Mecanismos de adaptación autonómica de arquitectura software para la plataforma Smart Campus UIS

Daniel David Delgado Cervantes

Trabajo de grado para optar el título de Ingeniero de Sistemas

Director

PhD. Gabriel Rodrigo Pedraza Ferreira

Escuela De Ingeniería De Sistemas e Informática

Codirector

MSc. Henry Andrés Jiménez Herrera

Escuela De Ingeniería De Sistemas e Informática

Universidad Industrial de Santander

Facultad De Ingenierías Fisicomecánicas

Escuela De Ingeniería De Sistemas E Informática

18 de enero de 2024

Dedicatoria

Agradecimientos

Al bicho siuuu

Contenido

Introducción **1**

1. Planteamiento del problema **2**

2. Objetivos **3**

2.1. Objetivo General 3

2.2. Objetivos Específicos 3

3. Estado del Arte **4**

3.1. Computación Autonómica 4

3.1.1. MAPE-K 5

3.1.2. Mecanismos de Descripción 7

3.1.3. Mecanismos de Adaptación 8

3.2. Sistemas IoT Autonómicos 9

3.2.1. Toma de Decisiones en Sistemas IoT Autonómicos 10

4. Marco Teórico **11**

4.1. Internet of Things 11

4.1.1. Sistemas Embebidos 11

4.1.2. Location-based Services 11

4.1.3. Smart Campus 12

4.2. Notación 12

4.2.1. Gramática 13

4.2.2. Sintaxis 13

4.2.3. Domain-Specific Languages 13

4.3. Serialización de Datos 13

4.3.1. Métodos de Serialización de Datos 14

5. Metodología de la Investigación **15**

5.1. Ambientación Conceptual y Tecnológica 15

5.2. Definición de la notación de la arquitectura 16

5.3. Mecanismos De Comparación 16

5.4. Mecanismos De Adaptación 17

5.5. Validación De Resultados 18

6. Lenguajes de Descripción de Arquitectura **19**

6.1. La necesidad de una arquitectura de referencia 19

6.2. Criterios de selección 19

6.3. Búsqueda de Alternativas 20

6.4. Un nuevo modelo para Smart Campus 23

6.5. Sintaxis de la notación 27

6.6. Implementando Una Validación 30

7. Estado Actual Frente al de Referencia **32**

7.1. Conociendo el estado actual 32

7.2. Centralizando los Datos 35

7.3. Implementado un Event-Handler 36

7.4. Comparando Arquitecturas 39

8. Adaptando la Arquitectura **42**

8.1. Identificando Los Problemas, Estableciendo Acciones 42

8.2. De Acciones, a Instrucciones 44

8.3. De Instrucciones a Acciones 46

9. Desarrollo Experimental **50**

9.1. Escenario Experimental A 50

9.2. Escenario Experimental B 50

10. Análisis de Resultados **50**

10.1. Resultados De Escenario Experimental A 50

10.2. Resultados De Escenario Experimental B 50

11. Conclusiones y Trabajo Futuro **50**

Referencias **A**

A. Apéndices **G**

A.1. Ejemplo de un YAML de una aplicación válida . . . . . G

Lista de figuras

1. El ciclo auto-adaptativo MAPE-K (Gorla, Pezzè, Wuttke, Mariani, y Pastore, 2010) 6
2. Metodología del proyecto planteada 15
3. Versión 2 del modelo concepto planteado por H. A. Jiménez Herrera 24
4. Mensaje JSON enviado por los dispositivos en Smart Campus UIS 25
5. Versión 1 del metamodelo planteado para SCampusADL 26
6. Diagrama de rail de la sintaxis definida para la notación de SmartCampusADL 27
7. Diagrama de rail de la sintaxis de declaración de la aplicación 28
8. Diagrama de rail de la sintaxis definida para la notación del contexto geográfico de la aplicación 28
9. Diagrama de rail de la sintaxis definida para la notación de los componentes de la aplicación 29
10. YAML de una posible aplicación de monitorio 30
11. Diagrama de flujo del proceso realizado por el módulo *Lexical* 30
12. Diagrama de flujo para validación y procesamiento de las locaciones 31
13. Diagrama de flujo del procesamiento de los requerimientos de datos 32
14. Arquitectura del prototipo de Smart Campus definido por 34
15. Arquitectura actual del proyecto con el agregador 35
16. Diagrama de flujo del proceso realizado por Lexical actualizado 36
17. Primera propuesta del proceso a realizar por *Looker* 37
18. Proceso realizado por el observador durante el consumo de los mensajes enviados por los dispositivos 38
19. Proceso reloj realizado por el observador para la actualización del estado de la aplicación 40
20. Arquitectura actual del proyecto 41

21. Proceso para la identificación de problemas de los requerimientos de datos . 43
22. UML de la estructura de directivas y ordenes 44
23. UML de la estructura de directivas y ordenes 45
24. Proceso base de DoThing 47
25. Proceso para la ejecución de ordenes de adición 47
26. Proceso para la ejecución de ordenes de reinicio 48
27. Proceso para la ejecución de ordenes de reinicio 49
28. Proceso para la ejecución de ordenes de reinicio 49

Lista de tablas

1. Criterios usados para la determinación de la notación a utilizar 20
2. Evaluación de las alternativas en función de los criterios establecidos 23

Glosario

IoT: Internet of Things

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

Introducción

La computación autonómica, concebida inicialmente por IBM en el año 2001, se refiere al uso de sistemas auto-gestionados con la capacidad de operar y adaptarse, o en lo posible, sin la intervención de un ser humano. En este sentido, este acercamiento tiene como objetivo la creación de sistemas computacionales capaces de reconfigurarse en respuesta a cambios en las condiciones del entorno de ejecución al igual que los objetivos del negocio (Horn, 2001).

Esta autonomía es adquirida con el uso de ciclos de control, en el caso de la computación autonómica, de los ciclos más populares es el ciclo MAPE-K, sigla de *Monitor Analyze Plan Execute - Knowledge* (Arcaini, Riccobene, y Scandurra, 2015). Estos le dan la capacidad al sistema de monitorear tanto su estado actual como el entorno en el que este se encuentra, analizar la información recolectada para luego planear y ejecutar los cambios requeridos sobre el sistema a partir de una base de conocimiento (Rutanen, 2018).

Una de las áreas que pueden verse especialmente beneficiadas de la computación autonómica es la del internet de las cosas (IoT). Algunos de estos aspectos están relacionados con la heterogeneidad de estos dispositivos, en cuanto a marcas, protocolos y características; la dinamicidad, en cuanto al movimiento que estos presentan entre entornos de ejecución o incluso una desconexión; al igual que la distribución geográfica de estos lo cual dificulta la intervención directa sobre ellos (Tahir, Mamoon Ashraf, y Dabbagh, 2019).

Esto puede verse el Smart Campus UIS, una plataforma IoT de la Universidad Industrial de Santander, que permite usar dispositivos con el fin de monitorear y recolectar de información en tiempo real con el objetivo de apoyar la toma de decisiones, mejora de servicios, entre otros (H. Jiménez Herrera, Cárcamo, y Pedraza, 2020).

Ahora, esta plataforma ha tenido esfuerzos en el desarrollo de características propias de un sistema autonómico. Uno de estos ha sido la integración de mecanismos para la auto-descripción de la arquitectura desplegada en un momento dado (H. A. Jiménez Herrera, 2022). En el contexto de la computación autonómica, esta capacidad hace parte del monitoreo

dentro del ciclo de control.

Partiendo de lo anterior, y con la intención de dar continuidad con los esfuerzos de desarrollo realizados en Smart Campus UIS, se plantea como caso de estudio, a partir de las capacidades de auto-descripción que la plataforma, se pretende el proveer a esta la capacidad de auto-configuración a partir de un conjunto de mecanismos de adaptación que permitan, desde la definición de un objetivo a lograr, la alteración de la arquitectura desplegada.

1 Planteamiento del problema

La complejidad de los sistemas computacionales tiene origen en diversos factores. El aumento de la cantidad de dispositivos que los componen; la heterogeneidad debido a diferentes marcas, protocolos y características; e incluso las cambiantes condiciones de sus entornos de ejecución; dificulta la administración de los sistemas computacionales (Salehie y Tahvildari, 2005).

Una de las posibles maneras de dar solución a esta problemática, en cuanto al manejo de sistemas, está en el área de la computación autonómica. Este acercamiento basado en conceptos biológicos busca solventar los problemas de complejidad, heterogeneidad e incertidumbre (Salehie y Tahvildari, 2005) a partir de la abstracción de las metas de los administradores y delegación del manejo del sistema a sí mismo (Lalanda, Diaconescu, y McCann, 2014).

Considerando lo anterior, una de las aplicaciones de la computación autonómica, dentro del IoT, son los Smart Campus; una variación de las Smart Cities en las cuales se busca la recolección de información y monitoreo en tiempo real con el fin de apoyar la toma de decisiones, mejora de servicios, entre otros (Min-Allah y Alrashed, 2020). Es en este tipo de aplicaciones donde los problemas, en especial la heterogeneidad, dinamicidad, al igual que la distribución geográfica; donde la reducción de la dependencia de intervención humana, facilitaría el manejo de estos.

