UNIVERSIDAD NACIONAL DE LA PATAGONIA SAN JUAN BOSCO

Facultad de Ingeniería

Tesina de Licenciatura en Informática

Arquitectura de Comunicación y Almacenamiento para Internet de las Cosas

- **Alumno:** A.P.U. Etchezar, Facundo
- **Tutor:** Lic. Rosales, Pablo Sebastian

Mayo, 2023

Agradecimientos

Si cuando era adolescente me hubieran preguntado dónde me veo en el futuro, como adulto, no hubiera sabido qué responder. Siendo licenciado no creo, encontrando algo en lo que soy bueno y poder vivir de ello probablemente tampoco. Sin embargo ya sea por mérito propio, por temas fortuitos, o un poco de ambos, me encuentro culminando este trayecto después de mucho tiempo.

Agradezco fundamentalmente a mi familia, en especial a mi madre, Miryam De Brito, por apoyarme en todo este proceso, y en una infinidad de cosas más. Agradezco a mi padre, Eduardo Etchezar, que aunque el tiempo empuje su fallecimiento cada vez más hacia el pasado, ha influido en mi vida de una forma que hasta el día de hoy sigo tratando de comprender.

Agradezco a mi tutor, Pablo Rosales, y a mis amigos, por haberme acompañado durante todo este tiempo. Hago mención especial a Ezequiel Suazo y Luis Luna por compartirme su experiencia previa sobre sus respectivos trabajos de tesina.

Agradezco a la universidad, por darme una oportunidad, y a tantas personas más.

Y me agradezco a mi mismo, por querer terminar lo que empecé, y por querer seguir avanzando.

Sea aparente o no, vuelco en esta tesina años de mi experiencia personal y profesional. Espero que para quien la lea sea de utilidad y puedan aprender algo nuevo, y espero que el futuro me enseñe aún mucho más.

Resumen

El avance de la conectividad en el mundo moderno implica que cada vez más contextos comerciales o industriales dependen de la misma para sus procesos, nutriéndose de la información que generan dispositivos y sensores interconectados contínuamente.

En la presente tesina se investigan diversas tecnologías que ofrecen soluciones puntuales a cada eslabón de la cadena de emisión, transmisión y almacenamiento de mediciones, para definir una arquitectura que sea capaz de almacenar estos datos de forma robusta, económica, y que pueda dejarlos disponibles para poder consultarlos de forma estandarizada. Se acompañan las tecnologías seleccionadas con servicios puntuales desarrollados para esta tesina que cumplen funciones no alcanzadas por las herramientas utilizadas.

También se detallan pruebas realizadas para comprender las capacidades en términos de rendimiento y espacio de la arquitectura planteada, y así poder contextualizar los posibles casos de uso de la misma.

Esta tesina está organizada en los siguientes capítulos:

1. Objetivos. Se listan los objetivos puntuales de la tesina.
2. Introducción. Se contextualiza la tesina con un breve repaso de la actualidad en términos de conectividad, dispositivos y mediciones.
3. Revisión del estado del arte. Se detallan diversas tecnologías existentes y otras arquitecturas con fines similares, para entender cuales son los distintos caminos que se pueden tomar al realizar una arquitectura para estos fines.
4. Tecnologías adicionales utilizadas. Se detallan otras tecnologías utilizadas en la tesina que condicionan la arquitectura, pero que van más allá del uso puntual que se les está dando.
5. Desarrollo realizado. Se procede a recorrer y detallar la arquitectura establecida, paso a paso, para comprender cómo cada una de las tecnologías seleccionadas y servicios desarrollados funcionan logrando la arquitectura de historización.
6. Resultados, discusión y conclusiones. Se recolectan los resultados y conclusiones derivadas del desarrollo de la tesina.
7. Trabajos futuros. Se comenta sobre posibles alternativas para expandir el alcance de la arquitectura.

Índice

1 Objetivos 7
1.1 Generales 7
1.2 Específicos 7
2 Introducción 8
3 Revisión del estado del arte 10
3.1 Arquitectura de Microsoft en Azure para IoT 10
3.2 Apache IoTDB 12
3.3 PostgreSQL 13
3.3.1 TimescaleDB 14
3.3.2 Esquema relacional 15
3.4 Message Queuing Telemetry Transport (MQTT) 16
3.4.1 Conceptos y funcionamiento 16
3.4.2 Calidad de servicio 19
3.5 Apache NiFi 19
4 Tecnologías adicionales utilizadas 21
4.1 Eclipse Vert.x 21
4.1.1 Verticle 22
4.1.2 EventBus 22
5 Desarrollo realizado 23
5.1 Dispositivo de borde 25
5.1.1 Broker y cliente MQTT 25
5.1.2 Eclipse Mosquitto 28
5.1.3 Servicio historizr-device 29
5.1.4 Procesamiento de muestras 30
5.1.5 Configuración de señales 32
5.1.6 Base de datos embebida 32
5.1.7 H2 33
5.1.8 Esquema de base de datos 33
5.1.8.1 Tabla data_type 34
5.1.8.2 Tabla signal 35
5.1.9 API HTTP 36
5.1.9.1 /signal 36
5.1.9.2 /device 37
5.1.9.3 /device/discardsamplestate 38
5.1.9.4 /device/discardsamplestats 38
5.1.10 MiNiFi 38
5.2 Servidor de historización 39
5.2.1 NiFi 39
5.2.1.1 PutSQL 43

5.2.2 Esquema de base de datos........ 44
5.2.2.1 Tabla device........ 44
5.2.2.2 Tabla signal........ 45
5.2.2.3 Tabla sample........ 45
5.2.2.4 Integridad del dato........ 45
5.2.2.5 Almacenamiento compacto........ 47
5.2.2.6 Consultas de ejemplo........ 49
5.2.3 Servicio historizr-server........ 50
5.3 Rendimiento en la arquitectura........ 51
5.3.1 Servicio historizr-capture........ 52
5.3.2 Hardware para el dispositivo de borde........ 53
5.3.2.1 Raspberry Pi 3B........ 53
5.3.2.2 Aplicación y capacidades........ 54
5.3.3 Hardware para el servidor de historización........ 55
5.3.4 Entorno de pruebas........ 55
5.3.5 Características de hardware y software........ 56
5.3.6 Método de envío desde MiNiFi........ 57
5.3.7 Benchmark de método de envío........ 58
5.3.8 Benchmark de concurrencia........ 59
5.3.9 Benchmark de hardware........ 59
6 Resultados, discusión y conclusiones........ 62
6.1 Discusión........ 62
7 Trabajos futuros........ 64
7.1 Esquema de seguridad........ 64
7.2 Esquema de múltiples históricos........ 64
8 Glosario........ 66
9 Referencias bibliográficas........ 67

Índice de figuras

Figura 1. Cantidad de dispositivos conectados a internet en el tiempo 8
Figura 2. Crecimiento futuro y de los últimos años de la industria IoT........... 9
Figura 3. Componentes de la arquitectura IoT en Azure 11
Figura 4. Arquitectura de aplicación de IoTDB 12
Figura 5. Ejemplo de payload MQTT en JSON 17
Figura 6. Esquema de comunicación en MQTT 18
Figura 7. Interacción de distintos Verticles con el Eventbus 22
Figura 8. Esquema general de los elementos básicos del entorno 23
Figura 9. Elementos específicos de la arquitectura 24
Figura 10. Ejemplo de estructura en JSON 27
Figura 11. Ejemplo de muestra en JSON 27
Figura 12. Arquitectura interna del servicio historizr-device 29
Figura 13. Flujo de procesamiento de muestras 31
Figura 14. JSON que se emite hacia el servidor 32
Figura 15. Esquema de base de datos en el dispositivo de borde 34
Figura 16. Tipos de datos admitidos por el sistema 34
Figura 17. Ejemplo de señales con su configuración 35
Figura 18. Ejemplo del JSON retornado por el endpoint de /device 37
Figura 19. Flujo de transmisión de mensajes hacia la instancia de NiFi principal 38
Figura 20. Flujo de historización dentro de la instancia principal de NiFi 41
Figura 21. INSERT de la muestra sobre la base 43
Figura 22. Esquema de base de datos en el servidor de historización 44
Figura 23. Sentencia CREATE de SQL para la tabla sample 46
Figura 24. Sentencia CREATE de SQL para la tabla sample simplificada 46
Figura 25. Comparativa entre almacenamiento comprimido, en forma de hypertable, y en tabla tradicional 48
Figura 26. Consulta de estadísticas para una señal 49
Figura 27. Consulta de estadísticas por dispositivo................................... 49
Figura 28. Arquitectura interna del servicio historizr-server 51
Figura 29. Raspberry Pi 3B 54
Figura 30. Comparativa de métodos de envío de NiFi y MiNiFi 58
Figura 31. Comparativa de concurrencia por HTTP 59
Figura 32. Comparativa de los tres dispositivos modelo 60
Figura 33. Comparativa de hardware contra una máquina virtual 61

1 Objetivos

A continuación se detallan los objetivos generales y específicos de la tesina.

1.1 Generales

Desarrollar una arquitectura de comunicación para dispositivos y sensores IoT robusta que contemple los problemas del ciclo de lectura/emisión de los datos tomando en cuenta el procesamiento intermedio y almacenamiento eficiente de las series temporales, utilizando herramientas de bajo costo y ampliables en funcionalidad.

1.2 Específicos

- Optimizar la comunicación entre dispositivos de muestreo y el servidor histórico evitando la emisión de datos redundantes.
- Gestionar dispositivos de muestreo conectados al sistema y las señales que emiten, definiendo mecanismos robustos para configurarlos en la arquitectura.
- Desarrollar un esquema de almacenamiento temporal en los dispositivos de muestreo para que no se pierda información en casos de falla de comunicación.
- Almacenar los datos históricos emitidos, por medio de herramientas que permitan su gestión a largo plazo.
- Diseñar un esquema de base de datos eficiente en almacenamiento y práctico para realizar consultas sobre datos en series temporales.

2 Introducción

En la actualidad la conectividad ha permeado muchos aspectos de la vida cotidiana, con una cantidad diversa de dispositivos que cada vez están más interconectados. El entorno profesional o industrial ha seguido un mismo camino, son cada vez más las industrias que dependen de la conectividad de sus dispositivos para poder funcionar. Se puede imaginar una fábrica que necesita monitorear su maquinaria constantemente para saber su estado de funcionamiento, un interconectado de servicios públicos con diversos dispositivos de monitoreo y medición para determinar presiones de agua, temperaturas, caudales, líneas eléctricas siendo supervisadas para detectar bajadas de tensión, entre otros ejemplos.

A partir de esta idea se establece el concepto de "internet of things", internet de las cosas, o simplemente, IoT. Dave Evans de Cisco menciona que, si bien es un concepto más antiguo, que la internet de las cosas nació aproximadamente en el 2008 (Evans, 2011), visualizado en la Figura 1, cuando se estima que comenzó a haber más dispositivos conectados a internet que personas, una diferencia que fue ampliándose con el tiempo.

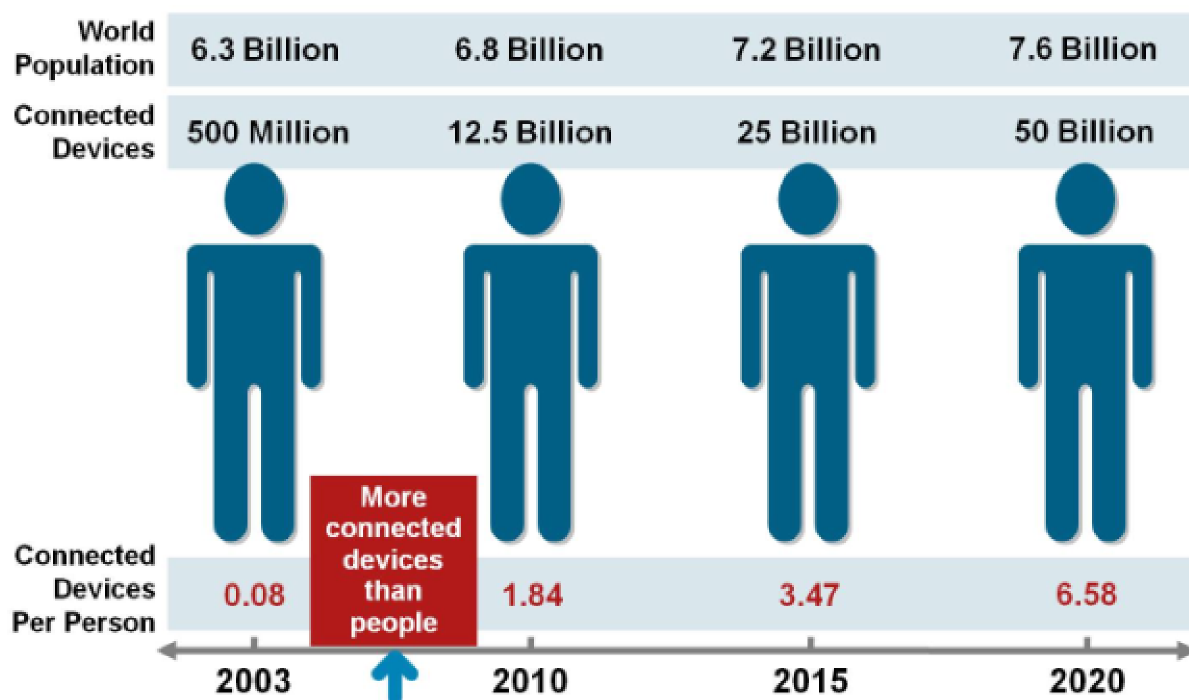

Figura 1. Cantidad de dispositivos conectados a internet en el tiempo.

Incluso con la recesión económica impactando la industria actualmente se espera un crecimiento continuado en los años entrantes, aunque quizás a menor ritmo que en años anteriores como se ve en la Figura 2, realizada por un estudio del mercado por "IoT

Analytics", una empresa dedicada al análisis e inteligencia de negocios para la industria IoT (IoT Analytics, 2023).

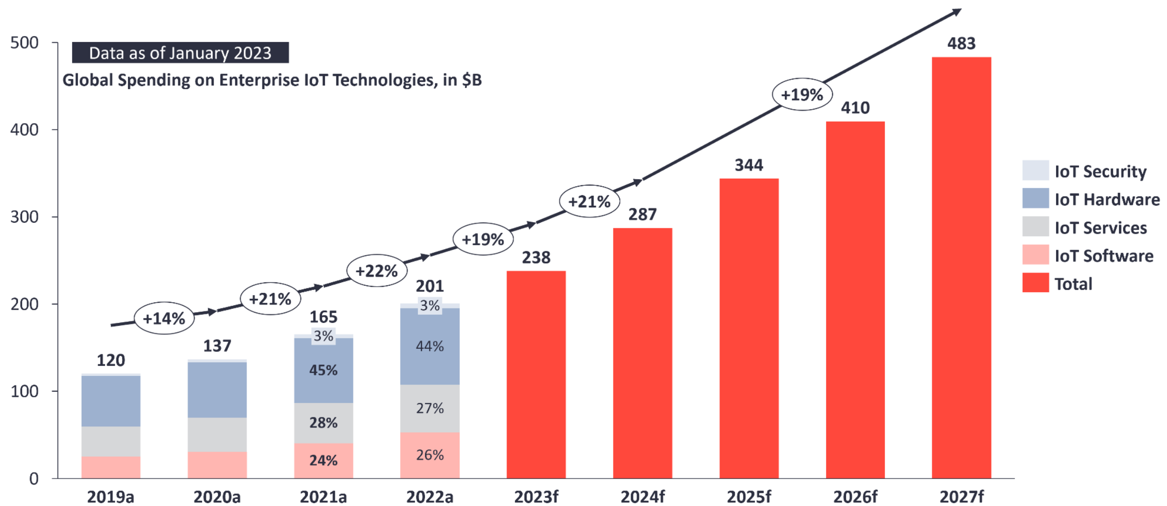

Figura 2. Crecimiento futuro y de los últimos años de la industria IoT.

El monitoreo de estos sensores puede cumplir diversos objetivos en simultáneo: Para detectar problemas de funcionamiento, para determinar acciones preventivas de mantenimiento, o incluso para identificar puntos de mejora, aspectos en los que se podría aumentar la eficiencia de los elementos involucrados.

Además, existe una problemática adicional: el almacenamiento a largo plazo de los datos. Esto se debe a que es necesario conservarlos para realizar análisis posteriores por parte de un experto, generar indicadores y reportes, o incluso aplicar técnicas de machine learning e inteligencia artificial con el fin de extraer el máximo valor de la información disponible.

En esta tesina se propone una arquitectura que da solución a estos aspectos problemáticos, planteando el flujo de información desde su fuente en los dispositivos con sensores, hasta su almacenamiento en una base de datos.

3 Revisión del estado del arte

En el contexto de herramientas o servicios de historización, existen muchas vertientes posibles para tomar. Se encuentran soluciones completas, que buscan otorgar una herramienta para cada etapa del dato, y también existen soluciones puntuales, que restringen su alcance a un solo aspecto de todo el ciclo.

Entre estas se pueden dividir en dos tipos:

- Soluciones de tipo cloud, íntegras, y gestionadas por un proveedor de servicios tercero.
- Soluciones gestionadas propiamente, donde el usuario es el responsable de sus dispositivos y servidores.

Además existen otros aspectos fundamentales a tener en cuenta en todo el flujo de historización, como los protocolos de comunicación utilizados, herramientas de procesamiento de flujos de datos contínuos, entre otras.

En este capítulo se exploran algunos ejemplos de las herramientas que se pueden encontrar, de tal forma de otorgar una perspectiva de lo que existe en el mercado, y también se procede a describir algunas herramientas que efectivamente se utilizaron en el desarrollo de la tesina.

3.1 Arquitectura de Microsoft en Azure para IoT

Microsoft ha desarrollado una arquitectura de referencia para IoT basada en su plataforma de servicios cloud Azure[1]. Esta arquitectura permite a sus clientes entender qué servicios de Azure son útiles en cada etapa del proceso de captura, procesamiento y almacenamiento de datos (figura 3), e implementarla si así lo desean. Esta cubre todo el ciclo de vida del dato, desde su medición hasta una visualización final o analítica que se pueda realizar sobre el mismo, ofreciendo herramientas para cada etapa, permitiendo una gestión completa de los datos en todo momento (Microsoft, 2021).

[1] https://azure.microsoft.com/en-us/

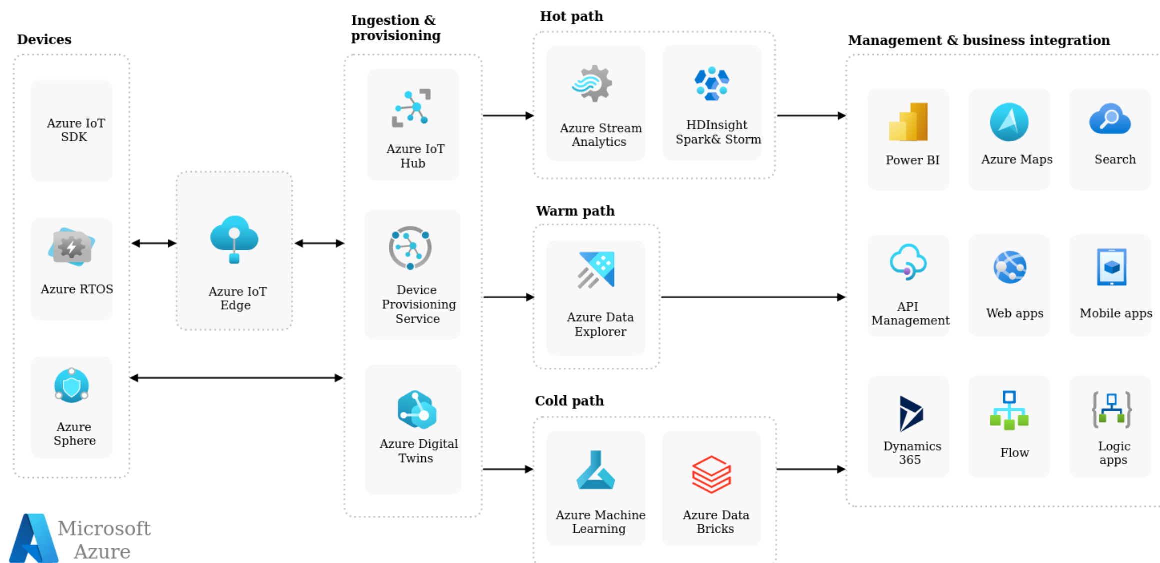

Figura 3. Componentes de la arquitectura IoT en Azure.

Azure, al ser una plataforma cloud, es escalable, ya que abarca todos los data center de los que dispone Microsoft. Cada componente puede escalar en función de lo que necesite, pero por supuesto, esto viene atado a un incremento en los costos.

