Optimización de la gestión de memoria en simuladores cuánticos mediante la compresión de datos sin pérdida de precisión

Daniel Adrián González Buendía

Javier Andres Sarmiento Salazar

Trabajo de grado para optar el título de Ingeniero de Sistemas

Director

Luis Alberto Nuñez Martínez

PhD en Ciencias de la Computación

Codirector

Gilberto Javier Díaz Toro

MsC en Ingeniería de Sistemas e Informática

Universidad Industrial de Santander

Facultad De Ingenierías Fisicomecánicas

Escuela De Ingeniería De Sistemas e Informática

Pregrado en Ingeniería de Sistemas e Informática Bucaramanga

2025

Dedicatoria

Este proyecto va dedicado a todas las personas que me apoyaron y aportaron a mi desarrollo como persona, y como profesional. A todas las personas que quiero.

A mis padres.

A mi hermana.

A mis amigos.

A Daniel.

Contenido

Lista de figuras

Lista de tablas

Lista de apéndices

Glosario

IoT: Internet of Things

LDA: Lenguaje de Descripción de Arquitectura (Architecture Description Language)

Resumen

Título: Optimización de la gestión de memoria en simuladores cuánticos mediante la compresión de datos sin pérdida de precisión[1]

Autor: Daniel Adrián González Buendía

Javier Andres Sarmiento Salazar[2]

Palabras clave: Computación cuántica, Simulación cuántica, Gestión de memoria, Compresión de datos, Computación de Alto Rendimiento (HPC)

Descripción:

La simulación de sistemas cuánticos en hardware clásico enfrenta una limitación severa debido al crecimiento exponencial en la memoria requerida. A medida que aumenta el número de qubits, la cantidad de datos a almacenar supera rápidamente las capacidades de las computadoras convencionales y supercomputadoras. Esta restricción impide la exploración de algoritmos cuánticos avanzados y dificulta la validación de experimentos en entornos clásicos antes de su ejecución en hardware cuántico. Por ello, es fundamental desarrollar estrategias eficientes de gestión de memoria que permitan extender el alcance de estas simulaciones sin comprometer la fidelidad de los cálculos. A pesar de los avances en simuladores cuánticos y técnicas de optimización, la gestión de memoria sigue siendo un cuello de botella significativo. En particular, muchas soluciones actuales sacrifican fidelidad a cambio de eficiencia, lo que limita su aplicabilidad en contextos donde la precisión es fundamental. Por ello, es necesario explorar estrategias que reduzcan el consumo de memoria sin comprometer la exactitud de los resultados. Este trabajo propone el diseño e implementación de una estrategia de compresión de memoria sin pérdida de precisión en simuladores cuánticos, con el objetivo de encontrar un balance óptimo entre uso de memoria, rendimiento y fidelidad. Al final, se espera evaluar la viabilidad de la propuesta y compararla con otros enfoques existentes para determinar su impacto en la optimización de simulaciones cuánticas en hardware clásico.

[1]Trabajo de Investigación

[2]Facultad De Ingenierías Fisicomecánicas. Escuela De Ingeniería De Sistemas e Informática.
Director: PhD. Luis Alberto Nuñez Martínez, Codirector: PhD. Gilberto Javier Díaz Toro

Abstract

Title: Optimization of memory management in quantum simulators through data compression without loss of precision[3]

Autor: Daniel Adrián González Buendía

Javier Andres Sarmiento Salazar[4]

Palabras clave: Quantum Computing, Quantum Simulation, Memory Management, Data Compression, High-Performance Computing (HPC)

Descripción:

The simulation of quantum systems on classical hardware faces a severe limitation due to the exponential growth in memory requirements. As the number of qubits increases, the amount of data to be stored quickly exceeds the capabilities of conventional computers and supercomputers. This restriction hinders the exploration of advanced quantum algorithms and complicates the validation of experiments in classical environments before their execution on quantum hardware. Therefore, it is crucial to develop efficient memory management strategies that extend the scope of these simulations without compromising computational fidelity. Despite advances in quantum simulators and optimization techniques, memory management remains a significant bottleneck. In particular, many current solutions sacrifice fidelity in exchange for efficiency, limiting their applicability in contexts where precision is crucial. Therefore, it is necessary to explore strategies that reduce memory consumption without compromising the accuracy of the results. This work proposes the design and implementation of a lossless memory compression strategy for quantum simulators, aiming to find an optimal balance between memory usage, performance, and fidelity. Ultimately, the feasibility of the proposal will be evaluated and compared with existing approaches to determine its impact on optimizing quantum simulations on classical hardware.

[3]Research Work

[4]Faculty of Physics-Mechanics Engineering. School of Systems Engineering and Computer Science. Advisor: PhD. Luis Alberto Nuñez Martínez, Co-Advisor: PhD. Gilberto Javier Díaz Toro

Introducción

La simulación cuántica en computadoras clásicas enfrenta un desafío fundamental debido al crecimiento exponencial del espacio de estados conforme aumenta el número de qubits. Por ejemplo, la representación de un sistema cuántico de 40 qubits requiere aproximadamente $8\,\text{bytes (double)} \times 2\,\text{(real, imag)} \times 2^{40} = 16\,\text{TB}$ de memoria RAM, mientras que para 50 qubits la demanda se dispara a petabytes (PB), superando la capacidad de las computadoras convencionales (Wu y cols., 2019). Esta limitación de memoria restringe el estudio de algoritmos cuánticos avanzados y su validación en hardware clásico antes de su implementación en procesadores cuánticos (Zhang y cols., 2024). Ante esta problemática, es necesario desarrollar estrategias eficientes de gestión de memoria que permitan extender el alcance de las simulaciones cuánticas sin comprometer la viabilidad computacional.

Considerando la creciente importancia de la computación cuántica, optimizar la memoria en simuladores cuánticos no solo permite ampliar el rango de algoritmos simulables en hardware clásico, sino que también reduce la dependencia exclusiva de procesadores cuánticos experimentales. Además, una gestión eficiente de memoria podría reducir significativamente los costos computacionales y el tiempo de simulación en supercomputadoras, lo que resulta en un beneficio directo tanto para la investigación en computación cuántica como para aplicaciones prácticas en la industria y la ciencia.

Para abordar este problema, se han explorado diversas estrategias que pueden agruparse en varias familias de métodos. Entre ellas, la compresión de datos ha sido ampliamente estudiada con enfoques como la compresión con pérdida controlada (e.g., SZ, ZFP), que permite reducir la carga de almacenamiento de vectores de estado sin afectar significativamente la fidelidad de los cálculos (Wu y cols., 2018). Otra línea de investigación relevante es la de los métodos de podado de estados cuánticos, los cuales eliminan amplitudes irrelevantes para disminuir el consumo de memoria, aunque con un impacto variable en la precisión del resultado (Song, Li, y Wu, 2023). Asimismo, la computación de alto rendimiento (HPC o High

Performance Computing) ha demostrado ser una solución clave, empleando distribución de datos entre nodos y optimizaciones en el acceso a memoria para maximizar la eficiencia de la simulación en infraestructuras de supercomputadoras (Díaz, Steffenel, Barrios, y Couturier, 2024).

A pesar de los avances en estas áreas, muchas de estas estrategias introducen errores acumulativos en la simulación, lo que puede comprometer la validez de los resultados. En este contexto, esta investigación propone una estrategia basada en compresión de datos sin pérdida de precisión, con el objetivo de optimizar el almacenamiento de vectores de estado sin alterar las amplitudes cuánticas originales. Para ello, el proyecto se desarrollará en varias etapas. Inicialmente, se llevará a cabo una revisión del estado del arte para identificar las técnicas de compresión más relevantes. Luego, se seleccionará un simulador cuántico adecuado que permita modificaciones en su gestión de memoria para evaluar dichas técnicas. Posteriormente, se diseñará una estrategia para seleccionar e integrar técnicas de compresión sin pérdida de precisión existentes, seguida de su implementación y optimización en el simulador elegido. Finalmente, se realizarán experimentos controlados para evaluar el impacto de la compresión en términos de ahorro de memoria, tiempo de simulación y fidelidad del resultado, comparándolo con enfoques tradicionales. En las siguientes secciones, se detallarán cada una de estas etapas y su contribución al objetivo final del proyecto.

1 Planteamiento del problema

La simulación de sistemas cuánticos en computadoras clásicas se enfrenta a una barrera casi insuperable: el crecimiento exponencial de la memoria requerida. Representar un estado cuántico de n qubits implica almacenar 2^n números complejos, lo que rápidamente supera la capacidad de cualquier sistema de memoria convencional. Las técnicas actuales, como la poda de estados, reducen drásticamente el consumo de recursos pero a costa de la precisión, mientras que la compresión con pérdida, aunque ofrece mayores índices de reducción, sacrifica irremediablemente la fidelidad de los cálculos. Este dilema –la imposibilidad de simular

algoritmos cuánticos complejos sin perder información crítica– constituye el núcleo del problema, amenazando la validez y aplicabilidad de las simulaciones en el desarrollo de nuevas estrategias cuánticas.

Este desafío se agrava al considerar que la simulación precisa es esencial para la validación y desarrollo de algoritmos cuánticos en entornos clásicos. La degradación de la precisión debido a la pérdida de datos no solo impide obtener resultados confiables, sino que también limita la escalabilidad y el rendimiento de los simuladores cuánticos. Por ello, es crucial explorar alternativas que permitan optimizar el uso de recursos computacionales sin comprometer la exactitud de los cálculos, destacando la urgente necesidad de investigar métodos de compresión sin pérdida que ofrezcan un equilibrio entre eficiencia y fidelidad en la simulación cuántica.

Con este planteamiento se proponen los siguientes objetivos para abordar esta problemática.

2 Objetivos

2.1 Objetivo General

- Diseñar e implementar una estrategia eficiente de gestión de memoria en simuladores cuánticos basada en técnicas de compresión sin pérdida de precisión, con el fin de optimizar el uso de recursos computacionales sin comprometer la fidelidad de las simulaciones.

2.2 Objetivos Específicos

- Analizar el impacto del uso de herramientas de compresión de datos sin pérdida de precisión en el consumo de memoria y precisión de los resultados en simuladores cuánticos.
- Diseñar una estrategia integral de gestión de memoria basada en compresión sin pérdida de precisión, definiendo el algoritmo, los parámetros de compresión-descompresión y los puntos de integración con el simulador cuántico seleccionado.
- Desarrollar e integrar la estrategia diseñada en el simulador elegido, optimizando la sobrecarga computacional para garantizar un ahorro significativo de memoria sin afectar la fidelidad de la simulación.
- Implementar dos algoritmos cuánticos distintos en un simulador cuántico para evaluar el uso de memoria, velocidad y precisión con y sin compresión de datos.
- Comparar los resultados obtenidos con estrategias tradicionales y métodos que emplean compresión con pérdida de precisión, identificando ventajas y desventajas de cada enfoque.

3 Estado del Arte

Con el objetivo de explorar a fondo el panorama actual de la computación autonómica, y en particular, los mecanismos de adaptación de arquitecturas de software y los requisitos

fundamentales para su implementación, se llevó a cabo una revisión de la literatura en diversas bases de datos. Esta revisión abarcó un recorrido que partió de una visión general y se adentró en aspectos cada vez más específicos. Durante este proceso, se examinaron detalladamente las propuestas y componentes clave de sistemas de software autonómicos, así como las diversas nociones relacionadas con notaciones, algoritmos para la comparación de estructuras de datos y los mecanismos esenciales para la adaptación de arquitecturas de software.

3.1 Computación Autonómica

El concepto de computación autonómica, definido inicialmente por IBM (2001), se refiere a un conjunto de características que presenta un sistema computacional el cual le permite actuar de manera autónoma, o auto-gobernarse, con el fin de alcanzar algún objetivo establecido por los administradores del sistema (Lalanda, Diaconescu, y McCann, 2014).

Los 8 elementos clave, definidos por IBM, que deberían presentar este tipo de sistemas son:

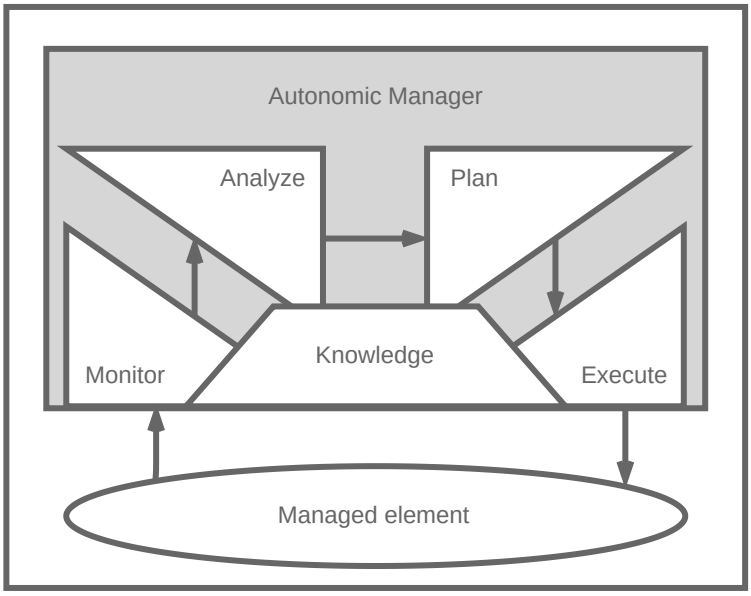
1. Auto-conocimiento: habilidad de conocer su estado actual, las interacciones del sistema.
2. Auto-configuración: capacidad de reconfigurarse frente a los constantes cambios en el entorno.
3. Auto-optimización: búsqueda constante de optimizar el funcionamiento de sí mismo.
4. Auto-sanación: aptitud de restaurar el sistema en el caso de que se presenten fallas.
5. Auto-protección: facultad de protegerse a sí mismo de ataques externos.
6. Auto-conciencia: posibilidad de conocer el ambiente en el que el sistema se encuentra.
7. Heterogeneidad: capacidad de interactuar con otros sistemas de manera cooperativa.
8. Abstracción: ocultar la complejidad a los administradores del sistema con objetivos de alto nivel de abstracción.

En el caso de que un sistema tenga una implementación parcial de las características mencionadas, este podría considerarse autonómico. Por ende debería tener la capacidad de lidiar con los problemas como la complejidad, heterogeneidad e incertidumbre (Salehie y Tahvildari, 2005) al igual que reducir la cantidad de recursos tanto técnicos como humanos requeridos para mantener los sistemas en funcionamiento.

3.1.1 MAPE-K

En cuanto a la implementación de las características, IBM propone un modelo de ciclo auto-adaptativo, denominado MAPE-K (Krikava, 2013). Este acercamiento, compuesto de cinco fases, es uno de los ciclos de control más usados en implementaciones de sistemas auto-adaptativos y computación autonómica (Arcaini, Riccobene, y Scandurra, 2015). En la figura 1, se presentan las fases que el *manejador* debe desarrollar para así administrar cada uno de los elementos del sistema computacional fundamentado en una base de conocimiento común (Gorla y cols., 2010).

Figura 1:
El ciclo auto-adaptativo MAPE-K (Gorla y cols., 2010)



Cada una de estas fases son:

- Monitorear (M): Esta fase se compone de la recolección, filtración y reporte de la información adquirida sobre el estado del elemento a manejar.
- Analizar (A): La fase de análisis se encarga de interpretar el entorno en el cual se encuentra, así como predecir posibles situaciones comunes y diagnosticar el estado del sistema.
- Planear (P): Durante la planificación se determinan las acciones a tomar, con el fin de llegar a un objetivo establecido a partir de una serie de reglas o estrategias.
- Ejecutar (E): Finalmente, se ejecuta lo planeado usando los mecanismos disponibles para el manejo del sistema.

Es de resaltar que este modelo, aunque útil para el desarrollo de este tipo de sistemas, es bastante general en cuanto a la estructura y no usan modelos de diseño establecidos (Ouareth, Boulehouache, y Mazouzi, 2018).

3.1.2 Mecanismos de Descripción

La fase de monitoreo dentro del ciclo MAPE-K es vital para el funcionamiento del manejador autonómico, pues es a partir de la información que se construirá la base de conocimiento requerida por las demás partes del ciclo. Parte de esta, está compuesta por el *estado del sistema*, el cual incluye la descripción de este en un momento dado (Weiss, Zeller, y Eilers, 2011).

Existen varias implementaciones de mecanismos de auto-descripción, y la utilidad de cada uno de ellos varía dependiendo en el tipo de sistema de software que se esté usando. Para el marco del proyecto, son relevantes aquellos ,mecanismos que estén orientados a los sistemas embebidos e IoT, algunos de estos son:

- **JSON Messaging**: Iancu y Gatea (2022) plantean un protocolo que emplea mensajería entre *gateways* con el fin de recibir información sobre estas. En términos simples,

estas funcionan como un *ping* hacia el nodo que luego retorna sus datos, al igual que los dispositivos conectados a ella, al encargado de recolectar toda esta información con el fin de construir una descripción del sistema.-

- **IoT Service Description Model**: O IoT-LMsDM, es un servicio de descripción desarrollado por Wang, Sun, y Aiello (2021) el cual está orientado al contexto, servicios e interfaz de un sistema IoT. De este se espera poder contar no solo con descripciones del estado del sistema en términos del ambiente, pero la funcionalidad (es decir, los *endpoints* a usar) al igual que las estructuras de datos que estos consumen.

- **Adaptadores de Auto-descripción**: En este acercamiento a los mecanismos de auto-descripción, se tienen adaptadores los cuales emplean los datos generados por los sensores del componente gestionado con el fin de realizar la determinación de la arquitectura desplegada. De igual manera, este acercamiento permite realizar modificaciones a la descripción de manera manual en caso de que se detecten problemas (H. A. Jiménez Herrera, 2022).

De esto podemos ver no solo las diferentes maneras en las que las implementaciones realizan las descripciones de los sistemas asociados, sino que también el alcance de estos en cuanto a lo que pueden describir.

3.1.3 Mecanismos de Adaptación

La adaptación, en el contexto de la computación autonómica, es la parte más importante en cuanto a la auto-gestión de un sistema de software se refiere. Así mismo, presenta el mayor reto debido a la necesidad de modificar código de bajo nivel, tener que afrontarse a incertidumbre de los efectos que pueden tener dichas alteraciones al sistema al igual que lidiar con esto en *runtime* debido a los problemas que el *downtime* tendría en los negocios (Lalanda y cols., 2014).

Esta adaptabilidad puede exponerse en múltiples puntos dentro de un sistema de software.

Pueden realizarse adaptaciones a nivel de kernel, lenguaje de programación, arquitectura e incluso datos (Lalanda y cols., 2014).

Manteniéndose en el marco del proyecto, son relevantes aquellas soluciones relacionadas con la modificación de la arquitectura. Siendo así, se consideraron únicamente los mecanismos de adaptación de componentes, o de reconfiguración:

- **Binding Modification**: Este mecanismo hace referencia a la alteración de los vínculos entre los diferentes componentes de la arquitectura. Estos tienen el objetivo de modificar la interacción entre componentes, lo que es especialmente común en implementaciones con *proxies*. Este tipo de mecanismo de adaptación fue usado por Kabashkin (2017) para añadir fiabilidad a la red de comunicación aérea.

- **Interface Modification**: Las interfases funcionan como los puntos de comunicación entre los diferentes elementos de la arquitectura. Siendo así, es posible que la modificación de estos sea de interés con el fin de alterar el comportamiento de un sistema al igual que soportar su heterogeneidad. Esto puede observarse en el trabajo desarrollado por Liu, Parashar, y Hariri (2004) en donde se define la utilidad de dichas adaptaciones al igual que la implementación de las mismas.

- **Component Replacement, Addition and Subtraction**: En términos simples, este mecanismo se encarga de alterar los componentes que hacen parte de la arquitectura; de esta manera, modificando su comportamiento. Ejemplos de lo anterior pueden verse en el trabajo de Huynh (2019), en el cual se evalúan varios acercamientos a la reconfiguración de arquitecturas a partir del remplazo de componentes tanto a nivel individual al igual que grupal.

  Este acercamiento a la mutación de la arquitectura también puede verse en el despliegue de componentes en respuesta a cambios en los objetivos de negocio de las aplicaciones, así como reacción a eventos inesperados dentro de la aplicación. Esto puede verse en trabajos como el de Patouni y Alonistioti (2006) donde se realizan este tipo de

implementaciones.

3.2 Sistemas IoT Autonómicos

Partiendo de lo anteriormente establecido, una de las áreas en las que estos conceptos de computación autonómica se ha hecho presente es el campo del IoT. Los acercamientos entre estas dos ramas de las ciencias de la computación se ha venido presentado con el objetivo de dar a los sistemas IoT la capacidad de adaptarse a su ambiente, teniendo el fin de optimizar los recursos disponibles y reducir la necesidad de interacción humana, a partir de la automatización de la configuración y procesos para mantener la disponibilidad de los servicios (Ashraf, Tahir, Habaebi, y Isoaho, 2023).

Ejemplos de estas implementaciones de propiedades autonómicas en sistemas IoT pueden apreciarse en trabajos como el de Rajan, Balamuralidhar, Chethan, y Swarnahpriyaah (2011), donde después de trabajar con diferentes protocolos de manejo de redes con el fin de poder realizar una administración de una red de sensores, se determinó que el manejo de un sistema de tal magnitud sería la limitante principal en el crecimiento del mismo. Esto terminó en la creación de un *control loop*, el cual se encargaba de la configuración automática de la red de sensores a partir de parámetros ambientales y objetivos de calidad de servicio.

3.2.1 Toma de Decisiones en Sistemas IoT Autonómicos

Algo a resaltar es el tipo de métricas con las cuales se determina la validez. Estas varían dependiendo de las necesidades de las implementación al igual que los objetivos definidos por los administradores del sistema. Durán-Polanco y Siller (2023), identificaron cuatro tipos de maneras de tomar decisiones sobre un sistema IoT.

- **Criterion-Driven**: Orientado principalmente al uso de algoritmos con el fin de establecer el camino a seguir en la toma de decisiones. Este acercamiento puede tener asociado un modelo matemático, algoritmos genéticos, Q-Learning, etc. dependiendo de la complejidad del sistema.

- **Data-Driven**: En este, a partir de datos de entrada y salida definidos, se busca determinar cómo generar datos de nuevo salida definidos dadas unas entradas. Estas están se usan principalmente en implementaciones que usan aprendizaje supervisado o sin supervisión.

- **Utility-Driven**: En el caso de la utilidad, se refiere a la comparación y evaluación de diferentes alternativas de modelos matemáticos con el fin de establecer una mejor toma de decisiones. Este está relacionado con algunos conceptos usados en teoría de juegos y modelos multicriterio.

