INSTITUTO TECNOLÓGICO BUENOS AIRES

Proyecto Final

Proyecto TiX

Desarrollo e implementación de una arquitectura horizontalmente escalable

Autores
Matías Gabriel
Domingues
Facundo Nahuel
Martinez Correa
Javier Pérez Cuñarro

Tutor
Dr. Ing. José Ignacio
Alvarez-Hamelin

19 de diciembre de 2017

Índice

1. Introducción

1.1. Sobre el Proyecto TiX

El Proyecto TiX (Traffic Information eXchange) busca crear una herramienta para la medición de calidad y uso en las conexiones a Internet con la menor saturación posible del canal.

El sistema del Proyecto consta de dos partes bien definidas: el cliente, instalado en cada uno de los usuarios, y el servicio, instalado en la infraestructura de TiX. Mediante la recolección de los tiempos de viaje de paquetes entre el cliente y el servicio, se hace un análisis usando un algoritmo que devuelve el porcentaje de Utilización y de Calidad de conexión, tanto de subida como de bajada.

Dichas mediciones son realizadas con paquetes UDP [1] en una especie de ping-pong entre el cliente y el servidor. Estos paquetes cuentan con cuatro marcas de tiempo con la cantidad de nanosegundos desde el comienzo del día. Los mismos corresponden a los momentos en que:

- se envía el paquete desde el cliente al servidor
- se recibe en el servidor
- se envía el paquete desde el servidor al cliente
- se recibe en el cliente

1.1.1. Sincronización de Relojes

Para la toma de tiempos en las mediciones se usan los relojes de los Sistemas Operativos donde residen el cliente y el servicio. Si bien muchos Sistemas Operativos mantienen sus relojes sincronizados mediante el uso de NTP [2], esto no necesariamente es cierto para todos los equipos conectados a Internet. A su vez, el protocolo NTP no es lo suficientemente preciso, dando en el mejor de los casos sincronizaciones dentro del milisegundo. También puede haber saltos abruptos de tiempo debido a una sincronización de relojes que pudo haber sucedido entre dos tomas de tiempo de una misma medición.

Todo esto concluye en que los tiempos que son usados en las mediciones no son confiables. Es por ello que hay un método de sincronización de relojes dentro del sistema que permite mejorar la precisión de las diferencias entre las marcas de tiempo.

1.1.2. Estimación de Calidad y Uso de la Conexión

Para la estimación de la Calidad y Uso de la Conexión modelamos el tráfico en Internet como si fuera un proceso auto-similar. Existen estudios que indican

que el tráfico paquetizado en Internet tiene varias propiedades típicas de estos procesos [3, 4]. Además, este modelo permite describir el comportamiento del tráfico tanto para corta como para larga dependencia con sólo tres parámetros: la media de la tasa de tráfico, la varianza del tráfico y el coeficiente de Hurst.

1.2. Sobre la presente iteración

1.2.1. Motivación

El proyecto se tomó luego de dos iteraciones por parte de dos equipos distintos. En estas los objetivos fueron de inicio y avance del proyecto, agregando mejoras y funcionalidades.

Por un lado, existía la necesidad de seguir agregando funcionalidades. Tales incluían, pero no se limitaban, a:

- Portar el cliente a dispositivos móviles para medir la calidad de las conexiones de redes celulares
- Lograr que el sistema tenga una alta estabilidad y disponibilidad
- Lograr que el sistema tenga una alta escalabilidad, suficiente para cubrir más de cinco mil usuarios
- La posibilidad de lograr pruebas comparativas simultáneas de distintos métodos para las estimaciones.

Por otro lado, se quería mejorar la calidad del producto con la idea de que sea considerado un proyecto de Código Libre de calidad.

Sin embargo, por diversos motivos, la calidad del sistema en ese momento era difícil de medir y comprobar. La instalación era compleja y su reproducción laboriosa, ya que se necesitaba que el entorno estuviese preparado de manera similar al servidor de pruebas. Además, muchos de estos conocimientos no estaban documentados, con lo que hubo que recurrir a los creadores originales.

Todo esto redundó en la imperiosa necesidad de ordenar el código y la arquitectura, con el fin de que el sistema pudiera llegar a los ideales especificados. Esto también implicaba la provisión de documentación on-line, en el código y facilidades para la implementación del sistema del servicio por parte de privados.

1.2.2. Objetivos

Dadas las motivaciones y las necesidades de continuidad del proyecto, se destilaron los siguientes objetivos:

- Hacer un sistema que logre proveer servicio a por lo menos 5.000 usuarios concurrentes.
- El cliente del sistema debe ser de fácil instalación para el usuario y debe estar pensado para múltiples plataformas por diseño.
- El servicio del sistema debe ser de fácil instalación para cualquier persona que desee replicarlo en su infraestructura, dados los conocimientos básicos necesarios para el mantenimiento de la misma.
- El Proyecto TiX debe contar con una calidad suficiente para que pueda ser continuado de forma ininterrumpida por otro grupo totalmente ajeno al proyecto.

1.2.3. Desarrollo del presente documento

A lo largo de este documento se procede a explicar el desarrollo de la iteración actual, intentando justificar con la mayor profundidad posible la lógica detrás de las decisiones tomadas.

En la sección de Arquitectura se explica en alto nivel el diseño del sistema en general. Se hace un repaso de la arquitectura previa, su funcionamiento, y las ventajas y desventajas de dicho enfoque.
Luego se detalla la arquitectura actual, haciendo hincapié en las razones del cambio, su funcionamiento, y las ventajas y desventajas del nuevo enfoque.
También se detalla y justifica la selección de herramientas y tecnologías de desarrollo, operaciones, orquestación, pruebas y puesta en producción.

En la sección de Implementación se explica a bajo nivel cada una de las partes del sistema. Explayándose debidamente de acuerdo a su funcionalidad, diseño e interoperabilidad con el resto de las partes del sistema. Es necesario aclarar que dicha sección está pensada como documentación explícita para desarrolladores del sistema. Todos los repositorios necesarios se encuentran en el grupa TiX Measurements de GitHub, bajo la URL https://github.com/TiX-measurements/.

En la sección de Ensayos se explican los distintos tipos de pruebas que le fueron realizados al sistema con el fin de comprobar su actual calidad y el grado de cumplimiento para con los objetivos pactados.

Por último, en la sección de Conclusiones se desarrollan los resultados finales de la actual iteración, así como se plantean los lineamientos para la continuación del proyecto.

2. Arquitectura

2.1. Arquitectura previa

El Proyecto TiX en principio fue pensado como un conjunto de subsistemas. En la Figura 1 se puede ver un diagrama de los servicios que comprendían y que, a grandes rasgos, aún comprenden el actual sistema de TiX. Estos son el Subsistema de Presentación y Administración, el Subsistema de Procesamiento y el Cliente del sistema.

2.1.1. Responsabilidades e implementación de los subsistemas

Subsistema de Presentación y Administración
Este subsistema estaba integrado exclusivamente por la aplicación Web con la que interactuaba el usuario. La misma era una aplicación monolítica hecha en Java [5], usando la combinación de Spring [6], Hibernate [7] y Wicket [8]. Su instalación era manual, usando un Apache Tomcat en el servidor de la infraestructura mediante un archivo WAR [9] que se compilaba en la computadora del desarrollador. Cumplía también el rol de administrar los usuarios y sus distintas instalaciones.

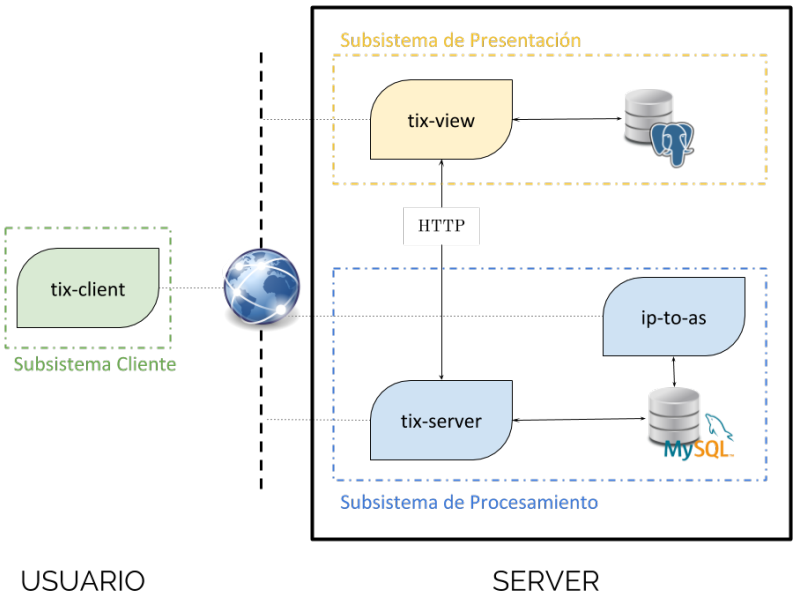

Figura 1: Arquitectura previa del Sistema

Subsistema de Procesamiento
Este subsistema estaba integrado por dos servicios que compartían la misma base de código.

Por un lado estaba el Servicio de Procesamiento de Reportes, el cual era un sistema monolítico, con código principalmente hecho en Python 2.7 [10] y con algunas llamadas externas a código escrito en R [11]. Su instalación se realizaba a través de Capistrano [12], una herramienta escrita en Ruby [13], y de forma remota en el servidor. La función de este servicio era la de interactuar con el cliente, actuando como servicio de respuesta ante los paquetes enviados.