Sin embargo, como desventaja, se podría mencionar que aunque la mayoría de las herramientas para interactuar con la plataforma son de código abierto y gratuitas[2], la plataforma en sí misma está desarrollada a puertas cerradas y totalmente fuera de control del usuario. Como solución es altamente integrada implicando muy poca fricción en la interacción de sus componentes, por esto también son esperables complicaciones para interactuar con otras herramientas fuera de la plataforma. Por lo tanto, implementar la arquitectura parcialmente, involucrando otros componentes externos, puede hacer que la solución final sea más compleja de lo esperado.

Algo importante a considerar es la dificultad de poder establecer costos al utilizar estas plataformas. Cada componente tiene un esquema de costos diferente y particular, por lo que implementar una solución sin haber especificado su alcance en números precisos (uso de red, cantidad de dispositivos, datos, señales, etc). puede llevar a generar sorpresas en el futuro, especialmente teniendo en cuenta que todos estos costos se manejan en dólares estadounidenses.

Por estas razones se decide no utilizar ninguna de las herramientas que ofrece Microsoft Azure en esta tesina, aunque su descripción sirve para tener una referencia de cómo son las soluciones de tipo cloud que ofrecen los referentes del mercado, ya que es similar a la oferta de Google Cloud y de Amazon Web Services.

[2] https://github.com/Azure

3.2 Apache IoTDB

Los orígenes de Apache IoTDB[3] se ubican en la universidad de Tsinghua, en Pekín, China, en el año 2019 (Wang et al., 2020), es gratuito y de código abierto[4]. En la Figura 4 se ve cómo plantea su despliegue en diversas capas.

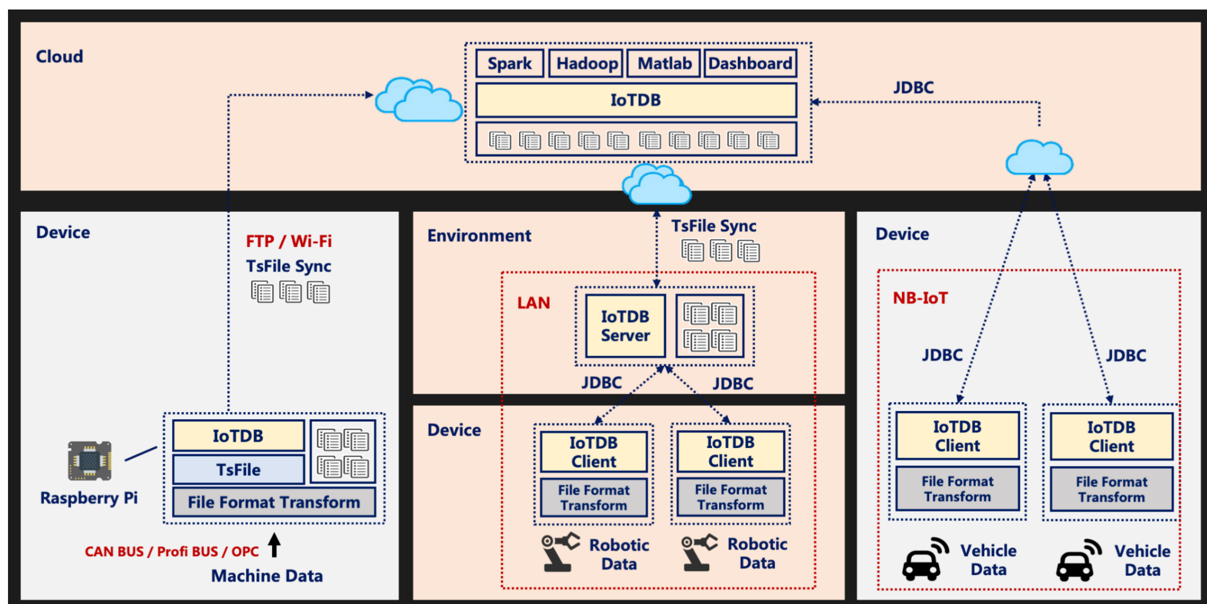

Figura 4. Arquitectura de aplicación de IoTDB.

Apache IoTDB responde a una serie de necesidades en el espacio IoT:

- Almacenamiento. Cómo almacenar series de datos en el tiempo a largo plazo de forma eficiente.
- Integración. Cómo interactuar con este almacenamiento tanto para lectura como para escritura, ya sea por alto rendimiento de ingesta de datos como alto rendimiento en consultas para posterior análisis.
- Escalabilidad. Cómo ampliar las capacidades de la solución de forma dinámica por medio de clustering.

El almacenamiento es administrado por IoTDB en un formato interno llamado TsFile, por lo tanto tiene que ofrecer APIs específicas para interactuar con el mismo. Ofrece compresión y soporte para esquemas "anchos", cada lectura del dispositivo emite múltiples datos al mismo tiempo, y esquemas "angostos", cada señal individual emite datos aislados con su propio timestamp.

Las interfaces ofrecidas para interactuar con el software son flexibles pero limitadas en número:

[3] https://iotdb.apache.org/
[4] https://github.com/apache/iotdb

- REST. Comunicación por HTTP tanto para ingesta de datos como lectura.
- MQTT. Actúa como un broker Message Queuing Telemetry Transport (MQTT), los mensajes publicados desde clientes estándar MQTT son guardados en la base de datos interna.
- Protocolo Influx. Para compatibilidad con clientes de InfluxDB.
- JDBC. Cliente estándar para Java, soportando de forma limitada SQL para consultas. La documentación indica que no es recomendable usar esta vía para escrituras.
- Nativa. Librerías cliente que interactúan por medio de un protocolo interno directamente con IoTDB para Java, Python, C++ y Go.

A su vez IoTDB posee integración con Apache Spark para poder realizar analíticas complejas y distribuidas en ecosistemas como Hadoop sobre los TsFiles.

Los escenarios de trabajo en esta plataforma requieren que en cada dispositivo que se quiera historizar, ya sea de borde o de cualquier tipo, se deba instalar un cliente IoTDB que permita la ingesta de los datos por alguna de las interfaces ofrecidas, por lo tanto no ofrece integraciones con otros protocolos que vayan más allá de los mencionados.

IoTDB tiene aún cierto camino que recorrer. En comparación con otras tecnologías mencionadas, es bastante reciente y se apoya en aspectos no estándares, prefiriendo implementaciones propias. Una característica que se puede identificar como falencia es la forma limitada en la que soporta consultas por SQL[5]. Al ser una base de datos de propósito específico, no está preparada para agregar otros conceptos a su esquema de datos que no sean series de datos temporales, puede llegar a pesar en cualquier sistema que se apoye en la misma, teniendo que acompañar con una base de datos tradicional aledaña para asociar conceptos organizacionales que vayan más allá del dato, dispositivo o señal en sí.

Por estas razones no se selecciona IoT DB para utilizar en esta tesina. Sirve de referencia para comprender el estilo de bases de datos de uso específico que se pueden encontrar.

3.3 PostgreSQL

Como propuesta alternativa a una base de datos específica como lo es IoTDB, también existen bases de datos relacionales genéricas capaces de almacenar información histórica.

PostgreSQL[6] es una base de datos objeto-relacional de amplia trayectoria, tiene sus orígenes en el proyecto POSTGRES en la University of California en Berkeley, y tiene más de 35 años de desarrollo activo en su existencia. Ha tenido una excelente adhesión al estándar SQL, y como ejemplo, en su versión 15 implementa al menos 170 de las 179 características obligatorias del estándar SQL:2016 Core, que en su momento de ser lanzado no era igualado por ninguna otra base de datos en el mercado (The PostgreSQL Global Development Group, 2023).

[5] https://iotdb.apache.org/UserGuide/V1.1.x/Query-Data/Overview.html
[6] https://www.postgresql.org/

Posee ciertas características que lo hacen muy atractivo para el uso en esta tesina. Principalmente soporta todos los sistemas operativos principales, es libre y de código abierto, es utilizado ampliamente por lo tanto es muy sencillo encontrar recursos de como instalarlo y administrarlo sea la plataforma que se esté usando.

Además cuenta con un ecosistema de extensiones principales y comunitarias amplio, del cual se beneficia directamente esta tesina al utilizar el plugin TimescaleDB para una de las tablas más importantes, la tabla histórica de muestras.

También brinda drivers para una gran cantidad de lenguajes de programación y plataformas, dado que su protocolo de comunicación es estable y versionado[7], a tal punto de que existen varias implementaciones para un mismo lenguaje con distintas características. Por ejemplo, el driver asincrónico que utiliza el framework de servicios utilizado en esta tesina, Vert.x, que es distinto del driver convencional JDBC utilizado usualmente en Java.

3.3.1 TimescaleDB

TimescaleDB[8] es una extensión para PostgreSQL, desarrollada en C, libre y de código abierto. El modelo de negocio de la empresa que lo desarrolla, Timescale, apunta a clientes en la nube. Ofrecen un servicio pago de instancias de PostgreSQL con TimescaleDB en la nube (AWS, Azure, entre otros) y gestionado por la empresa. Además, su extensión se ofrece para realizar "self-hosting" si se lo prefiere. Es decir, uno puede simplemente agregar su repositorio e instalarla sobre una base de datos propia, sin costo alguno. La extensión surge de la necesidad de poder guardar información que abarque grandes cantidades de tiempo y muestras.

Las bases de datos relacionales con sus características por defecto quizás no están lo mejor adaptadas para este tipo de información sin hacer esquemas complejos de particionamiento y distribución de los datos.

TimescaleDB entiende que para un mismo servidor, una misma base de datos, e incluso en la misma tabla, pueden haber distintas necesidades de consulta de datos dependiendo del objetivo, si es monitoreo en vivo de datos recientes o analítica en masa de datos antiguos. Para solventar este problema presenta un concepto nuevo, la "hypertable". La hypertable es una tabla convencional relacional que por debajo puede estar particionada en distintas tablas con distintos criterios de almacenamiento.

En el caso concreto de tener una tabla con larga historia de métricas de diverso tipo, y entendiendo que se puede querer consultar por ejemplo el ultimo dia para monitoreo en vivo, y un año entero para realizar analitica offline, ofrece la posibilidad de captar estas dos

[7] https://www.postgresql.org/docs/current/protocol.html
[8] https://www.timescale.com/

necesidades sin tener que distribuir esta información en distintas tablas de distinto propósito o distintas bases de datos.

En este caso se puede definir un hypertable con esta información, que se divide en una tabla nueva o “chunk” cada 3 días, almacenados en un dispositivo de alto rendimiento (SSD, M2, etc). A su vez, se puede definir que los chunks más antiguos de 10 días se muevan a otro tipo de almacenamiento en discos tradicionales, que suelen ser más baratos y se dispone de más capacidad en los mismos.

De esta forma, con una misma hypertable y completamente transparente al uso, se puede elaborar un esquema que permita disponer de la información reciente y más consultada en vivo en almacenamiento rápido pero potencialmente costoso, a medida que continuamente se migra la información menos consultada al almacenamiento más amplio, lento y a su vez menos costoso.

Este es uno de los principales casos de uso de la extensión pero sus capacidades son muy amplias, algunas de sus características principales son:

- Compresión
  - Logrando almacenar en 1/10 del espacio la misma información, y potencialmente mejorando los tiempos de consulta al necesitar la base de datos leer menos de disco para resolver las consultas.
- Políticas de retención de información
  - Para eliminar información demasiado antigua que no se quiera almacenar automáticamente.
- Funciones analíticas adicionales
  - Varias funciones de interpolación o agregados por intervalos de tiempo, difíciles de realizar en PostgreSQL base.
- Agregados continuos de datos
  - Para calcular promedios, máximos, etc y almacenarlos a medida que llega la información a la tabla destino, de tal forma de evitar costosos procesos de ETL o consultas masivas para resolverlos.
- Hypertables distribuidas
  - Agrega capacidades multi nodo para distribuir las hypertables en distintos servidores transparentemente.

Puntualmente en esta tesina se utiliza la funcionalidad de hypertables[9], su división de chunks y la de compresión para realizar distintas pruebas de capacidad de almacenamiento.

3.3.2 Esquema relacional

Existen distintas soluciones para almacenar información histórica, llamada también “time series”, con gran parte de éstas siendo software de propósito específico, como por ejemplo

[9] https://docs.timescale.com/getting-started/latest/create-hypertable

InfluxDB, IoTDB o Proficy Historian, herramientas que no responden a las interfaces comunes de las bases de datos relacionales. De forma contrapuesta a este aspecto, cabe destacar el gran beneficio de poder almacenar este tipo de información en una base de datos relacional y tradicional, ya que elimina una gran cantidad de fricciones en su uso. Muchas herramientas de reportes soportan la mayoría de las bases de datos relacionales, así que no es necesario desarrollar ni pagar por extensiones específicas para alguna base de datos para información temporal.

TimescaleDB se gestiona de la misma forma que uno gestionaría cualquier base de datos PostgreSQL, así que las herramientas convencionales funcionan de igual manera (gestores visuales como DBeaver, DataGrip, Squirrel SQL, pgAdmin, etc). No se necesita ningún conector especial así que es posible utilizar todos los entornos y lenguajes para los que PostgreSQL ya funciona (Java, C#, JavaScript, Python, etc).

Al ser una base de datos convencional, la metodología de consulta sigue siendo la misma así que cualquier usuario que sepa SQL va a poder utilizar TimescaleDB y realizar las consultas que requiera. En este sentido se torna mucho menos complejo incorporar esta herramienta en un entorno convencional con otros sistemas y equipos de personas, ya que no requiere de un conocimiento específico para realizar consultas básicas (Freedman, 2020).

3.4 Message Queuing Telemetry Transport (MQTT)

Si se busca información relacionada a internet de las cosas, es imposible no encontrarse con la mención de MQTT. Si bien no ofrece un mecanismo de almacenamiento, su simpleza y eficiencia lo hacen idóneo para ser implementado en un diverso conjunto de dispositivos de medición. MQTT es un protocolo de mensajería ligero, sigue un esquema publicador-consumidor, diseñado para dispositivos de bajos recursos y con una conexión poco confiable.

El protocolo tiene cierta antigüedad, ya que fue inventado en 1999 por el Dr. Andy Standford-Clark de IBM y Arlen Nipper de Arcom (ahora Eurotech), es utilizado actualmente en un espectro amplio para interconectar diversos dispositivos entre sí (MQTT.org, 2022).

Es utilizado en contextos industriales, por ejemplo en un whitepaper de HMS Industrial Networks, se ve como MQTT se ha implementado efectivamente como protocolo de transmisión de alta frecuencia entre controladores emitiendo métricas de puntos de soldadura en una fábrica, y una base de datos temporal para su posterior análisis e historización (HMS Industrial Networks, 2019).

3.4.1 Conceptos y funcionamiento

MQTT posee algunos conceptos claves como broker, suscriptor, publicador, tópico, cliente, calidad de servicio/quality of service (QoS), entre otros.

El broker es un proceso que funciona independientemente, es usualmente un servicio que se instala en un dispositivo particular y corre continuamente. El protocolo establece un esquema de múltiples publicadores y múltiples suscriptores, por lo tanto es el broker el que tiene que arbitrar este traspaso de mensajes.

Los clientes publicadores son los que generan mensajes para transmitir por el protocolo MQTT, estos están conectados a un broker, y publican sus mensajes con un payload específico a lo que se le dice un tópico. El tópico es una cadena de texto que identifica la casilla donde estos mensajes en particular van a ser publicados a los clientes que quieran suscribirse. Por otro lado, los clientes suscriptores anuncian al broker a qué tópicos quieren suscribirse, y el broker les envía los mensajes publicados a los mismos a medida que llegan.

Si bien no es obligatorio para el funcionamiento del protocolo, los tópicos suelen tener la siguiente forma:

dispositivo1/instrumento1/sensor1
dispositivo1/instrumento1/sensor2
dispositivo1/instrumento2/sensor3
dispositivo1/instrumento3/sensor1

Se distingue entre los slashes / “niveles” de los tópicos, en el ejemplo dispositivo1 es el 1er nivel, instrumento1 e instrumento2 son el 2do nivel, y finalmente sensor1, sensor2 y sensor3 son el 3er nivel de los tópicos existentes en el broker.

En cada uno de estos tópicos, los clientes publicadores emiten los mensajes que quieren comunicar al broker. Suponiendo que el sensor2 del instrumento1 es una medición de temperatura ambiental, un payload para este tópico podría ser el JSON de la Figura 5:

```
1  {
2    "value": 18.5,
3    "tstamp": "2022-06-17T23:31:00"
4  }
```

Figura 5. Ejemplo de payload MQTT en JSON.

Esto denota que la temperatura del sensor2 a las 23:31 del 17 de junio fue de 18 grados y medio.

MQTT no tiene restricción alguna con respecto al formato del payload en sí. Puede ser texto plano, un objeto binario, o algún elemento serializado en texto como el JSON ejemplificado.

El protocolo, en base al esquema descrito de tópicos, también soporta una forma genérica de suscripción.

Continuando el ejemplo anterior, en un broker pueden haber 3 clientes distintos, cada uno suscripto a cada sensor individualmente, y cada uno recibiendo mensajes sólo del sensor que les interesa. Si bien esto es posible, lo más común es que los suscriptores estén interesados en un abanico de tópicos en conjunto, en vez de uno solo en particular.

MQTT define máscaras para hacer estas suscripciones maś prácticas por medio de carácteres especiales: “#” y “+”.

El carácter “#” en la suscripción indica que se quiere suscribir a una cantidad de niveles indefinida a derecha del carácter especial. Mientras tanto el carácter “+” indica que sólo se quiere suscribir a un nivel en particular de los tópicos. Por ejemplo, si un cliente se quisiera suscribir a todos los sensores del dispositivo1, podría hacerlo con la siguiente máscara de suscripción:

dispositivo1/#

Esto le indicaría al broker que el cliente está interesado en el sensor1 y sensor2 del instrumento1, sensor3 del instrumento2 y sensor1 del instrumento3.

A su vez, si un cliente fuera más particular, y solo quisiese suscribirse al sensor1 de cada instrumento presente, lo podría hacer de la siguiente forma:

dispositivo1/+/sensor1

El ejemplo queda esquematizado en la Figura 6:

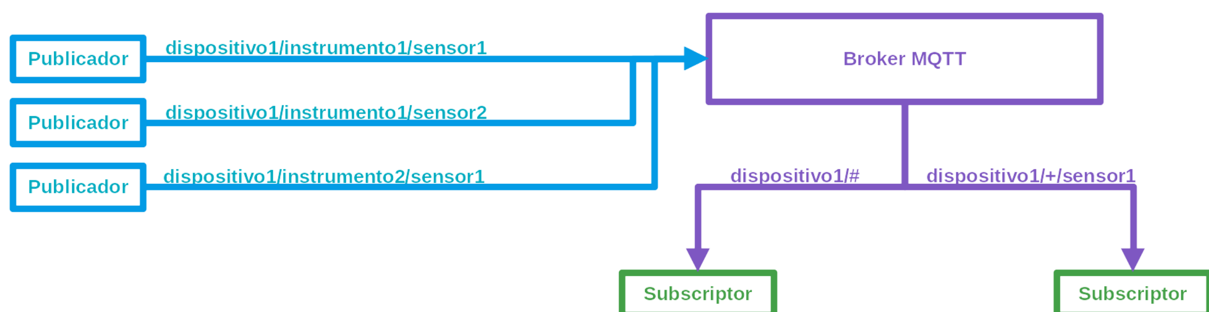

Figura 6. Esquema de comunicación en MQTT.

De esta forma se indica que sólo se quiere un nivel entre sensor1 y dispositivo1. En este caso el broker entrega al suscriptor los valores de sensor1 tanto de instrumento1 como de instrumento3 (siguiendo el ejemplo, instrumento2 no tiene un sensor1, por lo tanto no existen mensajes para el mismo que emitir).

3.4.2 Calidad de servicio

MQTT posee un concepto llamado calidad de servicio o quality of service (QoS). Categorizado en 3 niveles, se comunica al broker en el momento de la publicación o suscripción del cliente[10], sus características:

0. El nivel 0 se llama "at most once", a lo sumo una vez. Esto implica que la entrega del mensaje a los suscriptores del mismo puede suceder o no. Es un nivel de alto rendimiento ya que no tiene garantías de entrega, lo que reduce trabajo en el broker y carga en la red. Sólo se recomienda en redes con una comunicación garantizada o para mensajes que quizás no sean muy importantes.

1. El nivel 1 se llama "at least once", al menos una vez. Este nivel implica que el broker tiene que garantizar a los suscriptores que los mensajes que maneja en un tópico tienen que ser emitidos al menos una vez a los suscriptores. Este modo también es de alto rendimiento ya que solo espera una respuesta de llegada del cliente, con la contra de que si no responde lo suficientemente rápido, el broker puede emitir el mensaje más de una vez. Por lo tanto, los clientes tienen que estar preparados para posiblemente recibir información duplicada en casos de problemas en la conexión.