- **Probability-Driven**: Este hace uso de modelos probabilísticos, tales como estocásticos o bayesianos, para la toma de decisiones. Normalmente se emplean para aprendizaje no supervisado para la determinación de parámetros a partir de un punto común dados unos datos.

4 Marco Teórico

4.1 Internet of Things

El Internet de las cosas, o IoT (Internet of Things); es una de las áreas de las ciencias de la computación en la cual se embeben diferentes dispositivos en objetos del día a día. Esto les da la capacidad de enviar y recibir información con el fin de realizar monitoreo o facilitar el control de ciertas acciones (Berte, 2018).

Esta tecnología, debido a su flexibilidad al igual que el alcance que puede tener, presenta una gran cantidad de aplicaciones que van desde electrónica de consumo hasta la industria. Encuestas realizadas en el 2020 reportan su uso en smart homes, smart cities, transporte, agricultura, medicina, etc. (Dawood, 2020).

4.1.1 Sistemas Embebidos

Los sistemas de cómputo embebidos hacen referencia a un sistema compuesto de microcontroladores los cuales están orientados a llevar a cabo una función o un rango de funciones específicas (Heath, 2002). Este tipo de sistemas, debido a la posibilidad de combinar hardware y software en una manera compacta, se ha visto en multiples campos de la industria como lo son el sector automotor, de maquinaria industrial o electrónica de consumo (Deichmann, Doll, Klein, Mühlreiter, y Stein, 2022).

4.1.2 Location-based Services

Location-Based Services, o servicios basados en ubicación, hace referencia a aquellos servicios que integran la ubicación geográfica de un dispositivo como una parte fundamental la cual da un valor agregado al usuario (Schiller y Voisard, 2004). Este tipo de servicios han sido principalmente usados en aplicaciones relacionas con entornos inteligentes, modelado espacial, personalización, conciencia de contexto, comunicación cartográfica, redes sociales, análisis de datos masivos, entre otros (Gartner y Huang, 2015; Allied Market Research, 2023).

4.1.3 Smart Campus

Un Smart Campus, equiparable con el concepto de Smart City, es una plataforma en la que se emplean tecnologías, sumado a una infraestructura física, con la cual se busca la recolección de información y monitoreo en tiempo real (Min-Allah y Alrashed, 2020). Los datos recolectados tienen el objetivo de apoyar la toma de decisiones, mejora de servicios, entre otros (Anagnostopoulos, 2023).

Estas plataformas, debido a su escala y alcance en cuanto a la cantidad de servicios que pueden ofrecer, requieren de infraestructuras tecnológicas las cuales den soporte a los objetivos del sistema. Es posible ver implementaciones orientadas a microservicios en trabajos de H. Jiménez Herrera, Cárcamo, y Pedraza (2020) donde se desarrolla una plataforma de software escalable con la cual se pueda lograr interoperatividad y alta usabilidad para todos.

4.2 Notación

De manera general, notación se refiere a una serie de caracteres, símbolos, figuras o expresiones usadas con el fin de expresar un sistema, método o proceso (Merriam, Webster, 2023a). Más específicamente en el contexto de las ciencias de la computación, las notaciones han sido usadas para la representación de estructuras y arquitecturas en el software, como lo son lenguajes de modelado tales como UML (Object Management Group, 2005); lenguajes de programación, como C/C++, Ruby (Bansal, 2013); algoritmia de manera visual (Rutanen, 2018); entre otros.

4.2.1 Gramática

La gramática, en el caso de los lenguajes, es un conjunto de reglas la cual es usada para describir tanto la sintaxis como la semántica de una lenguaje de programación (Sebesta, 2012). Siendo así, estas cumplen la función de describir una frase considerada válida dentro del contexto de un lenguaje dado (Sipser, 2012, p. 101). Existen varios tipos de gramática, tales como gramáticas de atributos, gramáticas libres de contexto, etc. (Sebesta, 2012).

4.2.2 Sintaxis

La sintaxis se refiere a la forma en la cual elementos, pertenecientes a un lenguaje, se estructuran de forma ordenada con el fin de formar estructuras más grandes (Merriam, Webster, 2023b). En el contexto de las ciencias de la computación, esta definición se puede expresar como un conjunto de reglas con las cuales regimos la forma en que escribimos en diferentes lenguajes de programación, marcado, entre otros (Friedman y Wand, 2008, p. 51).

4.2.3 Domain-Specific Languages

Los *Domain-Specific Languages* (DSL), o Lenguaje de dominio específico, son lenguajes de programación empleados para un fin específico. Estos están orientados principalmente al uso de abstracciones de un mayor nivel debido al enfoque que estos tienen para la solución de

un problema en específico (Kelly y Tolvanen, 2008). Ejemplos de este tipo de acercamiento, en el caso del modelado (*Domain-specific modeling*), pueden observarse en los diagramas de entidad relación usados en el desarrollo de modelos de base de datos (Celikovic, Dimitrieski, Aleksic, Ristić, y Luković, 2014).

4.3 Serialización de Datos

La serialización de datos se refiere a la traducción de una estructura de datos a una manera en la que pueda ser almacenada o transmitida de manera eficiente (Mozilla Foundation, 2023a). Este proceso ha sido, principalmente, adoptado por lenguajes de programación orientada a objetos, en donde existe un soporte nativo, tales como Go, JavaScript, o PHP; o existe algún tipo de framework o librería que permite hacerlo, como lo es en el caso de C/C++, Rust o Perl.

4.3.1 Métodos de Serialización de Datos

Dentro del ámbito de la programación, se encuentran diversas opciones en cuanto a formatos y técnicas empleadas para la serialización de datos. Cada uno de estos enfoques presenta sus particularidades, lo que resulta en ventajas y desventajas específicas, las cuales deben ser consideradas según las necesidades del proyecto. Las dos opciones principales, en cuanto a la serialización de datos, son:

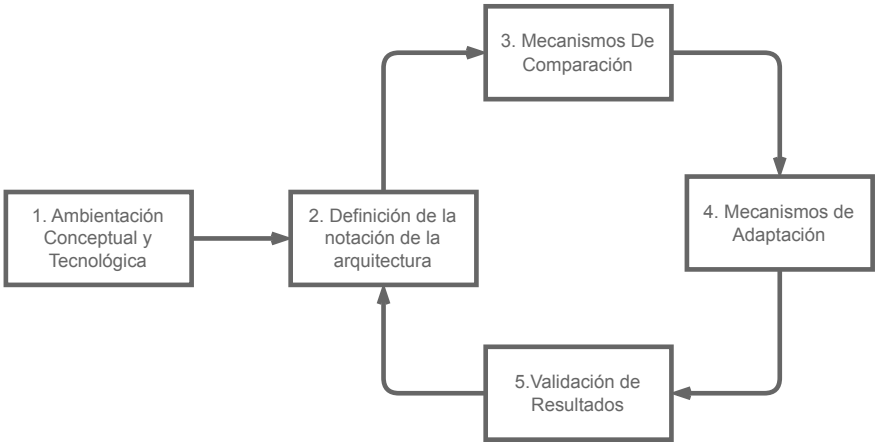
- **Serialización en Formato de Texto**: Esta forma de serialización emplea un formato legible por humanos, generalmente utilizando caracteres y símbolos. Este enfoque es especialmente útil cuando se necesita que los datos sean legibles, y editables, directamente por humanos, lo que los hace adecuados para la configuración de aplicaciones y la comunicación entre sistemas heterogéneos; o cuando se requiere tener compatibilidad entre diferentes sistemas y lenguajes de programación (Grochowski, Breiter, y Nowak, 2019).

- **Serialización Binaria**: Esta implica la representación de datos en un formato binario. Este enfoque es altamente eficiente en cuanto a espacio y velocidad, pero no es legible por humanos. Los formatos de serialización binaria, están diseñados para minimizar el tamaño de los datos serializados y optimizar el rendimiento en aplicaciones donde la velocidad y la eficiencia son críticas, sacrificando la estandarización e interoperatividad con otras implementaciones de este tipo de serialización (Grochowski y cols., 2019).

5 Metodología de la Investigación

Para el desarrollo del trabajo de grado, se propuso un modelo de prototipado iterativo compuesto de 5 fases (Ver fig. 2), que permitió avanzar a medida que se fue completando la fase anterior y permitió el poder iterar sobre lo que se ha desarrollado anteriormente.

Figura 2: Metodología de investigación



5.1 Ambientación Conceptual y Tecnológica

La primera fase de la metodología se basó en la revisión de la literatura, al igual que de lo presente en la industria, lo anterior fué necesario para cubrir las bases tanto conceptuales como técnicas requeridas para el desarrollo del proyecto.

Actividades

1. Identificación de las características principales de un sistema auto-adaptable.
2. Análisis de los mecanismos de adaptación de la arquitectura.
3. Análisis los algoritmos empleados para la comparación de las arquitecturas.
4. Determinación de los criterios de selección para el lenguaje de notación.
5. Evaluación de los posibles lenguajes de programación para la implementación a realizar.

Productos

1. Lista de criterios de selección para el lenguaje de notación.
2. Evaluación de los posibles lenguajes de programación para la implementación.

5.2 Definición de la notación de la arquitectura

La segunda fase consistió en la definición del cómo se realiza la declaración de la arquitectura. Partiendo de los criterios de selección establecidos en la fase 1, se determinó un lenguaje de notación el cual permitió definir la arquitectura objetivo a alcanzar, al igual que la gramática correspondiente para poder realizar dicha declaración.

Actividades

1. Selección del lenguaje de marcado a usar a partir de los criterios establecidos.
2. Definición de la gramática a usar para la definición de la arquitectura.
3. Determinación de como se realizará la representación de los componentes y partes de la arquitectura.
4. Implementación de una validador de la notación propuesta.

Productos

1. Notación a usar para la declaración de los requerimientos de las aplicaciones.
2. Validador de los archivos de configuración de la notación definida

5.3 Mecanismos De Comparación

Durante la tercera fase del proyecto, se buscó determinar e implementar cómo se realiza la comparación entre el estado de la arquitectura obtenido durante la auto-descripción de la misma y el objetivo establecido. Así mismo, y con el fin de reportar a los administradores de los sistemas, también fue necesario definir los *niveles* de similitud entre las 2 arquitecturas.

Actividades

1. Selección del mecanismo de comparación a usar para evaluación de estado de la arquitectura.
2. Implementación del mecanismo de comparación seleccionado.
3. Determinación de los diferentes niveles de similitud entre arquitecturas.

Productos

1. Implementación de un mecanismos que permita establecer el estado del sistema.
2. Implementación de una comparación entre el estado de referencia y el estado actual.

5.4 Mecanismos De Adaptación

La cuarta fase del proyecto fue orientada a la selección, al igual que la implementación en Smart Campus UIS, del conjunto de mecanismos de adaptación de la arquitectura.

Actividades

1. Definición el conjunto de mecanismos de adaptación.
2. Implementación el conjunto de mecanismos de adaptación seleccionados.

Productos

1. Implementación de un servicio encargado de la planeación para la ejecución de los mecanismos de adaptación definidos

2. Implementación de un servicio encargado de la ejecución de las acciones planeadas.

5.5 Validación De Resultados

La fase final del proyecto se centró principalmente en la realización de pruebas de los mecanismos implementados, los resultados obtenidos al igual que la documentación de todo lo que se desarrolló durante el proyecto.

Actividades

1. Realización de las pruebas del funcionamiento de la implementación realizada con diversas arquitecturas objetivo.
2. Recopilación la documentación generada durante el desarrollo de cada una de las fases del proyecto.
3. Compilación de la documentación para generar el documento final.
4. Correcciones y adiciones para la presentación final del proyecto de grado.

Productos

1. Validación del funcionamiento de los mecanismos implementados.
2. Código fuente de los diferentes servicios y librerías implementadas durante el transcurso del proyecto
3. Documento final detallando el desarrollo del proyecto
4. Presentación para la sustentación del proyecto

6 Lenguajes de Descripción de Arquitectura

6.1 La necesidad de una arquitectura de referencia

La necesidad de una arquitectura de referencia, o arquitectura objetivo; es una parte esencial dentro de la computación autonómica. Esta forma parte de la base de conocimiento

(K), y, de manera indirecta, de los objetivos del administrador del sistema (Lalanda y cols., 2014, p. 24).

Con el fin de establecer un objetivo para el sistema autonómico, fue necesario determinar una manera de realizar la declaración de dicha arquitectura, que estableciera un estado de referencia. De esta manera, podría evaluarse el estado del sistema en tiempo de ejecución, y así tomar las acciones necesarias para adaptarlo hacia el estado de referencia.

Ahora, fue también necesario establecer el punto desde el cual se lleva a cabo la comparación entre los estados del sistema. Esto guió la búsqueda para determinar el como se realizó la declaración de los estados objetivo de Smart Campus UIS, al igual que los datos a recolectar.

Siendo así, y dados los objetivos que buscan cumplir los ecosistemas inteligentes para la toma de decisiones (Anagnostopoulos, 2023), se enfocó no sobre la arquitectura sobre la cual trabaja la plataforma, sino sobre las aplicaciones desarrollándose sobre esta. Es decir, los datos requeridos. Son las necesidades definidas por las aplicaciones, las descritas y usadas para evaluar el estado del sistema. Siendo así, que el enfoque debía estar orientado a cumplir con las necesidades de los datos establecidas por Smart Campus. Esto se expondrá de manera más clara durante las fases de comparación y adaptación.

6.2 Criterios de selección

Con el fin de establecer la notación a usar para la declaración de las arquitecturas objetivo, se establecieron unos lineamientos con los cuales se realizaría la evaluación de las diferentes notaciones ya desarrolladas anteriormente. De esta manera se podría escoger la manera a representar los modelos, o en el caso de ser necesario, establecer los criterios por el cual se podría desarrollar uno.

Tabla 1:
Criterios usados para la determinación de la notación a utilizar

	Criterio	Explicación
C1	Describir la arquitectura de un sistema IoT	Este criterio es una base a establecer con el fin de descartar aquellos lenguajes de notación generales o no necesariamente usados para la descripción de arquitecturas IoT.
C2	Permitir la especificación de la ubicación del componente	La especificación de la ubicación de los componentes es importante en los sistemas IoT, especialmente dentro del contexto del proyecto en el cual se está trabajando con un Smart Campus; ya que los componentes pueden estar distribuidos en diferentes ubicaciones físicas y la evaluación de su integridad puede depender de su presencia en un lugar dado.
C3	Habilitar el modelado del componente a nivel de sus entradas	Es importante poder describir las entradas de los componentes, específicamente los datos que manejan, así como su rol dentro del sistema.
C4	Modelar el comportamiento de los componentes	La notación debe permitir modelar el comportamiento de los componentes, de manera que se puedan entender sus interacciones y su función en el sistema IoT. Dentro del contexto del proyecto, no es tan relevante, ya que no se está evaluando la funcionalidad del sistema, sin embargo, para futuros trabajos, podría facilitar la extensión de Smart Campus UIS.
C5	Posibilitar el establecer los estados de los componentes	La notación debe poder definir los estados de los componentes. Estos estados pueden ser tanto de comportamiento o operacionales.
C6	Permitir de exportar el modelo descrito a gráficas u otros formatos	Es importante que la herramienta permita exportar el modelo descrito en diferentes formatos para facilitar su integración con otras herramientas y sistemas, y para permitir su visualización en diferentes formatos.

6.3 Búsqueda de Alternativas

Una vez establecidos los criterios de selección, se realizó una exhaustiva búsqueda de alternativas disponibles en la literatura y en la industria para describir arquitecturas de sistemas IoT. Este proceso se realizó a partir de la revisión en diferentes bases de datos, como *Scopus*; al igual que algunas de las revistas especializadas en el tema, y el internet en

general.

Durante la búsqueda, se identificaron una gran variedad de opciones. Sin embargo, la gran mayoría de estas se filtraron, o descartaron; a partir de los criterios de selección establecidos. Esto se debió a que los LDAs usados tanto en la industria y academia, como AADL (Architecture Analysis and Design Language), tienen un enfoque a los campos de aviónica, equipos médicos y aeronáutica (lo que complicaría su implementación hacia sistemas de software IoT) (Carnegie Mellon University, 2022, 2017); o SysML, que son bastante genéricos y abarcan hardware, software e incluso personas y procesos (Object Management Group, 2015).

Es así como, de las posibles opciones de notación, se seleccionaron cinco, las cuales fueron evaluadas con el fin de determinar si alguna de estas hubiera podido ser usada, o si era necesario desarrollar una notación propia para el proyecto. A continuación, se presentan las alternativas evaluadas:

- **MontiThings:** Basado en MontiArc, otro LDA más general; está diseñado para el modelado y prototipado de aplicaciones de IoT. Este realiza las descripciones de sus arquitecturas en un modelo componente-conector, donde los componentes están compuestos por otros componentes; y los conectores definen la manera en la que se comunican dichos componentes a nivel de los datos y la dirección de los mismos. (Kirchhof, Rumpe, Schmalzing, y Wortmann, 2022b, 2022a)

- **Eclipse Mita**: Mita, creado por la Eclipse Foundation; es un lenguaje de programación orientado al facilitar la programación de sistemas IoT. Aunque como tal no es un LDA, está orientada a la descripción de los componentes y el comportamiento del sistema establecido, de esta manera, puede generar el código que debe correr en los dispositivos embebidos (Eclipse Foundation, 2018).

- **SysML4IoT:** Es un perfil de SysML[5] en el cual se usan estereotipos de UML con el

[5]Los perfiles se refieren a extensiones a UML, en este caso es una extensión de SysML en sí (Andre, 2007).

fin de abstraer las diferentes partes de los sistemas de IoT. Al igual que SysML, este permite el modelado más allá de dispositivos incluso llegando a personas y procesos, con la diferencia del enfoque dado al *dominio de IoT* [6] (Costa, Pires, y Delicato, 2016).

- **ThingML:** Similar a Mita, ThingML, es un lenguaje de modelado el cual tiene capacidades de generar el código requerido por los dispositivos embebidos. En términos del proyecto, este permite el modelado de los sistemas de IoT a partir de *state machine models*[7] los cuales permiten describir los componentes del sistema al igual que el comportamiento de estos (Harrand, Fleurey, Morin, y Husa, 2016).

- **IoT-DDL**: Iot-DDL es un LDA, implementado en XML, que describe objetos dentro de los ecosistemas IoT con base en sus componentes, identidad y servicios entre otros. Este tiene la capacidad de describir parte de la base de conocimiento que tienen los diferentes componentes (Principalmente relaciones y asociaciones entre componentes) (Khaled, Helal, Lindquist, y Lee, 2018).

Una vez seleccionadas las alternativas a evaluar, se utilizó la matriz de evaluación en la tabla 2 para determinar la solución ideal para el desarrollo del proyecto.

[6] El *dominio del IoT*, hace referencia al *Architecture Reference Model* establecido por IoT-A, un consorcio Europeo el cual buscaba el establecer un modelo para la interoperatividad de dispositivos IoT. (IoT-A, 2014)

[7] Los *state machine model*, también conocidas como Autómatas Finitos; son modelos matemáticos que describen todos los posibles estados de un sistema a partir de unas entradas dadas (Mozilla Foundation, 2023b).

Tabla 2: Evaluación de las alternativas en función de los criterios establecidos

	MontiThings	Eclipse Mita	SysML4IoT	ThingML	IoT-DDL
C1	✓	✓	✓	✓	✓
C2	✗	✗	✗	✗	✗
C3	✓	✓	✓	✓	✓
C4	✓	✓	✓	✓	✗
C5	✓	✓	✗	✓	✗
C6	✗	✗	✗	✓	✗

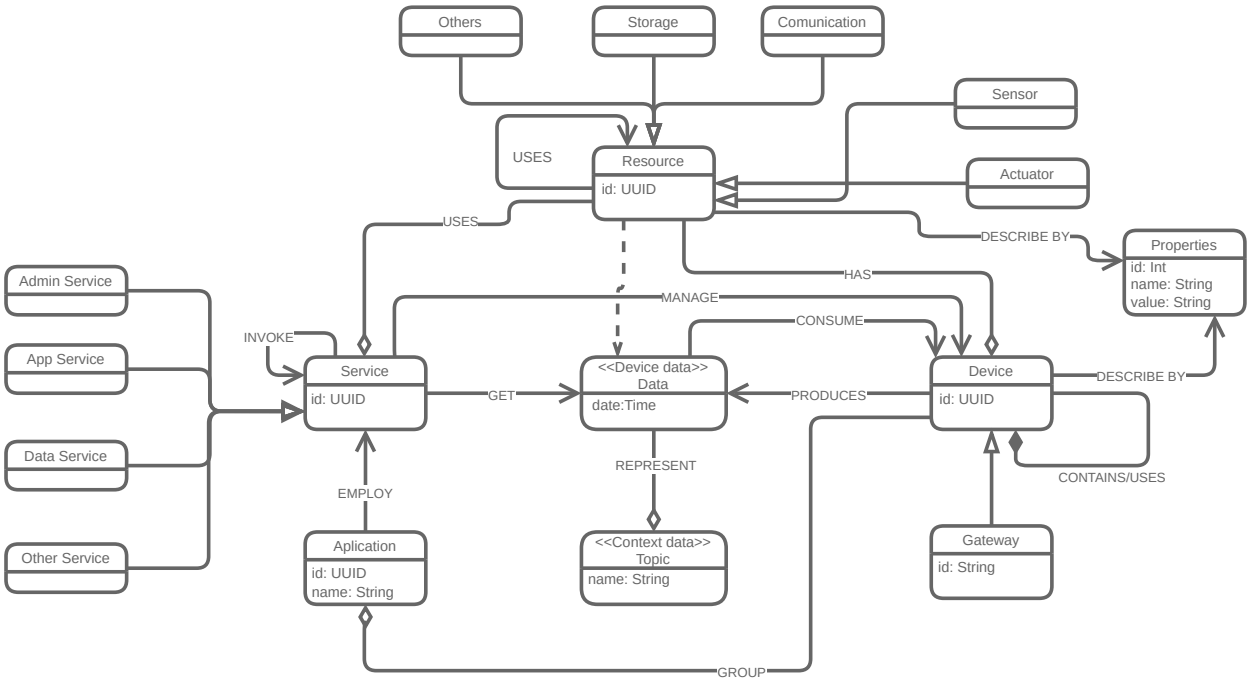
Partiendo de los resultados de la evaluación, se puede apreciar que ninguno de estos LDA cumple con los criterios establecidos para el proyecto. Aunque estos están orientados hacia la descripción y desarrollo de sistemas IoT, no están enfocados hacia su contexto en términos de aplicación más allá de la definición de comportamiento. Esto se debe a los objetivos de cada una de estas notaciones, de una u otra manera; buscan el representar como tal el sistema IoT en términos de su funcionalidad técnica y no la aplicación en si.

