**Experimento 1 Final**

**Equipo: πpo**

**Sergio Cárdenas**

**Juan Sebastián Díaz**

**Sergio Guzmán**

**Julián Manrique**

**Carlos Mario Sarmiento**

**Arquitectura y Diseño de Software**

**2018-10**

**Universidad de los Andes**

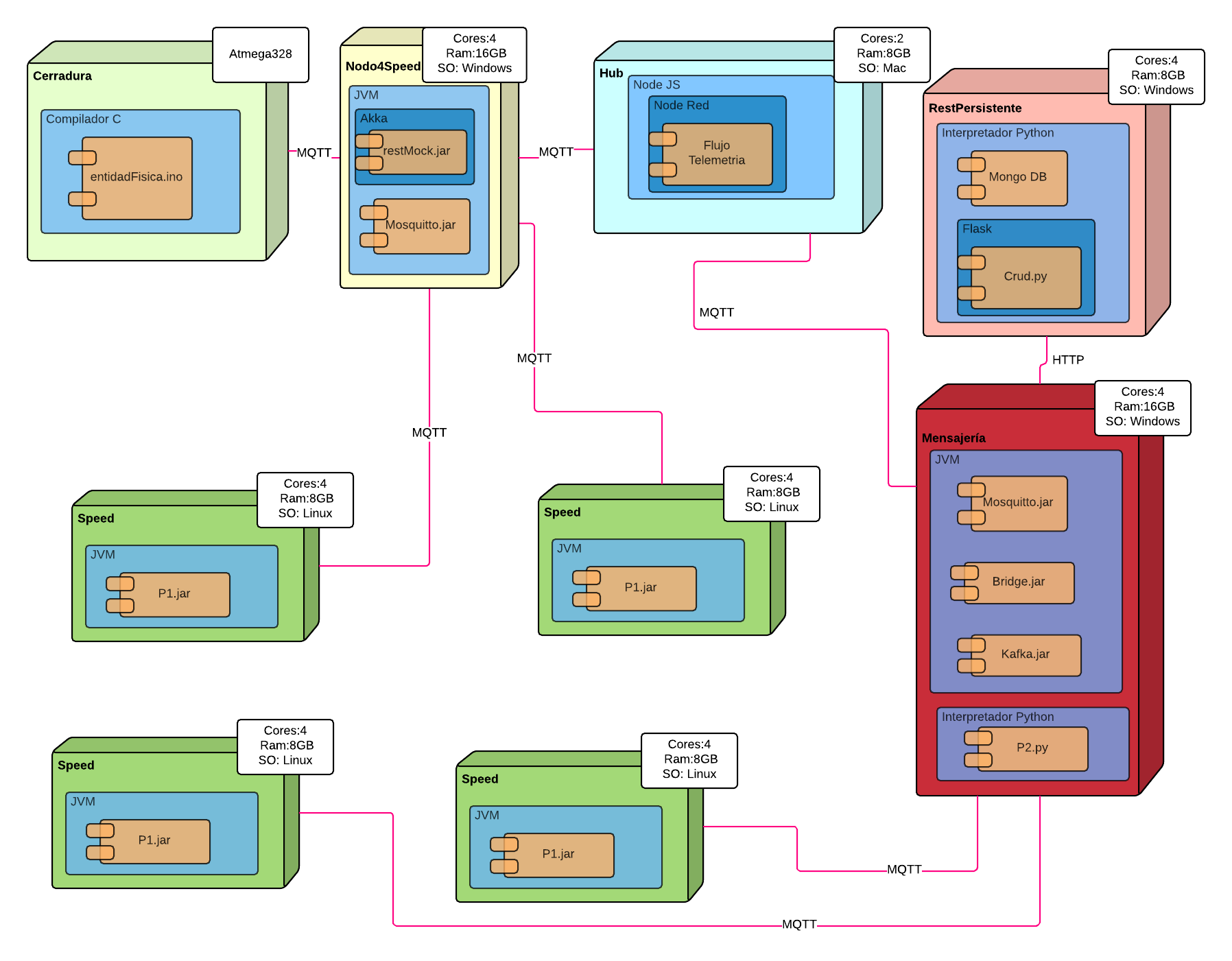
1. **Población de la base de datos**

Para el requerimiento de persistencia, se usó MongoDB, que es una base de datos no relacional documental. Esto se debe a que queremos que las consultas sobre a base se hagan rápidamente. La estructura del documento es como sigue: Se tienen varias unidades residenciales, con datos básicos y un conjunto de inmuebles; cada inmueble tiene sus datos básicos y exactamente un hub. Cada hub tiene un identificador único, información parametrizable, un conjunto de fallos, y una cerradura asociada. Las cerraduras tienen más información parametrizable como el nivel de batería crítico o las claves, un arreglo de horas permitidas para el ingreso, una lista de claves, y un histórico de las emergencias, en donde cada emergencia tiene una fecha, el id de la emergencia, y una breve descripción.