De lo anterior, surge la pregunta del cómo realizar dichas adaptaciones a la arquitectura, o estado del sistema. Qué clase de mecanismos y requerimientos han de ser satisfechos con

el fin de considerar, a un sistema computacional, con la capacidad de auto-adaptarse dados unos objetivos o metas establecidas por los administradores del sistema.

Dentro del marco del presente proyecto, se tiene Smart Campus UIS, una plataforma de IoT de la Universidad Industrial de Santander, en la cual se han realizado implementaciones parciales de una arquitectura autonómica con capacidad de auto-describirse (H. A. Jiménez Herrera, 2022), una de las características principales de un sistema autonómico (Horn, 2001).

Dicho esto, se busca explorar algunas de las maneras en las que se realizan adaptaciones con características autonómicas en sistemas computacionales. De esto, se buscaría probar un conjunto de estos mecanismos de modificación del estado, tomando como caso de estudio la implementación de estos en la plataforma de Smart Campus UIS.

2 Objetivos

2.1 Objetivo General

- Diseñar un conjunto de mecanismos autonómicos para permitir la adaptación de la Arquitectura Software IoT respecto a un modelo objetivo en la plataforma Smart Campus UIS

2.2 Objetivos Específicos

- Proponer una notación (lenguaje) para describir una arquitectura objetivo de un sistema software IoT.
- Diseñar un mecanismo para determinar las diferencias existentes entre una arquitectura actual en ejecución y una arquitectura objetivo especificada.
- Diseñar un conjunto de mecanismos de adaptación que permitan disminuir las diferencias entre la arquitectura actual y la arquitectura objetivo.

- Evaluar la implementación realizada a partir de un conjunto de pruebas con el fin de establecer la efectividad de los mecanismos usados.

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 y cols., 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.

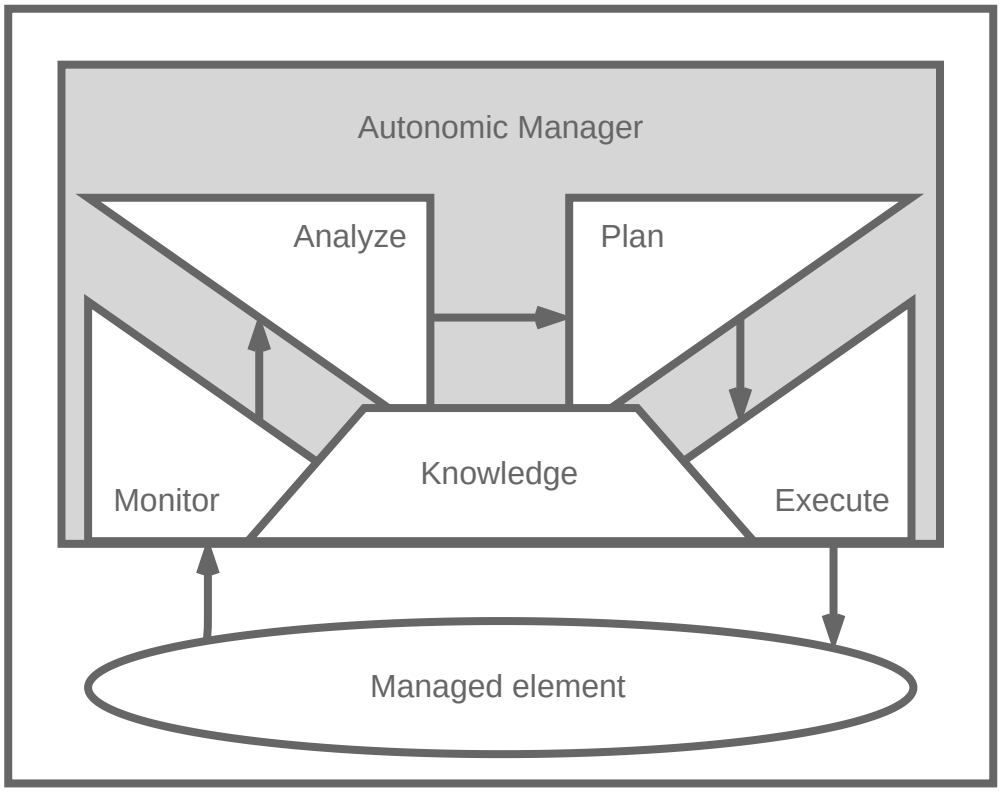
8. Abstracción: ocultar la complejidad a los administradores del sistema con objetivos de alto nivel de abstracción.

En el caso de que un sistema tenga una implementación parcial de estas características, este podría considerarse autonómico. En este sentido 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

IBM, en cuanto a la implementación de las características, propone una modelo de ciclo auto-adaptativo, denominado MAPE-K (Krikava, 2013). Este acercamiento, compuesto de cinco fases, es uno de ciclos de control más usado en implementaciones de sistemas auto-adaptativos y computación autonómica (Arcaini y cols., 2015). En la figura 1, se presentan las fases que *manejador* debe desarrollar para así administrar cada uno de los elementos del sistema computacional basado 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 reportar la información adquirida sobre el estado del elemento a manejar.
- Analizar (A): La fase de análisis se encarga del interpretar el entorno en el cual se encuentra, el predecir posibles situaciones comunes y diagnosticar el estado del sistema.
- Planear (P): Durante la planificación se determina 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 del sistema en un momento dado (Weiss, Zeller, y Eilers, 2011).

Existen varias maneras de realizar implementaciones de mecanismos de auto-descripción y la utilidad de cada uno de estos varía dependiendo en el tipo de sistema de software que se esté usando. Para el marco del proyecto, son relevantes aquellas 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 modi-

ficaciones 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 en sistema operativo, 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 centraron en 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 componentes de la arquitectura. Siendo así, es posible que la mo-

dificación de estos sea de interés con el fin de alterar el comportamiento de un sistema al igual que soportar la heterogeneidad del sistema. Esto puede verse en el trabajo desarrollado por Liu, Parashar, y Hariri (2004) en donde definen 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 componen la arquitectura; de esta manera, modificando su comportamiento. Ejemplos de esto puede 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 a nivel individual al igual que grupal.

  Este acercamiento a la mutación de la arquitectura también puede verse en el despliegue de componentes como respuesta a cambios en los objetivos de negocio de las aplicaciones al igual que como respuesta a cambios 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. Estos acercamientos entre estas dos ramas de las ciencias de la computación se ha venido presentado con la búsqueda de dar a los sistemas IoT la capacidad de adaptarse en a su ambiente con el fin de realizar optimizaciones de los recursos disponibles con el fin de reducir costos, al igual que la necesidad de interacción humana, con la automatización de la configuración así como los procesos de mantener la disponibilidad de los servicios de estos (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 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 va 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 los cuales 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 y cols. (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 los cuales se usan con 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:

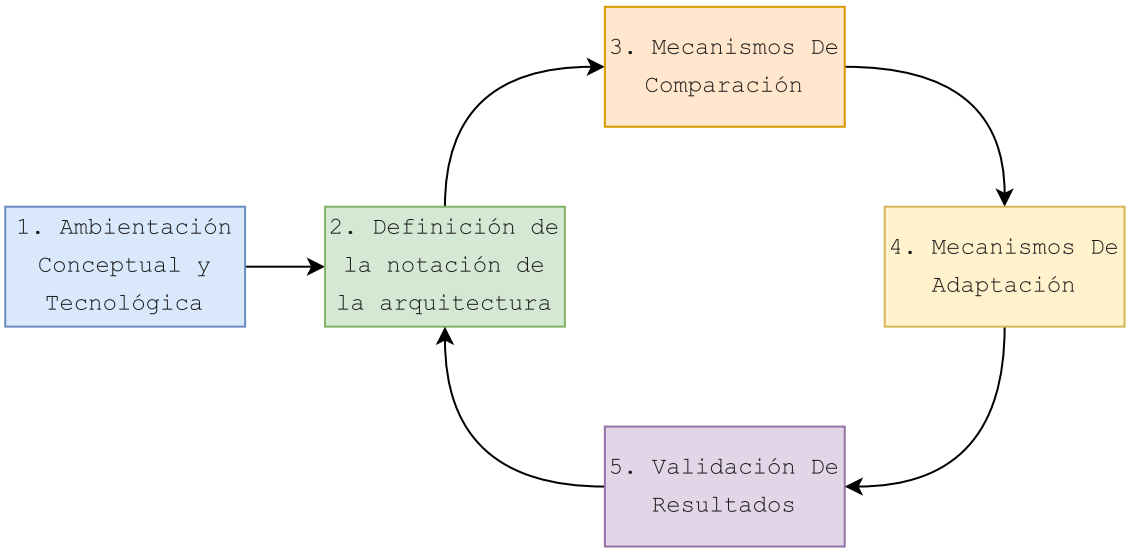
- **Serialización en Formato de Texto**: Esta forma de serialización emplea un formato legible por humanos, generalmente utilizando caracteres y símbolos. Este enfoque es especialmente útil cuando se necesita que los datos sean legibles, y editables, directamente por humanos, lo que los hace adecuados 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 propone un modelo de prototipado iterativo compuesto de 5 fases (Ver fig. 2). De esta manera, se avanzará a medida que se va completando la fase anterior y permitirá a futuro el poder iterar sobre lo que se ha desarrollado anteriormente.