6.4 Un nuevo modelo para Smart Campus

Dado que no hay una alternativa que se adapte completamente a las necesidades del proyecto, se tuvo que definir una notación propia, la cual permita modelar las arquitecturas a nivel de aplicación bajo el contexto de un Smart Campus, tomando en cuenta los criterios definidos como guía para el desarrollo.

Primeramente, fue necesario definir un metamodelo que permitiera establecer la manera en la que se ven estas arquitecturas. Para ello, y partiendo de la implementación a realizar, se apoyó parcialmente en el modelo establecido por H. A. Jiménez Herrera (2022, p. 63).

Figura 3: Versión 2 del modelo concepto planteado por H. A. Jiménez Herrera



Basarse en el modelo de la figura 3, da la capacidad de describir a un nivel técnico un sistema IoT. Ahora, aunque se podría usar para el desarrollo del proyecto, fue necesario modificarlo, con el fin de acercarnos más hacia la descripción de un sistema IoT a nivel de aplicación.

El primer paso fue establecer el contexto de los dispositivos. Esto específicamente se refiere al criterio *C2* de la tabla 1, en donde, dada la necesidad de establecer la ubicación geográfica en algunas de las aplicaciones de los Smart Campus, era necesario poder describir los lugares pertenecientes a la aplicación.

Así mismo, se cubre el criterio *C3* cambiando las propiedades del dispositivos de una clase, externa a los dispositivos; a un atributo, interno, el cual le permite a los componentes manejar su propia información en cuanto a los datos que estos manejan. Estas propiedades pueden referirse a las entradas que tienen, en el caso de ser actuadores o procesadores de la información; o a los valores que reportan al sistema en el caso de ser sensores.

Ahora, algo importante a tener en cuenta, es la manera en la que se están reportando los

datos desde los sensores hacia Smart Campus UIS. Esto se debe a la necesidad de tener en cuenta los datos a los cuales tenemos realmente acceso desde cada uno de los dispositivos.

Cada uno de los dispositivos de Smart Campus, reporta un ‘JSON‘ similar al presente en la figura 4. Este mensaje contiene información que permite identificar al dispositivo, gracias al *UUID*; al igual que el momento y los datos que este reporta.

Figura 4: Mensaje JSON enviado por los dispositivos en Smart Campus UIS

(H. A. Jiménez Herrera, 2023)


```
{
"deviceUUID": "1",
"topic": "temperatura",
"timeStamp": "06-01-2024 10:39:02",
"values": {
    "temperature": 10.0,
    "co2": 15.0,
    "location": "AP2"
},
"status": "OK",
"alert": false
}
```


Es importante destacar que, aunque contamos con suficiente información para identificar los dispositivos, en estos mensajes no está presenta información crucial para el mapeo físico de los dispositivos. Por lo tanto, será necesario proporcionar manualmente estos datos durante la etapa de adaptación para llevar a cabo los ajustes necesarios en la arquitectura.

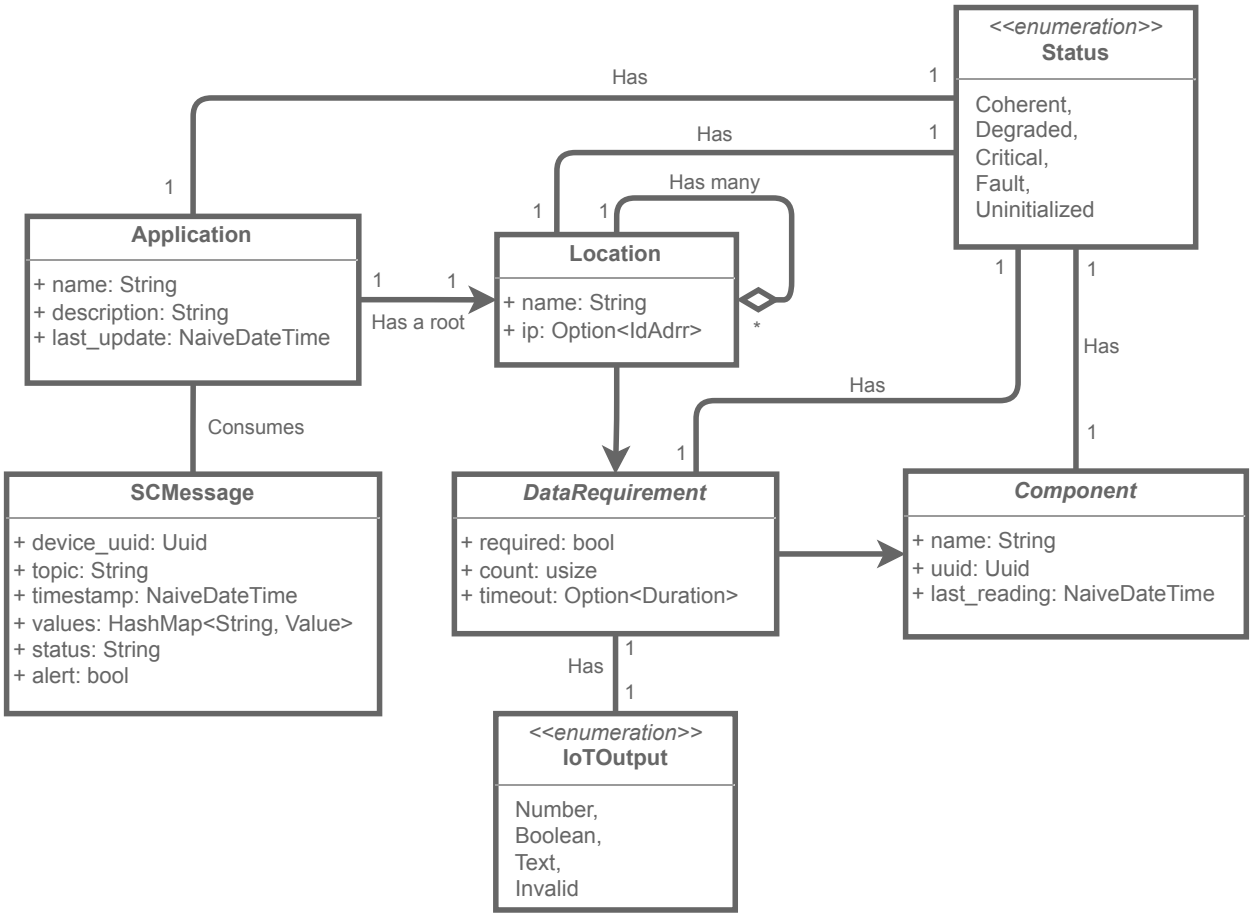
Partiendo de esto, se desarrolló el UML presente en la figura 5. La base de este es la aplicación (*Application*). Este contiene, a partir de una ubicación raíz, todas las demás locaciones (*location*) pertinentes para el correcto funcionamiento de esta.

Las locaciones, tienen una serie de requerimientos de datos (*DataRequirements*), las cuales son las representaciones lógicas, o de software, de las necesidades de la aplicación. Estos pueden ser de 2 tipos: requeridos, que son aquellos con los cuales de no tenerlos, la aplicación no podría funcionar correctamente; y los opcionales, los cuales, aunque buenos de tener, no impiden el funcionamiento.

Cada uno de estos requerimientos de datos, contiene la cantidad de dispositivos, y a su

1

Figura 5:
Versión 1 del metamodelo planteado para SCampusADL



vez datos, requeridos por la locación, al igual que el tipo de datos esperado (*IoTOutput*), y el que el tiempo máximo permitido entre reportes de datos.

Los requerimientos de datos, llevan un registro de los componentes (*Component*) que han reportado datos en la locación. Esto permite evaluar cada uno de estos, y en la misma medida, determinar el estado (*Status*) de cada una de las locaciones registradas en la aplicación. Estos estados buscan describir qué tan diferente es lo observado de lo esperado al igual que la validez de cada una de las partes.

Ahora, las aplicaciones, en cierta medida, con el fin de llevar el registro de los datos enviados, consumen el mensaje JSON visto en la figura 4. Este ha sido deserializado y mapeado en *SCMessage*. De esta manera, se podrá trabajar de manera sencilla en el procesamiento y

actualización del estado de las aplicaciones.

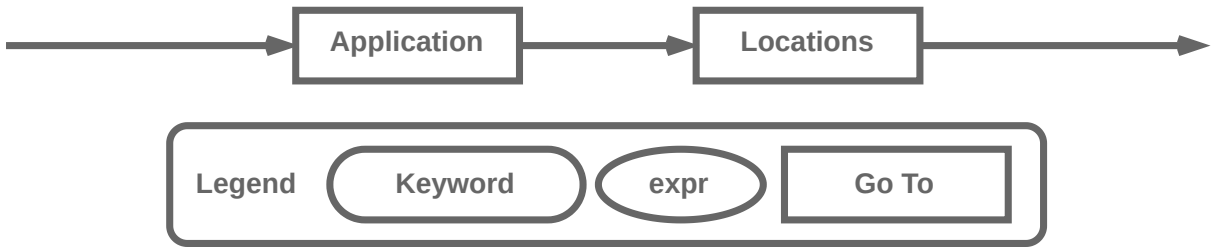
De este modelo, se desarrolló e implementó una librería, apodada *StarDuck*[8]. Esta librería contiene todas las estructuras de datos definidas en el modelo, las cuales serán usadas por el resto de los servicios y aplicativos necesarios para la declaración, evaluación y adaptación de las arquitecturas de software en el proyecto.

6.5 Sintaxis de la notación

Partiendo de esto, lo siguiente que se realizó fue la definición de la sintaxis de la notación a usar, basados en lo definido por el metamodelo. Para esto, se decidió usar `YAML`, un lenguaje de serialización de datos orientado a la legibilidad, reconocido y usado principalmente para la creación de archivos de configuración (Red Hat Foundation, 2023).

La sintaxis, como puede observarse en la figura 6, se compone de 2 partes: `Application`, donde se declara la aplicación; y `Locations`, en donde se define el contexto geográfico al igual que las necesidades de datos de estas.

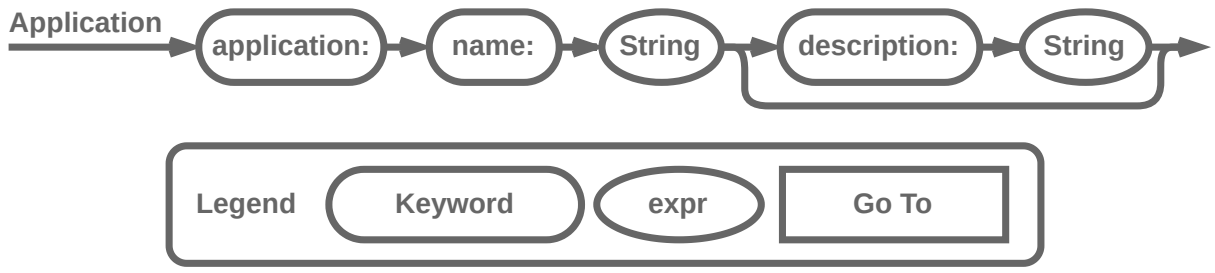
Figura 6: Diagrama de rail de la sintaxis definida para la notación de SmartCampusADL



Esta primera parte, como puede observarse en la figura 7 se refiere al nombre que se usará para referirse a la aplicación al rededor de todos los servicios, y una descripción opcional con fines de documentación.

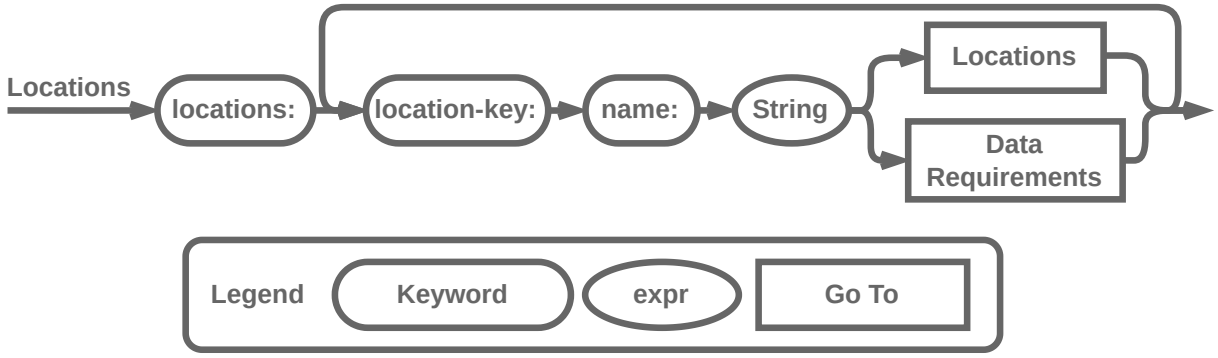
[8]El código fuente de la librería puede encontrarse en `https://github.com/ChipDepot/StarDuck`, y su respectivo paquete en crates.io, en `https://crates.io/crates/starduck`.

Figura 7: Diagrama de rail de la sintaxis de declaración de la aplicación



Para especificar las `locaciones` dentro del contexto de la aplicación, se definió la sintaxis que puede observarse en la figura 8. Partiendo de la locación raíz de la aplicación, podrán declararse n cantidad de locaciones, cada una con un nombre diferente.

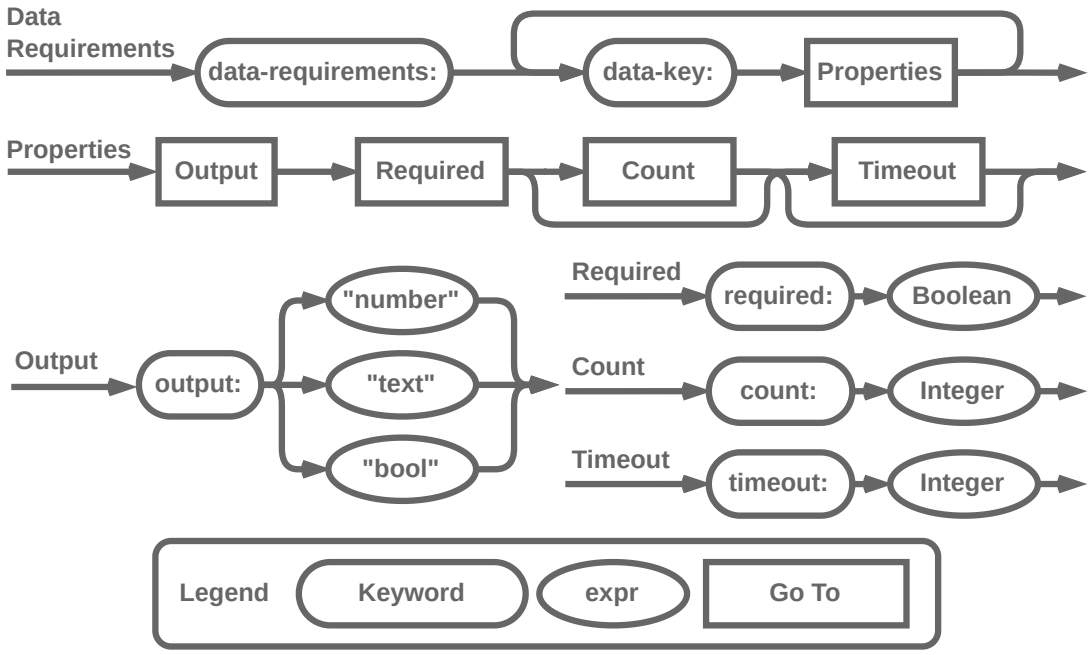
Figura 8: Diagrama de rail de la sintaxis definida para la notación del contexto geográfico de la aplicación



Las locaciones pueden funcionar de una de dos maneras. La primera es como agrupadores de otras locaciones, que puede verse como varios edificios de un campus o como varios pisos de un edificio, dependiendo de la granularidad que requiera la aplicación. Y, la segunda son puntos de origen de datos requeridos por la aplicación, reportados por diversas clases de dispositivos presentes.

Los requerimientos de datos, como se puede ver en la figura 9, están compuestos de varias partes. Las `data-key` se refieren al nombre del requerimiento de dato de la locación. Se espera que estos sean los mismos usados en la parte de `values` vista en los mensajes de la plataforma Smart Campus de la figura 4.

Figura 9: Diagrama de rail de la sintaxis definida para la notación de los componentes de la aplicación



A continuación, se procedería a definir las propiedades de dicho requerimiento de datos. Para esta primera versión de la notación, se han establecido las 4 cuatro propiedades vistas en el modelo (ver figura 5), dos son obligatorias para la declaración. La primera es `output`, que hace referencia al tipo de dato esperado [9], y la segunda es `required`, que especifica si el requerimiento es obligatorio u opcional. Las otras dos propiedades opcionales: `count`, que representa la cantidad de fuentes de datos necesarias en la ubicación, y `timeout`, que indica el tiempo máximo entre los informes de datos. En ambos casos, de no declararlas, se tomaría como valor por defecto 0.

El resultante de esta notación serían archivos de declaración de arquitecturas, en formato YAML, similares a la que se puede observar en la figura 10.

Con esto hemos creado un marco sólido para representar las aplicaciones de Smart Campus UIS. Este enfoque bien definido, permitirá avanzar en el desarrollo de las funcionalidades para comparación de los modelos, y la implementación de los mecanismos de adaptación a usar.

[9]Este output, en el modelo es el *enum* `IoTOutput`, que representa los tipos de datos.

Figura 10: YAML de una posible aplicación de monitorio

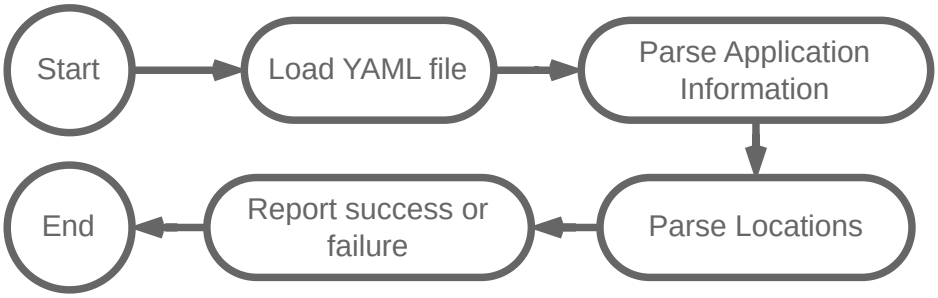
```yaml
application:
  name: Chip
  description: "Test App"

locations:
  campus-central:
    name: "Campus Central"
    locations:
      laboratorios-pesados:
        name: "Laboratorios Pesados"
        data-requirements:
          temperature:
            output: number
            required: true
            count: 1
            timeout: 30
          co2:
            output: number
            required: false
            count: 1
```

6.6 Implementando Una Validación

Partiendo de la notación definida para la declaración de las arquitecturas de referencia, se realizó la implementación de un cliente que permitiera realizar la validación de los archivos de configuración.

Este cliente, apodado *Lexical*, tiene como objetivo el tomar dichos archivos, y deserializarlos en las diferentes estructuras de datos definidas en el modelo, validando que estas cumplan con la sintaxis definida y que contengan los valores esperados. El grueso de este proceso se puede ver en la figura 11.

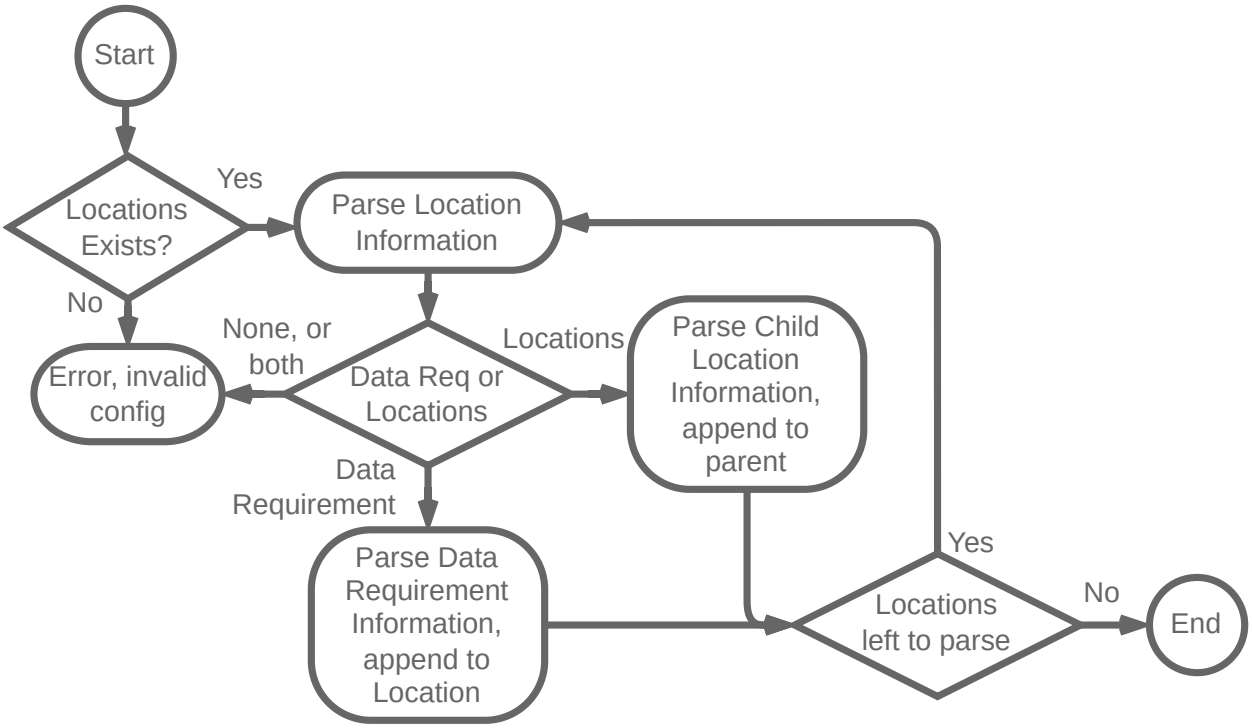
Figura 11:
Diagrama de flujo del proceso realizado por el módulo *Lexical*



Las 2 primeras partes del proceso son bastante directas. Lo primero es cargar el archivo indicado, validando que sea de tipo YAML; y tomar los datos del manifesto de la aplicación, en este caso el nombre y la descripción.

Seguido de esto, para la deserialización de las locaciones, se tendrán que recorrer cada una de las llaves presentes, procesando los valores internos de la locación. Como se estableció anteriormente, estas pueden ser, o más locaciones, hijas de la locación superior; o requerimientos de datos. El diagrama de dicho proceso se puede ver en la figura 12.