2. El nivel 2 se llama "exactly once", exactamente una vez. Este nivel implica que el mensaje publicado al broker se va a entregar exactamente una única vez a los suscriptores del tópico a donde fue publicado. Este es el modo de más bajo rendimiento ya que requiere de una interacción más compleja entre el broker y los suscriptores para garantizar que el mensaje no es emitido más de una vez. Este modo solo se recomienda en casos donde los clientes bajo ningún motivo pueden manejar mensajes duplicados si los reciben.

Relacionando con el whitepaper de HMS, su arquitectura está armada en una red interna dentro de una misma ubicación física, por lo tanto las publicaciones y suscripciones se pueden realizar con QoS en 0. No sería el caso si esta red fuera por aire y requiriera de garantías más fuertes de la transmisión del dato.

3.5 Apache NiFi

Apache NiFi[11] es un proyecto que tiene sus orígenes en la National Security Agency de los Estados Unidos de América. Fue relanzado como software de código abierto en 2014 y donado a la Apache Software Foundation (Bridgwater, 2015).

Es una herramienta de amplias capacidades pero fundamentalmente su objetivo es procesar flujos contínuos de datos de forma dinámica y fácilmente configurable, por medio de nodos "procesadores" interconectados. Posee una interfaz web para poder configurar sus flujos de procesamiento, y una gran cantidad de nodos con funcionalidad predefinida para elegir.

[10] https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/
[11] https://nifi.apache.org/index.html

El proyecto cumple una larga lista de ambiciosos objetivos como lo son:

- Seguridad
    - NiFi posee muy buenas capacidades de seguridad como manejo de certificados, versiones seguras de los protocolos individuales que puede manejar (como HTTP vs HTTPS), permisos de usuario con su respectiva visibilidad del sistema, etc.
- Robustez del dato
    - NiFi almacena su estado en memoria en un repositorio persistente en disco con capacidad de restauración en caso de paros inesperados del servicio.
- Flexibilidad
    - La base de NiFi es la composición de piezas lógicas predefinidas que realizan tareas puntuales llamados Processors, estos implementan procedimientos individuales como parsear un mensaje en JSON, realizar una petición HTTP, leer un archivo de disco, guardarlo, suscribirse a un broker MQTT, o emitir mensajes en el mismo, etc.
    - Provee una interfaz visual web para configurar y armar “flujos” de estos procesadores fácilmente.
    - Posee la capacidad de scripting, que permite aplicar fácilmente lógica extra no contemplada en los Processors que vienen por defecto con la aplicación.
- Escalabilidad
    - NiFi posee capacidad de cómputo distribuido, repartiendo sus mensajes para ser procesados en varias instancias de NiFi separadas.
    - Cada processor puede ser configurado para correr una X cantidad de instancias concurrentes del mismo, por lo cual es muy sencillo, por medio de la interfaz web, paralelizar un flujo para tomar ventaja de los CPUs multi núcleo que abundan hoy en día.
- Comunicación
    - NiFi posee una gran capacidad de comunicación por diversos protocolos con los Processors que vienen por defecto (SQL, AMQP, MQTT, HTTP, FTP, entre otros), además de la capacidad de comunicación NiFi a NiFi que posee internamente, de tal forma de emitir mensajes directamente hacia otros NiFi como si estuvieran todos en un mismo flujo de Processors.

4 Tecnologías adicionales utilizadas

A lo largo del desarrollo de esta tesina, se seleccionan varios elementos que realizan tareas fundamentales, puntualmente ciertas tecnologías como MQTT, PostgreSQL, NiFi y TimescaleDB. Más allá de estas tecnologías base, se necesitan desarrollar algunos servicios puntuales, para estos se selecciona un framework base en Java que se detalla en este capítulo.

4.1 Eclipse Vert.x

Eclipse Vert.x[12] es un proyecto implementado en Java que abarca una serie de herramientas orientadas a desarrollar servicios de distinta índole, principalmente con tecnologías web y de mensajería distribuida, que tiene sus orígenes en 2011. Desarrollado por Tim Fox mientras trabajaba en VMWare, luego fue migrado a la Eclipse Foundation cuando Tim Fox comenzó a trabajar en Red Hat (Phipps, 2013).

Inspirado originalmente por Node.js, busca ser eficiente y modular. Posee un diseño orientado a operaciones asincrónicas, es decir, en vez de un solo hilo de lógica que maneje una petición entrante y bloquee en operaciones de larga duración (ej, acceso a base de datos), está diseñado para otorgar varias interfaces asincrónicas las cuales se utilizan por medio de callbacks que se llaman al terminar cada sub-operación. De esta manera, el desarrollador esencialmente indica qué se tiene que realizar una vez que una operación bloqueante es realizada sin tener que esperar el resultado.

Esta orientación en su diseño permea todo el proyecto y todas sus funciones, sean de acceso a base de datos, sistema de archivos, lectura de configuración, emisión de mensajes distribuidos, inicialización de sub-servicios, etc. Todas estas funciones se realizan de forma no sincrónica y bajo el entendimiento de que ninguna operación de potencialmente larga duración debería bloquear el hilo principal de trabajo.

Uno de los principios fundamentales de Vert.x es la modularidad. Vert.x core define una serie de funciones básicas expuestas en módulos individuales que dependen del módulo principal vertx-core, que pueden ser combinadas según lo requiera la aplicación[13].

Cabe destacar que Vert.x es un proyecto amplio y que en esta tesina solo utiliza un subconjunto de sus funciones, particularmente el módulo vertx-web con el cual se definen APIs REST, vertx-jdbc-client y vertx-pg-client para manipular el acceso a base de datos, entre otros.

Otras características interesantes de Vert.x pero no utilizadas en esta tesina en particular podrían ser el clustering para realizar servicios con capacidades distribuidas, funciones de alta disponibilidad y failover, entre otras.

[12] https://vertx.io/
[13] https://vertx.io/docs/

4.1.1 Verticle

Un concepto principal de Vert.x es el Verticle[14]. Es un conjunto de lógica relacionada, una serie de tareas a realizar en base al objetivo lógico de la aplicación. Los Verticles se podrían describir como sub-servicios de una aplicación. De esta forma una aplicación puede estar conformada por varios Verticles funcionando en paralelo, de forma independiente, pero comunicándose entre sí mediante eventos para realizar tareas en conjunto. Para implementar un verticle se extiende la clase abstracta AbstractVerticle.

4.1.2 EventBus

Para interconectar estos Verticles, Vert.x implementa un bus de eventos apropiadamente llamado EventBus[15]. El EventBus posee rutas a las que se pueden emitir mensajes, o suscribir para recibir mensajes emitidos a las mismas. Estas rutas no son más que un string que identifica la “casilla” a donde son registrados los mensajes manejados por el EventBus.

Responde a una clásica arquitectura productor-consumidor en ese sentido, donde distintos Verticles pueden generar unidades de trabajo o notificaciones de operaciones realizadas, y otros Verticles pueden estar suscritas a las mismas, para poder realizar otro tipo de tareas con los resultados. En la Figura 7 se ilustra un esquema de comunicación entre verticles por medio del EventBus.

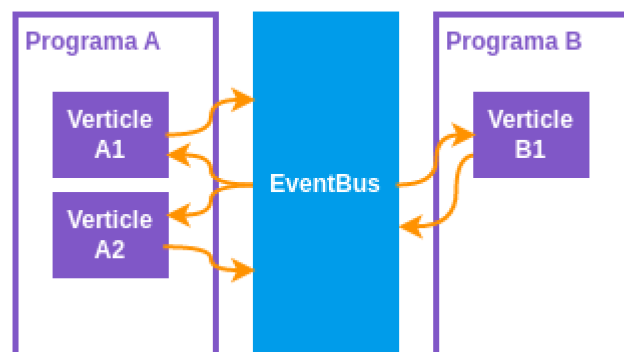

Figura 7. Interacción de distintos Verticles con el Eventbus.

[14] https://vertx.io/docs/vertx-core/java/#_verticles
[15] https://vertx.io/docs/vertx-core/java/#event_bus

5 Desarrollo realizado

Dado este marco de trabajo, se parte del siguiente modelo abstracto del problema, representado en la Figura 8.

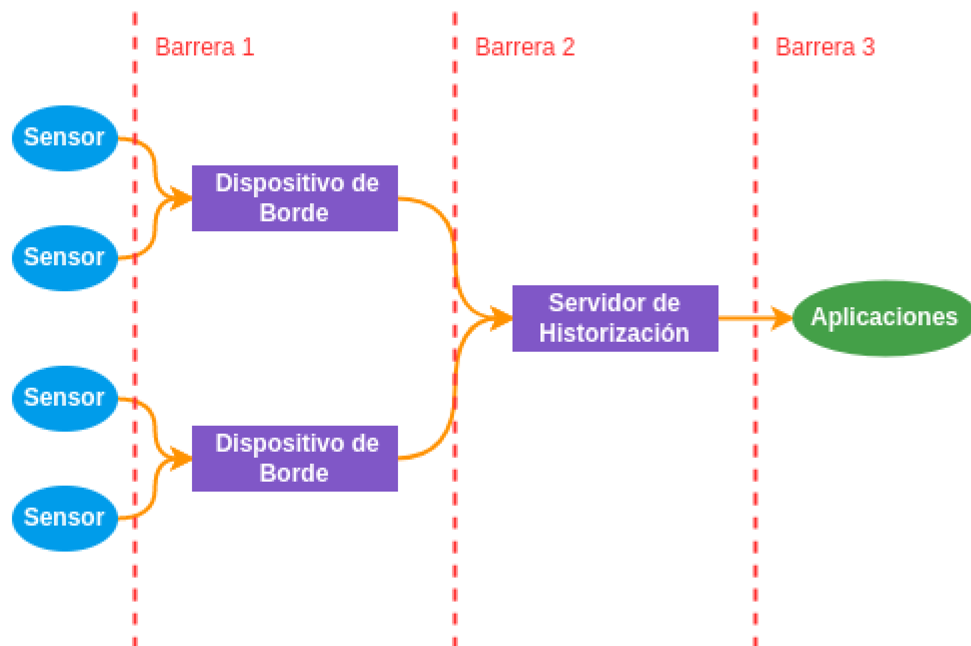

Figura 8. Esquema general de los elementos básicos del entorno.

De izquierda a derecha se tienen a los sensores, que pueden emitir información de distinta índole, los dispositivos de borde, computadoras que cumplen el propósito de encuestar los sensores o recibir sus datos emitidos, el servidor de colección e historización de datos, que se conecta con los dispositivos de borde para recibir, organizar, y almacenar estos datos potencialmente a largo plazo. Finalmente se tienen las aplicaciones, el software que se alimenta de estos datos para funcionar, sean reportes, servicios de analítica, o visualizadores para que expertos puedan evaluar la información que deseen.

Con esto se pueden determinar 3 barreras fundamentales para la información:

1. La barrera entre el sensor y el dispositivo de borde concentrador. Esta puede no existir si el mismo sensor es un componente complejo capaz de emitir los datos hacia el servidor de historización independientemente. Pero también se puede tener un dispositivo de borde, como se ve en la Figura 8, que concentra la información de múltiples sensores conectados al mismo, directamente a través de pines de lectura digital o a través de una conexión de red por ejemplo. Esta conectividad va a estar determinada por los protocolos manejados por el sensor que se desea incorporar. Se entiende que al ser un dispositivo de borde, la ubicación física del dispositivo es cercana al sensor, por lo tanto la conectividad debería ser estable y suficiente.
2. La barrera entre el dispositivo de borde y el servidor de historización. Esta conectividad puede ser lograda por un protocolo estándar como una red TCP/IP, o por

algún protocolo propietario dependiendo de la tecnología utilizada. Se entiende que esta conectividad puede ser inestable dependiendo del dominio del problema. Por ejemplo, una red interna dentro de una fábrica puede ser muy estable, cableada y de gran ancho de banda, mientras que una estación metereológica a kilómetros de distancia puede verse afectada por cuestiones climatológicas, o por el alcance de una red inalámbrica. Es importante que el dispositivo de borde sea capaz de restablecer la conexión con el servidor de historización en caso de pérdida de conectividad, y es deseable que sea capaz de almacenar la información generada en casos de falla de conexión para poder enviarla cuando se restablezca.

3. La barrera entre el servidor de historización y las aplicaciones que van a utilizar la información almacenada. Se entiende que esta conectividad es confiable ya que partiría de una serie de herramientas dentro de una organización que requieran de estos datos internamente, pero más allá de eso es importante el protocolo por el cual se disponibiliza esta información. Idealmente debería ser un protocolo muy usado, de tal forma que la fricción al incorporar aplicaciones de reportes, analítica y demás, sea la menor posible.

Teniendo este panorama descrito, se procedió en esta tesina a redefinir cada una de estas partes, aplicando tecnologías concretas en cada etapa y de la siguiente forma, ilustrado en la Figura 9:

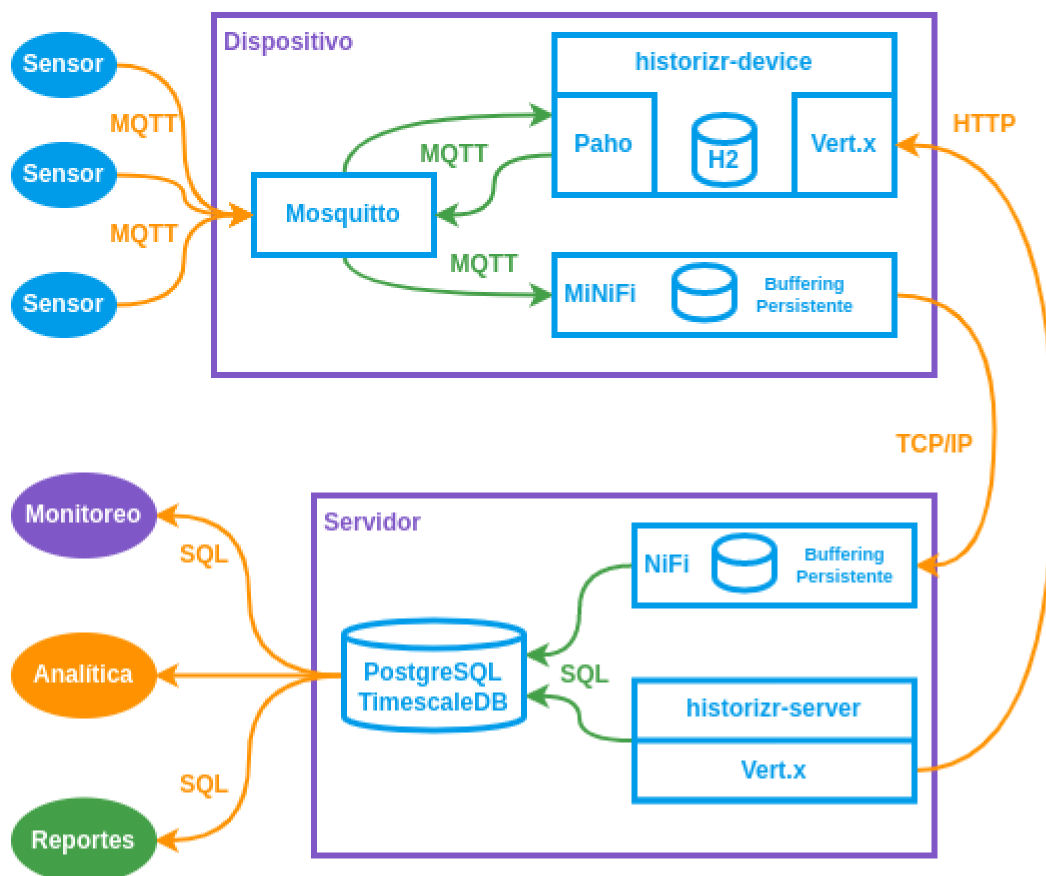

Figura 9. Elementos específicos de la arquitectura.

Se definen dos entornos principales en los cuales la solución va a operar:

En primer lugar se tiene el dispositivo de borde, el cual va a concentrar una serie de servicios dispuestos para la captura, preprocesamiento, configuración y almacenamiento temporal de las muestras medidas. El objetivo de este dispositivo es disponibilizar los datos de una forma que permita su ingesta fácilmente en el servidor historizador, además de proveer facilidades para poder reducir la emisión de información redundante si se quisiera. Se llama dispositivo de "borde" ya que idealmente se encuentra físicamente cerca de los instrumentos de medición que están emitiendo los datos.

En segundo lugar se encuentra el servidor historizador. Éste dispone de una serie de servicios para recibir los datos de los dispositivos de borde, gestionarlos y almacenarlos, dejándolos disponibles de una forma adecuada y eficiente para su posterior consulta.

La solución asume conectividad por una red TCP/IP entre los dispositivos de borde y el servidor historizador, aunque no asume que la calidad de esta conectividad es buena. Por lo tanto, si se toma en cuenta que puede ser intermitente, la solución está ideada para manejar pérdidas de comunicación entre el dispositivo de borde y el servidor historizador. Si bien no se determina un piso de ancho de banda necesario, se entiende que éste, junto a las capacidades de hardware de los dispositivos involucrados, van a ser la limitante a la hora de determinar qué capacidad de transmisión y almacenamiento de datos se puede lograr. En otras palabras, un mejor hardware y conectividad van a dar lugar a una ampliación de las capacidades de la arquitectura propuesta ya que varios de los componentes elegidos en la arquitectura son muy escalables (NiFi, MiNiFi, PostgreSQL, etc).

Tanto los servicios desarrollados puntualmente para la tesina como scripts adicionales, archivos de configuración y resultados de pruebas se dejaron disponibles en un repositorio público en GitHub, en el siguiente link: https://github.com/dustContributor/historizr

5.1 Dispositivo de borde

El dispositivo de borde en la arquitectura es de propósito general con tres servicios base: un broker MQTT, una instancia del servicio historizr-device, y una instancia del servicio MiNiFi.

5.1.1 Broker y cliente MQTT

El punto de entrada de los datos de los sensores a la arquitectura es un broker MQTT. Se parte de este por una serie de factores importantes:

Como protocolo, MQTT es sencillo de entender y existen varias herramientas gratuitas para poder manipularlo. Esto hace práctico el debugging del envío de los mensajes, ya que se

pueden interceptar los mismos conectándose al broker MQTT con cualquier cliente convencional, e inspeccionar los datos que se están emitiendo.

Al ser sencillo, MQTT es ubicuo en aplicaciones IoT, existen muchos dispositivos que lo implementan, hay multitud de librerías para desarrollar clientes que responden a este protocolo en cualquier lenguaje de programación popular, y nuevamente, la mayoría de estos son gratuitos, de tal forma que la “barrera de entrada” para desarrollar un programa driver/gateway que muestre algún dato de un sensor, y lo emita hacia un broker MQTT, es muy baja.

Puntualmente el broker seleccionado es Eclipse Mosquitto[16], muy eficiente en su uso de memoria, compacto, multi plataforma, multi arquitectura y gratuito. Estos factores lo hacen idóneo para poder ubicarse en el borde, es decir, cerca de la emisión del dato.

La librería cliente elegida para Java, Eclipse Paho[17], es de amplio uso por lo tanto existe buena documentación para la resolución de problemas.

Habiendo definido el punto de entrada en el broker MQTT, es necesario definir cómo se expone este punto de entrada. En base a lo descrito en el capítulo 3.4 se tiene que elegir un formato de tópicos de entrada, y su contenido, para que la arquitectura pueda adquirirlo y comience el ciclo de historización sobre el dato.

En principio se establece que cada señal historizada tiene un nombre, configurado por medio de la API REST por HTTP del servicio historizr-device instalado y funcionando en el mismo dispositivo. Se define un tópico de entrada el cual respeta un nombre “input” para que claramente sea identificado como el mismo. Juntando los dos conceptos se establece que los tópicos de entrada van a respetar el formato:

input/[nombre de señal]

Por ejemplo, si se estuviera midiendo dos temperaturas:

input/temperatura1
input/temperatura2

De esta forma se logra la independencia de funcionamiento entre señales. Con este esquema es posible tener distintos servicios que obtengan muestras de distintos instrumentos a diferentes velocidades. Por ejemplo, una temperatura de algún componente electrónico podría requerir tasas de emisión rápidas cada 1 segundo, mientras tanto una temperatura ambiente podría ser emitida cada 20 minutos.

[16] https://mosquitto.org/
[17] https://www.eclipse.org/paho/

Cada tópico tiene que tener un formato de mensaje comprensible por el servicio historizr-device para que sea capturado, por facilidad se establecieron dos formatos posibles, uno “simplificado” y uno “completo”.

El formato completo sería que el contenido del mensaje en el tópico sea texto JSON con el siguiente formato, ilustrado en la Figura 10:

```
{
  "v": [valor de la muestra],
  "t": "[timestamp de la muestra]",
  "q": [calidad de la muestra]
}
```

Figura 10. Ejemplo de estructura en JSON.