Figura 2: Metodología del proyecto planteada



5.1 Ambientación Conceptual y Tecnológica

La primera fase de la metodología se basa en la investigación de la literatura, al igual que de la industria, necesaria para cubrir las bases tanto conceptuales como técnicas necesarias 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 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 está en la definición del cómo se realiza la declaración de la arquitectura. Partiendo de los criterios de selección establecidos en la fase 1, se espera determinar un lenguaje de notación la cual permita 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 la gramática a usar para la definición de la arquitectura.
3. Determinación 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 buscará poder determinar e implementar cómo se realizará 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 será necesario definir *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 está 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 encargará principalmente de 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 de 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 fijar 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, es necesario establecer el punto desde el cual se realizará la comparación entre los estados del sistema. Esto guiará la búsqueda para determinar el como se realizará la declaración de los estados objetivo de Smart Campus UIS, al igual que los datos que se recolectarán.

Siendo así, y dados los objetivos que buscan cumplir los ecosistemas inteligentes para la toma de decisiones (Anagnostopoulos, 2023), se estableció que la métrica por la cual se describiría, y evaluaría, el estado del sistema sería a partir de los datos que estaban siendo recolectados. Es decir, que el enfoque debía estar orientado a cumplir con las necesidades de los datos establecidas por Smart Campus.

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	Valor
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.	Alto
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.	Alto
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.	Alto
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.	Bajo
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.	Medio
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.	Medio

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 para describir la arquitectura de sistemas IoT. Esta búsqueda 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 estos se filtraron, o descartaron; a partir de los criterios de selección establecidos. Esto se debe a que los LDAs usados en la 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 demasiado genéricos y abarcan hardware, software e incluso personas y procesos (Object Management Group, 2015).

Entonces, de las posibles opciones de notación, se seleccionaron cinco las cuales fueron evaluadas con el fin de determinar si alguna de las opciones hubiera podido ser usada, o si era necesario desarrollar notación propia al 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 estos componentes a nivel de los datos y la dirección de estos. (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[1] en la cual se usan estereotipos de UML con el 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* [2] (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*[3] 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.

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

[2]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)

[3]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	✗	✗	✗	✓	✗

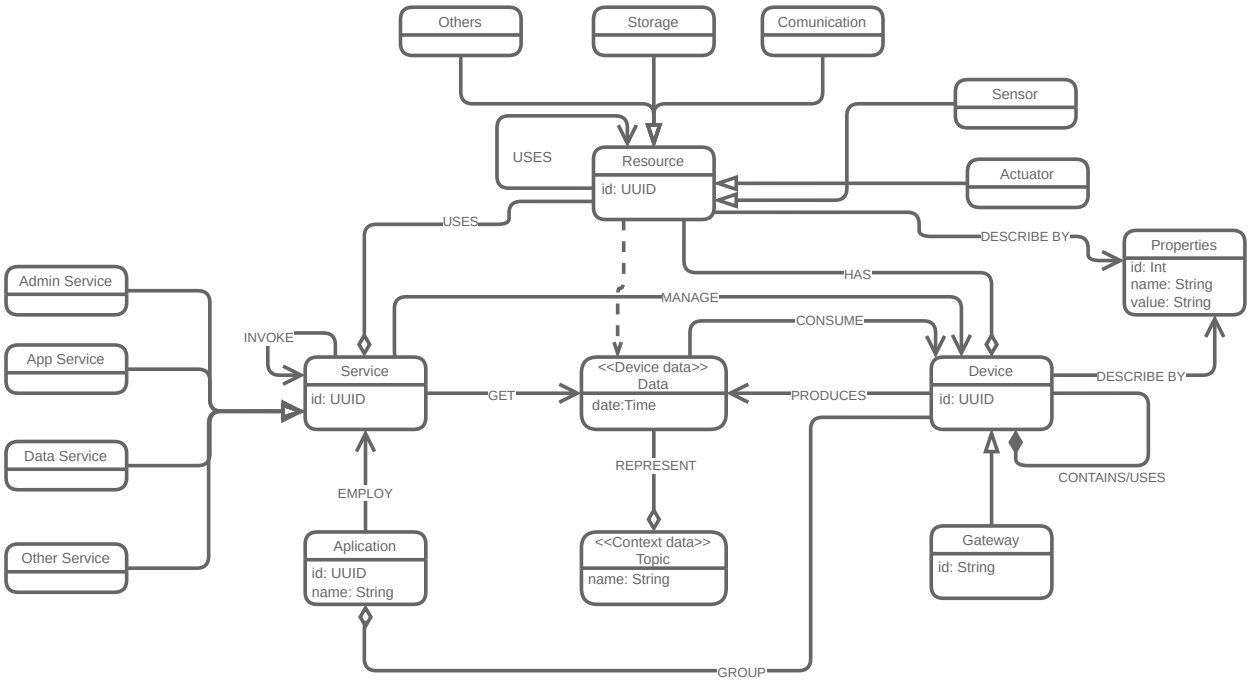
Partiendo de los resultados de la evaluación, se puede apreciar que ninguno de estos LDA cumple con los criterios establecidos para el proyecto. Aunque estos están orientados hacia la descripción y desarrollo de sistemas IoT, no están enfocados hacia su contexto en términos de aplicación más allá de 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 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 que definir un metamodelo el cual permita establecer la manera en la que se verán estas arquitecturas. Para ello, y partiendo de la implementación a realizar, se basó 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.

Lo primero fue el 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 puede 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.

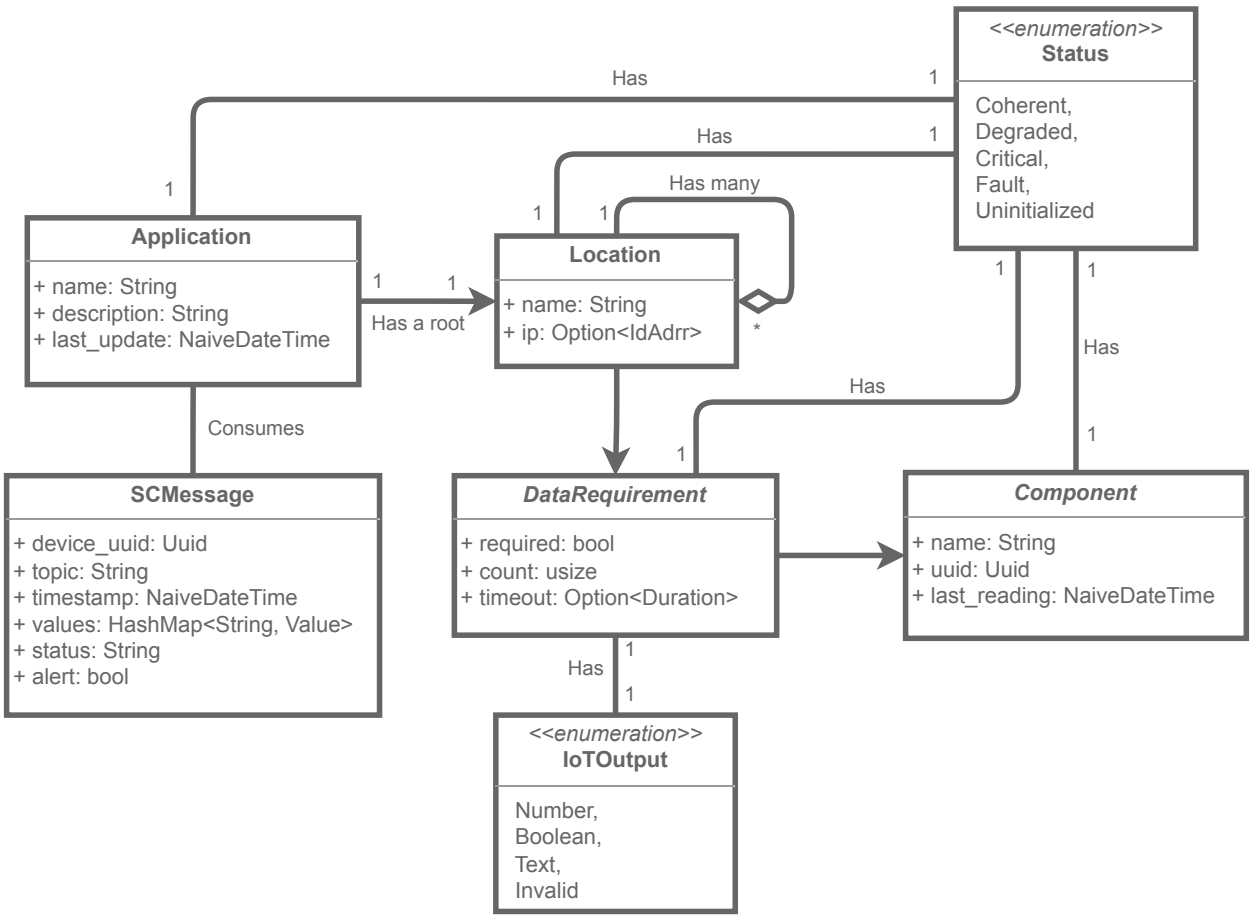
Partiendo de esto, se desarrolló el UML presente en la figura 5. La base de este es la aplicación (***Application***). Este contiene, a partir de una ubicación raíz, todas las demás locaciones (***location***) pertinentes para el correcto funcionamiento de esta.