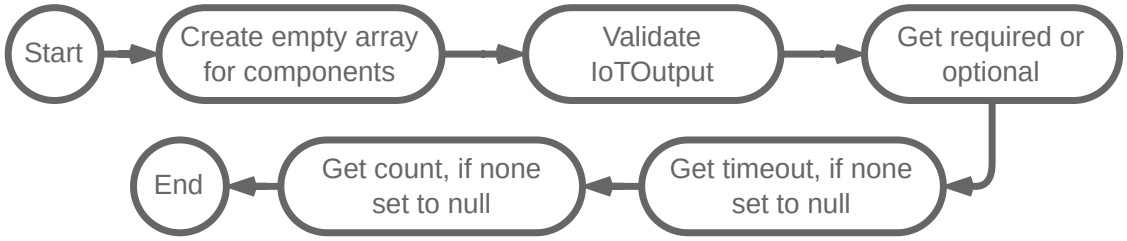
Figura 12:
Diagrama de flujo para validación y procesamiento de las locaciones



Este proceso de realiza para cada una de las locaciones registradas, lo que da como resultado el contexto geográfico en el que trabajaría la aplicación. Ahora, en cuanto al proceso de la validación de los requerimientos de datos, como se ve en la figura 13, es bastante directo en cuanto se toman las propiedades, definidas en el metamodelo; con la particularidad de crear un array vacío para los componentes. Esto se debe a que, en el momento de declarar la arquitectura, no conocemos ninguno de los datos necesarios para crearlos. Siendo así, estos

se crearan durante la ejecución, a partir de los mensajes de los dispositivos.

Figura 13:
Diagrama de flujo del procesamiento de los requerimientos de datos



El desarrollo de este módulos, se realizó en Rust. Escogido debido a su capacidad para garantizar la integridad de los datos y prevenir errores comunes, lo que es esencial en un entorno donde la precisión y la confiabilidad son críticas. Además, su ecosistema de herramientas y bibliotecas facilita la implementación eficiente de los módulos de validación y construcción de modelos.[10]

Con la implementación de este módulo de validación, hemos completado la primera fase del desarrollo de una notación propia para describir y una herramienta para validar arquitecturas de aplicaciones de Smart Campus. Ahora podemos avanzar en la implementación de las funcionalidades de comparación y adaptación de modelos, que son parte fundamental del enfoque a computación autonómica.

7 Estado Actual Frente al de Referencia

7.1 Conociendo el estado actual

Con la notación de las arquitecturas objetivo establecidas, al igual que el desarrollo del módulo encargado de la construcción y validación de los archivos de configuración; lo siguiente era la definición del proceso de la comparación entre la arquitectura actual y la arquitectura de referencia. Este proceso, permitirá evaluar el estado del sistema y, por consiguiente, establecer las acciones a tomar con el fin de adaptar la arquitectura hacia el estado

[10]El código fuente de Lexical puede encontrase en el repositorio de GitHub: `https://github.com/ChipDepot/Lexical`

objetivo establecido.

Esto requiere conocer el estado actual del sistema, al igual que el conocer el estado objetivo. Siendo así, y ya teniendo la posibilidad de declarar las necesidades de la aplicación, lo siguiente es establecer una manera de determinar el estado del sistema. Dado el enfoque hacia los datos recolectados, es necesario el precisar la manera en la que conoceríamos en qué estado se encuentra el sistema.

Lo primero era el identificar los puntos de acceso por los cuales podríamos acceder a los datos. En el caso de Smart Campus UIS, y teniendo en cuenta que el modelo que se definió depende de los mensajes enviados por los dispositivos, como se observa en la figura 14, existen dos opciones para procesar estos mensajes.

La primera, es consultar los mensajes enviados usando uno de los endpoints del servicio `data_microservice`, lo que da acceso a todos los mensajes registrados. Y, la segunda, usando el mismo bus de datos, empleado por los servicios de la plataforma, lo que permitiría el acceso, incluso de manera más directa, a los mensajes a medida que estos son enviados.

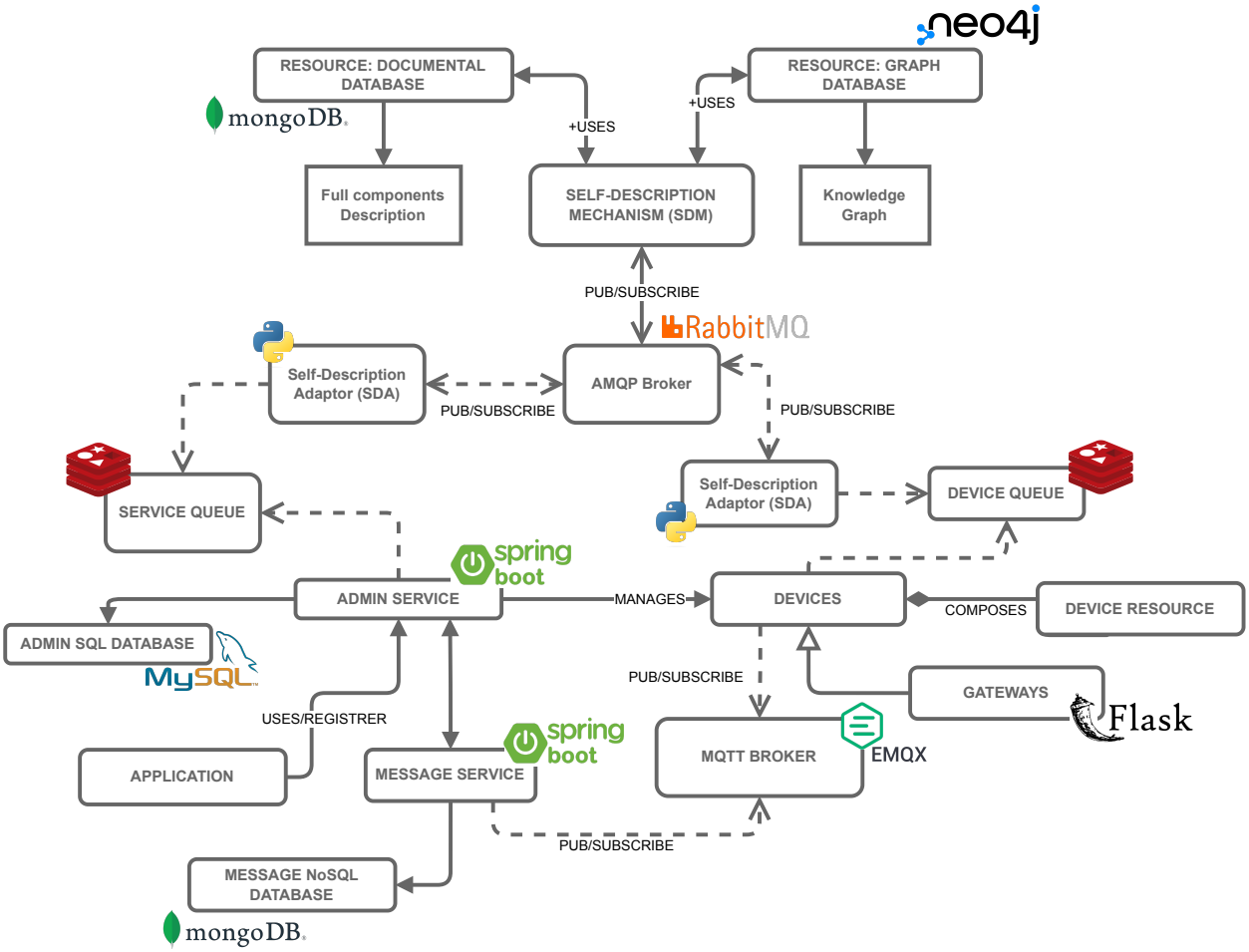
Ahora, cada una de las maneras de acceder a los datos es viable, y podría permitir la implementación correspondiente para determinar el estado del sistema. Sin embargo, de entre las dos opciones, se escogió el procesamiento de los mensajes enviados por los dispositivos, enviados por MQTT.

Esto se debe a varios factores, principalmente, debido a cómo se consultan los mensajes desde el endpoint. Para realizar esta consulta, es necesario conocer el *UUID* del dispositivo a consultar lo que, como se evidenció durante la validación de los requerimientos de datos en la sección 6.6, no es posible.

Aunque sería posible el realizar una modificación al servicio para poder ver todos los mensajes de los dispositivos, esto también podría presentar problemas por la manera en la que estos se listan. Ya que los mensajes se muestran todos en simultaneo, y no poseen una manera diferenciar 2 mensajes, más allá de su contenido (que no es necesariamente único); tendríamos que guardar todos los mensajes ya procesados y descartarlos cada vez que se

Figura 14:
Arquitectura del prototipo de Smart Campus definido por

H. A. Jiménez Herrera (2022)



consulte el endpoint, lo que generaría un *overhead* el uso de los recursos tanto de memoria como de procesamiento.

Siendo así, queda como opción el broker MQTT para la consulta de los mensajes. Este acercamiento permite el procesamiento, en tiempo real, de los mensajes enviados por los dispositivos, sin la necesidad de conocerlos; al igual que un aprovechamiento de los recursos ya que, una vez procesados, podemos librar el espacio usado. Esto le da una característica al proyecto en forma de una especie de *plug-in*, puesto que, al no requerir modificaciones sobre la plataforma, esta puede funcionar sin ninguna afectación en su estado actual.

Partiendo de lo anterior, se realizó el diseño e implementación de un servicio encargado

de procesar los mensajes que están siendo enviados por, y a, cada uno de los dispositivos registrados en Smart Campus UIS.

7.2 Centralizando los Datos

Lo primero para la implementación del observador, era determinar como este tendría acceso al estado de referencia. Hasta el momento, únicamente *Lexical* tiene el contexto definido por el usuario, por lo que era necesario definir una manera mediante la cual otros servicios puedan acceder a este.

Partiendo de la división de responsabilidades, se estableció el implementar un agregador, que contenga el estado definido por el usuario; desde el cual otros servicios puedan acceder a estos datos.

Esto tiene dos ramificaciones adicionales. La primera, es que abre a la posibilidad de declarar más de una aplicación en simultaneo, usando el agregador como un *hub* que permita guardar y consultar los estados de las aplicaciones que se registren, teniendo una instancia de observador por cada una de ellas.

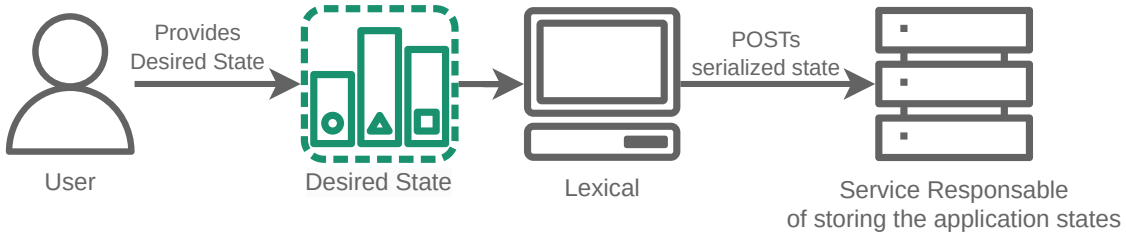
La segunda, posibilita el usar este agregador para almacenar los estados actuales reportados por el observador. Esto se debe a que, las condiciones para evaluar el estado de las aplicaciones, se encuentran en las propiedades definidas por el usuario durante la declaración del estado de referencia. Siendo así, es posible ver el estado objetivo a partir de las propiedades; y el estado actual basado en la evaluación de los componentes registrados en los requerimientos de datos.

De lo anterior, podemos planear un diagrama, visto en la figura 15, que refleje el estado actual de la arquitectura del proyecto.

La implementación de este agregador, apodado *Bran*[11], fue relativamente directa. Este tiene dos requerimientos, el primero, es la capacidad de recibir, guardar y actualizar los estados de las aplicaciones; y segundo, el poder enviar los estados almacenados cada vez que

[11]El código fuente de Bran puede encontrarse en el repositorio `https://github.com/ChipDepot/Bran`

Figura 15: Arquitectura actual del proyecto con el agregador

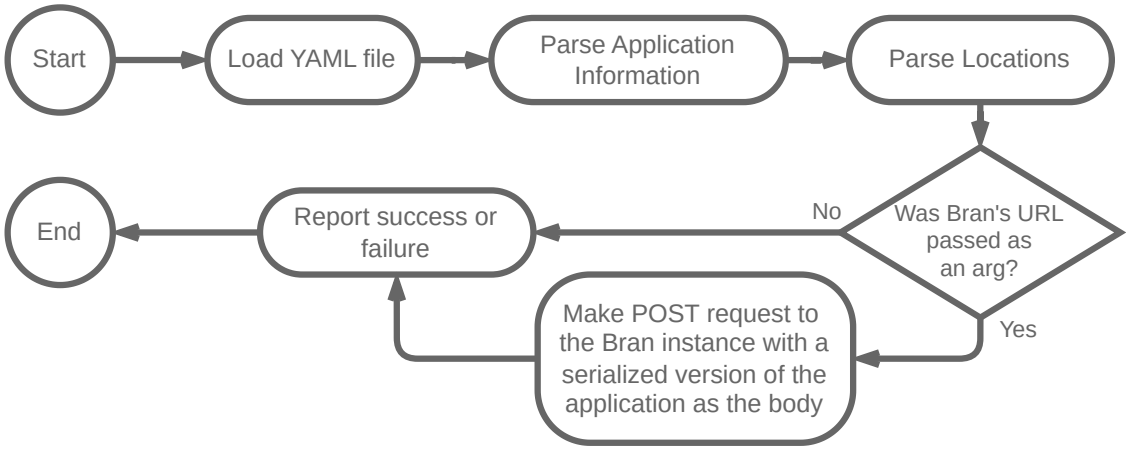


estos se soliciten.

Ambas de estas funcionalidades se implementaron usando el framework `Axum`[12]. Usando como base la librería *StarDuck*, se definió un único endpoint que toma, `apps/:app_name`, que acepta peticiones de tipo `GET`, para consultar el estado de la aplicación; `POST` y `PUT`, para el registro inicial de la aplicación, y posterior actualización de esta.

Ya con el servicio implementado se agregó, a *Lexical*, un cliente HTTP con el cual, tras validar la arquitectura objetivo proveída por el usuario, y como se ve en la figura 15, este realice una petición de tipo POST a la una instancia de *Bran* indicada con el fin de registrarla. Una versión actualizada del proceso realizado *Lexical* puede verse en la figura 16.

Figura 16: Diagrama de flujo del proceso realizado por Lexical actualizado



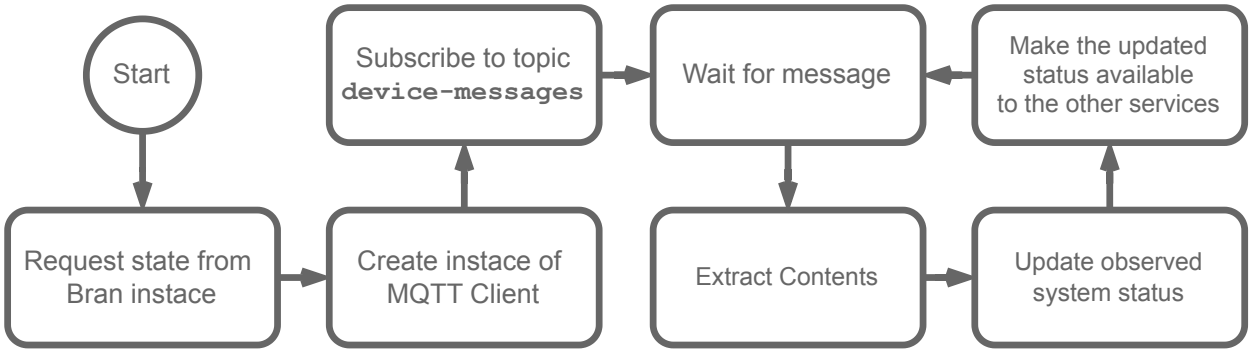
[12] Axum es un framework modular para aplicativos web Open Source desarrollado en Rust. Más información sobre este puede encontrarse en su repositorio `https://github.com/tokio-rs/axum`

7.3 Implementado un Event-Handler

En el contexto de Smart Campus UIS, cada uno de los mensajes enviados por los dispositivos es manejado como un evento. Es decir, cada mensaje recibido debe ser registrado y procesado por los microservicios de administración para lograr un objetivo final. Este procesamiento, recuerda al patrón de diseño *Observer* en el cual se busca el realizar una acción cada vez que el estado, de lo que se está observando, cambia (Shvets, 2019).

Siendo así, se estableció el realizar la implementación de un observador el cual se hará responsable de la captura de los mensajes enviados por los dispositivos y establecer el estado del sistema. De esto, como se puede ver en la figura 17 se propuso un proceso que debía realizar el observador, apodado *Looker*, a implementar.

Figura 17:
Primera propuesta del proceso a realizar por *Looker*

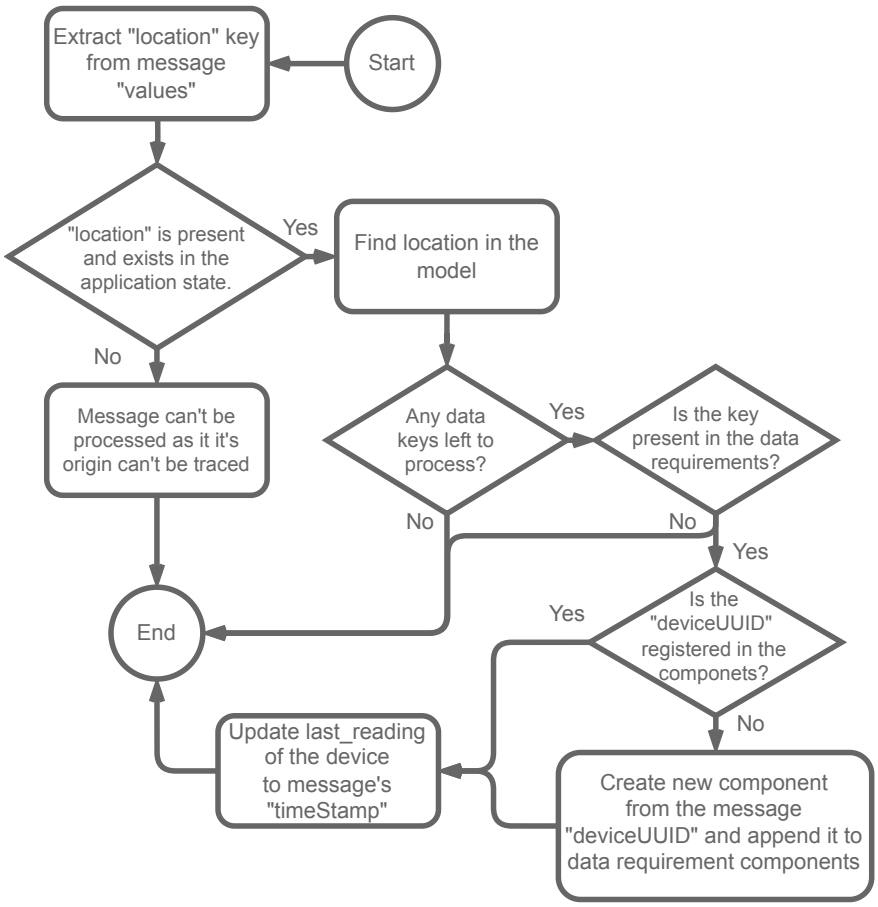


Este primer acercamiento, es bastante directo en cuanto a la implementación a realizar. Tras adquirir la definición del estado objetivo, se inicializa un cliente de MQTT, conectado el broker usado por Smart Campus UIS y subscrito al tópico usado por los dispositivos, `device-messages` (H. A. Jiménez Herrera, 2023), este empezaría a procesar los mensajes a medida que estos van llegando.

Cada mensaje recibido se somete a un conjunto de pasos para actualizar el estado de la aplicación. La figura 18 describe el proceso que el observador debe realizar con el fin de procesar los mensajes recibidos, dando como resultado, una versión actualizada del estado

de la aplicación.

Figura 18:
Proceso realizado por el observador durante el consumo de los mensajes enviados por los dispositivos



Lo primero a realizar, es ubicarse en el ubicación origen del mensaje. Esto se realiza buscando en las locaciones registradas en la aplicación, usando la llave (Ver figura 4) presente en el mensaje enviado por el dispositivo. En caso que esta, no esté presente, o la locación no haya sido referenciada en la declaración del estado objetivo, el mensaje será descartado. Se debe resaltar que, este puede ser uno de los principales puntos de falla debido a los posibles errores que se puedan presentar al usar nombres diferentes para una locación, sea en los mensajes enviados por problemas en la configuración; o en el momento de la creación del estado de referencia.

Una vez localizado el origen geográfico del mensaje, se recorrerán los datos reportados

por los dispositivos. Estos se cruzarán contra los requerimientos de datos, actualizando los tiempos de los componentes; o registrándolos en caso de que sea el primer mensaje publicado por ellos. Esto se realizará para todas los datos enviados por el dispositivo.

7.4 Comparando Arquitecturas

Ahora que se conoce el estado actual del sistema, al igual que estado de referencia; lo siguiente a realizar era la comparación de las arquitecturas. El resultado de esta, daría la base para poder decidir el qué hacer para adaptar el sistema y el cómo hacerlo.

Es necesario definir el como se realizará la evaluación del sistema. Esto no es tan sencillo como hacer una comparación usando un igual (`A == B`), ya que, el realizar esto, no resultaría realmente información sobre, qué, internamente, presenta problemas.