Siendo que el tópico de donde proviene este mensaje indicará a qué señal pertenece la muestra, no es necesario especificar el nombre o identificador de la señal en el mismo mensaje que contiene el dato. Se eligen abreviaciones de “value”, “timestamp” y “quality” para reducir la memoria requerida, sea para emitir o para recibir el mensaje. No es estrictamente necesario, pero no impacta mucho en la implementación de historizr-device y es una forma sencilla de reducir el tamaño de los mensajes.

Partiendo de esta información necesaria para la muestra se puede establecer un formato simplificado donde el contenido del mensaje sea el mismo valor que se desea historizar.

Siguiendo el ejemplo anterior, uno podría tener una temperatura ambiente de la siguiente forma “completa”:

Tópico:
input/temperatura1

Donde el contenido se ve en la Figura 11:

```
{
  "v": 21.5,
  "t": "2022-10-12T13:30:00Z",
  "q": true
}
```

Figura 11. Ejemplo de muestra en JSON.

Asumiendo que la calidad por defecto de la emisión del dato es buena, y que el timestamp es el de recepción del mismo, se podría elaborar una forma “simple” como se ve a continuación.

Tópico:
input/temperatura1

Contenido:
21.5

De esta forma el ingreso del dato a la arquitectura queda reducido a su mínima expresión: saber a qué señal corresponde por su tópico, y saber qué valor tiene por el contenido.

En este ejemplo en particular el payload del mensaje se vería reducido de 46 bytes en el caso “completo” a solamente 4 bytes en el caso simplificado, es decir, reduciendo en un 91% el tamaño del mensaje. Esto puede ser un aspecto importante a considerar dependiendo del caso de uso, por lo tanto se implementaron ambos métodos para ingresar datos a la arquitectura, un tipo de payload “completo” y uno “simple”.

Para empezar a capturar estos datos historizr-device no requiere más que suscribirse con la máscara input/# para empezar a recibir todos los datos emitidos bajo el tópico input. Es muy importante establecer el QoS al subscribirse de 1, para evitar que el broker MQTT no descarte mensajes en caso de congestión y que haya más de uno encolado para ser emitido.

5.1.2 Eclipse Mosquitto

Siendo este un servicio instalable y ampliamente disponible en varias plataformas, esta tesina se limita a detallar ciertos parámetros de configuración requeridos para que la arquitectura funcione. Si bien se recomienda Eclipse Mosquitto por sus escasos requerimientos de hardware, se podrían utilizar RabbitMQ o EMQX, entre otros.

Puntualmente el broker de Mosquitto tiene una configuración extremadamente importante para poder tener mejores garantías a la hora de manejar un flujo repentino de mensajes.

max_queued_messages = 0

Este parámetro debe setearse en 0 en el archivo de configuración de Mosquitto para que el broker no descarte silenciosamente mensajes si la cantidad “en vuelo” (es decir, emitidos pero que su receptor no ha respondido todavía) correspondientes a suscripciones con QoS 1 o 2 excede el límite seteado en esta configuración. Durante las pruebas de rendimiento de este desarrollo sucedieron casos donde el broker descartó mensajes y fue solventado con esta configuración.

Más allá de eso se requiere que el broker esté disponible en la interfaz de red de localhost del dispositivo de borde y no es necesario que esté expuesto a redes internas. Esta suele ser la configuración por defecto luego de su instalación inicial, que al estar cerrada a conexiones externas, es más segura.

No se utilizaron sus características de manejo de usuarios y permisos sobre tópicos particulares, pero es posible en caso de querer agregar un nivel más de seguridad a la hora de manipularlo. En tal caso tanto el servicio de historizr-device como el servicio de MiNiFi, que ambos interactúan con el broker MQTT, tendrían que tener estas credenciales configuradas para poder funcionar normalmente.

5.1.3 Servicio historizr-device

Este es un servicio específicamente desarrollado para esta tesina, su código está disponible en el siguiente link: https://github.com/dustContributor/historizr/tree/main/historizr-device. Está implementado en Java y corre permanentemente en la computadora de propósito general o dispositivo de borde donde se quiere que comience la incorporación de los datos a la arquitectura. Se ilustra la arquitectura interna del servicio en la Figura 12.

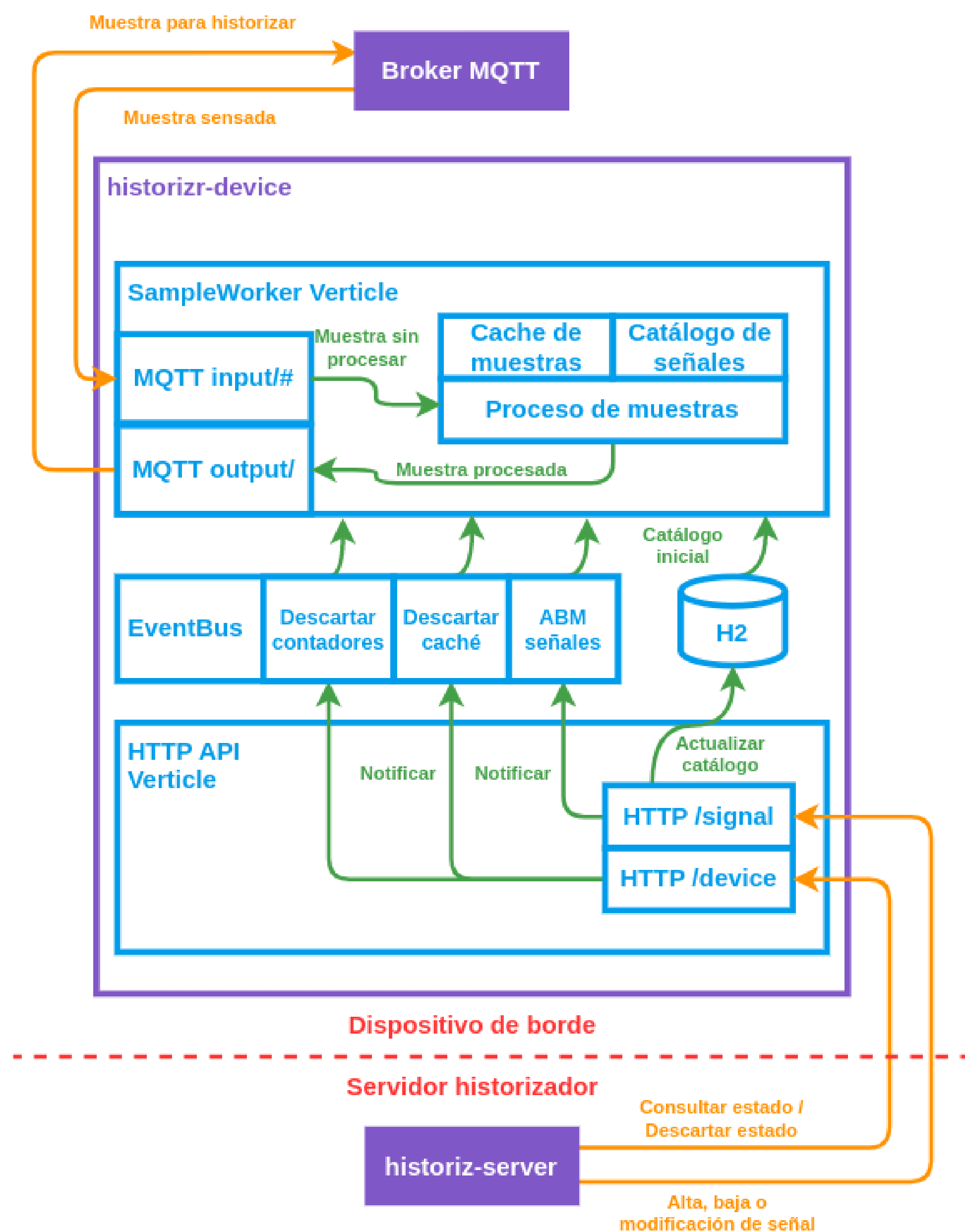

Figura 12. Arquitectura interna del servicio historizr-device.

El objetivo del servicio es poder consumir la muestra emitida hacia el broker MQTT mediante una suscripción a la máscara “input/#”, procesarla o descartarla según los criterios configurados para su señal una vez recibida, y emitir la muestra nuevamente al broker MQTT por el tópico de salida, para que sea capturada por MiNiFi y emitida hacia el servidor de historización. Además debe proveer una interfaz por HTTP que permita la configuración en vivo de las señales admitidas.

5.1.4 Procesamiento de muestras

El servicio posee un listado de señales permitidas y configuradas, cada una con el nombre del tópico completo al que están asociadas en el broker MQTT, por el cual se realiza la lectura de los datos entrantes. Si se recibe un mensaje fuera de este listado de tópicos/señales admitidas se ignora, en cambio si el mensaje entrante corresponde a una señal configurada en el dispositivo, se realizan una serie de procedimientos base que pueden determinar si el valor en cuestión se emite o no.

En base al tipo de dato configurado para la señal, se convierte el contenido del mensaje que posee formato textual, al tipo de dato correspondiente. Con los dos tipos de señales definidas, es decir, señales “completas” que deben especificar en un JSON su valor, timestamp y calidad, y señales “simples”, que solo indican su valor en el payload del mensaje MQTT, se procede de forma ligeramente distinta en cada caso. En el caso completo, si el parsing del payload falla, la muestra no se emite, ya que se entiende que el mensaje debería estar bien formado. En el caso de una señal simple, si falla el parsing se emite con un valor por defecto y en mala calidad, indicando que hubo un problema con el dato.

Para ahorrar recursos de red y procesamiento, una señal puede ser configurada para ser emitida sólo si cambia su valor con respecto al último recibido por el servicio de historizr-device, y además se le puede configurar una banda de tolerancia o “deadband” para esta detección, de tal forma que se admitan datos solo si exceden cierto porcentaje de cambio con respecto al último registrado.

Para realizar estas tareas es necesario que historizr-device mantenga un caché en memoria del último dato que va recibiendo por cada señal que tiene configurada. Este fue realizado aprovechando la amplia librería estándar de primitivas concurrentes que posee Java, ya que en la práctica la manipulación del caché es esencialmente concurrente: Puede ser modificado por la recepción de un dato nuevo, por el cambio de configuración de una señal existente, etc. Tareas que para el funcionamiento óptimo del servicio, son manejadas concurrentemente.

Si estos criterios para el manejo de los valores entrantes están configurados en la señal, se pueden ahorrar recursos ya que valores constantes o semi constantes, o quizás valores con “ruido” que no interesen mucho salvo que cambien drásticamente, pueden ser descartados y no emitidos por el servicio. Estos criterios se visualizan en el diagrama de flujo de la Figura 13.

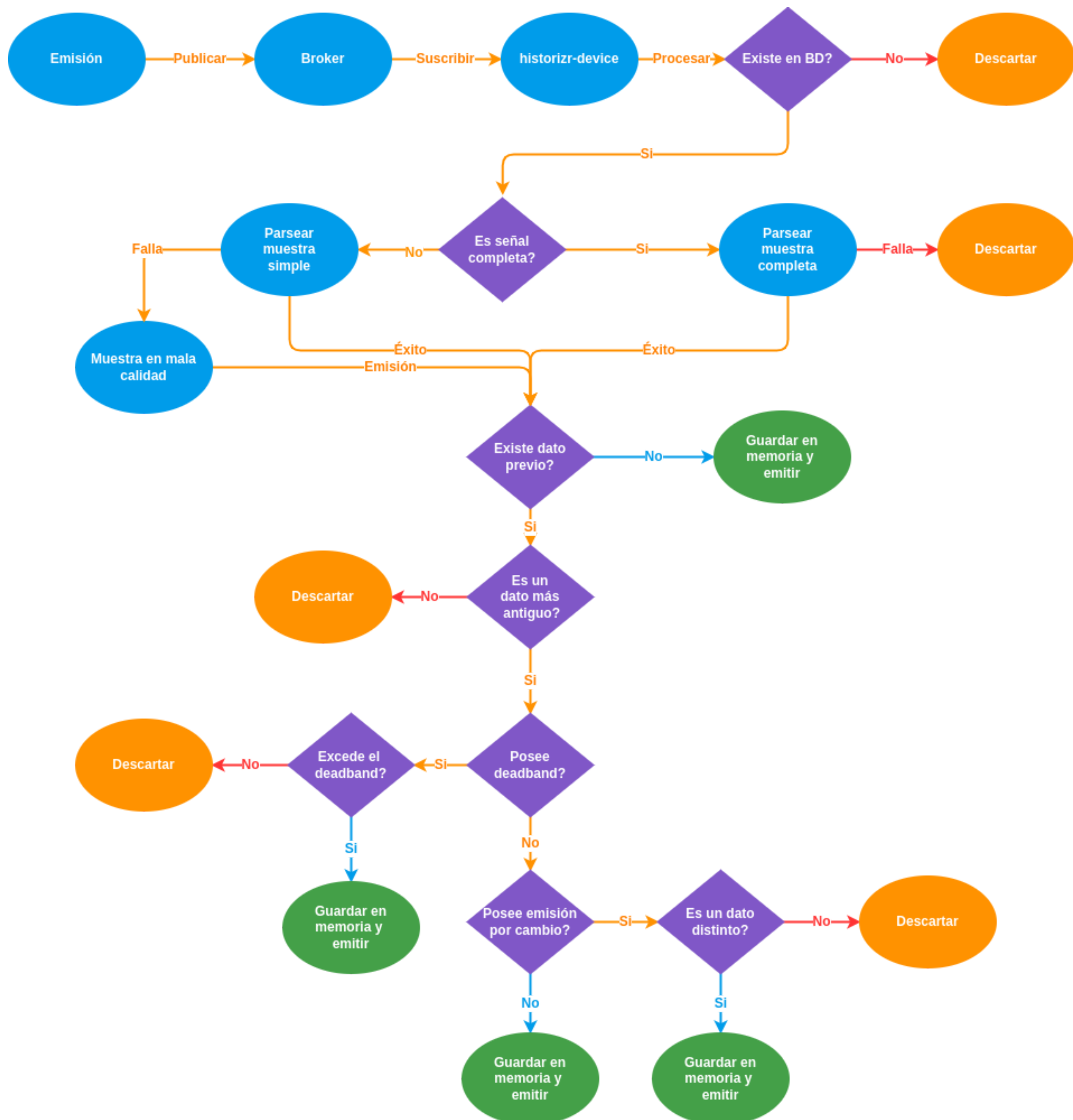

Figura 13. Flujo de procesamiento de muestras.

La configuración de emisión por cambios es una simple comparación por igualdad entre el dato entrante y el último registrado por el servicio. La configuración por banda de tolerancia o banda muerta es un porcentaje de cambio configurable con respecto al valor anterior registrado.

Una vez que el dato fue admitido ya sea porque nunca se recibió uno anterior en esta señal en particular, o porque pasó los criterios de emision por cambio o banda de tolerancia, se procede a registrar su último valor emitido en un caché en memoria, y se emite nuevamente al broker MQTT pero en un tópico particular de salida, con un formato JSON específico, y con un dato extra y fundamental, el identificador numérico de la señal, que facilita su

almacenamiento una vez que llegue al servidor de destino. Un ejemplo de este JSON se encuentra en la Figura 14, a continuación:

```
1  {
2    "id": 99,
3    "tstamp": "2022-11-04T03:00:00.001Z",
4    "quality": true,
5    "value": 7
6  }
```

Figura 14. JSON que se emite hacia el servidor.

Es importante en este punto tener en cuenta que la configuración de las señales en el dispositivo es un espejo de la configuración de las mismas en el servidor. Por lo tanto, los identificadores de señales son únicos por toda la arquitectura y conocidos entre ambos, el servidor y dispositivos de adquisición. Esto simplifica de forma particular la transmisión del dato hacia el servidor ya que no es necesario identificar la señal con su nombre completo, sino que basta su identificador numérico para almacenarla.

También es un detalle a tener en cuenta que el identificador sea numérico y entero, ya que su interpretación es inequívoca. Otro tipo de identificadores basados en nombres podrían dar lugar a conflictos por espacios, carácteres especiales, mayúsculas o minúsculas, entre otros.

5.1.5 Configuración de señales

El dispositivo posee una base de datos interna donde almacena las señales que comprende y su configuración, esta puede ser actualizada de varias formas. Se trata de una base de datos embebida, es decir, está pensada para sólo ser manipulada por un cliente en simultáneo, que en este caso sería el servicio historizr-device. A su vez, posee un esquema de almacenamiento sencillo, toda la información se encuentra en un solo archivo.

Si bien es posible editar esta base de datos manualmente, simplemente cargando el archivo en cualquier cliente que lo pueda leer, y manipulando las tablas dentro posteriormente reiniciando el servicio para que estos cambios se vean reflejados, la forma recomendada ofrecida por historizr-device para manipular su base de configuración de señales es a través de una API REST por HTTP.

5.1.6 Base de datos embebida

Normalmente los servicios de bases de datos tradicionales existen por sí mismos, es decir, son un proceso que es ejecutado independientemente, y tienen su propio ciclo de vida gestionado por el gestor de servicios persistentes del sistema operativo donde corre. En este esquema toda aplicación que quiera acceder se conecta a este servicio y realiza las tareas que necesite contra la base de datos. Además, admite concurrencia de accesos y dependiendo de la base de datos puede otorgar garantías de resiliencia, performance, entre otros.

Las bases de datos embebidas tienen su existencia vinculada a la aplicación que la utiliza, esto implica que su alcance es mucho más pequeño, posiblemente no admiten accesos concurrentes desde distintos procesos o desde interfaces de red, pero sí accesos concurrentes desde el proceso que la crea. A su vez, al tener un alcance limitado, pueden librarse de las restricciones o expectativas que se suelen tener sobre bases de datos convencionales, es decir que pueden no admitir un SQL estándar, o directamente no utilizar SQL para su manejo y operar conceptualmente como un diccionario persistido a disco.

La elección de una base de datos embebida surge de poder plantear un servicio que tenga pocas dependencias externas, de reducido uso de memoria, y en definitiva de necesidades limitadas con respecto al almacenamiento de sus catálogos necesarios para operar. Cabe destacar que se podría haber utilizado PostgreSQL ya que demostró tener un buen uso de recursos, a pesar del limitado hardware, pero se consideró no justificable tener que instalar un servicio extra a parte teniendo en cuenta que el servicio tiene necesidades muy acotadas en términos de concurrencia de accesos, volumen de tablas utilizadas, y características de SQL utilizadas. Hay que recordar que la intención del servicio es que pueda ser instalada en un dispositivo que tenga potencialmente hardware limitado.

5.1.7 H2

Esta base de datos embebida fue elegida por poseer un modo de compatibilidad con PostgreSQL, lo cual normaliza los identificadores utilizados y ciertas sentencias de SQL para que se asemejen a cómo funciona PostgreSQL. Si bien el dispositivo de borde no utiliza una base de datos PostgreSQL, es más cómodo para el desarrollo de los servicios involucrados si el acceso a base de datos es uniforme, sea en el servidor o en el dispositivo de borde.

Se puede habilitar el modo de compatibilidad con PostgreSQL desde la cadena de conexión, que puntualmente habilita el nombre a minúscula por defecto y ordena los nulos primero, de la siguiente forma:

jdbc:h2:./historizr.h2;MODE=PostgreSQL;DATABASE_TO_LOWER=TRUE;DEFAULT_NULL_ORDERING=HIGH

El rol de H2 en el dispositivo de borde es pequeño, solamente almacena los catálogos de señales admitidas y la configuración de cada una. Al ser una base de datos embebida, su existencia está atada al proceso de historizr-device, y sólo puede ser accedida por el mismo mientras está siendo ejecutado. Los datos de la base de datos se almacenan en un único archivo adyacente a la ubicación de los archivos del servicio historizr-device.

5.1.8 Esquema de base de datos

Este esquema responde a necesidades muy sencillas, de ahí desprende el hecho de que esté resuelto solamente en dos tablas, como se ve en la Figura 15. Se destaca que, como ambas tablas también están presentes en el servidor de historización con la misma definición de

columnas, se seleccionaron tipos de datos equivalentes entre el motor de bases de datos del dispositivo, H2, y el del servidor de historización, PostgreSQL.

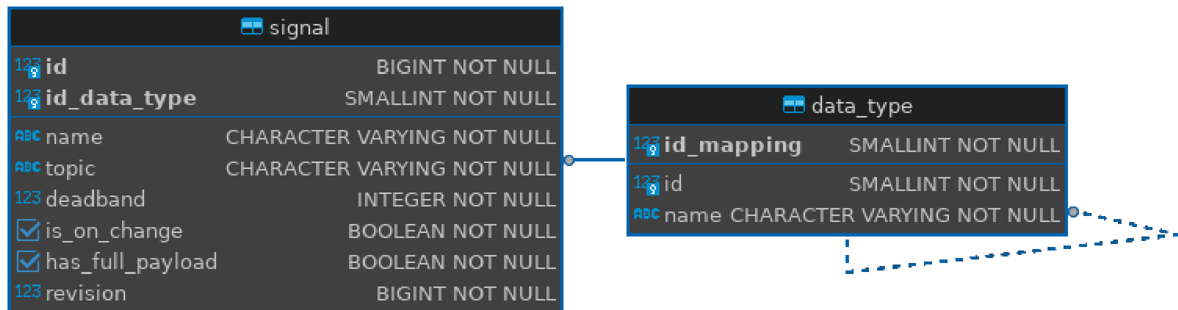

Figura 15. Esquema de base de datos en el dispositivo de borde.