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.

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.

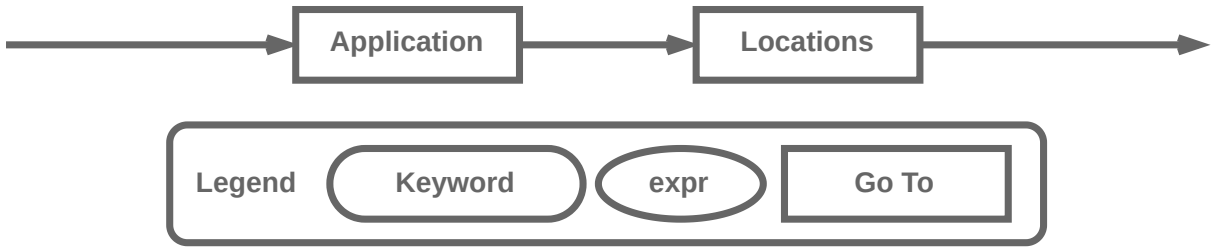
De este modelo, se realizó la implementación de una librería, apodada *StarDuck* [4]. Esta librería contiene todas las estructuras de datos definidas en el modelo, las cuales serán usadas por el resto de los servicios y aplicativos necesarios para la declaración, evaluación y adaptación de las arquitecturas 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.

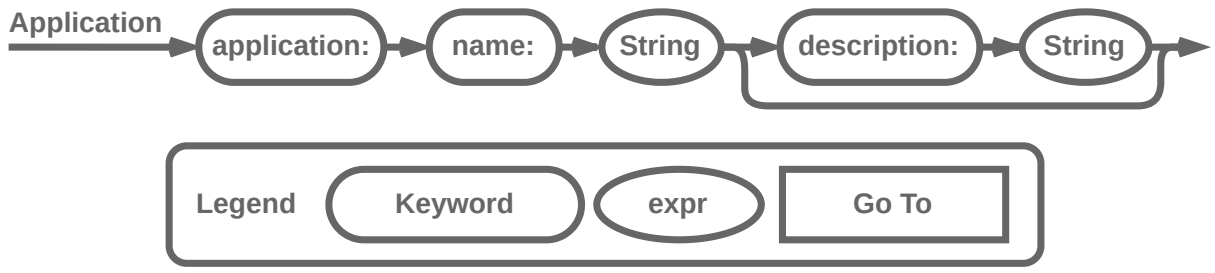
Figura 6: Diagrama de rail de la sintaxis definida para la notación de SmartCampusADL



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.

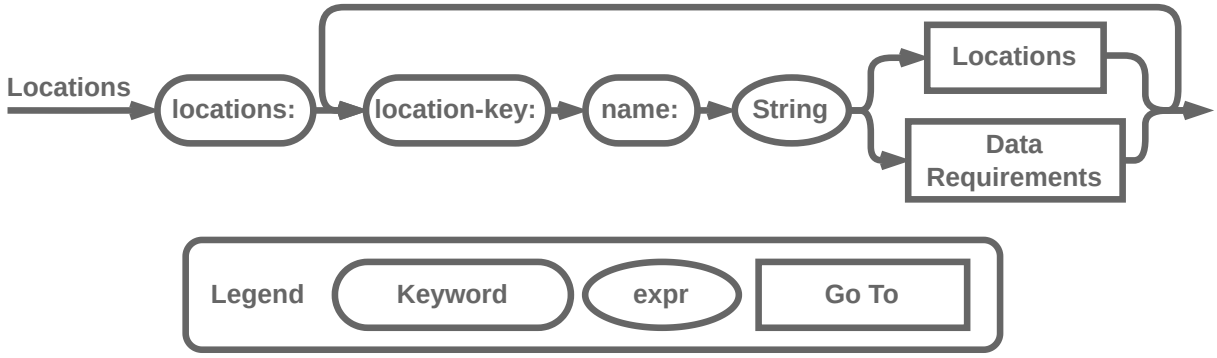
[4]El código fuente de la librería puede encontrarse en `https://github.com/ChipDepot/StarDuck`, y su respectivo paquete en crates.io, en `https://crates.io/crates/starduck`.

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.

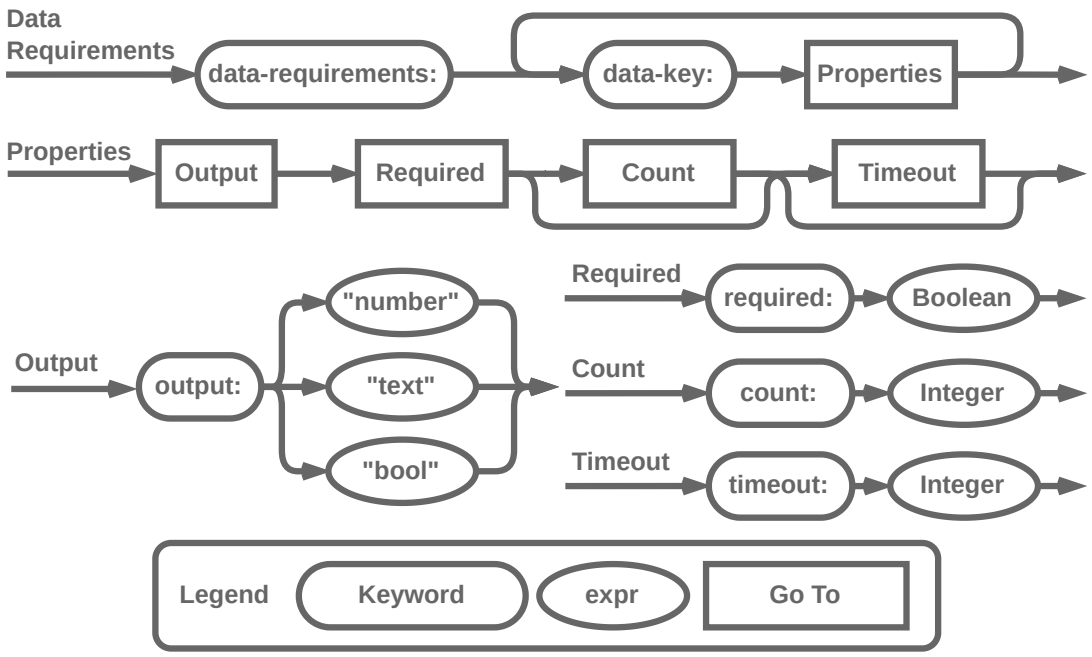
Figura 8: Diagrama de rail de la sintaxis definida para la notación del contexto geográfico de la aplicación



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 [5], 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.