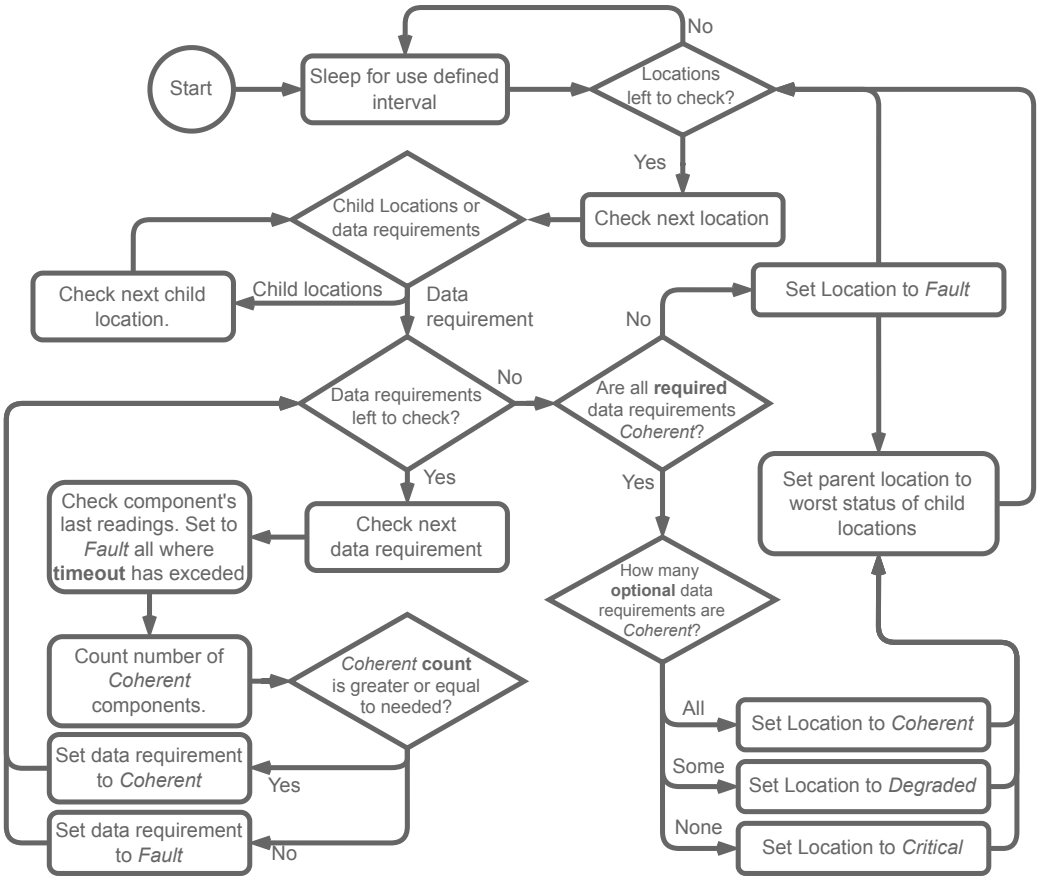
Ahora, la propiedades implementadas, definidas en los estados objetivo; son la manera las características principales por las cuales es posible realizar la evaluación del sistema. *Timeout*, cumpliendo la función de establecer el tiempo máximo entre los reportes de dispositivos, definiendo cuando estos pueden considerarse como "vencidos"; y *count*, como la cantidad de dispositivos en estado *Coherent* mínima para el desarrollo de la aplicación.

Siendo así, se definió un proceso de comparación consta de evaluar cada componente del sistema en función de las propiedades. A partir de estas, es posible el evaluar el estado de la aplicación. Este proceso es descrito por la figura 19.

El proceso, apodado reloj, debido a su ejecución dado un intervalo de tiempo definido por el usuario; realiza la evaluación de la aplicación, recorriendo la estructura locación por locación. Siendo así, y con el fin de evitar una dispersión grande de los lugares de procesamiento, se implementó en *Looker*.

Los componentes, pueden existir únicamente en un estado binario, sea *Coherent*, en el caso de que cumplan con las condiciones establecidas en el requerimiento; o *Fault* de lo contrario. Estos se basan únicamente en la propiedad *timeout*, marcándolos de manera correspondiente a partir de su último reporte registrado.

Figura 19:
Proceso reloj realizado por el observador para la actualización del estado de la aplicación



Así mismo, los requerimientos de datos, con la misma característica de su estado binario; usan propiedad *count* para definir su condición. En caso de que la cantidad total de componentes sea igual a la requerida, se marcará como *Coherent*; de lo contrario, estará en *Fault*. Se ha de resaltar que, debido a que estas propiedades son opcionales durante la declaración, podrán presentarse casos en los que, dada una configuración específica, con un único reporte (o incluso ninguno) estos podrán considerarse válidos durante todo el de ejecución de la aplicación.

Una vez evaluadas los componentes y los requerimientos de datos, lo siguiente es evaluar la locación. Esto dependerá del tipo de la locación. Si es un agrupador, su estado será el peor de los estados de las hijas. De lo contrario, si tiene requerimientos de datos, su estado depende de sus requerimientos de datos y del tipo de cada uno de ellos.

El estado de estas locaciones se determina en dos pasos. En el primer paso, se verifica si alguno de sus requerimientos obligatorios está en estado *Fault*. En ese caso, la locación se considera en estado *Fault*, ya que no puede cumplir con los requisitos mínimos de la aplicación. Para las locaciones sin requerimientos opcionales, este será el final del proceso.

El segundo paso, en el caso de locaciones con requerimientos opcionales, el factor determinante será la cantidad de estados *Coherent*. La locación tendrá un estado *Critical*, *Degraded*, o *Coherent* según si todos, algunos o ninguno de estos requerimientos está en estado *Fault*, respectivamente.

Esta manera de evaluar las locaciones permite el tener una idea más clara del estado de cada una de las locaciones, y por consiguiente, el estado de la aplicación. A partir de esta evaluación, se realizarán las diferentes adaptaciones necesarias con el fin de acercar el estado de la aplicación al estado de referencia esperado.

Una vez actualizado, y evaluado el estado; se tendría que reportar al agregador el nuevo estado de la aplicación. Para esto, se agregó un paso final tras la ejecución del proceso reloj, en la cual, *Looker*, realiza una petición tipo `PUT` al endpoint designado en *Bran*, con el fin de actualizar el registro. La arquitectura del proyecto, hasta el momento, se puede ver en la figura 20.

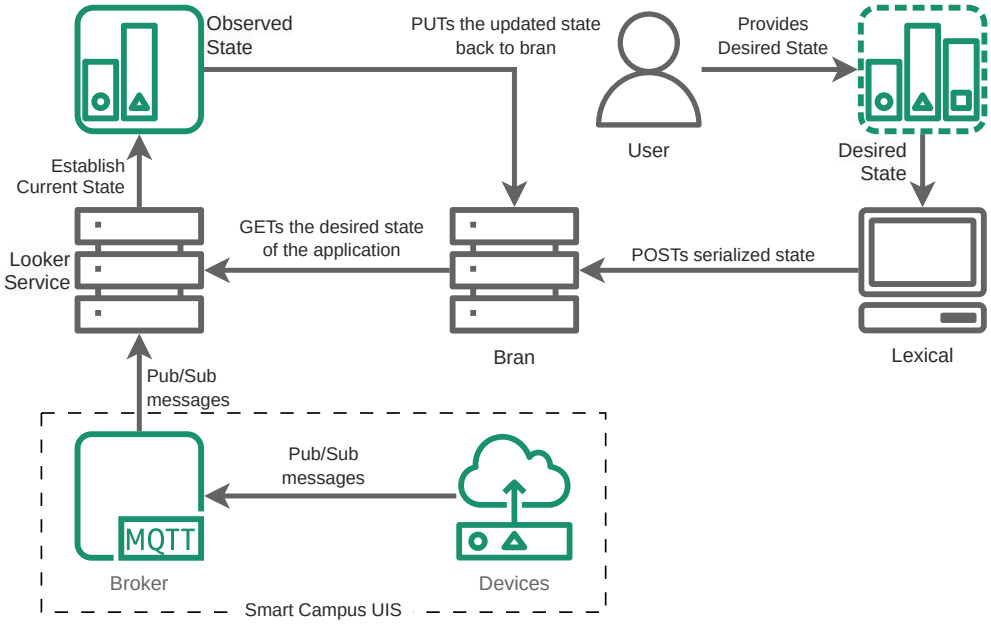
8 Adaptando la Arquitectura

8.1 Identificando Los Problemas, Estableciendo Acciones

Ya con una manera de evaluar los estados observados de las aplicaciones definidas, lo siguiente a realizar era el establecer un proceso por el cual se pudieran identificar los problemas en la aplicación, y a partir de esto, establecer las acciones a realizar con el fin de modificar la arquitectura de la aplicación hacia el estado de referencia definido.

Como se estableció durante la sección 7.4, el estado de la aplicación, en términos generales, es determinada por los estados de sus requerimientos de datos. Siendo así, se definió un proceso por el cual, a partir de las condiciones en las que se encuentren estos requerimientos,

Figura 20:
Arquitectura actual del proyecto



se determinen las acciones a realizar con el fin de adaptar la arquitectura.

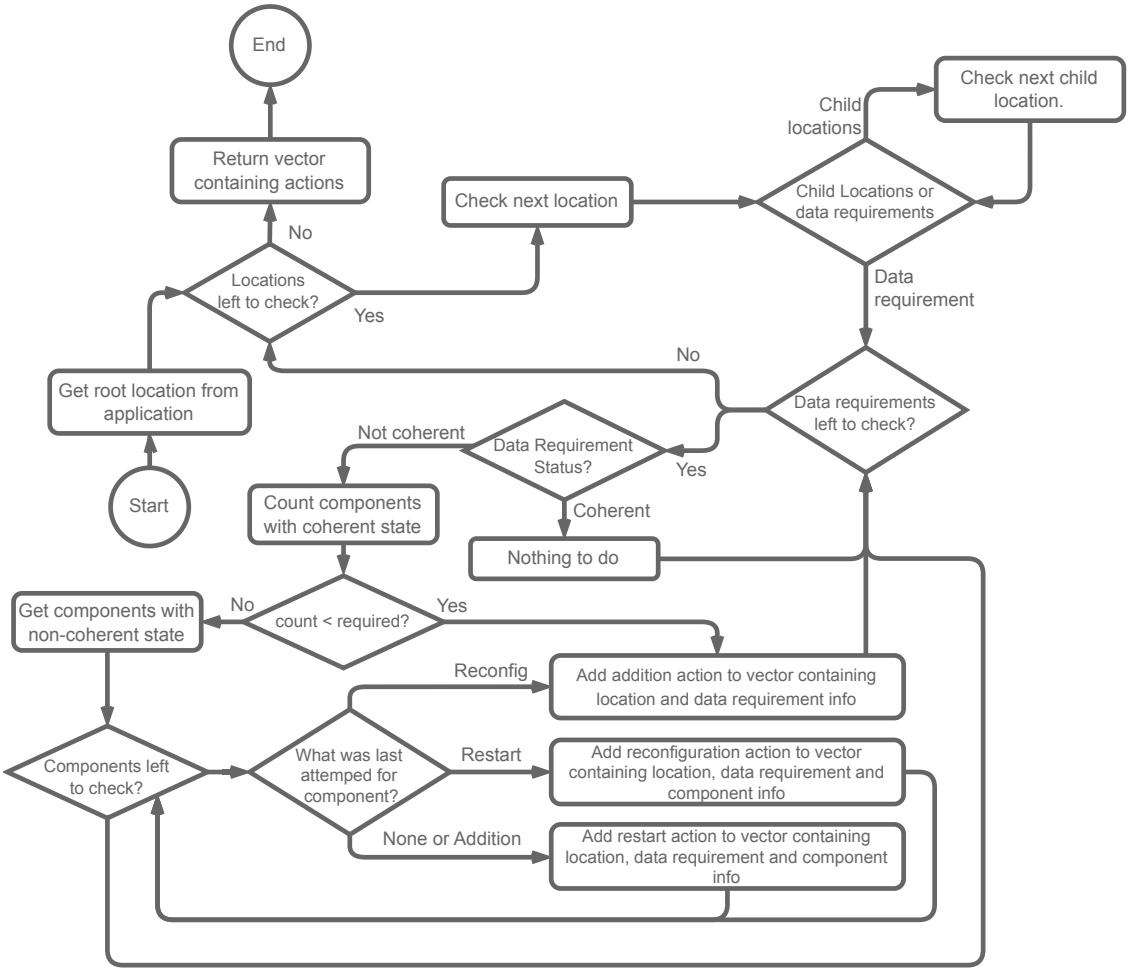
Partiendo de esto, se definió el proceso, descrito por la figura 21, en dos partes. La primera, está encargada de la búsqueda e identificación de los problemas dentro de la aplicación; y la segunda, de manera interna, el determinar las acciones a seguir para adaptar la arquitectura.

La manera en la de que se identifican los problemas es similar a la vista en la figura 19. Se recorren las locaciones, buscando requerimientos de datos. Una vez en una de estas locaciones, se validan los requerimientos y se definen las acciones.

Ahora, las posibles acciones a tomar para la arquitectura de la aplicación, como se vió durante la sección 3.1.3, dependen de los objetivos y los requerimientos del sistema. Siendo así, se tomaron 3 tipos de acciones por las cuales se puede modificar el estado del sistema: *Addition*, *Restart* y *Reconfigure*.

Todos los requerimientos en un estado diferente a *Coherent*, son evaluados con el fin de establecer la acción a realizar. Lo primero, es revisar la cantidad de componentes registrados en el requerimiento; si se tiene menos de los requeridos, se realizará un acción tipo *Addition* con el fin de cumplir con el numero de fuentes de datos esperada.

Figura 21:
Proceso para la identificación de problemas de los requerimientos de datos



En el caso de tener la cantidad requerida de dispositivos, pero estado aún en estado *Fault*; la siguiente opción es reiniciar el servicio. Esto busca poner nuevamente en un estado válido el componente, en caso de que haya quedado en un estado inválido; o poner nuevamente en ejecución el servicio en el escenario de que este haya muerto.

Si el reiniciar no tiene efecto en el estado del componente, la siguiente opción es reconfigurar el componente. Esto busca la modificación de algún aspecto, sea la actualización una variable de ambiente o parámetro, con el fin de retornarlo a un estado válido.

Finalmente, si todo falla y no es posible recuperar el dispositivo; la última opción es iniciar un nuevo servicio el cual pueda suplir las condiciones de su predecesor. De esta manera, aún

es posible retornar la aplicación hacia un estado válido, independiente de los componentes en un estado irrecuperable.

La implementación de este proceso, se realizó en el agregador *Bran*. Esto debido a que, al tener el estado de las aplicaciones, cortesía de lo observado por *Looker*; deja a este servicio como el punto medio de la aplicación, encargado de definir las acciones a realizar.

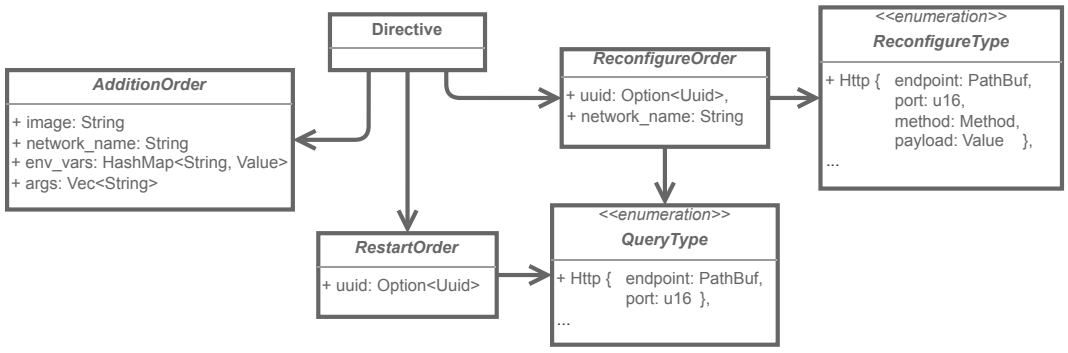
8.2 De Acciones, a Instrucciones

Ahora, se tienen las acciones a realizar para poder adaptar la arquitectura de la aplicación hacia el estado de referencia; sin embargo, estas acciones no contienen la información necesaria para ejecutar ninguno de los procesos necesarios.

Partiendo de esto, se determinó la necesidad de una colección estructuras de datos, que contuvieran las instrucciones a seguir, para cada una de las acciones definidas. Esta, debe especificar la locación de la aplicación en la cual se ejecutan, con el fin de tener una mejor granularidad y control sobre la manera en la que se realizan las modificaciones en la arquitectura.

En consecuencia, como se observa en la figura 22, se definió una estructura de datos llamada `Directive`. Esta contiene tres campos adicionales para las estructuras `AdditionOrder`, `RestartOrder`, y `ReconfigureOrder`; que corresponden con las acciones *Addition*, *Restart* y *Reconfigure*, respectivamente.

Figura 22:
UML de la estructura de directivas y ordenes



Cada una estas directivas, y ordenes, se agregan a un registro en *Bran*, usando el nombre de la aplicación y la locación a la pertenecen para su indexación. Cada una se registra usando un endpoint dedicado a la recepción de las órdenes.

La orden de adición contiene el nombre de la imagen del servicio a desplegar, la red a la cual debe conectarse para poder interactuar con los demás servicios, y los argumentos que se deben pasar sea como variables de ambiente o argumentos de línea de comando.

Las órdenes de reinicio y reconfiguración poseen el *UUID* del dispositivo que buscan afectar. Siendo así, al ser necesario tener una manera por la cual identificar qué servicio es qué componente, se implementó el enum `QueryType`. Este, contiene la información de como debe buscarse el contenedor en la red. Para esta primera implementación, su única variante es una búsqueda *Http* hacia un endpoint y un puerto.

Finalmente, la orden de reconfiguración tiene como parte extra la red en la que se buscara el servicio a reiniciar y el tipo de reconfiguración (`ReconfigureType`) a realizar. Este funciona de manera similar a QueryType, en tanto indica la manera en la que se aplicará la reconfiguración. En este caso, para la variante *Http* implementada, indica el endpoint y el puerto a usar, junto con el método y payload que deben ir en la petición para hacer la reconfiguración efectiva.

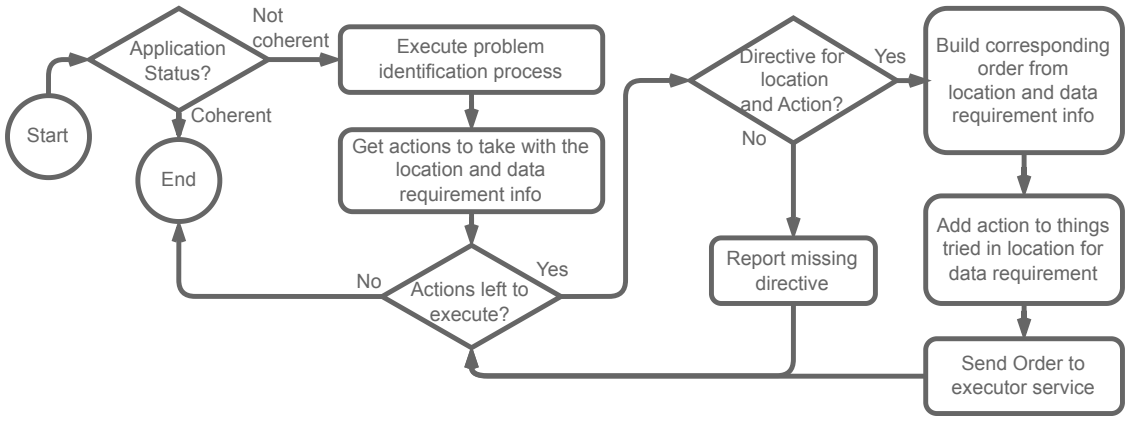
Se ha de resaltar que, estas ordenes, aunque contienen las instrucciones para poder realizar modificaciones a la arquitectura, en el momento de su declaración, no están completas y requieren de información extra para poder ejecutarse.

El proceso de la figura 23, muestra el proceso por el cual las órdenes presentes en el registro de directivas de *Bran*, son usadas como base para la construcción de las órdenes finales, y son enviadas para su posterior ejecución.

Su ejecución inicia validando el estado de la aplicación reportada. En caso de este sea diferente a *Coherent*, se llamará al proceso de identificación de problemas para obtener las acciones a realizar para alterar el estado de aplicación.

Una vez recibido el vector conteniendo las acciones, y la información asociada correspon-

Figura 23:
Proceso de decisión de acciones base realizado por Bran



diente, se realizará la construcción de cada una de las órdenes a ejecutar. Esta construcción se realiza a partir de las instrucciones bases establecidas en las directivas usando el UUID para las acciones de *Restart* y *Reconfigure*; y la información de la locación y el requerimiento de dato en las acciones de *Addition*[13].

Ya con las ordenes creadas, estas se enviaran al servicio encargado de ejecutar las instrucciones, dando como como resultado, modificaciones en la arquitectura.

8.3 De Instrucciones a Acciones

Ya con las instrucciones necesarias para poder realizar cambios en la arquitectura de las aplicaciones, se realizó la implementación de un servicio encargado de ejecutar las ordenes recibidas desde *Bran*.

Ahora, Smart Campus UIS, como plataforma, se ejecuta sobre contenedores de docker; y para efectos de esta primera versión del proyecto, toda la arquitectura de presente, se ejecuta sobre contenedores. Como consecuencia, era necesario que el servicio implementado, tuviera la de ejecutar comandos de docker con el fin de poder llevar a cabo las acciones.

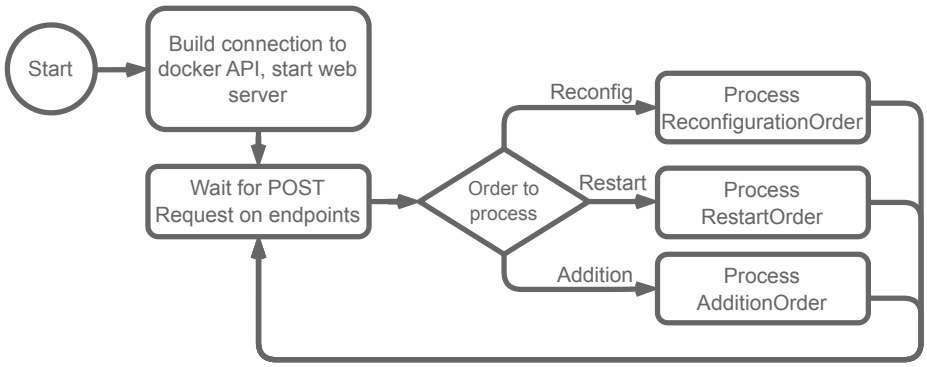
Esto se solucionó usando la Docker Engine API como el canal de comunicación entre

[13]Es necesario resaltar que estas construcciones, en especial para el caso de las acciones de *Addition*, fueron implementadas para funcionar con los servicios *Mocker* desarrollados específicamente para el proyecto. Esto se explorará más a fondo durante la sección 9 y 11.

el servicio, y el resto de la arquitectura. Esto se debe a que, esta API REST, permite la interacción directa con el daemon de Docker (Docker Inc, 2023), posibilitando la ejecución de todos los comandos relacionados con docker, desde el iniciar y detener servicios hasta crear y eliminar contenedores.

El servicio, apodado *DoThing*, realiza visto en la figura 24. Este está orientado principalmente al procesamiento de las peticiones realizadas, por lo que es relativamente sencillo en cuanto a las acciones que este realiza.