Para poblar la tabla, se registraron 4 unidades residenciales, con 5 inmuebles por cada unidad. También se registró un hub y su cerradura asociada por cada inmueble, con valores arbitrarios para los datos que son parametrizables, tanto para la cerradura como para el hub. Finalmente, se registró un listado de emergencias para cada cerradura, con su fecha, id, y la descripción. No se persistió ninguna alarma debido a que el alcance de esta entrega no las contempla. Para más detalles sobre los datos con los que se pobló la base de datos, la colección de postman que se ejecutó se encuentra en el repositorio de GitHub, en la ruta persistencia/colecciones/Poblar REST Yale - Completo.postman\_collection.json.

1. **Diagrama de despliegue**

A continuación, se presenta el diagrama de despliegue de nuestra arquitectura en donde se omite parte de la capa *speed*:

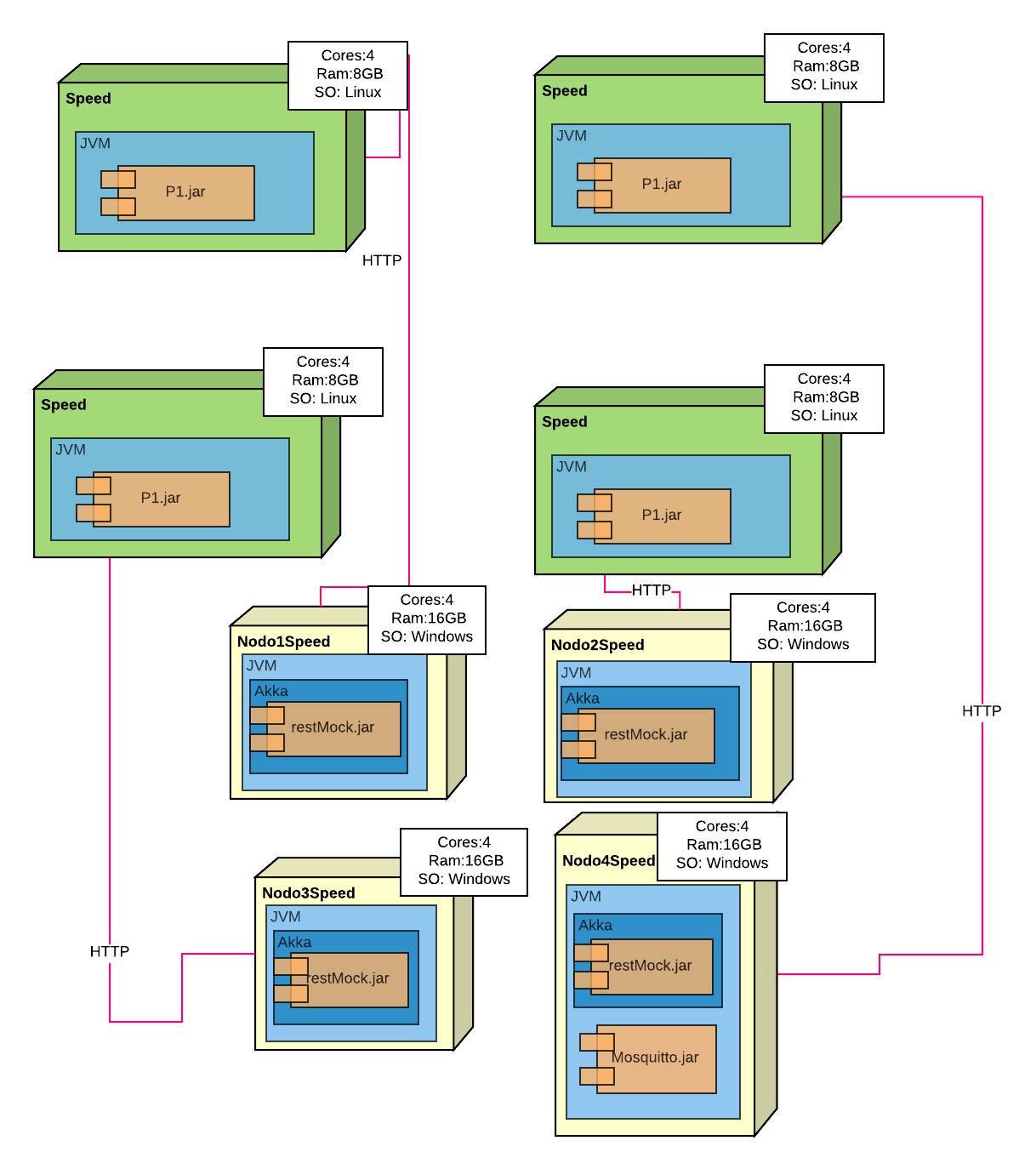


Se decidió usar un servidor de mosquitto para poder establecer una comunicación inalámbrica entre la cerradura (simulada por el arduino) y el Hub (Simulado por Node Red). El Hub procesará la información a través de un flujo de telemetría y la publicará en uno de los dos servidores de mosquitto dependiendo de la región en la que se encuentra el hub, esto con el fin de distribuir la información de emergencias para que los servidores de mensajería no colapsen en caso en que se envíen muchas emergencias al mismo tiempo. Se puede observar que también se tiene un servidor de mensajería Kafka, del que el programa P2 leerá los mensajes para persistir la información de las emergencias a través de un servicio rest que está sobre *Flask*.

En cuanto a la capa *Speed*, se tienen 4 copias de P1 corriendo en máquinas distintas, en donde 2 están suscritas a un servidor de mosquitto y las otras 2 al otro. Para asegurarnos de que cada alarma no llegue a distintas copias de P1, cada copia está suscrita a tópicos distintos para distintas regiones.

Es necesario aclarar que para la arquitectura es necesario que los dos servidores cuenten con un bridge a Kafka para poder persistir las alarmas que provienen de todas las regiones, pero dada la limitación en la cantidad de máquinas disponibles y ya que simulamos únicamente la cerradura y el hub de un solo apartamento, decidimos implementar la persistencia solo para uno de los servidores de mosquitto.

Se muestra la parte restante de la capa *speed* a continuación:



Se observa que tenemos varias copias de computación del rest Mock, en donde cada una está asociada a una copia de P1 para poder soportar una gran cantidad de peticiones en un periodo corto de tiempo. Debido a la poca disponibilidad de máquinas, se decidió ejecutar en una misma máquina un servidor de mosquitto y una copia del rest Mock; de igual manera, el otro servidor de mosquitto, Kafka, el bridge entre estos dos, y el programa P2 corren en una sola máquina.