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

Figura 10: YAML de una posible aplicación de monitorio

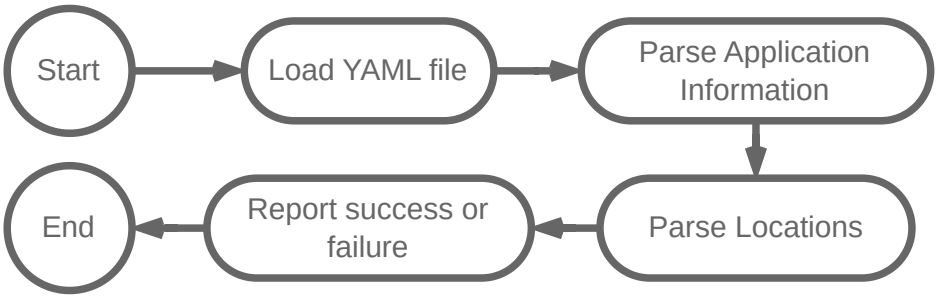
```yaml
application:
  name: Chip
  description: "Test App"

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*

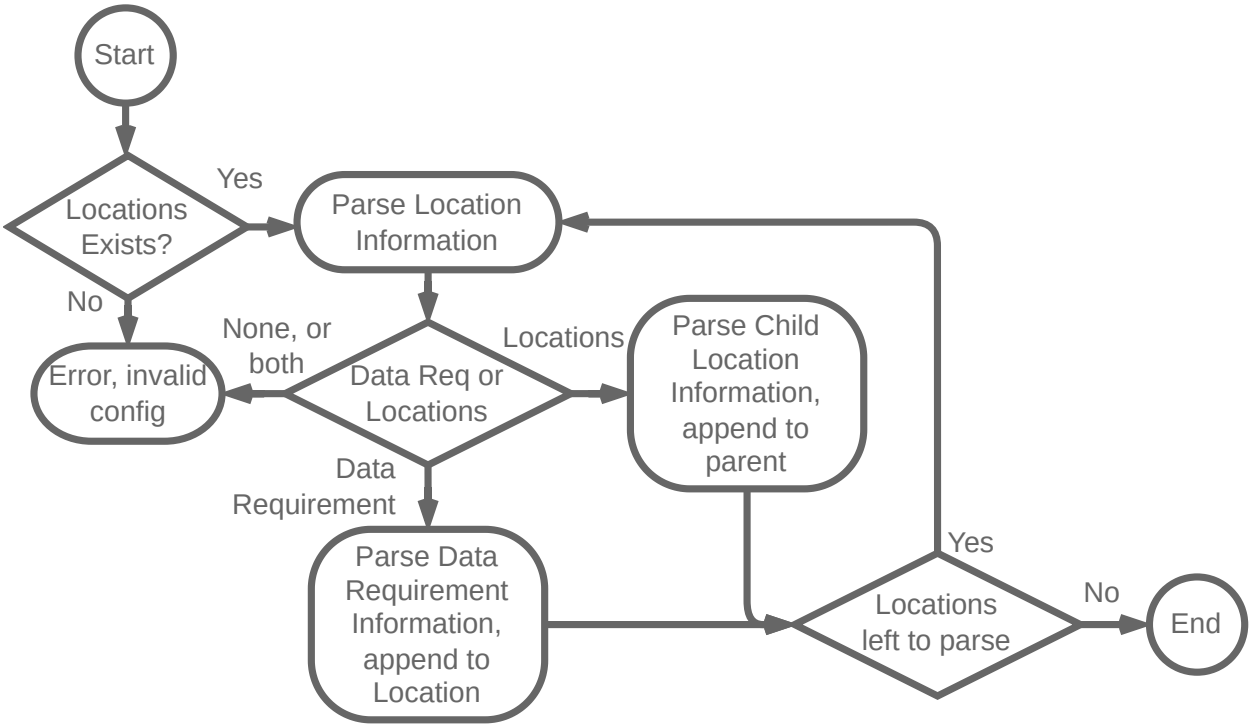


Las 2 primeras partes del proceso son bastante directas. Lo primero es cargar el archivo

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.

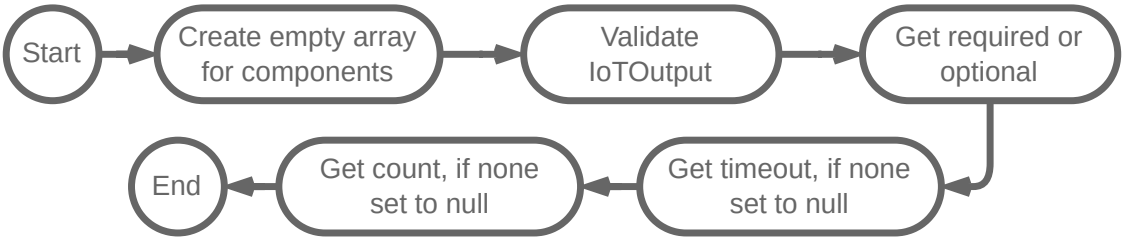
Figura 12: Diagrama de flujo para validación y procesamiento de las locaciones



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.

El desarrollo de este módulos, se realizó en Rust. Escogido debido a su capacidad pa-

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



ra 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.[6]

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 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,

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

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.

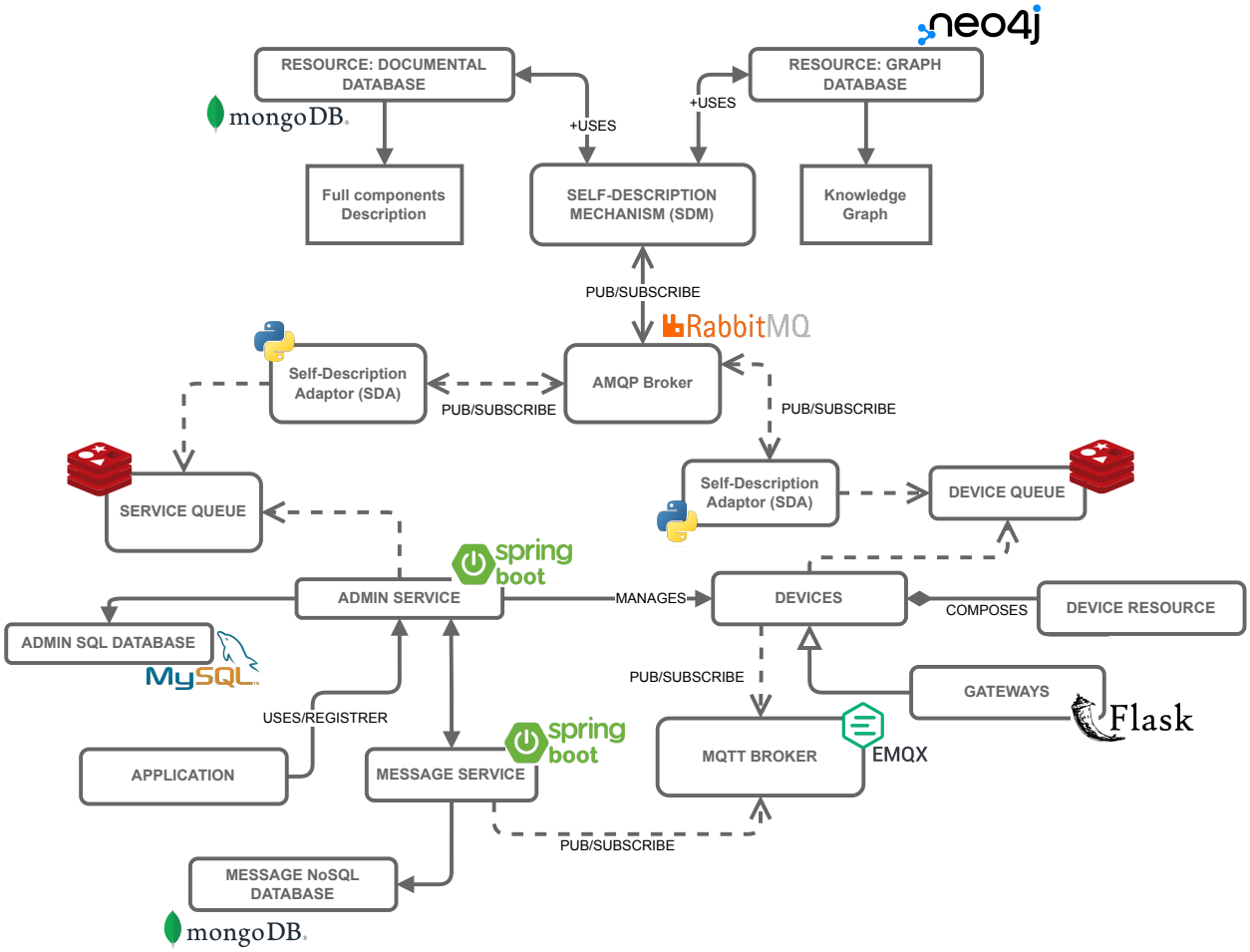
Ahora, cada una de las maneras de acceder a los datos es viable, y podría permitir la implementación correspondiente para determinar el estado del sistema. Sin embargo, de entre las dos opciones, se escogió el procesamiento de los mensajes enviados por los dispositivos, enviados por MQTT.

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 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

Figura 14: Arquitectura del prototipo de Smart Campus definido por H. A. Jiménez Herrera (2022)



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.

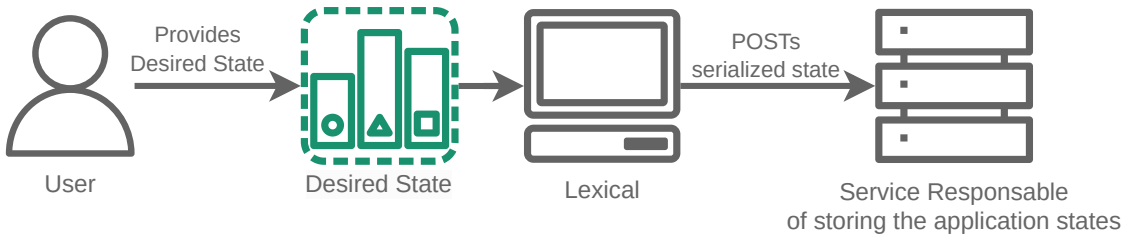
Partiendo de la división de responsabilidades, se estableció el implementar un agregador, que contenga el estado definido por el usuario; desde el cual otros servicios puedan acceder a estos datos.

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.

Figura 15: Arquitectura actual del proyecto con el agregador

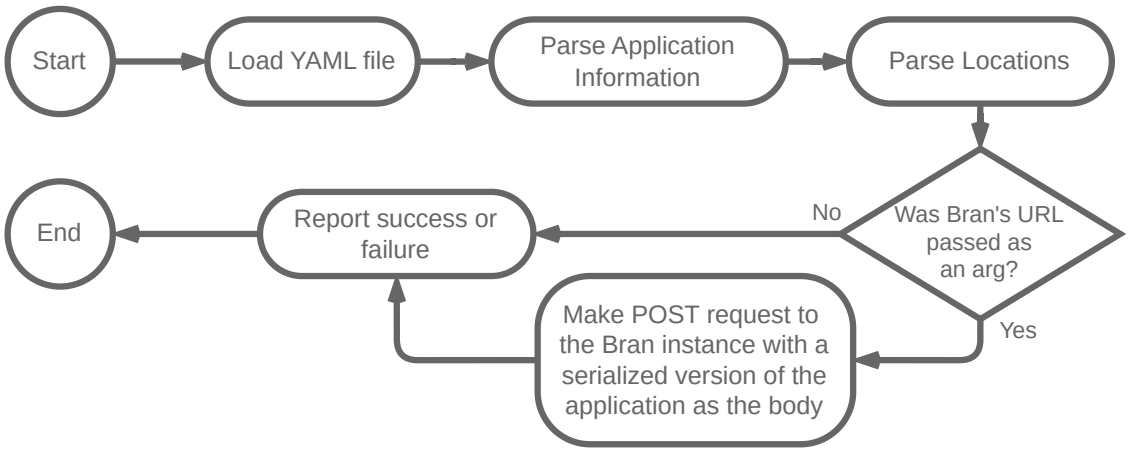