Por otro lado estaba el IP2AS, el cual era un servicio escrito en Python. El mismo se encargaba de mantener actualizada la tabla que relacionaba direcciones IP con Sistemas Autónomos y Proveedores de Servicio de Internet. El mismo funcionaba en base a una tarea agendada que se ejecutaba automáticamente gracias a un Cron [14].

Subsistema Cliente
Este subsistema era un archivo ejecutable de Python que corría en segundo plano. Su principal y única función era la de enviar los distintos paquetes hacia el Subsistema de Procesamiento con el fin de realizar las mediciones.

2.1.2. Funcionamiento conjunto del sistema

El sistema, para un usuario determinado, comenzaba a funcionar tras la creación de la cuenta y la posterior descarga del cliente a su computadora a través del Subsistema de Presentación y Administración. En ese momento se creaba una instalación desde la cual se enviaban paquetes UDP y reportes al Subsistema de Procesamiento. Éste recibía los reportes con las mediciones tomadas, comprobaba su integridad y los almacenaba en un directorio del Sistema Operativo. Tras contar la cantidad de reportes recibidos, si éstos llegaban a una cantidad suficiente, se procedía a procesarlos con el fin de estimar el uso y calidad de la conexión para el intervalo de tiempo correspondiente. Una vez hecho el procesamiento, se cotejaba el Sistema Autónomo y Proveedor al que pertenecía la IP de donde habían llegado los reportes. Luego el Subsistema de Procesamiento enviaba el resultado al Subsistema de Presentación y Administración a través de una pequeña interfaz REST [15] expuesta. Así es que finalmente el usuario podía visualizar los resultados a través del Subsistema de Presentación y Administración.

2.1.3. Fortelazas de esta arquitectura

La principal fortaleza de esta arquitectura era el buen balance que había entre la distribución de los distintos componentes y el diseño monolítico de los mismos. Esto permitía que estén bien separadas las capas y que pudieran ser montadas o desplegadas en distintos servidores de manera autónoma en infraestructuras

que cumplan con los requerimientos propios a cada estrato. A su vez, el hecho de que cada una de estas capas sea monolítica en sí misma confería una gran facilidad en el seguimiento de errores y su eventual solución.

El hacer el cliente exclusivamente en Python tenía como principal idea permitir iteraciones rápidas, y la entrega de actualizaciones constantes y minimalistas al usuario. Esto sería a través de actualizar exclusivamente los archivos de Python modificados, en una suerte de repositorio constante.

2.1.4. Problemas y debilidades de esta arquitectura

Tras trabajar durante unos meses con este sistema, se hicieron evidentes varios problemas y posibles dificultades futuras.

La forma en que el Subsistema de Presentación y Administración había sido diseñado mostraba problemas sustanciales al momento de interactuar con el resto de los subsistemas, así como para evolucionarlo. Estaba construido con una versión muy vieja de Spring y pensado exclusivamente como una aplicación Web, no como un sistema que pudiera interactuar con otros otros. La pequeña interfaz REST que se exponía fue una adición que excedía al diseño original, con lo que se convertía en código con mucha deuda técnica y de muy difícil modificación o mejora.

El haber empaquetado todo el subsistema en un único archivo, hacía que fuera muy difícil referenciar ficheros que estuvieran por fuera del paquete. Por este motivo, se terminó decidiendo empaquetar en el mismo archivo WAR los binarios para los clientes. Esto trajo como obvias consecuencias que dicho archivo fuera muy grande y que debía ser re-empaquetado ante una actualización del cliente.

El Subsistema de Procesamiento era un servicio en Python que se levantaba y corría de forma autónoma. Sin embargo, su diseño estaba pensado para que corriese exclusivamente en la máquina del desarrollador y en el servidor de pruebas de TiX. Hacerlo correr en un entorno distinto a estos implicaba tocar variables definidas en el código de la aplicación para que su configuración sea la adecuada. Estas variables estaban distribuidas a lo largo de toda la aplicación y con poca o nula documentación.

La aplicación Cliente que se instalaba en las computadoras de los usuarios estaba pensada para tener una fácil actualización. Sin embargo, dependía de que el usuario volviese a empaquetar la aplicación. Algo que no se le podía pedir.

Finalmente, el protocolo de comunicación, implementado en su totalidad en Python, estaba escrito dos veces - en el cliente y en el Subsistema de Procesamiento - haciéndolo muy difícil de mantener. Además, la implementación no era clara ni mantenía las especificaciones dadas por el protocolo.

2.2. Arquitectura actual

En base a las observaciones presentes en el capítulo anterior, se decidió hacer una evaluación de la arquitectura. Se hizo evidente que para continuar con el desarrollo y las subsecuentes mejoras se debían hacer cambios profundos en el sistema. Se llegó a la conclusión de que los siguientes tres pilares eran necesarios para finalizar el proyecto exitosamente:

1. El Proyecto tiene que poder ser encarado y progresado por terceros ajenos al desarrollo inicial y con escasa o nula ayuda.
2. El Sistema debe ser fácil de medir, para poder ser mejorado de manera continua.
3. El Sistema debe ser robusto, confiable y performante, para cumplimentar con los objetivos del presente Proyecto de Grado.

Como consecuencia de los tres postulados anteriores, se destiló que:

- El sistema debía ser sencillo, evitando la repetición de código donde fuera posible, aislando las funcionalidades en módulos concretos y usando un único servicio externo por tipo, a menos que hubieran razones de performance que lo justificaran.
- Todas las partes debían contar con pruebas comprensivas, que ayuden a formalizar el código y a evidenciar su intención.
- Todas las partes debían tener una instalación automatizada que garantizaran el correcto despliegue de las mismas independientemente del entorno donde se encontrasen.

Teniendo en cuenta estos últimos dos ítems, es que se decidió utilizar tres herramientas fundamentales en la arquitectura y el trabajo de operaciones del sistema.

- **TravisCI [16]** como sistema para automatizar no sólo las pruebas en cada uno de los repositorios, sino también el despliegue.
- **Docker [17]** como forma de mantener congeladas las dependencias y el estado de cualquiera de las partes del sistema que se instalasen en el servidor.
- **DockerHub [18]** como repositorio para guardar y distribuir todas las partes con el fin de que más gente lo quisiera adoptar y mejorar, además de como facilidad para la instalación en el servidor.
- **Docker-Compose [19]** como mecanismo básico de orquestación para los servicios del sistema.

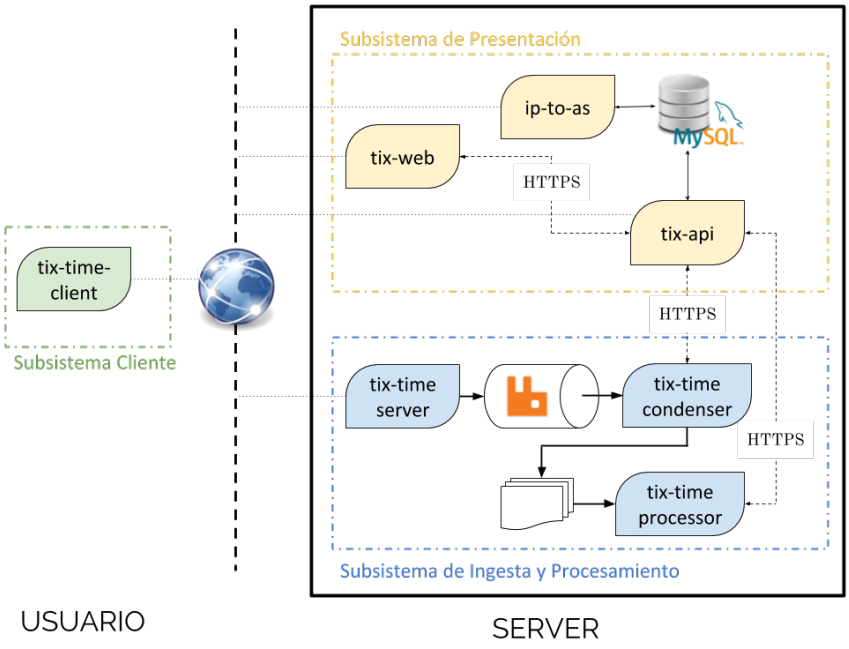

Figura 2: Arquitectura actual del Sistema

En lo que respecta al primer punto, se tuvieron en consideración los siguientes ítemes:

- La arquitectura y la separación de responsabilidades entre los subsistemas existentes era buena, aunque en su implementación se filtrasen algunas entre los distintos módulos. por ejemplo, el IP2AS.
- No era necesario el uso de dos bases de datos.
- Se debía formalizar el protocolo de comunicación entre el Subsistema Cliente y el Subsistema de Procesamiento mediante una biblioteca que fuera común a ambos.
- La información de negocio, y en consecuencia, la de visualización debía estar contenida exclusivamente en el Subsistema de Presentación.
- Cada una de las partes debía ser homogénea en sí misma, evitando ejecutar procedimientos que pudieran vulnerar la integridad del sistema.

En base a todo esto, se entendió que el Sistema en general debía tener la arquitectura que se muestra en la imagen de la Figura 2.

