, incluyendo la serialización de contenedores de la librería de plantillas estándar, objetos que contienen punteros a otros objetos (gestionando correctamente la serialización de grafos de objetos para evitar duplicaciones y bucles infinitos), y soporta versiones de objetos para facilitar la compatibilidad hacia adelante y hacia atrás entre versiones. Una de las características clave de Boost.Serialization es su flexibilidad en cuanto a los formatos de salida, soporta varios formatos de serialización, incluyendo binario, texto y XML.

Para que una clase pueda ser serializada o deserializada en Boost.Serialization es necesario definir funciones y métodos a mano para especificar cómo se deben guardar y cargar los datos de los miembros de la clase. Esto se realiza generalmente mediante la implementación de la función amiga “serialize” dentro de la clase que se desea serializar. La función “serialize” utiliza el operador “&” para separar los datos miembros a procesar.

¹⁶ Boost.Serialization: https://www.boost.org/doc/libs/1_79_0/libs/serialization/doc/tutorial.html

¹⁷ Conjunto de librerías de código abierto creadas por Beman Dawes y David Abrahams que buscan extender las funcionalidades de C++.

Diseño de serialización de Yu Qi.

Yu Qi es un ingeniero de Zhuhai, China, fundador y actual director técnico de la organización PureCpp¹⁸, una comunidad de investigación libre que pública código abierto y artículos técnicos utilizando C++ moderno y que tiene por objetivo y misión mejorar la eficiencia y productividad empresarial de China y del mundo a través de la innovación tecnológica.

Qi (2017) presentó en la conferencia anual CppCon¹⁹ un conjunto de técnicas innovadoras para conseguir un primer mecanismo básico de reflexión en tiempo de compilación utilizando C++17. Estas técnicas combinan el uso de macros y metaprogramación de plantillas, y con ellas expuso una demostración de nuevas formas de implementar librerías de serialización, frameworks de mapeo relacional de objetos (ORMs) e incluso un potencial protocolo para llamadas a procedimientos remotos (RPC).

Cinco años más tarde, en una edición posterior de CppCon, Qi (2022) compartió la reingeniería de sus técnicas de reflexión en tiempo de compilación, abandonando por completo el empleo de macros y reemplazándolos por Conceptos, una herramienta incorporada en C++20. Junto con esta reingeniería, también introdujo el diseño de una librería de serialización binaria basada en estas técnicas de reflexión estática, y demostró que, dadas unas circunstancias específicas, su librería podía serializar y deserializar objetos desde veinte hasta cien veces más rápido que algunas de las librerías más utilizadas en la industria, como Protobuf y MessagePack.

El diseño de Qi (2022) se cimienta en la aplicación de la reflexión estática para construir un esquema de los datos en tiempo de compilación, y para optimizar los algoritmos de serialización basándose en las propiedades de los objetos a ser serializados. Por ejemplo, al inspeccionar el tipo de un objeto es posible conocer de antemano si éste se aloja en una región contigua de memoria, y en base a este tipo de características elegir una función de serialización especializada. En este diseño, el formato de los datos de un objeto en estado serializado es el mismo que su

¹⁸ Página web oficial de PureCpp: <http://purecpp.cn/>

¹⁹ CppCon es una conferencia anual sobre C++ que reúne a profesionales y expertos para compartir conocimientos y avances en el lenguaje.

representación de bytes en memoria, cualidad que permite hacer copias de regiones de memoria directamente en un *buffer* o archivo de datos. Sin embargo, esto último tiene implicaciones negativas, puesto que impide que el archivo de datos serializado pueda ser distribuido a otras plataformas; al copiar una región de memoria directamente se transcriben consigo bytes de relleno que son introducidos por el compilador, y tanto el orden de estos bytes de relleno como los bytes que sí contienen los datos son dependientes de la plataforma donde ocurre el proceso de serialización y posteriormente dependientes de donde ocurre el proceso contrario. Esto significa que distintas plataformas escribirán y leerán los datos de manera diferente, afectando la integridad de la información.

Qi (2022) publicó pruebas de rendimiento donde demostraba que, ignorando el ordenamiento de bytes y la alineación en memoria requerida por la computadora para acceder a los objetos, una librería de serialización basada en reflexión en tiempo de compilación requería veinte veces menor tiempo para serializar y deserializar objetos que otras librerías. Sin embargo, no hizo pública las mediciones del tamaño de archivos ocupado por su diseño, ni abordó el tema de portabilidad multiplataforma.

Capítulo III

Marco Metodológico

En este capítulo se identifica, se discute y se justifica la metodología elegida para desarrollar la presente investigación.

Metodología Utilizada

Para la realización del presente Trabajo Especial de Grado se seleccionó la metodología incremental descrita por Pressman (2010) junto con la metodología de pruebas de software de Somerville (2005). El presente proyecto tuvo como objetivo desarrollar una librería de software siguiendo seis objetivos específicos donde cada objetivo dependía directamente de la consecución de los objetivos previos, haciendo que la decisión de utilizar una metodología incremental fuese assertiva, debido a que esta metodología tiene como característica principal el desarrollo de software dividido en partes completas, donde cada parte representa un prototipo funcional del producto de software final.

Para Pressman (2010) el modelo incremental combina elementos de los flujos de proceso lineal y paralelo, aplicando secuencias lineales en forma escalonada a medida que avanza el calendario de actividades, como se muestra en la Figura 1. Cada secuencia lineal produce incrementos de software susceptibles de ser entregados. En los modelos incrementales cada incremento representa una versión funcional del software desarrollado.

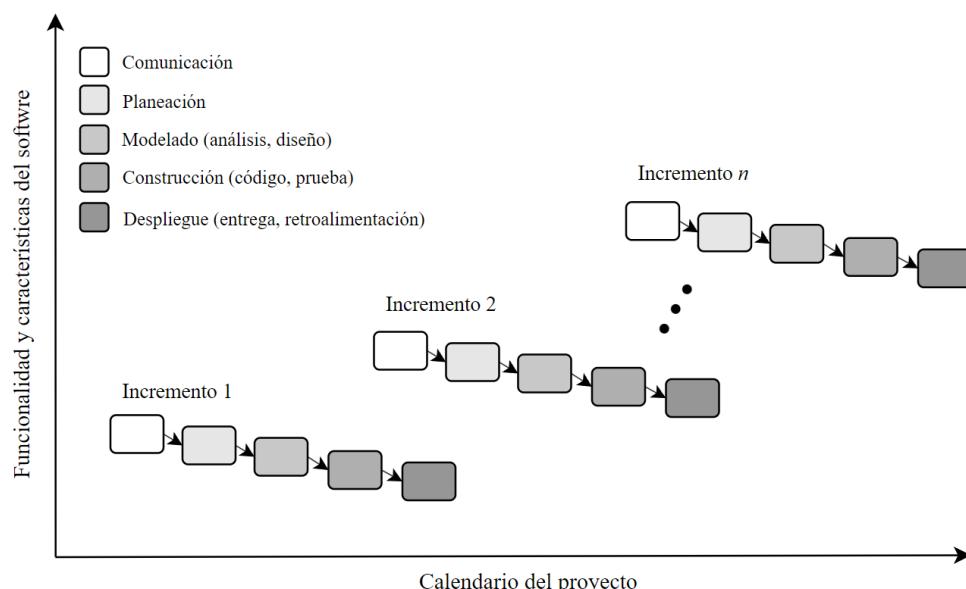


Figura 1. Modelo de proceso incremental. Tomado de “*Ingeniería del Software: Un enfoque práctico*” (p. 36) de Pressman (2010).

El proceso de software es un conjunto de actividades, acciones y tareas que conducen a la creación o evolución de un producto de software (Pressman, 2010). Una librería de código se considera un producto de software ya que posee un propósito definido, características específicas, requisitos de calidad y puede ser distribuida y utilizada por usuarios finales o desarrolladores de software. Una estructura de proceso general para la ingeniería de software consta de cinco actividades estructurales: comunicación, planeación, modelado, construcción y despliegue. La comunicación efectiva es fundamental antes de iniciar cualquier trabajo técnico, ya que permite entender los objetivos y reunir los requisitos necesarios para definir las características del software. La planeación actúa como un mapa que guía el proyecto, detallando tareas técnicas, riesgos, recursos, productos y cronogramas. El modelado permite crear representaciones del software para comprender mejor el problema y su solución. La construcción incluye la generación de código y las pruebas necesarias para detectar errores. Finalmente, el despliegue entrega el software al consumidor para su evaluación y retroalimentación.

Toda construcción de código debe ir acompañada de pruebas. Según Sommerville (2005) “las pruebas exhaustivas, en las que cada posible secuencia de ejecución del programa es probada son imposibles. Las pruebas, por lo tanto, tienen que basarse en un subconjunto de posibles casos de prueba” (p. 493). Sommerville (2005) describe un modelo general para pruebas que consta de cuatro etapas: diseño de los casos de prueba, preparación de los datos de prueba, ejecución de las pruebas y comparación de los resultados obtenidos con los datos de prueba.

Las actividades del proyecto fueron realizadas en un plazo de veinte semanas, siguiendo los procesos mostrados en la Figura 2. Se contó con una primera etapa lineal donde se diseñó una especificación técnica de formato de archivo binario. Seguida de un proceso incremental, donde se trabajó un incremento del proyecto por cada módulo que se desarrolló de la librería. Finalmente, hubo una última etapa lineal donde se evaluó el rendimiento del producto terminado. Más detalladamente:

1. Diseño de un formato de archivo binario: fue la etapa más temprana del proyecto; representó el análisis y diseño de la especificación técnica del formato de archivo binario, cubriendo tanto cómo estarían representados en bytes los objetos serializados como cuál información se dispone en la cabecera del formato y qué distribución se ha de utilizar en la carga útil.

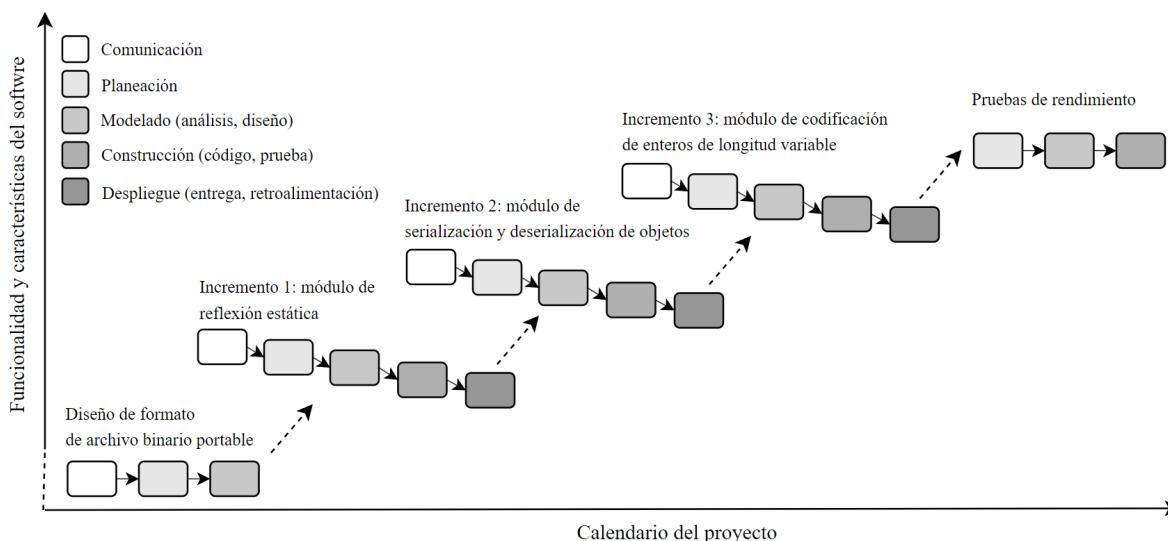


Figura 2. Calendario del proyecto siguiendo el modelo incremental de Pressman. Adaptado de “Ingeniería del Software: Un enfoque práctico” (p. 36) de Pressman (2010).

2. Primer incremento, módulo de reflexión en tiempo de compilación: constó del análisis y diseño de clases, conceptos y funciones genéricas de introspección de tipos de datos, a la par del análisis y diseño de un mecanismo para la configuración y manejo de alineación de datos. Seguido tuvo lugar la implementación de código, diseño y preparación de casos de prueba, ejecución de las pruebas y comparación de los resultados. Y por último, se documentó la interfaz pública de programación del módulo.
3. Segundo incremento, módulo de serialización y deserialización de objetos: se realizó el análisis y diseño de clases y funciones de serialización y deserialización, junto con el análisis y diseño de un mecanismo de detección y conversión de orden de bytes. Posteriormente se desarrolló el código, y se hizo el diseño, preparación y ejecución de las pruebas unitarias y de integración. Como cada módulo, también se documentó la interfaz pública de programación.
4. Tercer incremento, módulo de codificación de enteros de longitud variable: el último de los módulos dentro del alcance del proyecto; en este incremento se realizó el análisis y selección de los algoritmos de codificación de números enteros de longitud variable, y estos fueron implementados en la librería. Se crearon, prepararon y ejecutaron los casos de prueba. Como los dos anteriores módulos, este también contó con su debida documentación de interfaz pública de programación.
5. Pruebas de rendimiento: representó el último objetivo específico del proyecto; se siguió un flujo de proceso lineal independiente de los incrementos a razón del análisis,

diseño y desarrollo de las herramientas de monitoreo que se utilizaron. Se hizo el desarrollo de un conjunto de pruebas de rendimiento de la librería de serialización resultante del proyecto, y el desarrollo de estas mismas pruebas con las librerías de serialización más utilizadas en la industria. Por último, se evaluó y comparó los resultados obtenidos y se confeccionó un informe de rendimiento.

Para llevar a cabo la investigación se organizó un cronograma de trabajo, como se muestra en la Tabla 1. Este cronograma sirvió de referencia para la ejecución de las actividades realizadas durante las veinte semanas de duración del proyecto.

Tabla 1

Plan de trabajo por semana empleado para el desarrollo de la investigación. Autoría propia.

Actividad	Semana																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
bytes.																				
Impl. clases y funciones genéricas de serialización y deserialización de objetos.														#####						
														9-11						
Impl. mecanismo de detección y conversión de orden de bytes.														#####						
														9-11						
Impl. mecanismos de manejo de errores: excepciones y códigos de error.														####						
														10-11						
Diseño, construcción y ejecución de pruebas unitarias y pruebas de integración.														#####						
														10-12						
Documentación de interfaz de programación.														####						
Incremento 3: Módulo de Enteros de Longitud Variable.														#####						
Análisis, selección y diseño de métodos y algoritmos de codificación de enteros de longitud variable.														####						
														12-13						
Diseño de clases y funciones de enteros de longitud variable.														####						
														13-14						
Impl. clases y funciones de enteros de longitud variable.														#####						
														13-15						
Diseño, construcción y ejecución de pruebas unitarias y pruebas de integración.														#####						
														14-16						
Documentación de interfaz de programación.														####						
Pruebas de rendimiento.														#####						
														15-16						
Selección, diseño y desarrollo de herramientas de <i>benchmarking</i> y <i>profiling</i> .														####						
														16-17						
Diseño de casos de prueba de rendimiento y preparación de los datos.														####						
														16-17						
Desarrollo de pruebas de rendimiento de la librería de serialización.														####						
														17-18						
Desarrollo de pruebas de rendimiento de las librerías más utilizadas en la industria.														####						
														19-20						
Evaluación, comparación y análisis de rendimiento. Informe de rendimiento.														#####						
														17-20						

Capítulo IV

Desarrollo y Resultados

En este capítulo se describen en detalle los procedimientos ejecutados para la obtención de los resultados del estudio y se presentan los hallazgos de la investigación.

Formato de Archivo Binario

La primera etapa del proyecto correspondió con el diseño de una especificación de formato de archivo binario. Partiendo del trabajo de Qi (2022), se creó un formato portable independiente del ordenamiento de bytes y con capacidad de compresión de números enteros.

Análisis del formato de Yu Qi.

En el diseño de Qi (2022) los objetos se codifican a un formato binario de manera idéntica a cómo se representan en memoria, implicando que en caso de tener bytes de rellenos estos también son codificados. Esta técnica de serialización vuelve el formato binario incompatible entre diferentes plataformas por dos razones: la inyección de estos bytes de relleno y el orden de bytes de los objetos serializados depende completamente de la arquitectura de procesador en donde ocurren los procesos de serialización y deserialización; cuando una plataforma intenta deserializar un archivo binario que fue serializado en otra plataforma diferente, la interpretación de los datos obtenidos será potencialmente distinta, y la integridad de los datos estará comprometida. Este diseño no tiene un mecanismo de validación en estos casos, e ignora la alineación de los objetos y el ordenamiento de los bytes.

El formato binario propuesto por Qi (2022) posee dos simples componentes: un hash de esquema seguido de la carga útil. El hash es calculado según el esquema de los datos a serializar y funciona como método de verificación. Como mecanismo para incorporar retrocompatibilidad, Qi (2022) propuso una clase genérica que actúa como envoltorio de un objeto opcional. Al momento de generar el hash de verificación todos los objetos de dicho envoltorio son ignorados, haciendo que un hash de verificación de una versión anterior sea compatible con versiones posteriores. Aunque ingeniosa, esta estrategia tiene un inconveniente serio: los objetos compatibles no generan un hash de verificación, por lo que no son verificados al deserializarlos y pueden ser interpretados incorrectamente.

Diseño del formato de archivo binario.

El diseño de formato binario resultado de la investigación se fundamenta desde el diseño original de Qi (2022) pero aborda y soluciona todos los problemas de portabilidad multiplataforma.

Alineación de objetos.

En favor de implementar un formato de archivo binario portable, se consideró las situaciones y razones por las cuales un objeto en memoria posee una alineación específica, siendo estas dependientes de la arquitectura del procesador. Se decidió que la codificación de los objetos hacia un estado serializado no debe incluir nunca los bytes de relleno que pueda tener una representación de objeto en memoria. Al hacer esto, un objeto serializado sólo comprenderá los bytes requeridos por su representación de valor.

Más específicamente, los objetos en un estado serializado seguirán una especificación de codificación que no permite bytes de relleno para satisfacer la alineación de objetos. En este sentido, los objetos serializados se representan como estructuras de datos empaquetadas. Esta decisión de diseño puede afectar el rendimiento de los procesos de serialización y deserialización, pero es necesaria para establecer la independencia multiplataforma del formato de archivo binario.

Ordenamiento de bytes.

Se decidió que el ordenamiento de los bytes no debe estar determinado por la plataforma que realiza los procesos de serialización y deserialización (plataforma host), sino que el orden de bytes tiene que estar bien definido en el formato de archivo binario para la correcta codificación y decodificación de los objetos de modo en que se soporte la portabilidad multiplataforma. Se delimitó dos ordenamientos posibles: little-endian y big-endian. El orden debe estar establecido en una opción de formato situada en la cabecera del archivo binario.

Con este diseño, queda como responsabilidad de la plataforma host cambiar el ordenamiento de los bytes en caso de ser requerido para acceder a

los datos. Por ejemplo, si un archivo binario tiene un orden big-endian y la plataforma encargada de deserializar dicho archivo posee una arquitectura little-endian, será mandatorio cambiar el orden de bytes de los objetos serializados. Se fijó la utilización de little-endian como orden de bytes por defecto en caso de que ningún orden haya sido establecido en las opciones de formato.

Codificación de objetos.

Al igual que en el diseño de serialización de Qi (2022), los objetos son codificados hacia un estado serializado en una secuencia de bytes que sigue una estructura uniforme y predecible. La unidad mínima de codificación está expresada en objetos fundamentales: números enteros y reales. A partir de estos objetos fundamentales se conforman objetos compuestos: estructuras, tuplas, contenedores, objetos opcionales, punteros propietarios y variantes.

Los objetos fundamentales son los únicos afectados por el ordenamiento de bytes, cuando un cambio de bytes es requerido son estos objetos los que se deben convertir de un orden a otro. La excepción a esta regla son los objetos fundamentales que ocupan un tamaño de un byte, como los enteros fijos de 8 bits.

Esquema de datos.

El esquema de datos del archivo binario se constituye mediante la composición de objetos con una codificación válida, es decir, una codificación admitida por el formato. Este esquema define el orden y la disposición de los objetos a ser serializados y deserializados. Para mantener el tamaño total del archivo lo más reducido posible, el archivo binario no incorpora directamente el esquema utilizado para codificar los objetos, sino que emplea un mecanismo de verificación.

Se determinó que la cabecera del archivo binario contendrá un conjunto de códigos hash correspondientes a las distintas versiones del esquema de datos utilizado para codificar los objetos. Como parte del soporte de retrocompatibilidad, un esquema de datos puede tener múltiples versiones,

y de cada versión se genera un código hash. Estos códigos hash se utilizan durante el proceso de deserialización de los objetos contenidos en el archivo binario, permitiendo identificar si el esquema empleado para la deserialización es el correcto para cada versión específica.

Retrocompatibilidad.

Para implementar la retrocompatibilidad entre esquemas de datos de versiones anteriores se consideró la estrategia propuesta por Qi (2022) pero se amplió su definición y se incluyeron restricciones. Se determinó la inclusión de objetos compatibles en la codificación de objetos, estos actúan como un envoltorio de objetos opcionales, pero poseen ciertas condiciones especiales. Los objetos compatibles forman parte del esquema de datos únicamente a partir de una versión posterior a la versión original del esquema (primera versión), y no es mandatorio que estén presentes en la carga útil. A diferencia del diseño de Qi (2022), los objetos compatibles sí son considerados para la digestión del código hash generado por las versiones del esquema de datos. A nivel de formato de archivo, los objetos compatibles no acarrean información directa sobre la versión de la cual forman parte, pero sí lo hacen a nivel de estructura de datos, y por ende indirectamente a través de los códigos hash. Todos los objetos compatibles de una misma versión son agrupados para generar el código hash de dicha versión. Esta técnica habilita la compatibilidad con versiones anteriores del esquema de datos al precio de unos pocos bytes de códigos bytes en la cabecera del archivo binario.

Estructura del archivo binario.

Se estableció que el archivo binario estará comprendido por dos segmentos principales: la cabecera de archivo y la carga útil. Los datos contenidos en ambos segmentos se representan mediante la codificación de objetos.

Cabecera de archivo. En la cabecera se sitúa un identificador de archivo, las opciones de formato y un arreglo de códigos hash del esquema de datos. Este segmento contiene la información que permite identificar y manejar el archivo binario correctamente.

Carga útil. La carga útil contiene los datos de los objetos serializados en el archivo binario. A su vez, la carga útil está dividida en subsegmentos que alojan los datos. Se asigna un subsegmento a cada versión del esquema de datos, a razón de dar soporte de retrocompatibilidad.

Especificación técnica del formato de archivo binario.

A partir del diseño de formato de archivo binario se creó una especificación técnica que describe la codificación de los objetos hacia un estado serializado y detalla los componentes del formato: la disposición de los datos de la cabecera y los detalles de implementación de la carga útil.

Codificación de objetos.

La codificación de los objetos hacia un estado serializado se puede dividir en dos clasificaciones al igual que ocurre con los tipos de datos en C++, codificación de objetos fundamentales y codificación de objetos compuestos. Los objetos fundamentales expresan la unidad mínima de codificación: números enteros con signo y sin signo (tanto de ancho de bit fijo como variable) y números reales, booleanos y caracteres; aunque estos dos últimos son solamente alias de números enteros sin signo, véase Tabla 2. Los objetos compuestos se componen de objetos fundamentales o de otros objetos compuestos y se codifican recursivamente.

Números enteros fijos. Los números enteros de ancho fijo se clasifican en dos categorías: con signo y sin signo. Ocupan un tamaño en bytes determinado por su ancho de bits, que puede ser de 8, 16, 32 o 64 bits. Los enteros sin signo y los enteros con signo positivo se representan directamente en binario, mientras que los enteros con signo negativo se codifican utilizando el complemento a dos.

Números reales. La representación de números reales se realiza utilizando el estándar IEEE-754 para aritmética de coma flotante, teniendo soporte para los formatos "binary32" y "binary64". Estos formatos suelen ser conocidos como coma flotante de precisión simple

y coma flotante de precisión doble respectivamente, ocupando un ancho de 32 y 64 bits.

Tabla 2

Codificación de objetos fundamentales del formato de archivo binario “Teg”. Autoría propia.

Clasificación	Nombre técnico	Valor mínimo	Valor máximo	Tamaño en bytes
Números enteros de ancho fijo sin signo.	u8	0	2^8	1
	u16	0	2^{16}	2
	u32	0	2^{32}	4
	u64	0	2^{64}	8
Números enteros de ancho fijo con signo.	i8	2^{8-1}	$2^{8-1} - 1$	1
	i16	2^{16-1}	$2^{16-1} - 1$	2
	i32	2^{32-1}	$2^{32-1} - 1$	4
	i64	2^{64-1}	$2^{64-1} - 1$	8
Números reales (Aritmética de coma flotante IEEE-754)	f32	$2^{-126} \approx 1.18 \times 10^{-38}$	$2^{128} \approx 3.4 \times 10^{38}$	4
	f64	$2^{-1022} \approx 2.23 \times 10^{-308}$	$2^{1024} \approx 1.8 \times 10^{308}$	8

Booleanos. Los booleanos son codificados utilizando un entero fijo sin signo de 8 bits. Un valor de `0x00` representa “falso” y cualquier valor contenido en el rango [`0x01` , `0xFF`] representa “verdadero”.

Números enteros de longitud variable. Este tipo de dato no forma parte de los tipos fundamentales codificados en la librería sino que es un tipo compuesto. Los números enteros de longitud variable se codifican utilizando enteros de 8-bits sin signo (bytes). La librería utiliza la codificación ULEB-128 para estos números (véase Capítulo 4.4).

Caracteres. Los caracteres son codificados utilizando números enteros fijos sin signo de ancho de 8, 16, 32 y 64 bits. Esto permite representar cualquier esquema de codificación de caracteres (como UTF-8, UTF-32 o Latin1).

Estructuras. Una estructura es codificada con la secuencia de bytes de la codificación de sus miembros variables. Los miembros variables se codifican uno a uno, en orden, desde el primer miembro hasta el último siguiendo su propia secuencia de codificación.

Tuplas. Las tuplas a veces son llamadas estructuras anónimas, y su codificación es similar, pero en lugar de miembros las tuplas tienen elementos. Se codifica cada elemento de una tupla en orden, del primer elemento al último, siguiendo su propia secuencia de codificación.

Contenedores. Un contenedor se representa codificando un entero de ancho fijo o un entero de longitud variable con el valor del tamaño del contenedor, seguido de la secuencia de bytes de codificación de cada elemento contenido. Los elementos poseen una codificación no uniforme compuesta de otros objetos, como por ejemplo enteros, caracteres, otros contenedores u otros objetos. Esto último significa que dos elementos de un mismo contenedor pueden ocupar un tamaño de bytes diferente dependiendo de su codificación inherente.

Punteros propietarios. Los punteros propietarios codifican directamente su elemento contenido. No es posible representar punteros nulos.

Objetos opcionales. Los objetos opcionales se codifican con un booleano que indica la presencia del elemento contenido. De ser “verdadero” el valor booleano se sigue la secuencia de bytes con la codificación de objeto del elemento contenido; en caso contrario la secuencia de bytes termina y no se incluye ninguna codificación adicional.

Variantes. Un objeto variante es representado mediante el índice y valor de una de su alternativa activa (o alternativa contenida). Se codifica primero un entero fijo sin signo con el valor del índice de la alternativa seguido de la secuencia de bytes de codificación de objeto de la alternativa.

Objetos compatibles. Los objetos compatibles representan un envoltorio de un objeto opcional, por lo que se codifican como objetos opcionales con una condición especial. Los objetos opcionales estarán representados en la secuencia de bytes como mínimo con un objeto

booleano. Este no es el caso para los objetos compatibles, que pueden no estar presentes en la carga útil en lo absoluto.

Codificación del esquema de datos.

Los esquemas de datos se codifican en un formato de texto y luego se procesan utilizando el algoritmo MD5. En la Tabla 3, se presenta la especificación de codificación de los esquemas según los diferentes tipos de datos. Para cada versión del esquema, se genera su codificación y se obtienen los primeros 32 bits del hash MD5. Estos valores se insertan en una tabla de hashes ubicada en la cabecera útil, funcionando como un mecanismo de verificación del formato de datos.

Tabla 3

Codificación a formato de texto de los esquemas de datos en la librería “Teg”. Autoría propia.

Tipo de dato	Codificación		
	Prefijo	Sufijo	Valor
Números enteros sin signo. 8 a 64 bits.			u8, u16, u32, u64
Números enteros con signo. 8 a 64 bits.			i8, i16, i32, i64
Números de coma flotante. 32 y 64 bits.			f32, f64
Números enteros de longitud variable.			varint32, varint64, varuint32, varuint64
Estructuras de datos	{	}	Codificación de variables miembros recursivamente
Tuplas	()	Codificación de los elementos recursivamente
Contenedor	[]	Codificación de los elementos recursivamente
Arreglos de C y contenedores de tamaño fijo	#		Codificación del tamaño en número arábigos seguido de la codificación del contenedor.
Objeto opcional	?		Codificación del elemento contenido
Puntero propietario	*		Codificación del elemento contenido
Variantes	<	>	Codificación de las alternativas
Objetos compatibles	@		Codificación del elemento contenido
Codificación definida por el usuario			Nombre de espacio unido con “::” al nombre de la clase.

Cabecera.

La cabecera del archivo binario se codifica con orden de bytes little-endian y posee tres componentes: un identificador de archivo binario (palabra mágica), las opciones del formato de archivo binario, y una tabla de hashes del esquema de datos.

Palabra mágica. La palabra mágica es un identificador utilizado para reconocer el formato de archivo binario. Está compuesta de 4 bytes. Los primeros tres bytes son constantes y deben ser iguales a `0x54`, `0x45`, y `0x47`; siendo estos una cadena de caracteres ASCII que representa la palabra `TEG` (nombre oficial de la librería). El último byte es un número que especifica la versión del formato. La versión actual de formato es `TEG1` y su palabra mágica es `0x54454701`.

Opciones de formato. Representadas mediante banderas de un entero fijo sin signo de 32 bits, son las opciones con las cuales se codificaron los objetos. La serialización de objetos se efectúa con unas opciones determinadas, y por consiguiente, el proceso de deserialización necesita esta misma configuración para poder decodificar los datos de forma correcta.

Como se muestra en la Tabla 4, las opciones incluyen la especificación del orden de bytes (little-endian o big-endian), el límite de alojamiento permitido, el tamaño máximo de los contenedores y el número máximo de alternativas de los objetos variantes. Las opciones están organizadas en grupos y tienen una prioridad asignada dentro de cada grupo. Esto significa que, si se configuran varias opciones de un mismo grupo, prevalecerá la opción con mayor prioridad. Algunos rangos de bits no tienen asignados opciones y están reservados para versiones posteriores del formato de archivo binario.

Tabla 4

Opciones del formato de archivo binario “Teg” desarrollado en la investigación. Autoría propia.

Opción	Bit	Grupo	Prio.	Descripción
native_endian	0	endian	0	Ordenar los bytes utilizando el endian predeterminado (nativo) de la arquitectura de procesador de la plataforma host.
little_endian	1	endian	2	Ordenar los bytes utilizando little-endian.
big_endian	2	endian	1	Ordenar los bytes utilizando big-endian.
allocation_limit_1gib	4	allocation	0	Limitar el alojamiento máximo de bytes en 1 GiB.
allocation_limit_2gib	5	allocation	2	Limitar el alojamiento máximo de bytes en 2 GiB.
allocation_limit_4gib	6	allocation	1	Limitar el alojamiento máximo de bytes en 4 GiB.
container_size_native	8	container	0	Utilizar un entero fijo sin signo con el mismo ancho de bits que un puntero nativo de la plataforma host para codificar el tamaño del contenedor.
container_size_varint	9	container	1	Utilizar un entero de longitud variable para codificar el tamaño de los contenedores.
container_size_1b	10	container	2	Utilizar un entero sin signo de 8, 16, 32 o 64 bits respectivamente para codificar el tamaño de los contenedores.
container_size_2b	11	container	3	
container_size_4b	12	container	5	
container_size_8b	13	container	4	
variant_index_native	14	variant	0	Utilizar un entero fijo sin signo con el mismo ancho de bits que un puntero nativo de la plataforma host para codificar el índice de la alternativa activa en variantes.
variant_index_1b	15	variant	4	Utilizar un entero sin signo de 8, 16, 32 o 64 bits respectivamente para codificar el índice de la alternativa activa en variantes.
variant_index_2b	16	variant	1	
variant_index_4b	17	variant	2	
variant_index_8b	18	variant	3	
force_varint	24	varint	0	Forzar la conversión de todos los números enteros fijos mayores de 2 bytes a números de longitud variable.
S/A	3, 7, 19..23, 25..32	S/A	S/A	Bits reservados.