Figura 24: Proceso base de DoThing



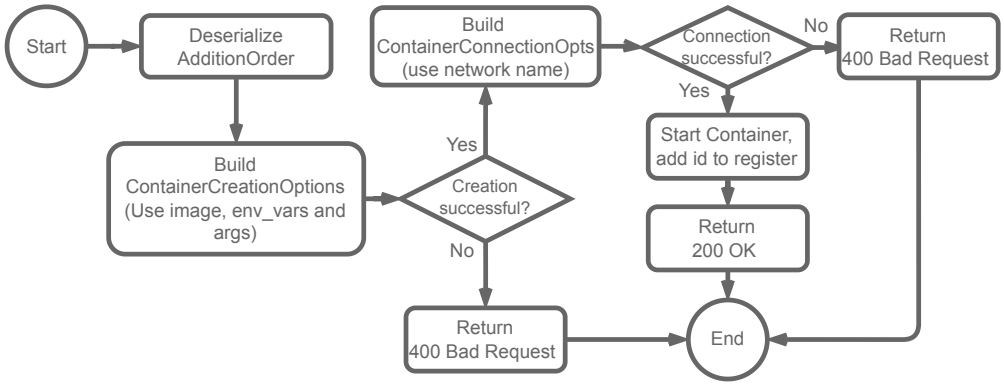
Tras inicializar la conexión con la API de Docker, iniciará un servidor web por el cual estará recibiendo las órdenes a ejecutar. Cada tipo de acción tiene su propio endpoint donde se realiza un proceso específico para poder ejecutarla. Cada acción realizada, registrará los contenedores afectados en un registro interno, con el fin de facilitar el procesamiento de múltiples acciones sobre un servicio.

Las ordenes de *Addition*, como se referencia en la figura 25, realizan el proceso a partir de la `image`, `env_vars` y `args`, declarados en la orden, para construir el contenedor con su respectivo servicio.

Una vez se tenga el contenedor creado, usando `network_name`, se conectará el contenedor creado a la red indicada. De no presentarse errores durante este proceso, se iniciará el contenedor, se agrega al registro y retorna `200 OK`.

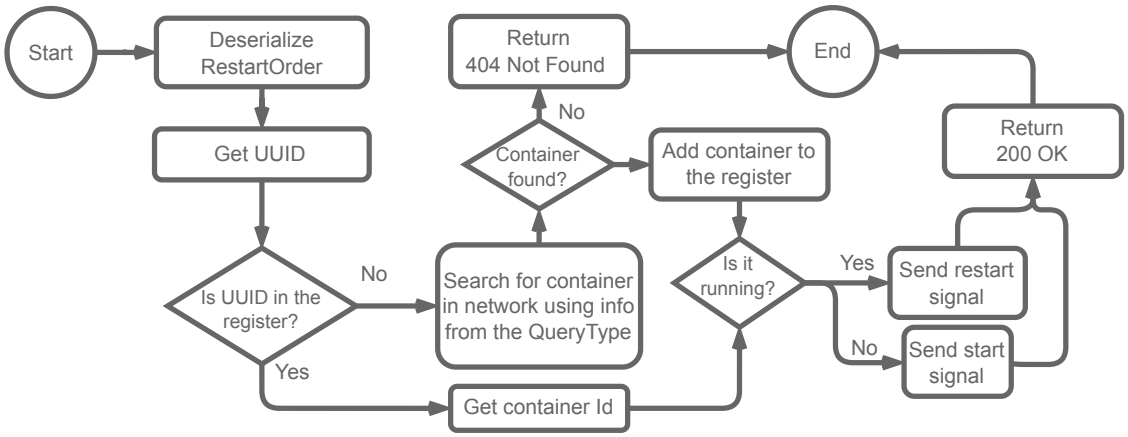
Las órdenes de tipo *Restart*, referenciadas en la figura 26 pueden ejecutarse de una de

Figura 25: Proceso para la ejecución de ordenes de adición



dos maneras, dependiendo del estado del servicio a afectar.

Figura 26: Proceso para la ejecución de ordenes de reinicio



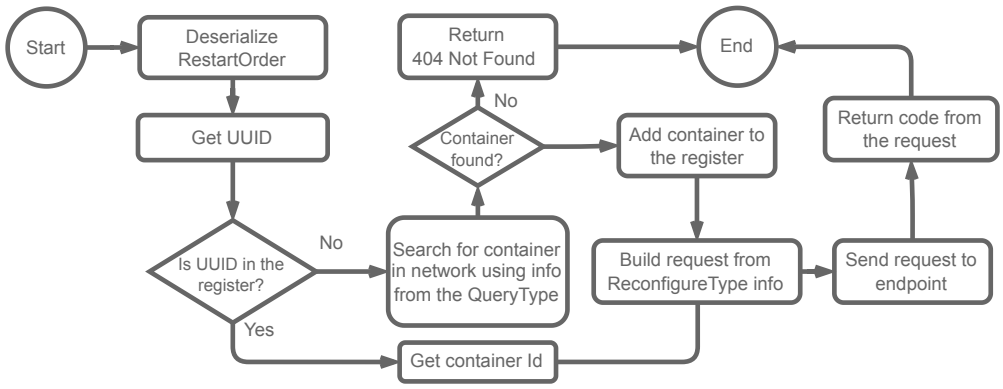
Primero, se realiza la búsqueda del contenedor en el registro; de no estar presente, se realizará la búsqueda de este usando la información proveída en el *QueryType* de la orden.

Una vez identificado el servicio, si está corriendo[14], la orden de reinicio será equivalente al comando `docker restart <container>`, esto con el objetivo de eliminar *cuelgues* en el servicio. Por el contrario, si el contenedor no está en ejecución, la orden ejecutará el equivalente al comando `docker start <container>`, para así iniciar nuevamente el servicio.

Finalmente, las órdenes de tipo *Reconfigure*, cuyo proceso puede verse en la figura 27, es similar al realizado para las órdenes de tipo *Restart*.

[14]Referido en la documentación de Docker Engine API como *Running*

Figura 27: Proceso para la ejecución de ordenes de reconfiguración



Una vez ubicado el servicio, se construirá, a partir de los datos reportados en *ReconfigureType*, la petición a realizar en el servicio. Esta, en consecuencia, se enviará usando un cliente *Http* y esperará a la respuesta de esta. El resultado, de esta transacción, será el mismo reportado por *DoThing* hacia *Bran*.

Ahora, con la implementación de este servicio, encargado de ejecutar las acciones requeridas para la adaptación de las arquitecturas de las aplicaciones, completa el ciclo autonómico MAPE-K implementado para Smart Campus UIS.

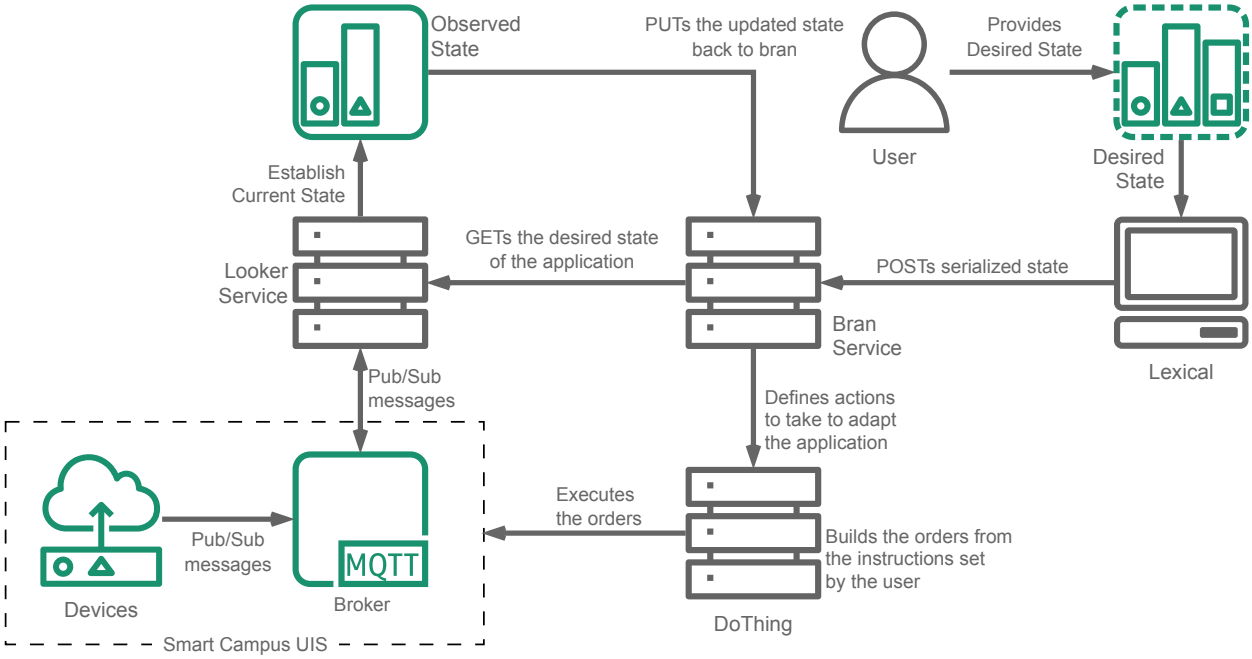
Una vez ejecutadas las órdenes; el efecto de estas, serían captadas por el observador, evaluando nuevamente el estado de la aplicación, e, idealmente, reportando un cambio positivo hacia la coherencia esperada con el estado de referencia definido en un principio.

La arquitectura final del presente proyecto, con toda la implementación de los servicios, puede ver se en la figura 28.

9 Validación de la Implementación

Lo último a realizar fue la validación de las funcionalidades de la aplicación desarrollada. Para ello, se establecieron escenarios de prueba por los cuales se pueda comprobar el funcionamiento de aplicación. Estos, se definieron con el fin de probar la capacidad y la efectividad de los mecanismos implementados de modificar el estado inicial de la aplicación hacia el estado de referencia, al igual que la identificación de algunas de las falencias de esta.

Figura 28: Arquitectura completa del proyecto



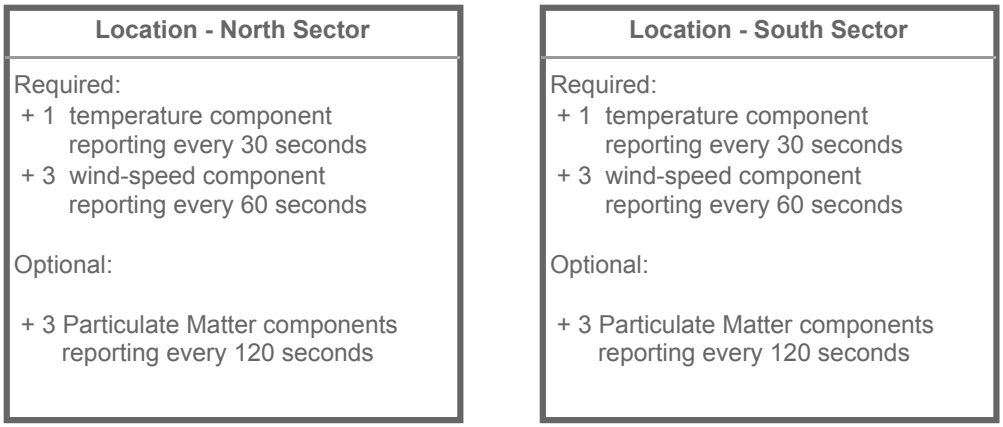
Ahora, se debe resaltar que, las pruebas se realizaron usando un servicio genérico, apodado Mocker, encargado de la generación de datos, especialmente desarrollado para funcionar en condiciones ideales.

Para cada una de estas pruebas, se definió un estado de referencia, dadas unas condiciones iniciales, y una serie de eventos los cuales modificarán nuevamente el estado de la aplicación, las cuales tendrán que ser nuevamente manejadas.

9.1 Escenario Experimental A

El primer escenario, busca el evaluar la capacidad de la implementación realizada de iniciar toda una aplicación desde cero. Partiendo de esto, se definió la aplicación vista en la figura 29. Esta tiene dos locaciones raíz, con requerimientos de datos iguales en ambas.

Figura 29:
Diagrama del escenario experimental A



Inicialmente, no habrán dispositivos presentes en ejecución, por lo que el sistema tendrá que identificar la falla, definir las acciones de adición correspondientes, y ejecutarlas para poder suplir con los requerimientos de datos.

Para ello, tomando como referencia la figura 29, se realizó la declaración del estado de referencia, presente en el anexo 1, y, como observa en la figura 30, se validó usando *Lexical*.

Figura 30:
Validación de la declaración de referencia realizada para el escenario experimental A

```
INFO  lexical] Lexical started
INFO  lexical::parsing::parser] Loading file
INFO  lexical::parsing::parser] YAML File loaded
INFO  lexical::parsing::parser] Creating Application instance
INFO  lexical::parsing::parser] Application instance created
INFO  lexical::parsing::parser] Creating Locations
INFO  lexical::location::location] Processing temperature in Sector Norte
INFO  lexical::location::location] Processing wind-speed in Sector Norte
INFO  lexical::location::location] Processing particulate-matter in Sector Norte
INFO  lexical::location::location] Processing wind-speed in Sector Sur
INFO  lexical::location::location] Processing particulate-matter in Sector Sur
INFO  lexical::location::location] Processing temperature in Sector Sur
INFO  lexical::parsing::parser] Locations created
INFO  lexical] Application Malaga definition was validated!
```

Ya con las condiciones iniciales, y el estado de referencia definido, se estructuraron, las directivas, vistas en el anexo 2, que permitirían realizar la adaptación de la arquitectura hacia el estado de objetivo.

Partiendo de todo lo anterior, se definió la serie de pasos a realizar para este primer escenario experimental:

1. Desplegar los servicios de Smart Campus UIS

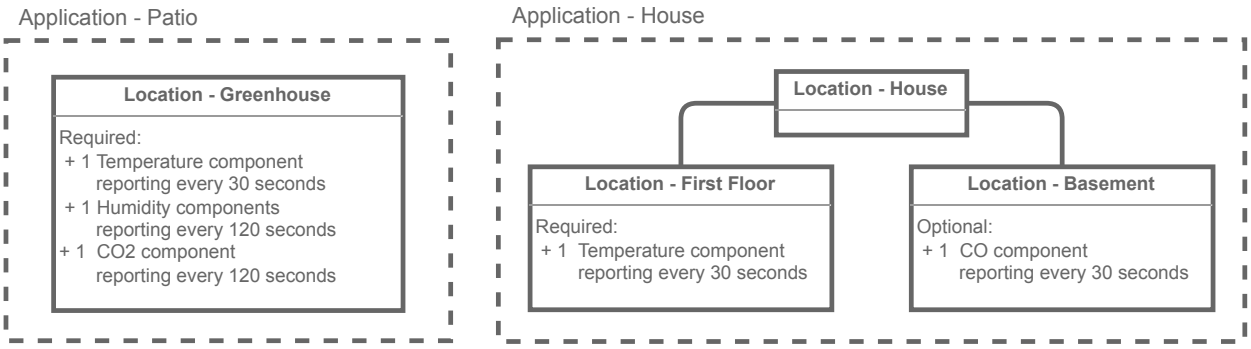
2. Desplegar los servicios *Bran* y *DoThing*.
3. Usando *Lexical*, establecer el estado de referencia de la aplicación.
4. Declarar en los endpoints de *Bran*, las directivas definidas.
5. Desplegar el servicio *Looker* para la aplicación.
6. A la nivelación del estado de la aplicación.

Al final de la simulación de este escenario, se eliminarán todos los servicios desplegados con el fin de asegurar una ejecución limpia de futuros procesos.

9.2 Escenario Experimental B

El segundo escenario, busca el evaluar la capacidad de la implementación en donde ya existen componentes en ejecución, al igual que el manejo de múltiples aplicaciones. Para esto, se definieron dos aplicaciones, como se observa en la figura 31,

Figura 31:
Aplicaciones definidas para el desarrollo del escenario experimental B



Para esta simulación, todos los componentes de la aplicación ya estarán ejecutándose, con el problema de que no todos estarán dentro de los parámetros establecidos. Esto requerirá que la aplicación identifique el servicio correspondiente, y realice la reconfiguración de sus parámetros.

De la misma manera que se realizó en escenario experimental A, se realizó la declaración de los estados de referencia de cada una de las aplicaciones, al igual que las directivas para

su ejecución. Los YAML de esta declaración, y las directivas definidas, se encuentran en los apéndices 3, 4, 5 y 6.

Partiendo de todo lo anterior, se definió la serie de pasos a realizar para este primer escenario experimental:

1. Desplegar los servicios de Smart Campus UIS
2. Desplegar los servicios *Bran* y *DoThing*.
3. Usando *Lexical*, establecer el estado de referencia de la aplicación Patio y House.
4. Declarar en los endpoints de *Bran*, las directivas definidas.
5. Desplegar los servicios de simulación de datos *Mocker* respecto a las expectativas, tal que el estado sea coherente.
6. Desplegar los servicios *Looker* para cada una de las aplicaciones.
7. Matar los servicios desplegados y eliminar uno de los contenedores creados.
8. Esperar a ejecución de las órdenes *Restart* y *Addition*.

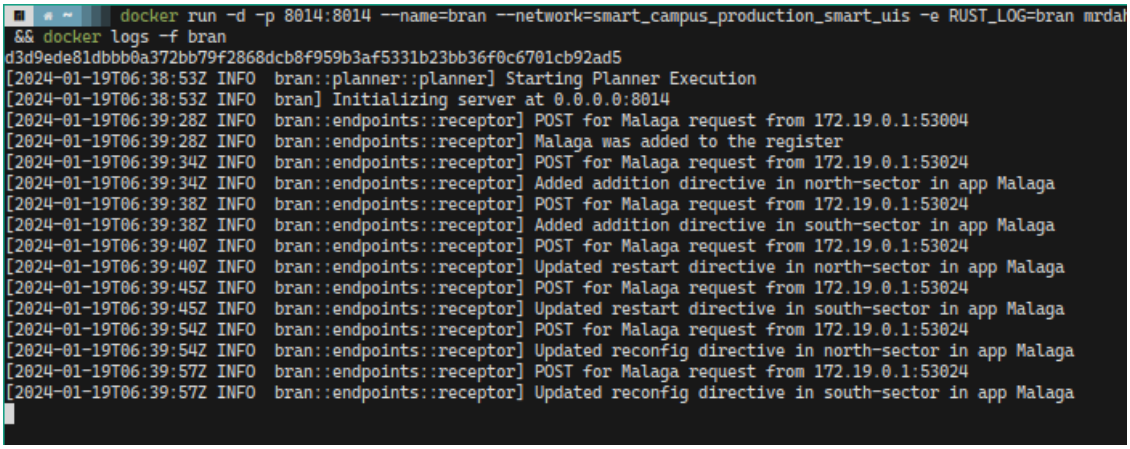
10 Análisis de Resultados

10.1 Resultados De Escenario Experimental A

El objetivo de las pruebas realizadas en el presente escenario experimental, era validar que, los procesos implementados, efectivamente realizaran la adaptación de las aplicaciones desde un estado de completa falla, en donde ninguno de los datos necesarios estuviera presente.

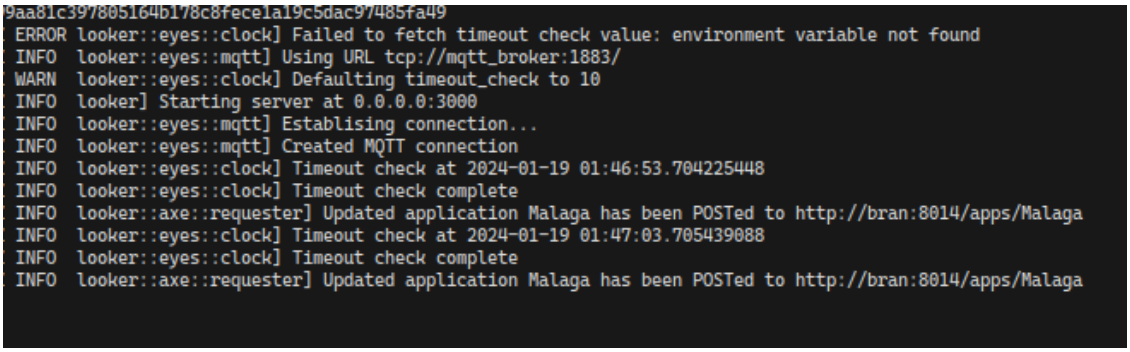
Siguiendo los pasos definidos en la sección 9.1, se realizó el despliegue de los servicios base de Smart Campus UIS, al igual que *Bran* y *DoThing*. Como se presenta en la figura 33, se registró el estado de referencia al agregador tras validarlo usando *Lexical*, y se fijaron las directivas a usar para la adaptación de la arquitectura.

Figura 32:
Aplicación y directivas registradas en Bran



Una vez se inició el servicio *Looker*, como se ve en la figura 33, este empezó a evaluar el estado de la aplicación; que al no tener ningún tipo de reporte de algún componente, entró directamente en estado de falla.

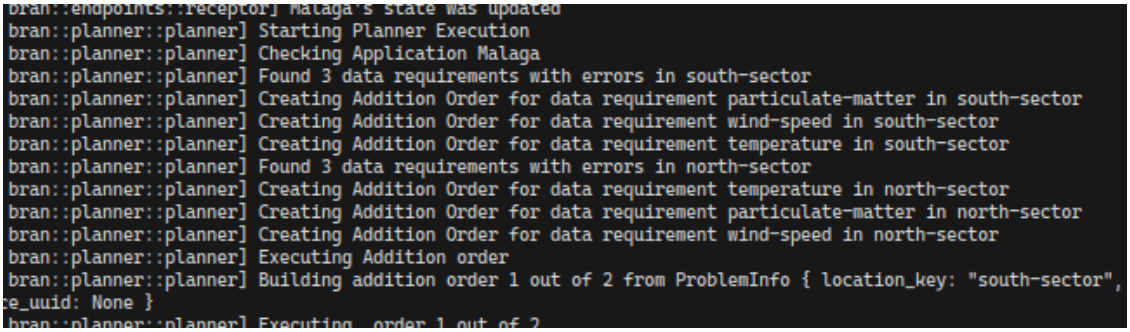
Figura 33:
Looker evalúa el estado de la aplicación



El estado observado, se reportó a *Bran*, el cual, realizó el recorrido para identificar los puntos de falla de la aplicación. Como se observa en la figura 34, como era de esperar, los tres requerimientos definidos en cada una de las locaciones estaba en estado de falla. Al no haber componentes reportando datos, como era de esperarse, se definieron acciones *Addition* para todos los componentes de la aplicación.