La implementación de este agregador, apodado *Bran*[7], fue relativamente directa. Este tiene dos requerimientos, el primero, es la capacidad de recibir, guardar y actualizar los estados de las aplicaciones; y segundo, el poder enviar los estados almacenados cada vez que estos se soliciten.

Ambas de estas funcionalidades se implementaron usando el framework `Axum`[8]. 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



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

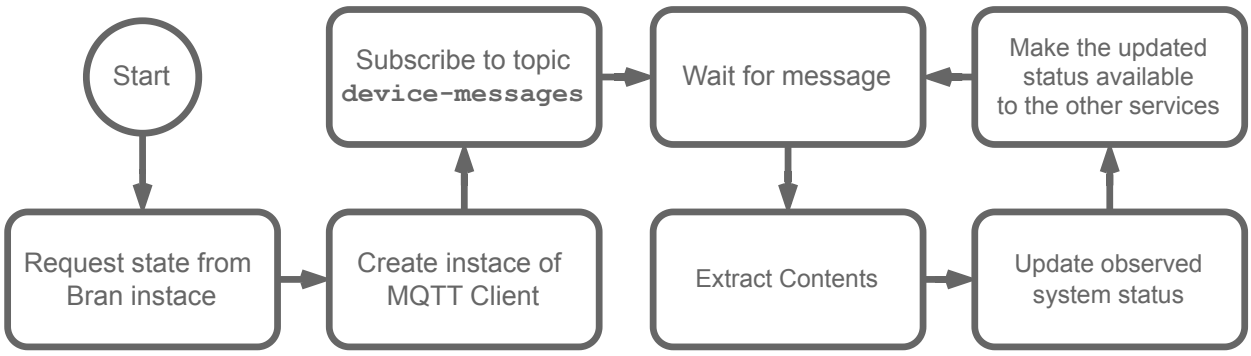
[7]El código fuente de Bran puede encontrarse en el repositorio `https://github.com/ChipDepot/Bran`

[8]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`

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*

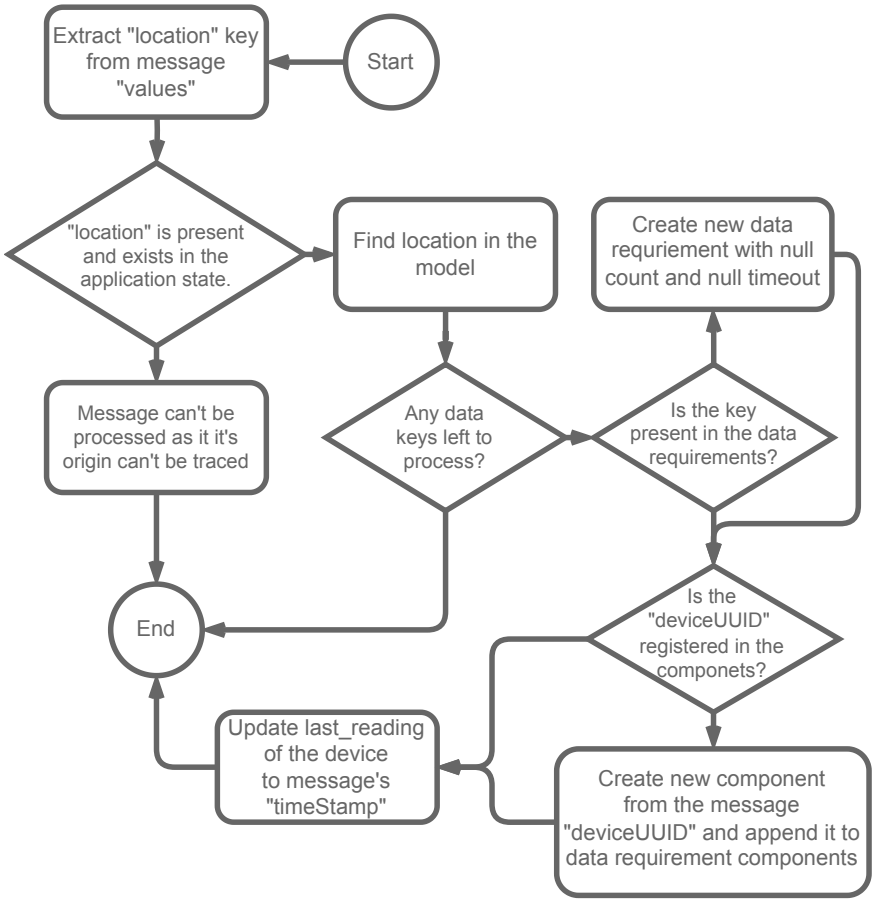


Este primer acercamiento, es bastante directo en cuanto a la implementación a realizar. Tras adquirir la definición del estado objetivo, se inicializa un cliente de MQTT, conectado el broker usado por Smart Campus UIS y subscrito al tópico usado por los dispositivos, `device-messages` (H. A. Jiménez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

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

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



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.

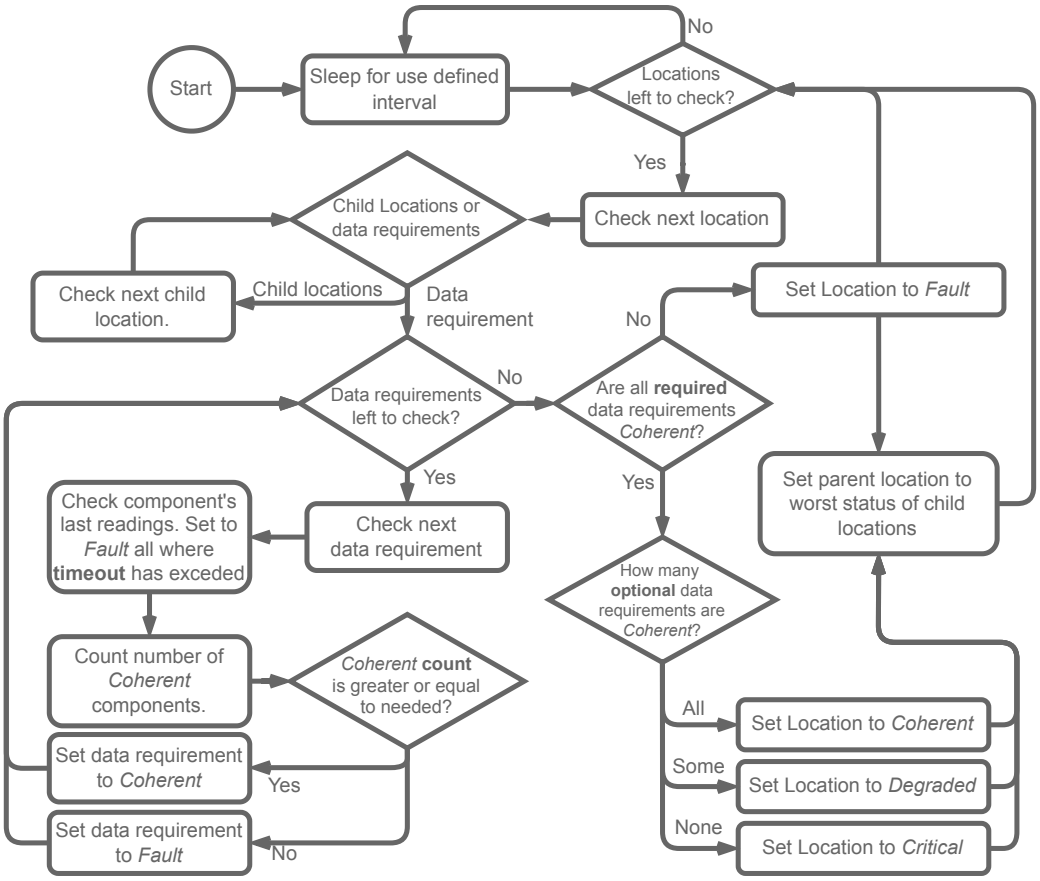
Siendo así, se definió un proceso de comparación consta de evaluar cada componente del sistema en función de las propiedades. A partir de estas, es posible el evaluar el estado de la aplicación. Este proceso es descrito por la figura 19.

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.

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,

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



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.

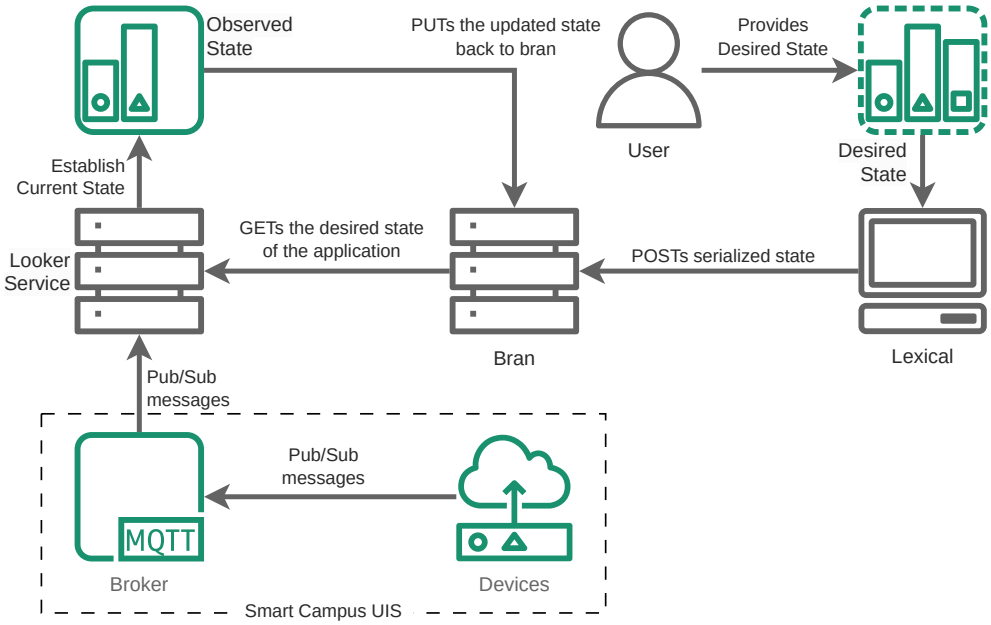
El estado de estas locaciones se determina en dos pasos. En el primer paso, se verifica si alguno de sus requerimientos obligatorios está en estado *Fault*. En ese caso, la locación se considera en estado *Fault*, ya que no puede cumplir con los requisitos mínimos de la aplicación. Para las locaciones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