Por defecto se utilizan las opciones de cada grupo con la mayor prioridad: ordenar los bytes utilizando little-endian, limitar el alojamiento de bytes en 2 GiB, utilizar enteros sin signo de 32 bits para codificar el tamaño de los contenedores (permitiendo almacenar hasta 2^{32} elementos) y utilizar enteros sin signo de 8 bits para los codificar los índices de la alternativa activa en variantes (permitiendo hasta 255 alternativas).

Tabla de hashes. Posterior a la palabra mágica y a las opciones del formato de archivo binario, se sitúa en la cabecera un contenedor con los hashes de cada versión del esquema de datos utilizado en el proceso de serialización. Esto manifiesta un mecanismo de verificación que tiene por objetivo garantizar la integridad de la estructura de datos al momento de deserializar un archivo binario. Este contenedor de hashes permite la retrocompatibilidad entre archivos binarios sin comprometer la integridad de los esquemas de datos más antiguos. Cada versión se expresa con los primeros 32 bits de la digestión MD5 del esquema de datos.

Carga útil.

Seguido de la cabecera, en la carga útil se almacenan todas las secuencias de bytes de los objetos codificados según el esquema de datos. La carga de datos está seccionada según las versiones del esquema de datos y en cada nueva versión se sitúan los nuevos objetos en la cola de la versión anterior. Por ejemplo, si un esquema tiene tres versiones, la carga útil tendrá tres secciones de datos, siendo las últimas dos secciones las que contendrán los nuevos datos compatibles agregados al esquema.

Módulo de Reflexión en Tiempo de Compilación

En el primer incremento del producto de software se desarrolló el módulo de reflexión estática. En su interfaz de programación, este módulo exporta las funciones de introspección e intersección de tipos agregados. “Teg” es el nombre del formato de archivo binario y de la librería desarrollada en la investigación; todos los símbolos que se declaran bajo el nombre de espacio `teg` son producto del presente proyecto.

Mecanismos de reflexión.

Qi (2022) publicó un algoritmo de metaprogramación de plantillas en el cual se aprovechaba la inicialización de tipos agregados para contar el número de variables miembros de una clase agregada. Junto con esto, también dio a conocer una función de orden superior, ésta utiliza estos metadatos para invocar una función pasando como argumento todas las variables miembros de un objeto. En la librería estándar existe

una función similar a esta, `std::apply`²⁰, que invoca una función con los elementos de una tupla como argumentos, pero no es posible utilizarla para miembros de una clase.

Partiendo de la investigación de Yu Qi, hemos rediseñado los algoritmos de reflexión cambiando por completo el uso de metaprogramación de plantillas a favor de los Conceptos, que facilitan la composición de requerimientos genéricos y optimizan el proceso de compilación. Además, habiendo profundizado en la investigación, hallamos oportunidades de implementación de nuevos algoritmos que permiten lo siguiente: (a) atar las variables miembros de una clase agregada a una tupla de referencias; (b) acceder a las variables miembros de una clase agregada por sus índices de posición; y (c) comparar dos clases agregadas a nivel de variables miembros.

Conteo de variables miembros de una clase.

En el lenguaje C++20 no existe ningún mecanismo estándar para obtener el número de variables miembros de una clase. Qi (2022) demostró un algoritmo capaz de obtener esta meta información en tipos de clases agregadas. Como parte de la investigación, hemos revisado este algoritmo e implementado nuestra propia versión (véase Apéndice C.2.1).

Este algoritmo se centra en la idea de explotar la naturaleza de las clases agregadas. Los tipos de datos agregados tienen una característica única conocida como la inicialización agregada, que permite crear nuevas instancias de estos tipos utilizando una lista de elementos (objetos) sin necesidad de que previamente se haya definido un constructor. Los elementos de la lista deben coincidir en tipo y orden con la definición de las variables miembro de la clase.

Existen otros dos componentes que son necesarios para poder implementar la función de conteo de variables miembros. El primero, un Concepto a través del cual se prueba si una inicialización agregada es válida para una clase, dada una lista de elementos. Y el segundo, una clase parcialmente definida, con la capacidad de ser convertida en un tipo genérico

²⁰ Referencia de la función estándar `std::apply`: <https://en.cppreference.com/w/cpp/utility/apply>

cualquiera. Este segundo componente, la clase genérica parcialmente definida, toma ventaja de las características del lenguaje y del funcionamiento interno del compilador, ya que cuando se utiliza como argumento para inicializar una clase agregada, el compilador realiza una conversión (en inglés, *cast*) hacia el tipo de dato de la variable miembro que se intenta inicializar.

Finalmente, la función recursiva de conteo de miembros se ejecuta de la siguiente manera: dada la clase `T` se inicia con una lista vacía de elementos `A`, en cada recursión se prueba si con dicha lista es posible inicializar `T`, de no ser así, se añade un elemento genérico a la lista y se llama nuevamente a la función. En el momento en que la prueba de inicialización sea positiva, se devuelve el número de elementos de la lista, que corresponde también con el número de miembros variables de la clase.

El costo por utilizar este algoritmo es prácticamente nulo, o básicamente es el mismo que cuando se asigna una variable con un número. Como se muestra en la Figura 3, la función es evaluada en tiempo de compilación y no genera líneas de lenguaje ensamblador de establecimiento de registros, ni saltos, ni de llamadas a otras funciones; únicamente la asignación del número total de variables miembros.

```

1 class User {
2 public:
3     uint64_t id;
4     std::string username;
5     std::string email;
6     uint64_t created_at;
7     uint64_t modified_at;
8 };
9
10 auto const number_of_members = teg::members_count<User>();
11 // Assembly msvc-x64 generated for line 10
12     mov     qword ptr [number_of_members],5

```

Figura 3. Ejemplo del conteo de variables miembros de una clase agregada. Instrucciones ensamblador generadas de la función `teg::members_count()`. Autoría propia.

Visita a variables miembros.

Junto con el algoritmo para obtener el número de variables miembros de una clase agregada, Qi (2022) también mostró el diseño de una función de orden superior que permite acceder de manera genérica e indirecta a los miembros de una clase. Se ha revisado e implementado desde cero este algoritmo (véase Apéndice C.2.2), pero hemos optado por utilizar una estrategia de sobrecarga de funciones en lugar de los condicionales anidados del diseño original. A nivel de tiempo de compilación, esto puede traducirse en una optimización; no obstante, en tiempo de ejecución esto no supone una diferencia significativa.

En la Figura 4, se muestra un ejemplo de la función de visita, donde se pasa como argumento una función anónima (o función *lambda*) y el objeto al cuál se accederán a sus variables miembros. Nótese que la función anónima tiene como parámetro una lista empaquetada de objetos, en este caso corresponden a las cinco variables de la clase `User`. Dentro de la función de visita se desempaqueteta ésta lista llamando a la función `print` por cada miembro.

```

1 auto user = User{1, "Juan Carlos", "jnc99@gmail.com", 1738714000, 1738742800 };
2
3 teg::visit_members(
4     [](auto const&... members) {
5         (print(members), ...);
6     },
7     user
8 );

```



```

1 out
2 1 Juan Carlos jnc99@gmail.com 1738714000 1738742800

```

Figura 4. Ejemplo de utilización de la función genérica de visita aplicada a las variables miembros de una clase agregada. Autoría propia.

Internamente la función genérica `teg::visit_members` opera utilizando vinculación de estructuras (del inglés, *structure binding*), una característica del lenguaje incorporada en C++17. La vinculación de estructuras permite desglosar un objeto en sus respectivos subobjetos

asignándolos a nuevos identificadores. Más detalladamente, dada una función `f` de tipo `F` y un objeto `t` de tipo `T`, se crean tantos nuevos identificadores como variables miembros posea `T` y se vinculan desde el objeto `t`, luego se invoca `f` pasando como argumentos los identificadores creados, efectivamente particionando `t` en todas sus variables y habilitando la manipulación individual de éstas.

Un problema inherente a esta técnica es que la vinculación de estructuras no genera el número de identificadores de manera automática. Esto significa que se deben definir manualmente cuántos identificadores sean necesarios, implicando la implementación de una función por cada número posible de variables miembros (o bien una única función con este mismo número de condicionales anidados). Por defecto, nuestra librería soporta la visita de clases agregadas compuestas de hasta 255 variables miembros, aunque esto es configurable (véase Apéndice A.2).

Entonces encontramos que este algoritmo presenta dos limitaciones significativas. La primera es que solo se puede utilizar con tipos de datos agregados, consecuencia de la limitación del algoritmo de conteo de miembros. Y la segunda es que el número de variables miembros accesibles está restringido según la configuración de la librería. Aun así, es posible reconocer estas restricciones y abordarlas adecuadamente; no solo desde un punto de vista de diseño sino también de implementación, a través del concepto `teg::accesible_aggregate` se determina si un tipo de dato es apto para ser reflejado (véase Apéndice C.2.1).

Acceso a variables miembros por índice de posición.

Hallamos que es posible aplicar algoritmos similares a los presentados por Qi (2022) con el fin de expandir las soluciones a disposición del enfoque de reflexión en tiempo de compilación. En este caso, desarrollamos una función que permite acceder a una variable miembro a través del índice de posición que ocupa en su declaración de clase. Esto, entre otras circunstancias, resulta de utilidad en los casos en que no es necesario referirse a todos los miembros de una clase a la vez.

La estrategia para implementar este algoritmo se apoya en tratar las clases agregadas como si fuesen una tupla (véase Apéndice C.2.3 y Apéndice C.2.4). En sí, una tupla puede verse como una estructura de datos anónima, muy similar en esencia a un tipo agregado. Además, la librería estándar tiene una cabecera dedicada a la manipulación de tuplas; siendo una de sus funciones más útiles `std::get`²¹, que permite acceder individualmente a los elementos de estos objetos, según sus respectivas posiciones o índices.

Utilizando la técnica de vinculación de estructuras en conjunto con la función de la librería estándar `std::tie`²² se consigue atar un objeto de tipo agregado a una tupla de referencias. En la Figura 5 se ve un ejemplo de esto. De esta manera, es posible crear una función genérica de acceso a variables miembros de una clase agregada como se muestra en la Figura 6.

```

1 auto user = User{1, "Juan Carlos", "jnc99@gmail.com", 1738714000, 1738742800 };
2 using Tuple = std::tuple<uint64_t&, std::string&, std::string&,
                           uint64_t&, uint64_t>;
3
4 Tuple user_as_tuple = teg::tie_members(user);
5 print(std::get<0>(user_as_tuple), std::get<2>(user_as_tuple));

1 out
2 1 jnc99@gmail.com

```

Figura 5. Demostración de la función `teg::tie_members` que ata las variables miembro de una clase agregada a una tupla de referencias. Autoría propia.

```

1 auto user = User{1, "Juan Carlos", "jnc99@gmail.com", 1738714000, 1738742800 };
2
3 teg::get_member<2>(user) = "juan@juan-enterprise.com";
4 teg::get_member<4>(user) = 1738749123;
5
6 print(user.email, user.modified_at);

1 out
2 juan@juan-enterprise.com 1738749123

```

Figura 6. Demostración de la función `teg::get_member` que permite el acceso a variables miembros de una clase agregada mediante sus índices de posición. Autoría propia.

²¹ Referencia a la función estándar `std::get`: <https://en.cppreference.com/w/cpp/utility/tuple/get>

²² Referencia a la función estándar `std::tie`: <https://en.cppreference.com/w/cpp/utility/tuple/tie>

Es importante aclarar que si se accede a un miembro de un objeto constante sus referencias también serán constantes. La función de acceso por índice está restringida por las mismas dos limitaciones que el algoritmo de visita de miembros. Solo es posible aplicarlo a tipos agregados y la cantidad de miembros accesibles está delimitada, pero es configurable (véase Apéndice A.2).

Comparación de igualdad a nivel de variables miembros.

El lenguaje C++ no define operaciones de comparación de manera implícita para las clases. Es el usuario quien debe implementar estas operaciones o utilizar operaciones por defecto pero estas deben ser declaradas de manera explícita. A raíz de esto, y tomando ventaja de los mecanismos de reflexión construidos para la investigación, desarrollamos una función genérica que compara la igualdad de dos objetos instancia de clases agregadas (véase Apéndice C.2.5).

Para tipos fundamentales, la operación de comparación de igualdad sí está definida, y esto también es así para cualquier par de tipos que cumplen el Concepto estándar `std::equality_comparable_with`²³. Aprovechando este predicado booleano, y pudiendo acceder de manera secuencial a todas las variables miembros de un tipo agregado, es posible comparar la igualdad de dos objetos miembro a miembro.

En la Figura 7 se muestra la operación de esta función, dónde la clase `Vector3` no tiene definido el operador de igualdad y en su lugar se utiliza la comparación a nivel de miembros. Nótese que no existe un costo adicional por utilizar esta función, las instrucciones generadas en lenguaje ensamblador son las correspondientes a la comparación de tres pares de números de coma flotante.

²³ Referencia del Concepto estándar `std::equality_comparable_with`: https://en.cppreference.com/w/cpp/concepts/equality_comparable

```

1 class Vector3 {
2     public: float x, y, z;
3 };
4
5 auto v1 = Vector3{ 1.0f, 2.0f, 3.0f };
6 auto v2 = Vector3{ 1.0f, 2.0f, 3.0f };
7 // bool test = v1 == v2;           // Error: operator == is not defined.
8 bool test = teg::memberwise_equal(v1, v2); // Ok: true.
9 // Assembly msvc-x64 generated for line 10.
10    ucomiss    xmm7, xmm6          // v1.x == v2.x
11    movd       xmm0, r8d
12    cvtdq2ps   xmm0, xmm0
13    jp         SHORT $LN12@main
14    jne        SHORT $LN12@main
15    ucomiss    xmm9, xmm10         // v1.y == v2.y
16    jp         SHORT $LN12@main
17    jne        SHORT $LN12@main
18    ucomiss    xmm8, xmm0          // v1.z == v2.z
19    jp         SHORT $LN12@main
20    jne        SHORT $LN12@main

```

Figura 7. Ejemplo de la comparación de igualdad a nivel de variables miembros de una clase agregada. Autoría propia.

Con esta técnica se pueden implementar más operaciones de comparación, como menor que, mayor que, entre otros, haciendo posible la comparación de clases agregadas que no tienen definidos operadores de comparación. Por el alcance de la investigación sólo presentamos la comparación de igualdad.

Enfoque de reflexión sobre la alineación de objetos.

Como parte de la primera iteración del producto de software se planificó el análisis y diseño de algún tipo de mecanismo que pudiera ser de utilidad para abordar de manera correcta la alineación de objetos. Esto con el objetivo de ofrecer una interfaz de programación con la cual se pudiese determinar qué bytes pertenecientes a un objeto son necesarios para su correcta representación de forma portable. Al llevar a cabo esta investigación encontramos ciertos desafíos pero también hallamos oportunidades de aplicación del enfoque de reflexión en tiempo de compilación.

Una clase tiene una estructura empaquetada si su representación de objeto en memoria está libre de bytes de relleno. Identificar cuándo un objeto goza de esta característica permite la selección y aplicación eficiente de algoritmos. Por ejemplo, es posible copiar la región de memoria que ocupa un objeto de estructura empaquetada con la garantía de que cada byte de su representación de objeto es también un byte de su representación de valor.

El lenguaje C++ no posee ninguna forma estándar de detectar cuándo una clase incluirá bytes de relleno debido a su requerimiento de alineación. Dada esta limitación, hemos desarrollado un algoritmo novedoso que permite la detección de bytes de relleno en tipos de datos agregados (véase Apéndice C.2 6). Este algoritmo funciona de la siguiente manera: se construyen dos objetos `r` y `m` instancias del tipo `T` donde cada byte contiene el valor `0x00`, el primero se utiliza como un objeto de referencia y el segundo como un objeto mutable. Se itera cada byte de la representación de objeto del objeto mutable y se altera su valor a `0x01`. Comparando la igualdad de `r` y `m` se puede detectar la presencia de bytes de relleno, puesto que, si este es el caso, en alguna iteración se modificará un byte de la representación de objeto que no tiene peso alguno en la representación de valor. En la Figura 8 se demuestra la detección de estructuras empaquetadas con nuestra técnica.

```

1 struct alignas(8) NonPacked {           // sizeof(NonPacked) == 8
2     uint32_t a;                          // 4 bytes
3                               // 3 padding bytes
4     uint8_t b;                          // 1 byte
5 };
6
7 constexpr bool is_packed = teg::concepts::packed_layout<NonPacked>; // false

1 #pragma pack(push, 1)
2 struct Packed {                      // sizeof(Packed) == 5
3     uint32_t a;                        // 4 bytes
4     uint8_t b;                         // 1 byte
5 };
6 #pragma pack(pop)
7
8 constexpr bool is_packed = teg::concepts::packed_layout<Packed>; // true

```

Figura 8. Ejemplo de detección de estructuras empaquetadas utilizando el Concepto `teg::concepts::packed_layout`. Autoría propia.

Módulo de Serialización y Deserialización Binaria

En la segunda iteración del producto de software se desarrolló el módulo de serialización y deserialización binaria de objetos. En su interfaz de programación, este módulo exporta las clases y funciones de serialización que consumen el formato de archivo binario diseñado en la investigación (véase Apéndice B.2.2).

Diseño de la interfaz de serialización.

La fase de diseño del módulo de serialización destacó como una de las más cruciales del proyecto, donde se puso especial énfasis en lograr un equilibrio entre simplicidad y flexibilidad. Este enfoque se centró en garantizar una adecuada separación de las funcionalidades de la librería, con especial atención a la manipulación de un mismo formato de archivo, pero a través de diferentes medios, como la memoria principal y el almacenamiento secundario. En la Figura 9 y la Figura 10 se presentan diagramas detallados del diseño de los procesos de serialización y deserialización.

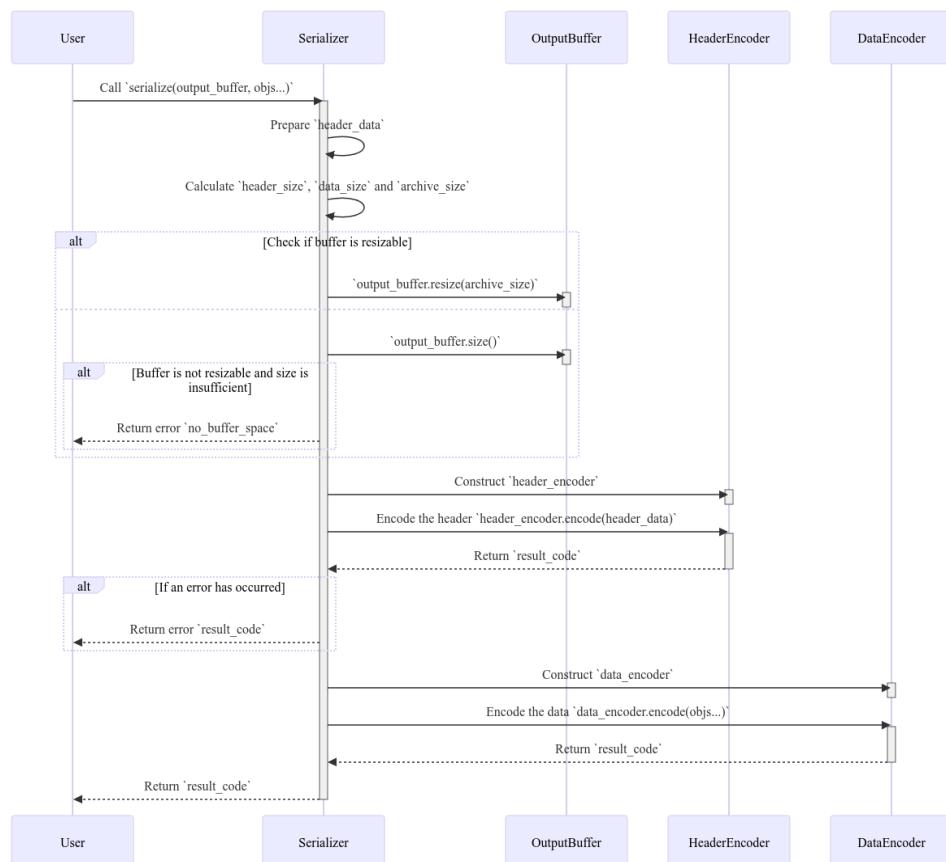


Figura 9. Diagrama de secuencia UML del proceso de serialización binaria de objetos. Autoría propia.

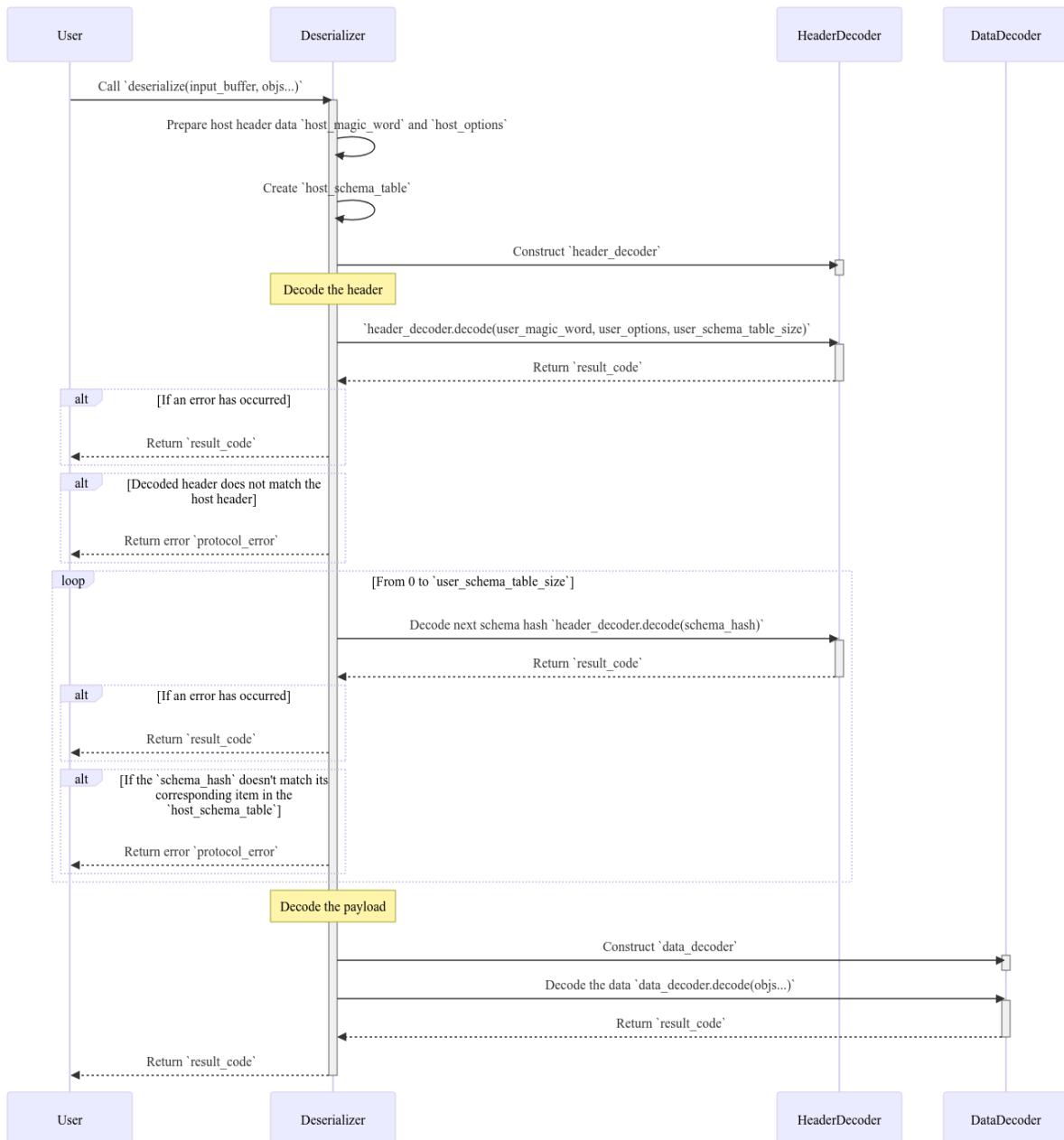


Figura 10. Diagrama de secuencia UML del proceso de deserialización binaria de objetos. Autoría propia.

Se decidió dividir el módulo en dos procesos estrechamente relacionados pero diferenciados: los procesos de serialización y los procesos de codificación. En el marco de esta investigación, los procesos de codificación y decodificación se centran exclusivamente en la transformación directa de los datos hacia una secuencia de bytes. Por otro lado, los procesos de serialización y deserialización involucran la creación completa del archivo binario, considerando los siguientes aspectos: (a) la codificación y decodificación de la cabecera; (b) la verificación de integridad del

formato, que incluye la validación de la versión, las opciones de formato y la integridad del esquema de datos; y finalmente, (c) la codificación y decodificación de la carga útil.

Los procesos de codificación están diseñados para interactuar, ya sea escribiendo o leyendo, con los datos codificados en un medio; en los diagramas se muestra un buffer en memoria, que es el caso por defecto, pero también existe soporte para escribir hacia un flujo de datos, como puede ser hacia un archivo en disco o un *socket* de internet. La Figura 11 y la Figura 12 muestran una vista general de los procesos de codificación y decodificación.

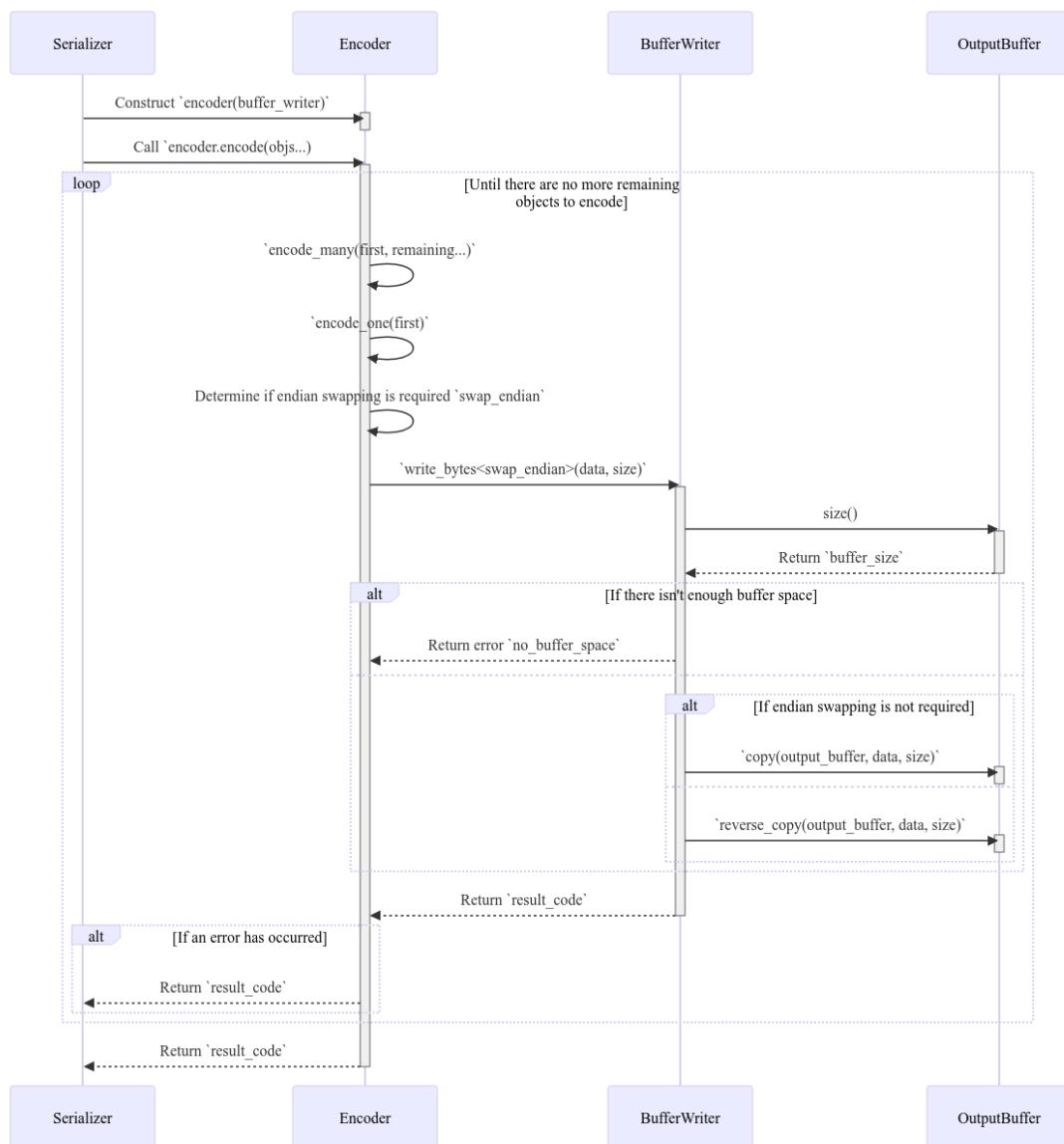


Figura 11. Diagrama de secuencia UML del proceso de codificación binaria de objetos. Autoría propia.

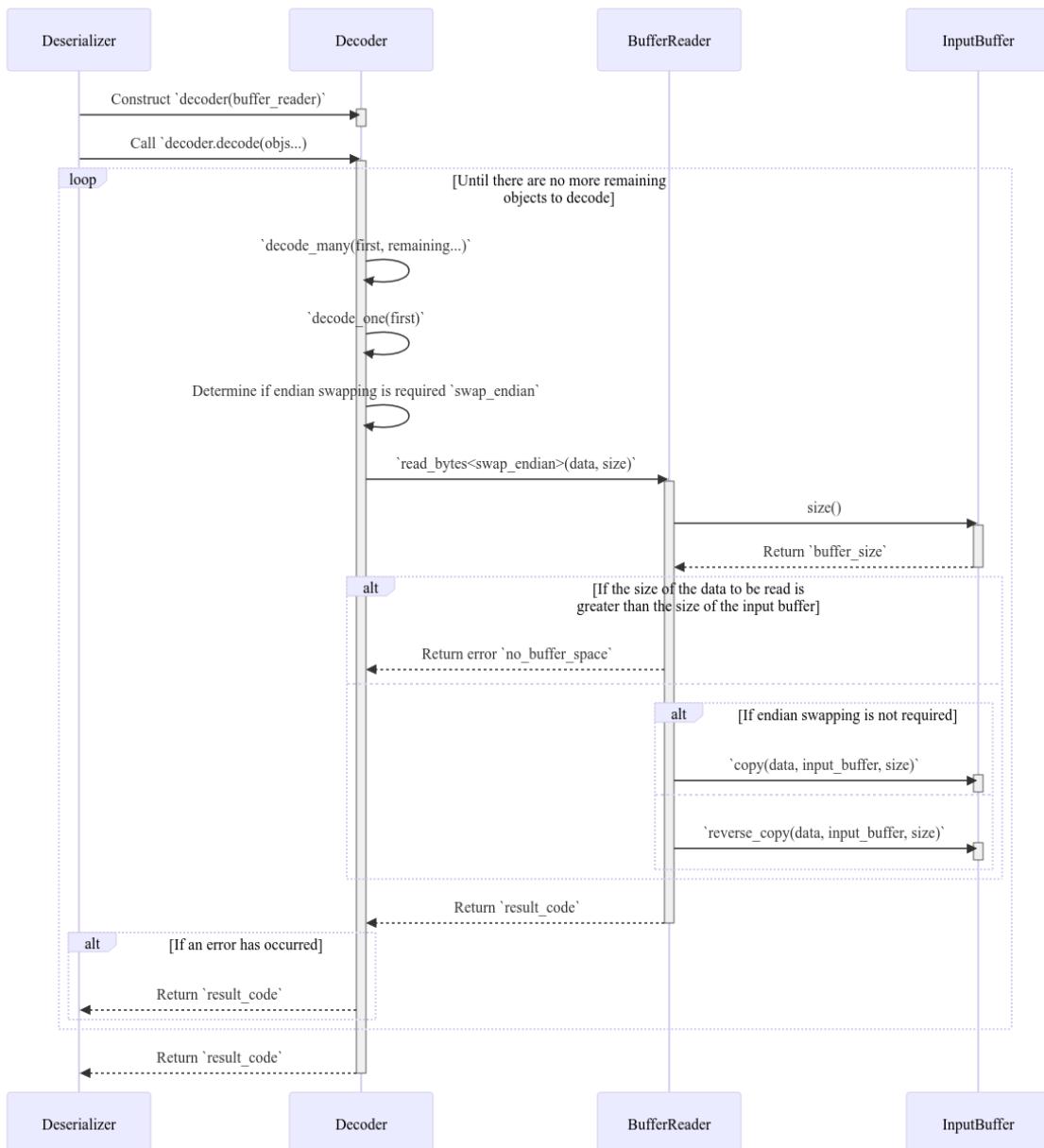


Figura 12. Diagrama de secuencia UML del proceso de decodificación binaria de objetos. Autoría propia.