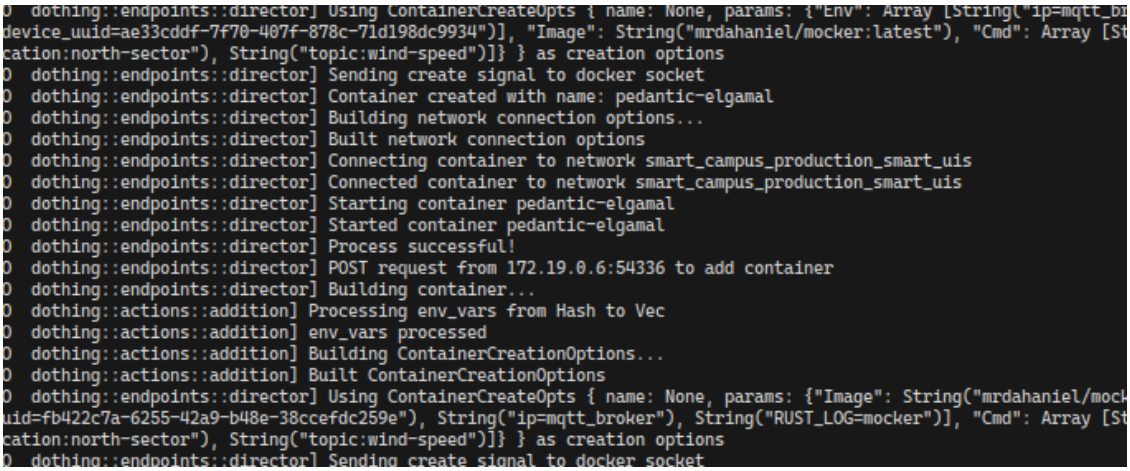
A partir de las acciones, se crearon las órdenes para ejecución usando las directivas como base. Como se puede ver en la figura 35, cada una de estas se envió a *DoThing*, el cual realizó el procesamiento requerido para la construcción y arranque de los contenedores. El resultado

Figura 34:
Bran identifica los problemas y establece las acciones



de este procesamiento se puede ver en la figura 36

Figura 35:
DoThing ejecuta las acciones establecidas por Bran

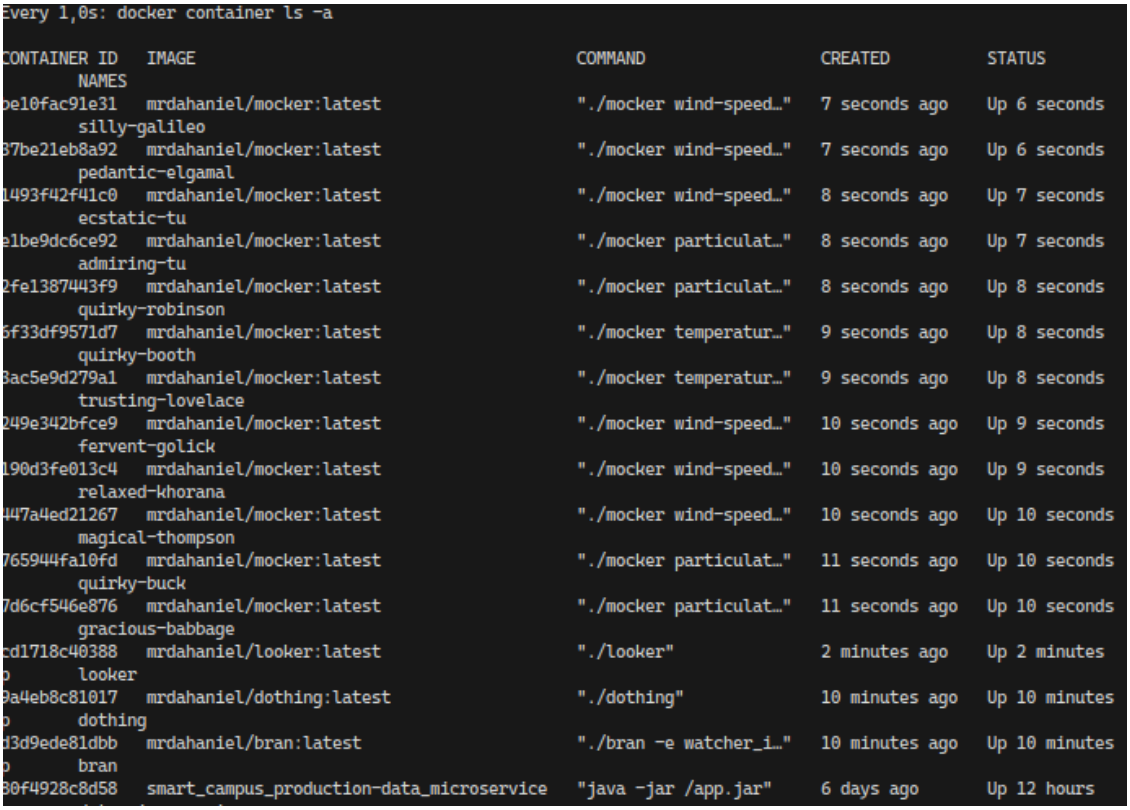


En total, se crearon 12 contenedores, que corresponden a los 12 componentes requeridos para el desarrollo de la aplicación. Cada uno de ellos configurado para suplir las necesidades de operación establecidas en el estado de referencia. Estos despliegues, como era de esperarse, fueron detectado por *looker*, el cual realizó el nuevas valoraciones del estado.

Los valores por defecto de los servicios usados para estas pruebas es un intervalo de tiempo entre mensajes de un minuto. Este valor por defecto, aunque para los requerimientos de datos cuyos *timeout*, sean superiores, supliría las necesidades de la aplicación. Sin embargo, en el marco de esta simulación, esta frecuencia de reportes no es suficiente para cumplir con el estado de referencia de los requerimiento de temperatura de las locaciones.

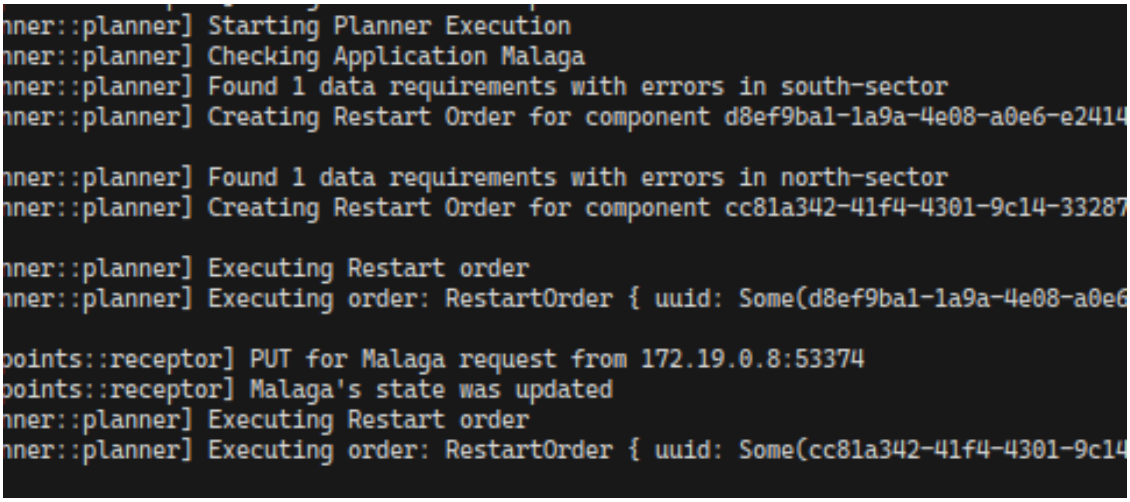
Como consecuencia, el estado de la aplicación regresó a estado *Fault*, y *Bran*, al detectar

Figura 36:
Contenedores creados por DoThing en respuesta a las órdenes recibidas



los problemas, emitió las acciones tipo *Restart* con el fin de traer los contenedores a un estado válido. Esto se puede ver en la figura 37.

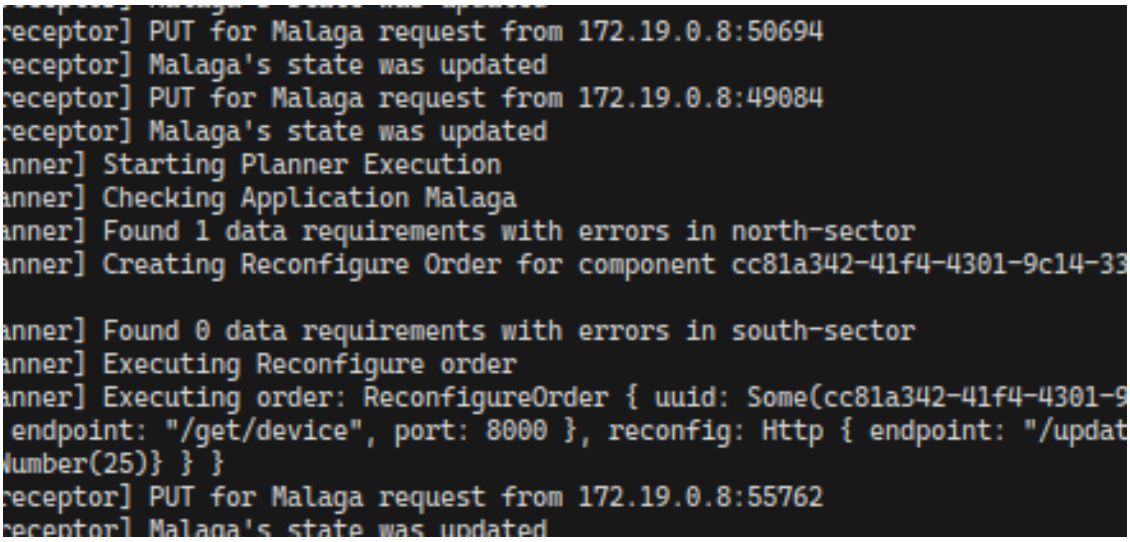
Figura 37:
Bran identifica otros problemas en la aplicación



Este intento de reiniciar los servicios, como era de esperarse, no fue realmente efectiva

debido a que la falla se debe a un problema en la configuración del servicio. Siendo así, la aplicación se mantuvo en estado de falla. Se esperaba que, ambos componentes encargados de la temperatura, al ya haber descartado el reinicio como un mecanismo efectivo para esta situación, fueran sometidos a una acción de reconfiguración, sin embargo, como se observa en la figura 38, en ese instante, sólo el componente de `north-sector` fue afectado.

Figura 38:
Bran establece acciones de reconfiguración para adaptar la aplicación



Esta reconfiguración fue ejecutada por *DoThing*, y tomó efecto, alterando el valor por defecto entre mensajes, pasando de un reporte cada minuto, a uno cada 25 segundos[15]. Con el nuevo valor, los componentes de temperatura, empezaban a reportar con mayor frecuencia, cumpliendo con las expectativas.

Poco tiempo después, el componente de temperatura de `south-sector`, reportó el estado de falla, por lo se ejecutó la misma acción de reconfiguración que con el componente de la otra locación, llevando el estado de la aplicación a *Coherent*.

A partir de esta simulación, se pudo validar que el proceso implementado, tiene la capacidad de realizar la adaptación desde el peor escenario, hacia un esto de coherencia con la aplicación declarada inicialmente. Todas las acciones definidas por *Bran*, permitieron modificar el estado de manera positiva.

[15]Según lo definido en las directivas presentes en el apéndice 2

Sin embargo, se identificaron algunas limitaciones de esta implementación. Como se observa en la figura 38, se muestra que `south-sector` no presenta problemas en sus requerimientos, cuando, debido a su configuración, debía estar en estado de falla.

Esto se debe a un *quirk* de la implementación realizada. Al manejar los procesos cada cierto intervalo de tiempo, es posible que un componente se encuentren en estado de falla, pero alcance a pasar a coherente justo antes de la evaluación de las acciones a tomar. Eventualmente, el componente fallaría en reportar antes de una una de las evaluaciones desencadenando en una acción, como fue el caso con la tardía reconfiguración del componente de `south-sector`.

10.2 Resultados De Escenario Experimental B

El objetivo principal del escenario experimental B, era evaluar la capacidad de la implementación realizada de recuperar una aplicación en estado de coherencia, en donde se presenta una falla masiva, nuevamente a un estado válido; al igual que la capacidad de Bran de manejar más de una aplicación, como se había nombrado en la sección 7.2.

Igual que durante el escenario experimental A, se realizó el despliegue de los servicios *Bran* y *DoThing*. Como se observa en la figura 39, se definieron las arquitecturas objetivo de las dos aplicaciones a monitorear, al igual que las directivas a usar durante este proceso.

Seguidamente, como se observa en la figura 40, se desplegaron los servicios de *Looker* para cada una de las aplicaciones a monitorear. Ya con esta base lista, y como se definió en los pasos a seguir, se desplegaron de manera manual, un total de 3 servicios *Mocker* con configuraciones específicas (tanto de los datos que estos reportaban, como de los tiempos de reporte) con el fin de cumplir con los requerimientos de datos definidos para las dos aplicaciones.

Figura 39:
Aplicaciones y directivas registradas en Bran

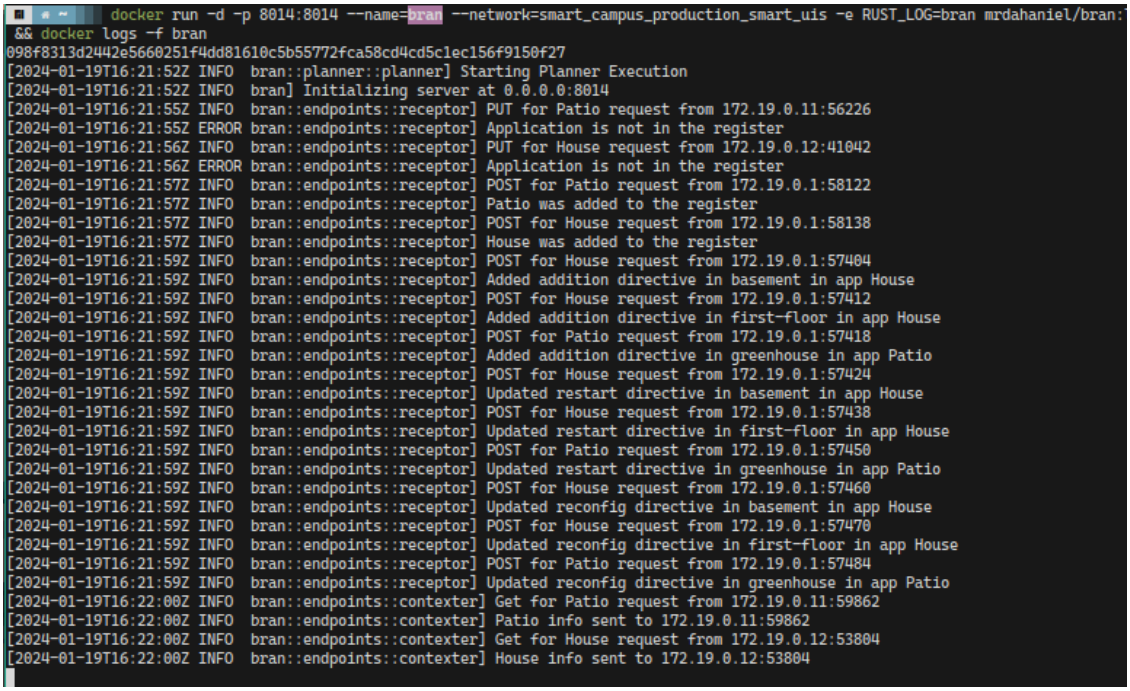
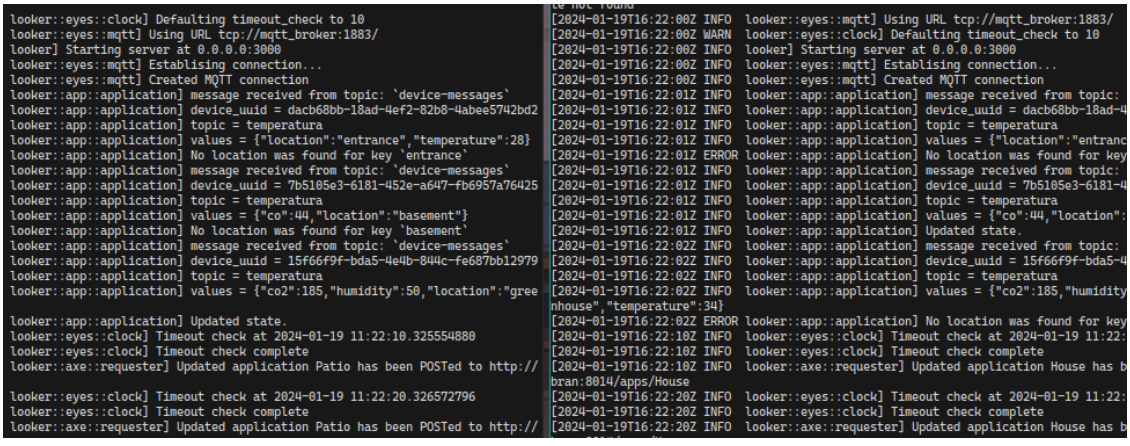
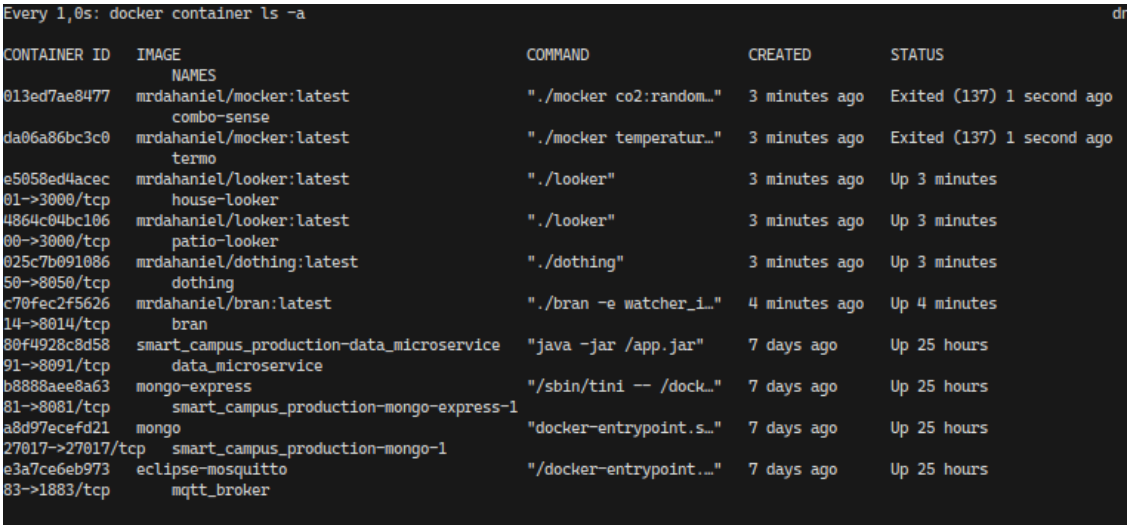


Figura 40:
Dos instancias de looker monitoreando las aplicaciones declaradas



Como era de esperarse, con las configuraciones hechas a la medida de las aplicaciones, el estado de estas era *Coherent*, sin la necesidad de ningún tipo de cambios en los dispositivos. Siendo así, como se ve en la figura 41, se matan los 3 contenedores desplegados, y se elimina uno de ellos.

Figura 41:
Se matan los contenedores con los servicios iniciales

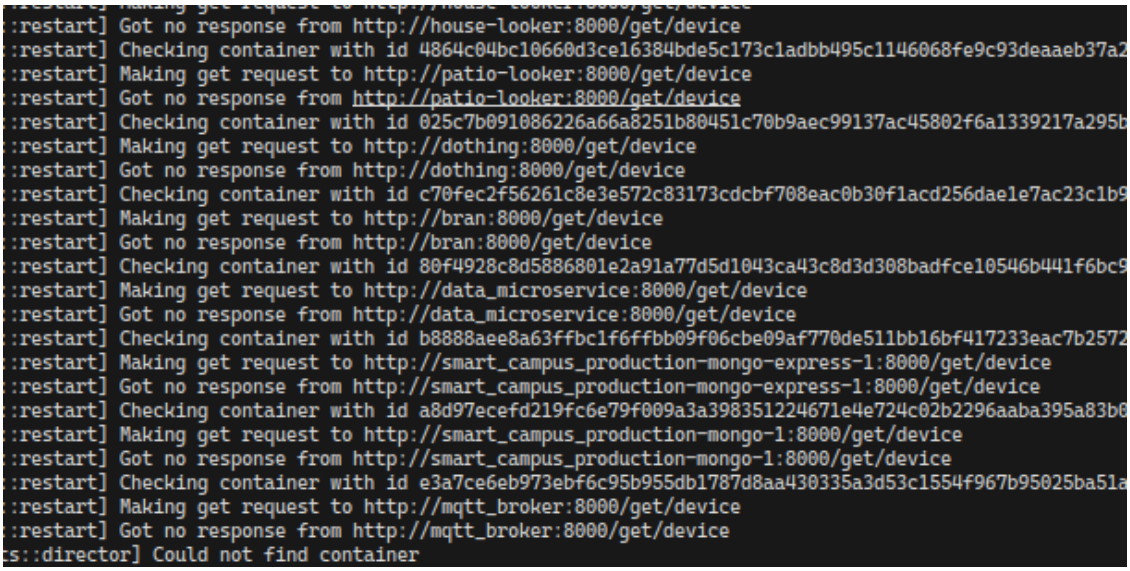


La figura 42, muestra el como *Bran*, define las órdenes para reiniciar los servicios para traer la aplicación a un estado nuevamente. En condiciones normales, se esperaría que esto sea lo único a realizar para adaptar la arquitectura, sin embargo, este no fue el caso.

Figura 42:
Bran intenta reiniciar los contenedores

Ya que los contenedores no fueron desplegados, o ha sido modificados, de ninguna manera por *DoThing*, no estos no están mapeados. Siendo así, como se ve en la figura 43, es necesario buscar los contenedores en la red, pero, al estar los servicios muertos, y depender de la consulta *Http* para poder identificarlos, no puede encontrarlos.