5.1.8.1 Tabla data_type

Esta tabla cumple dos propósitos. El primero determina el catálogo de tipos de datos posibles que el sistema puede aceptar. Este es un catálogo fijo (figura 16) por lo tanto no se consideró ofrecer un endpoint para modificarlo. El segundo propósito es poder hacer un mapeo del tipo de dato configurado de la señal, al tipo de dato de almacenamiento.

	id	id_mapping	name
1	1	1	bool
2	2	9	u8
3	3	9	i8
4	4	9	u16
5	5	9	i16
6	6	9	u32
7	7	9	i32
8	8	9	u64
9	9	9	i64
10	10	11	f32
11	11	11	f64
12	12	12	str

Figura 16. Tipos de datos admitidos por el sistema.

Dependiendo de cómo se arme la tabla de muestras, se pueden reducir la cantidad de columnas posibles al guardar, por ejemplo, flotantes de 32 bits (f32) y de 64 bits (f64) en una misma columna de flotantes de 64 bits (f64) en el servidor historizador. Lo mismo puede suceder con enteros pequeños (i16, i32, etc) que pueden ser guardados en una única columna de 64 bits (i64), o se puede hacer una distinción individual y tener un destino con cada tipo de dato posible. Esta tabla posee una relación con sí misma ya que define esta conversión del tipo fuente original al tipo de dato destino en el servidor historizador, que realiza el servicio historizr-device al procesar las muestras entrantes.

Como se ve en la Figura 16, se tienen 4 tipos de destino los cuales van a tener su razonamiento explicado en el capítulo 5.2.2.4.

- Los enteros se guardan en una columna entera de 64 bits (i64 de nombre en tabla, int8 en PostgreSQL).
- Los flotantes se guardan en una columna flotante de 64 bits (f64 de nombre en tabla, float8 en PostgreSQL).
- Los booleanos se guardan en una columna booleana (bool de nombre en tabla, boolean en PostgreSQL).
- Las cadenas de texto se guardan en una columna de tipo texto (str de nombre en tabla, text en PostgreSQL).

5.1.8.2 Tabla signal

Esta tabla es más elaborada. Posee el catálogo de señales admitidas por el dispositivo de borde, es decir, cada dispositivo de borde va a tener su catálogo propio de señales. Se ve un ejemplo de este catálogo con algunos datos cargados en la Figura 17 a continuación:

	id	id_device	id_data_type	name	topic	deadband	is_on_change	has_full_payload	revision
1	91	5	9	hwmon0_temp1_crit	input/91	0	[v]	[ ]	2
2	92	5	9	hwmon0_temp1_input	input/92	0	[v]	[ ]	2
3	94	5	9	meminfo_mem_total	input/94	0	[v]	[ ]	2
4	95	5	9	meminfo_mem_free	input/95	0	[v]	[ ]	2
5	96	5	9	meminfo_mem_available	input/96	0	[ ]	[v]	1
6	97	5	9	meminfo_buffers	input/97	0	[ ]	[v]	1
7	98	5	9	meminfo_cached	input/98	100	[ ]	[v]	2
8	99	5	9	meminfo_swap_cached	input/99	100	[ ]	[v]	2
9	100	5	9	meminfo_active	input/100	100	[ ]	[v]	2
10	101	5	9	meminfo_inactive	input/101	0	[ ]	[v]	1

Figura 17. Ejemplo de señales con su configuración.

Los datos de esta tabla tienen que estar presentes tanto en el dispositivo puntual como en el catálogo unificado de señales en historizr-server. A continuación se listan sus definiciones:

id: Identificador numérico único generado por historizr-server al dar de alta la señal en el servidor de historización.
id_device: Identificador numérico único del dispositivo. Se encuentra en esta tabla por cuestión de mantener uniformidad con la tabla del servidor, sin embargo en este caso todos los datos presentes van a corresponder al mismo dispositivo.
name: Nombre único de señal para ese dispositivo.
topic: El tópico de entrada donde cual sea el proceso que ingresa datos al broker MQTT tiene que publicar las muestras de cada señal.
deadband: Valor expresado en fracciones de 1000 para evitar posibles problemas de redondeo. En el ejemplo se ven algunas señales con un deadband de 100, es decir, 100/1000 = 10%.
is_on_change: Flag que indica si la señal emite valores hacia el servidor sólamente si hubo un cambio con respecto al último valor en el caché de muestras de historizr-device dentro del dispositivo.

has_full_payload: Flag que indica si la señal tiene un cuerpo JSON completo con todas sus propiedades, o uno textual simplificado.
revision: Contador de conveniencia que se incrementa cada vez que se actualiza la señal. Es proporcionado por el historizr-server. Su propósito es tener una forma sencilla de evaluar si alguna señal difiere entre el catálogo maestro de señales del servidor de historización, y la almacenada en la base de datos del dispositivo de borde.

5.1.9 API HTTP

La API no es muy amplia, y debe poder ser manipulada por el servicio del lado del servidor de historización, historizr-server.

El endpoint fundamental es el de signal, por medio de este es posible consultar las señales configuradas en el dispositivo, dar de alta nuevas, y borrar o modificar existentes.

El proceso corre asincrónicamente con respecto a la captura de muestras, de tal forma que la parte del servicio que maneja la API corre independientemente de la parte del servicio que muestrea y procesa los mensajes del broker MQTT. Para poder comunicar uno con el otro, puntualmente, para que la API pueda modificar de cualquier operación realizada sobre el catálogo de señales y que el servicio pueda adaptar su procesamiento de las mismas de forma acorde, se utiliza el EventBus de Vert.x, el cual como indica su nombre, es un bus de eventos al cual otras partes del servicio se pueden suscribir y ser notificadas de esta manera.

En un caso concreto, si por ejemplo se quisiese configurar un deadband en una señal que no lo tiene, la operación llegaría a la API HTTP de historizr-device, se impactaría en su base de datos interna, y luego se emitirá al EventBus la notificación de lo sucedido adjuntando los datos de la señal cambiada. La parte del servicio que procesa los mensajes del broker MQTT captura este cambio, lo refleja en sus catálogos internos, y empieza a realizar esta lógica de deadband sobre la señal que previamente no la tenía configurada. De esta forma se logra un trabajo “online” sin retrasar la emisión de los datos de las otras señales que no fueron impactadas por este cambio.

A continuación se describen los endpoints del servicio historizr-device.

5.1.9.1 /signal

El endpoint de signal ofrece 4 métodos posibles por HTTP, se puede realizar peticiones al mismo por POST, GET, PUT y DELETE.

- GET. Devuelve el listado de señales que el dispositivo conoce en formato JSON.
- DELETE. Recibe como parámetro el identificador de una señal para borrar de la base embebida del dispositivo.
- PUT. Permite actualizar la configuración de una señal existente en la base del dispositivo.
- POST. Permite registrar una señal con su configuración en la base del dispositivo.

5.1.9.2 /device

El endpoint de device ofrece sólo un método posible para realizar peticiones, GET, con un ejemplo del resultado ilustrado en la Figura 18.

- GET. Devuelve una serie de datos pertenecientes al dispositivo y al estado interno del servicio historizr-device.
    - Cantidad de señales en el catálogo del dispositivo.
    - Cantidad de señales configuradas para emitir en caso de cambios.
    - Cantidad de señales con deadband.
    - Cantidad de muestras en el registro en memoria de las últimas muestras recibidas por señal.
    - Sumatoria de las revisiones de todas las señales del catálogo del dispositivo.
    - Contador de bytes y mensajes recibidos por MQTT. Este es el punto de entrada al servicio por el tópico input/#
    - Contador de bytes y mensajes emitidos por MQTT. Este es el punto de salida al del servicio hacia el broker y luego MiNiFi por el tópico output/#
    - Contador de mensajes procesados. Un mensaje procesado se considera un mensaje que fue recibido, su señal encontrada y parseado correctamente.
    - Contador de mensajes salteados. Un mensaje salteado es un mensaje que además de haber sido procesado, se descartó por el criterio configurado de la señal (falta de cambio, no excede el deadband, etc).
    - Contador de mensajes fallidos. Un mensaje fallido es un mensaje que no pudo ser publicado al broker MQTT al pasar por todo el flujo de tratado de mensajes e intentar ser emitido para que sea distribuido por el broker a MiNiFi.


```
1  {
2    "signalsRevision": 54,
3    "signalsWithFullPayload": 54,
4    "host": "rpi3",
5    "signalsTotal": 54,
6    "samplesInRegistry": 10,
7    "signalsWithDeadband": 0,
8    "signalsWithOnChange": 0,
9    "sampleStats": {
10     "receivedCount": 110010,
11     "receivedBytes": 8112124,
12     "processedCount": 110010,
13     "skippedCount": 0,
14     "publishedCount": 110010,
15     "failedCount": 0,
16     "publishedBytes": 9530240
17   }
18 }
```


Figura 18. Ejemplo del JSON retornado por el endpoint de /device.

5.1.9.3 /device/discardsamplestate

El endpoint de discardsamplestate ofrece sólo un método posible para realizar peticiones, POST.

- POST. Descarta el registro de muestras recibidas en memoria como si el servicio hubiera sido recién iniciado, de tal forma que las próximas muestras válidas que sean recibidas por el servicio van a ser emitidas inmediatamente, ya que no van a existir muestras previas para comparar cambios o deadband. Utilizado para pruebas fundamentalmente pero podría ser útil en el caso de que se quiera forzar una emisión de datos desde el dispositivo.

5.1.9.4 /device/discardsamplestats

El endpoint de discardsamplestats ofrece sólo un método posible para realizar peticiones, POST.

- POST. Descarta las estadísticas y contadores registrados en el servicio. Fundamentalmente utilizado para pruebas o descartar estadísticas de pruebas ya realizadas sin reiniciar el servicio.

5.1.10 MiNiFi

Una vez la muestra fue procesada y validada por historizr-device, siendo emitida al broker MQTT en el mismo dispositivo nuevamente, llega a una instancia del servicio MiNiFi en el mismo dispositivo por medio de la suscripción a la máscara “output/#”, para luego ser enviada a una instancia de NiFi principal en el servidor de historización.

MiNiFi cumple un rol fundamental en toda la cadena de emisión del dato para realizar buffering de mensajes no enviados en casos de falla de comunicación, y para poder enviar mensajes encolados que no hayan podido ser transmitidos por una caída del dispositivo entero (ej, corte de energía) tras la restauración del funcionamiento normal del dispositivo.

El buffering de mensajes no enviados no es trivial de resolver, y es de mucha importancia, por eso se optó por utilizar una herramienta reconocida por su robustez y de larga trayectoria para realizarlo. En la Figura 19 a continuación se visualiza la interfaz web que posee NiFi con el flujo de MiNiFi descrito y configurado.

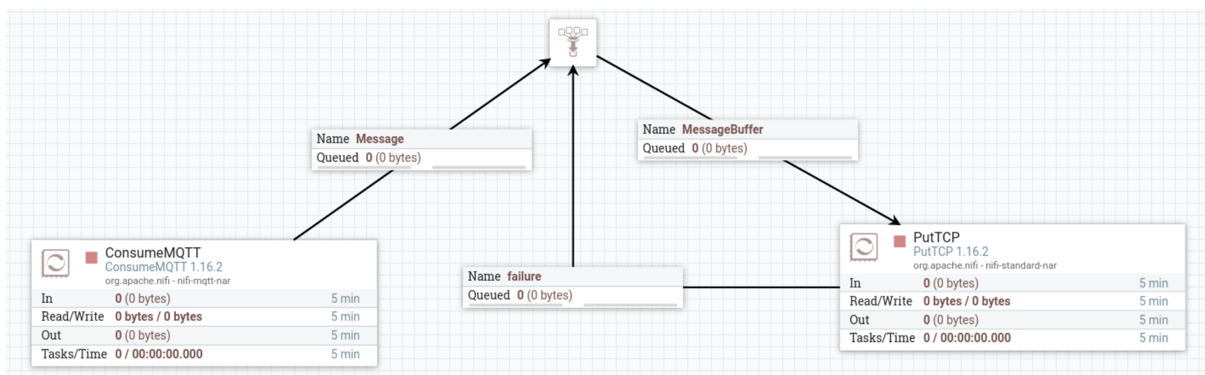

Figura 19. Flujo de transmisión de mensajes hacia la instancia de NiFi principal.

Para realizar estas tareas es necesario configurar MiNiFi con un flujo de procesamiento sencillo pero con piezas clave:

1. Una instancia del nodo procesador de MQTT (nodo “ConsumeMQTT” en el flujo) suscrito a la máscara “output/#” para capturar todas las muestras procesadas por historizr-device.
2. Una cola de mensajes preferiblemente configurada para poseer un amplio espacio de almacenamiento y de cantidad de mensajes (conector “MessageBuffer” en el flujo), en caso de que la falla de comunicación sea prolongada, para realizar buffering de muestras. El buffering es configurable por tamaño y cantidad de mensajes. Puntualmente, permite aprovechar al máximo las características del dispositivo de campo, dándole una amplia autonomía si el dispositivo está equipado con el almacenamiento suficiente.
3. Un nodo procesador de comunicación que emite las muestras hacia el servidor de historización principal (nodo “PutTCP” en el flujo), puntualmente hacia la instancia de NiFi que se encuentra en el mismo. MiNiFi posee varios métodos para transmitir estos datos que se van a analizar más adelante. En caso de falla de comunicación, este procesador devuelve los mensajes a la cola para ser re-enviados posteriormente (conector “failure” en el flujo). En caso de éxito, los mensajes se “consumen” y se descartan, habiendo terminado su ciclo en el dispositivo de borde.

5.2 Servidor de historización

El servidor de historización es compuesto por tres servicios principales: Una instancia de NiFi, que captura los datos emitidos desde el dispositivo de borde y los inserta en la base de datos, una instancia de PostgreSQL con el plugin TimescaleDB, tanto para almacenar los datos de los sensores a largo plazo como para almacenar la configuración de las señales y dispositivos gestionados por la arquitectura, y una instancia del servicio historizr-server desarrollado para esta tesina, que sirve para manipular la configuración de dispositivos y señales mencionada.

5.2.1 NiFi

NiFi se ubica como punto clave en la arquitectura para capturar todos los datos emitidos por los dispositivos. Recordando de donde provienen estos datos, hay que tener en cuenta que los mensajes ya fueron manipulados y confeccionados por el servicio historizr-device en el dispositivo de borde, por lo tanto el trabajo que NiFi tiene que realizar sobre estos datos es mínimo para su almacenamiento. El objetivo de esto es aprovechar al máximo el hecho de que se dispone de dispositivos de borde de propósito general, para maximizar la eficiencia al momento de almacenar los datos en la base de datos.

El flujo de procesamiento en NiFi (figura 20) en consecuencia es sencillo: ya es conocido en este punto de la arquitectura que los datos están validados con respecto al catálogo de señales existente y sus tipos de datos en concreto, así que basta con transformar los JSONs entrantes

en INSERTs de SQL, y ejecutarlos contra la base de datos que se encuentra en el mismo servidor.

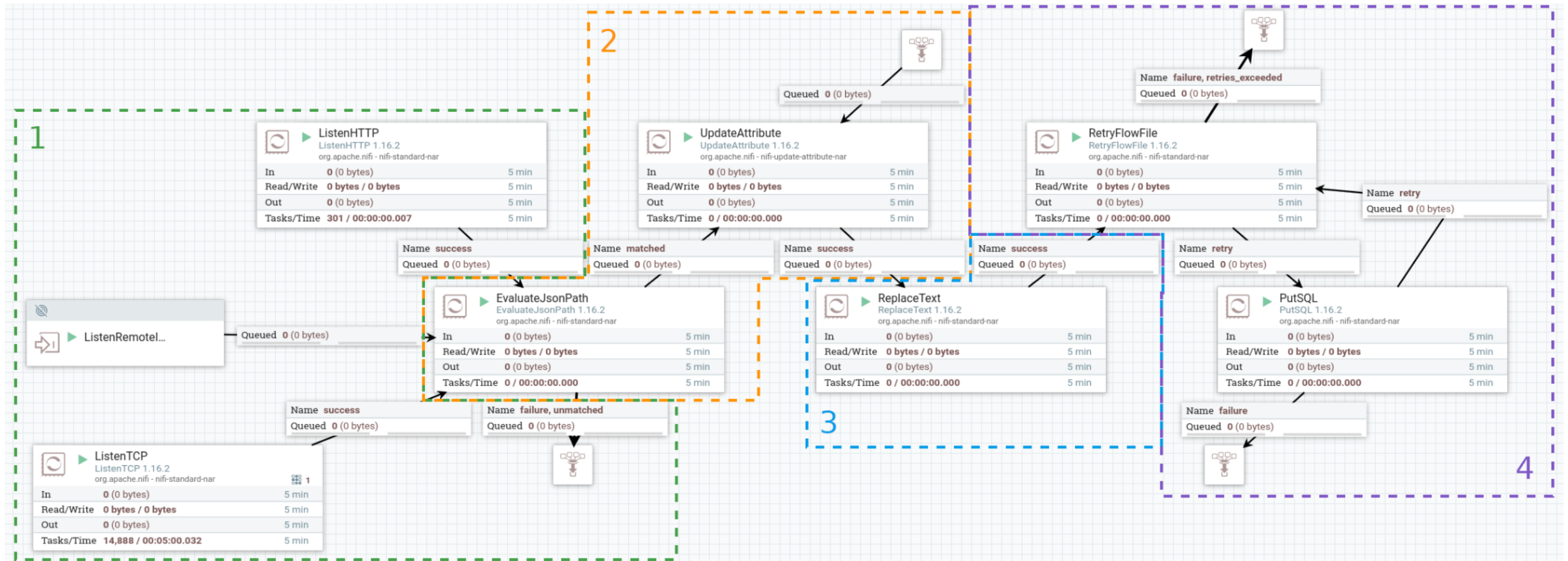

Figura 20. Flujo de historización dentro de la instancia principal de NiFi.

Si bien se puede considerar como sencillo, el flujo de NiFi en el servidor historizador es más elaborado que el de MiNiFi que se encuentra en el dispositivo de borde. Es capaz de capturar los mensajes entrantes de todos los dispositivos de borde presentes, convertirlos de JSON a INSERTs de SQL, y ejecutarlos en la base de datos de historización. Puntualmente se definen 4 etapas del flujo:

1. Los mensajes son recibidos por los nodos procesadores. En este caso se ven 3 métodos de recepción de mensajes distintos: Por HTTP (ListenHTTP), por TCP (ListenTCP) y por protocolo de puerto remoto de NiFi (ListenRemoteInput). Con uno solo de estos métodos configurados tanto en los dispositivos de borde como en el servidor de historización es suficiente, pero en la Figura 20 se ve cómo se configuraron los 3 métodos en simultáneo para realizar las pruebas en el capítulo 5.3.7.
2. En esta etapa los JSONs de los mensajes son descompuestos en sus atributos básicos para ser insertados en la tabla de muestras. Esto implica el timestamp, el id de la señal, el valor de la señal, para el cual la columna de destino dependerá del tipo de dato de la misma, y también se asignan al mensaje meta atributos que sirven para indicarle el tipo de dato correspondiente para cada columna a NiFi en el momento de ejecutar el INSERT. Todos estos campos se colocan como atributos del mensaje o FlowFile de NiFi.
3. El nodo de ReplaceText cumple una tarea sencilla que es reemplazar el contenido del FlowFile entrante por el INSERT de SQL a realizar (figura 21). Como se dispone del ID de la señal, se puede realizar un JOIN con el catálogo para colocar el valor en la columna correspondiente. Como se ve la Figura 21, para simplificar el manejo de tipo de datos desde el flujo de NiFi lo que se hace es emitir el INSERT con el parámetro de valor como un string, y hacer la conversión en el mismo INSERT de SQL, que se ve en las líneas 19, 21 y 22. Esto es más sencillo que preparar el flujo de procesamiento para organizar los valores entrantes en sus columnas correspondientes.
4. La última etapa está compuesta por dos procesadores: Uno que expira los mensajes que tienen un campo de reintentos que excede cierta cantidad, y otro, PutSQL, que ejecuta los FlowFiles con SQL en su contenido contra la base de datos. Si el mensaje falla, vuelve por el flujo de reintentos, incrementando su contador de reintentos, hasta que se exceda el límite configurado. El tratamiento de los mensajes que exceden el límite de reintentos puede ir desde un descarte por tiempo (por ejemplo, descartar mensajes que son más antiguos de 10 días) o se podrían insertar en otra tabla de la base de datos para su posterior análisis.

```sql
insert into historizr.sample(
  id,
  tstamp,
  quality,
  value_i64,
  value_f64,
  value_bool
)
with tmp as materialized (select
  ?::int8 as id,
  ?::timestamptz as t,
  ?::bool as q,
  ? as v)
select
  v.id,
  v.t,
  v.q,
  -- i64
  case d.id_mapping when 9 then cast(v.v as int8) else null end,
  -- f64
  case d.id_mapping when 11 then cast(v.v as float8) else null end,
  -- bool
  case d.id_mapping when 1 then cast(v.v as bool) else null end
from tmp v
join signal s on s.id = v.id
join data_type d on d.id = s.id_data_type

```

Figura 21. INSERT de la muestra sobre la base.