El serializador y el deserializador se encargan de orquestar los codificadores y decodificadores necesarios para la creación e interpretación del archivo binario. Por otra parte, los codificadores y decodificadores interactúan con interfaces de lectura y escritura que se comunican con un medio de datos.

Tipos de datos serializables.

Siguiendo la especificación de formato de archivo binario se determinó un conjunto de tipos de datos a los que se les da soporte directo de serialización utilizando clases estándar del lenguaje C++ y los mecanismos de reflexión desarrollados en la investigación. A través del Concepto `teg::concepts::serializable` (véase Apéndice C.1) se soportan los siguientes tipos: (a) tipos built-in, tipos fundamentales y enumeraciones, que son codificados a números enteros y números de coma flotante; (b) tipos agregados, cuyas variables miembros deben ser también tipos serializables; (c) contenedores, estáticos o dinámicos, de elementos serializables; (d) objetos opcionales de tipos serializables; (e) tuplas, con elementos serializables; (f) punteros propietarios, no nulos y que apunten a un tipo serializable; (g) variantes, en las que todas las alternativas son tipos serializables; (h) tipos compatibles, que cuentan con una secuencia de versión válida y que contienen un tipo serializable; (i) y, finalmente, los tipos con codificación definida por el usuario.

Esquema de datos.

Otras librerías de serialización binaria suelen emplear métodos complejos para definir esquemas de datos, como archivos externos con lenguajes específicos, macros de preprocesador o funciones de conversión manuales. Por el contrario, nuestra solución utiliza la reflexión en tiempo de compilación, lo que simplifica significativamente el proceso. Con este enfoque, un esquema de datos válido se obtiene simplemente mediante la composición de uno o varios tipos que modelen el Concepto `teg::concepts::serializable`, en donde cualquier tipo `T` que cumpla con dicho Concepto actúa como esquema de datos válido.

Para garantizar la integridad del formato de datos, se necesita comparar el esquema de serialización con el esquema de deserialización, asegurando que ambos sean idénticos. Como mecanismo de verificación, el esquema de datos se convierte en un formato de texto según las especificaciones de codificación del archivo binario, y posteriormente se genera un hash utilizando el algoritmo MD5.

La codificación del esquema de datos y la obtención de su hash es un proceso que se hace en tiempo de compilación, como se muestra en la Figura 13. Para lograr

esto se utiliza una cadena de caracteres en tiempo de compilación creada específicamente para la librería (véase Apéndice C.5). Por otro lado, el algoritmo de digestión MD5 también se ejecuta en tiempo de compilación, donde también se provee una interfaz para obtener los primeros 32, 64 y 128 bits según se necesite (consulte el Apéndice B.2.4).

```

1 class User {
2     public:
3         uint64_t id;
4         std::string username;
5         std::string email;
6         uint64_t created_at;
7         uint64_t modified_at;
8         std::optional<std::array<uint8_t, 6>> sms_code;
9     };
10 // schema type is deducted as teg::fixed_string<31>
11 constexpr auto schema = teg::schema<1, User>();
12 print("Schema: ", schema);
13
14 constexpr teg::fixed_string<32> hash = teg::md5::hash(schema);
15 print("Hash:    ", hash);

output
Schema: {u64 [u8] [u8] u64 u64 ?#6[u8]}
Hash:   3d09f1ec594b90e3c3d18761ef6092b3

```

Figura 13. Demostración de la codificación del esquema de datos a un formato de texto y su digestión MD5. Autoría propia.

Si un esquema está compuesto con objetos compatibles entonces posee más de una versión, y cada versión se codifica por separado. El serializador binario se encarga de insertar la tabla de los hashes de todas las versiones del esquema de datos en la cabecera del archivo binario, para que posteriormente el decodificador obtenga esta tabla y valide si el esquema que se intenta deserializar corresponde correctamente. Esta comparación si ocurre en tiempo de ejecución y previene la decodificación incorrecta de los objetos.

Procesos de codificación.

Los procesos de codificación corresponden con la transformación atómica de un objeto en memoria hacia una secuencia de bytes siguiendo la especificación de

formato de archivo binario. Los codificadores y decodificadores binarios fueron diseñados para proporcionar una capa de abstracción entre el proceso de serialización y los detalles de implementación de la lectura y escritura de datos (véase Figura 14) con el patrón de diseño *Strategy* de Gamma et al (1994). Este diseño aplica el Principio de Inversión de Dependencias (Martin, 2017) permitiendo la separación de responsabilidades y facilitando la intercomunicación entre los procesos, asegurando la modularidad, reusabilidad y mantenibilidad del código.

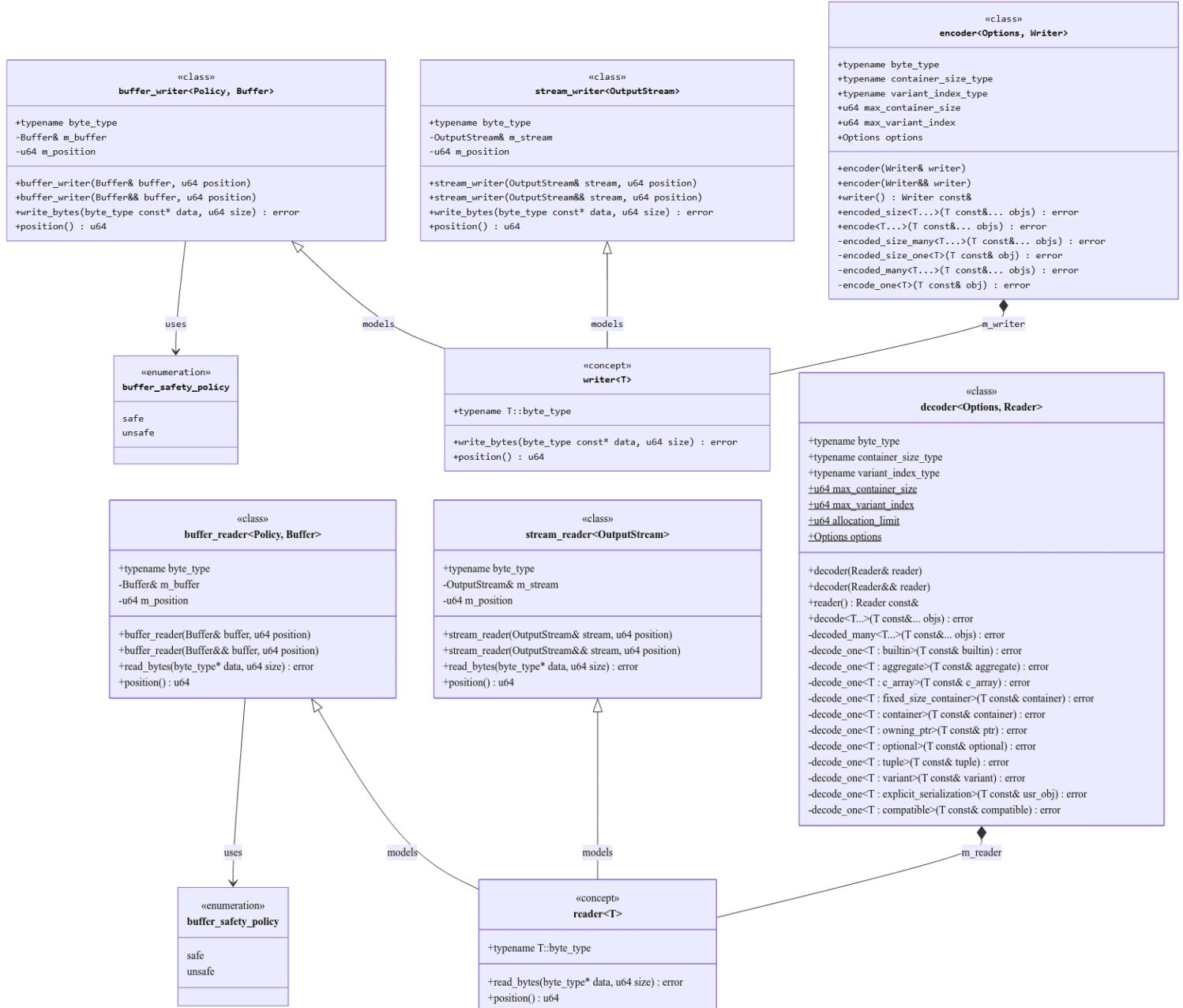


Figura 14. Diagrama de clases UML de los codificadores y decodificadores binarios y las interfaces de lectura y escritura implementadas en la librería “Teg”. Autoría propia.

Interfaz de lectura y escritura de datos.

El usuario puede definir y utilizar su propia clase de lectura y escritura de datos modelando el Concepto `teg::concepts::writer` y `teg::concepts::reader`, implementando las funciones `read_bytes` y `write_bytes`. Estas interfaces representan las estrategias del patrón Strategy.

En la Figura 15 se ilustra el funcionamiento de un escritor binario personalizado. La función `hex_print` se encarga de imprimir cada byte de información en un formato hexadecimal en el terminal de salida. En este ejemplo, se codifica una cadena de texto, y la secuencia de bytes resultante consiste en un número de 32 bits que representa el tamaño del texto, seguido de los caracteres de 8 bits de la cadena. Nótese que se utilizó exclusivamente un codificador, por lo que no se incluye ninguna cabecera de archivo, sólo una secuencia de bytes.

```

1 class custom_writer {
2 public:
3     using byte_type = char;
4
5     template <bool SwapEndian = false>
6     auto write_bytes(byte_type const* data, teg::usize size) -> teg::error {
7         for (auto i = 0; i < size; i++) {
8             hex_print(data[i]);
9         }
10        m_position += (teg::u64) size;
11        return {};
12    }
13    auto position() -> teg::u64 { return m_position; }
14 private:
15     teg::u64 m_position;
16 };
17
18 auto encoder = teg::encoder<teg::default_mode, custom_writer>(custom_writer());
19 encoder.encode(std::string{ "Hola Mundo!" }).or_throw();

output
b 0 0 0 48 6f 6c 61 20 4d 75 6e 64 6f 21 // u32: 11, u8[11]: Hola Mundo!

```

Figura 15. Ejemplo de definición de una clase de escritura personalizada para utilizar la interfaz de codificación binaria. Autoría propia.

Para la primera versión de la librería, se implementaron dos pares de clases de lectores y escritores binarios, correspondientes al manejo de arreglos de bytes (redimensionables o fijos) y manejo de flujo de datos respectivamente (véase Apéndice B.2.1). Estas clases habilitan las operaciones de entrada y salida de datos hacia regiones de memoria, la consola de comandos, constructores de cadenas de texto y archivos en memoria secundaria (ej. disco duro).

Ordenamiento de bytes.

Las clases de codificadores se parametrizan en tiempo de compilación (a través de plantillas) con las opciones de formato de archivo, en las cuales se encuentra el orden de bytes establecido por el usuario. Sin embargo, la arquitectura objetivo puede tener un orden de bytes diferente. En esos casos, el codificador emplea algoritmos que cambian el orden de los bytes en los tipos de datos necesarios. La selección de estos algoritmos ocurre en el momento de la compilación, y el cambio de orden se efectúa en tiempo de ejecución. Al hacer esto, el codificador no desperdicia instrucciones verificando si debe realizar el cambio, ya que esta decisión se ha tomado previamente.

Existen tipos de datos que se consideran neutrales con respecto al orden de bytes, estos son aquellos tipos que ocupan un único byte en memoria (véase Apéndice C.6). Ya que siempre se mantiene el mismo orden de bit, los tipos de un solo byte no requieren cambio alguno. A su vez, arreglos y contenedores contiguos que contengan estos tipos como elementos también son neutrales con respecto al ordenamiento de bytes. Poder identificar estos tipos ayuda a seleccionar algoritmos más eficientes para los procesos de codificación y decodificación.

Codificación definida por el usuario.

En algunos casos el usuario puede requerir la serialización de objetos que no son tipos agregados. Para abordar esta necesidad, hemos diseñado un mecanismo de codificación explícita. Dicho mecanismo funciona con polimorfismo estático, lo que implica que el usuario debe definir funciones

libres de codificación y decodificación que modelen el concepto `teg::concepts::user_defined_encoding` (véase Apéndice C.2.7).

La Figura 16 muestra la codificación explícita de una clase `vec3` que no es agregada, en este caso pierde esta propiedad porque tiene variables miembros privadas. Los mecanismos de reflexión desarrollados en la investigación solo pueden acceder a tipos agregados, y, dada esta limitación, el usuario debe definir explícitamente cómo se codifican y decodifican las clases no agregadas. El codificador binario pasa como argumento una función lambda que sirve para requerir la transformación de tipos serializables u otros tipos con codificación explícita a su respectiva secuencia de bytes, funcionando de manera recursiva y con la misma especificación de formato.

```

1 class Vector3 {
2 public:
3     constexpr Vector3() : x(0), y(0), z(0) {}
4     constexpr Vector3(uint32_t x, uint32_t y, uint32_t z) : x(x), y(y), z(z) {}
5
6     constexpr uint32_t get_x() const { return x; }
7     constexpr uint32_t get_y() const { return y; }
8     constexpr uint32_t get_z() const { return z; }
9
10    constexpr void set_x(uint32_t x) { this->x = x; }
11    constexpr void set_y(uint32_t y) { this->y = y; }
12    constexpr void set_z(uint32_t z) { this->z = z; }
13 private:
14     uint32_t x, y, z;
15 };
16
17 template <class F>
18 constexpr auto usr_encode(F&& encode, Vector3 const& vec) -> teg::u64 {
19     return encode(vec.get_x(), vec.get_y(), vec.get_z());
20 }
```

```

20 template <class F>
21 constexpr auto usr_decode(F&& decode, Vector3& vec) -> teg::error {
22     std::tuple<uint32_t, uint32_t, uint32_t> values;
23
24     if (auto const result = decode(values); teg::failure(result)) {
25         return result;
26     }
27     vec.set_x(std::get<0>(values));
28     vec.set_y(std::get<1>(values));
29     vec.set_z(std::get<2>(values));
30     return {};
31 }
32
33 teg::byte_array buffer;
34 teg::serialize(buffer, Vector3(1, 2, 3)).or_throw(); // Ok: user explicit encode

```

Figura 16. Ejemplo de codificación explícita definida por el usuario en una clase que no es agregada. Autoría propia.

Manejo de errores

Durante el tiempo de ejecución, los procesos de serialización y deserialización pueden fallar por diversos motivos, tales como espacio insuficiente en el medio de almacenamiento, corrupción del archivo binario, opciones de formato inválidas, o incompatibilidad del esquema de datos, entre otros. Para abordar estas situaciones, hemos desarrollado la librería con dos mecanismos de manejo de errores: mediante códigos de error y mediante excepciones. Dado que en algunos sistemas el soporte para excepciones puede estar desactivado, siempre se mantiene la opción de utilizar códigos de error. La Figura 17 presenta un ejemplo de ambos casos de uso.

```

1 teg::fixed_byte_array<1> buffer; // A buffer with just 1 byte of space
2 auto const result = teg::serialize(buffer, "A text (that occupies >1B)")
3
4 if (teg::success(result)) {
5     // Do something with the buffer ... (e.g. send it over the network)
6 }
7 else {
8     // Handle the error ...
9 }

```

```

1 try {
2     teg::fixed_byte_array<1> buffer; // A buffer with just 1 byte of space
3     teg::serialize(buffer, "Just an ordinary string").or_throw();
4 }
5 catch(std::exception e) {
6     print(e.what()); // std::system_error: no buffer space
7 }
```

Figura 17. Manejo de errores en la librería. El recuadro superior ilustra el uso de códigos de error, mientras que el recuadro inferior demuestra el uso de excepciones. Autoría propia.

Modos de serialización

El formato de archivo binario Teg admite múltiples opciones que determinan cómo se serializan y deserializan los datos. Estas opciones se definen estáticamente mediante parámetros de plantillas al invocar las funciones de la librería. Para simplificar este proceso, se han creado dos modos preconfigurados que los usuarios pueden utilizar sin necesidad de ajustar los detalles manualmente: el modo por defecto `teg::default_mode` y el modo compacto `teg::compact_mode`. El modo por defecto establece las opciones de formato recomendadas: orden de bytes little-endian, y capacidad de hasta 2^{32} elementos en un mismo contenedor, capacidad de hasta 2^8 alternativas en una misma variante, y alojamiento máximo de 2 GiB. Por otra parte, el modo compacto, está diseñado para su uso en situaciones donde se requieren tamaños de archivo más reducidos; en este modo se fuerza la conversión de todos los números enteros hacia números de longitud variable, se mantiene el orden de bytes little-endian y el alojamiento máximo permitido es de 1 GiB.

Retrocompatibilidad

Siguiendo la especificación de formato de archivo binario, se implementó un mecanismo de retrocompatibilidad a través de la clase `teg::compatible` (véase Apéndice C.1.6). Esta clase consta de dos parámetros genéricos: un tipo de dato `T` y un número de versión de esquema `v`. El tipo de dato `T` es envuelto y codificado como un tipo opcional, mientras que el número de versión `v` se utiliza para asegurar la integridad del esquema.

La Figura 18 muestra un ejemplo de retrocompatibilidad entre dos versiones de un mismo esquema de datos. En este caso, un servidor hace una actualización del

esquema `image_data` para incorporar un texto alternativo a las imágenes enviadas. Sin embargo, el cliente aún no recibe ninguna actualización y se queda con una versión anterior de este esquema. En estas circunstancias el cliente consumirá la secuencia de bytes como si fuese la primera versión, ignorando por completo los campos agregados en versiones posteriores.

```

1 // Server side
2 struct ImageDataV2 {
3     std::string id;
4     std::string content_base64;
5     teg::compatible<std::string, 2> alt_text;
6 };
7
8 ImageDataV2 img_v2 = { "header_banner", "AABCCDD...", "Company logo" };
9 teg::byte_array data{};
10 teg::serialize(data, img_v2).or_throw();
11 send_to_client(data);

1 // Client side
2 struct ImageDataV1 {
3     std::string id;
4     std::string content_base64;
5 };
6
7 ImageDataV1 img_v1;
8 teg::byte_array data = receive_from_server();
9 teg::deserialize(data, img_v1).or_throw();

```

Figura 18. Retrocompatibilidad entre dos versiones de esquemas de datos. Autoría propia.

Serialización en tiempo de compilación.

Todas las funciones de la librería, tanto de codificación como de serialización, han sido diseñadas para aprovechar la evaluación de contextos constantes en C++. En nuestra investigación sobre la reflexión en estática hallamos que, bajo ciertas condiciones, es posible ejecutar los procesos de serialización y deserialización en tiempo de compilación.

Para poder realizar esto, hace falta que dichos procesos sean invocados en un contexto constante, lo que restringe los posibles tipos de datos tanto de los medios como de los objetos a procesar. Por ejemplo, es posible serializar tipos fundamentales, tipos trivialmente copiables y de estructura estándar, y tuplas y arreglos de tamaño

fijo que contengan estos tipos. A su vez, sólo podemos utilizar buffers de tamaño fijo, lo que significa que se debe preestablecer un tamaño de bytes en memoria lo suficientemente grande para contener los objetos en su formato serializado. En la Figura 19 se puede observar la serialización en tiempo de compilación de un número entero.

```

1 constexpr teg::fixed_byte_array<17> serialize_number(teg::i32 number) {
2     teg::fixed_byte_array<17> buffer{};
3     teg::serialize(buffer, number).or_throw();
4     return buffer;
5 }
6
7 constexpr auto buffer = serialize_number(99);
8 // Assembly msvc-x64 generated for line 7.
9     mov      byte ptr [buffer],54h
10    mov      byte ptr [rsp+29h],45h
11    mov      byte ptr [rsp+2Ah],47h
12    mov      byte ptr [rsp+2Bh],1
13    mov      byte ptr [rsp+2Ch],22h
14    mov      byte ptr [rsp+2Dh],90h
15    mov      byte ptr [rsp+2Eh],0
16    mov      byte ptr [rsp+2Fh],0
17    mov      byte ptr [rsp+30h],1
18    mov      byte ptr [rsp+31h],52h
19    mov      byte ptr [rsp+32h],0DBh
20    mov      byte ptr [rsp+33h],9Eh
21    mov      byte ptr [rsp+34h],0F8h
22    mov      byte ptr [rsp+35h],63h
23    mov      byte ptr [rsp+36h],0
24    mov      byte ptr [rsp+37h],0
25    mov      byte ptr [rsp+38h],0

```

Figura 19. Instrucciones de lenguaje ensamblador generadas en la serialización en tiempo de compilación de un número entero fijo de 32-bits. Autoría propia.

Internamente, el compilador evaluará y ejecutará las funciones de serialización y deserialización en tiempo de compilación y solo generará las instrucciones necesarias para alojar el resultado en memoria. Como se muestra en la Figura 19, se serializa un número entero de 32-bits pero no ocurre ninguna llamada a ninguna función de serialización, sino que el buffer resultante de la serialización es embebido directamente en el archivo binario ejecutable; las 17 instrucciones `mov` corresponden con los 17 bytes que ocupa el número entero en su estado serializado, tomando en

cuenta que la cabecera ocupa 13 de estos 17 bytes, y los últimos 4 bytes corresponden al número 99 en orden little-endian. De manera similar, si se tiene un buffer de datos válido correspondiente a la serialización de un objeto, es posible deserializar dicho buffer hacia un objeto en tiempo de compilación. El resultado es similar al mostrado en la Figura 19, pero en este caso las instrucciones de lenguaje ensamblador generadas serán para inicializar el objeto con sus datos correspondientes.

Módulo de Enteros de Longitud Variable

El Módulo de Enteros de Longitud Variable constituye la tercera iteración del producto de software de la presente investigación. Se desarrolló con el objetivo de investigar y aplicar técnicas para la codificación de enteros de ancho de bit variable como mecanismo para reducir el tamaño de los archivos binarios cuando sea posible.

Codificaciones de enteros de longitud variable.

Lemire, Kurz, y Rupp (2018) recopilaron y compararon las codificaciones listadas en la Tabla 5, y hallaron que las técnicas de codificación optimizadas con SIMD²⁴ generalmente ofrecen una mayor tasa de procesamiento debido a la capacidad de operar sobre múltiples datos en paralelo. Sin embargo, estas técnicas necesitan del soporte del repositorio de instrucciones de microprocesador SSSE3²⁵, que solo están disponibles en procesadores fabricados alrededor del año 2010 en adelante.

Tabla 5

Descripción general de las técnicas de compresión de enteros orientadas a bytes. Tomado de “STREAM VBYTE: Faster Byte-Oriented Integer Compression” (p. 4) de Lemire, Kurz, y Rupp (2018).

Nombre del codificador	Formato de datos	SIMD
VByte (LEB-128)	7 bits de datos por byte, 1 bit como bandera de continuación.	No
Masked VByte	Idéntico a VByte.	Sí
Varint-GB	Número fijo de enteros (4) comprimidos a un número variable de bytes (4-16), precedido por un byte de control.	No
Varint-G8IU	Número fijo de bytes comprimidos (8) para un número variable de enteros (2-8), precedido por un byte de control.	Sí
Stream VByte	Bytes de control y bytes de datos en flujos separados.	Sí

Lemire et al. (2018) apuntan que VByte, también conocida como LEB-128, es una de las técnicas más utilizadas por su facilidad de implementación, y, aunque en sus experimentos reporta tener una tasa de procesamiento inferior a las demás codificaciones, también es la que ocupa un menor espacio en promedio. Según Creager (2021) Protobuf utiliza una codificación muy similar a LEB-128, conocida

²⁴ SIMD (*Single Instruction, Multiple Data*) es un tipo de procesamiento en paralelo que permite la ejecución simultánea de una sola instrucción de microprocesador sobre múltiples datos.

²⁵ SSSE3 (Supplemental Streaming SIMD Extensions 3) es un repositorio de extensión de instrucciones de microprocesador que permite la ejecución de operaciones SIMD.

como VLQ, pero con la diferencia de que sigue un orden de bytes big-endian. Formalmente tanto VLQ como LEB-128 utilizan como bit de control un bit activo (bit establecido en 1), pero algunas implementaciones optan por utilizar un bit nulo (bit establecido en 0).

La Figura 20 muestra los algoritmos de codificación y decodificación ULEB-128, versión para enteros sin signo. Se seleccionó esta codificación para trabajar con enteros de longitud variable en la librería de serialización binaria por las siguientes razones: (a) aunque otras codificaciones recopiladas por Lemire et al. (2018) muestran una mayor velocidad de procesamiento, están especializadas para ser utilizadas con listas y grupos de números, necesitan código vectorizado e instrucciones SIMD, y/o ocupan un mayor tamaño al incorporar múltiples bytes de

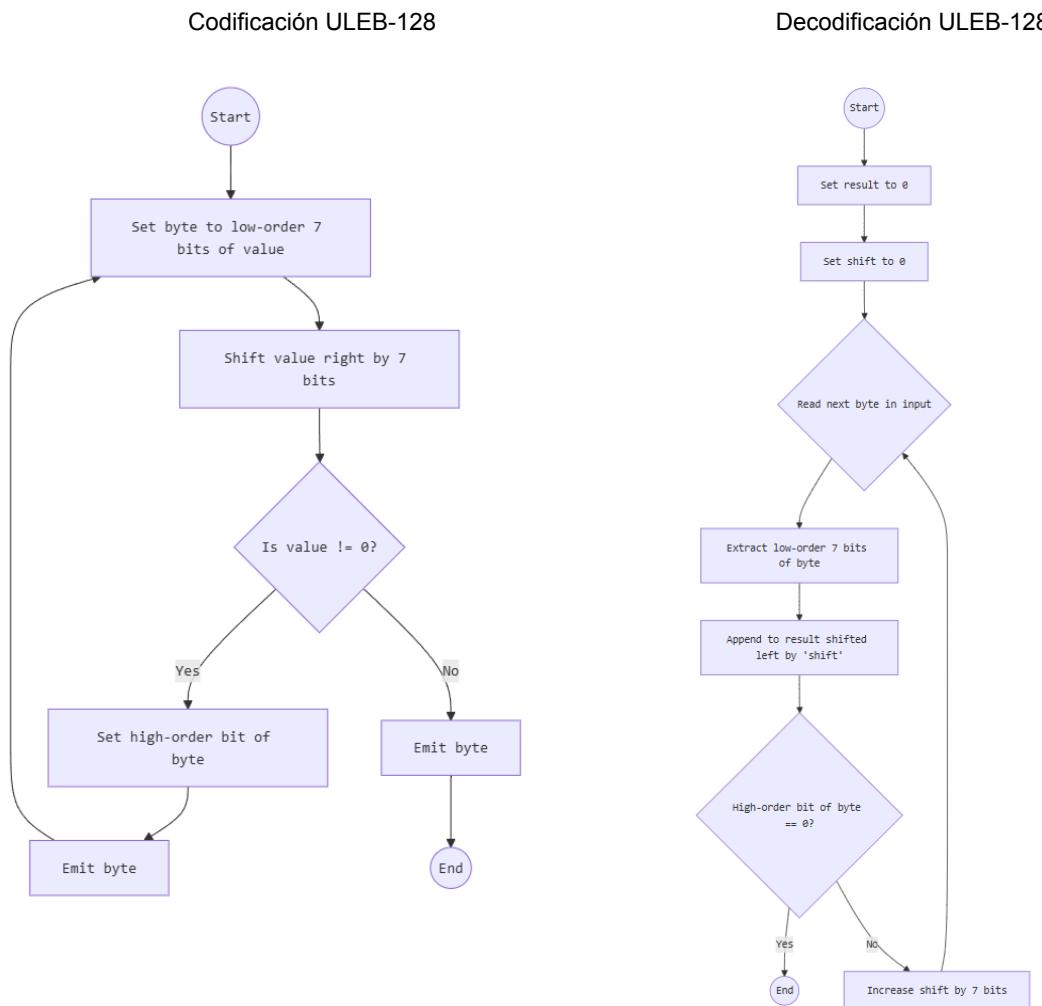


Figura 20. Diagrama de flujo del algoritmo de codificación y decodificación de números enteros sin signos ULEB-128. Autoría propia.

control; (b) la codificación ULEB-128 resulta fácil de implementar y ofrece un buen rendimiento ya que emplea operaciones de bits sencillas; (c) es ampliamente utilizada en formatos de archivos binarios como WebAssembly²⁶ y DWARF²⁷; y (d) no se necesita de instrucciones SIMD, y este algoritmo puede ejecutarse en cualquier procesador antiguo o moderno.

Clase genérica `teg::varint`.

Como resultado del análisis de compresiones de números enteros, se seleccionó la codificación LEB-128, más específicamente su versión para números sin signo ULEB-128 (por sus siglas en inglés, *Unsigned Little-Endian Base 128*). Sin embargo, se buscó codificar números con o sin signo por igual. Para lograr esto, los números con signo son procesados una primera vez con la codificación ZigZag para luego ser codificados con ULEB-128.

La Figura 21 ilustra un ejemplo del espacio ocupado en memoria de un número entero con signo. Los números negativos suelen tener un alto número de bits establecidos en "1" debido a la representación en complemento a dos. Sin embargo, tras aplicar la codificación ZigZag estos bits se vuelven nulos. Posteriormente, en este caso, con la codificación ULEB-128 se consigue convertir un número de 8 bytes en otro de tan solo 2 bytes, representando eficientemente el mismo número y descartando aquellos bits que no poseen peso en la representación binaria.

```

1 int64_t n = -200;
2 // Memory object representation of `n` at line 1.
3 // 11111111 11111111 11111111 11111111 11111111 11111111 11111111 00111000
4 uint64_t z = teg::zigzag::encode(n);
5 // Memory object representation of `z` at line 4.
6 // 00000000 00000000 00000000 00000000 00000000 00000000 00000001 10001111
7 std::vector<uint8_t> buffer {};
8 buffer.resize(teg::uleb128::encoded_size(z))
9 teg::uleb128::encode(buffer, z);
10 // Memory of vector `buffer` at line 9.
11 // 0: 10001111 1: 00000011

```

Figura 21. Ejemplo de la codificación ZigZag y codificación ULEB-128 en un número entero con signo de 64 bits. Autoría propia.

²⁶ LEB-128 en WebAssembly: <https://webassembly.github.io/spec/core/binary/values.html#integers>

²⁷ LEB-128 en DWARF: <https://dwarfstd.org/doc/Dwarf3.pdf>

La clase genérica `teg::varint` está diseñada como un envoltorio de un número entero `T` (véase Apéndice C.4). Esta clase ofrece una interfaz transparente para operar con números enteros, permitiendo además la serialización de estos números de acuerdo con la codificación ULEB-128. A nivel de memoria, un número de longitud variable seguirá teniendo un ancho de bits fijo; es en su estado serializado donde sí tendrá una representación de ancho de bits variable.

La serialización hacia el formato de archivo binario de la clase `teg::varint` está implementada de manera explícita a través del mecanismo de codificación definida por el usuario. En la especificación de archivo binario portable no se introdujo ninguna codificación especial para números enteros de longitud variable, y siguiendo este diseño, tampoco se implementó ninguna función específica de codificación en el módulo de serialización para estos tipos de datos. Esto es así debido a que hallamos que la codificación explícita de tipos permite extender el módulo de serialización de objetos sin necesidad de que este sea modificado.

Internamente, la codificación de esta clase consiste en una secuencia variable de bytes, tratados en el archivo binario como enteros sin signo de 8 bits (véase Apéndice C.4). Teóricamente, el número de secciones (bytes) de un entero de longitud variable puede ser tan grande como bytes pueda contener el archivo binario, representando así un entero de cualquier tamaño. Sin embargo, la clase desarrollada en la librería para manipular este tipo de números sólo permite alojar hasta 8 bytes en memoria. En base a esto, en la práctica, una secuencia de bytes de un número de longitud variable en su formato serializado ocupará como máximo 10 bytes, que es el número de bytes necesarios para representar el máximo entero de 64 bits sin signo $2^{64} - 1$.

Pruebas de Software

Herramientas de *testing*.

Con la finalidad de realizar las pruebas de software se desarrolló una librería de testing con un diseño de dependencia mínima y con la inclusión de herramientas de testeo no solo en tiempo de ejecución sino también en tiempo de compilación. Hubo tres razones principales por las cuales se tomó la decisión de desarrollar una librería de testing en lugar de incorporar una librería de terceros:

1. El sistema de construcción usado en la librería, Meson²⁸, permite clasificar y manejar objetivos ejecutables como un conjunto de prueba, más no incluye ninguna librería de testeo. Al utilizar librerías de terceros con este sistema se suele crear confusión en los comandos de consola necesarios para ejecutar las pruebas.
2. Las pruebas unitarias deben cubrir aserciones en tiempo de compilación, una característica que no es común en librerías de terceros.
3. Al no depender de código externo, se consigue tener una herramienta de testing especializada y mínima que cubre las necesidades de prueba de software del proyecto.