2.2.1. Responsabilidades y consideraciones de los subsistemas

Generalidades
Como se explaya en la subsección previa, se tuvieron en cuenta dos consideraciones generales y que tocan a más de un subsistema.

La primera es la necesidad de una biblioteca que formalice el protocolo de TiX. Esta se expone a través de un repositorio central, del cual puede ser descargado como cualquier administrador de dependencias. Dicha biblioteca, tix-time-core, posee una gran batería de pruebas automatizadas debido a que es una pieza central de toda la arquitectura y por ese motivo se intenta que sea lo más confiable posible. Por otro lado, la idea de hacerla en Java [5] y exponerla a través de Maven [20] como dependencia responde a las necesidades tanto del Subsistema Cliente como del Subsistema de Ingesta y Procesamiento. Java es un lenguaje multiplataforma, que óptimo para hacer un cliente enfocado a distintos Sistemas Operativos. Por otro lado, la JVM [21] es una Máquina Virtual con grandes prestaciones y excelente performance en relación a otros lenguajes de programación. Además, Java cuenta con un rico ecosistema de bibliotecas y frameworks que permiten el fácil desarrollo de sistemas.

La otra gran consideración es la de mover el servicio de IP2AS del Subsistema de Ingesta y Procesamiento al Sistema de Presentación. Dos fueron las ideas detrás de esta decisión:

1. Simplificar el enriquecimiento de los datos para la visualización del usuario, dejando esta tarea exclusivamente al Subsistema de Presentación.
2. Usar un único servicio de base de datos para guardar la información.

Con respecto a esta última idea, se decidió usar el Sistema de Manejo de Bases de Datos Relacional MySQL por ser altamente confiable, de código libre, con grandes prestaciones y sencillo de usar.

Subsistema de Presentación
El Subsistema de Presentación fue pensado bajo los mismos principios que se repiten a lo largo del proyecto. Se buscó generar una arquitectura que fuera altamente escalable y de fácil instalación, por lo que se dividió el subsistema en dos módulos.

Capa de presentación
Esta capa permite al Usuario:

- Visualizar los reportes de sus distintas instalaciones
- Visualizar y editar las instalaciones creadas por el usuario
- Editar sus preferencias de usuario (email/contraseña)
- Visualizar sus reportes mensuales por instalación

También le permite al Administrador:

- Visualizar las instalaciones de todos los usuarios del sistema y sus reportes
- Editar el rol de los demás usuarios, así como deshabilitar cuentas.
- Visualizar gráficos de utilización generales filtrados por Proveedor de Servicios de Internet (ISP), así como también descargar los informes en un archivo con formato de Valores Separados por Comas (CSV) para su posterior análisis.

La principal tecnología de desarrollo es ReactJS [22], la cual permite, mediante el uso de empaquetadores como Webpack [23], servir el código ya compilado mediante una Red de Distribución de Contenidos (CDN). Esto asegura que la aplicación pueda llegar a un gran número de usuarios de forma efectiva y a muy bajo costo.

Capa de de Administración y Reglas de Negocio
Esta capa expone una interfaz, la cual puede ser alcanzada por cualquier cliente que sea capaz de autenticarse vía Json Web Token (JWT) [24] o Basic Auth [25]. Esto permite que si en el futuro se decidiera implementar una aplicación mobile, entonces el desarrollador podría de manera muy simple conectarla el servicio y consultar o agregar reportes.

Existen dos roles definidos para los usuarios del sistema, el de Usuario Común y el de Administrador. Cuando el usuario se autentica mediante usuario y contraseña, se le confieren ciertos permisos de acuerdo al rol.

Los permisos para el rol Usuario Común son:

- Consultar sus reportes
- Editar su email
- Editar su contraseña
- Consultar, editar o borrar sus instalaciones.

Mientras tanto, si el usuario autenticado tiene rol de Administrador, además de tener los mismos permisos que un usuario regular, este puede:

- Consultar los reportes basado en ISP
- Consultar los reportes basado en un usuario particular
- Consultar, editar o borrar instalaciones de otros usuarios
- Generar nuevos reportes para un usuario.

El Rol de Administrador también es usado por el Subsistema de Ingesta para que pueda ingresar datos del usuario sin tener que depender de las credenciales del mismo.

En busca nuevamente de escalabilidad, esta capa se encuentra detrás de un servidor Nginx. Esto permite que, de ser necesario, se puedan instalar múltiples instancias de este servicio y realizar un balanceo de carga.

Finalmente, los datos consultados y creados por esta capa son almacenados en MySQL, en junto con los datos de IP2AS.

Subsistema de Ingesta y Procesamiento
El Subsistema de Ingesta y Procesamiento fue pensado con las siguientes especificaciones en mente:

- Debe ser capaz de ingerir y procesar la información de 5000 clientes en simultáneo
- Debe ser altamente escalable
- Debe poder validar los paquetes recibidos desde el cliente y asegurar que pertenezcan a instalaciones activas usuarios activos del sistema

Y además debe bajo las especificaciones generales del Sistema TiX presentes al principio de este capítulo.

Para logralo se decidió ir por una arquitectura de microservicios. Esto confería tres grandes ventajas:

1. Fácil escalamiento horizontal y selectivo de acuerdo al cuello de botella en cualquier momento dado
2. Alta especialización de cada uno de sus componentes, permitiendo usar las tecnologías que mejor se ajusten al trabajo que realicen
3. Sencillez en el diseño de cada uno de los componentes

Con esto en mente, se definieron cuatro servicios con funcionalidades bien definidas.

tix-time-server
Es el primer microservicio y la puerta de entrada al sistema. Su trabajo consiste en recibir paquetes, validar que sean paquetes íntegros del protocolo de TiX y devolverlos con las marcas de tiempo correspondientes. Además, si estos paquetes cuentan con reportes completos, enviarlos al siguiente servicio en la tubería de ingesta.

La decisión de implementarlo en Java sigue la idea que se explica anteriormente respecto de su gran performance. Es un lenguaje que cuenta con un amplio ecosistema de bibliotecas. Siendo Netty[26] una de ellas. Ésta permite la creación

de servidores de alta performance. Además, es una biblioteca que se encuentra disponible tanto para Java para Computadoras como para dispositivos móviles.

tix-time-condenser
Es el segundo microservicio del sistema. El objetivo de éste es validar que los mensajes de reportes sean correctos y pertenezcan a instalaciones activas de usuarios activos del sistema. Una vez hecha esta validación debe dejar los archivos de reportes en el directorio correspondiente al usuario y la instalación para su eventual procesamiento.

Este servicio también está implementado en Java, pero mediante el *framework* Spring Boot [27]. El cual permite la creación de micro-servicios de una forma sencilla y rápida, ofreciendo gran flexibilidad de configuración y un rico entorno para hacer pruebas. Todo esto sin dejar de ofrecer la performance natural de Java y una baja marca de tamaño en memoria.

tix-time-processor
Es el tercer y último microservicio del sistema. Fue creado con la idea de ser un servicio que corra de forma programada en un cierto periodo de tiempo. Toma los archivos de tal como los haya dejado el anterior servicio, para luego calcular las principales métricas para cada instalación.

La decisión de implementarlo en Python se debe a que este servicio cumple un rol más bien analítico y cercano a lo que es la Ciencia de Datos. Python cuenta con un ecosistema de bibliotecas y entornos de trabajo muy favorecido en este sentido, como lo son NumPy [28], SciPy [29], Matplotlib [30] y Scikit-learn [31]. Además, dado su rol especial, algunas pruebas del sistema no pueden ser ejecutadas de forma automática. Estas pruebas son las que se refieren al análisis y la mejora de la calidad en la estimación de las métricas. Para eso se tiene en cuenta el uso de Notebooks de Jupyter [32], un entorno de trabajo natural para la Ciencia de Datos y Python, principalmente para el análisis exploratorio de los datos.

Por último y no por ello menos relevante, el hacerlo en Python también cumple con la motivación de hacer un servicio sencillo y escalable mediante el uso de Celery [33]. Celery es un entorno de trabajo y una cola de trabajos implementada enteramente en Python. La misma cuenta con varios modos de uso. El elegido por nosotros es el modo programado, el cual se ejecuta de forma autónoma en un determinado período. De esta manera, se puede paralelizar el trabajo de procesar los datos en varios servicios replicados.

Cola de mensajes
Es el cuarto servicio del subsistema. Tiene la responsabilidad de comunicar el tix-time-server con el tix-time-condenser. También sirve como soporte en la comunicación del maestro y los esclavos de la cola de trabajos del tix-time-processor.

La decisión de usar el servicio RabbitMQ [34] para esto fue natural. Es una cola de mensajes con una gran cantidad de conectores para varios lenguajes, y

cuenta con funcionalidad directa con Celery y con Spring Boot. Además, es una cola de mensajes muy performante y fácil de administrar.

Como se puede ver, cada componente del Subsistema es autónomo en sí mismo e idempotente. Lo que confiere una gran escalabilidad horizontal.

A su vez, el que el tix-time-server sea sumamente sencillo y que las dos operaciones más pesadas sean la de procesar el paquete UDP como un paquete válido del protocolo TiX, y poner el paquete de reportes en la cola de mensajes, hace que el sistema entero tenga una gran capacidad de respuesta.