5.2.1.1 PutSQL

El procesador clave para el rendimiento de todo el flujo es el de inserción de los mensajes entrantes, PutSQL. Posee un tamaño de batch configurable, que es posible regular en conjunto con un intervalo de ejecución si se desea. Existen algunas estrategias que se pueden tomar para esta configuración:

Se puede aumentar el throughput del procesador aumentando el tamaño de batch, y aumentando los intervalos de ejecución del procesador a 1, 2, 3 o más segundos para dejarlo acumular mensajes. Este tipo de configuración por periodo en un lado deja que el procesador ejecute más INSERTs juntos en un mismo batch aumentando el throughput y reduciendo el uso de CPU tanto en NiFi como en la base de datos, pero aumenta la latencia de inserción ya que los mensajes van a tener que esperar en promedio la mitad del intervalo para ser insertados.

Si lo que se quiere hacer es reducir la latencia de inserción al máximo posible, se puede configurar para que el nodo sea invocado apenas haya un mensaje para procesar, en el momento en el cual procede a tomar todos los mensajes disponibles en la cola hasta el tamaño de batch, y ejecutarlos en base de datos. Esto va a hacer que los mensajes se inserten inmediatamente, pero va a dejar menos margen al nodo para que acumule mensajes en el

batch a ejecutar, lo que reduce el throughput de mensajes insertados ya que al ejecutarse el nodo más frecuentemente aumenta el potencial uso de CPU por parte de NiFi y la base de datos.

Ambas configuraciones son factibles, pero a su vez dependen de otros factores: Hardware disponible, tolerancia del sistema en general a historizar muestras de forma más tardía, flujo de mensajes entrantes, etc. Lo importante a destacar es que su configuración es muy sencilla de hacer desde la interfaz web de NiFi así que no presenta obstáculo alguno para decidir entre una estrategia u otra.

5.2.2 Esquema de base de datos

El esquema final resulta bastante sencillo, con pocas tablas, como se ve en la Figura 22.

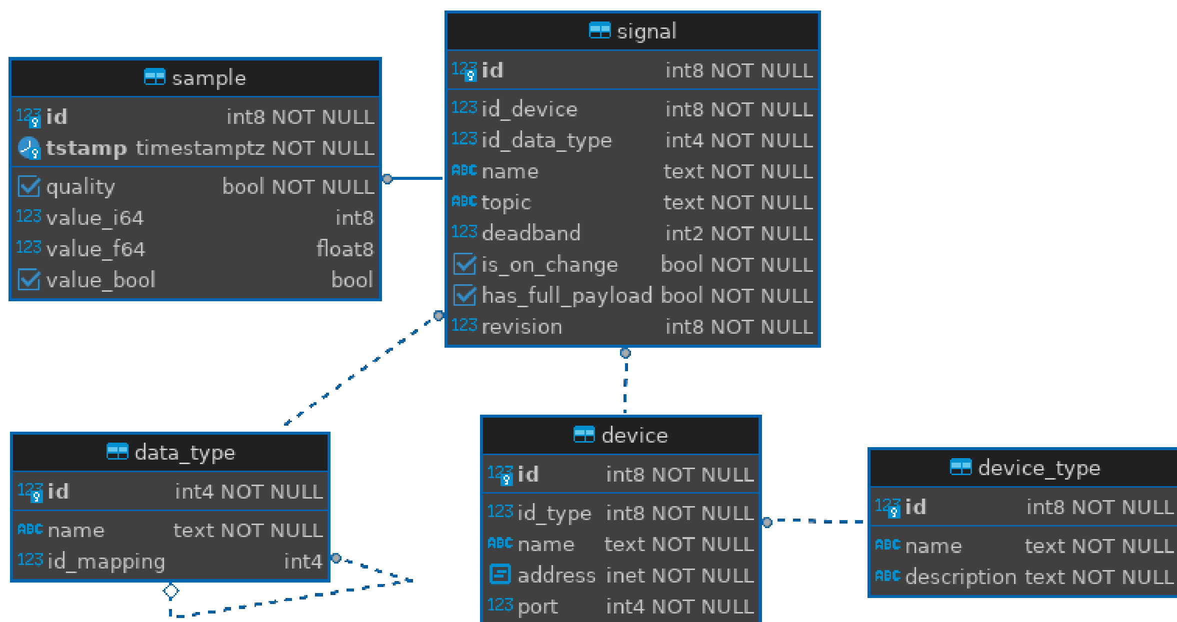

Figura 22. Esquema de base de datos en el servidor de historización.

Las dos tablas catálogo principales son la de signal y la de device. Además se tiene el catálogo de data_type con el mismo contenido que en el dispositivo de borde (tipos de datos y sus respectivos mapeos), y un catálogo de device_type que sirve de manera ilustrativa para asignar "tipos" de dispositivos si se quisiera, que no influyen en su funcionamiento.

5.2.2.1 Tabla device

La tabla device representa el catálogo de dispositivos presentes en el sistema. Está compuesta por un identificador único numérico, un nombre, un tipo asociado y fundamentalmente una IP y puerto para comunicarse con la API REST del servicio historizr-device que tiene que estar funcionando en el dispositivo de borde.

5.2.2.2 Tabla signal

La tabla signal representa el catálogo de señales medidas en el sistema entero, a través de todos los dispositivos de borde. Es una tabla que no debe conocer muchos detalles, cada señal tiene un nombre, un identificador numérico único, que tipo de dato maneja y a qué dispositivo pertenece. Refleja la configuración de cada señal en el dispositivo de borde, por lo que aplican los mismos campos descritos en el capítulo 5.1.8.2.

5.2.2.3 Tabla sample

Se puede decir que la tabla sample es la tabla principal de la arquitectura del lado del servidor. Esta tabla concentra los datos emitidos por todos los dispositivos. Se entiende que si se quiere almacenar historia de estos datos a largo plazo, esta tabla es fundamental para que su almacenamiento sea compacto y las consultas realizadas sobre la misma sean eficientes.

El diseño actual deriva de una serie de compromisos:

- Se desea mantener la integridad del dato como fue emitido.
  - Esto implica que los tipos de datos de las columnas donde se guardan estas mediciones son cruciales, tienen que ser lo suficientemente amplios para guardar la información entrante sin mayores inconvenientes.
- Se desea mantener la estructura compacta en disco.
  - Esto implica que si bien se quiere preservar la integridad del dato, tampoco se puede usar cualquier tipo de almacenamiento de longitud arbitraria o representación ineficiente que pueda hacer excesivo el costo en espacio de cada dato guardado a largo plazo.
- Se desea que la estructura de las consultas hechas sobre la tabla no sea compleja.
  - Esto implica tener en cuenta que en definitiva la información no es útil si no se puede consultar, así que un esquema demasiado complejo, por más eficiente que sea, no lograría el objetivo.

Si bien estos objetivos pudieran parecer difíciles de lograr a simple vista, se presenta un esquema que los puede lograr de manera razonable.

5.2.2.4 Integridad del dato

Lograr la integridad del dato se realiza de forma directa teniendo en cuenta las limitaciones del motor de base de datos. Esto implica las longitudes de cada tipo de dato utilizado y sus restricciones.

Se van a soportar 3 tipos de datos: Números enteros con signo, números flotantes con signo y booleanos. Si bien el sistema se podría extender a cadenas de carácteres, no se va a contemplar en esta tesina, aunque se entiende que el tipo text de PostgreSQL sería el indicado para esto.

- Para números enteros PostgreSQL ofrece varios tipos nativos, int2, int4 e int8, los 3 son tipos con signo con distintas longitudes fijas (2, 4 y 8 bytes respectivamente), lo

cual condiciona el desarrollo en el sentido de que los tipos no signados de 64 bits no están directamente soportados.

- Para números flotantes se tiene float4 y float8, quizás más conocidos como float y double, o flotantes de precisión simple y precisión doble, 32 y 64 bits respectivamente. Estos responden al estándar de manejo de flotantes IEEE 754 (IEEE, 2008) que es ubicuo para el manejo de este tipo de datos, por lo que no hay limitantes directas para almacenarlos.
- Para booleanos se tiene un solo tipo boolean, que almacena dos valores: Verdadero o falso.

Tomando un acercamiento directo, una tabla contemplando todos estos tipos de datos se puede elaborar con la sentencia SQL de la Figura 23.

```
1   create table sample(
2     id int8 not null,
3     tstamp timestamptz not null,
4     value_i16 int2,
5     value_i32 int4,
6     value_i64 int8,
7     value_f32 float4,
8     value_f64 float8,
9     value_bool boolean,
10    primary key (id, tstamp)
11  )
```

Figura 23. Sentencia CREATE de SQL para la tabla sample.

Esta tabla posee un identificador para la señal y un timestamp para el valor, ambos obligatorios, y luego una serie de columnas para cada tipo de dato, que pueden estar ocupadas o no según el tipo de dato que se esté registrando.

Esta tabla se puede simplificar de dos formas: Se sabe que técnicamente en la columna value_i64 se podría almacenar cualquier valor de la columna value_i32 o value_i16, y que en la columna value_f64 se podría almacenar cualquier valor de la columna value_f32.

Se concluye con una estructura simplificada de la siguiente forma, ilustrada en la Figura 24:

```
1   create table sample (
2     id int8 not null,
3     tstamp timestamptz not null,
4     value_i64 int8,
5     value_f64 float8,
6     value_bool boolean,
7     primary key (id, tstamp)
8   )
```

Figura 24. Sentencia CREATE de SQL para la tabla sample simplificada.

En esta representación claramente se puede almacenar datos enteros, flotantes y booleanos, de la manera más apropiada para cada uno dadas las capacidades de la base de datos. De este punto se destacan dos aspectos: Si bien es posible almacenar enteros de gran amplitud en las columnas de datos flotantes, se puede perder precisión en valores altos, lo que implica una pérdida de la integridad del dato. Por lo tanto, preventivamente, se hace la diferenciación entre enteros y flotantes explícitamente almacenados en columnas con tipos distintos. Siguiendo esta idea también se podría almacenar en las columnas enteras los valores booleanos, estableciendo algún criterio como 0 = false y 1 = true, pero esto agrega lógica en el momento de interpretar los valores, y esta lógica puede dar lugar a errores de aplicación ya que el criterio de qué valor define a una muestra verdadera o falsa tiene que ser conocido por todos los lectores posibles del dato inequívocamente. Por lo tanto, también preventivamente se hace la diferenciación de datos booleanos en su columna propia con tipo booleano.

5.2.2.5 Almacenamiento compacto

Partiendo de estas dos propuestas de tabla sample, se tiene que determinar cuales son las desventajas de simplificar la tabla de esta forma, recordando que en el esquema simplificado potencialmente se están guardando enteros de 4 bytes en una columna de 8 bytes, y flotantes de 4 bytes en columnas de 8 bytes.

El set de datos a utilizar es una serie de métricas del sistema donde se monta el historizr-device, puntualmente valores de distintos tipos de memoria que tiene internamente el kernel de linux, frecuencias del CPU, temperatura, entre otras. Todas estas métricas fueron capturadas con el servicio historizr-capture por medio de rutas estándar del sistema. Éstas en sí no son relevantes, lo importante es que sean señales con un cierto grado de continuidad que representen cambios reales en un sistema.

Se adquirieron 10 horas de datos emitidos aproximadamente cada 1 segundo, para 46 métricas distintas, de tipo entero, por lo tanto se realiza una transformación de las mismas para re-interpretarlas como booleanas en el caso de la tabla con datos booleanos. Se amplió la cantidad de información guardada a 24h repitiendo las métricas recibidas de forma invertida para mantener la continuidad de las señales, de esa forma se logró un set de datos de aproximadamente 4 millones de muestras. Los resultados se representan en el gráfico de la Figura 25.

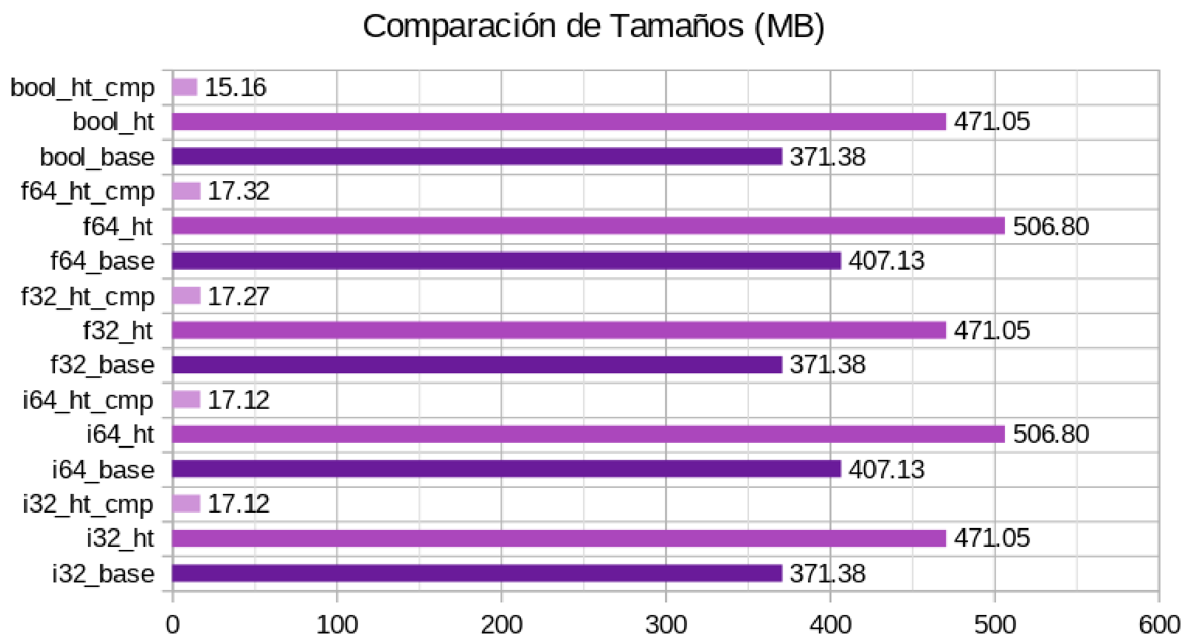

Figura 25. Comparativa entre almacenamiento comprimido, en forma de hypertable, y en tabla tradicional.

Los sufijos de cada entrada son los siguientes:

- "_ht_cmp". Hypertable comprimida de TimescaleDB.
- "_ht". Hypertable sin comprimir de TimescaleDB.
- "_base". Tabla tradicional de PostgreSQL.

Se destaca que el ancho de cada tipo individual (4 y 8 bytes) queda claramente representado, dado que tanto en sus representaciones como hypertable sin comprimir como en tabla tradicional, los tamaños ocupados por los datos en formato flotante y entero son iguales, comparando entre pares de tipo del mismo ancho (int4 y float4, e int8 y float8). Por otro lado se nota una ligera diferencia de tamaño una vez comprimidas las tablas.

Lo que se busca en este punto es mejorar el almacenamiento de los datos comprimidos. Se entiende que el hypertable con compresión habilitada solamente va a tener la parte temporalmente más reciente descomprimida, por lo que el efecto más grande en el almacenamiento a largo plazo va a ser de los rangos de tiempo comprimidos.

De estos resultados se pueden sacar algunas conclusiones:

i32 e i64

La información entera ocupa exactamente lo mismo una vez comprimida, tanto en los casos de 4 bytes como de 8 bytes, por lo tanto se decide que la simplificación del guardado de datos enteros puede realizarse, utilizando una sola columna de tipo int8.

f32 y f64

La información flotante ocupa casi lo mismo una vez comprimida, tanto en los casos de 4 bytes como de 8 bytes, por lo tanto también se decide que la simplificación del guardado de datos flotantes puede realizarse, utilizando una sola columna de tipo float8.

bool

La información booleana una vez comprimida ocupa menos espacio que el resto de los tipos, entre el 12% y el 14% menos según el tipo con el que se compara. Además, tanto su representación como tabla normal como hypertable también es más compacta, así que termina siendo más compacto en todos los casos. Se decide también mantener esta columna específica para los tipo booleanos.

5.2.2.6 Consultas de ejemplo

A manera de referencia, se pueden ver consultas de ejemplo de cómo se pueden calcular estadísticas sobre los datos de la tabla sample fácilmente.

Aprovechando las funciones básicas de agregación de SQL, se puede computar promedios, mínimos y máximos, como se ve en la Figura 26:

```
1  select avg(value_f64), max(value_f64), min(value_f64)
2  from sample
3  join signal using (id)
4  where name = 'hwmon0_temp1_input'
5    and tstamp >= '2022-11-04 04:00:00'
6    and tstamp < '2022-11-04 08:00:00'
```

Figura 26. Consulta de estadísticas para una señal.

Esta consulta computa las estadísticas para una señal en particular.

Siendo un esquema relacional convencional, también es posible asociar con las tablas de catálogos para realizar consultas más elaboradas, por ejemplo, computando las mismas estadísticas para la temperatura de todos los dispositivos presentes en la arquitectura, como se ve en la Figura 27:

```
1  select d.id, avg(v.value_f64), max(v.value_f64), min(v.value_f64)
2  from sample v
3  join signal s on v.id = s.id
4  join device d on s.id_device = d.id
5  where name like '%temp%'
6    and tstamp >= '2022-11-04 04:00:00'
7    and tstamp < '2022-11-04 08:00:00'
8  group by d.id
9  order by 2 asc
```

Figura 27. Consulta de estadísticas por dispositivo.

Por si no es aparente, se destaca que los beneficios son muy amplios. Estos ejemplos son sencillos pero se pueden aplicar todo tipo de filtros y asociaciones con otros metadatos que se pueden tener presentes en la base de datos, directamente desde el SQL ejecutado contra la base, sin tener que cruzar datos de distintas fuentes. Los dispositivos podrían tener información adicional como posición geográfica, o estar agrupados por conceptos más abarcativos como instalaciones o sitios, todos potencialmente disponibles en el esquema relacional. Esto no es posible o se hace muy complejo si se utiliza una solución dividida, es decir, donde la información histórica se encuentra en una base de datos de propósito específico y los catálogos de dispositivos en una base de datos convencional.

5.2.3 Servicio historizr-server

Este es un servicio desarrollado para esta tesina, implementado en Java, su arquitectura interna queda representada en la Figura 28, y su código está disponible en el siguiente link: https://github.com/dustContributor/historizr/tree/main/historizr-server. Cumple un propósito más sencillo que su contraparte en el dispositivo de borde, el servicio historizr-device. Sus responsabilidades son:

- Permitir gestionar los dispositivos y señales en la arquitectura:
  - Para los dispositivos basta con actualizar el catálogo principal.
  - Para las señales el proceso es más elaborado ya que, además del catálogo en el servidor, requiere realizar una actualización sobre el dispositivo de borde por medio de su API HTTP, para indicarle si la señal fue borrada, modificada o dada de alta. De esta forma el servicio historizr-device puede actualizar su estado interno y procesar las muestras entrantes debidamente.
- Permitir revisar el estado de los dispositivos en la arquitectura:
  - Por medio de los endpoints de contadores y estadísticas, es posible solicitarle a hsitorizr-server que consulte y reporte el estado de alguno de los dispositivos dados de alta.