Entorno de pruebas (*fixtures*).

La librería cuenta con una interfaz genérica para crear entornos de prueba (conocidos como *fixtures*, en inglés). Estos entornos permiten al usuario, de así necesitarlo, configurar el estado inicial de las pruebas a ejecutar. Estos entornos cuentan con un proceso de inicialización y terminación que es transparente para las pruebas, aportando las preparaciones necesarias como la creación de objetos, la carga de archivos, la configuración de los sistemas a probar, entre otros.

Casos de prueba y aserciones.

Para aumentar la facilidad de uso de la librería de testing, se desarrolló un conjunto de macros que automatizan la creación y agrupación de casos de prueba, permiten separar el código a probar por secciones y habilitan el uso de aserciones. Cada caso de prueba tiene asignado un entorno de pruebas, definido implícitamente por la librería o explícitamente por el usuario, y en estos se ejecutan y se reportan las aserciones mediante los macros de aserciones, como se muestra en la Figura 22.

²⁸ Sistema de construcción Meson: <https://mesonbuild.com/>

```

1 #define _ASSERT_BINARY(source, a, b, op) \
2     do { \
3         const auto& val_a = a; \
4         const auto& val_b = b; \
5         auto current_test = test::test_fixture::current_test(); \
6         current_test->eval((val_a) op (val_b), source, __FILE__, __LINE__, \
7             std::format("{} {} {}", val_a, #op, val_b).c_str()); \
8     } while (false); \
9 \
10 #define _ASSERT_UNARY(source, cond) \
11     do { \
12         auto current_test = test::test_fixture::current_test(); \
13         current_test->eval(cond, source, __FILE__, __LINE__, \
14             std::format("{} {}", cond).c_str()); \
15     } while (false); \
16 \
17 #define ASSERT_EQ(a, b) _ASSERT_BINARY("ASSERT_EQ(\"#a\", \"#b\")", a, b, ==) \
18 #define ASSERT_NE(a, b) _ASSERT_BINARY("ASSERT_NE(\"#a\", \"#b\")", a, b, !=) \
19 #define ASSERT_LT(a, b) _ASSERT_BINARY("ASSERT_LT(\"#a\", \"#b\")", a, b, <) \
20 #define ASSERT_LE(a, b) _ASSERT_BINARY("ASSERT_LE(\"#a\", \"#b\")", a, b, <=) \
21 #define ASSERT_GT(a, b) _ASSERT_BINARY("ASSERT_GT(\"#a\", \"#b\")", a, b, >) \
22 #define ASSERT_GE(a, b) _ASSERT_BINARY("ASSERT_GE(\"#a\", \"#b\")", a, b, >=) \
23 #define ASSERT(cond) _ASSERT_UNARY("ASSERT(\"#cond\")", cond)

```

Figura 22. Macros de aserciones definidos en la librería de testing desarrollada para el proyecto de investigación. Autoría propia.

También es posible realizar aserciones y reportar los resultados en tiempo de compilación, a través de los macros mostrados en la Figura 23. Esto permite ampliar la cobertura de las pruebas, probando código que solo se ejecuta en tiempo de compilación o que puede ser ejecutado en ambos contextos.

```

1 TEST_CASE("MD5 digests should be computed at compile-time") { \
2     constexpr auto msg = teg::make_fixed_string("It's over Anakin, I have the \
3         high ground"); \
4     constexpr auto hash = teg::md5::hash(msg); \
5     COMPTIME_ASSERT_EQ(hash, "d3f8915ed09820841add8cdb39387700"); \
6 }

```

Figura 23. Ejemplo de aserciones en tiempo de compilación, herramienta especializada de la librería de testing del proyecto. Autoría propia.

Diseño, construcción y ejecución de las pruebas de software.

Para asegurar el correcto funcionamiento de la librería de serialización binaria se diseñaron y construyeron un conjunto de pruebas unitarias y pruebas de integración. La Tabla 6 describe cada una de las pruebas desarrolladas. Estas pruebas fueron diseñadas considerando los siguientes factores:

1. Cobertura de código: se buscó tener la mayor cobertura de código posible. A veces resulta imposible tener una cobertura de toda la base de código fuente, por esto, se optó por cubrir todas las interfaces de programación expuestas por la librería, cerciorando tanto los casos comunes como los casos borde.
2. Ejecución y depuración: El conjunto de pruebas unitarias y de integración fue construido de tal forma que cada prueba se puede ejecutar de manera independiente. Esto permite utilizar herramientas de depuración y profundizar en la ejecución de las líneas de código donde fallan las aserciones. La librería y las pruebas son compiladas con los símbolos de depuración, por lo que se tiene acceso a los detalles de implementación de la librería, pudiendo inspeccionar de forma secuencial la ejecución de cada línea de código.
3. Automatización: Cada que el código fuente sea modificado y se guarden los cambios en el repositorio de código, se debe poder ejecutar todo el conjunto de pruebas y reportar los resultados.

Tabla 6

Repertorio de pruebas unitarias y de integración de la librería “Teg”. Autoría propia.

Prueba (nombre técnico)	Módulo	Descripción
test_aggregates	serialization	Prueba de serialización y deserialización de tipos de datos agregados (composición de todos los tipos serializables).
test_alignment	reflection	Pruebas de identificación de tipos de estructuras empaquetadas con y sin requerimientos explícitos de alineación de memoria. Aserciones de las propiedades de estos tipos de datos.
test_buffer	serialization	Pruebas de arreglos dinámicos y estáticos de datos y sus limitaciones de alojamiento en memoria. Aserción de casos de error de lectura o escritura en buffers sin espacio.
test_c_array	reflection, serialization	Aserción de Conceptos de arreglos de estilo C. Pruebas de serialización de arreglos unidimensionales y multidimensionales con estructuras planas y clases agregadas.
test_compatible	reflection, serialization	Pruebas de retrocompatibilidad. Aserción de casos de error de uso de clases compatibles.
test_comptime_serialization	serialization	Pruebas de serialización y deserialización de objetos en

<code>test_concepts</code>	<code>reflection</code>	tiempo de compilación.
<code>test_container</code>	<code>reflection, serialization</code>	Pruebas de modelado de Conceptos generales.
<code>test_endianness</code>	<code>serialization</code>	Pruebas de reflexión de tipos de datos que modelan los Conceptos de contenedor. Serialización y deserialización básica de contenedores.
<code>test_fixed_string</code>	<code>serialization</code>	Pruebas de serialización y deserialización convirtiendo entre ordenamiento de bytes little-endian y big-endian. Aserción de errores en casos de incompatibilidad de formato.
<code>test_fundamental</code>	<code>serialization</code>	Pruebas de la interfaz de programación de cadenas de texto en tiempo de compilación (clase <code>`teg::basic_fixed_string`</code>). Pruebas de serialización y deserialización de esta misma clase.
<code>test_map</code>	<code>serialization</code>	Pruebas de serialización y deserialización de tipos de datos fundamentales (booleanos, números reales, números enteros, enumeraciones).
<code>test_md5</code>	<code>serialization</code>	Pruebas de serialización y deserialización de contenedores asociativos clave-valor (en específico <code>`std::map`</code> , <code>`std::unordered_map`</code> , <code>`std::multimap`</code> y <code>`std::unordered_multimap`</code>).
<code>test_optional</code>	<code>serialization</code>	Aserción de la implementación del algoritmo de digestión MD5 en tiempo de compilación.
<code>test_overload_resolution</code>	<code>reflection, serialization</code>	Aserción de funcionalidad de sublevación de Conceptos utilizados en los algoritmos de codificación.
<code>test_owning_ptr</code>	<code>serialization</code>	Pruebas de serialización y deserialización de punteros propietarios (específicamente <code>`std::shared_ptr`</code> y <code>`std::unique_ptr`</code>).
<code>test_reflection</code>	<code>reflection</code>	Pruebas de reflexión en tiempo de compilación de tipos agregados. Aserción de las limitaciones.
<code>test_schema</code>	<code>serialization</code>	Aserción de la codificación de esquemas y la creación de tablas de hashes de esquemas.
<code>test_set</code>	<code>serialization</code>	Prueba de serialización y deserialización de contenedores asociativos (en específico <code>`std::set`</code> , <code>`std::unordered_set`</code> , <code>`std::multiset`</code> y <code>`std::unordered_multiset`</code>).
<code>test_stream</code>	<code>serialization</code>	Pruebas de serialización y deserialización utilizando los flujos de datos de entrada y salida de la librería estándar (lectura y escritura de archivos en disco).
<code>test_string</code>	<code>serialization</code>	Pruebas de serialización y deserialización de cadenas de texto unicode (UTF-8, UTF-16 y UTF-32).
<code>test_tuple</code>	<code>serialization</code>	Pruebas de serialización y deserialización de tuplas y pares.
<code>test_user_serialization</code>	<code>reflection, serialization</code>	Pruebas de serialización y deserialización de codificación definida por el usuario (codificación explícita).
<code>test_variant</code>	<code>serialization</code>	Pruebas de serialización y deserialización de tipos variantes (específicamente <code>`std::variant`</code>).
<code>test_varint</code>	<code>serialization, varint</code>	Pruebas de la interfaz de programación y serialización de las clases <code>`teg::vint32`</code> , <code>`teg::vint64`</code> , <code>`teg::vuint32`</code> , y <code>`teg::vuint64`</code> .
<code>test_vector</code>	<code>serialization</code>	Pruebas de serialización y deserialización de vectores (arreglos dinámicos).

Evaluación de Rendimiento

Como última etapa del proyecto se diseñaron, construyeron y ejecutaron pruebas de rendimiento con el objetivo de evaluar la librería de serialización desarrollada en la presente investigación con respecto a otras librerías utilizadas en la industria del software.

Diseño de los experimentos.

Métricas de rendimiento.

Según Lilja (2004), una métrica de rendimiento es un valor derivado de medidas específicas que describen el desempeño de un sistema. Estas medidas pueden incluir el conteo de la frecuencia de eventos, la duración de intervalos de tiempo, y el tamaño de ciertos parámetros. Al normalizar estos valores a una base de tiempo común, como operaciones por segundo, obtenemos métricas de velocidad o rendimiento, útiles para comparar distintos sistemas o configuraciones. Para evaluar las librerías de serialización hemos seleccionado las siguientes métricas:

Tasa de procesamiento de datos. La tasa de procesamiento de datos se define como la proporción entre el tiempo promedio de ejecución empleado en el procesamiento de los datos y la cantidad de datos procesados, y se expresa en gigabytes por segundo (GiB/s). Esta métrica se calcula tanto para los procesos de serialización como para los de deserialización.

Tiempo total de serialización y deserialización. Se obtiene como resultado de sumar el tiempo de ejecución promedio de la serialización y el tiempo promedio de ejecución de la deserialización, expresados en milisegundos (ms). Al combinar ambas medidas, se consigue una métrica con la cual se puede observar el comportamiento completo del proceso, lo que permite evaluar el rendimiento general de la serialización y la deserialización.

Ratio de compresión. Expresado en porcentaje, es la relación entre el tamaño de los archivos binarios generados en los procesos de serialización en comparación al tamaño de los datos procesados.

Uso de la memoria principal. Se calcula el pico de uso de memoria promedio en los procesos de serialización y deserialización, en megabytes (MiB).

Uso del procesador. Uso mínimo y máximo del porcentaje de todos los núcleos de procesador.

Para las mediciones del tiempo de ejecución de los procesos se utiliza el tiempo de reloj de pared. Lilja (2004) explica existen dos tipos de mediciones del tiempo de ejecución de un programa: (a) el tiempo de reloj de pared (del inglés, *wall clock*) que mide el tiempo real transcurrido desde el inicio hasta la finalización de una tarea, incluyendo cualquier tiempo de espera; y (b) el tiempo CPU, que mide el tiempo que la CPU realmente dedica a ejecutar el código, excluyendo tiempos de inactividad. En sistemas operativos multitareas, el planificador de tareas se encarga de asignar un tiempo de ejecución a cada proceso, y durante la ejecución de una tarea esta puede ser detenida y reanudada a través de cambios de contexto.

Para medir el uso del procesador, se decidió utilizar la técnica de traza de eventos. Friedman (2012) explica que las dos técnicas más comunes son el muestreo y la traza de eventos. El muestreo captura el estado del sistema en intervalos fijos, proporcionando una visión general pero con posibles lagunas entre muestras. En cambio, la traza de eventos registra cada evento relevante en tiempo real, como cambios de contexto e interrupciones, ofreciendo una precisión mucho mayor al capturar todos los eventos que afectan el uso del CPU.

Herramientas de benchmarking y profiling.

Para tomar las medidas necesarias y obtener las métricas de rendimiento se seleccionó y se utilizó el siguiente conjunto de herramientas de software de *benchmarking* y *profiling*:

Google Benchmark²⁹. Es una librería de micro-benchmarking escrita en C++ que permite medir el rendimiento de fragmentos de código.

²⁹ Documentación oficial de Google Benchmark: <https://github.com/google/benchmark>

Está diseñada para funcionar tanto en aplicaciones de un solo hilo como en aplicaciones multihilo y tiene la capacidad de medir tanto el tiempo de reloj de pared como el tiempo CPU.

Windows Performance Recorder. Forma parte del Windows Performance Toolkit, Windows Performance Recorder³⁰ (WPR) es una herramienta que graba eventos del sistema y de aplicaciones para su posterior análisis. Utiliza el rastreo de eventos para Windows (ETW) y permite identificar áreas específicas de rendimiento y comprender el consumo de recursos de todo el sistema.

Windows Performance Analyzer. También forma parte del Windows Performance Toolkit, Windows Performance Analyzer (WPA)³¹ se utiliza para analizar y visualizar los datos grabados por WPR. Esta herramienta permite crear perfiles de análisis y exportar los datos que sean requeridos para una evaluación de rendimiento, como el uso de la memoria principal y el uso del CPU.

Experimentos de evaluación de rendimiento.

Con el fin de evaluar las librerías de serialización binaria se diseñaron y construyeron dos experimentos:

Experimento 1: Query e-commerce. El primer experimento consiste en la serialización y la deserialización del resultado de una búsqueda de productos en una página de comercio electrónico. En la Figura 24 se exponen los esquemas de los datos de prueba. Esta prueba contiene un alto porcentaje de cadenas de caracteres, de contenedores y de clases anidadas; en general, la estructura de los datos es relativamente compleja, tanto en un sentido de alojamiento en memoria como de transversalidad de los datos.

³⁰ Documentación oficial de WPR: <https://learn.microsoft.com/es-es/windows-hardware/test/wpt/windows-performance-recorder>

³¹ Documentación oficial de WPA: <https://learn.microsoft.com/es-es/windows-hardware/test/wpt/windows-performance-analyzer>

<pre> 1 struct ecommerce_user { 2 uint64_t uuid; 3 string name; 4 string email; 5 vector<string> 6 recent_searches; 7 }; 8 9 enum ecommerce_product_ 10 category : uint8_t { 11 electronics, books, 12 clothing, home, 13 garden, toys, food, 14 baby, pets, health, 15 beauty 16 }; </pre>	<pre> 17 struct ecommerce_product { 18 uint64_t uuid; 19 string name; 20 string description; 21 ecommerce_product_category 22 category; 23 vector<string> tags; 24 string image_lo_res_url; 25 string image_hi_res_url; 26 double price; 27 double discount; 28 uint32_t stock; 29 uint8_t rating; 30 uint32_t reviews; 31 }; 32 </pre>	<pre> 33 struct ecommerce_page { 34 ecommerce_user user; 35 string permanent_url; 36 string query; 37 uint32_t page; 38 uint32_t total_pages; 39 uint32_t results_per_page; 40 uint32_t total_results; 41 vector<ecommerce_product> 42 products; 43 }; 44 45 46 47 48 </pre>
---	--	---

Figura 24. Esquema de los datos de prueba del experimento #1. Autoría propia.

Experimento 2: Modelo 3D. El segundo experimento trata la serialización y la deserialización de datos que representan la geometría de un objeto tridimensional, a veces conocidos como modelo 3D o malla 3D. La Figura 25 muestra el esquema de los datos de prueba, conformados por un objeto raíz que contiene una lista de vértices, una lista de vectores normales y una lista de caras. En este experimento la estructura de los datos es relativamente sencilla y directa; se trabaja únicamente con números enteros y números de coma flotante, y se busca evaluar el rendimiento de las librerías en este tipo de situaciones donde predominan los datos numéricos.

<pre> 1 struct ivec3 { 2 int64_t x, y, z; 3 }; 4 5 struct fvec3 { 6 double x, y, z; 7 }; </pre>	<pre> 8 using vertex = fvec3; 9 using normal = fvec3; 10 11 struct face { 12 ivec3 vertex_index; 13 ivec3 normal_index; 14 }; </pre>	<pre> 15 struct obj_3d { 16 vector<vertex> vertices; 17 vector<normal> normals; 18 vector<face> faces; 19 }; 20 21 </pre>
---	--	--

Figura 25. Esquema de los datos de prueba del experimento #2. Autoría propia.

En los experimentos, los procesos de serialización se realizan a partir de datos de prueba previamente cargados en memoria, transformándolos a su estado serializado en un buffer también ubicado en memoria. Se hace de esta manera para evitar incorporar factores externos que puedan influenciar la medición de los procesos, como ocurre con el caso de la lectura y escritura directa en disco de los archivos binarios. Las pruebas son ejecutadas con un

rango de tamaño de datos de entrada de 1 a 1024 MiB. El tiempo de generación de estos datos no es tomado en cuenta para las mediciones del tiempo de ejecución de los procesos de serialización y deserialización ni de las mediciones del uso del CPU, pero el tamaño ocupado en memoria por estos datos si es considerado para las mediciones de uso de la memoria principal.

Librerías a medir y evaluar.

En la Tabla 7 se listan las librerías de serialización a medir y evaluar en las pruebas de rendimiento con sus respectivas configuraciones. La librería “Teg” es el producto de la presente investigación, y se mide en sus dos modalidades: modo por defecto y modo compacto. Las demás son una recopilación de las librerías más utilizadas en la industria según Biswal y Almallah (2019) y Qi (2022), que son Protocol Buffers, FlatBuffers, MessagePack y Boost.Serialization.

Tabla 7

*Configuración de las librerías de serialización a evaluar en las pruebas de rendimiento.
Autoría propia.*

Librería	Versión	Desarrollador(es)	Configuración
Teg	1.0.0	Autoría propia.	Modo por defecto: ordenamiento de bytes little-endian, enteros fijos de 32 bits para el tamaño de los contenedores y capacidad máxima de 2 GiB.
Teg (modo compacto)	1.0.0	Autoría propia.	Modo compacto: ordenamiento de bytes little-endian, compresión ULEB-128 para todos los números enteros y capacidad máxima de 1 GiB.
Protocol Buffers	25.2	Google LLC.	Configuración predeterminada. La medición del tiempo de ejecución excluye los procesos de conversión del formato generado por el compilador `protoc`.
FlatBuffers	24.3.6	Google LLC.	Configuración predeterminada. En la medición de los tiempos de deserialización se incluye la iteración transversal de los datos.
MessagePack	7.0.0	Sadayuki Furuhashi y Takatoshi Kondo.	Configuración predeterminada. En las mediciones se incluyen los procesos de empaquetado y desempaquetado pero no los de conversión.
Boost.Serialization	1.87.0	Robert Ramey.	Configuración según la documentación oficial. Utilización de flujos de datos estándar y archivos binarios de Boost.

Datos de prueba.

Se desarrolló una librería mínima y ligera de utilidad para suministrar a los experimentos de datos de prueba. A través de una función, dada un número arbitrario de bytes y dependiendo del experimento, se crean y retornan un conjunto de datos del tamaño requerido, como se muestra en la Figura 26. Los datos de prueba son generados con números pseudoaleatorios utilizando un motor de Mersenne³², siendo establecida una única semilla para todos los experimentos, lo que asegura que los datos sean consistentes y uniformes en todas las rutinas de medición.

```

1 auto generate_benchmark_data(std::size_t bytes) -> obj_3d {
2     obj_3d test_data {};
3
4     std::seed_seq seed{99, 5, 11};
5     std::mt19937_64 engine {seed};
6
7     const auto vert_bytes = bytes * (2.f/10.f);
8     const auto norm_bytes = bytes * (3.f/10.f);
9     const auto face_bytes = bytes * (5.f/10.f);
10    const auto remaining = bytes - (vert_bytes + norm_bytes + face_bytes);
11
12    return obj_3d{
13        random_vertices(engine, vert_bytes),
14        random_vertices(engine, norm_bytes),
15        random_faces(engine, face_bytes + remaining)
16    };
17 }
18
19 // Generate 1 MiB of data.
20 const auto data_out = test::generate_benchmark_data(1024 * 1024);

```

Figura 26. Generación de datos de prueba en función de un tamaño en bytes. Autoría propia.

³² El Mersenne Twister es un algoritmo de generación de números pseudoaleatorios desarrollado por Makoto Matsumoto y Takuji Nishimura en 1997. La librería estándar posee una implementación de este motor `std::mt19937` : https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine

Informe de rendimiento.

Las pruebas de rendimiento se ejecutaron en un equipo con un procesador Intel i5-4690S 3.4 GHz de arquitectura x86-64 y memoria RAM 32 GB dual-channel DDR3 1600 MHz, y sistema operativo Windows 11 Pro. Habiendo realizado las mediciones pertinentes (véase Apéndice D), se hallaron los siguientes resultados:

Experimento 1: Query e-commerce.

En la Figura 27 y Figura 28 se observa la tasa de procesamiento de serialización y deserialización en función del tamaño de los datos de prueba. La librería Teg, producto de la investigación, reporta una velocidad de procesamiento notablemente mayor que el resto de las librerías evaluadas, llegando a serializar en promedio 118% más GiB/s que Protocol Buffers, 552% más GiB/s que FlatBuffers y hasta un 450% más GiB/s que Boost.Serialization. Sin contar FlatBuffers, en la deserialización Teg también sobresale llegando a ser 178%, 353% y 725% más rápida que Protocol Buffers, Boost.Serialization y MessagePack, en ese orden.

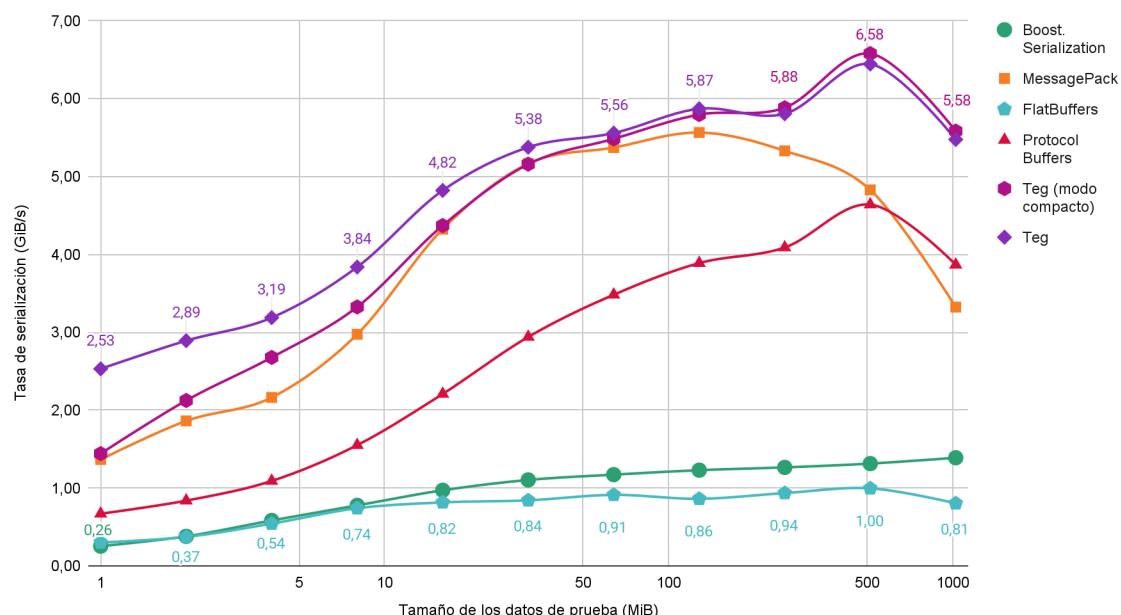


Figura 27. Tasa de serialización de objetos (GiB/s) en función del tamaño de los datos de prueba (MiB) en el experimento #1. Autoría propia.

Con respecto a MessagePack, Teg muestra una mejora de entre 32% a 87% en la velocidad en los procesos de serialización, sin embargo, su ventaja se hace mucho más evidente en la deserialización, donde Teg logra ser desde 195% y hasta 725% más rápida. Por otra parte, FlatBuffers se destaca por su

rápida deserialización en comparación con otras librerías, gracias a un diseño que invierte más tiempo en la serialización, lo que minimiza significativamente el tiempo necesario para este proceso. Aun así, para tamaños de archivos pequeños Teg incluso supera a FlatBuffers en un rango de 15% a 300% más GiB/s al deserializar, y llega a ser en promedio un 55% más lenta solo en entradas de datos más pesados.

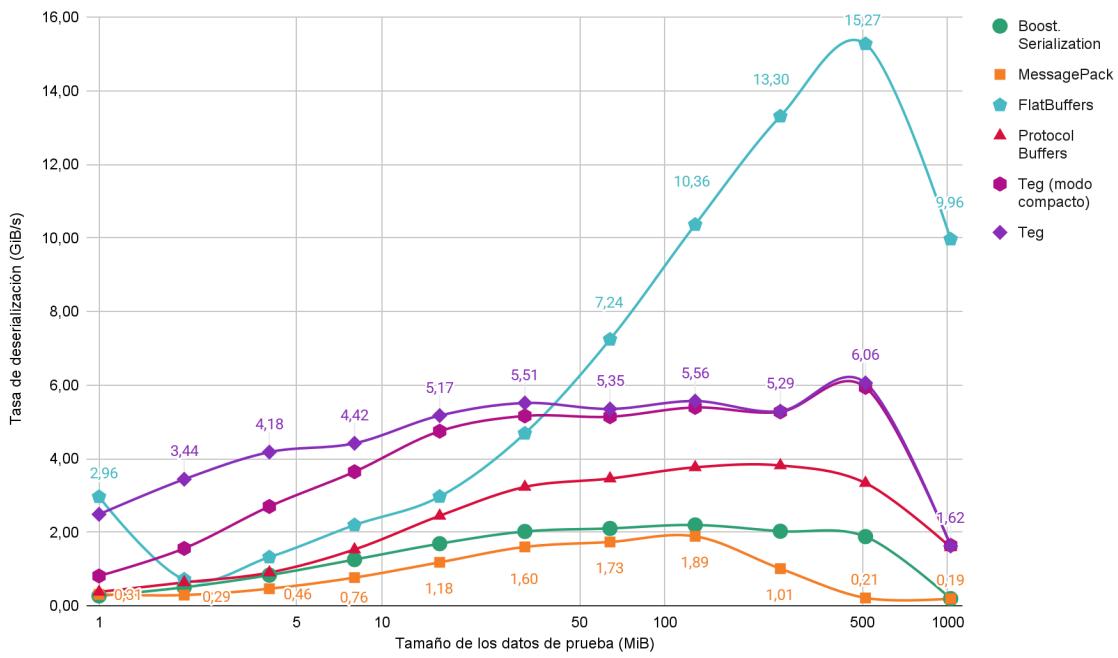


Figura 28. Tasa de deserialización de objetos (GiB/s) en función del tamaño de los datos de prueba (MiB) en el experimento #1. Autoría propia.

En la Figura 29 se presenta una comparación de los tiempos de ejecución totales para los procesos de serialización y deserialización de las librerías evaluadas. Los resultados muestran que Teg es notablemente más rápida que las demás opciones. Al considerar ambos procesos, Teg supera a FlatBuffers con un tiempo de ejecución inferior. En promedio, Teg requiere un 50% menos de tiempo de ejecución que Protocol Buffers, un 66% menos que FlatBuffers, un 71% menos que MessagePack y un 78% menos que Boost.Serialization.

La Figura 30 muestra una comparación del tamaño de los archivos binarios generados en el proceso de serialización de cada librería. Se observa que no hay una diferencia considerable entre los distintos formatos. No obstante, Teg en modo por defecto llega a ocupar en promedio entre un 2% a 6% más tamaño que el resto de alternativas.

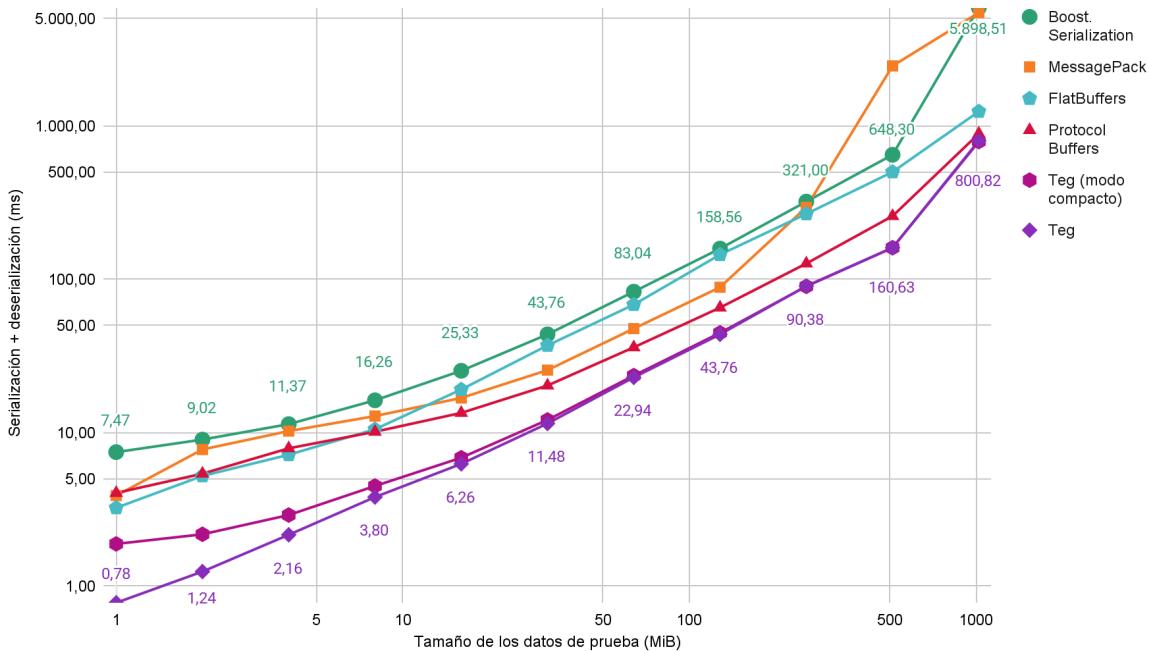


Figura 29. Tiempo total de serialización y deserialización (ms) de las librerías en función del tamaño de los datos de prueba (MiB) en el experimento #1. Autoría propia.

Teg en modo compacto presenta casi la misma velocidad de procesamiento que en modo por defecto, siendo en promedio un 26% más lento con esta configuración. Y, en cuanto a tamaño de los archivos binarios, en este caso, el modo compacto ahorra en promedio un 6,5% de espacio.

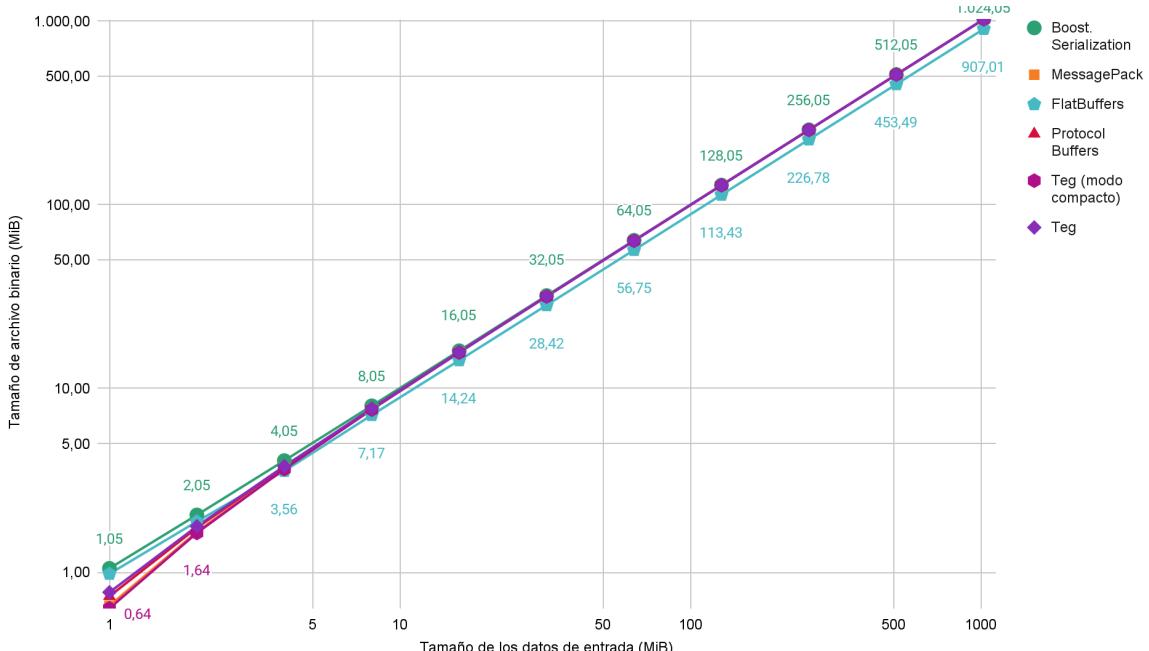


Figura 30. Tamaño de los archivos binarios (MiB) generados en el proceso de serialización de objetos en función del tamaño de los datos prueba (MiB) en el experimento #1. Autoría propia.