Subsistema Cliente
Para este subsistema, la consideración primaria se resume en que un único cliente se pueda usar en distintas plataformas, o por lo menos, que el código permita su reuso y adaptación simple para diversas plataformas. Al referirse a plataformas se piensa en computadoras, celulares, consolas, smartwatches, smart TVs, etc.

Tomando esto como punto de partida, se decide programar el cliente en lenguaje Java pensando en su gran ventaja: es un lenguaje multiplataforma. Existen intérpretes del lenguaje para casi todas las plataformas conocidas, incluyendo Windows, Linux, Mac y Android. El código resultante es independiente de la plataforma en la que terminará corriendo, ya que corre sobre la JVM (Java Virtual Machine) que se encarga de interpretar el conjunto de instrucciones al lenguaje de máquina donde se esté ejecutando el programa.

Otra consideración importante es que sea fácil de instalar para un usuario promedio, especialmente usuarios de sistemas operativos populares. Se decide no asumir que el usuario sabe usar una terminal de comandos, y proveer una interfaz visual y cotidiana para la instalación del cliente. Siguiendo el mismo proceso que cualquier otra aplicación, el usuario no tiene dificultades para comenzar a usar este software.

La tercera consideración tiene que ver con proveer una facilidad adicional al usuario: alimentar a la aplicación con actualizaciones de forma automática. Este proceso debe ser transparente para el usuario, es decir, no debe requerir que entre al sitio web de TiX para hacer descargas adicionales. Y tiene otra gran ventaja: asegura que un gran porcentaje de usuarios de TiX usan la versión más actualizada del cliente. Esto es muy útil para asegurarse de que cualquier cambio o agregado al software tendrá efecto rápidamente y los usuarios podrán beneficiarse de estos cambios.

Por último, y siguiendo en línea con los puntos anteriores, se toma la premisa de que el cliente sea lo más simple posible en cuanto a la interfaz gráfica y la interacción que requiere de parte del usuario. Lejos de interrumpir su trabajo diario, el cliente debe exigir datos de autenticación e instalación para una configuración inicial, y a partir de ese instante permanecer latente en la barra de estados.

2.2.2. Funcionamiento conjunto del sistema

Al igual que con la vieja arquitectura, el sistema actual comienza a funcionar tras la creación de la cuenta del usuario a través del servicio tix-web en el servicio tix-api, siendo el primero sólo una interfaz amigable del segundo. Luego de la descarga del cliente a su computadora, se crea la instalación y se comienza a enviar reportes al tix-time-server. Cuando el tix-time-server recibe un reporte, lo pone en la Cola de Mensajes para que sea tomado por el tix-time-condenser. Éste valida que el mensaje sea de una instalación activa de un usuario activo contra tix-api. Entonces chequea la integridad del reporte, para asegurar que no haya sido corrompido. Luego deja el mensaje en el sistema de archivos, en el directorio correspondiente al usuario y su instalación. Cuando el momento es correcto, el tix-time-processor toma todos los reportes de la instalación, con los que estima las métricas para esa ventana de tiempo y envía el dato a la tix-api. La misma enriquece los datos del reporte aportando el Sistema Autónomo y el Proveedor de Servicios Internet, cruzando la IP que figura en el reporte con la información que actualiza el servicio IP2AS. Así es que finalmente el usuario puede visualizar los resultados a través del servicio tix-web que consulta al servicio tix-api.

2.2.3. Fortelazas de esta arquitectura

Las principales fortalezas de esta arquitectura nacen de su naturaleza totalmente distribuida.

- Es una arquitectura altamente escalable, incluso dentro de un solo servidor.
- Es una arquitectura robusta, en el sentido que si uno de los componentes del subsistema se cae, el servicio se vería degradado pero no necesariamente interrumpido.
- El uso de tecnologías idóneas para cada servicio en línea con las tendencias actuales y las mejores prácticas presentes.
- La gran performance aparente, consecuencia de poder elegir lo que mejor se comporta en cada caso.
- Cada servicio es medible en sí mismo como una caja negra, permitiendo identificar cuellos de botella de forma preventiva mediante el uso de pruebas de carga aisladas.
- El tener un servicio central con una interfaz estándar como la tix-api, confiere la posibilidad de hacer clientes e interfaces amigables al usuario de una forma más sencilla y rápida además de una excelente integración con otros sistemas.

2.2.4. Debilidades y problemas de esta arquitectura

Naturalmente, las debilidades y problemas también nacen de la forma distribuida de la arquitectura.

Las modificaciones entre Subsistemas (e incluso entre servicios) debido a un cambio en el protocolo de comportamiento entre sí, pueden llevar a tiempos de desarrollo muchísimo más extensos que con otras arquitecturas. Además, requiere de pruebas y ensayos mucho más severos que con una arquitectura monolítica.

Los errores espontáneos o en tiempo de ejecución son mucho más difíciles de encontrar que en una aplicación monolítica. Esto se debe a la distribución de la bitácora de ejecución a lo largo de todo el sistema, y no en un solo lugar.

2.3. Conclusiones de la arquitectura actual

Al principio de la sección previa se declaran tres pilares sobre los cuales se basa esta iteración del proyecto, que buscan satisfacer los objetivos y motivaciones planteadas en la Introducción. Estos son:

- El Proyecto tiene que poder ser encarado y progresado por terceros ajenos al desarrollo inicial y con escasa o nula ayuda.
- El Sistema debe ser fácil de medir, para poder ser mejorado de manera contínua.
- El Sistema debe ser robusto, confiable y performante, para cumplimentar con los objetivos del presente Proyecto de Grado.

Con expuesto, es evidente que la elección de tecnologías de apoyo y operaciones están orientadas a satisfacer el primer punto. En tanto que el cambio de arquitectura y la distribución de servicios estan orientados a satisfacer los últimos dos puntos.

3. Implementación

3.1. Protocolo TiX

Como se explica en la sección de Arquitectura, el protocolo de paquetes de TiX fue implementado como una biblioteca común para el Cliente y el servicio tix-time-server del Subsistema de Ingesta y Procesamiento. El código está completamente en Java, contiene las clases básicas para el manejo de los paquetes tanto de parte del cliente como del servidor.

A su vez, está basado en la biblioteca Netty, la cual contiene facilidades para la implementación de clientes y servidores de alta performance tanto en TCP [35] como UDP.

El código y la historia de esta librería se encuentra en el repositorio https://github.com/TiX-measurements/tix-time-core.

3.1.1. Diseño general del Protocolo

El protocolo de TiX cuenta con dos tipos paquetes, uno corto y uno largo. El paquete largo es una extensión del paquete corto en el sentido que los primeros campos son similares. A su vez, el paquete largo puede ser un paquete largo simple o un paquete de datos. La diferencia entre los últimos dos, es que el segundo contiene el reporte con mediciones tomadas en el último período de tiempo.

El paquete corto actualmente ocupa el tamaño de cuatro números de tipo long de la JVM 8, es decir, 32 bytes. En tanto que el paquete largo ocupa lo mismo que seis números de tipo long de la JVM 8 más 4.400 bytes, es decir, 4.448 bytes.

Mientras que la funcionalidad del paquete corto es exclusivamente la de crear mediciones para la calidad del enlace, el paquete largo también busca evaluar el ancho de banda. Sin embargo, esta funcionalidad actualmente está deprecada y bajo revisión debido a que un paquete de poco menos de 4,5 KB no alcanza para medir el ancho de banda de los enlaces actuales. Además, es un paquete considerablemente grande para lo que son los límites actuales de transferencia de datos en los planes para celulares. Bajo esta óptica, es necesario estudiar una mejor forma de obtener el ancho de banda real del enlace minimizando el costo para líneas de teléfonos móviles.

Los paquetes del protocolo TiX están pensados para poder ser usados con cualquier protocolo de capa de transporte, como TCP y UDP. De hecho, al no tener ningún tipo de control sobre la transmisión y el correcto arribo de los mensajes, es recomendable utilizarlo sobre TCP. Sin embargo, dados los algoritmos de reconstrucción de paquetes y optimizaciones en el protocolo TCP, como el algoritmo de Nagle [36], las mediciones pueden verse afectadas. Por este motivo, solamente el protocolo UDP es utilizado hasta que exista alguna otra forma

de garantizar las correctas mediciones usando TCP. El puerto por defecto del protocolo TiX es el 4444.

3.1.2. Funcionamiento del Protocolo

El protocolo tiene dos formas de funcionamiento, de alto uso de ancho de banda y de bajo uso de ancho de banda. La primer forma es la que usaba el viejo Sistema TiX, mientras que el nuevo Sistema usa la segunda.

El funcionamiento general es como se describe a continuación.

1. Se define una frecuencia f, con la cual se enviarán los mensajes entre el cliente y el servidor.
2. Se define N, que será la cantidad de mediciones acumuladas en los reportes.
3. Se define M, que será la cantidad de reintentos de envío del paquete de datos.
4. A los $1/f$ segundos se marca un paquete con la cantidad de nanosegundos desde el comienzo del día y se envía al servidor.
5. El servidor al recibir el paquete, lo marca con la cantidad de nanosegundos desde el principio del día.
6. Antes de devolver el paquete al cliente, vuelve a marcarlo con la cantidad de nanosegundos desde el principio del día y lo envía.
7. El cliente recibe el paquete desde el servidor y lo marca con la cantidad de nanosegundos desde el principio del día.
8. Tras N intercambios y $1/f$ segundos, el cliente genera un paquete de datos y lo marca con la cantidad de nanosegundos desde el principio del día. Con esto se repiten los pasos 5, 6 y 7. En caso del cliente no recibir el paquete en el paso 7, reintentará el envío M cada $1/f$ segundos, tras lo cual descartará esas mediciones. En caso de recibir el paquete, lo usará como una medición más y volverá al paso 4.