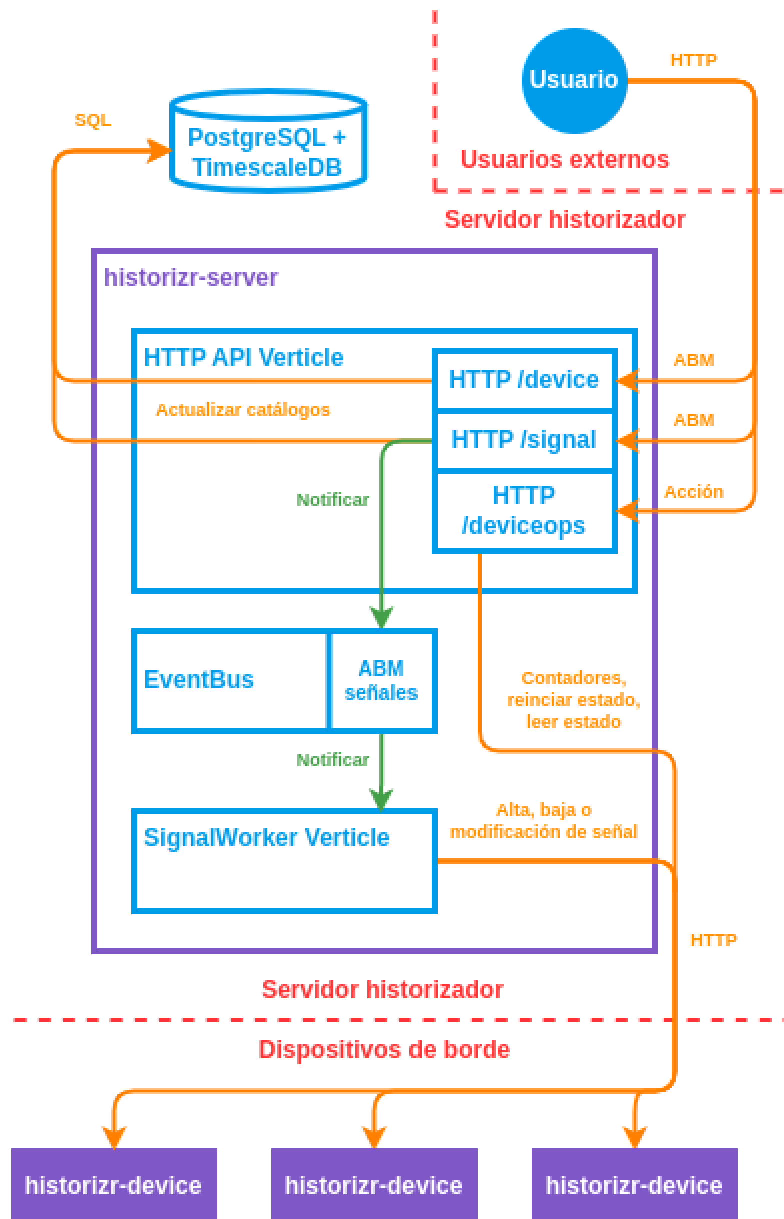

Figura 28. Arquitectura interna del servicio historizr-server.

5.3 Rendimiento en la arquitectura

Para la arquitectura es importante no solo detallar las capacidades de almacenamiento esperadas con el esquema establecido, sino también tener en cuenta la capacidad de procesamiento de mensajes que puede tener.

La capacidad de emisión de mensajes condiciona todo lo que sucede desde ese punto en adelante en la arquitectura. Si la capacidad de emisión es baja, la resolución de las mediciones se puede ver afectada, y posteriormente una posible analítica sobre las mismas también, así que es importante establecer que la arquitectura es capaz de otorgar buenas prestaciones.

Se entiende que quizás la parte más flexible de la arquitectura es la del servidor de almacenamiento, donde reside historizr-server. Hoy en día el hardware que se puede adquirir es muy amplio y se puede escalar verticalmente de formas que hace algunos años no hubiera sido posible (con CPUs de docenas de núcleos, discos de estado sólido de gigabytes de ancho de banda, etc). Sin embargo, la parte más sensible es el dispositivo de adquisición donde reside el historizr-device. Estos dispositivos tienden a ser los eslabones más “débiles” de la cadena en cuestión de características, por cuestiones de cantidad y costos, invertir en hardware nuevo en un solo servidor es factible, invertir en potencialmente cientos de estos dispositivos de borde de la misma forma no es económicamente escalable.

Para establecer un punto de referencia de las capacidades de la arquitectura planteada, se necesitan una serie de componentes:

- Un servicio que capture datos y los emita hacia el broker MQTT del dispositivo de borde, para que sean capturados por la arquitectura e historizados en base al esquema planteado.
- Hardware de referencia para confeccionar un dispositivo de borde modelo.
- Hardware de referencia para confeccionar un servidor historizador modelo.

5.3.1 Servicio historizr-capture

En la arquitectura se establece que el punto de entrada de los datos es el protocolo MQTT, y para demostrar cómo sería esto posible, se elaboró un script de ejemplo.

El servicio historizr-capture es un script hecho en JavaScript utilizando el runtime de Deno, que se consideró apropiado por su fácil despliegue, que es un solo archivo ejecutable por el medio del cual se pueden ejecutar scripts. Su código fuente está disponible en el siguiente link: https://github.com/dustContributor/historizr/tree/main/historizr-capture. El objetivo de este script es poder emitir periódicamente métricas del sistema donde esté corriendo.

Linux como sistema operativo ofrece una serie de rutas en su filesystem que poseen archivos que pueden ser leídos como texto plano, a su vez estos archivos poseen información en vivo y útil del sistema, sus dispositivos, características generales, etc. En particular en historizr-capture se vale de los archivos dentro de los directorios de /sys/class/hwmon (para leer información de temperatura del CPU de la Raspberry Pi) y de /proc/meminfo, para leer varias métricas de uso de memoria del sistema operativo. Son actualizados por el sistema operativo automáticamente de forma constante y pueden ser leídos fácilmente, lo cual se realiza en este script de ejemplo de forma periódica.

El servicio historizr-capture, al leer varias sub rutas en estas dos carpetas principales, las confecciona en señales individuales, las cuales son emitidas a tópicos del broker MQTT referenciando el nombre de cada una, y emitiendo su valor como contenido del mensaje. Esto se realiza como tarea periódica, y con este bucle de lectura se obtiene un flujo de mensajes constantes hacia el broker con ciertas métricas útiles del sistema. Al ser publicadas, entran en

el flujo de mensajes de la arquitectura, son emitidas al servicio de historizr-device que está suscrito al tópico genérico de entrada de datos, y son procesadas normalmente.

La intención de este ejemplo es demostrar como se puede reducir el trabajo de captura de datos a una expresión mínima, un script de pequeño alcance que lo único que tiene que comprender es el dato que está muestreando y cómo emitir por MQTT. Sólo con ésto ya se aprovecha el resto de la arquitectura de forma transparente ya que todo el procesado e historización de la muestra es estándar y común a todas las señales a partir de este punto en el flujo de datos.

5.3.2 Hardware para el dispositivo de borde

Las fuentes de datos telemedidas pueden encontrarse en muchos lugares con particularidades propias, desde el exterior en algún lugar físicamente alejado, hasta simplemente en una red local dentro de una empresa o fábrica, con conexión estable de electricidad y red.

Esta tesina se ideó abarcando la posibilidad de un dispositivo de borde que capture datos directamente. Este hardware debería ser de propósito general ya que se utilizaron tecnologías que dependen de una instalación de un sistema operativo convencional, y con recursos suficientes para poder correr los servicios necesarios de captura y emisión de datos, no así el de almacenamiento que debería correr en un servidor más amplio en capacidad.

5.3.2.1 Raspberry Pi 3B

Por disponibilidad se eligió la Raspberry Pi 3B. Es una mini computadora de propósito general, diseñada por la Raspberry Pi Foundation, del Reino Unido, y lanzada en 2016[18].

Ésta placa se distribuye sin carcasa, requiere de una fuente de alimentación externa de 5v y una tarjeta micro SD para almacenar una instalación de un sistema operativo compatible. Se caracteriza por sus dimensiones pequeñas[19] (85.60 mm × 56.5 mm, Figura 29), excelente conectividad (ethernet, bluetooth, wifi, USB y 40 pines de GPIO), buena capacidad de memoria RAM (1GB) y buena capacidad de procesamiento en su tamaño (4 núcleos ARM Cortex A53, 1.2Ghz).

[18] https://www.raspberrypi.org/about/
[19] https://www.raspberrypi.com/products/raspberry-pi-3-model-b/

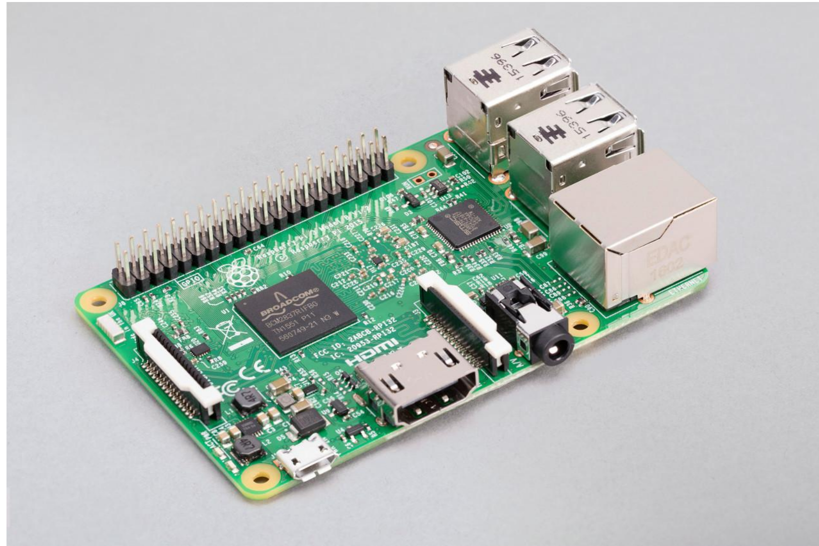

Figura 29. Raspberry Pi 3B.

La Raspberry Pi Foundation posee una distribución propia de Linux Debian llamada Raspbian, que es el sistema operativo recomendado para utilizar en las Raspberry Pi. Sin embargo, para esta tesina se utilizó Debian para arm64, sin modificaciones particulares.

La idea del dispositivo es que sea pequeño, de bajo consumo, bajo costo, y una buena plataforma para todo tipo de aplicaciones, desde fines educativos hasta industriales. Cabe destacar que si bien se dispone de hardware más potente, como lo sería una Raspberry Pi 4, se optó por trabajar sobre la Raspberry Pi 3B, ya que permite ilustrar mejor el propósito de la arquitectura, la cual debería ser posible de montar en hardware de bajo costo, quizás menos potente, y aún así manteniendo un rendimiento aceptable.

5.3.2.2 Aplicación y capacidades

Este dispositivo tiene buena sintonía con las tecnologías utilizadas. Debian es una distribución de Linux de amplia trayectoria, estable y con un amplio repertorio de paquetes instalables. Utilizado de base en este hardware, resulta muy natural la selección de tecnologías y herramientas utilizadas (Java, NiFi, Mosquitto, etc), ya que todas suelen ser utilizadas en estos contextos y se encuentran muchos recursos online relacionados. De esta misma forma es práctico lograr un dispositivo que sea útil en una variedad de contextos. Hay aplicaciones de la Raspberry Pi 3B en contextos hogareños e industriales[20] y al ser probablemente uno de los dispositivos más populares en su categoría, se encuentran muchísimos recursos tanto escritos como audiovisuales de como aplicarlo para diversos fines.

Cómo ejemplo de aplicación, se puede lograr un dispositivo autónomo con una batería y un panel solar, se puede lograr un dispositivo que mida directamente datos conectando distintos tipos de sensores a sus pines GPIO, puede servir de intermediario con otro tipo de sistema por su interfaz de red ethernet convencional, puede ser accedido remotamente y a distancia por su interfaz de red wireless, etc. La flexibilidad que otorga en su despliegue en campo es

[20] https://www.raspberrypi.com/for-industry/

excelente, y nuevamente, al ser popular, es muy posible encontrar guías o ideas de otras personas que ya hayan intentado utilizarlo de formas iguales o parecidas a lo que uno quiera realizar con el mismo.

Un punto más a destacar es la flexibilidad de almacenamiento que posee. Se pueden utilizar tanto su slot micro SD o sus diversos puertos USB para ampliar el mismo. Con la alta capacidad que tienen los dispositivos de almacenamiento sea en forma de tarjeta SD o de pendrive USB en los últimos años, no es muy difícil construir un dispositivo de altas capacidades de almacenamiento (32GB+) que pueda lograr una excelente autonomía en caso de falla de comunicación y que requiera almacenar muestras por una larga duración de tiempo.

5.3.3 Hardware para el servidor de historización

Las características del servidor de almacenamiento de historia son convencionales. Si bien la selección de tecnologías en teoría permitiría montar el servidor entero en otra Raspberry Pi (por ejemplo, la Raspberry Pi 4 de 8GB de RAM seria una buena candidata), se optó por una solución tradicional con un sistema operativo Debian x86_64, en una máquina virtual de 4 núcleos y 4GB de RAM.

No se hizo hincapié en los requerimientos del lado del servidor ya que en términos de hardware reciente, no hace falta nada particular para esta tesina, sólamente capacidad de almacenamiento para los datos y un sistema operativo compatible.

5.3.4 Entorno de pruebas

Las pruebas se realizan emitiendo una cierta cantidad de mensajes del set de datos de referencia, lo más rápido posible.

Para independizar lo que se consideraría como la emisión del dato, de su registro en el dispositivo de borde a partir del cual la muestra comienza el ciclo de historización, la emisión se genera a parte del set de datos de referencia desde una computadora externa hacia el broker de MQTT instalado en el dispositivo de borde. Se determinó utilizar esta metodología para afectar lo menos posible los recursos utilizados por el dispositivo de destino al procesar los mensajes, ya que el script de emisión de los datos puede impactar los resultados si corriera en el mismo dispositivo que se está probando. El script que se utilizó para ejecutar los benchmarks, como los CSVs de los resultados, están disponibles en el siguiente link: https://github.com/dustContributor/historizr/tree/main/historizr-misc/historizr-benchmarks.

Tanto la emisión inicial como la recepción del dato para su almacenamiento están realizadas por ethernet, para eliminar variaciones que puedan ocurrir utilizando una red wireless. Las interfaces de red de 100Mbps de los dispositivos no se determinaron como limitantes para la prueba, ya que la transmisión de datos ronda el orden de algunos KB/s. De todas formas todos los dispositivos están conectados a un switch de 1Gbps y la computadora externa donde se emite el dato tiene una interfaz de red de 1Gbps. La medición de mensajes historizados por

el intervalo de tiempo dado se hace en la base de datos por medio de un trigger, ya que se considera la muestra historizada una vez que está guardada en la base de datos de destino.

Para soportar el volumen de mensajes enviados se utilizaron configuraciones particulares para Mosquitto y para el cliente MQTT utilizado en historizr-device, Eclipse Paho.

Puntualmente en Mosquitto:

- max_queued_bytes 0
    - Deja sin límite la cantidad de bytes de mensajes que puede tener el broker encolados sin emitir.
- max_queued_messages 0
    - Deja sin límite la cantidad de mensajes que puede tener el broker encolados sin emitir.

La consecuencia de no tener esta configuración es que el broker puede llegar a descartar mensajes si el flujo de mensajes emitidos hacia el broker excede el flujo de mensajes procesados por los suscriptores.

En el cliente MQTT utilizado en historizr-device:

- maxInflight 512
    - Incrementa la cantidad de mensajes "en vuelo" que puede tener encolados el cliente al publicar, si es que el broker se está demorando en su recepción.

La consecuencia de no aumentar este límite es que el cliente puede descartar mensajes si se están publicando demasiado rápido por exceder el límite de mensajes encolados.\

5.3.5 Características de hardware y software

- Raspberry Pi 3 B (llamado RPI3 en adelante)
    - 1GB de RAM LPDDR2
    - ARM-A53, 1.2Ghz, 4 núcleos, Broadcom BCM2837
    - 100Mbps ethernet
    - Debian Linux, kernel 6.1.0 arm64
    - Java 17 LTS, de los repositorios oficiales de Debian
    - Mosquitto MQTT broker 2.0.11, de los repositorios oficiales de Debian

- TV Box NogaPC Ultra 2 (llamado TVB en adelante)
    - 1GB de RAM LPDDR3
    - ARM-A53, 1.2Ghz, 4 núcleos, Amlogic S905D
    - 100Mbps ethernet
    - CoreELEC Linux, kernel 4.9 arm64, 32 bit userspace
    - Java 17 LTS, de Azul Zulu para armhf

    - Mosquitto MQTT broker 2.0.15 compilado para armhf

- Máquina virtual (llamada VM1 en adelante)
    - 1GB de RAM DDR4
    - Ryzen 9 3900X, 4 núcleos asignados
    - 1Gbps ethernet virtualizada
    - Debian Linux, kernel 6.1.0, x86_64
    - Java 17 LTS, de los repositorios oficiales de Debian
    - Mosquitto MQTT broker 2.0.11, de los repositorios oficiales de Debian

Esta tesina se concibió originalmente para realizarla en una mini computadora Raspberry Pi 3 B como dispositivo de borde, sin embargo se decidió incorporar otro dispositivo de características similares y bajo costo a las pruebas para tener de comparativa.

No se pudo lograr una misma instalación de Debian Linux en ambos dispositivos ya que los SoC Amlogic no están bien soportados, por lo que se optó por CoreELEC para este, que es una de las pocas distribuciones que tiene soporte para estos SoC. El objetivo de la distribución CoreELEC es aplicaciones multimedia, por lo tanto se deshabilitaron las características gráficas innecesarias y fue necesario instalar Java y Mosquitto manualmente, este último siendo compilado para la arquitectura armhf también manualmente.

Adicionalmente se preparó una máquina virtual funcionando en la misma computadora donde se encuentra la máquina virtual donde se aloja el historizr-server y la base de datos, con características similares en cuestión de núcleos y memoria que los dispositivos físicos. Esta máquina virtual sirve como una línea base de lo que podría llegar a ser posible en condiciones óptimas: Donde la conexión entre el servidor y el dispositivo de borde es por medio de una red local de alta velocidad, y conformando el dispositivo de borde por una computadora con hardware moderno.

5.3.6 Método de envío desde MiNiFi

MiNiFi posee varias formas de emitir información hacia el exterior, que en este caso sería nuestra instancia de NiFi principal en el servidor.

Las formas evaluadas son:

- **PutTCP**. Un procesador de comunicación directa por socket, enviando mensajes como un stream continuo de bytes, separados por una secuencia especial.
    - Se configura en el NiFi de destino creando un procesador **ListenTCP**, que escucha conexiones entrantes en un puerto determinado y separa los mensajes por la secuencia especial configurada.
- **RemoteProcessGroup**. Un procesador de comunicación entre NiFis por un protocolo interno. Posee método de transporte por HTTP y RAW, se utilizó el RAW para esta prueba.

    - Se configura el NiFi de destino habilitando la comunicación remota en su configuración, y creando un procesador de tipo **ListenRemoteInput** de entrada en su flujo principal.
    - Un punto a tener en cuenta es que el input configurado en el NiFi principal tiene un ID único que hay que configurar en los MiNiFis que se conecten. Si el input se borra y se crea de nuevo, es necesario reconfigurar los MiNiFis con el input nuevo.
- **PostHTTP**. Un procesador de comunicación por protocolo HTTP, método POST en este caso para las peticiones salientes.
    - Se configura el NiFi de destino con un procesador **ListenHTTP**, que crea un pequeño servidor web, que escucha en un puerto determinado por peticiones HTTP entrantes.

En la configuración de MiNiFi se puede establecer la cantidad de tareas concurrentes máximas permitidas globalmente, y por nodo procesador dentro del flujo configurado, por lo tanto se procedió a testear el impacto de esto teniendo en cuenta que el hardware de prueba posee 4 núcleos de procesamiento en su CPU, probando varias configuraciones de concurrencia en el nodo que emite los mensajes hacia el NiFi principal.

5.3.7 Benchmark de método de envío

Para este test se procedió a testear los distintos métodos de envío posibles desde MiNifi. Se emitieron las primeras 10 mil muestras del set de datos de referencia hacia el dispositivo de borde, con 5 repeticiones de este mismo test para cada configuración. Se ven los resultados en la Figura 30 a continuación.

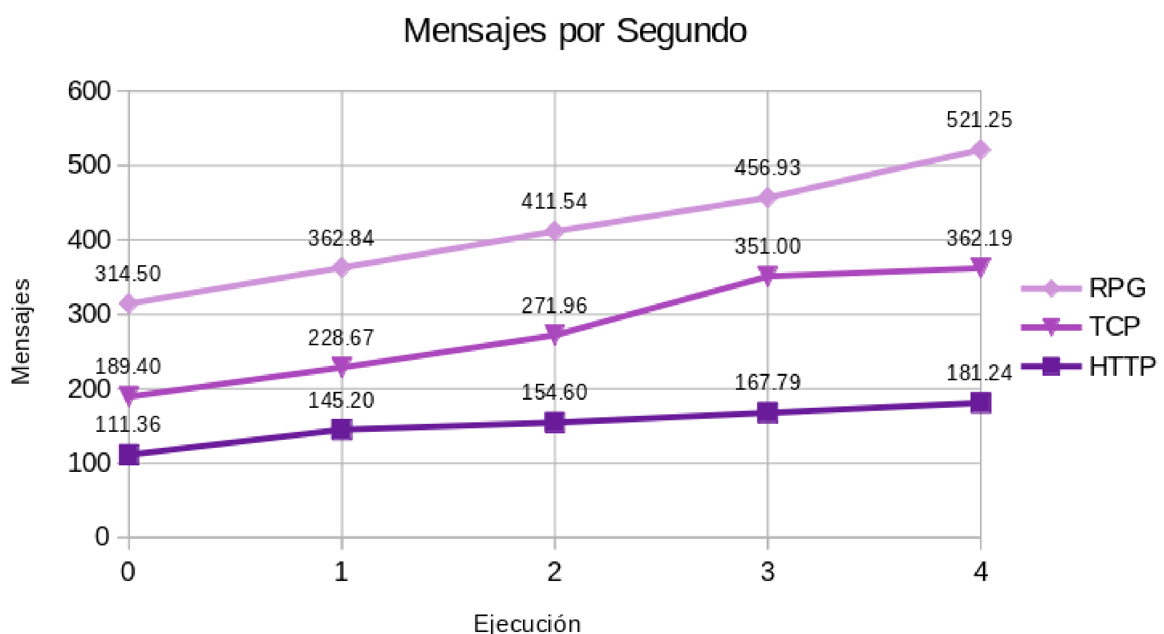

Figura 30. Comparativa de métodos de envío de NiFi y MiNiFi.