De todas las librerías evaluadas, Teg en sus dos configuraciones demostró ser la más eficiente en términos de uso de memoria principal, como se puede observar en la Figura 31 y la Figura 32. Teg consume entre un 17% y un 43% menos memoria que Protocol Buffers, entre un 8% y un 12% menos que FlatBuffers, entre 4% a 40% menos que MessagePack y de 32% a 45% menos que Boost.Serialization.

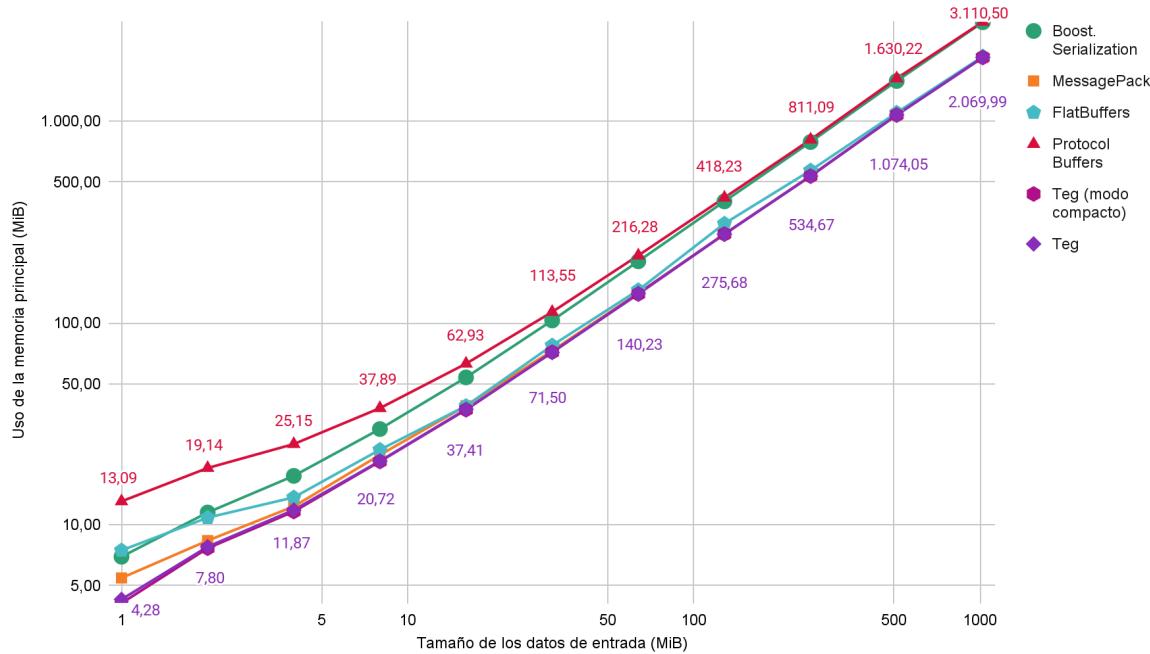


Figura 31. Uso de la memoria principal (MiB) en el proceso de serialización de objetos en el experimento #1. Autoría propia.

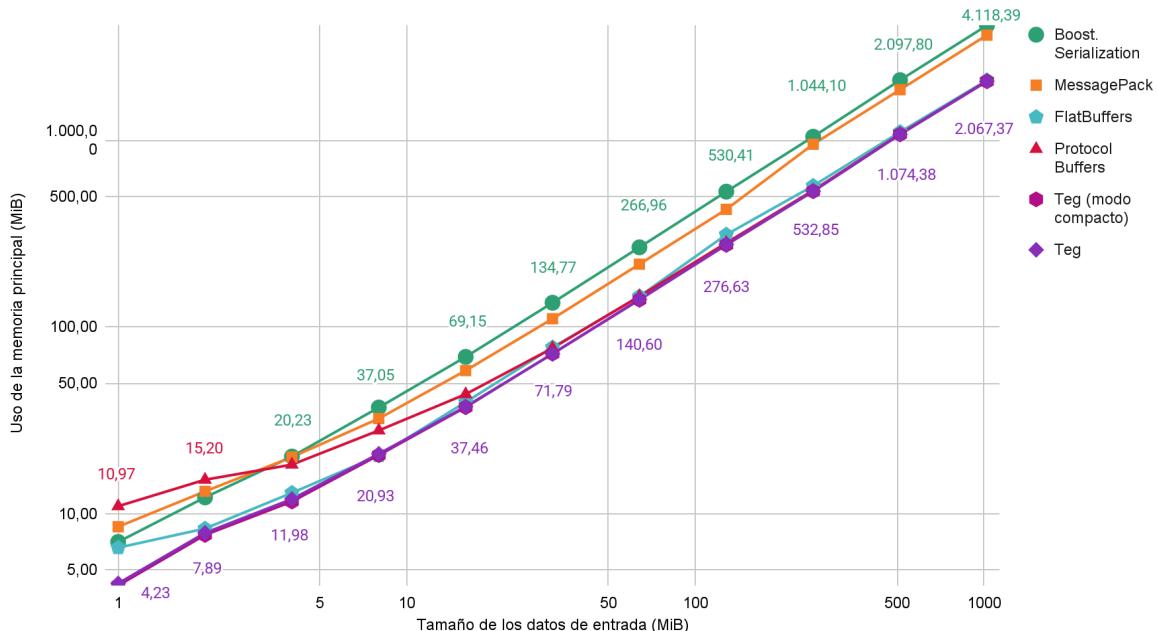


Figura 32. Uso de la memoria principal (MiB) en el proceso de deserialización de objetos en el experimento #1. Autoría propia.

Experimento 2: Modelo 3D.

Los resultados mostrados en la Figura 33 y Figura 34 indican que Teg en modo por defecto supera por un amplio margen a las otras librerías de serialización en el experimento #2 en cuanto a la tasa de procesamiento. En comparación, al serializar, Teg es entre 1.500% y 3.300% más rápida que Protocol Buffers, entre 500% y 2.000% más rápida que FlatBuffers, entre 1.000% y 2.000% más rápida que MessagePack y desde 4.000% hasta 11.600% más rápida que Boost.Serialization. En la deserialización, a excepción de FlatBuffers, nuestra librería presenta un margen incluso aún más vasto. Específicamente para tamaños de datos entre 1 y 4 MiB, Teg llega a procesar hasta 7.300% más GiB/s que MessagePack, 4.200% más GiB/s que Protocol Buffers y hasta 11.000% más GiB/s que Boost.Serialization.

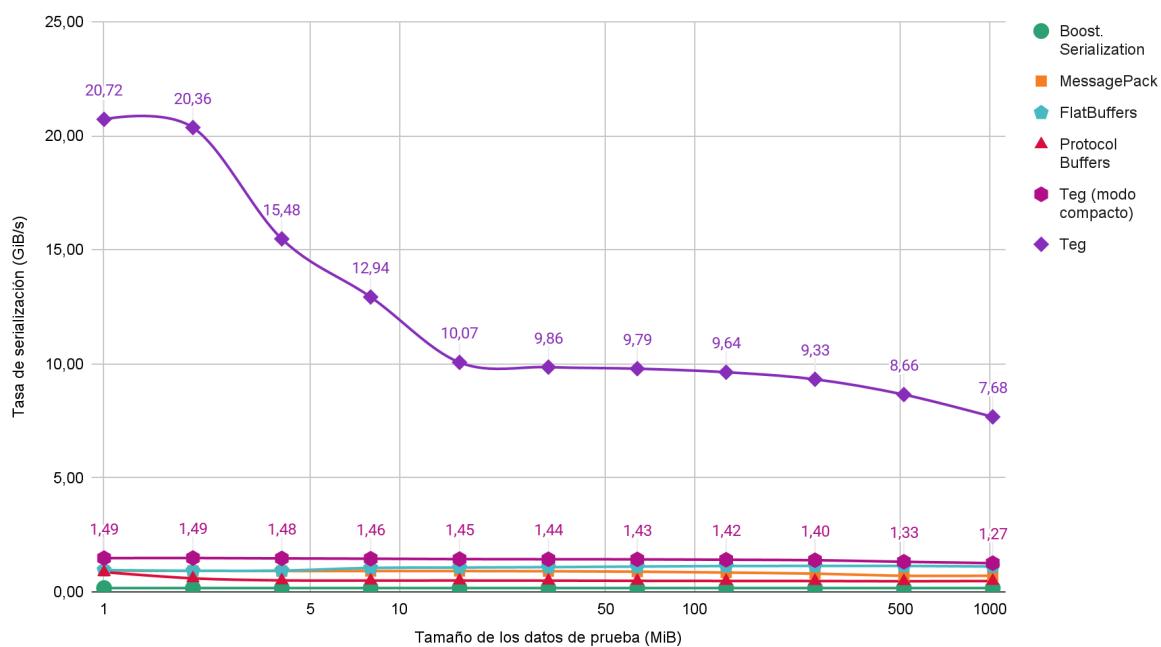


Figura 33. Tasa de serialización de objetos (GiB/s) en función del tamaño de los datos de prueba (MiB) en el experimento #2. Autoría propia.

Al igual que ocurre en el experimento #1, FlatBuffers reporta en promedio una mayor tasa de deserialización. Aun así, para tamaños de datos pequeños, Teg supera ligeramente a FlatBuffers pudiendo procesar entre 8% y 13% más GiB/s. Es a partir de una entrada de datos más grande donde Teg llega a ser desde un 30% a un 50% más lenta. No obstante, cuando se consideran ambos procesos, como se muestra en la Figura 35, se puede

observar que Teg requiere significativamente un menor tiempo para tratar la misma cantidad de datos. La etapa de serialización de FlatBuffers merma gravemente su rendimiento general, y, en cambio, Teg en modo por defecto no solo supera a FlatBuffers en términos de serialización sino que también mantiene un buen desempeño en la deserialización.

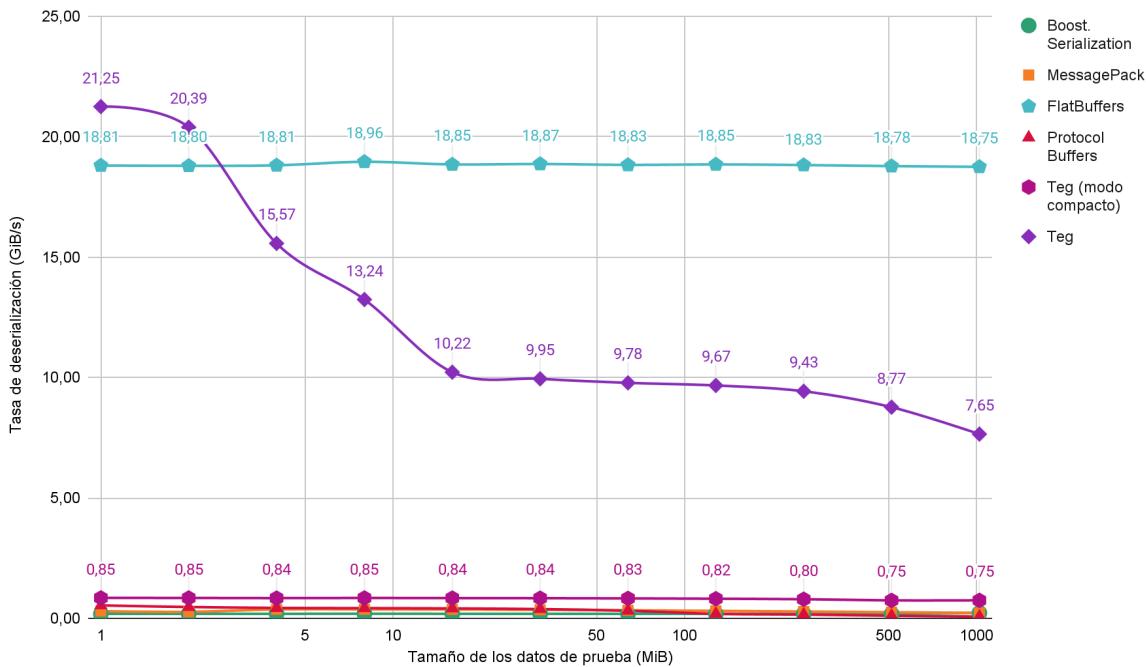


Figura 34. Tasa de deserialización de objetos (GiB/s) en función del tamaño de los datos de prueba (MiB) en el experimento #2. Autoría propia.

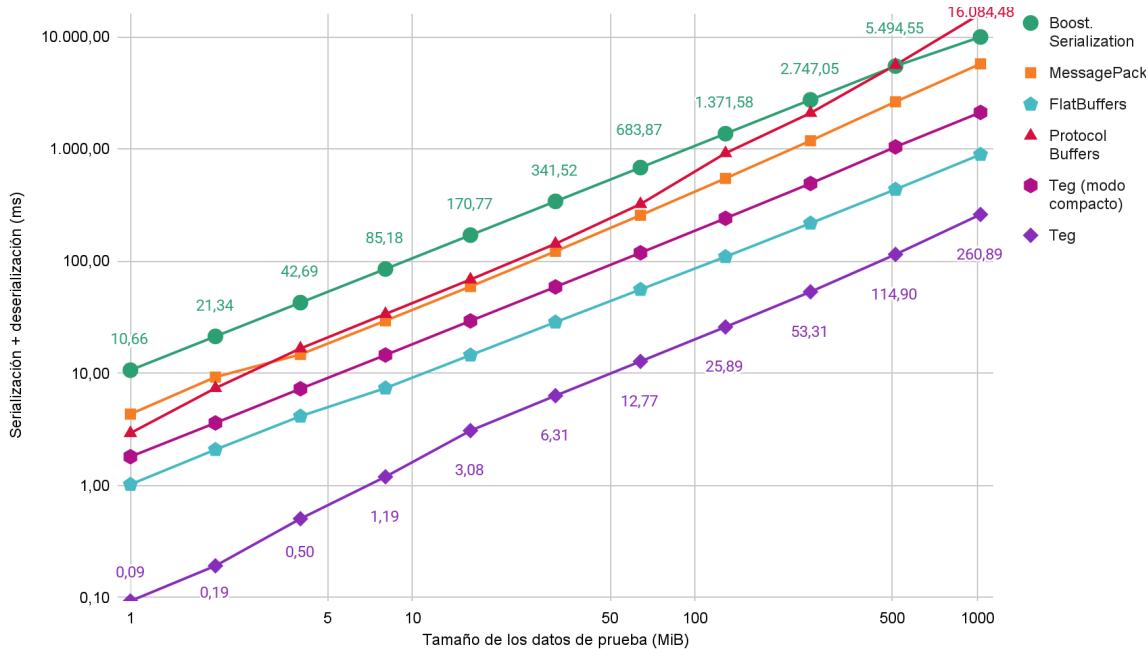


Figura 35. Tiempo total de serialización y deserialización (ms) en función del tamaño de los datos de prueba (MiB) en el experimento #2. Autoría propia.

En modo compacto, Teg reporta un tiempo de ejecución que, en promedio, es 1000% veces mayor en comparación con su configuración por defecto. Sin embargo, los resultados del experimento revelan que, incluso en este modo, Teg requiere un menor tiempo de procesamiento que otras librerías de serialización. Específicamente, ocupa un 64% menos tiempo de ejecución que Protocol Buffers, 56% menos que MessagePack y 82% menos que Boost.Serialization. Aunque su rendimiento en comparación con FlatBuffers implica un aumento del 102% en milisegundos al procesar los datos, Teg en modo compacto continúa siendo una opción competitiva frente a las demás librerías analizadas.

En la Figura 36 se muestra una comparación de los tamaños de archivo binarios generados. Teg en modo compacto requiere un menor tamaño que el resto de las librerías, ocupando un 16% menos que MessagePack, un 24% menos que Protocol Buffers y un 33% menos que Teg en modo por defecto, FlatBuffers y Boost.Serialization. Por otra parte, con la configuración por defecto, Teg no presenta compresión, utilizando un tamaño de archivo muy similar al tamaño de los datos de entrada, necesitando un 13% más espacio que Protocol Buffers y un 25% más espacio que MessagePack.

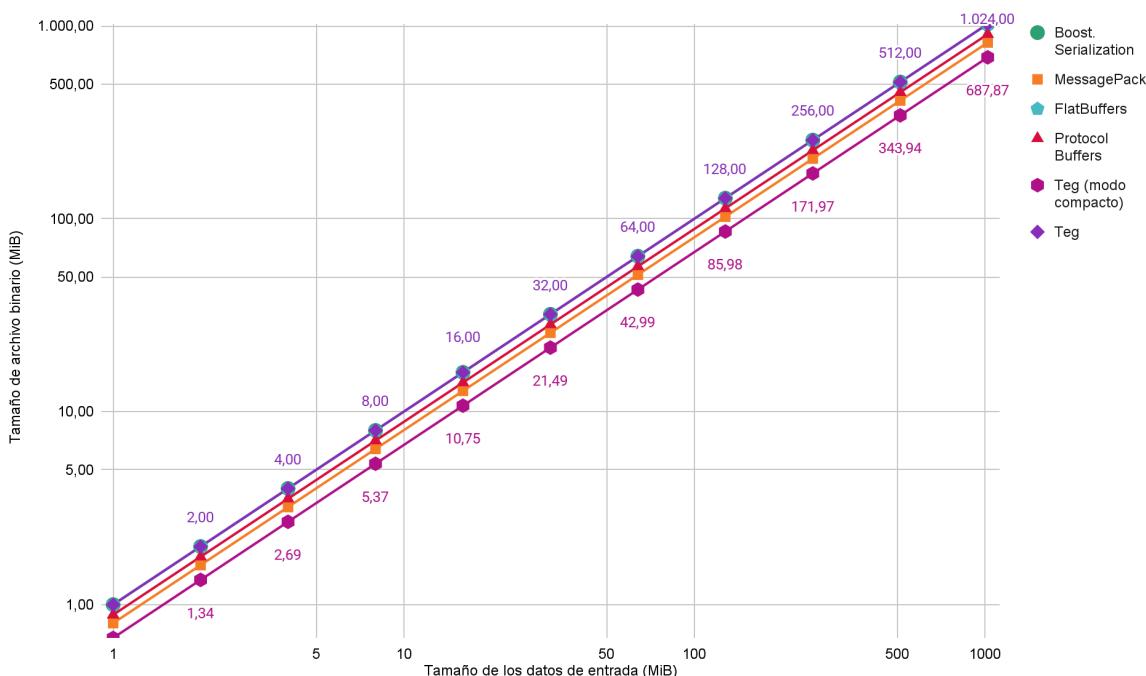


Figura 36. Tamaño de los archivos binarios (MiB) generados en los procesos de serialización en función del tamaño de los datos prueba (MiB) en el experimento #2. Autoría propia.

Como se muestra en la Figura 37, Teg en su configuración compacta logra una reducción del 33% en el espacio ocupado por el archivo binario con respecto al tamaño de los datos de entrada, alcanzando un ratio de compresión de 1:1,49, siendo mayor que al de Protocol Buffers, que presenta un ratio de 1:1,13, y mayor que el de MessagePack con un ratio de 1:1,25. En modo por defecto, Teg no realiza ninguna compresión de datos, comportamiento que también se observa en FlatBuffers y Boost.Serialization.

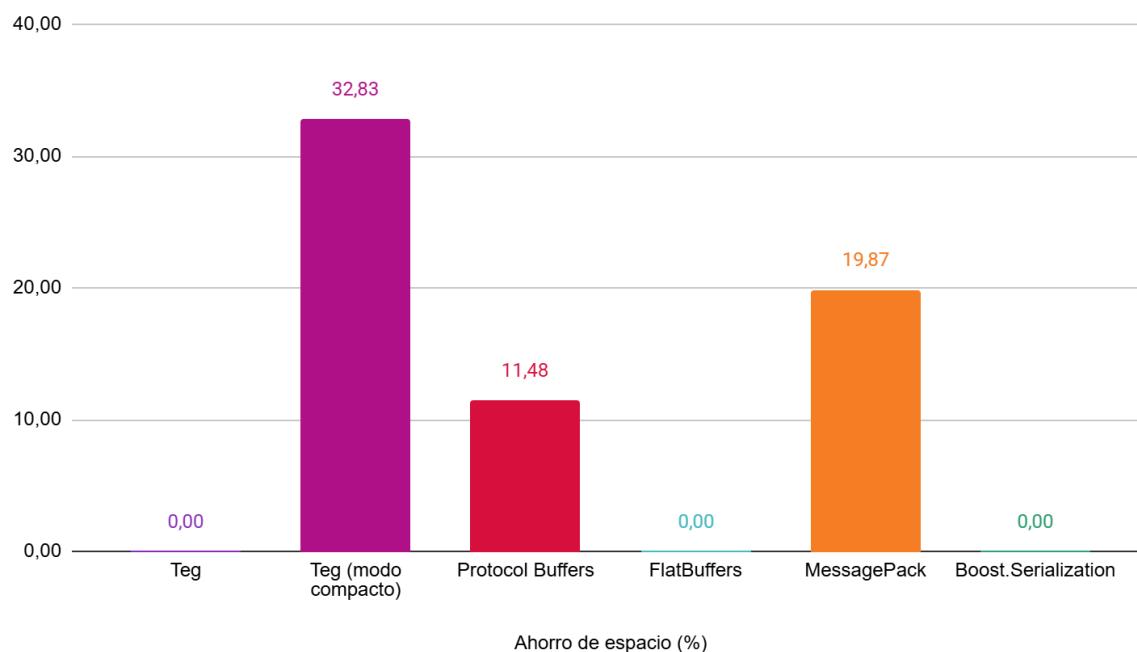


Figura 37. Ahorro de espacio en los archivos binarios generados por las librerías en comparación al tamaño de los datos procesados en el experimento #2. Autoría propia.

La Figura 38 y la Figura 39 ilustran la utilización de la memoria principal. Similar al experimento #1, la librería desarrollada en la investigación demuestra una menor utilización de memoria en las dos configuraciones evaluadas con respecto a las demás alternativas. Teniendo en cuenta ambos procesos, Teg consume entre un 18% y un 43% menos de memoria que Protocol Buffers, entre un 8% y un 12% menos que FlatBuffers, entre un 4% y un 40% menos que MessagePack, y entre un 32% y un 45% menos que Boost.Serialization.

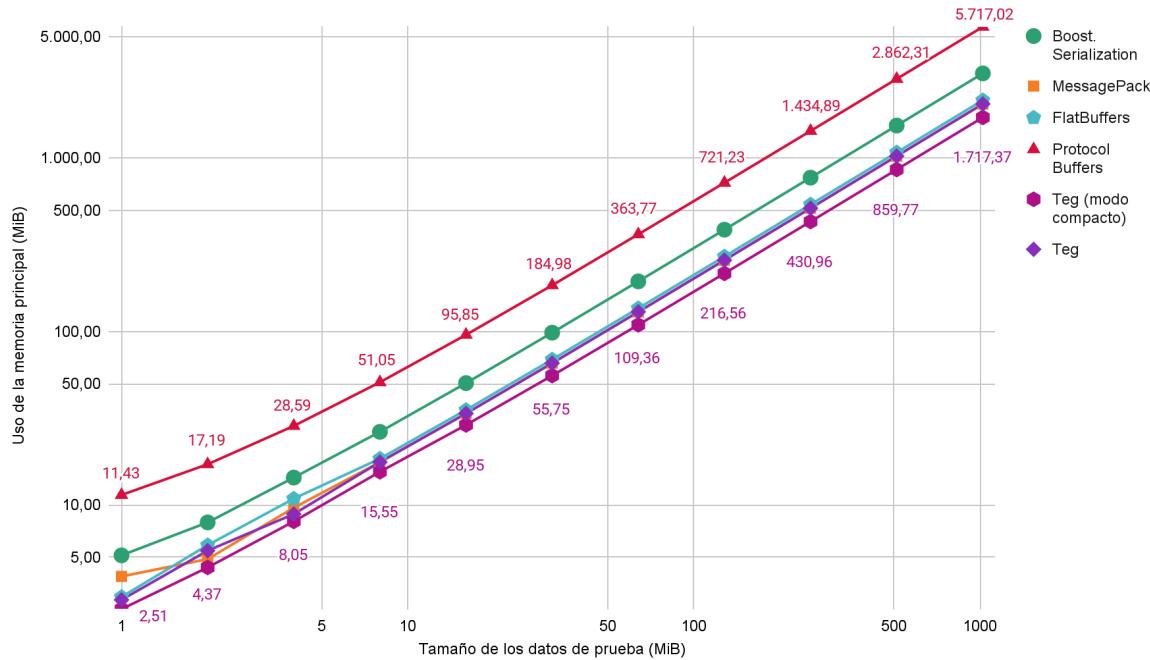


Figura 38. Uso de la memoria principal (MiB) durante la serialización de objetos en el experimento #2. Autoría propia.

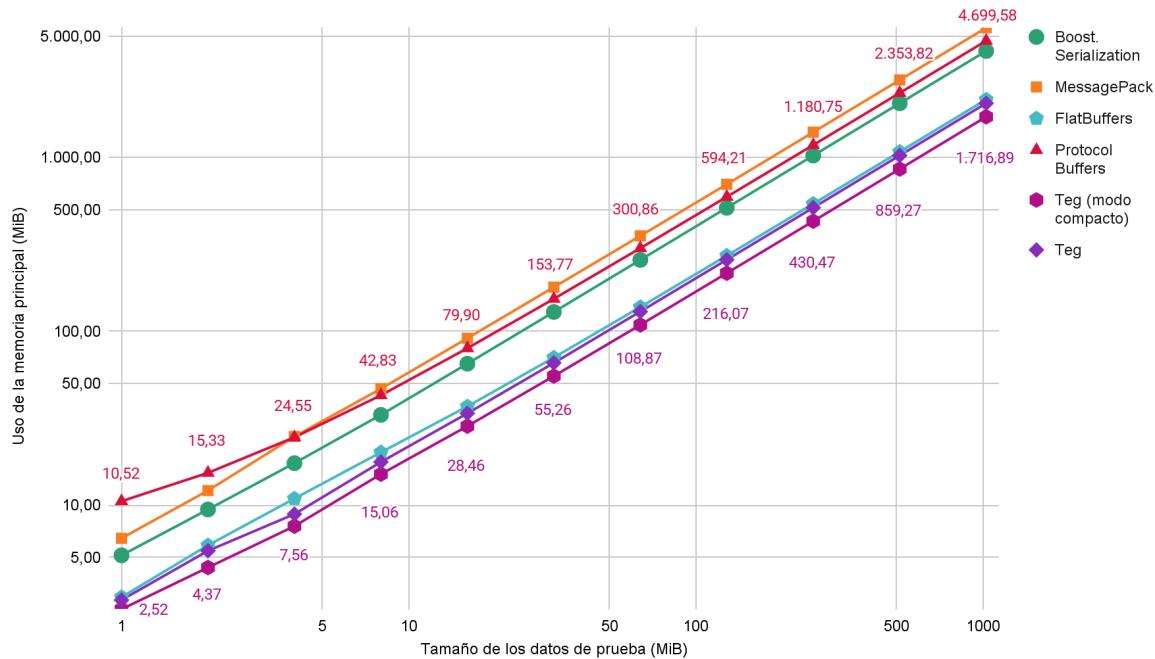


Figura 39. Uso de la memoria principal (MiB) durante la deserialización de objetos en el experimento #2. Autoría propia.

Finalmente, en ambos experimentos realizados, al medir la utilización del CPU, se observó consistentemente que todas las librerías evaluadas presentaban valores que oscilaban entre el 24% y el 25% (véase Apéndice D). La Figura 40 ilustra este comportamiento. Cabe destacar que todas las pruebas

de rendimiento se ejecutaron en un único hilo y en una computadora con 4 núcleos. En este caso, una utilización del 25% del CPU corresponde al uso completo de uno de los núcleos. En relación a esto, no se observaron diferencias significativas en la utilización del procesador, más allá del hecho evidente de que aquellas librerías con menor tiempo de ejecución para un mismo tamaño de datos también reportan un menor tiempo de uso total de la CPU.

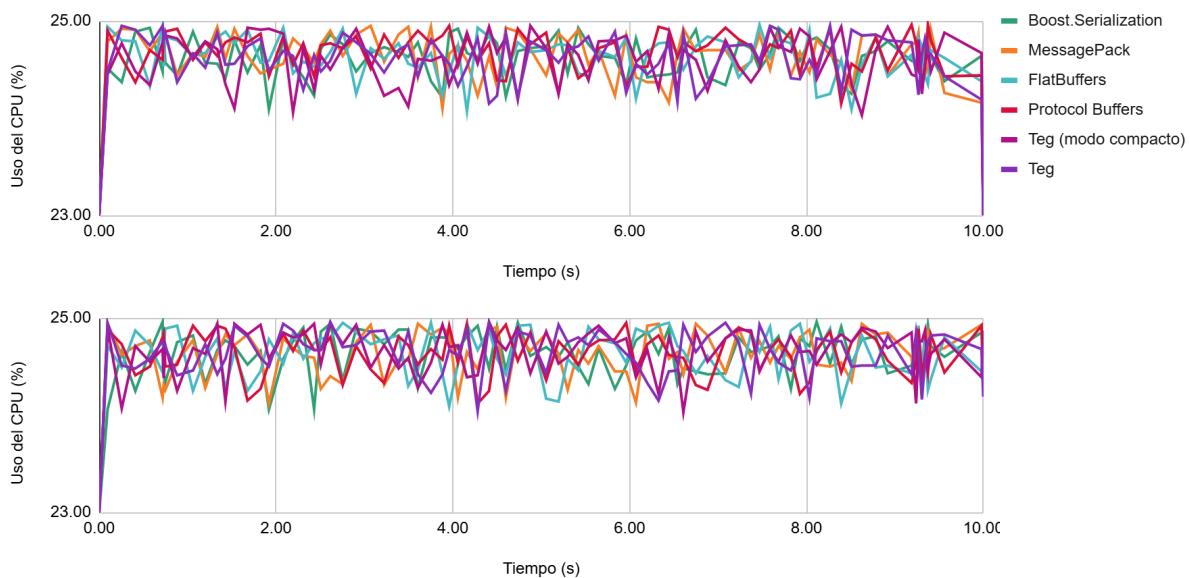


Figura 40. Utilización del CPU (porcentaje de todos los núcleos) en los procesos de serialización y deserialización (cuadro superior e inferior respectivamente) de 1 MiB de datos de prueba en el experimento #2. Autoría propia.

Discusión de los resultados.

Los resultados de las pruebas de rendimiento indican que Teg, la librería desarrollada en la investigación, exhibe una tasa de procesamiento de datos considerablemente mayor, alcanzando en algunos casos un rendimiento de entre 20 y 100 veces más gigabytes por segundo en comparación a las librerías evaluadas. Además, Teg consume menos memoria principal y presenta un uso del procesador equivalente al de las otras opciones. Incluso en comparación con FlatBuffers, que prioriza la optimización de la deserialización a costa de la eficiencia en la serialización, la solución propuesta ha demostrado, en promedio, una tasa de deserialización competitiva e incluso, dependiendo del volumen de datos, superior.

Teg en modo compacto llega a reducir desde 10% hasta un 33% de espacio según el esquema y tamaño de los datos, y aun así muestra una tasa de procesamiento semejante o, incluso en algunos casos, mayor al del resto de librerías medidas. Aunque la configuración por defecto suele ser más rápida, cuando el número de datos que pueden ser comprimidos es mínimo, este modo reporta un rendimiento muy similar. Esto es así porque en el desarrollo de la librería se siguió el principio de sobrecarga nula, donde solo se paga, en este caso en recursos computacionales, por las características que se utilizan.