Figura 20: Arquitectura actual del proyecto



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, 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.

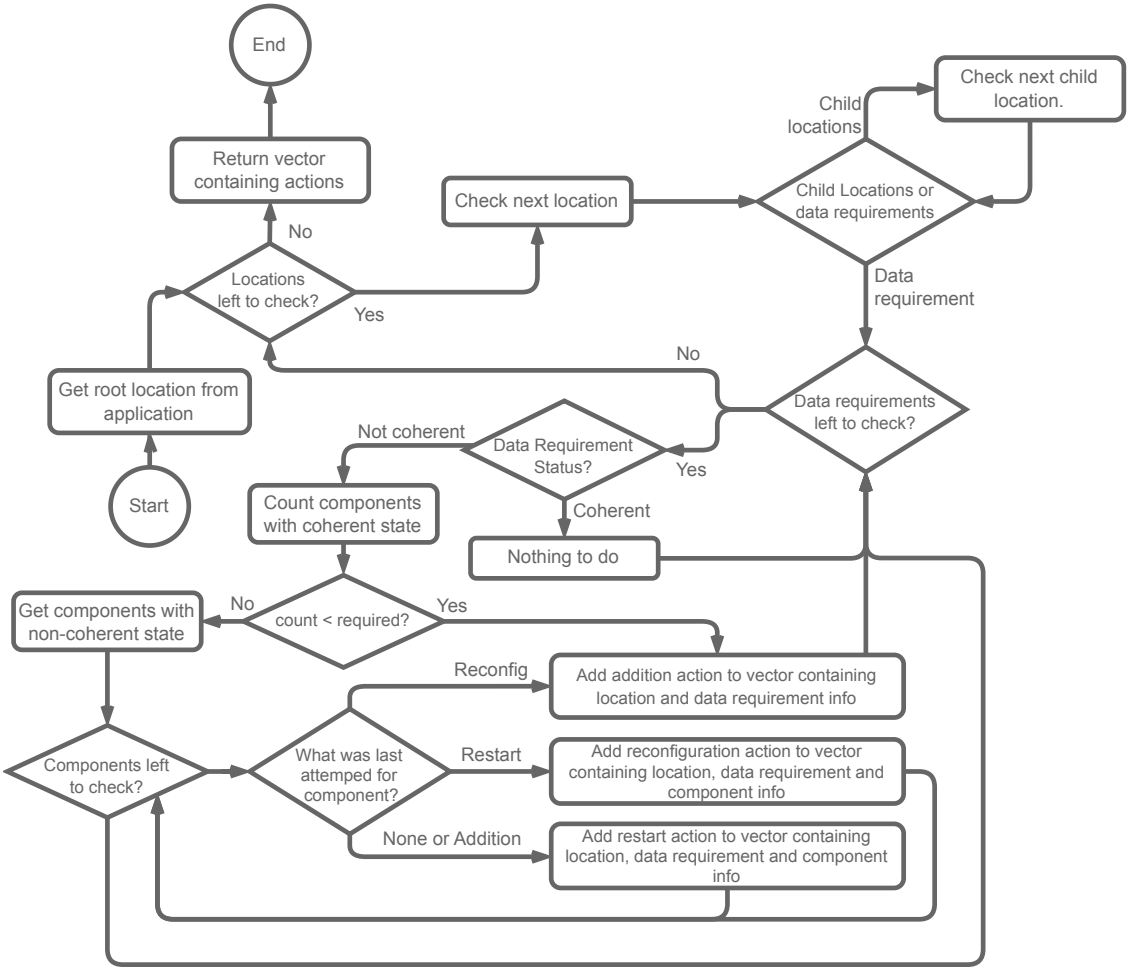
La manera en la de que se identifican los problemas es similar a la vista en la figura 19. Se recorren las locaciones, buscando requerimientos de datos. Una vez en una de estas locaciones, se validan los requerimientos y se definen las acciones.

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.

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

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



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.

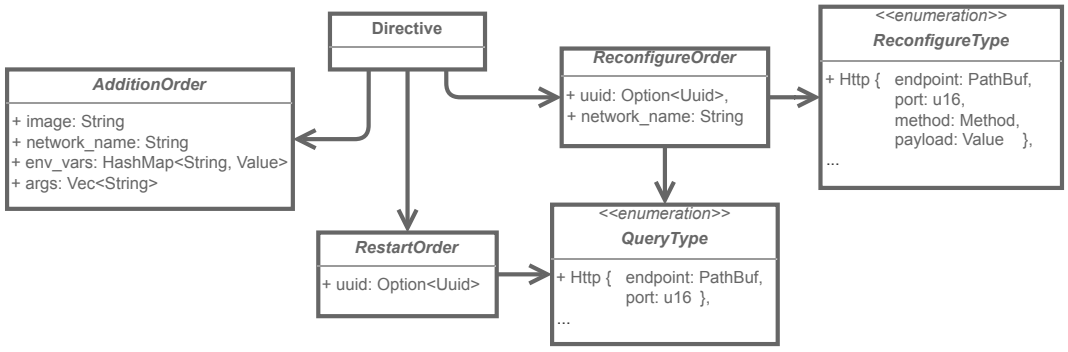
8.2 De Acciones, a Instrucciones

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.

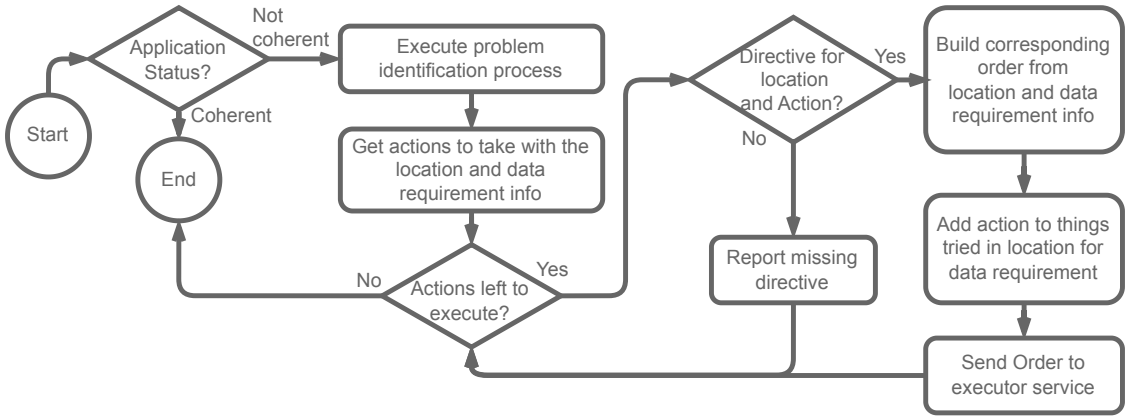
Las órdenes de reinicio y reconfiguración poseen el *UUID* del dispositivo que buscan afectar. Siendo así, al ser necesario tener una manera por la cual identificar qué servicio es qué componente, se implementó el enum `QueryType`. Este, contiene la información de como debe buscarse el contenedor en 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.

Figura 23: UML de la estructura de directivas y ordenes



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 correspondiente, 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*[9].

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.

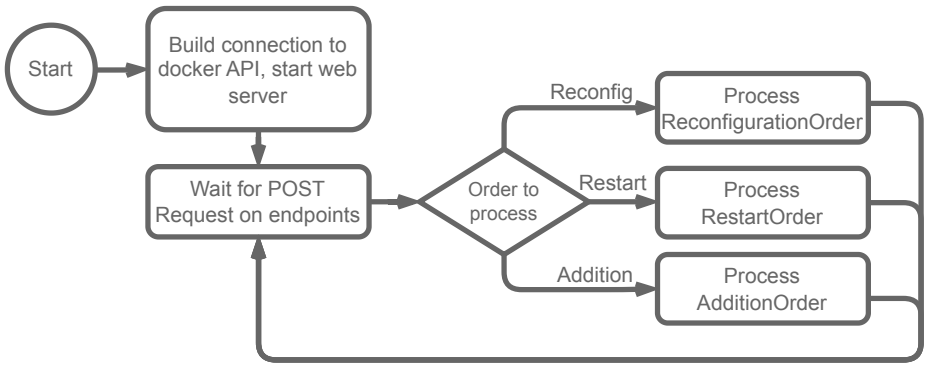
Esto se solucionó usando la Docker Engine API como el canal de comunicación entre el servicio, y el resto de la arquitectura. Esto se debe a que, esta API REST, permite la interacción directa con el daemon de Docker (Docker Inc, 2023), posibilitando la ejecución de todos los comandos relacionados con 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 principal-

[9]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.

mente al procesamiento de las peticiones realizadas, por lo que es relativamente sencillo en cuanto a las acciones que este realiza.