Actualmente, f está definida en 1, N en 60 y M en 5.

3.1.3. Taxonomía de los paquetes

Los paquetes cortos se definen de la siguiente forma:

t_1: long	t_2: long	t_3: long	t_4: long

donde cada campo son 8 bytes que contienen un número de tipo long de la JVM 8 en complemento dos a la base. Dichos números son la cantidad de nanosegundos desde el principio del día cuando el cliente envió el paquete (`t_1`), el servidor recibió el paquete (`t_2`), el servidor devuelve el paquete al cliente (`t_3`) y el cliente recibe paquete (`t_4`).

Los paquetes largos simples se definen de la siguiente manera:

t_1: long	t_2: long	t_3: long	t_4: long
padding			

donde los cuatro primeros campos son idénticos y con la misma funcionalidad que los del paquete corto. El último campo, padding, contiene un conjunto de caracteres aleatorios, con el fin de hacer que el paquete ocupe el tamaño deseado. Por último los paquetes largos con datos se definen de la siguiente forma:

t_1: long	t_2: long	t_3: long	t_4: long
data_header: char string			
data_delimiter: char string			
user_id: long		installation_id: long	
public_key: byte string			
data_delimiter: char string			
report: char string			
data_delimiter: char string			
signed_hash: byte string			
data_delimiter: char string			
padding			

donde los cuatro primeros campos son idénticos y con la misma funcionalidad que los del paquete corto y largo simple.

Los campos llamados `data_delimiter` son usados para diferenciar los distintos campos de datos dentro del paquete. Estos campos con la cadena de caracteres "`;;`" codificada en UTF-8 [37].

El campo data_header es un campo diferencial que contiene la cadena de caracteres "`DATA`" codificada en UTF-8. Este campo se usa para distinguir un paquete largo común de un paquete largo con datos.

El campo `user_id` es un campo de 8 bytes que contiene un número de tipo long de la JVM 8 con el ID del usuario que realizó las mediciones.

El campo `installation_id` es un campo de 8 bytes que contiene un número de tipo long de la JVM 8 con el ID de la instalación donde se realizaron las mediciones.

El campo `public_key` contiene una cadena de bytes de largo variable con la clave pública de la instalación del usuario que realizó las mediciones. La misma

se realiza con el algoritmo RSA [38] con 2048 bits y está encodeada en formato X.509 [39].

El campo `report` contiene el reporte del usuario con las mediciones realizadas por esa instalación. El mismo es una cadena de bytes encodeada en base64, lo cual resulta en una cadena de caracteres de largo variable.

El campo `signed_hash` contiene el hash del reporte firmado por el usuario con la clave privada de la instalación. El mismo es una cadena de bytes que corresponde al algoritmo de firma SHA1 [40] con RSA [41].

El último campo, `padding`, contiene un conjunto de caracteres aleatorios, con el fin de hacer que el paquete ocupe el tamaño deseado.

3.1.4. Taxonomía de los reportes

Los reportes enviados en los paquetes largos de datos están serializados en binario, con el fin de que ocupen el menor espacio posible. Estos reportes tienen una estructura de lista o bitácora en donde cada entrada está formada por cinco campos de la siguiente forma:

`day_timestamp: long`	`t_1: long`	`t_2: long`	`t_3: long`	`t_4: long`

Cada uno de estos campos ocupa 8 bytes y representa un número de tipo `long` de la JVM 8 en complemento dos a la base. El campo `day_timestamp` representa la cantidad de segundos desde el 1 de Enero de 1970 a las 00:00 en que fue realizada esa medición según el reloj del Cliente. Los siguientes campos representan la cantidad nanosegundos desde el comienzo del día cuando se envió el paquete del cliente al servidor, cuando el servidor recibe el paquete, cuando el paquete es enviado desde el servidor al cliente, y cuando finalmente el cliente vuelve a recibir el paquete; respectivamente. Los nanosegundos allí registrados son en relación a los relojes de los respectivos sistemas y no están sincronizados entre sí.

3.2. El Subsistema de Ingesta y Procesamiento

3.2.1. Servicio tix-time-server

Este es un micro-servicio que está implementado enteramente en Java. Su código se encuentra en el repositorio https://github.com/TiX-measurements/tix-time-server.

Usa la biblioteca Netty para crear el servidor UDP del Protocolo TiX y un servidor HTTP con el fin de verificar la salud del servicio. Ambos servidores

pueden configurar su puerto. El punto de entrada para verificar la salud del servicio responde a la dirección /health.

También importa la biblioteca tix-time-server para poder interpretar el Protocolo TiX y comunicarse con el Cliente.

Otra importante dependencia es la del cliente AMPQ [42] para RabbitMQ. Esta la utiliza para poder conectarse con la cola de mensajes y poner allí los paquetes de datos del Protocolo TiX.

Con el fin de garantizar una forma estándar de comunicarse con los servicios que consuman de la cola que alimenta, los utiliza la biblioteca Jackson para convertir los objetos del tix-time-core en cadenas de caracteres de formato JSON [43]. Su configuración se realiza a través del objeto ConfigurationManager, el cual tiene la capacidad de tomar los valores de configuración por distintas fuentes, como archivos de configuración, variables de entorno, etc. con distintas precedencias entre sí.

Además adopta el concepto de entorno, el cual define valores por defecto para las distintas variables para un entorno dado.

Esto confiere una gran flexibilidad al momento de configurar el servicio, ya que la precedencia de los valores de configuración proveniente de las distintas fuentes combinado con el uso de entornos permite tener valores por defecto globales, sumados a valores de entorno propios del servicio más valores sobrescritos por las otras fuentes para el caso de datos sensibles, como pueden ser contraseñas, o de valores dinámicos, como pueden ser niveles de reportes de eventos.

Como se puede apreciar en la Figura 3, el diseño es sencillo, de acuerdo con lo que debe ser un micro-servicio.

El servicio cuenta con pruebas de unidad que aseguran el correcto funcionamiento de cada uno de sus puntos más importantes. También posee una prueba de integración que asegura la correcta interoperabilidad con los demás servicios.

3.2.2. Servicio tix-time-condenser

Alojado en el repositorio https://github.com/TiX-measurements/tix-time-condenser. Al igual que el servicio tix-time-server, este micro-servicio está hecho enteramente en Java. Pero en vez de ser estar hecho desde cero usando varias bibliotecas, éste usa el entorno de trabajo SpringBoot.

El entorno de trabajo SpringBoot está pensado específicamente para el prototipado y la creación de micro-servicios. Provee una gran cantidad de facilidades e importa de forma automática muchas bibliotecas que facilitan el trabajo. Además provee varias facilidades para las pruebas de unidad y de integración.

Tal es así, que el servicio cuenta con tan solo una clase con lógica de negocio. El resto de las clases son para abstraer las respuestas de la comunicación con el servicio tix-api. Esto se ve representado en el UML de la Figura 4.

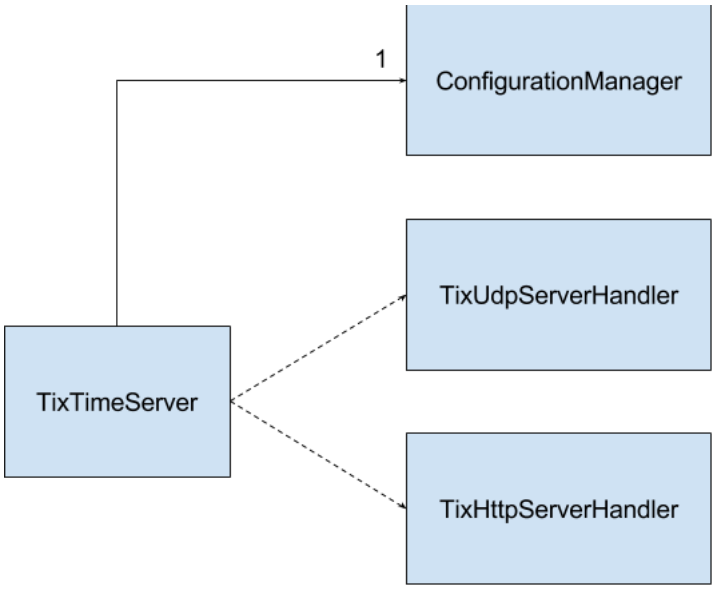

Figura 3: Diagrama UML del servicio tix-time-server

Este servicio también importa la biblioteca tix-time-core. Esto lo hace para poder traducir los mensajes en JSON que consume de la cola de mensajes proveniente de tix-time-server. Además, hace uso de las facilidades de esta biblioteca para validar la integridad de dichos mensajes.

Una vez recibido y verificado el mensaje, el servicio hace llamadas a tix-api para validar el usuario y la instalación del cual provienen. En caso de éxito, procede a dejar el mensaje en el directorio configurado, bajo la ruta acordada. Dicha ruta tiene tiene la forma de /{user_id}/{installation_id}.

Cabe destacar la complejidad y cantidad de operaciones que realiza para ser un micro-servicio, pese a la gran sencillez del diseño.