En la Tabla 8 se muestra una comparación de algunas de las características que ofrecen las librerías de serialización analizadas en la investigación. Al seleccionar una librería, es fundamental considerar sus atributos, ventajas y desventajas, en función del contexto específico de las aplicaciones que se pretenden desarrollar, lo que permitirá tomar una decisión bien fundamentada.

Tabla 8

Comparación de características funcionales de las librerías de serialización binaria evaluadas en la investigación. Autoría propia.

Característica	Teg	Teg (compacto)	Protocol Buffers	FlatBuffers	MessagePack	Boost .Serialization
Formato binario portable	✓	✓	✓	✓	✓	✗
Configuración de	✓	✓	✗	✗	✗	✗

Característica	Teg	Teg (compacto)	Protocol Buffers	FlatBuffers	MessagePack	Boost .Serialization
ordenamiento de bytes						
Validación de formato	✓	✓	✓	✓	✓	✗
Compatibilidad de esquema hacia atrás	✓	✓	✓	✓	✓	✗ ¹
Compatibilidad de esquema hacia adelante	✗	✗	✓	✓	✓	✗ ¹
Compresión de números enteros	✗	✓	✓	✗	✓	✗
Soporte en múltiples lenguajes	✗ ²	✗ ²	✓	✓	✓	✗ ²
Acceso aleatorio a los datos (deserialización parcial)	✗	✗	✗	✓	✓	✗

Notas.

1. Boost.Serialization incorpora un mecanismo de versionado de clases, pero la gestión de la compatibilidad, tanto hacia versiones anteriores como futuras, debe ser implementada manualmente por el desarrollador.

2. Actualmente solo están implementados en C++.

FlatBuffers y Protocol Buffer requieren de un proceso previo de conversión de datos antes de la serialización, y esto también ocurre de manera parecida en la deserialización, donde se suele recomendar crear envoltorios de las clases automáticamente generadas por los compiladores de estas librerías³³.

En las pruebas de rendimiento hemos optado por no incluir el tiempo de ejecución de varios de estos procesos para estas dos librerías, ya que suelen ser dependientes de las necesidades específicas de los usuarios, y existe un alto margen para que sean optimizados según sea necesario. Con nuestra solución, utilizando la codificación explícita, es posible serializar y deserializar los datos directamente a clases del dominio, lo que puede resultar más eficiente y conveniente que las clases envoltorio.

Las librerías Teg, Protocol Buffers y Boost.Serialization requieren de la deserialización completa de los datos antes de poder acceder a ellos, y, en cambio, FlatBuffers y MessagePack admiten el acceso aleatorio. Esto significa que en las situaciones donde no se necesita leer todos los datos sino solo unos específicos, soluciones como MessagePack o FlatBuffers suelen ser más

³³ Recomendaciones con respecto al uso de la API de Protocol Buffers: <https://protobuf.dev/getting-started/cpptutorial/#parsing-serialization>

adecuadas. Por otra parte, Teg permite la configuración específica del ordenamiento de bytes, característica que no suele ser común en otras librerías de serialización. Por ejemplo, Protocol Buffers trabaja con little-endian y MessagePack con big-endian. La conversión del orden de bytes puede influir significativamente en el rendimiento general de los procesos de codificación y decodificación. En ciertos escenarios, como en arquitecturas cliente-servidor, esta flexibilidad permite optimizar la deserialización en el lado del cliente, mejorando así la eficiencia del sistema.

Además de una alta eficiencia de desempeño demostrada en las pruebas de rendimiento, Teg es el resultado de un proceso de desarrollo enfocado en cumplir rigurosas características de calidad. En términos de compatibilidad, la librería fue diseñada promoviendo la interoperabilidad en dos aspectos ortogonales: formato de archivo binario e interfaz de programación. En este sentido, los archivos binarios son compatibles en cualquier plataforma y, a su vez, poseen un mecanismo de retrocompatibilidad de esquema de datos. Por otra parte, el código fuente ha sido implementado utilizando únicamente el lenguaje C++ estándar, lo que implica que puede ser compilado y usado en cualquier arquitectura de computador antigua o moderna.

A su vez, Teg se distingue por su capacidad de mantenibilidad y reusabilidad, gracias a su arquitectura modular y el uso de patrones de diseño como Strategy que permiten la separación de responsabilidades y la inversión de dependencias. El enfoque de reflexión en el que está basado la librería simplifica el proceso de definición de esquemas, eliminando la necesidad de archivos externos e inyección de código intrusivo, ofreciendo una interfaz transparente, clara y sencilla de utilizar. Por otra parte, la librería es flexible, adaptable, y escalable, ya que brinda la capacidad de configurar el formato de archivo binario a través de opciones y modos predefinidos, permitiendo balancear rendimiento y capacidad según sea necesario.

Capítulo V

Conclusiones y Recomendaciones

En este capítulo se presentan las conclusiones de esta investigación, basadas en el análisis de los resultados y el marco teórico. Además, se incluyen recomendaciones sobre futuros estudios y sugerencias de cómo éstos pueden ser abordados.

Conclusiones

La serialización y deserialización son procesos fundamentales en el manejo eficiente de datos en diversos campos de la Informática, especialmente en entornos donde la transferencia y el almacenamiento de información son críticos. La presente investigación tuvo como resultado el desarrollo de una librería de serialización y deserialización binaria, denominada "Teg", fundamentada en un novedoso enfoque de reflexión en tiempo de compilación que utiliza exclusivamente características estándares de C++20 (ISO/IEC 14882:2020).

En primer lugar, se diseñó una especificación de formato de archivo binario portable. Este diseño abordó explícitamente los desafíos de portabilidad que existían en el diseño original de Qi (2022), incorporando soporte configurable para el ordenamiento de bytes y garantizando la ausencia de bytes de relleno a través de una codificación formal de tipos fundamentales, resultando en un formato de datos compatible multi-plataforma.

Seguidamente, se desarrolló un módulo de reflexión en tiempo de compilación como pilar fundamental de la librería. Utilizando las capacidades de C++20 como los Conceptos y la evaluación de expresiones constantes, este módulo proporciona funcionalidades avanzadas de introspección para tipos de datos agregados. Se implementaron algoritmos para obtener metadatos esenciales, como el conteo de variables miembros, y se crearon funciones genéricas de visita para aplicar operaciones sobre los miembros de una clase agregada de forma secuencial.

Basado en el formato y el módulo de reflexión previamente establecidos, se desarrolló el módulo principal de serialización y deserialización. En este se implementó la lógica central para la conversión de objetos hacia y desde la representación binaria especificada. Su diseño presenta una interfaz que separa los procesos de serialización/deserialización de alto nivel de los de codificación/decodificación. Incorpora manejo configurable del ordenamiento de

bytes, soporte para codificación explícita definida por el usuario, mecanismos robustos de gestión de errores y soporte de ejecución en tiempo de compilación.

Posteriormente, se desarrolló un módulo independiente para la codificación y decodificación de enteros de longitud variable. Tras analizar diversas técnicas, se optó por implementar la codificación ULEB-128, combinada con la codificación ZigZag para manejar eficientemente números enteros. Integrado de forma extensible, este módulo permite una reducción significativa de los tamaños de los archivos binarios generados.

Finalmente, se evaluó exhaustivamente el producto de software. Las pruebas de rendimiento evidenciaron un desempeño sobresaliente por parte de la librería Teg en términos de tasa de procesamiento de datos, uso de memoria, carga del procesador y ratio de compresión de los archivos binarios. Al trabajar con cadenas de caracteres, Teg reportó tasas de serialización y deserialización 100% y 700% mayores en comparación a las librerías más populares, tales como Protocol Buffers, MessagePack y Boost.Serialization. Más aún, Teg mostró un rendimiento notablemente superior al manipular números enteros y números reales, alcanzando cifras de entre 1.000% y 3.000% más gigabytes procesados por segundo con respecto a las librerías evaluadas. Además, al configurar la librería en modo compacto, Teg logró reducir el tamaño de los archivos binarios entre un 10% y un 33%, mientras mantenía una tasa de procesamiento de datos 100% y 300% veces mayor en comparación con las alternativas.

Asimismo, la librería Teg se distingue por sus atributos de calidad, los cuales se definieron en conformidad con el estándar ISO/IEC 25010:2023: eficiencia de desempeño, evidenciada en la optimización de recursos y elevada tasa de procesamiento; compatibilidad, por su interoperabilidad tanto en interfaz de programación como en formato de datos; mantenibilidad, debido a su arquitectura modular y reusabilidad a través de patrones de diseño; y flexibilidad, al ser adaptable y escalable según el entorno y la carga de trabajo.

Los hallazgos de esta investigación subrayan el potencial considerable de la reflexión en tiempo de compilación para la optimización de los procesos de serialización y deserialización. La librería Teg representa una solución funcional y de alto rendimiento, y, además, destaca cómo las características modernas del lenguaje pueden ser aprovechadas para obtener mejoras sustanciales, ofreciendo una alternativa de valor para el desarrollo de software en lenguaje C++.

Recomendaciones

La presente investigación aborda un enfoque interesante, siendo la reflexión en tiempo de compilación un tema que aún necesita de más trabajo. Por otra parte, la librería de serialización desarrollada también posee un gran potencial de mejora. A continuación, se presentan las recomendaciones para continuar el estudio:

- **Integridad de los datos:** Se desarrolló un mecanismo para verificar la integridad del formato de los datos, pero se requiere una revisión más extensa sobre la integridad de los datos. Una práctica común consiste en aplicar una suma de verificación (en inglés, *checksum*) para detectar cambios en los archivos binarios. Promovemos el uso de esta técnica y, basándonos en el estudio de Lynch (2021), recomendamos los algoritmos XXH3 o XXH128 en entornos seguros, y BLAKE3 en entornos no seguros.
- **Reflexión estática sobre formatos de texto:** Recomendamos la investigación sobre la aplicación de la reflexión en tiempo de compilación en la serialización y deserialización de formatos de texto, tales como JSON, XML y YAML usando C++20. Para esto, es fundamental explorar la viabilidad de reflejar los metadatos asociados a los nombres de las clases y sus variables miembros, necesarios para los campos de clave o nombres de etiquetas característicos de estos formatos.
- **Soporte de la librería en múltiples lenguajes:** La librería desarrollada está disponible sólo en C++20, pero la especificación de formato de archivo es independiente, pudiéndose utilizar en otros lenguajes. Se espera que en lenguajes compilados con reflexión en tiempo de compilación, el rendimiento sea comparable, mientras que en lenguajes interpretados o sin esta característica, el rendimiento podría ser inferior.
- **Futuras versiones de C++:** el lenguaje C++ evoluciona continuamente gracias al trabajo del comité estándar y a las contribuciones de la comunidad.. Existen propuestas aprobadas y pendientes por aprobación que traen características nuevas al lenguaje con las que se podría mejorar significativamente en un futuro la implementación de la librería producto de esta investigación. En concreto, en C++26 y C++29 será posible remover las limitaciones actuales de visita de miembros, e incluso expandir el repertorio de tipos de datos serializables (véase Apéndice E).

Referencias Bibliográficas

- Biswal, A. K., y Almallah, O. (2019). *Analytical assessment of binary data serialization techniques in IoT context* [Evaluación analítica de técnicas de serialización de datos binarios en el contexto de IdC] (Tesis de maestría). Politecnico di Milano, Milán, Italia. Recuperado de https://www.politesi.polimi.it/bitstream/10589/150617/1/Thesis_ObadaAlmallah.pdf
- Boldo, S., Jeannerod, C.-P., Melquiond, G., & Muller, J.-M. (2023). Floating-point arithmetic [Aritmética de coma flotante]. *Acta Numerica*, 32, 203-290. doi: 10.1017/S0962492922000101
- Bonifácio, R., Carvalho, F., Ramos, G. N., Kulesza, U., y Coelho, R. (2015). The Use of C++ Exception Handling Constructs: A Comprehensive Study [El uso de constructos de manejo de excepciones en C++: Un estudio exhaustivo]. *IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Bremen, Alemania. IEEE. doi: 10.1109/SCAM.2015.7335398
- Carrera Castillo, D., Rosales, J., y Torres Blanco, G. A. (2018). Optimizing Binary Serialization with an Independent Data Definition Format [Optimizando la serialización binaria con un formato de definición de datos independiente]. *International Journal of Computer Applications*, 180(28), 15-18. doi: 10.5120/ijca2018916670
- Carter, C., y Niebler, E. (8 de junio de 2018). *Standard Library Concepts* [Librería estándar de conceptos] (Doc. técnico no. P0898R3). ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee. Recuperado de <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0898r3.pdf>
- Casey, A. M. (2022). *Performance of Serialization Libraries in a High Performance Computing Environment* [Rendimiento de librerías de serialización en un entorno de computación de alto rendimiento] (Tesis de maestría). University of Houston, Texas, Estados Unidos. Recuperado de <https://uh-ir.tdl.org/server/api/core/bitstreams/d10140e3-00d6-4774-9e06-8e7253c06039/content>
- Chiu, K. (2004). XBS: A streaming binary serializer for high performance computing [Un serializador binario de transmisión para la computación de alto rendimiento]. *Proceedings of the High Performance Computing Symposium*. Simposio llevado a cabo en Hyatt Regency Crystal City, Virginia, Estados Unidos. Recuperado de <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=4e4e8112a2f0f38b9315859c501f28f0a0336e20>
- Creager, D. (8 de marzo de 2021). *A better varint* [Un mejor varint] (Entrada de un blog). dcreager.net. Recuperado de <https://dcreager.net/2021/03/a-better-varint/>
- Demers, A., y Malenfant, J. (1995). Reflection in logic, functional and object-oriented programming: A short comparative study [Reflexión en la programación lógica, funcional y orientada a objetos: Un breve estudio comparativo]. *IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, 29–38. Recuperado de <http://www-public.int-evry.fr/~gibson/Teaching/Teaching-ReadingMaterial/DemersMalenfant95.pdf>

- Dos Reis, G., Stroustrup, B., y Maurer, J. (17 de abril de 2007). *Generalized Constant Expressions — Revision 5* [Expresiones Constantes Generalizadas — Revisión 5] (Doc. técnico no. N2235=07-0095). ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee. Recuperado de <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2235.pdf>
- Fertig, A. (2021). *Programming with C++20: Concepts, Coroutines, Ranges, and more* [Programación con C++20: Conceptos, Corrutinas, Rangos y más]. Berlín, Alemania: Fertig Publications.
- Fitzgerald, B. (17 de septiembre de 2015). Zig-zag encoding [Codificación Zig-Zag] (Entrada de un blog). Recuperado de <https://neurocline.github.io/dev/2015/09/17/zig-zag-encoding.html>
- Friedman M., B. (2012). Measuring processor utilization in Windows [Midiendo la utilización del procesador en Windows]. Demand Technology Software. Recuperado de <https://demandtech.com/wp-content/uploads/2012/04/Measuring-Processor-Utilization-in-Windows.pdf>
- Fu, W.-H. A. (2023). *Using Quantization and Serialization to Improve AI Super-Resolution Inference Time on Cloud Platform* [Uso de la cuantificación y la serialización para mejorar el tiempo de inferencia de superresolución de IA en la plataforma en la nube]. KTH Royal Institute of Technology. Recuperado de <https://www.diva-portal.org/smash/get/diva2:1783237/FULLTEXT01.pdf>
- Gamma, E., Helm, R., Johnson, R., y Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* [Patrones de Diseño: Elementos de Software Orientado a Objetos Reutilizables]. Estados Unidos: Addison-Wesley.
- Hinnant, H. E. (13 de julio de 2017). *endian, Just endian* [endian, Solo endian] (Doc. técnico no. P0463R1). ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee. Recuperado de <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0463r1.html>
- ISO/IEC. (2023). *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models* (ISO/IEC 25010:2023).
- ISO/IEC. (2017). *Programming languages — The C++ Language* (ISO/IEC 14882:2017). International Organization for Standardization.
- ISO/IEC. (2020). *Programming languages — The C++ Language* (ISO/IEC 14882:2020). International Organization for Standardization.
- Jabot, Corentin. (19 de enero de 2022). *Compatibility between tuple, pair and tuple-like objects* [Compatibilidad entre tuplas, pares y objetos que pueden ser tratados como tuplas] (Doc. técnico no. P2165R3). ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee. Recuperado de <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2165r3.pdf>
- Kazakova, A. (15 de enero de 2024). *The C++ Ecosystem in 2023: Growth of C++20, Wider Adoption of Clang-based Tools, AI Gaining Developers' Confidence* [El ecosistema de C++ en 2023: crecimiento de C++20, mayor adopción de herramientas basadas en

Clang, la IA gana la confianza de los desarrolladores]. The CLion Blog. Recuperado de <https://blog.jetbrains.com/clion/2024/01/the-cpp-ecosystem-in-2023/>

Lemire, D., Kurz, N., y Rupp, C. (2018). Stream VByte: Faster byte-oriented integer compression [Stream VByte: Una compresión de enteros orientada a bytes más rápida]. *Information Processing Letters*, 130, 1-6. doi: 10.1016/j.ipl.2017.09.011

Li, W., Meng, N., Li, L., y Cai, H. (Mayo de 2021). Understanding Language Selection in Multi-Language Software Projects on GitHub [Entendiendo la selección de lenguaje en proyectos de software multilenguaje en GitHub]. *IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (pp. 256-257). Conferencia llevada a cabo en Melbourne, Australia. doi: 10.1109/ICSE-Companion52605.2021.00119

Lilis, Y., y Savidis, A. (2019). A Survey of Metaprogramming Languages [Un Estudio de Lenguajes de Metaprogramación]. *ACM Computing Surveys*, 52(6), 1–39. doi: 10.1145/3354584

Lilja, D. J. (2005). Measuring computer performance: A practitioner's guide [Medición del rendimiento de computadoras: Una guía para profesionales]. Cambridge: Cambridge University Press.

Lippman, S. B., Lajoie, J. y Moo, B. E. (2013). *C++ Primer* (5ta ed.). Boston, Massachusetts: Addison-Wesley.

Lynch J. (2021). Use Fast Data Algorithms [Use algoritmos de datos más rápidos]. Recuperado de: https://jolynch.github.io/posts/use_fast_data_algorithms/

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design* [Guía de un Artesano para la Estructura y el Diseño de Software]. Estados Unidos: Prentice Hall.

Márton G. y Porkoláb Z. (2010). C++ Compile-Time Reflection and Mock Objects [Reflexión en tiempo de compilación y objetos simulados en C++]. *Studia Universitatis Babes-Bolyai Informatica*, 55(1), 1-21. Recuperado de https://martong.github.io/compile-time-reflection_macs_2014.pdf

Maurer, J. (20 de junio de 2016). *constexpr if: A slightly different syntax* [constexpr if: Una sintaxis ligeramente diferente] (Doc. técnico no. P0292R2). ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee. Recuperado de <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0292r2.html>

Molero, X., Juiz, C. y Rodeño, M. J. (2004). *Evaluación y modelado del rendimiento de los sistemas informáticos*. Pearson Educación.

Morris, M. (2003). *Diseño Digital* (3ra ed.). México: Pearson Educación.

Naumann, A. (21 de marzo de 2016). *Variant: a type-safe union for C++17 (v7)* [Variantes: una unión de tipos segura para C++17 (v7)] (Doc. técnico no. P0088R2). ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee. Recuperado de <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0088r2.html>

- Pressman, R. S. (2010). *Ingeniería del software: un enfoque práctico* (7ma ed.). Ciudad de México, México: McGraw-Hill Interamericana Editores.
- Qi, Y. (Septiembre de 2017). Compile-time reflection, Serialization and ORM [Reflexión en tiempo de compilación, serialización y ORM]. CppCon, The C++ Conference. Congreso llevado a cabo en Bellevue, Washington, Estados Unidos.
- Qi, Y. (Septiembre de 2022). A Faster Serialization Library Based on Compile-time Reflection and C++20 [Una librería de serialización más rápida basada en reflexión en tiempo de compilación y C++20]. CppCon, The C++ Conference. Congreso llevado a cabo en Aurora, Colorado, Estados Unidos.
- Sommerville, I. (2005). *Ingeniería del software* (7ma ed.). Madrid, España: Pearson Educación.
- Soukup J. y Macháček P. (2014). *Serialization and Persistent Objects: Turning Data Structures into Efficient Databases* [Serialización y objetos persistentes: Transformando estructuras de datos en bases de datos eficientes]. Berlín, Alemania: Springer. doi: 10.1007/978-3-642-39323-5
- Stroustrup, B. (18 de noviembre de 2019). *C++ exceptions and alternatives* [Excepciones y alternativas en C++] (Doc. técnico no. P1947). ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee. Recuperado de <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1947r0.pdf>
- Stroustrup, B. (2007). *Evolving a language in and for the real world: C++ 1991-2006* [Evolucionando un lenguaje en y para el mundo real: C++ 1991-2006]. Texas A&M University, Texas, Estados Unidos. Recuperado de <https://stroustrup.com/hopl-almost-final.pdf>
- Stroustrup B. (2013). *The C++ Programming Language* [El lenguaje de programación C++] (4ta ed.). Massachusetts, Estados Unidos: Addison-Wesley.
- Sutton, A. (2017). *Wording paper: C++ extensions for Concepts* [Documento de redacción: Extensión de C++ para “Conceptos”] (Doc. técnico no. P0734R0). ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee. Recuperado de <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0734r0.pdf>
- Tanenbaum, A. S., y Austin, T. (2013). *Structured Computer Organization* [Organización Estructurada de Computadoras] (6ta ed.). New Jersey, Estados Unidos: Pearson.
- Tauro C., Ganesan N., Mishra S. y Bhagwat A. (2012). Object Serialization: A Study of Techniques of Implementing Binary Serialization in C++, Java and .NET [Serialización de objetos: un estudio de técnicas para implementar la serialización binaria en C++, Java y .NET]. *International Journal of Computer Applications*, 45(6), 25-29. doi: 10.5120/6785-9088

Apéndice A.

Uso de la Librería

A. 1 Dependencias

Para configurar y compilar la librería, es necesario contar con las siguientes dependencias instaladas:

1. Python $\geq 3.10.0$: Utilizado en los scripts de configuración.
2. Meson $\geq 1.5.1$: Sistema de construcción utilizado en el proyecto.
3. MSVC ≥ 14.20 , o GCC ≥ 8.0 , o Clang ≥ 9.0 : Compilador C++20.

A. 2 Configuración e Instalación

Habiendo instalado las dependencias y obtenido una copia del código fuente se ejecutan los siguientes comandos en el directorio raíz, según se muestra en la Figura 41:

```
# Setup the build dir `out`
> meson setup out/ --buildtype=release

# Change dir
> cd out/

# Configure `max_visit_members` (Optional)
out> meson configure -Dmax_visit_members=255

# Install
out> meson install
```

Figura 41. Comandos de configuración e instalación de la librería Teg. Autoría propia.

La opción `max_visit_members` se utiliza para definir el número máximo de miembros que pueden ser accedidos mediante las funciones de reflexión en tipos agregados. Por defecto, este valor está establecido en 255, pero se puede aumentar según las necesidades del proyecto. Es crucial que, después de configurar esta opción, se ejecute el comando `meson install`. Este comando generará las cabeceras necesarias para que la configuración de `max_visit_members` sea efectiva. El proceso de instalación copiará las cabeceras del proyecto en una ubicación por defecto, como `C:\include\teg` en Windows y `/include/teg` en Linux. Meson imprimirá en la salida estándar la ubicación exacta de estos archivos, y éstos estarán incluidos en la variable de entorno `PATH`.

A. 3 Serialización y Deserialización de Objetos

La Figura 42 y la Figura 43 muestran un caso típico de serialización y deserialización de objetos en una arquitectura cliente-servidor. El servidor se encarga de consultar y crear el objeto de factura, serializarlo a un archivo binario y enviarlo al cliente. El cliente recibe el archivo desde el servidor y lo deserializa. Nótese que, tanto el servidor como el cliente deben tener un esquema de datos compatible, de no ser el caso, la operación de deserialización fallará y lanzará una excepción que deberá ser atrapada por el programa cliente.

```

1  /// \file: server.cpp
2  #include "teg/teg.h"
3
4  // Define the schema.
5  class InvoiceLine {
6  public:
7      std::string description;
8      teg::u16 quantity;
9      teg::f64 discount;
10     teg::f64 price;
11     teg::f64 subtotal;
12 };
13
14 class Invoice {
15 public:
16     Date date = {};
17     std::string customer = {};
18     std::vector<InvoiceLine> lines = {};
19     teg::f64 total = {};
20 };
21
22 // Serialize.
23 Invoice invoice{};
24 invoice.date = Date(2021, 10, 1);
25 invoice.customer = "John Doe";
26 invoice.lines.push_back(InvoiceLine{"Product 1", 2, 0.0, 50.0, 100.0});
27 invoice.lines.push_back(InvoiceLine{"Product 2", 1, 0.0, 50.0, 50.0});
28 invoice.lines.push_back(InvoiceLine{"Product 3", 1, 0.0, 50.0, 50.0});
29 invoice.total = 200.0;
30
31 teg::byte_array archive{};
32 teg::serialize(archive, server_invoice).or_throw();
33
34 // Do something with the archive, send it over the network or store it on disk.
35 send_to_client(archive);
36 ...

```

Figura 42. Ejemplo de uso de la librería: serialización de una factura. Autoría propia.

En el diseño del esquema de datos, se ha optado por utilizar alias de tipos como `teg::f64` y `teg::u16`. Sin embargo, es posible emplear los tipos estándar del lenguaje, como `double` y `uint16_t`. Recomendamos la utilización de tipos fundamentales definidos explícitamente con un ancho de bit fijo. Por ejemplo, es preferible utilizar `int32_t` en lugar de `int`, ya que el tamaño en bytes de `int` puede variar según la arquitectura objetivo, mientras que `int32_t` garantiza una representación constante de bytes en todas las plataformas.

```

1  /// \file: client.cpp
2  #include "teg/teg.h"
3
4  // Define the same schema as the server-side.
5  class InvoiceLine {
6  public:
7      std::string description;
8      teg::u16 quantity;
9      teg::f64 discount;
10     teg::f64 price;
11     teg::f64 subtotal;
12 };
13
14 class Invoice {
15 public:
16     Date date = {};
17     std::string customer = {};
18     std::vector<InvoiceLine> lines = {};
19     teg::f64 total = {};
20 };
21
22 // Receive the data from server.
23 teg::byte_array archive = receive_from_server();
24
25 // Deserialize.
26 Invoice invoice{};
27 teg::deserialize(archive, client_invoice).or_throw();
28
29 // Do something with the invoice.
30 fiscal_print(invoice);
31 ...

```

Figura 43. Ejemplo de uso de la librería: deserialización de una factura. Autoría propia.

A.4 Codificación y Decodificación Explícita

```

1  /// \file: date.hpp
2  class Date {
3  public:
4      Date(uint16_t year, uint16_t month, uint16_t day)
5          : m_year(year), m_month(month), m_day(day) {}
6      auto get_year() const -> uint16_t { return m_year; }
7      auto get_month() const -> uint16_t { return m_month; }
8      auto get_day() const -> uint16_t { return m_day; }
9  private:
10     uint16_t m_year;
11     uint16_t m_month;
12     uint16_t m_day;
13 };

```



```

1  /// \file: date_serialization.cpp
2  #include "proj/date.h"
3  #include "teg/teg.h"
4
5  template <class F>
6  constexpr auto usr_serialized_size(F&& size, Date const& date) -> teg::u64 {
7      return size(teg::u16{}, teg::u16{}, teg::u16{});
8  }
9
10 template <class F>
11 constexpr auto usr_serialize(F&& encode, Date const& date) -> teg::error {
12     auto const result = encode(
13         (teg::u16) date.get_year(),
14         (teg::u16) date.get_month(),
15         (teg::u16) date.get_day());
16     if (teg::failure(result)) {
17         return result;
18     }
19     return {};// Success.
20 }
21
22 template <class F>
23 constexpr auto usr_deserialize(F&& decode, Date & date) -> teg::error {
24     struct {
25         teg::u16 year; teg::u16 month; teg::u16 day;
26     } data;
27
28     auto const result = decode(data.year, data.month, data.day);
29     if (teg::failure(result)) {
30         return result;
31     }
32     date = Date(data.year, data.month, data.day);
33     return {};// Success.
34 }
```

Figura 44. Ejemplo de uso de la librería: codificación explícita definida por el usuario de una clase no agregada. Autoría propia.

La Figura 44 muestra un ejemplo de implementación de una codificación explícita para una clase no agregada `Date`. En algunas circunstancias se necesita trabajar con clases que no pueden ser implícitamente serializadas por la librería. Para solucionar esos casos, se puede definir tres funciones libres como se muestra en la Figura 44, `usr_serialize_size`, `usr_serialize` y `usr_deserialize`. La lógica de codificación y decodificación es implementada a través de una función genérica que es pasada como argumento a las funciones libres. Esto actúa como una capa de abstracción y permite que se siga la secuencia de bytes del archivo binario.

A. 5 Configuración del Archivo Binario

Es posible configurar el formato de archivo binario como se muestra en la Figura 45. Es importante notar que tanto el proceso de serialización como el de deserialización deben tener una configuración compatible, de lo contrario se lanzará una excepción. En el caso de ejemplo, en el primer cuadro se muestra que se estableció un orden de bytes big-endian y que se fuerza la compresión de números enteros, siendo una configuración personalizada. En el segundo cuadro se muestra la configuración de modo compacto, usada en múltiples ocasiones en la investigación. Más detalles sobre todas las opciones disponibles se pueden consultar en el Capítulo 4.

```

1 constexpr teg::options custom_mode =
2     teg::options::big_endian | teg::options::force_varint;
3
4 teg::byte_array archive;
5 std::string str0 = "A very important text";
6 teg::serialize<custom_mode>(archive, str0).or_throw();
7
8 ...
9
10 std::string str1;
11 teg::deserialize<custom_mode>(archive, str1).or_throw();

12
13 constexpr teg::options compact_mode =
14     teg::options::compact | teg::options::no_compression;
15
16 teg::byte_array archive;
17 std::string str0 = "A very important text";
18 teg::serialize<compact_mode>(archive, str0).or_throw();
19
20 ...
21
22 teg::byte_array archive;
23 std::string str1;
24 teg::deserialize<compact_mode>(archive, str1).or_throw();

```

Figura 45. Ejemplo de uso de la librería: configuración del archivo binario. Autoría propia.

Apéndice B.

Interfaz de Programación

B. 1 Cabeceras

Tabla 9

Cabeceras de la librería de serialización y deserialización binaria Teg. Autoría propia.

Cabecera	Descripción
<code>`teg.h`</code>	Inclusión de todas las cabeceras. Este archivo puede utilizarse cuando se desee compilar el código fuente como una librería de cabecera única (del inglés, <i>single header library</i>).
<code>`def.h`</code>	Definición de alias de tipos fundamentales (enteros y reales) con un ancho fijo de bit. Estos alias aseguran que los tipos de datos built-in tengan siempre el mismo tamaño en memoria, sin importar en qué entorno se estén utilizando, ya que normalmente el tamaño de estos tipos puede variar según el compilador que se esté usando.
<code>`alignment.h`</code>	Conceptos y algoritmos utilizados para determinar la alineación de los tipos de datos.
<code>`endian.h`</code>	Funciones y conceptos para el manejo del ordenamiento de bytes en el formato de archivo binario.
<code>`buffer.h`</code>	Definición de arreglos dinámicos y estáticos de bytes, utilizados como buffers por defecto en los procesos de serialización y deserialización.
<code>`core_concepts.h`</code>	Conceptos fundamentales de la librería: arreglos de C, agregados, tuplas, tiposopcionales, variantes y punteros propietarios.
<code>`container_concepts.h`</code>	Conceptos de contenedores según los requerimientos del estándar ISO/IEC 14882 (2020).
<code>`serialization_concepts.h`</code>	Conceptos de serialización y deserialización de objetos.
<code>`encoder.h`</code>	Clases de codificación de objetos. Implementación de codificación de objetos hacia memoria y hacia flujos de datos de salida (como archivos en disco y otros buffers).
<code>`decoder.h`</code>	Clases de decodificación de objetos. Implementación de decodificación de objetos hacia memoria y hacia flujos de datos de entrada (como archivos en disco y otros buffers).
<code>`serialization.h`</code>	Funciones de serialización: Estas funciones permiten serializar objetos sin requerir al usuario la creación de codificadores, abstrayendo los detalles de serialización y ofreciendo una interfaz más sencilla y fácil de utilizar.
<code>`deserialization.h`</code>	Funciones de deserialización: Estas funciones permiten deserializar objetos sin requerir al usuario la creación de instancias de clases de decodificación, abstrayendo los detalles complejos del proceso y ofreciendo una interfaz más sencilla y fácil de utilizar.
<code>`members_count.h`</code>	Definición de la función genérica para el conteo de variables miembros de un tipo agregado.
<code>`members_equal.h`</code>	Definición de la función genérica de comparación de igualdad de variables miembros de tipos agregados.
<code>`members_get.h`</code>	Definición de la función genérica para la obtención de una referencia a la variable miembro de un objeto instancia de un tipo agregado dado un índice válido.