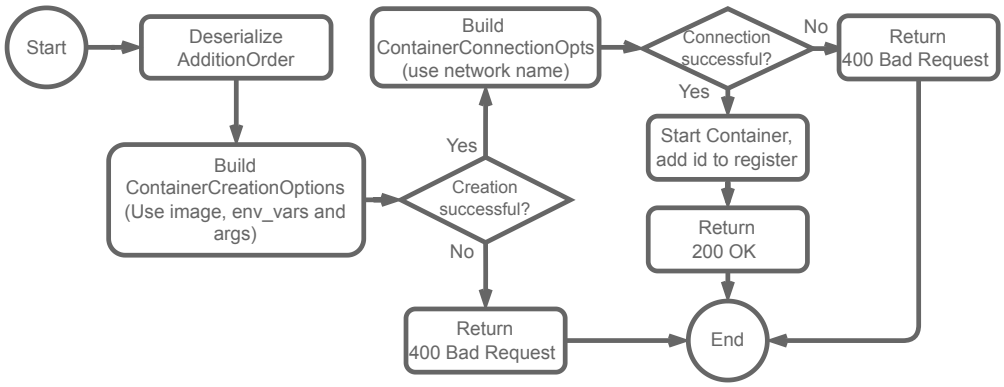
Figura 24: Proceso base de DoThing



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.

Figura 25: Proceso para la ejecución de ordenes de adición

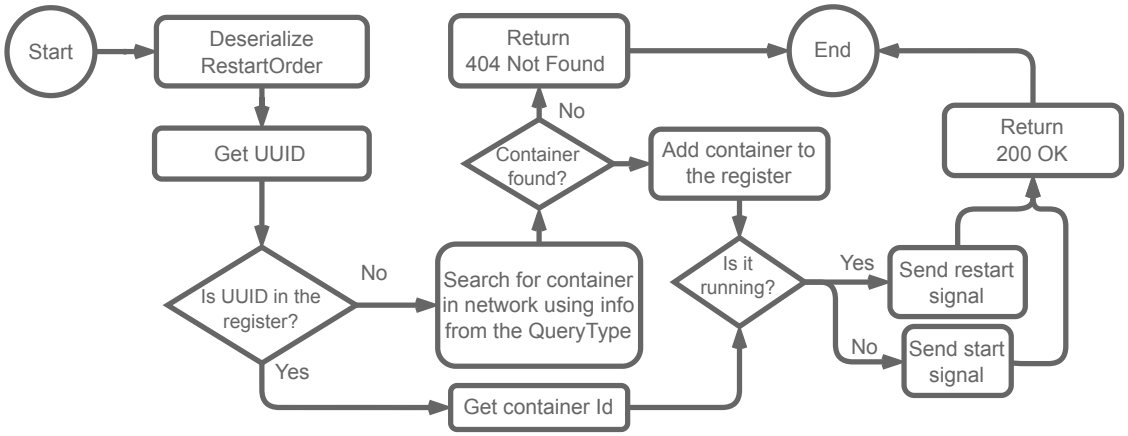


Una vez se tenga el contenedor creado, usando `network_name`, se conectará el contene-

dor 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 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[10], 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.

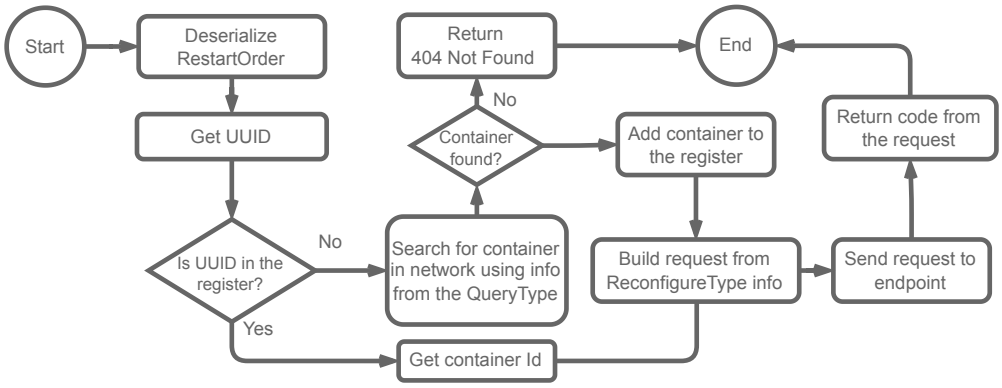
Finalmente, las órdenes de tipo *Reconfigure*, cuyo proceso puede verse en la figura 27, es similar al realizado para las órdenes de tipo *Restart*.

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

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

Figura 27: Proceso para la ejecución de ordenes de reinicio

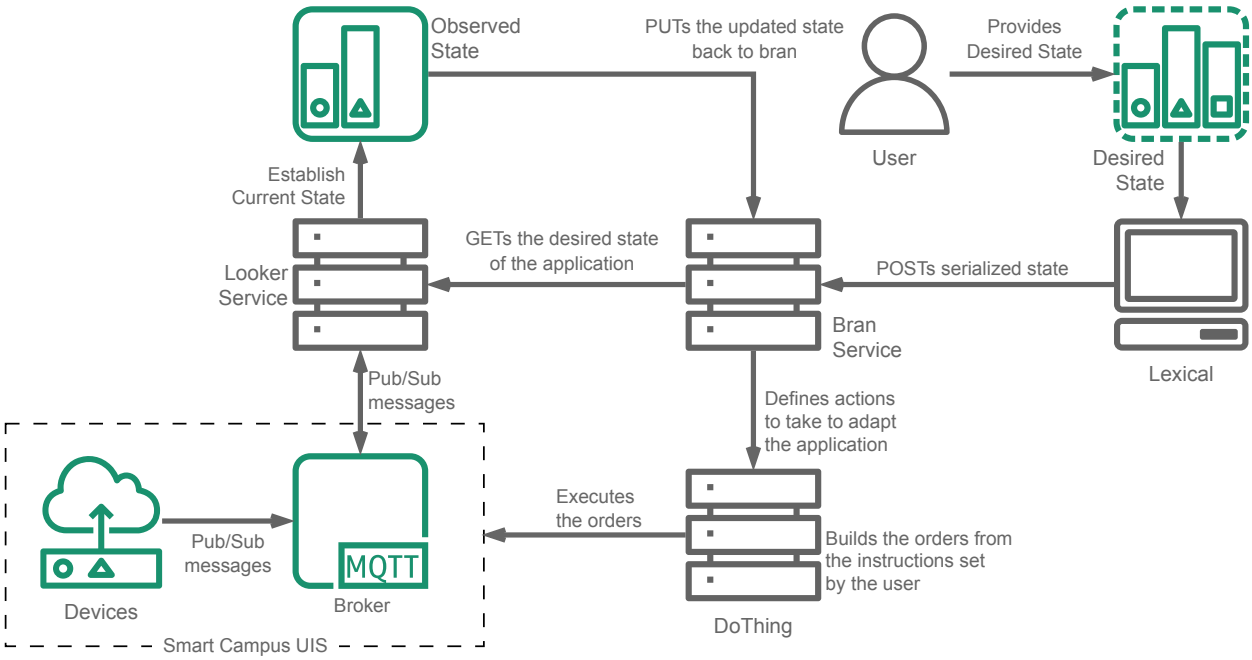


MAPE-K implementado para Smart Campus UIS.

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.

Figura 28: Proceso para la ejecución de ordenes de reinicio



9 Desarrollo Experimental

9.1 Escenario Experimental A

9.2 Escenario Experimental B

10 Análisis de Resultados

10.1 Resultados De Escenario Experimental A

10.2 Resultados De Escenario Experimental B

11 Conclusiones y Trabajo Futuro

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

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.

Tahir, M., Mamoon Ashraf, Q., y Dabbagh, M. (2019). Towards enabling autonomic computing in iot ecosystem. En *2019 ieee intl conf on dependable, autonomic and secure computing, intl conf on pervasive intelligence and computing, intl conf on cloud and big data computing, intl conf on cyber science and technology congress (dasc/picom/cbdcom/cyberscitech)* (p. 646-651). doi: 10.1109/DASC/PiCom/CBDCom/CyberSciTech.2019.00122

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

A Apéndices

A.1 Ejemplo de un YAML de una aplicación válida

```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
```