Al igual que el tix-time-server, posee pruebas de unidad y de integración para asegurar el correcto funcionamiento e interoperabilidad con el resto de los sistemas.

Como detalle importante, este es el único servicio que tiene una prueba de integración en vivo, es decir, que se hace con el servicio compilado y corriendo como si fuera en su entorno normal mediante el uso de Docker.

3.2.3. Servicio tix-time-processor

A diferencia de los otros dos micro-servicios, este está implementado enteramente en Python3. Se basa en el entorno de trabajo Celery. Su código se encuen-

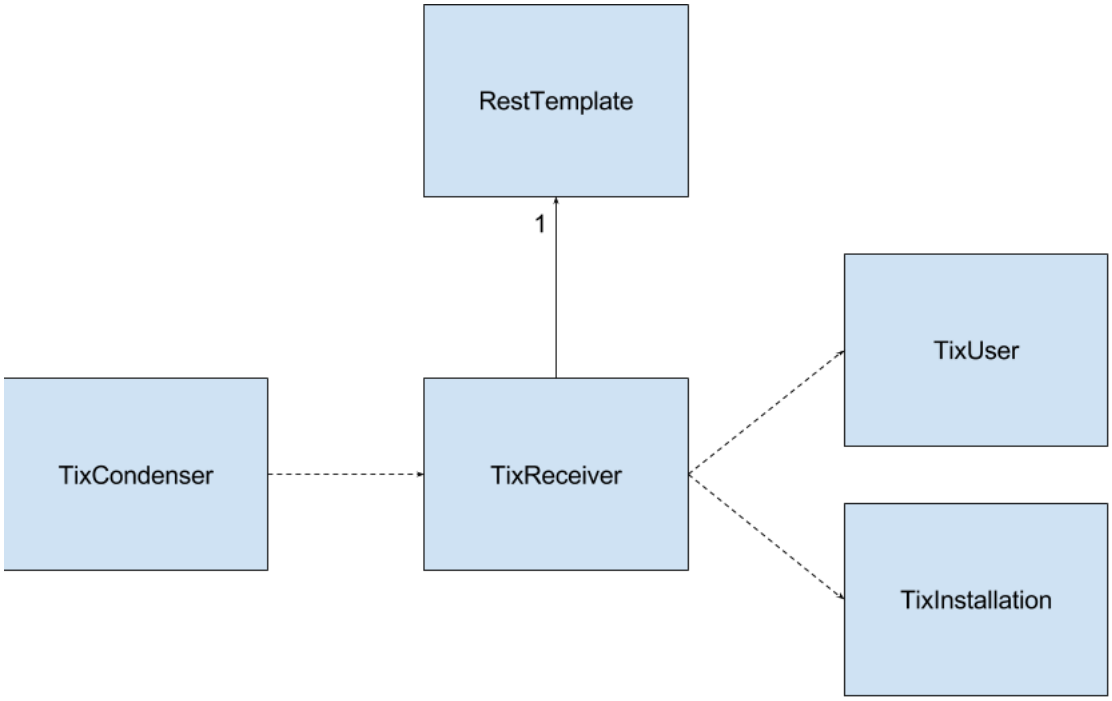

Figura 4: Diagrama UML del servicio tix-time-server

tra alojado en el repositorio https://github.com/TiX-measurements/tix-time-processor.

Celery es una cola de trabajos asíncrona y distribuida hecha casi en su totalidad en Python. Tiene una importante interoperabilidad con el servicio de cola de mensajes RabbitMQ, aunque puede utilizar otros similares también. Una de sus principales características es la posibilidad de programar la ejecución de tareas y distribuirla entre sus distintos esclavos. Esto es lo que hizo que Celery sea una respuesta acertada al trabajo por lotes distribuido del procesamiento de estimación de uso de red y calidad de conexión a partir de los reportes de los clientes.

Dentro de las dependencias más importantes de este servicio están las bibliotecas Numpy y SciPy. Ambas bibliotecas son de las más conocidas e importantes de la comunidad científica y matemática de Python. Se usan principalmente para hacer algunos cálculos como es la estimación del coeficiente de Hurst, regresiones lineales y simplificar el trabajo matemático.
La tercer dependencia importante de este servicio es la famosísima requests. La cual usa para enviar los resultados de las estimaciones al servicio tix-api.
Por último, también usa la biblioteca PyWavelets [44], con el fin casi exclusivo de computar la transformada wavelet en la estimación del parámetro de Hurst.

Como se puede ver en las Figuras 5, 6 y 7, este servicio es más complejo que los otros dos de este subsistema. Esto se debe a que probablemente tanto el ReportsHandler como el Analyzer deberían ser bibliotecas aparte, con su propia funcionalidad.

El Analyzer es el caso más obvio para esta futura mejora. Ya que las posibilidades de hacer pruebas de unidad con esta parte del código son limitadas y precisa de un análisis exhaustivo de los resultados que arroja. Es por eso que el repositorio posee un Notebook de Python que lo acompaña. Para hacer las veces de forma de prueba exploratoria de los resultados de este paquete.

Por otro lado, el ReportsHandler es prácticamente una biblioteca de serialización y deserialización completa de los reportes del Protocolo TiX. Con lo que puede ser meritoria de su propia base de código aislada.

Por consiguiente, las pruebas de unidad en este servicio están limitadas a los paquetes que son más sencillos de probar, ReportsHandler y el paquete api_communication.

3.2.4. Despliegue y puesta en producción

Para automatizar la tarea de desplegar el Subsistema de Ingesta y Procesamiento se recurrió a la herramienta Docker Compose. La misma permite mediante el uso de un archivo YAML describir la configuración que debe tener el sistema, los distintos servicios que lo componen y cómo interactúan entre sí, además de sus dependencias. Además posee comandos sencillos que permiten levantar, actualizar, parar y desarmar el sistema.

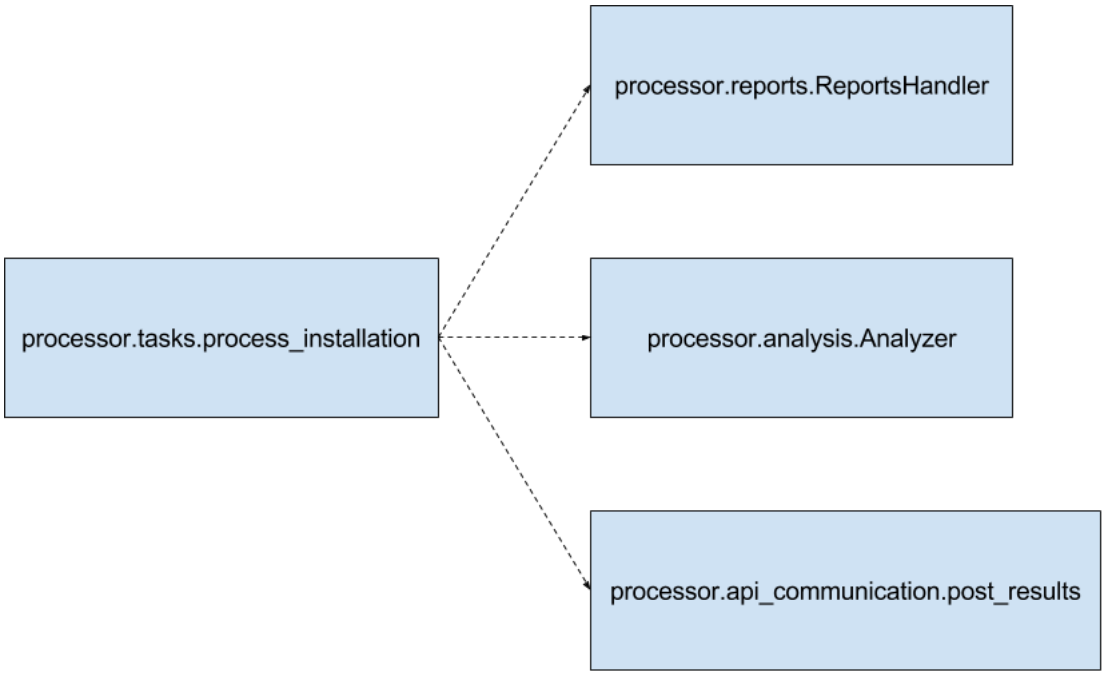

Figura 5: Diagrama UML del servicio tix-time-server

Esto, junto con la posibilidad de interoperar con la herramienta Docker Swarm, permiten que el sistema en su conjunto pueda ser manejado con sencillez y robustez, además de ser escalado a lo largo de un clúster de máquinas, si es necesario.

Todo esto convierten a la herramienta en indispensable para el manejo de microservicios como los aquí explicados.

Este documento se encuentra en el repositorio tix-time-deploy y cuenta también con variables sensibles ya declaradas que se esperan existan definidas en el entorno para poder levantar el sistema completo. Estas variables son credenciales de algunos de los servicios de este Subsistema y de otros servicios de otros subsistemas.

Un ejemplo de esto se encuentra en el repositorio https://github.com/TiX-measurements/tix-time-deploy, donde está la configuración mínima para dejar corriendo todo el Subsistema junto con documentación para hacerlo.