1. **Pruebas de carga**

Para evaluar nuestra arquitectura, se hicieron distintas pruebas de carga, en donde se requería procesar todas las emergencias sin errores y con un tiempo de respuesta inferior a un segundo. Más específicamente, el escenario de calidad exige que el sistema debe soportar el envío de 300000 alarmas de emergencia que se envían en una ventana de tiempo de 1 minuto, en donde el tiempo de respuesta debe ser inferior a 1 segundo. Para verificar este escenario, se realizaron varias pruebas de carga a través de Apache JMeter, tanto sobre P1 como sobre el rest Mock. Además de estas pruebas, se hizo una prueba de 200 peticiones para poder comparar los resultados en cada caso.

**Pruebas de carga sobre el Rest Mock:**

En la arquitectura propuesta inicialmente, se tenía un balanceador de carga para distribuir las peticiones entre 4 copias del rest Mock. Se decidió usar varias instancias de JMeter para que éste no colapsara y así evitar que los resultados se vean afectados. En total se hicieron 3 pruebas diferentes, en donde se usaron entre 6 y 8 instancias de JMeter corriendo en máquinas distintas, con una distribución de las peticiones que variaba en cada prueba para observar el efecto que podía tener JMeter en los resultados de las pruebas. En las tres pruebas todas las instancias de JMeter ejecutaron la prueba al mismo tiempo.

Para la primera prueba, se emplearon 6 instancias de JMeter, en donde las 300,000 peticiones se distribuyeron de la siguiente manera: 5 instancias enviaron 40,000 peticiones y la restante envió 100,000 peticiones, todas con un ramp-up de 60 segundos. De las 200,000 peticiones que se enviaron equitativamente en las 5 máquinas mencionadas, 4 de esas peticiones fallaron, y la latencia promedio fue de aproximadamente 62 milisegundos. En cambio, de las 100,000 peticiones que se enviaron en una sola instancia de JMeter, fallaron 1800, y la latencia promedio fue de 197161 milisegundos.

Para la segunda prueba, se usaron 7 instancias de JMeter, en donde 6 de ellas enviaron 40,000 peticiones en un minuto y la restante envió 60,000 peticiones en un minuto. Esta vez, de las 240,000 peticiones enviadas por las 6 máquinas que ejecutaron la misma prueba, no hubo ningún error y con una latencia promedio de 27 milisegundos, y de las 60,000 peticiones enviadas por una sola máquina, 10 fallaron, y se obtuvo una latencia promedio de 600 milisegundos.

En la tercera prueba se empleó una instancia más de JMeter: 6 instancias enviaron 40,000 peticiones y las otras 2 30,000 peticiones en un minuto. En esta prueba, no falló ninguna petición y se obtuvo una latencia media de 34 milisegundos.