Cabecera	Descripción
`members_tie.h`	Función genérica que retorna una tupla de referencias a las variables miembros de un objeto instancia de un tipo agregado.
`members_visitor.h`	Función genérica de aplicación de una función anónima (función <i>lambda</i>) pasando como argumentos todas las variables miembros de un objeto instancia de un tipo agregado.
`schema.h`	Codificación y decodificación genérica de esquemas de datos y sus manejos de versiones.
`compatible.h`	Clases y funciones para el soporte de la retrocompatibilidad de versiones de esquemas de datos.
`varint.h`	Implementación de clases de enteros de longitud variable LEB128.
`varint_serialization.h`	Definición de funciones explícitas de serialización y deserialización de enteros de longitud variable LEB128.
`util.h`	Funciones auxiliares misceláneas.
`c_array.h`	Funciones auxiliares para copiar arreglos multidimensionales de C.
`md5.h`	Implementación de funciones de digestión de datos en tiempo de compilación utilizando el algoritmo de hash MD5.
`index_table.h`	Funciones auxiliares para manipular índices en tiempo de ejecución de tipos variantes y convertirlos a índices constantes accesibles en tiempo de compilación, a través de tablas de indirección de memoria.
`fixed_string.h`	Definición de una clase de cadenas de caracteres de tamaño fijo utilizables en tiempo de compilación (en contextos constantes).
`options.h`	Definición y manejo de las opciones del archivo binario.
`error.h`	Definición y manejo de errores de la librería: códigos de retorno y lanzamientos de excepciones.
`version.h`	Definición de versión y palabra mágica de la librería.

B. 2 Módulos

Tabla 10

Módulos de la librería de serialización y deserialización binaria Teg. Autoría propia.

Módulo	Cabeceras
Módulo de Reflexión Estática	`alignment.h`, `core_concepts.h`, `container_concepts.h`, `members_count.h`, `members_equal.h`, `members_get.h`, `members_tie.h` y `members_visitor.h`.
Módulo de Serialización de Objetos de objetos	`serialization_concepts.h`, `buffer.h`, `error.h`, `options.h`, `endian.h`, `fixed_string.h`, `md5.h`, `schema.h`, `version.h`, `compatible.h`, `c_array.h`, `encoder.h`, `decoder.h`, `serialization.h` y `deserialization.h`.
Módulo de Enteros de Longitud Variable	`varint.h` y `varint_serialization.h`.
Módulo global	`def.h` y `util.h`

B. 2. 1 Módulo de Reflexión Estática.

Tabla 11

*Conceptos, clases y funciones definidas en el módulo de reflexión en tiempo de compilación.
Autoría propia.*

Símbolo	Descripción
<code>template <class T> concept fundamental;</code>	Verifica si `T` es un tipo fundamental (tipos integrales, flotantes, void, nullptr_t).
<code>template <class T> concept is_enum</code>	Verifica si `T` es un tipo enumerado.
<code>template <class T> concept is_class</code>	Verifica si `T` es una clase o una unión.
<code>template <class T> concept character</code>	Verifica si `T` es un tipo de carácter (char, wchar_t, char8_t, char16_t, char32_t).
<code>template <class T> concept trivially_copyable</code>	Verifica si `T` es un tipo trivialmente copiable.
<code>template <class T> concept standard_layout</code>	Verifica si `T` tiene un diseño estándar (del inglés, <i>standard layout</i>).
<code>template <class T> concept c_array</code>	Verifica si `T` es un arreglo de C.
<code>template <class T> concept aggregate</code>	Verifica si `T` es un tipo agregado (clase agregada).
<code>template <class T, std::size_t I> concept tuple_element</code>	Dada una tupla `T`, verifica que se puede acceder al elemento de dicha tupla en la posición `I`.
<code>template <class T> concept tuple_size</code>	Verifica si `T` tiene un tamaño de tupla definido en tiempo de compilación.
<code>template <class T> concept tuple</code>	Verifica si `T` es un tipo de tupla (<i>tuple-like</i>).
<code>template <class T> concept pair</code>	Verifica si `T` es un tipo de par (<i>pair-like</i>).
<code>template <class T> concept optional</code>	Verifica si `T` es un tipo opcional.
<code>template <class C, class T> concept container_element;</code>	Determina si un contenedor `C` puede contener elementos de tipo `T`.
<code>template <class C> concept container;</code>	Determina si `C` es un contenedor según los requerimientos especificados por el estándar.
<code>template <class C> concept reversible_container;</code>	Determina si `C` es un contenedor reversible según los requerimientos especificados por el estándar.
<code>template <class C> concept sized_container;</code>	Determina si `C` es un contenedor y que tiene un tamaño definido (redimensionable o no).
<code>template <class C> concept clearable_container;</code>	Determina si `C` es un contenedor que se puede vaciar.
<code>template <class C> concept resizable_container;</code>	Determina si `C` es un contenedor que se puede redimensionar.
<code>template <class C> concept reservable_container;</code>	Determina si `C` es un contenedor en el que se puede reservar la alocación de memoria.
<code>template <class C> concept random_access_container;</code>	Determina si `C` es un contenedor de acceso aleatorio según los requerimientos especificados por

Símbolo	Descripción
<code>template <class C> concept contiguous_container;</code>	el estándar.
<code>template <class C> concept fixed_size_container;</code>	Determina si `C` es un contenedor contiguo.
<code>template <class C> concept inplace_constructing_container;</code>	Determina si `C` es un contenedor de tamaño fijo (su tamaño es conocido en tiempo de compilación).
<code>template <class C> concept back_inplace_constructing_container;</code>	Determina si `C` es un contenedor que puede construir elementos en su lugar.
<code>template <class C> concept front_inplace_constructing_container;</code>	Determina si `C` es un contenedor que puede construir elementos en su lugar al final.
<code>template <class C> concept range_constructing_container;</code>	Determina si `C` es un contenedor que puede construir elementos a partir de un rango.
<code>template <class C> concept associative_container;</code>	Determina si `C` es un contenedor asociativo según los requerimientos especificados por el estándar.
<code>template <class C> concept set_container;</code>	Determina si `C` es un contenedor asociativo de conjuntos (contenedor de claves).
<code>template <class C> concept map_container;</code>	Determina si `C` es un contenedor asociativo clave-valor (como las tablas hash).
<code>template <class T> requires aggregate<T> constexpr members_count() -> std::size_t;</code>	Función de reflexión: Retorna la cantidad de variables miembros declaradas en la clase agregada `T`.
<code>template <class T> concept accesible_aggregate;</code>	Prueba que `T` es una clase agregada y que tiene una cantidad de variables miembros menor o igual al número permitido por la implementación de la librería para los procesos de reflexión.
<code>template <class F, class T> requires accesible_aggregate<T> constexpr visit_members(F&& f, T&& t) -> decltype(auto);</code>	Función de reflexión: Invoca la función (o objeto invocable) `f` pasando como argumentos todos los variables miembros de `t`. Se retorna el resultado de llamar `f`.
<code>template <class T> requires accesible_aggregate<T> constexpr tie_members(T& t) -> decltype(auto);</code>	Función de reflexión: Ata un objeto `t` a una tupla de referencias hacia sus variables miembros. Retorna la tupla de referencias.
<code>template <std::size I, class T> requires accesible_aggregate<T> && I < members_count<T>() constexpr get_member(T& t) -> decltype(auto);</code>	Función de reflexión: Retorna una referencia a la variable miembro del objeto `t` que ocupa la posición `I` (en orden de declaración de la clase `T`).
<code>template <class T1, class T2> constexpr memberwise_equal(T1& l, T2& r) -> bool;</code>	Función de reflexión: Retorna la comparación de igualdad entre los objetos `l` y `r`, comparando de forma recursiva a nivel de variables miembros.
<code>template <class T> concept packet_layout;</code>	Determina si el tipo `T` posee una estructura empaquetada (no tiene bytes de relleno en su representación de objeto).

B. 2. 2 Módulo de Serialización de Objetos.

Tabla 12

Conceptos, clases y funciones definidas y exportadas en el módulo de serialización y deserialización binaria de objetos. Autoría propia.

Símbolo	Descripción
<code>enum class options : u32;</code>	Enumeración de las opciones del formato de archivo binario.
<code>struct error;</code>	Clase que abstrae los resultados de los procesos de serialización utilizando códigos de errores de la librería estándar.
<code>constexpr success(error e) -> bool;</code>	Función de utilidad: retorna verdadero en caso de que no haya ocurrido ningún error.
<code>constexpr failure(error e) -> bool;</code>	Función de utilidad: retorna verdadero en caso de que haya ocurrido algún error.
<code>template <class T></code> <code>concept endian_neutral;</code>	Detecta si `T` es un tipo neutral con respecto al ordenamiento de bytes (es indiferente el orden).
<code>template <class T></code> <code>concept endian_neutral_container;</code>	Detecta si `T` es un contenedor cuyos elementos son neutrales con respecto al ordenamiento de bytes.
<code>template <class T></code> <code>concept endian_neutral_c_array;</code>	Detecta si `T` es un arreglo de C cuyos elementos son neutrales con respecto al ordenamiento de bytes.
<code>template <class T, options Opt></code> <code>concept endian_swapping_required;</code>	Detecta si el tipo `T` requiere un cambio de orden de bytes dada las opciones `Opt` .
<code>template <class T></code> <code>concept byte;</code>	Verifica que `T` es un tipo de dato utilizable como byte.
<code>template <class T></code> <code>concept byte_buffer;</code>	Verifica que `T` es un arreglo dinámico o estático contiguo de bytes.
<code>template <class T></code> <code>concept serializable;</code>	Verifica si `T` es un tipo serializable con el formato de archivo binario diseñado en la investigación.
<code>template <class T></code> <code>concept user_defined_encoding;</code>	Prueba que `T` tiene definida una codificación explícita (codificación brindada por el usuario).
<code>template <class T></code> <code>concept memory_copyable;</code>	Detecta si `T` es un tipo copiable de manera segura; su representación de objeto es contigua y no contiene bytes de relleno.
<code>template <class T></code> <code>concept trivially_serializable;</code>	Detecta si `T` es un tipo trivialmente serializable; si puede copiarse en memoria y no requiere conversión de orden de bytes ni conversión hacia enteros de longitud variables.
<code>template <class T></code> <code>concept non_trivially_serializable;</code>	Detecta si `T` es un tipo no trivialmente copiable. Por ejemplo, los tipos con codificación definida por el usuario no son trivialmente copiables.
<code>template <class T></code> <code>concept trivially_serializable_container;</code>	Detecta si `T` es un contenedor trivialmente copiable.
<code>class schema_analyzer;</code>	Clase dedicada al análisis de los esquemas de datos a serializar.
<code>class schema_encoder;</code>	Clase encargada de la codificación de los esquemas de datos a serializar para las verificaciones de formato.

Símbolo	Descripción
<code>template <serializable... T> constexpr version_count() -> u64;</code>	Función que retorna el número de versiones existentes en un esquema de datos `T` .
<code>template <u64 V, serializable... T> constexpr schema() -> decltype(auto);</code>	Función que retorna una cadena de caracteres fija con la codificación del esquema de datos `T` en su versión `V` .
<code>template <serializable... T> constexpr schema_hash_table() -> decltype(auto);</code>	Función que crea y retorna una tabla con la digestión MD5 de todas las codificaciones de las versiones del esquema de datos `T` .
<code>template <class T> concept writer;</code>	Verifica que la clase `T` modela de forma válida la escritura de datos binarios.
<code>enum class buffer_safety_policy;</code>	Política de seguridad del buffer de datos en memoria.
<code>template <buffer_safety_policy P, class Buf> class buffer_writer;</code>	Clase de escritura de datos binarios hacia un buffer en memoria.
<code>class stream_writer;</code>	Clase de escritura de datos hacia un flujo de datos (como por ejemplo, un archivo en disco).
<code>template <options Opt, class Writer> class encoder;</code>	Clase de codificación de objetos.
<code>template <class T> concept reader;</code>	Verifica que la clase `T` modela de forma válida la lectura de datos binarios.
<code>template <buffer_safety_policy P, class Buf> class buffer_reader;</code>	Clase de lectura de datos binarios desde un buffer en memoria.
<code>class stream_reader;</code>	Clase de lectura de datos desde un flujo de datos (como por ejemplo, un archivo en disco).
<code>template <options Opt, class Writer> class decoder;</code>	Clase de codificación de objetos.
<code>template <options Opt, class Buf, class... T> constexpr serialize(Buf& buf, T const&... obj) -> error;</code>	Función de serialización de objetos hacia un arreglo de bytes.
<code>template <options Opt, class C, class... T> constexpr serialize (std::basic_ostream<C>& stream, T const&... obj) -> error;</code>	Función de serialización de objetos hacia un flujo de datos (como un archivo en disco).
<code>template <options Opt, class Buf, class... T> constexpr deserialize(Buf& buf, T&... obj) -> error;</code>	Función de deserialización de objetos dado un arreglo de bytes con un archivo binario válido..
<code>template <options Opt, class C, class... T> constexpr deserialize (std::basic_istream<C>& stream, T&... obj) -> error;</code>	Función de deserialización de objetos dado un flujo de datos de entrada (ej. un archivo en disco) con un archivo binario válido.

B. 2. 3 Módulo de enteros de longitud variable.

Tabla 13

Conceptos, clases y funciones definidas y exportadas en el Módulo de Enteros de Longitud Variable. Autoría propia.

Símbolo	Descripción
<code>class zigzag;</code>	Clase estática de utilidad para la codificación y decodificación ZigZag.
<code>class uleb128;</code>	Clase estática de utilidad para la codificación y decodificación ULEB-128.
<code>template <class T></code> <code>class varint;</code>	Clase genérica de envoltura de números de longitud variable.
<code>using vint32 = varint<i32>;</code> <code>using vint64 = varint<i64>;</code> <code>using vuint32 = varint<u32>;</code> <code>using vuint64 = varint<u64>;</code>	Alias de tipos para el manejo de números de longitud variable con almacenamiento en memoria de 32-bits y 64-bits, con signo y sin signo.

B. 2. 4 Módulo global.

El módulo global hace referencia a aquellos símbolos que son definidos bajo el nombre de espacio “teg”, y que no pertenecen a un módulo específico. Estos símbolos son comunes entre todos los módulos y son de uso general; incluyen constantes para obtener la versión de la librería y clases útiles para manipular cadenas de texto y números enteros en tiempo de compilación, como se muestra en la Figura 14. Este módulo no representa un objetivo directo de la investigación, sino que es el resultado de la organización de directorios y archivos de código fuente C++20.

Tabla 14

Conceptos, clases y funciones definidas y exportadas en el módulo global. Autoría propia.

Símbolo	Descripción
<code>#define TEG_VERSION_MAJOR</code> <code>#define TEG_VERSION_MINOR</code> <code>#define TEG_VERSION_PATCH</code> <code>#define TEG_VERSION</code>	Directivas de preprocesador (macros) que exponen la versión actual de la librería.
<code>typedef u8; typedef i8;</code> <code>typedef u16; typedef i16;</code> <code>typedef u32; typedef i32;</code> <code>typedef u64; typedef i64;</code> <code>typedef f32; typedef f64;</code> <code>typedef isize; typedef usize;</code>	Alias para tipos de datos fundamentales que aseguran consistencia en el tamaño de bits en cualquier plataforma objetivo, facilitando así la portabilidad del código.
<code>template <class C, std::size_t N, class TT></code> <code>class basic_fixed_string;</code>	Clase genérica: cadena de caracteres en tiempo de compilación.

Símbolo	Descripción
<code>template <std::size_t N> using fixed_string;</code>	Alias para especificar el tipo de carácter utilizado en las cadenas de texto en tiempo de compilación. Carácteres UTF-8, UTF-16 y UTF-32.
<code>template <std::size_t N> using fixed_u8string;</code>	
<code>template <std::size_t N> using fixed_u16string;</code>	
<code>template <std::size_t N> using fixed_u32string;</code>	
<code>template <class C, std::size_t N> constexpr make_fixed_string(const C(&str)[N]) -> basic_fixed_string<C, N-1></code>	Función de utilidad para crear cadenas de caracteres en tiempo de compilación dado un arreglo de caracteres.
<code>struct md5::digest;</code>	Estructura de datos usada para representar los 16 bytes de una digestión MD5.
<code>constexpr md5::hash_u32(std::string_view msg) -> u32;</code>	Algoritmo de digestión hash MD5 en tiempo de compilación. Retorna los primeros 4 bytes.
<code>constexpr md5::hash_u64(std::string_view msg) -> u64;</code>	Algoritmo de digestión hash MD5 en tiempo de compilación. Retorna los primeros 8 bytes.
<code>constexpr md5::hash_u128(std::string_view msg) -> md5::digest;</code>	Algoritmo de digestión hash MD5 en tiempo de compilación. Retorna los primeros 8 bytes.
<code>constexpr md5::hash(std::string_view msg) -> fixed_string<32>;</code>	Algoritmo de digestión hash MD5 en tiempo de compilación. Retorna los 16 bytes de digestión en una cadena de texto con los bytes en formato hexadecimal.

Apéndice C.

Detalles de Implementación

C. 1 Tipos serializables

C. 1. 1 Tipos de datos built-in.

```

1 template <class T>
2 concept fundamental = std::is_fundamental_v<T>;
3
4 template <class T>
5 concept is_enum = std::is_enum_v<T>;
6
7 template <class T>
8 concept character =
9     std::is_same_v<std::remove_cv_t<T>, char>
10    || std::is_same_v<std::remove_cv_t<T>, signed char>
11    || std::is_same_v<std::remove_cv_t<T>, unsigned char>
12    || std::is_same_v<std::remove_cv_t<T>, wchar_t>
13    || std::is_same_v<std::remove_cv_t<T>, char8_t>
14    || std::is_same_v<std::remove_cv_t<T>, char16_t>
15    || std::is_same_v<std::remove_cv_t<T>, char32_t>;
16
17 template <class T>
18 concept trivially_copyable = std::is_trivially_copyable_v<T>;
19
20 template <class T>
21 concept standard_layout = std::is_standard_layout_v<T>;
22
23 template <class T>
24 concept unbounded_c_array = std::is_unbounded_array_v<T>;
25
26 template <class T>
27 concept c_array = std::is_array_v<T>;
28
29 template <class T>
30 concept aggregate = std::is_aggregate_v<T> && !unbounded_c_array<T>;

```

Figura 46. Implementación de los Conceptos que modelan tipos de datos fundamentales, clases, arreglos y las propiedades de estos tipos. Autoría propia.

C. 1.2 Contenedores.

```

1 template <class C>
2 concept container =
3     /// Member types.
4     container_element<C, typename C::value_type>
5     // Reference types.
6     && std::same_as<typename C::reference, typename C::value_type&>
7     && std::same_as<typename C::const_reference, typename C::value_type const&>
8     // Iterator types.
9     && std::forward_iterator<typename C::iterator>
10    && std::forward_iterator<typename C::const_iterator>
11    && std::same_as<std::iter_value_t<typename C::iterator>, typename
12 C::value_type>
13    && std::same_as<std::iter_value_t<typename C::const_iterator>, typename
14 C::value_type>
15    && std::convertible_to<typename C::iterator, typename C::const_iterator>
16    // Difference type.
17    && std::signed_integral<typename C::difference_type>
18    && std::same_as<
19        typename C::difference_type,
20        typename std::iterator_traits<typename C::iterator>::difference_type>
21    && std::same_as<
22        typename C::difference_type,
23        typename std::iterator_traits<typename
24 C::const_iterator>::difference_type>
25    // Size type.
26    && std::unsigned_integral<typename C::size_type>
27    && (std::in_range<typename C::size_type>(std::numeric_limits<typename
28 C::difference_type>::max()))
29     /// Member functions.
30    && requires (std::remove_cv_t<C> a, std::remove_cv_t<C> const b) {
31        { a.begin() } -> std::same_as<typename C::iterator>;
32        { a.end() } -> std::same_as<typename C::iterator>;
33        { b.begin() } -> std::same_as<typename C::const_iterator>;
34        { b.end() } -> std::same_as<typename C::const_iterator>;
35        { a.cbegin() } -> std::same_as<typename C::const_iterator>;
36        { a.cend() } -> std::same_as<typename C::const_iterator>;
37        { a.max_size() } -> std::same_as<typename C::size_type>;
38        { a.empty() } -> std::convertible_to<bool>;
39    };

```

Figura 47. Implementación del Concepto de contenedor `teg::conceptos::container` según los requerimientos especificados por el estándar del lenguaje ISO/IEC 14882:2020. Autoría propia.

```

1 template <class C>
2 concept sized_container = container<C> && requires(C const a) {
3     { a.size() } -> std::same_as<typename C::size_type>; };
4
5 template <class C>
6 concept reservable_container = sized_container<C>
7     && requires(C a, typename C::size_type const n) { a.reserve(n); };
8
9 template <class C>
10 concept random_access_container =
11     reversible_container<C> && sized_container<C>
12     && std::random_access_iterator<typename C::iterator>
13     && std::random_access_iterator<typename C::const_iterator>
14     && requires(
15         std::remove_cv_t<C> a, std::remove_cv_t<C> const b,
16         typename C::size_type const i
17     ) {
18         { a[i] } -> std::same_as<typename C::reference>;
19         { b[i] } -> std::same_as<typename C::const_reference>;
20     };
21
22 template <class C>
23 concept contiguous_container =
24     random_access_container<C>
25     && std::contiguous_iterator<typename C::iterator>
26     && std::contiguous_iterator<typename C::const_iterator>
27     && std::contiguous_iterator<typename C::pointer>
28     && std::contiguous_iterator<typename C::const_pointer>
29     && std::convertible_to<typename C::pointer, typename C::const_pointer>
30     && requires(std::remove_cv_t<C> a, std::remove_cv_t<C> const b) {
31         { a.data() } -> std::same_as<typename C::pointer>;
32         { b.data() } -> std::same_as<typename C::const_pointer>;
33     };
34
35 template <typename C>
36 concept fixed_size_container = contiguous_container<C> &&
37     std::integral_constant<std::size_t, C{}.size()>::value > 0;

```

Figura 48. Implementación de Conceptos de contenedores específicos según las propiedades inherentes de estos como `teg::conceptos::contiguos_container` y `teg::conceptos::fixed_size_container`. Autoría propia.

C. 1. 3 Tuplas.

```

1 template<class T, std::size_t I>
2 concept tuple_element = requires(T t) {
3     typename std::tuple_element_t<I, std::remove_const_t<T>>;
4     { std::get<I>(t) } -> std::convertible_to<const std::tuple_element_t<I,T>&>;
5 };
6
7 template <class T>
8 concept tuple_size = requires { typename std::tuple_size<T>::type; };
9
10 template<class T>
11 concept tuple = tuple_size<T>
12     && std::derived_from<
13         std::tuple_size<T>,
14         std::integral_constant<std::size_t, std::tuple_size_v<T>>
15     >
16     && []<std::size_t... I>(std::index_sequence<I...>) constexpr {
17         return (tuple_element<T, I> && ...);
18     }(std::make_index_sequence<std::tuple_size_v<T>>());

```

Figura 49. Implementación de los Conceptos para verificar tuplas (e interfaces que se comportan como tuplas). Autoría propia.

C. 1. 4 Tipos opcionales.

```

1 template <class T>
2 concept optional = requires(T a) {
3     typename T::value_type;
4     a.value();
5     a.has_value();
6     a.operator bool();
7     a.operator*();
8 };

```

Figura 50. Implementación del Concepto `teg::concepts::optional` usado para detectar tiposopcionales. Autoría propia.

C. 1. 5 Variantes.

```

1 template <class T>
2 constexpr inline bool is_variant_v = false;
3
4 template <class... T>
5 constexpr inline bool is_variant_v<std::variant<T...>> = true;
6
7 template <class T>
8 concept variant = is_variant_v<T>;

```

Figura 51. Implementación del Concepto `teg::concepts::variant` usado para comprobar que un tipo es exactamente la clase estándar `std::variant`. Autoría propia.

C. 1. 6 Punteros propietarios.

```

1 template <class T>
2 struct is_unique_ptr : std::false_type {};
3 template <class T>
4 struct is_unique_ptr<std::unique_ptr<T, std::default_delete<T>>>
5     : std::true_type {};
6 template <class T>
7 constexpr bool is_unique_ptr_v = is_unique_ptr<T>::value;
8
9 template <class T>
10 struct is_shared_ptr : std::false_type {};
11 template <class T>
12 struct is_shared_ptr<std::shared_ptr<T>> : std::true_type {};
13 template <class T>
14 constexpr bool is_shared_ptr_v = is_shared_ptr<T>::value;
15
16 template <class T>
17 concept unique_ptr = is_unique_ptr_v<T>;
18
19 template <class T>
20 concept shared_ptr = is_shared_ptr_v<T>;
21
22 template <class T>
23 concept owning_ptr = unique_ptr<T> || shared_ptr<T>;

```

Figura 52. Implementación de los Conceptos de punteros `teg::concepts::unique_ptr` , `teg::concepts::shared_ptr` y `teg::concepts::owning_ptr`. Autoría propia.

C. 1. 6 Tipos compatibles.

```

1 template <class T>
2 constexpr inline bool is_compatible_v = false;
3
4 template <class T, u64 V>
5 constexpr inline bool is_compatible_v<compatible<T, V>> = true;
6
7 template <class T>
8 concept compatible = is_compatible_v<T>;

```

Figura 53. Implementación del Concepto `teg::concepts::compatible` usado para detectar clases de retrocompatibilidad en el formato de archivo binario. Autoría propia.

C. 1. 7 Codificación definida por el usuario.

```

1 template <class T>
2 concept user_defined_encoding = requires (
3     T const& const_type, T & type,
4     std::function<teg::u64()> && size_func,
5     std::function<teg::error()>&& encode_func)
6 {
7     { encoded_size(size_func, const_type) } -> std::convertible_to<u64>;
8     { encode(encode_func, const_type) }      -> std::same_as<teg::error>;
9     { decode(encode_func, type) }           -> std::same_as<teg::error>;
10 };

```

Figura 54. Implementación del Concepto `teg::concepts::compatible` usado para detectar clases de retrocompatibilidad en el formato de archivo binario. Autoría propia.

C. 2 Funciones de Reflexión en Tiempo de Compilación

C. 2. 1 Conteo de variables miembros en clases agregadas.

```

1  struct generic_type {
2      template <class T> constexpr operator T() const noexcept; };
3  struct generic_nullptr {
4      constexpr operator std::nullptr_t() const noexcept; };
5
6  template <class T, class P, class... A>
7  concept aggregate_initializable = aggregate<T>
8      && requires { T{ std::declval<A>() }..., { std::declval<P>() } }; };
9
10 template <class T, class... A> requires aggregate<T>
11 constexpr auto members_count() -> std::size_t {
12     if constexpr (aggregate_initializable<T, generic_type, A...>) {
13         return members_count<T, A..., generic_type>();
14     }
15     else if constexpr (aggregate_initializable<T, generic_nullptr, A...>) {
16         return members_count<T, A..., generic_nullptr>();
17     }
18     else { return sizeof...(A); }
19 }
20
21 template <class T> requires aggregate<T>
22 constexpr std::size_t members_count_v = members_count<T>();

```

Figura 55. Implementación de la función genérica de reflexión `teg::members_count`. Autoría propia.

C. 2. 2 Visita a variables miembros en clases agregadas.

```

1  constexpr static auto max_visit_members = 255;
2
3  template <class T>
4  concept accesible_aggregate = aggregate<T> &&
5      members_count_v<T> <= max_visit_members;
6
7  template <class F, class T>
8  constexpr decltype(auto) visit_members_impl(F&& f, T&& t,
9      std::integral_constant<std::size_t, 1>) {
10     auto&& [_001] = t;
11     return f(_001);
12 }
13
14 template <class F, class T>
15 constexpr decltype(auto) visit_members_impl(F&& f, T&& t,
16 std::integral_constant<std::size_t, 2>) {
17     auto&& [_001, _002] = t;
18     return f(_001, _002);
19 }
20
21 template <class F, class T>
22 constexpr decltype(auto) visit_members_impl(F&& f, T&& t,
23 std::integral_constant<std::size_t, 3>) {
24     auto&& [_001, _002, _003] = t;
25     return f(_001, _002, _003);
26 }
27
28 ... // 255 functions implementing "visit_members_impl".
29
30 template<class F, class T> requires accesible_aggregate<T>
31 constexpr decltype(auto) visit_members(F&& f, T&& t) noexcept {
32     return visit_members_impl(
33         std::forward<F>(f), std::forward<T>(t),
34         std::integral_constant<
35             std::size_t,
36             members_count_v<std::remove_cvref_t<T>>>{}
37     );
38 }
39 }
```

Figura 56. Implementación de la función genérica de reflexión `teg::visit_members` . Autoría propia.

C. 2. 3 Conversión de una clase agregada a una tupla.

```

1 template <class T>
2 constexpr decltype(auto) tie_members_impl(T& t,
3     std::integral_constant<std::size_t, 1> {
4     auto& [_001] = t;
5     return std::tie(_001);
6 }
7
8 template <class T>
9 constexpr decltype(auto) tie_members_impl(T& t,
10    std::integral_constant<std::size_t, 2> {
11    auto& [_001, _002] = t;
12    return std::tie(_001, _002);
13 }
14
15 template <class T>
16 constexpr decltype(auto) tie_members_impl(T& t,
17     std::integral_constant<std::size_t, 3> {
18     auto& [_001, _002, _003] = t;
19     return std::tie(_001, _002, _003);
20 }
21
22 ... // 255 functions implementing "tie_members_impl".
23
24 template <class T> requires accesible_aggregate<T>
25 constexpr auto tie_members(T& t) -> decltype(auto) {
26     return tie_members_impl(
27         t,
28         std::integral_constant<std::size_t, members_count_v<T>>()
29     );
30 }
```

Figura 57. Implementación de la función genérica de reflexión `teg::tie_members`. Autoría propia.

C. 2. 4 Acceso por índice a variables miembros de una clase agregada.

```

1 template <std::size_t I, class T>
2 requires accesible_aggregate<T> && I <= members_count_v<T>
3 constexpr auto get_member(T&& t) -> decltype(auto) {
4     return std::get<I>(tie_members(t));
5 }
```

Figura 58. Implementación de la función de reflexión `teg::get_member` . Autoría propia.

C. 2. 5 Comparación de igualdad miembro a miembro en clases agregadas.

```

1 template <class T1, class T2>
2 requires (std::equality_comparable_with<T1, T2>)
3     || (accesible_aggregate<T1> && accesible_aggregate<T2>)
4 constexpr auto memberwise_equal(T1 const& left, T2 const& right) -> bool {
5     if constexpr (std::equality_comparable_with<T1, T2>) {
6         if constexpr (c_array<T1>) {
7             return std::ranges::equal(left, right);
8         }
9         else {
10             return left == right;
11         }
12     }
13     else if constexpr (members_count_v<T1> != members_count_v<T2>) {
14         return false;
15     }
16     else {
17         auto const equal =
18             [&]<std::size_t... I>(std::index_sequence<I...>) constexpr -> bool {
19             return (
20                 memberwise_equal(get_member<I>(left), get_member<I>(right))
21                 && ...
22             );
23         };
24         return equal(std::make_index_sequence<members_count_v<T1>>{});
25     }
26 }
```

Figura 59. Implementación de la función genérica de reflexión `teg::memberwise_equal`. Autoría propia.

C. 2. 5 Detección de estructuras empaquetadas.

```

1 template <class T>
2 concept bit_castable = requires {
3     std::bit_cast<T>(std::array<std::byte, sizeof(T)>{});
4 }
5 && is_constexpr_friendly([] {
6     std::ignore = std::bit_cast<T>(std::array<std::byte, sizeof(T)>{});
7 });
8
9 template <class T>
10 requires bit_castable<T> && accesible_aggregate<T>
11 consteval auto has_packed_layout() -> bool {
12     auto bytes = std::array<u8, sizeof(T)>{};
13     const T reference = std::bit_cast<T>(std::array<u8, sizeof(T)>{});
14
15     for (std::size_t i = 0; i < sizeof(T); ++i) {
16         bytes[i] = 1u; // Perturb the object representation.
17         const T instance = std::bit_cast<T>(bytes);
18
19         if (memberwise_equal(instance, reference)) {
20             return false;
21         }
22         bytes[i] = 0u; // Restore the object representation.
23     }
24     return true;
25 }
26
27 template <class T>
28 concept packed_layout = has_packed_layout<T>();

```

Figura 60. Implementación del Concepto `teg::packet_layout` utilizado para identificar estructuras de datos empaquetadas. Autoría propia.