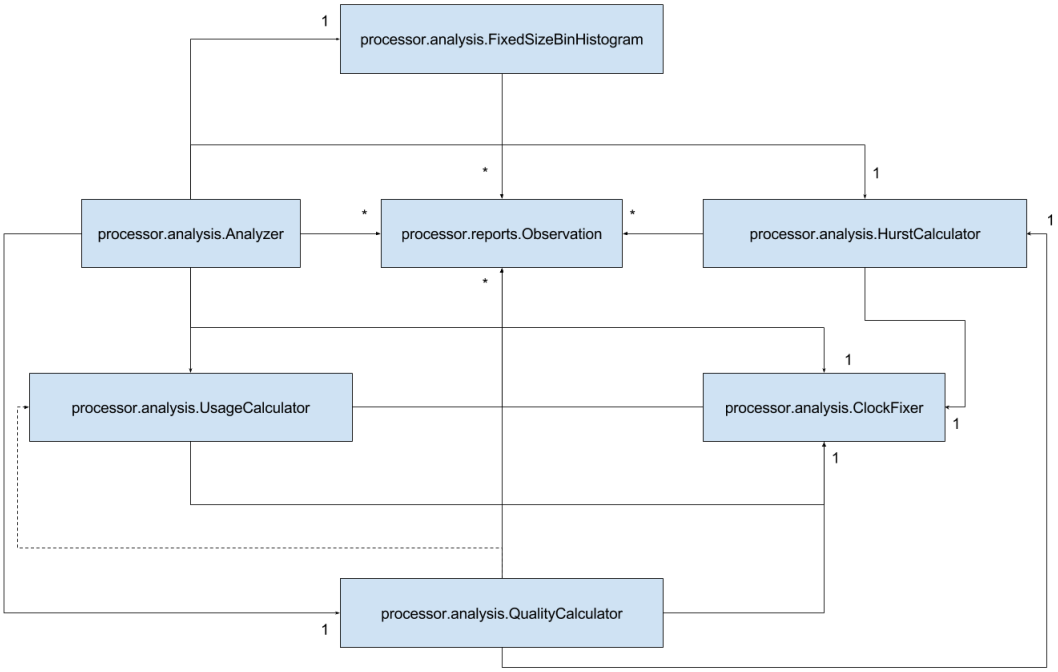

Figura 6: Diagrama UML del servicio tix-time-server

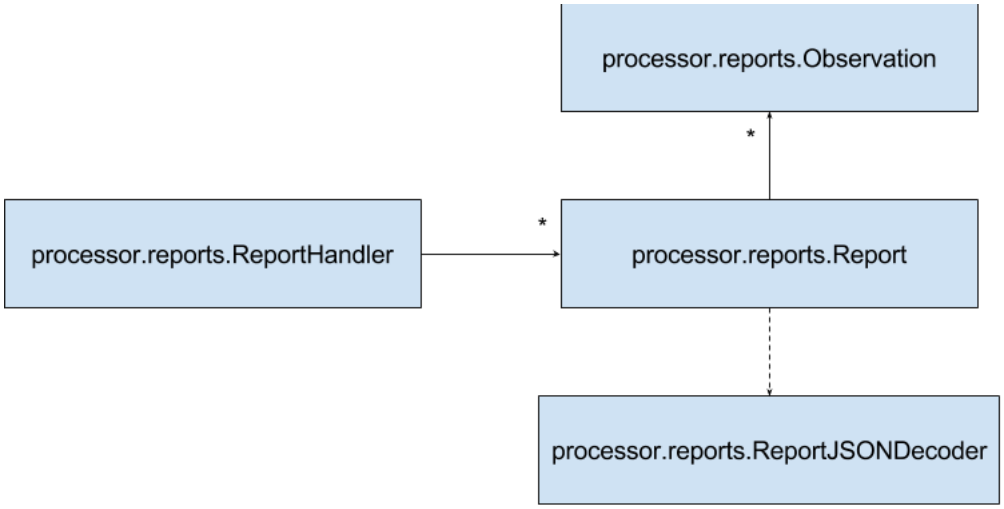

Figura 7: Diagrama UML del servicio tix-time-server

3.3. El Subsistema Cliente

3.3.1. Aplicación Cliente

El Cliente es uno de los subsistemas del proyecto que está programado en lenguaje Java y se encuentra alojado en el repositorio https://github.com/TiX-measurements/tix-time-client. Esto permite crear una única versión de la aplicación y despreocuparse por las diferencias de programar en código nativo para los distintos sistemas operativos. Actualmente funciona en computadoras personales, pero no sería difícil que funcione en dispositivos móviles o televisores que tengan la habilidad de ejecutar código Java. Para asegurar que el cliente no tenga las rutas del sistema definidas explícitamente en el código, se hace uso del método `System.getProperty()` que retorna variables propias del sistema que está ejecutando el programa. De esta manera, no es necesario definir una serie de sistemas operativos de antemano con sus respectivas rutas, sino que Java asume la responsabilidad de obtener los valores correctos ad-hoc.

Si bien el binario ejecutable es único, la aplicación se distribuye con un instalador nativo que en este caso sí varía según el sistema operativo. La secuencia esperada consiste en acceder a la página web de TiX, crear una cuenta nueva y en la misma sección descargar el instalador de Windows, el de Linux o el de OSX. En el caso de Windows, es un archivo .exe de tipo asistente y guía al usuario paso a paso. Para OSX, es un archivo .dmg que requiere que el usuario haga drag-and-drop del ícono de la aplicación en la carpeta Applications, un procedimiento familiar para estos usuarios. Para Linux, está disponible el instalador .deb, que se puede instalar de forma sencilla usando la herramienta dpkg desde línea de comando. Para otros sistemas operativos, como última alternativa, se puede usar la aplicación sin instalador ejecutando el archivo .jar.

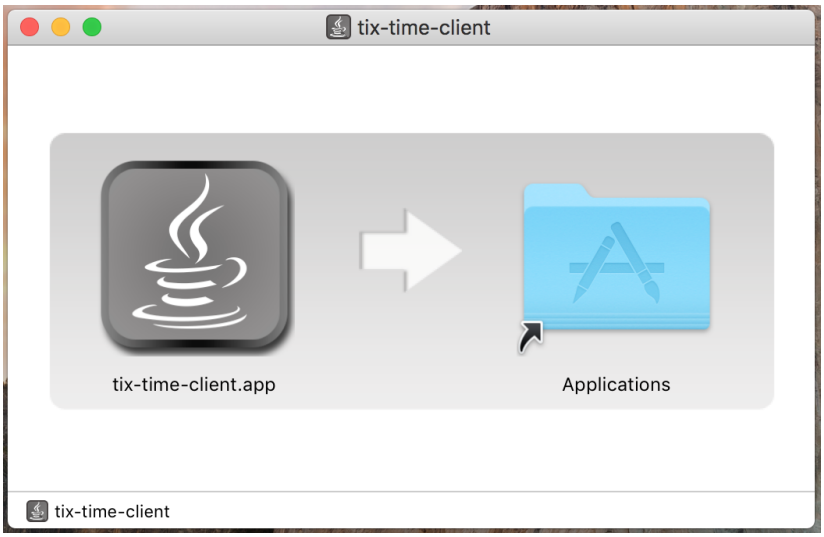

Figura 8: Instalador de la aplicación para OSX.

El cliente admite el uso de dos interfaces alternativas: una interfaz por línea de comandos (CLI) y una interfaz gráfica de usuario (GUI). La primera ha sido diseñada a modo de prueba y admite parámetros al correr el cliente - tales como nombre de usuario e instalación - para evitar el uso de un asistente que entorpezca el proceso de automatización. En cambio, la interfaz gráfica no contempla el uso de parámetros y será presentada a quienes usen el instalador de forma predeterminada.

Para la interfaz gráfica se hace uso de JavaFX [45], la plataforma gráfica de Java que ha remplazado a Swing [46] como el nuevo estándar hace varios años. El aspecto del programa resultante no equivale al de una aplicación de código nativo, pero de todas maneras es altamente customizable en apariencia. Las distintas ventanas del cliente fueron definidas mediante FXML [47], un lenguaje *markup* basado en XML pensado para interfaces de JavaFX.

Luego de instalar el cliente, el usuario que opte por el modo GUI (comportamiento predeterminado) debe ingresar mediante una interfaz visual las credenciales creadas previamente en el sitio web y definir el nombre de su nueva instalación. Tanto la autenticación como la creación de la instalación se realizan mediante el uso de forma segura de la API REST de TiX. La instalación consiste de un asistente de tres pasos en una ventana flotante de la aplicación. Con una interfaz simple y amigable, guía a los usuarios para que en menos de un minuto cuenten con el cliente ya funcionando en sus computadoras.