En cada una de las 3 pruebas se enviaron 300,000 peticiones en un minuto, pero se puede observar que la latencia promedio y los errores variaba dependiendo de cuántas instancias de JMeter se utilizaron y cómo era la distribución de peticiones en esas instancias.

En cuanto a la prueba de 200 peticiones en un minuto, se realizó con una sola instancia de JMeter. Se obtuvo un 0% de error con una latencia promedio de 98 milisegundos. Se observa que con 200 peticiones se obtuvo una mayor latencia promedio; esto puede deberse a que en el momento de realizar la prueba de 200 pudo haber más tráfico en la red o a que los recursos asignados a la máquina virtual disminuyeron al hacer la prueba.

**Pruebas de carga sobre el programa P1:**

Como se observó en las pruebas de carga sobre el rest Mock, los resultados entregados por JMeter podían variar de acuerdo con el número de peticiones que cada instancia, por lo que esta vez sólo se hará una prueba de carga. Para estas pruebas, dado que las peticiones ya se distribuían entre las 4 copias de P1, se decidió cambiar el despliegue del rest Mock a lo mostrado en el diagrama de despliegue (Esta vez no se usa balanceador de carga). Se usaron en total 8 instancias de JMeter con un plugin de MQTT que permitía publicar en un tópico; en donde 4 instancias enviaron 35,000 peticiones por minuto cada una y las otras 4 enviaron 40,000 peticiones cada una.

Como se mencionó anteriormente, se tienen 2 instancias de mosquitto y 4 instancias de P1, por lo que cada servidor de mosquitto debe aguantar 150,000 peticiones y cada instancia de P1 debe aguantar 75,000 peticiones. De esta manera, grupos de 2 instancias de JMeter enviaron por grupo 75,000 peticiones por minuto a los tópicos en los que las instancias de P1 leían, para de esta manera lograr que los mensajes se distribuían uniformemente y para que ningún mensaje se leyera por dos instancias de P1 distintas. El resultado fue que ninguna petición falló, y se consiguió una latencia promedio de 1,55 milisegundos (tiempo medido desde que la petición es leída del servidor de mosquitto hasta recibir la respuesta http del rest Mock).

También se realizó una prueba con 200 peticiones en un minuto, con una sola instancia de JMeter y hacia una única copia de P1. El porcentaje de erro fue 0% y la latencia media fue de 1,46 milisegundos. Podemos ver que la latencia es muy similar en ambos casos; esto se debe a que P1 siempre lee un mensaje de mosquitto a la vez, por lo que la capacidad del sistema depende de la capacidad de almacenar los mensajes en mosquitto y de la velocidad con los que estos mensajes son leídos.

1. **Evaluación de la arquitectura**

Con base en los resultados de las pruebas, podemos decir que nuestra arquitectura pudo cumplir con el escenario de calidad dado. Uno de los elementos más importantes de la arquitectura es el hecho de que las peticiones se distribuyeron en regiones, diferenciándolas a través de la URI de los tópicos en los que cada hub publicaba. Esto permitió tener varias instancias de P1, haciendo que cada una pudiera suscribirse a los tópicos de una región en específico. De esta manera, se logró distribuir las peticiones sin necesidad de usar un balanceador de carga, que pudo haber afectado considerablemente los tiempos de respuesta.

1. **Declaración de esfuerzo**

Los artefactos definidos, junto con la declaración de esfuerzo, se muestran a continuación:

1. Programa de cerradura (Simulado a través de arduino).
2. Programa de hub (Simulado a través de nodered)
3. Servidores de mensajería (Mosquitto, Kafka, Bridge de protocolos)
4. Programa P1 (Extrae de Mosquitto y manda al REST Mock)
5. REST Mock
6. Código de persistencia
7. REST de persistencia
8. Programa P2 (Extrae de Kafka y manda por REST a los servicios de la persistencia)
9. Balanceador de carga (usado en las pruebas de carga del Rest Mock)
10. Montaje físico de cerradura
11. Documentación
12. Video

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| INTEGRANTES\ARTEFACTO | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | Esfuerzo porcentual |
| Juan Sebastián Díaz | X |  |  |  | X |  |  |  | X | X |  |  | 20% |
| Sergio Guzmán M. |  | X | X | X | X |  |  |  |  |  |  | X | 20% |
| Sergio Eduardo Cárdenas | X |  |  |  |  |  |  | X |  | X | X |  | 20% |
| Julián Manrique |  |  |  | X |  | X | X |  |  |  |  |  | 20% |
| Carlos Mario Sarmiento | X |  |  | X |  |  |  | X |  | X |  |  | 20% |