Se ve como el método de RemoteProcessGroup (RPG) es el más eficiente dentro de los tres seleccionados, alcanzando los 520 mensajes por segundo en su quinta ejecución. En segundo lugar se encuentra PutTCP (TCP), con 360 mensajes por segundo, y por último PostHTTP (HTTP) con 180 mensajes por segundo.

Se notó que el “calentamiento” de los servicios en Java relacionados es muy relevante, ya que en ejecuciones consecutivas se notaron mejores resultados que las primeras, como se ve en la Figura 30 donde los resultados de la primera ejecución se ubican con casi la mitad de rendimiento comparando con la última.

5.3.8 Benchmark de concurrencia

No se notó diferencia en la configuración de concurrencia para los envíos por RPG y TCP, por lo tanto se realizó la misma prueba pero con un nivel de concurrencia de 16 hilos para el envío HTTP únicamente, 5 ejecuciones de 10 mil muestras cada una. Se grafican los resultados de HTTP y RPG de la prueba anterior como referencia. Se ven los resultados en la Figura 31 a continuación.

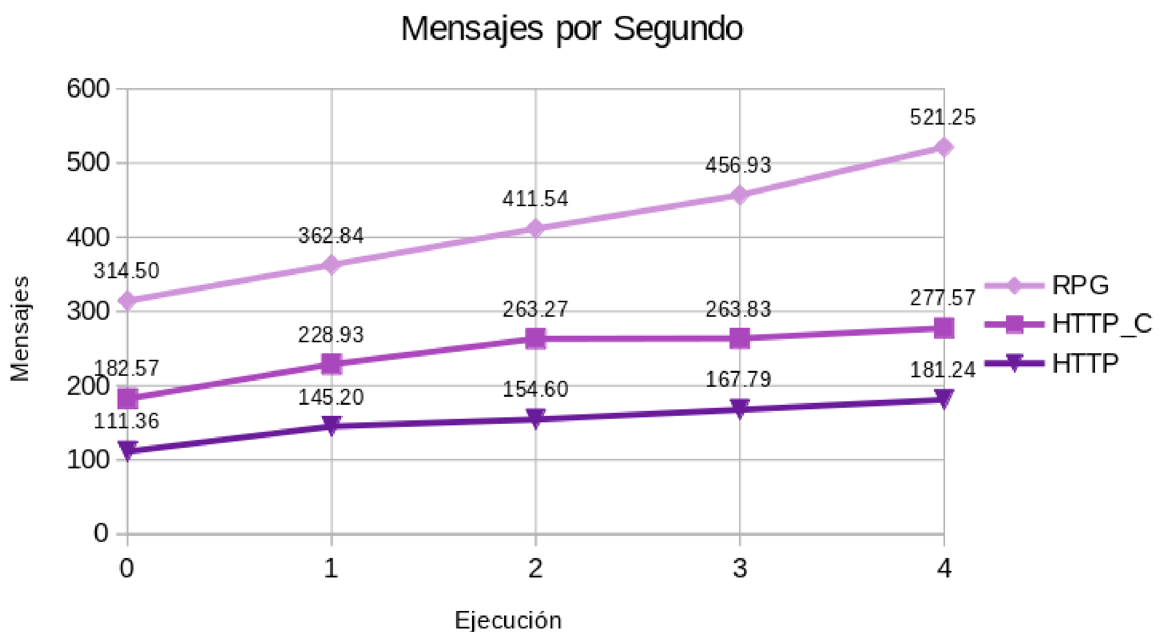

Figura 31. Comparativa de concurrencia por HTTP.

Se notó una diferencia de rendimiento positiva pero no fue la suficiente para cerrar la brecha con el método por RGP.

5.3.9 Benchmark de hardware

En base a los dos benchmarks anteriores, se seleccionó el método de envío RPG sin ninguna configuración de concurrencia adicional. En base a este método de envío, se procedió a

probar los tres dispositivos de borde modelos con este método. La diferencia con las pruebas anteriores es que se hicieron diez ejecuciones en total, de envíos de 10 mil mensajes, alternando el hardware a probar después del quinto test. Es decir, en cada serie de datos hubo 5 pruebas consecutivas, una pausa, y 5 pruebas más consecutivas. Se ven los resultados en la Figura 32 a continuación.

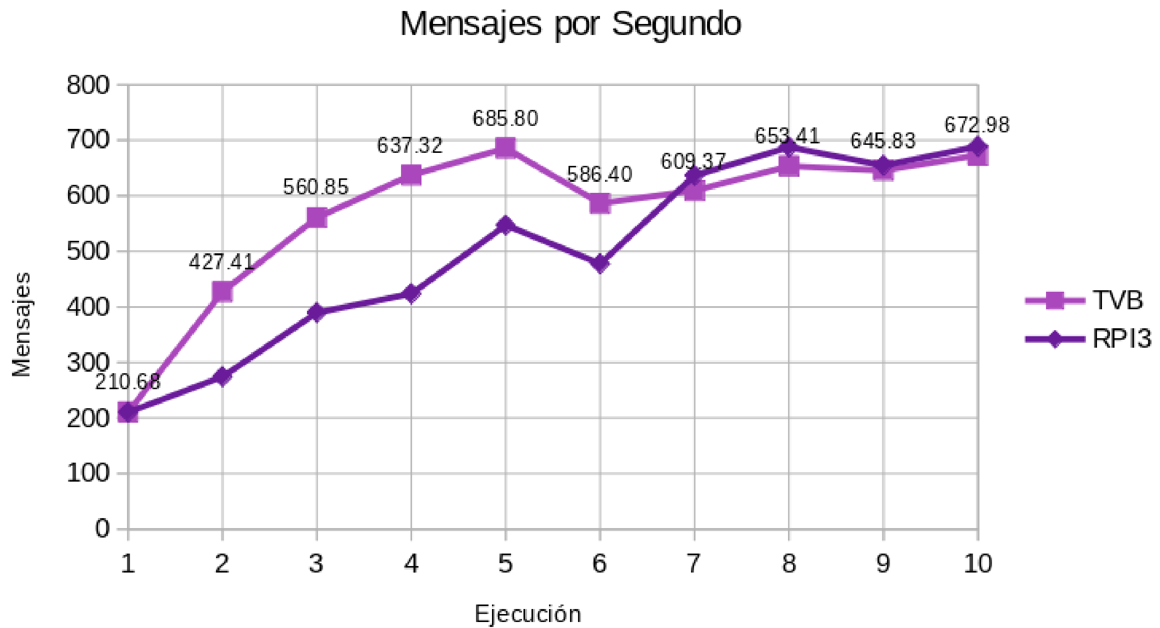

Figura 32. Comparativa de los tres dispositivos modelo.

Para empezar se evaluó el hardware disponible más similar, la Raspberry Pi 3 B (RPI3) y el TV Box NogaPC Ultra 2 (TVB). Inicialmente se vió como el TVB alcanzó más rápidamente un rendimiento estable, pero aún así no logra tener una clara ventaja a partir de la séptima ejecución del test comparado con la RPI3. De lo cual se puede deducir que poseen comportamientos similares luego de una etapa de precalentamiento.

Como referencia, se puso en contexto esta prueba comparando con la máquina virtual tradicional (VM1). Se ven los resultados en la Figura 33 a continuación.

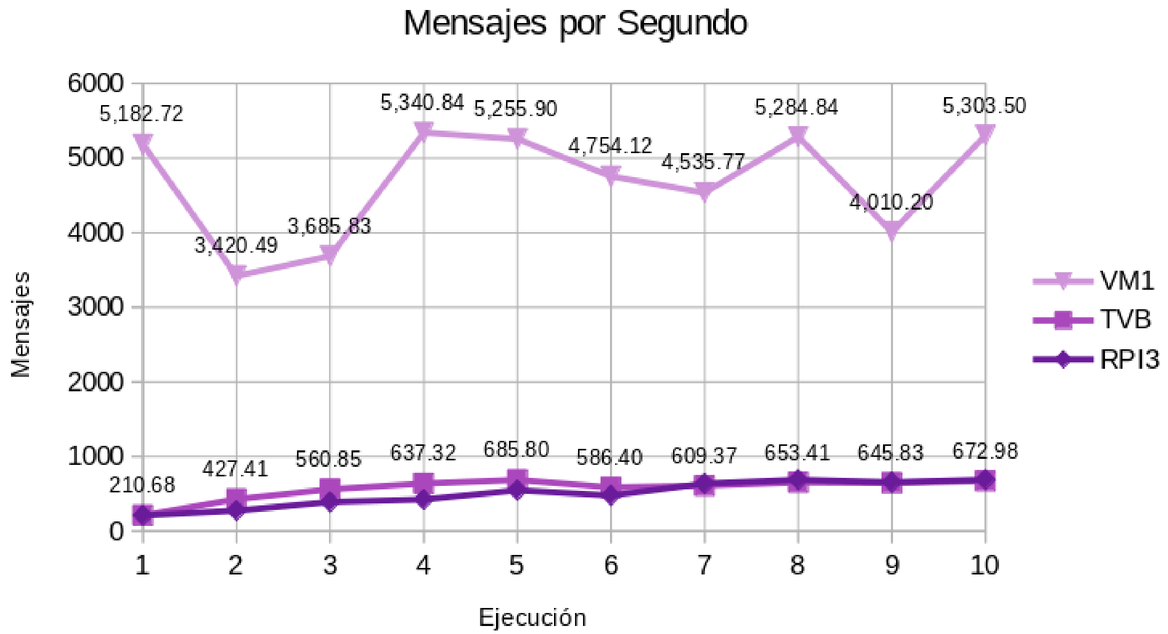

Figura 33. Comparativa de hardware contra una máquina virtual.

Si bien las características en números son similares, se puede ver claramente la ventaja en rendimiento de hardware convencional, con factores de hasta 10 veces más mensajes por segundo procesados por parte de la VM1 comparando con la RPI3 y el TVB. No es necesariamente sorprendente pero sirve para poner en perspectiva las capacidades de las que se están hablando.

Cabe destacar que el hardware de la VM1 y la RPI3 está diseñado para ser alimentado con aproximadamente 5W (1A a 5V), mientras que una computadora tradicional puede consumir de diez a veinte veces más que esa base.

6 Resultados, discusión y conclusiones

El proceso de historización y disponibilización de datos puede tornarse complejo, con una gran cantidad de dispositivos involucrados, sin embargo, el abanico de posibilidades para resolver esta problemática es muy amplio. Durante el desarrollo de esta tesina se lograron seleccionar un conjunto de herramientas y tecnologías muy potentes y fácilmente ampliables, se logró escalar satisfactoriamente el rendimiento demostrando una alta capacidad con hardware de bajos recursos, y también se logró desarrollar servicios necesarios para compensar los huecos de gestión y manipulación de información que se encontraron.

Tanto historizr-server como historizr-device lograron un desempeño más que aceptable sin tener una arquitectura ni desarrollo demasiado complejas, y el servicio de historizr-capture logró fácilmente ejemplificar cómo se puede ingresar datos medidos a la arquitectura por medio de MQTT. Quedó demostrado la viabilidad de Java y Vert.x como bases para desarrollar servicios tanto de gestión de catálogos del lado del servidor historizador, como para procesar directamente la ingesta de muestras en el dispositivo de borde.

En la capa de almacenamiento de datos se evidenciaron resultados muy positivos en términos de espacio consumido por la hypertable sample, por medio de la compresión de los datos lograda con TimescaleDB, y la interacción con la base de datos PostgreSQL fue natural al ser una base de datos relacional convencional, lo que quedó ilustrado en las consultas de ejemplo para confección de estadísticas sobre las muestras almacenadas. En esta tesina sólo se alcanzó a ver apenas la superficie del potencial que se encuentra en aplicar estas herramientas de esta forma.

El uso de herramientas gratuitas y de código abierto es totalmente factible para este tipo de casos de uso, resultando muy atractivas en términos de costos y de potencial integración con otras partes de un ecosistema establecido.

Por último, se mostró el buen rendimiento que se puede lograr con dispositivos pequeños pero de propósito general, como la Raspberry Pi 3 B, que no solo poseen una buena oferta de conectividad tanto por cable como por aire, sino también un rendimiento más que aceptable para ingestar datos medidos en la arquitectura.

6.1 Discusión

En la búsqueda de soluciones establecidas para esta problemática surge la comparativa entre aquellas de distinto estilo. En la tesina fundamentalmente se planteó una solución personalizada, donde en vez de utilizar una única herramienta que abarque todos los aspectos involucrados, se confeccionó a partir de herramientas establecidas pero con un enfoque más puntual en cada una. Por lo tanto no se puede afirmar que sería una solución idónea para todos los casos.

Hoy las necesidades de las organizaciones son muy cambiantes, y hay que evaluar cuidadosamente los costos explícitos, como lo son costos de licenciamiento, como los no explícitos, que son costos de mantenimiento y facilidad de uso a largo plazo. Por un lado se puede decir que una solución personalizada plantea un factor muy favorable de ampliación, ya que en el caso planteado, no hay limitantes a la hora de cambiar las tecnologías utilizadas, o de ampliar el esquema de base de datos, o los servicios desarrollados, la única limitante es el tiempo de mantenimiento y de realización de estas tareas, o las capacidades de desarrollo que pueda tener una organización que quiera implementar una de estas soluciones.

En contrapuesta a este aspecto, las soluciones pre-armadas pueden involucrar costos de licenciamiento y ser más abarcativas, pero más inflexibles. De esta forma hay una relación de costo beneficio entre priorizar la personalización y el control, asumiendo el mayor costo de mantenimiento, con priorizar el bajo costo de mantenimiento de una solución pre-armada, asumiendo que las necesidades futuras puedan tener que ser abordadas por fuera de la solución, o mediadas con el proveedor en caso de que sean abiertos a este tipo de sugerencias.

7 Trabajos futuros

Se encontraron dos aspectos importantes a lo largo de la tesina que no fueron contemplados en un principio pero se identificaron como podrían ser mejorados.

7.1 Esquema de seguridad

En la tesina no se contempló un esquema de seguridad que, por ejemplo, verifique que los dispositivos de borde con el servidor de historización sean validados, o que la comunicación sea segura.

Existen dos puntos en los que la comunicación tendría que ser validada para asegurar estos aspectos: La comunicación de las muestras desde el dispositivo de borde hacia el servidor de historización y las peticiones de configuración que se hacen desde historizr-service hacía la API de historizr-device en el dispositivo de borde.

Ambos son posibles de hacer más seguros. NiFi, y por extensión MiNiFi, soportan comunicación encriptada por medio de SSL y certificados, por lo que se podrían elaborar certificados para tanto el dispositivo como el servidor, para que el dispositivo se asegure de que está transmitiendo datos al servidor que corresponde, y a su vez que el servidor se asegure que está recibiendo los datos de uno de sus dispositivos asignados.

Por otro lado, de la misma forma se puede encriptar la comunicación por HTTP entre el servidor de historización y el dispositivo de borde asignándole un certificado reconocido al dispositivo de borde, que el servidor tiene que validar al establecer una conexión. Esto es posible utilizando comunicación por HTTPS entre ambos, lo cual es posible con Vert.x.

7.2 Esquema de múltiples históricos

En la tesina se planteó en un esquema donde múltiples dispositivos de borde se comunican con un solo servidor histórico, aunque la realidad implica que a gran escala, o incluso por particularidades organizacionales, es posible que esto no sea así y se quiera disponer de varios servidores históricos en simultáneo. Si bien esto escapa el alcance del dispositivo de borde, ya que sigue transmitiendo a un único servidor, si obliga a desacoplar el NiFi de historización y la base de datos, del servicio historizr-server que gestiona todos los dispositivos y sus señales.

En principio cada base de datos histórica tendría su propia tabla de muestras y su propio catálogo de señales, y el catálogo de dispositivos estaría centralizado en otra base de datos que manipularía historizr-server directamente.

Para el flujo básico de historización el catálogo de dispositivos no es estrictamente necesario, esto permite centralizar en otra instancia que manipule historizr-server y que contenga todos

los dispositivos configurados. Para tener un seguimiento habría que agregar un catálogo de históricos y asociar cada dispositivo al histórico al que aporta datos.

8 Glosario

- Benchmark. Prueba de rendimiento, o punto de referencia.
- Dispositivo de borde. Dispositivo usualmente compacto que se encuentra físicamente cerca del punto de medición.
- Endpoint. Refiere a una ruta para acceder a un recurso por HTTP.
- Failover. Refiere al proceso de un sistema de intercambiar automáticamente entre un componente o instancia que no esté funcionando, por uno que sí esté funcionando, para su contínua operación.
- IoT/Internet of things. Internet de las cosas, refiere al concepto de dispositivos interconectados entre sí de forma masiva.
- IoTDB/Internet of things database. Base de datos para la internet de las cosas, proyecto para comunicar y almacenar datos históricos de mediciones.
- MQTT/Message queuing telemetry transport. Transporte y encolado de mensajes de telemetría, protocolo de comunicación ampliamente utilizado en dispositivos de bajos recursos.
- Open source/código abierto. Refiere al código fuente público y accesible, pero también hace referencia a una metodología de desarrollo colaborativo, abierto y públicamente visible.
- Payload. Refiere al contenido o carga de un mensaje.
- Plataforma cloud/plataforma en la nube. Se refiere a un conjunto de herramientas y/o servicios que son ejecutados en servidores no específicos, llamados nube, administrados de forma transparente por un proveedor. Es un modelo de negocios por el cual un usuario adquiere un recurso o tiempo de uso de un recurso sin especificar en qué hardware el mismo va a ser ejecutado.
- SoC/System on a chip. Refiere a una forma de distribución de un conjunto de procesadores y coprocesadores en un solo chip, en vez de una serie de chips interconectados. Utilizado ampliamente en teléfonos móviles, y todo tipo de dispositivos compactos.
- Throughput. Refiere a la capacidad de procesamiento o rendimiento.
- Timestamp. Marca o estampa de tiempo.

9 Referencias bibliográficas

Bridgwater, A. (2015, Julio 21). NSA 'NiFi' Big Data Automation Project Out In The Open. *Forbes*. https://www.forbes.com/sites/adrianbridgwater/2015/07/21/nsa-nifi-big-data-automation-project-out-in-the-open

Evans, D. (2011). *The Internet of Things: How the Next Evolution of the Internet Is Changing Everything*. Cisco. https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf

Freedman, M. (2020, Agosto 3). *TimescaleDB vs. InfluxDB: Purpose-built for time-series data*. Timescale. https://www.timescale.com/blog/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299877/

HMS Industrial Networks. (2019). *MQTT used in production - a use case*. Anybus. https://www.anybus.com/docs/librariesprovider7/default-document-library/whitepapers/mqtt-used-in-production---a-case-study.pdf

IEEE. (2008, Noviembre 16). *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard for Floating-Point Arithmetic. https://ieeexplore.ieee.org/document/8766229

IoT Analytics. (2023, Febrero 7). *Global IoT market size to grow 19% in 2023—IoT shows resilience despite economic downturn*. IoT Analytics. https://iot-analytics.com/iot-market-size/

Microsoft. (2021, Abril 27). *Azure*. Microsoft Azure IoT Reference Architecture. https://azure.microsoft.com/mediahandler/files/resourcefiles/microsoft-azure-iot-reference-architecture/Microsoft_Azure_IoT_Reference_Architecture_2_1_1_update.pdf

MQTT.org. (2022). *FAQ*. MQTT. https://mqtt.org/faq/

Phipps, S. (2013, Enero 11). *Who controls Vert.x: Red Hat, VMware, or neither?* InfoWorld. https://www.infoworld.com/article/2613356/who-controls-vert-x--red-hat--vmware--or-neither-.html

The PostgreSQL Global Development Group. (2023). *About*. PostgreSQL. Retrieved April 6, 2023, from https://www.postgresql.org/about/

Wang, C., Huang, X., & Qiao, J. (2020). *Apache IoTDB: Time-series Database for Internet of Things*. VLDB Endowment. Retrieved April 5, 2023, from https://www.vldb.org/pvldb/vol13/p2901-wang.pdf