C.4 Interfaz de números enteros de longitud variable

```

1 template <class T = usize> requires (std::integral<T>)
2 class varint {
3 public:
4     using value_type = T;
5
6     constexpr varint() = default;
7     constexpr varint(value_type value) : m_value(value) {}
8     constexpr operator value_type() const { return m_value; }
9     constexpr operator value_type&() & { return m_value; }
10
11 private:
12     T m_value;
13 };
14
15 using vint32 = varint<i32>;
16 using vint64 = varint<i64>;
17 using vuint32 = varint<u32>;
18 using vuint64 = varint<u64>;

```

Figura 61. Clase genérica `teg::varint`, interfaz utilizada para serializar números de longitud variable. Autoría propia.

```

1 template <class F, class T>
2 constexpr auto usr_serialize(F&& encode, varint<T> var) -> error {
3     using value_type = std::make_unsigned_t<typename varint<T>::value_type>;
4     value_type value = [var](){ constexpr {
5         if constexpr (std::is_signed_v<T>) {
6             return zigzag::encode((T)var);
7         }
8         else {
9             return var;
10        }
11    }();
12
13    do {
14        u8 byte = value & 0b01111111;
15        value >>= 7;
16
17        if (value != 0) {
18            byte = byte | 0b10000000;

```

```

19         }
20
21     if (auto const result = encode(byte); failure(result)) {
22         return result;
23     }
24 }
25 while (value != 0);
26 return {};
27 }
```

Figura 62. Codificación de la clase genérica `teg::varint`. Autoría propia.

```

template <class F, class T>
constexpr auto usr_deserialize(F&& decode, varint<T>& var) -> error {
1    using value_type = std::make_unsigned_t<typename varint<T>::value_type>;
2
3    value_type value = 0;
4    u64 shift = 0;
5    u8 size = 0;
6
7    while (size <= uleb128::max_encoded_size<value_type>()) {
8        u8 byte;
9        if (auto const result = decode(byte); failure(result)) {
10            return result;
11        }
12
13        value |= u64(byte & 0b01111111) << shift;
14
15        if ((byte & 0b10000000) == 0) {
16            if constexpr (std::is_signed_v<T>) {
17                var = zigzag::decode(value);
18                return {};
19            }
20            else {
21                var = value;
22                return {};
23            }
24        }
25        shift += 7;
26        size += 1;
27    }
28    return error { std::errc::value_too_large };
}
```

Figura 63. Decodificación de la clase genérica `teg::varint`. Autoría propia.

C.5 Cadena de caracteres en tiempo de compilación

```

1 template<class C, std::size_t N, class TT = std::char_traits<C>>
2 class basic_fixed_string {
3 public:
4     // Member types.
5     using type_traits = TT;
6     using value_type = C;
7     using size_type = std::size_t;
8     using difference_type = std::ptrdiff_t;
9     using reference = value_type&;
10    using const_reference = const value_type&;
11    using pointer = value_type*;
12    using const_pointer = const value_type*;
13    using iterator = value_type*;
14    using const_iterator = const value_type*;
15    using reverse_iterator = std::reverse_iterator<iterator>;
16    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
17    using string_view_type = std::basic_string_view<value_type, type_traits>;
18 // Construct/copy/move.
19    constexpr basic_fixed_string() = default;
20    constexpr basic_fixed_string(basic_fixed_string&&) = default;
21    constexpr basic_fixed_string(basic_fixed_string const&) = default;
22    constexpr basic_fixed_string& operator=(basic_fixed_string&&) = default;
23    constexpr basic_fixed_string& operator=(basic_fixed_string const&) =
24 default;
25    constexpr basic_fixed_string(std::initializer_list<value_type> l);
26    constexpr basic_fixed_string(const value_type (&str)[N+1]);
27    constexpr basic_fixed_string(
28        std::basic_string_view<value_type, type_traits> view);
29    constexpr basic_fixed_string& operator=(const value_type (&str)[N+1]);
30 // Element access.
31    constexpr pointer data();
32    constexpr const_pointer data() const;
33    constexpr const_pointer c_str() const;
34    constexpr reference operator[](size_type i);
35    constexpr const_reference operator[](size_type i) const;
36    constexpr reference front();
37    constexpr const_reference front() const;
38    constexpr reference back();
39    constexpr const_reference back() const;
40    constexpr reference at(size_type i);
41    constexpr const_reference at(size_type i) const;
42 // Iterators.
43    constexpr iterator begin();
```

```

44     constexpr const_iterator begin() const;
45     constexpr const_iterator cbegin() const;
46     constexpr iterator end();
47     constexpr const_iterator end() const;
48     constexpr const_iterator cend() const;
49     constexpr reverse_iterator rbegin();
50     constexpr const_reverse_iterator rbegin() const;
51     constexpr const_reverse_iterator crbegin() const;
52     constexpr reverse_iterator rend();
53     constexpr const_reverse_iterator rend();
54     constexpr const_reverse_iterator crend();
55     // Capacity.
56     constexpr size_type length() const;
57     constexpr size_type size() const;
58     constexpr size_type max_size() const;
59     constexpr bool empty() const;
60     // Conversions.
61     constexpr operator string_view_type() const;
62 public:
63     // Storage.
64     value_type m_data[N+1] = {};

```

Figura 64. Clase genérica base `teg::basic_fixed_string`, interfaz utilizada para implementar las cadenas de caracteres en tiempo de compilación. Autoría propia.

```

1 template <std::size_t N>
2 using fixed_string = basic_fixed_string<char, N>;
3
4 template <std::size_t N>
5 using fixed_u8string = basic_fixed_string<char8_t, N>;
6
7 template <std::size_t N>
8 using fixed_u16string = basic_fixed_string<char16_t, N>;
9
10 template <std::size_t N>
11 using fixed_u32string = basic_fixed_string<char32_t, N>;
12
13 template <std::size_t N>
14 using fixed_wstring = basic_fixed_string<wchar_t, N>;

```

Figura 65. Alias de la interfaz de cadenas de caracteres en tiempo de compilación utilizadas para definir el tamaño de los caracteres en bits. Por defecto se utiliza `teg::fixed_string`. Autoría propia.

C. 6 Ordenamiento de bytes.

```

1  template <options Opt>
2  constexpr auto requires_endian_swap() -> bool {
3      if constexpr (Opt == options::native_endian) {
4          return false;
5      }
6      else {
7          constexpr auto system_endian = std::endian::native;
8          constexpr auto config_endian = (Opt & options::big_endian)
9              ? std::endian::big : std::endian::little;
10
11         return system_endian != config_endian;
12     }
13 }
14
15 template <class T>
16 concept endian_neutral = sizeof(T) == 1;
17
18 template <class T>
19 concept endian_neutral_container = container<T>
20     && endian_neutral<typename T::value_type>;
21
22 template <class T>
23 concept endian_neutral_c_array = bounded_c_array<T>
24     && endian_neutral<std::remove_all_extents_t<T>>;
25
26 template <class T, options Opt>
27 concept endian_swapping_required = requires_endian_swap<Opt>()
28     && !(endian_neutral<T> || endian_neutral_container<T>
29           || endian_neutral_c_array<T>);

```

Figura 66. Conceptos utilizados para determinar cuándo se debe realizar un cambio de ordenamiento de bytes en los procesos de codificación y decodificación. Autoría propia.

Apéndice D.

Resultados de las Pruebas de Rendimiento

D. 1 Experimento 1: Página E-commerce

D. 1. 1 Tiempos de ejecución.

Tabla 15

Tiempos de serialización (ms) requeridos por las librerías de serialización en función del tamaño de los datos de prueba procesados (MiB) en el experimento nº 1. Autoría propia.

Datos de prueba (MiB)	Tiempo de serialización (ms)					
	Teg	Teg (compacto)	Protocol Buffers	FlatBuffers	MessagePack	Boost.Serialization
1	0,39	0,68	1,45	3,24	0,71	3,82
2	0,68	0,92	2,33	5,23	1,05	5,12
4	1,23	1,46	3,57	7,18	1,80	6,67
8	2,04	2,35	5,03	10,53	2,63	10,04
16	3,24	3,57	7,07	19,13	3,61	16,06
32	5,81	6,06	10,63	37,07	6,05	28,27
64	11,26	11,40	17,94	68,39	11,63	53,27
128	21,29	21,57	32,14	144,82	22,46	101,57
256	43,11	42,49	61,13	266,91	46,90	197,36
512	77,66	76,01	107,72	501,04	103,57	380,12
1024	182,66	179,09	258,49	1.241,52	301,05	719,51

Tabla 16

Tiempos de deserialización (ms) requeridos por las librerías de serialización en función del tamaño de los datos de prueba procesados (MiB) en el experimento nº 1. Autoría propia.

Datos de prueba (MiB)	Tiempo de serialización (ms)					
	Teg	Teg (compacto)	Protocol Buffers	FlatBuffers	MessagePack	Boost.Serialization
1	0,39	1,21	2,59	0,33	3,18	3,66
2	0,57	1,26	3,08	2,72	6,73	3,90
4	0,94	1,45	4,34	2,96	8,44	4,70
8	1,77	2,14	5,11	3,56	10,21	6,23
16	3,02	3,29	6,39	5,27	13,24	9,27
32	5,67	6,06	9,68	6,67	19,54	15,50
64	11,69	12,17	18,08	8,64	36,07	29,77
128	22,47	23,19	33,22	12,07	66,32	56,99
256	47,27	47,46	65,62	18,81	247,94	123,64
512	82,98	84,61	150,58	32,79	2.357,01	268,19
1024	618,16	611,57	631,62	100,53	5.179,00	5.179,00

D. 1. 2 Tasa de procesamiento de datos.

Tabla 17

Tasa de serialización (GiB/s) de las librerías de serialización en función del tamaño de los datos de prueba procesados (MiB) en el experimento nº 1. Autoría propia.

Datos de prueba (MiB)	Tasa de serialización (GiB/s)					Boost.Serialization
	Teg	Teg (compacto)	Protocol Buffers	FlatBuffers	MessagePack	
1	2,53	1,44	0,67	0,30	1,37	0,26
2	2,89	2,13	0,84	0,37	1,86	0,38
4	3,19	2,68	1,09	0,54	2,16	0,59
8	3,84	3,33	1,55	0,74	2,98	0,78
16	4,82	4,37	2,21	0,82	4,33	0,97
32	5,38	5,16	2,94	0,84	5,17	1,11
64	5,56	5,48	3,48	0,91	5,37	1,17
128	5,87	5,80	3,89	0,86	5,57	1,23
256	5,81	5,88	4,09	0,94	5,33	1,27
512	6,44	6,58	4,64	1,00	4,83	1,32
1024	5,48	5,58	3,87	0,81	3,32	1,39

Tabla 18

Tasa de deserialización (GiB/s) de las librerías de serialización en función del tamaño de los datos de prueba procesados (MiB) en el experimento nº 1. Autoría propia.

Datos de prueba (MiB)	Tasa de deserialización (GiB/s)					Boost.Serialization
	Teg	Teg (compacto)	Protocol Buffers	FlatBuffers	MessagePack	
1	2,48	0,81	0,38	2,96	0,31	0,27
2	3,44	1,56	0,63	0,72	0,29	0,50
4	4,18	2,70	0,90	1,32	0,46	0,83
8	4,42	3,64	1,53	2,19	0,76	1,25
16	5,17	4,74	2,44	2,97	1,18	1,69
32	5,51	5,16	3,23	4,68	1,60	2,02
64	5,35	5,14	3,46	7,24	1,73	2,10
128	5,56	5,39	3,76	10,36	1,89	2,20
256	5,29	5,27	3,81	13,30	1,01	2,03
512	6,06	5,94	3,33	15,27	0,21	1,88
1024	1,62	1,64	1,59	9,96	0,19	0,19

D. 1. 3 Tamaño de los archivos binarios.

Tabla 19

Tamaño de los archivos binarios (MiB) creados por las librerías de serialización como resultado de los procesos de serialización en el experimento nº 1. Autoría propia.

Datos de prueba (MiB)	Tamaño de archivo binario (MiB)					
	Teg	Teg (compacto)	Protocol Buffers	FlatBuffers	MessagePack	Boost.Serialization
1	0,78	0,64	0,74	0,98	0,66	1,05
2	1,78	1,64	1,75	1,89	1,66	2,05
4	3,78	3,64	3,75	3,56	3,71	4,05
8	7,78	7,68	7,79	7,17	7,70	8,05
16	15,78	15,68	15,79	14,24	15,75	16,05
32	31,78	31,68	31,80	28,42	31,75	32,05
64	63,78	63,68	63,80	56,75	63,75	64,05
128	127,78	127,68	127,80	113,43	127,75	128,05
256	255,78	255,68	255,80	226,78	255,75	256,05
512	511,78	511,69	511,80	453,49	511,75	512,05
1024	1.023,78	1.023,73	1.023,85	907,01	1.023,76	1.024,05

D. 1. 4 Utilización de la memoria principal.

Tabla 20

Utilización de la memoria principal (bytes privados) de las librerías de serialización binaria en los procesos de serialización del experimento nº 1. Autoría propia.

Datos de prueba (MiB)	Uso de la memoria principal (MiB)					
	Teg	Teg (compacto)	Protocol Buffers	FlatBuffers	MessagePack	Boost.Serialization
1	4,28	4,09	13,09	7,48	5,46	6,96
2	7,80	7,66	19,14	10,87	8,36	11,52
4	11,87	11,64	25,15	13,69	12,35	17,50
8	20,72	20,61	37,89	23,58	22,22	29,86
16	37,41	37,20	62,93	38,88	38,92	53,80
32	71,50	71,73	113,55	77,52	72,97	103,00
64	140,23	139,31	216,28	145,70	140,82	202,38
128	275,68	275,95	418,23	311,20	276,32	400,85
256	534,67	534,29	811,09	570,41	536,25	788,86
512	1.074,05	1.073,24	1.630,22	1.103,36	1.077,19	1.582,77
1024	2.069,99	2.069,31	3.110,50	2.093,80	2.071,71	3.093,21

Tabla 21

Utilización de la memoria principal (bytes privados) de las librerías de serialización binaria en los procesos de deserialización del experimento nº 1. Autoría propia.

Datos de prueba (MiB)	Uso de la memoria principal (MiB)					
	Teg	Teg (compacto)	Protocol Buffers	FlatBuffers	MessagePack	Boost.Serialization
1	4,23	4,13	10,97	6,58	8,52	7,06
2	7,89	7,69	15,20	8,32	13,17	12,23
4	11,98	11,62	18,31	12,93	20,10	20,23
8	20,93	20,59	27,91	20,45	32,32	37,05
16	37,46	37,16	43,65	39,58	58,41	69,15
32	71,79	71,94	77,04	77,71	110,64	134,77
64	140,60	139,73	145,91	145,90	216,84	266,96
128	276,63	275,11	282,00	312,30	426,30	530,41
256	532,85	532,59	538,85	569,73	951,68	1.044,10
512	1.074,38	1.074,52	1.080,05	1.106,15	1.864,01	2.097,80
1024	2.067,37	2.066,90	2.074,97	2.094,55	3.653,39	4.118,39

D. 1. 5 Utilización del procesador.

Tabla 22

Utilización de la CPU (porcentaje de todos los núcleos) de las librerías de serialización binaria en los procesos de serialización en el experimento nº 1. Autoría propia.

Datos de prueba (MiB)	Uso de la CPU (%)											
	Teg		Teg (compacto)		Protocol Buffers		FlatBuffers		MessagePack		Boost.Serialization	
	Inf.	Sup.	Inf.	Sup.	Inf.	Sup.	Inf.	Sup.	Inf.	Sup.	Inf.	Sup.
1	24,15	24,93	24,54	24,60	24,15	24,37	24,02	24,95	24,11	24,86	24,74	24,93
2	24,03	24,37	24,36	24,88	24,55	24,75	24,09	24,67	24,23	24,86	24,21	24,23
4	24,03	24,90	24,34	24,65	24,10	24,58	24,09	24,78	24,01	24,56	24,71	24,78
8	24,21	24,43	24,78	24,88	24,46	24,96	24,24	24,76	24,15	24,58	24,08	24,28
16	24,78	24,79	24,15	24,24	24,00	24,83	24,60	24,66	24,43	24,50	24,15	24,70
32	24,26	24,86	24,56	24,93	24,72	24,82	24,58	24,72	24,48	24,60	24,02	24,62
64	24,11	24,24	24,10	24,30	24,25	24,77	24,77	24,94	24,22	24,82	24,22	24,66
128	24,88	24,91	24,48	24,85	24,13	24,84	24,21	24,63	24,33	24,68	24,49	24,76
256	24,11	24,68	24,22	24,26	24,11	24,27	24,50	24,68	24,59	24,74	24,44	24,87
512	24,21	24,39	24,52	24,69	24,09	24,53	24,06	24,17	24,22	24,40	24,14	24,57
1024	24,58	24,74	24,52	24,82	24,08	24,18	24,11	24,20	24,36	24,73	24,53	24,91

Tabla 23

Utilización de la CPU (porcentaje de todos los núcleos) de las librerías de serialización binaria en los procesos de deserialización en el experimento nº 1. Autoría propia.

Datos de prueba (MiB)	Uso de la CPU (%)											
	Teg		Teg (compacto)		Protocol Buffers		FlatBuffers		MessagePack		Boost.Serialization	
	Inf.	Sup.	Inf.	Sup.	Inf.	Sup.	Inf.	Sup.	Inf.	Sup.	Inf.	Sup.
1	24,57	24,81	24,26	24,45	24,22	24,67	24,85	24,89	24,30	24,43	23,83	24,39
2	24,81	24,94	24,47	24,82	24,45	24,52	24,18	24,59	24,22	24,36	24,56	24,66
4	24,02	24,38	24,42	24,78	24,51	24,71	24,30	24,32	24,36	24,80	24,67	24,76
8	24,69	24,80	24,26	24,66	24,18	24,42	24,26	24,83	24,40	24,77	24,68	24,69
16	24,28	24,83	24,34	24,62	24,53	24,72	24,66	24,90	24,09	24,26	24,69	24,84
32	24,54	24,79	24,12	24,16	24,36	24,53	24,56	24,88	24,26	24,33	24,59	24,67
64	24,20	24,87	24,06	24,40	24,31	24,44	24,12	24,92	24,74	24,81	24,34	24,86
128	24,27	24,46	24,57	24,88	24,14	24,39	24,33	24,86	24,55	24,89	24,11	24,33
256	24,51	24,85	23,27	24,16	24,61	24,89	24,72	24,83	24,15	24,71	24,89	24,95
512	24,32	24,91	24,61	24,88	24,49	24,92	24,35	24,35	24,33	24,40	24,04	24,46
1024	24,29	24,57	24,20	24,95	24,11	24,27	24,47	24,75	24,53	24,80	24,73	24,76

D. 2 Experimento 2: Modelo 3D

D. 2. 1 Tiempos de ejecución.

Tabla 24

Tiempos de serialización (ms) requeridos por las librerías de serialización en función del tamaño de los datos de prueba procesados (MiB) en el experimento nº 2. Autoría propia.

Datos de prueba (MiB)	Tiempo de serialización (ms)					
	Teg	Teg (compacto)	Protocol Buffers	FlatBuffers	MessagePack	Boost.Serialization
1	0,05	0,66	1,12	1,02	1,05	5,52
2	0,10	1,31	3,25	2,09	2,10	11,10
4	0,25	2,65	7,64	4,14	4,23	22,21
8	0,60	5,34	15,45	7,38	8,46	44,41
16	1,55	10,81	30,87	14,52	16,95	88,92
32	3,17	21,71	62,25	28,58	34,17	177,38
64	6,38	43,57	127,10	55,91	69,94	353,79
128	12,97	88,19	256,66	109,57	146,07	705,72
256	26,80	179,06	516,67	217,99	309,71	1.409,46
512	57,89	376,80	1.051,60	436,86	699,68	2.818,43
1024	130,22	789,69	2.060,48	894,07	1.396,64	5.641,64

Tabla 25

Tiempos de deserialización (ms) requeridos por las librerías de serialización en función del tamaño de los datos de prueba procesados (MiB) en el experimento nº 2. Autoría propia.

Datos de prueba (MiB)	Tiempo de deserialización (ms)				
	Teg	Teg (compacto)	Protocol Buffers	FlatBuffers	MessagePack
1	0,05	1,15	1,81	0,05	3,28
2	0,10	2,30	4,13	0,10	7,15
4	0,25	4,63	9,00	0,21	10,51
8	0,59	9,21	18,34	0,41	20,95
16	1,53	18,51	37,70	0,83	42,49
32	3,14	37,22	80,67	1,66	88,24
64	6,39	75,22	195,76	3,32	186,97
128	12,92	152,93	661,31	6,63	401,46
256	26,51	313,71	1.579,12	13,28	876,56
512	57,01	668,34	4.563,39	26,62	1.941,90
1024	130,67	1.332,53	14.024,00	53,33	4.354,65

D. 2. 2 Tasa de procesamiento de datos.

Tabla 26

Tasa de serialización (GiB/s) de las librerías de serialización en función del tamaño de los datos de prueba procesados (MiB) en el experimento nº 2. Autoría propia.

Datos de prueba (MiB)	Tasa de serialización (GiB/s)				
	Teg	Teg (compacto)	Protocol Buffers	FlatBuffers	MessagePack
1	20,72	1,49	0,87	0,96	0,93
2	20,36	1,49	0,60	0,94	0,93
4	15,48	1,48	0,51	0,94	0,92
8	12,94	1,46	0,51	1,06	0,92
16	10,07	1,45	0,51	1,08	0,92
32	9,86	1,44	0,50	1,09	0,91
64	9,79	1,43	0,49	1,12	0,89
128	9,64	1,42	0,49	1,14	0,86
256	9,33	1,40	0,48	1,15	0,81
512	8,66	1,33	0,48	1,14	0,71
1024	7,68	1,27	0,49	1,12	0,72

Tabla 27

Tasa de deserialización (GiB/s) de las librerías de serialización en función del tamaño de los datos de prueba procesados (MiB) en el experimento nº 2. Autoría propia.

Datos de prueba (MiB)	Tasa de deserialización (GiB/s)				
	Teg	Teg (compacto)	Protocol Buffers	FlatBuffers	MessagePack
1	21,25	0,85	0,54	18,81	0,30
2	20,39	0,85	0,47	18,80	0,27
4	15,57	0,84	0,43	18,81	0,37
8	13,24	0,85	0,43	18,96	0,37
16	10,22	0,84	0,41	18,85	0,37
32	9,95	0,84	0,39	18,87	0,36
64	9,78	0,83	0,32	18,83	0,34
128	9,67	0,82	0,19	18,85	0,31
256	9,43	0,80	0,16	18,83	0,29
512	8,77	0,75	0,11	18,78	0,26
1024	7,65	0,75	0,08	18,75	0,23

D. 2. 3 Tamaño de los archivos binarios.

Tabla 28

Tamaño de los archivos binarios creados por las librerías de serialización binaria como resultado de los procesos de serialización en el experimento nº 2. Autoría propia.

Datos de prueba (MiB)	Tamaño de archivo binario (MiB)				
	Teg	Teg (compacto)	Protocol Buffers	FlatBuffers	MessagePack
1	1,00	0,67	0,88	1,00	0,80
2	2,00	1,34	1,77	2,00	1,60
4	4,00	2,69	3,54	4,00	3,21
8	8,00	5,37	7,08	8,00	6,41
16	16,00	10,75	14,16	16,00	12,82
32	32,00	21,49	28,32	32,00	25,64
64	64,00	42,99	56,65	64,00	51,29
128	128,00	85,98	113,30	128,00	102,57
256	256,00	171,97	226,60	256,00	205,15
512	512,00	343,94	453,21	512,00	410,29
1024	1.024,00	687,87	906,41	1.024,00	820,58

D. 2. 4 Utilización de la memoria principal.

Tabla 29

Utilización de la memoria principal (bytes privados) de las librerías de serialización binaria en los procesos de serialización del experimento nº 2. Autoría propia.

Datos de prueba (MiB)	Uso de la memoria principal (MiB)					
	Teg	Teg (compacto)	Protocol Buffers	FlatBuffers	MessagePack	Boost.Serialization
1	2,84	2,51	11,43	2,95	3,87	5,13
2	5,46	4,37	17,19	5,87	4,88	7,94
4	8,87	8,05	28,59	10,89	9,64	14,40
8	17,69	15,55	51,05	18,52	17,67	26,43
16	33,72	28,95	95,85	35,35	33,70	50,48
32	65,78	55,75	184,98	69,02	65,76	98,57
64	129,91	109,36	363,77	136,34	129,89	194,76
128	258,16	216,56	721,23	271,01	258,14	386,90
256	514,66	430,96	1.434,89	540,33	514,64	771,89
512	1.027,66	859,77	2.862,31	1.079,00	1.027,64	1.541,40
1024	2.053,67	1.717,37	5.717,02	2.156,31	2.053,65	3.080,18

Tabla 30

Utilización de la memoria principal (bytes privados) de las librerías de serialización binaria en los procesos de deserialización del experimento nº 2. Autoría propia.

Datos de prueba (MiB)	Uso de la memoria principal (MiB)					
	Teg	Teg (compacto)	Protocol Buffers	FlatBuffers	MessagePack	Boost.Serialization
1	2,84	2,52	10,52	2,95	6,44	5,14
2	5,46	4,37	15,33	5,87	12,12	9,43
4	8,87	7,56	24,55	10,90	24,80	17,43
8	17,69	15,06	42,83	20,12	46,71	33,04
16	33,72	28,46	79,90	36,96	91,21	65,10
32	65,79	55,26	153,77	70,62	179,53	129,23
64	129,91	108,87	300,86	137,95	354,21	257,48
128	258,17	216,07	594,21	272,61	701,77	513,98
256	514,67	430,47	1.180,75	541,94	1.400,00	1.026,98
512	1.027,67	859,27	2.353,82	1.080,60	2.795,60	2.052,97
1024	2.053,68	1.716,89	4.699,58	2.157,91	5.584,93	4.104,90

D. 2. 5 Utilización del procesador

Tabla 31

Utilización de la CPU (porcentaje de todos los núcleos) de las librerías de serialización binaria en los procesos de serialización en el experimento nº 2. Autoría propia.

Datos de prueba (MiB)	Uso de la CPU (%)											
	Teg		Teg (compacto)		Protocol Buffers		FlatBuffers		MessagePack		Boost.Serialization	
	Inf.	Sup.	Inf.	Sup.	Inf.	Sup.	Inf.	Sup.	Inf.	Sup.	Inf.	Sup.
1	24,84	24,91	24,13	24,36	24,52	24,84	24,44	24,76	24,22	24,23	24,11	24,58
2	24,28	24,58	24,74	24,82	24,41	24,73	24,00	24,69	24,12	24,34	24,14	24,23
4	24,17	24,53	24,27	24,65	24,07	24,46	24,02	24,40	24,33	24,38	24,52	24,53
8	24,38	24,67	24,10	24,45	24,44	24,68	24,71	24,76	24,82	24,88	24,41	24,93
16	24,12	24,34	24,19	24,25	24,21	24,21	24,60	24,94	24,16	24,32	24,53	24,60
32	24,83	24,94	24,36	24,37	24,03	24,80	24,02	24,89	24,22	24,26	24,10	24,21
64	24,13	24,30	24,10	24,55	24,51	24,90	24,39	24,66	24,89	24,92	24,18	24,38
128	24,07	24,42	24,06	24,48	24,64	24,70	24,44	24,74	24,49	24,67	24,36	24,48
256	24,55	24,65	24,49	24,68	24,65	24,89	24,79	24,85	24,35	24,66	24,32	24,81
512	24,17	24,70	24,59	24,59	24,25	24,46	24,04	24,66	24,08	24,88	24,41	24,68
1024	24,18	24,49	24,21	24,25	24,25	24,39	24,06	24,51	24,55	24,56	24,05	24,89

Tabla 32

Utilización de la CPU (porcentaje de todos los núcleos) de las librerías de deserialización binaria en los procesos de serialización en el experimento nº 2. Autoría propia.

Datos de prueba (MiB)	Uso de la CPU (%)											
	Teg		Teg (compacto)		Protocol Buffers		FlatBuffers		MessagePack		Boost.Serialization	
	Inf.	Sup.	Inf.	Sup.	Inf.	Sup.	Inf.	Sup.	Inf.	Sup.	Inf.	Sup.
1	24,58	24,85	24,41	24,70	24,19	24,39	24,08	24,29	24,27	24,58	24,32	24,77
2	24,23	24,66	24,13	24,90	24,51	24,73	24,81	24,84	23,72	24,13	24,36	24,50
4	24,32	24,87	24,39	24,43	24,33	24,66	24,27	24,28	24,10	24,14	24,18	24,20
8	24,31	24,35	24,11	24,45	24,17	24,36	24,46	24,85	24,21	24,55	24,11	24,21
16	24,06	24,74	24,50	24,89	24,08	24,24	24,51	24,63	24,35	24,53	24,06	24,88
32	24,37	24,38	24,63	24,86	24,43	24,56	24,06	24,77	24,19	24,64	24,73	24,83
64	24,83	24,88	24,58	24,65	24,22	24,93	24,31	24,50	24,45	24,53	24,53	24,79
128	24,06	24,36	24,16	24,72	24,15	24,63	24,59	24,71	24,05	24,51	24,09	24,28
256	24,33	24,88	24,25	24,77	24,59	24,66	24,04	24,50	24,30	24,46	24,15	24,77
512	23,44	24,84	24,16	24,61	24,37	24,47	24,01	24,72	24,62	24,82	24,85	24,95
1024	24,68	24,69	23,80	24,19	24,92	24,94	24,54	24,82	24,38	24,86	24,10	24,72

Apéndice E.

Futuro del Lenguaje C++

La librería de serialización producto de la investigación utiliza las últimas características incorporadas en C++20. Es evidente que el lenguaje ha evolucionado enormemente desde sus primeras versiones, y es emocionante pensar en cuánto más crecerá. Siempre hay espacio para mejoras, y un pequeño vistazo al futuro nos puede preparar para lo que está por venir:

La propuesta P1061R10³⁴ ha sido aprobada y se incorporará en C++26. Esta introduce en el lenguaje la vinculación de estructuras a identificadores empaquetados. Para nuestra investigación, esto implicará la eliminación completa de la limitación en el número de variables miembro que se pueden visitar en una clase, ya que la vinculación de estructuras generará automáticamente todos los identificadores necesarios para acceder a los miembros. La Figura 67 muestra la implementación de `teg::visit_members` utilizando esta nueva característica (puede comparar este código con el Apéndice C.2.2).

```

1 template<class F, class T> requires accesible_aggregate<T>
2 constexpr decltype(auto) visit_members(F&& f, T&& t) noexcept {
3     auto&& [...members] = t;
4     return f(members...);
5 }
6 }
```

Figura 67. Implementación de la función genérica de reflexión `teg::visit_members` en C++26. Autoría propia.

Si se aprueba la propuesta P0528R0³⁵, el lenguaje contará con un mecanismo estándar, `std::has_padding_bits`, para detectar bytes de relleno en cualquier tipo de dato. Esto mejoraría los requisitos para utilizar `teg::concepts::packed_layout`, que actualmente exige que `T` sea un tipo trivialmente copiable y de estructura estándar.

Por otra parte, miembros del comité estándar, como Herb Sutter, Wyatt Childers, Peter Dimov, y Andrew Sutton, están promoviendo un conjunto de propuestas para incluir un

³⁴ Revzin B., y Wakely J. (2024). Structured Bindings can introduce a Pack [La vinculación de estructuras puede introducirse hacia un paquete de parámetros]. <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p1061r10.html>

³⁵ Bastien J. F., y Spencer M. (2016). The Curious Case of Padding Bits, Featuring Atomic Compare-and-Exchange [El curioso caso de los bits de relleno, con la participación de la operación atómica de comparación e intercambio]. <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0528r0.html>

soporte completo de reflexión estándar en C++26 y C++29³⁶³⁷. Una idea que revolucionaría por completo el lenguaje, y que ha causado todo tipo de opiniones en la comunidad. De ser esto posible, en un futuro, nuestra librería podría beneficiarse enormemente de esto, soportando la serialización de más tipos de datos, como por ejemplo, clases con miembros privados o protegidos.

³⁶ Sutter H. (Noviembre de 2024). Peering Forward: C++’s Next Decade [Mirando hacia adelante: La próxima década de C++]. CppCon, The C++ Conference. Congreso llevado a cabo en Aurora, Colorado, Estados Unidos.

³⁷ Childers W., Dimov P., Katz D., Revzin B., Sutton A., Vali F., y Vandevoorde D. (2025). P2996R9: Reflection for C++26 [Reflexión para C++26]. Recuperado de <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p2996r9.html>