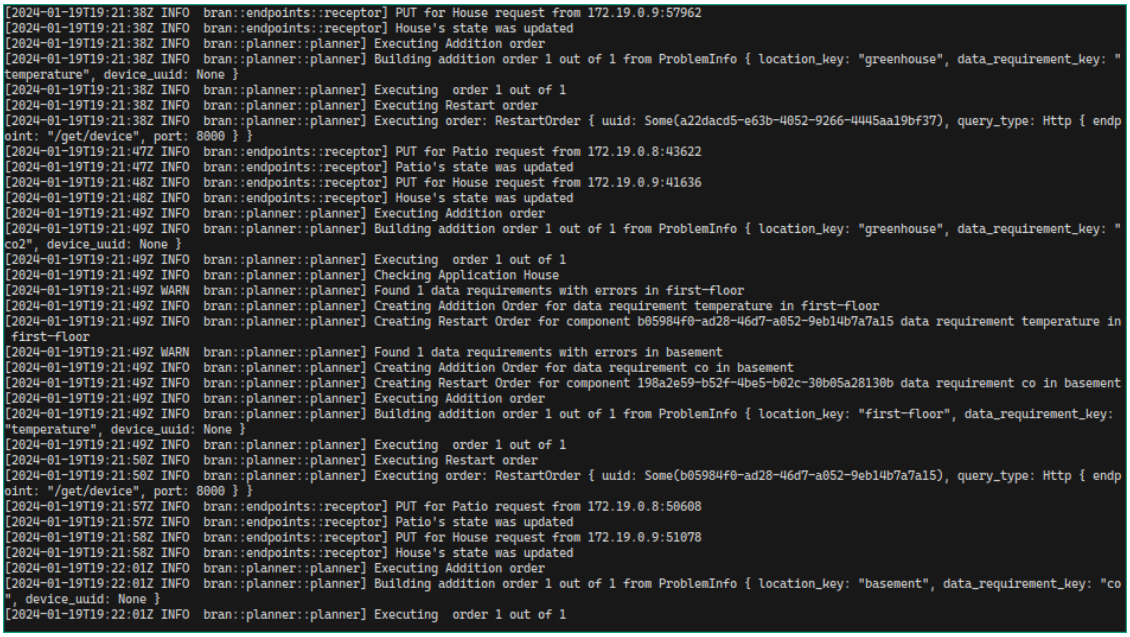
Figura 43:
DoThing no puede encontrar los contenedores



Esto demuestra otra de las falencias con la implementación realizada, que es la búsqueda del crecimiento de la fase de conocimiento. El haber identificado los componentes, antes de que se presentaran problemas, hubiera posibilitado la ejecución de órdenes cuando los servicios estén caídos. Así mismo, se hubiera podido establecer una búsqueda que no dependiera del estado de los componentes, quizás usando tags.

El resultado final, es que *Bran*, al no poder actuar de otra manera, como se observa en la figura 44, ordena acciones de adición para suplir los componentes. Esto, eventualmente, resulta en el regreso de la aplicación a un estado válido, pero sigue sin ser lo ideal.

Figura 44:
Bran crea órdenes de adicción



11 Conclusiones y Trabajo Futuro

El presente trabajo buscaba realizar la implementación de mecanismos de adaptación de arquitecturas a la plataforma Smart Campus UIS, que permitiera alterar el estado de una aplicación hacia un estado de referencia, de manera autonómica.

Para ello, se desarrolló e implementó un metamodelo capaz representar dichas arquitecturas, al igual que una manera de declararla tomando como foco los datos requeridos para su correcto funcionamiento; el cual cumple con el objetivo de establecer un estado de referencia.

También se diseño, e implementó, un servicio capaz de interpretar los mensajes de los dispositivos y evaluar el estado de la aplicación; al igual que una proceso de identificar los problemas y definir las acciones a tomar; cumpliendo con el objetivo de determinar una manera por la cual se pudieran realizar dichas comparaciones.

Así mismo, se diseñaron mecanismos que dieran la capacidad de modificar la arquitectura, hacia el estado de referencia definido; cumpliendo con el objetivo de reducir la cantidad de diferencias entre el estado observado, y el deseado.

Finalmente, se validaron las implementaciones realizadas, al igual que se identificaron

las falencias de las mismas, dando paso a un nuevo foco de trabajo, ajustando, afinando y extendiendo las maneras en las que se pueden adaptar las arquitecturas de manera autonómica, al igual que estandarizando y simplificando la manera en la que se trabajan con estas herramientas.

De esta primera implementación, rara vez se sale de una línea de comando, por lo que el desarrollo de interfases para los diversos servicios implementados, con el fin de facilitar la accesibilidad a la plataforma.

Así mismo, la extensión de manera de evaluar los estados del sistema, a partir de diferentes datos definidos por el usuario, como uso de promedios para flexibilizar las fallas o contadores, antes de tomar acciones para ignorar fallos temporales, son puntos importantes a mejorar en futuras versiones.

También se ha de considerar la reimplementación de algunos de los mecanismos, con el fin de aumentar su eficacia en modificar los estados. Al igual que la expansión de la base de conocimiento, con el fin de aprovechar todos los recursos disponibles, independiente del estado de algunos componentes.

Se recomienda el buscar formas de permitir, al usuario, definir procesamientos de datos extra para la evaluación de el estado de referencia. Idealmente de manera agnóstica al lenguaje de programación usado. Lo cual daría un valor extra a la flexibilidad de la plataforma.

Finalmente, se recomienda la realización de pruebas más extensas, con dispositivos no simulados, que permitan evolucionar este primer prototipo de plugin para Smart Campus UIS, a algo que se pueda desplegar con relativa facilidad, independientemente del la aplicación que se esté desarrollando.

Referencias

Allied Market Research. (2023). *Location based services market.* Online. Descargado de https://www.alliedmarketresearch.com/location-based-services-market

Anagnostopoulos, T. (2023). Smart campus. En *Iot-enabled unobtrusive surveillance systems for smart campus safety* (p. 17-25). doi: 10.1002/9781119903932.ch3

Andre, C. (2007). *Uml extensions: the sysml profile.* Descargado 2023-05-19, de https://www.i3s.unice.fr/~map/Cours/MASTER_TSM/Cours7_SysML.pdf (Presentation on UML extensions)

Arcaini, P., Riccobene, E., y Scandurra, P. (2015). Modeling and analyzing mape-k feedback loops for self-adaptation. En *2015 ieee/acm 10th international symposium on software engineering for adaptive and self-managing systems* (p. 13-23). doi: 10.1109/SEAMS.2015.10

Ashraf, Q. M., Tahir, M., Habaebi, M. H., y Isoaho, J. (2023). Toward autonomic internet of things: Recent advances, evaluation criteria, and future research directions. *IEEE Internet of Things Journal*, *10*(16), 14725-14748. doi: 10.1109/JIOT.2023.3285359

Bansal, A. K. (2013). *Introduction to programming languages.* Boca Raton, FL: CRC Press.

Berte, D.-R. (2018, 05). Defining the iot. *Proceedings of the International Conference on Business Excellence*, *12*, 118-128. doi: 10.2478/picbe-2018-0013

Carnegie Mellon University. (2017). *Aadl and osate: A tool kit to support model-based engineering.* Online. Carnegie Mellon University. Descargado de https://resources.sei.cmu.edu/asset_files/FactSheet/2017_010_001_506838.pdf

Carnegie Mellon University. (2022). *Architecture analysis and design language.* Carnegie Mellon University. Descargado de https://www.sei.cmu.edu/our-work/projects/display.cfm?customel_datapageid_4050=191439%2C191439

Celikovic, M., Dimitrieski, V., Aleksic, S., Ristić, S., y Luković, I. (2014). A dsl for eer data model specification. En *Integrated spatial databases.*

Costa, B., Pires, P. F., y Delicato, F. C. (2016). Modeling iot applications with sysml4iot. En *2016 42th euromicro conference on software engineering and advanced applications (seaa)* (p. 157-164). doi: 10.1109/SEAA.2016.19

Dawood, A. (2020, 09). Internet of things (iot) and its applications: A survey. *International Journal of Computer Applications*, *175*, 975-8887. doi: 10.5120/ijca2020919916

Deichmann, J., Doll, G., Klein, B., Mühlreiter, B., y Stein, J. P. (2022, Mar). *Cracking the complexity code in embedded systems development.* McKinsey's Advanced Electronics Practice.

Docker Inc. (2023). *Develop with docker engine api.* Online. Docker. Descargado de `https://docs.docker.com/engine/api/`

Durán-Polanco, L., y Siller, M. (2023). A taxonomy for decision making in iot systems. *Internet of Things*, *24*, 100904. Descargado de `https://www.sciencedirect.com/science/article/pii/S2542660523002275` doi: https://doi.org/10.1016/j.iot.2023.100904

Díaz, G., Steffenel, L. A., Barrios, C. J., y Couturier, J.-F. (2024). *How to build a software quantum simulator.* Preprints. Descargado de `https://doi.org/10.20944/preprints202409.1497.v1` doi: 10.20944/preprints202409.1497.v1

Eclipse Foundation. (2018). *Eclipse mita.* Eclipse Foundation. Descargado 2023-05-13, de `https://www.eclipse.org/mita/`

Friedman, D. P., y Wand, M. (2008). *Essentials of programming languages, 3rd edition* (3.ª ed.). The MIT Press.

Gartner, G., y Huang, H. (Eds.). (2015). *Progress in location-based services 2014.* Springer International Publishing. Descargado de `https://doi.org/10.1007/978-3-319-11879-6` doi: 10.1007/978-3-319-11879-6

Gorla, A., Pezzè, M., Wuttke, J., Mariani, L., y Pastore, F. (2010, 01). Achieving cost-effective software reliability through self-healing. *Computing and Informatics*, *29*, 93-115.

Grochowski, K., Breiter, M., y Nowak, R. (2019, 08). Serialization in object-oriented programming languages.. doi: 10.5772/intechopen.86917

Harrand, N., Fleurey, F., Morin, B., y Husa, K. (2016, 10). Thingml: a language and code generation framework for heterogeneous targets. En (p. 125-135). doi: 10.1145/2976767.2976812

Heath, S. (2002). *Embedded systems design*. Elsevier.

Horn, P. (2001, Oct). *autonomic computing: Ibm's perspective on the state of information technology.* IBM. Descargado de `https://homeostasis.scs.carleton.ca/~soma/biosec/readings/autonomic_computing.pdf`

Huynh, N.-T. (2019). An analysis view of component-based software architecture reconfiguration. En *2019 ieee-rivf international conference on computing and communication technologies (rivf)* (p. 1-6). doi: 10.1109/RIVF.2019.8713678

Iancu, B., y Gatea, A. (2022). Towards a self-describing gateway-based iot solution. En *2022 ieee international conference on automation, quality and testing, robotics (aqtr)* (p. 1-5). doi: 10.1109/AQTR55203.2022.9801938

IoT-A. (2014). *Internet of things architecture.* Online. Descargado 2023-05-20, de `https://www.iot-a.eu/public/`

Jiménez Herrera, H., Cárcamo, E., y Pedraza, G. (2020). Extensible software platform for smart campus based on microservices. *RISTI - Revista Iberica de Sistemas e Tecnologias de Informacao*, *2020*(E38), 270-282. Descargado de `www.scopus.com`

Jiménez Herrera, H. A. (2022). *Mecanismos autónomos para la autodescripción de arquitecturas iot distribuidas.* Descargado de `https://noesis.uis.edu.co/items/c5d1cce7-3f8f-4e3e-a083-66bd3c4745f3`

Jiménez Herrera, H. A. (2023). *Smart campus uis production.* Online. GitHub. Descargado de `https://github.com/UIS-IoT-Smart-Campus/smart_campus_production`

Kabashkin, I. (2017). Dynamic reconfiguration of architecture in the communication network of air traffic management system. En *2017 ieee international conference on computer*

and information technology (cit) (p. 345-350). doi: 10.1109/CIT.2017.13

Kelly, S., y Tolvanen, J.-P. (2008). *Domain-specific modeling*. Hoboken, NJ: Wiley-Blackwell.

Khaled, A. E., Helal, A., Lindquist, W., y Lee, C. (2018). Iot-ddl–device description language for the "t" in iot. *IEEE Access*, *6*, 24048-24063. doi: 10.1109/ACCESS.2018.2825295

Kirchhof, J. C., Rumpe, B., Schmalzing, D., y Wortmann, A. (2022a). *Montithings.* Online. Descargado 2023-05-13, de `https://github.com/MontiCore/montithings`

Kirchhof, J. C., Rumpe, B., Schmalzing, D., y Wortmann, A. (2022b). Montithings: Model-driven development and deployment of reliable iot applications. *Journal of Systems and Software*, *183*, 111087. Descargado de `https://www.sciencedirect.com/science/article/pii/S0164121221001849` doi: https://doi.org/10.1016/j.jss.2021.111087

Krikava, F. (2013, 11). *Domain-specific modeling language for self-adaptive software system architectures.*

Lalanda, P., Diaconescu, A., y McCann, J. A. (2014). *Autonomic computing: Principles, design and implementation*. Springer.

Liu, H., Parashar, M., y Hariri, S. (2004). A component-based programming model for autonomic applications. En *International conference on autonomic computing, 2004. proceedings.* (p. 10-17). doi: 10.1109/ICAC.2004.1301341

Merriam, Webster. (2023a). *Notation.* Descargado de `https://www.merriam-webster.com/dictionary/notation`

Merriam, Webster. (2023b). *Syntax.* Descargado de `https://www.merriam-webster.com/dictionary/notation`

Min-Allah, N., y Alrashed, S. (2020, agosto). Smart campus—a sketch. *Sustainable Cities and Society*, *59*, 102231. Descargado de `https://doi.org/10.1016/j.scs.2020.102231` doi: 10.1016/j.scs.2020.102231

Mozilla Foundation. (2023a). *Serialization.* `https://developer.mozilla.org/en-US/docs/Glossary/Serialization`.

Mozilla Foundation. (2023b). *State machine.* Online. Descargado de `https://developer`

.mozilla.org/en-US/docs/Glossary/State_machine

Object Management Group. (2005). *What is uml?* Descargado de https://www.uml.org/what-is-uml.htm

Object Management Group. (2015). *What is sysml?* Online. Descargado de https://www.omgsysml.org/what-is-sysml.htm

Ouareth, S., Boulehouache, S., y Mazouzi, S. (2018). A component-based mape-k control loop model for self-adaptation. En *2018 3rd international conference on pattern analysis and intelligent systems (pais)* (p. 1-7). doi: 10.1109/PAIS.2018.8598529

Patouni, E., y Alonistioti, N. (2006). A framework for the deployment of self-managing and self-configuring components in autonomic environments. En *2006 international symposium on a world of wireless, mobile and multimedia networks(wowmom'06)* (p. 5 pp.-484). doi: 10.1109/WOWMOM.2006.11

Rajan, M., Balamuralidhar, P., Chethan, K., y Swarnahpriyaah, M. (2011). A self-reconfigurable sensor network management system for internet of things paradigm [Conference paper]. Descargado de https://www.scopus.com/inward/record.uri?eid=2-s2.0-84858147900&doi=10.1109%2fICDECOM.2011.5738550&partnerID=40&md5=ed1f7dccedf65e520a328eb3a19b581e (Cited by: 17) doi: 10.1109/ICDECOM.2011.5738550

Red Hat Foundation. (2023). *What is yaml?* Descargado 2023-5-27, de https://www.redhat.com/en/topics/automation/what-is-yaml

Rutanen, K. (2018). Minimal characterization of o-notation in algorithm analysis. *Theoretical computer science*, *713*, 31–41.

Salehie, M., y Tahvildari, L. (2005, 01). Autonomic computing: emerging trends and open problems. *ACM SIGSOFT Software Engineering Notes*, *30*, 1-7.

Schiller, J., y Voisard, A. (2004). *Location-Based services*. Oxford, England: Morgan Kaufmann.

Sebesta, R. W. (2012). *Concepts of programming languages* (12.ª ed.). Pearson Education.

Descargado de https://books.google.com.co/books?id=vRQvAAAAQBAJ

Shvets, A. (2019). *Dive into design patterns* (R. S. Andrew Wetmore, Ed.). Refactoring Guru.

Sipser, M. (2012). *Introduction to the theory of computation* (3.ª ed.). Belmont, CA: Wadsworth Publishing.

Song, Y., Li, Z., y Wu, J. (2023). *Mera: Memory reduction and acceleration for quantum circuit simulation via redundancy exploration.* Descargado de https://arxiv.org/abs/2411.15332

Wang, Z., Sun, C.-a., y Aiello, M. (2021). Lightweight and context-aware modeling of microservice-based internet of things. En *2021 ieee international conference on web services (icws)* (p. 282-292). doi: 10.1109/ICWS53863.2021.00046

Weiss, G., Zeller, M., y Eilers, D. (2011). *Towards automotive embedded systems with self-x properties.* Fraunhofer-Gesellschaft. Descargado de https://publica.fraunhofer.de/handle/publica/224379 doi: 10.24406/PUBLICA-FHG-224379

Wu, X.-C., Di, S., Cappello, F., Finkel, H., Alexeev, Y., y Chong, F. T. (2018). Amplitude-aware lossy compression for quantum circuit simulation. En *Proceedings of the 4th international workshop on data reduction for big scientific data (drbsd-4) at sc18.* IEEE.

Wu, X.-C., Di, S., Dasgupta, E. M., Cappello, F., Finkel, H., Alexeev, Y., y Chong, F. T. (2019). Full-state quantum circuit simulation by using data compression. En *Proceedings of the international conference for high performance computing, networking, storage and analysis (sc'19).* IEEE/ACM. doi: 10.1145/3295500.3356155

Zhang, B., Fang, B., Ye, F., Gu, Y., Tallent, N., Tan, G., y Tao, D. (2024). *Overcoming memory constraints in quantum circuit simulation with a high-fidelity compression framework.* Descargado de https://arxiv.org/abs/2410.14088

Apéndices

1 Directivas declaradas la aplicación Malaga

```
application:
  name: Malaga
  description: "Test Scenario A"

locations:
  malaga:
    name: "Sede Malaga"
    locations:
      north-sector:
        name: "Sector Norte"
        data-requirements:
          temperature:
            output: number
            required: true
            count: 1
            timeout: 30
          wind-speed:
            output: number
            required: true
            count: 3
            timeout: 60
          particulate-matter:
            output: number
            required: false
            count: 2
            timeout: 120
      south-sector:
        name: "Sector Sur"
        data-requirements:
          wind-speed:
            output: number
            required: true
            count: 3
            timeout: 60
          particulate-matter:
            output: number
            required: false
            count: 2
            timeout: 120
          temperature:
            output: number
            required: true
            count: 1
            timeout: 30
```

2 Directivas de la aplicación Malaga


```
{
    "north-sector": {
      "addition": {
        "image": "mrdahaniel/mocker:latest",
        "network_name": "smart_campus_production_smart_uis",
        "env_vars": {
          "RUST_LOG": "mocker",
          "ip": "mqtt_broker"
        },
        "args": [
          "key:random(5,35)"
        ]
      },
      "reconfig": {
        "uuid": null,
        "network": "smart_campus_production_smart_uis",
        "query_type": {
          "Http": {
            "endpoint": "/get/device",
            "port": 8000
          }
        },
        "reconfig": {
          "Http": {
            "endpoint": "/update/interval",
            "port": 8000,
            "method": "PUT",
            "payload": {
              "nanos": 0,
              "secs": 25
            }
          }
        }
      },
      "restart": {
        "uuid": null,
        "query_type": {
          "Http": {
            "endpoint": "/get/device",
            "port": 8000
          }
        }
      }
    },
    "south-sector": {
      "addition": {
        "image": "mrdahaniel/mocker:latest",
        "network_name": "smart_campus_production_smart_uis",
        "env_vars": {
          "ip": "mqtt_broker",
          "RUST_LOG": "mocker"
        },
        "args": [
          "key:random(5,35)"
        ]
      },
      "reconfig": {
        "uuid": null,
        "network": "smart_campus_production_smart_uis",
        "query_type": {
          "Http": {
```

```
            "endpoint": "/get/device",
            "port": 8000
          }
        },
        "reconfig": {
          "Http": {
            "endpoint": "/update/interval",
            "port": 8000,
            "method": "PUT",
            "payload": {
              "nanos": 0,
              "secs": 25
            }
          }
        }
      },
      "restart": {
        "uuid": null,
        "query_type": {
          "Http": {
            "endpoint": "/get/device",
            "port": 8000
          }
        }
      }
    }
  }
```

3 YAML de la aplicación Patio

```yaml
application:
  name: Patio
  description: "Patio App"

locations:
  greenhouse:
    name: "Greenhouse"
    data-requirements:
      temperature:
        output: number
        required: true
        count: 1
        timeout: 30
      co2:
        output: number
        required: true
        count: 1
        timeout: 120
      humidity:
        output: number
        required: true
        count: 1
        timeout: 120
```

4 YAML la aplicación House

```yaml
application:
  name: House
  description: "House App"

locations:
  first-floor:
    name: "First Floor"
    data-requirements:
      temperature:
        output: number
        required: true
        count: 1
        timeout: 30
  basement:
    name: "Basement"
    data-requirements:
      co:
        output: number
        required: false
        count: 1
        timeout: 30
```

5 Directivas de la aplicación Patio


```
{
    "greenhouse": {
      "addition": {
        "image": "mrdahaniel/mocker:latest",
        "network_name": "smart_campus_production_smart_uis",
        "env_vars": {
          "RUST_LOG": "mocker",
          "ip": "mqtt_broker"
        },
        "args": [
          "key:random(5,35)"
        ]
      },
      "reconfig": {
        "uuid": null,
        "network": "smart_campus_production_smart_uis",
        "query_type": {
          "Http": {
            "endpoint": "/get/device",
            "port": 8000
          }
        },
        "reconfig": {
          "Http": {
            "endpoint": "/update/interval",
            "port": 8000,
            "method": "PUT",
            "payload": {
              "nanos": 0,
              "secs": 25
            }
          }
        }
      },
      "restart": {
        "uuid": null,
        "query_type": {
          "Http": {
            "endpoint": "/get/device",
            "port": 8000
          }
        }
      }
    }
  }
```

6 Directivas de la aplicación House


```
{
    "basement": {
      "addition": {
        "image": "mrdahaniel/mocker:latest",
        "network_name": "smart_campus_production_smart_uis",
        "env_vars": {
          "RUST_LOG": "mocker",
          "ip": "mqtt_broker"
        },
        "args": [
          "key:random(5,35)"
        ]
      },
      "reconfig": {
        "uuid": null,
        "network": "smart_campus_production_smart_uis",
        "query_type": {
          "Http": {
            "endpoint": "/get/device",
            "port": 8000
          }
        },
        "reconfig": {
          "Http": {
            "endpoint": "/update/interval",
            "port": 8000,
            "method": "PUT",
            "payload": {
              "nanos": 0,
              "secs": 25
            }
          }
        }
      },
      "restart": {
        "uuid": null,
        "query_type": {
          "Http": {
            "endpoint": "/get/device",
            "port": 8000
          }
        }
      }
    },
    "first-floor": {
      "addition": {
        "image": "mrdahaniel/mocker:latest",
        "network_name": "smart_campus_production_smart_uis",
        "env_vars": {
          "ip": "mqtt_broker",
          "RUST_LOG": "mocker"
        },
        "args": [
          "key:random(5,35)"
        ]
      },
      "reconfig": {
        "uuid": null,
        "network": "smart_campus_production_smart_uis",
        "query_type": {
          "Http": {
```

```
            "endpoint": "/get/device",
            "port": 8000
          }
        },
        "reconfig": {
          "Http": {
            "endpoint": "/update/interval",
            "port": 8000,
            "method": "PUT",
            "payload": {
              "nanos": 0,
              "secs": 25
            }
          }
        }
      },
      "restart": {
        "uuid": null,
        "query_type": {
          "Http": {
            "endpoint": "/get/device",
            "port": 8000
          }
        }
      }
    }
  }
```