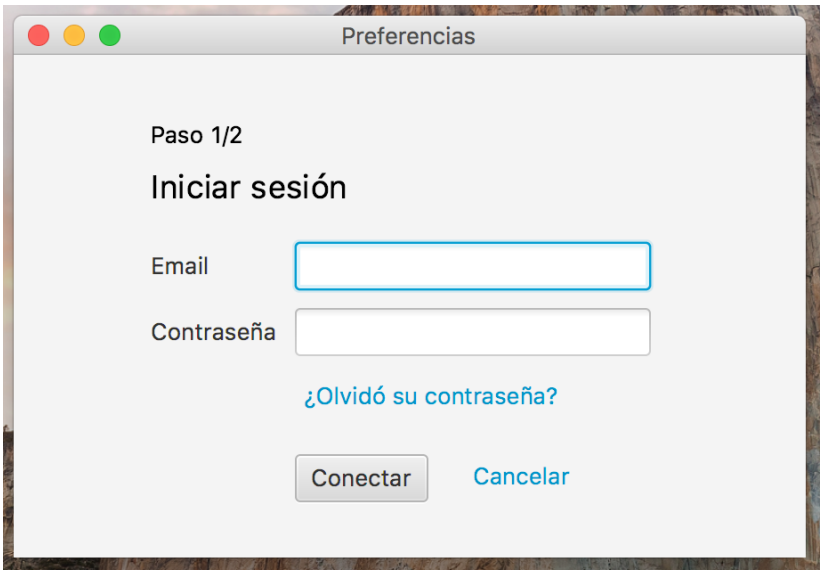

Figura 9: Ingreso de email y contraseña en cliente de OSX.

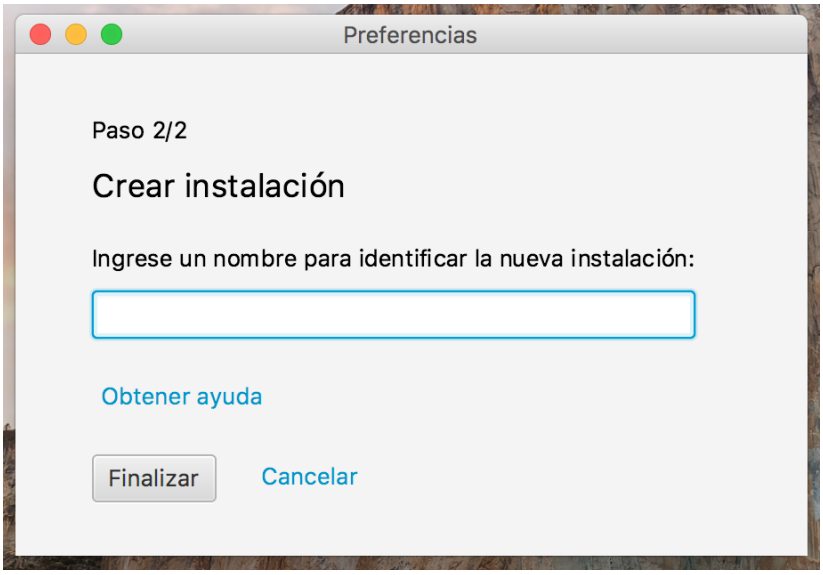

Figura 10: Ingreso de nombre de instalación en cliente de OSX.

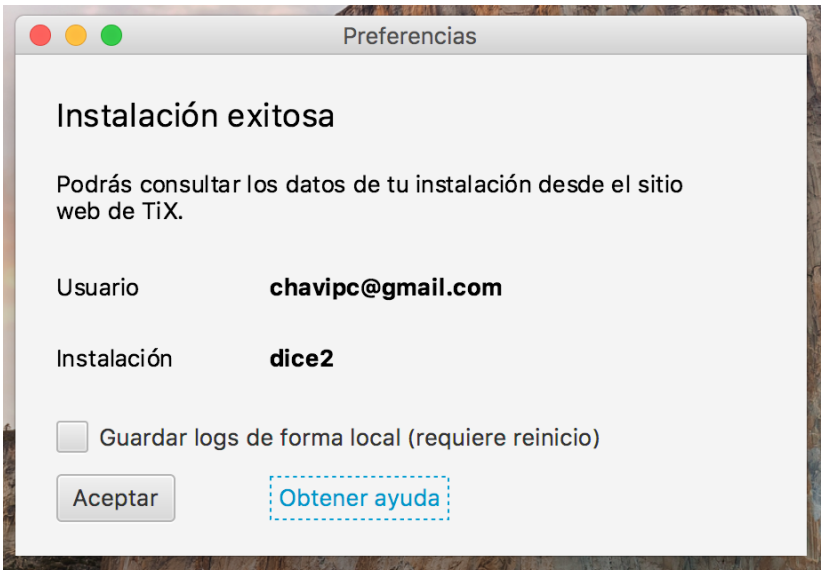

Figura 11: Fin de configuración de cliente de OSX.

Los datos de inicio de sesión y nombre de instalación son almacenados en la máquina de cada usuario. Se hace uso de la Java Preferences API [48] para almacenar las preferencias de forma eficiente según cada sistema operativo.

Una vez finalizada la configuración inicial, la aplicación queda en ejecución y comienza con las tareas de comunicarse con el servidor y realizar mediciones de tiempo. De aquí en más no es mandatoria la interacción de parte del usuario. En la barra de tareas del sistema aparece un ícono con el logo de TiX que permite acceder al menú de opciones, en caso de desearlo.

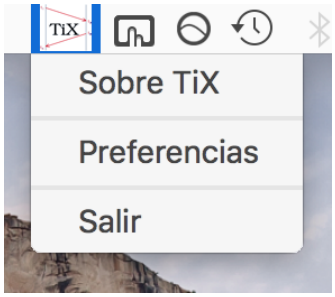

Figura 12: Menú de opciones del cliente en OSX.

La primer opción "Sobre TiX" abre una ventana que presenta información básica sobre el proyecto y provee una forma rápida de llegar a los reportes del usuario.

Figura 13: Ventana "Sobre TiX" en cliente de OSX.

La segunda opción "Preferencias" informa el nombre de usuario e instalación asociadas a la computadora, y permite almacenar registros de tiempos de forma local.

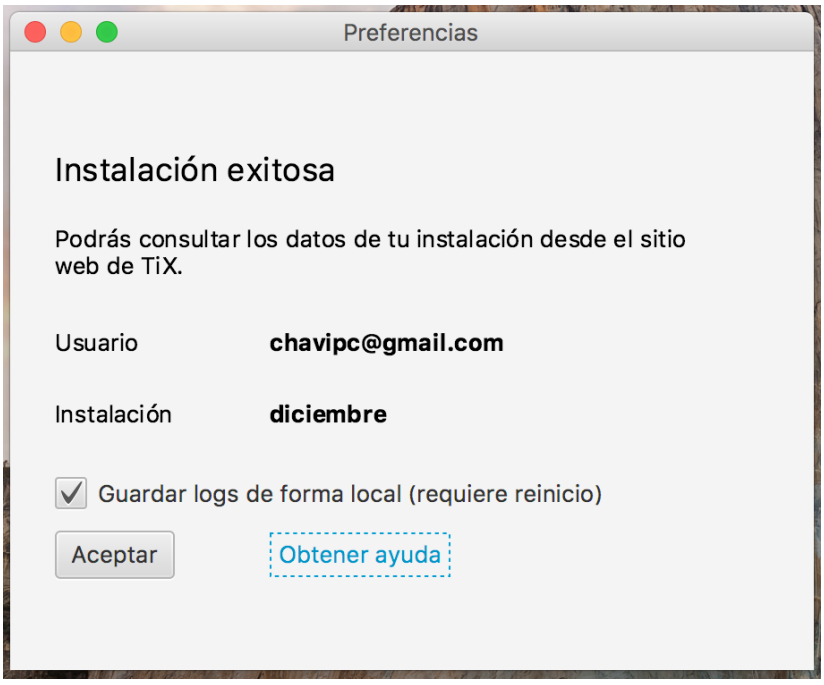

Figura 14: Ventana "Preferencias" en cliente de OSX.

La última opción "Salir" interrumpe las mediciones y la comunicación con el servidor, remueve el ícono de la barra de tareas y finaliza la ejecución del programa.

Con el objetivo de asegurar un rendimiento óptimo del cliente, las clases del código que no están relacionadas con la capa de presentación forman parte de un servicio que es independiente del hilo principal de la aplicación. Esto permite que, por ejemplo, la interfaz gráfica no se congele en ningún momento de su ejecución.

Cabe destacar el uso del *plugin* FXLauncher [49] para ofrecer actualizaciones automáticas para el usuario. Cada vez que se inicia el programa (por ejemplo, al encender la computadora), se compara la versión instalada con el último lanzamiento disponible en el servidor. En caso de existir una diferencia, se descarga la versión más reciente de la aplicación principal junto con las dependencias que hayan cambiado, sin necesitar de un accionar manual. De este modo, quien quiera mantenerse al día no tiene necesidad de acceder a la página web y descargar el programa completo, y por otro lado disminuye la probabilidad de tener clientes corriendo versiones descontinuadas.

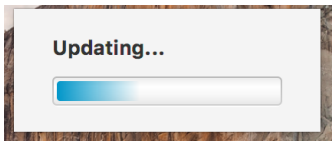

Figura 15: Búsqueda automática de actualizaciones en cliente de OSX.

3.3.2. Reporter

La tarea principal del cliente es intercambiar paquetes con el servidor y a partir de esto obtener mediciones de tiempo que reflejen la duración de estos intercambios.

La clase Reporter se dedica exclusivamente a conectarse con el servidor, mandar y recibir los paquetes, y persistirlos localmente según necesidad.

La primer tarea del Reporter es obtener la interfaz de red correcta y entablar un canal de comunicación con el servidor. Una vez que cuenta con la conexión, se hace uso de la clase auxiliar TixUdpClientHandler para recibir información externa y persistirla de forma temporal en un archivo local, en forma de bytes.

Los intercambios puntualmente consisten en envíos de pequeños paquetes UDP una vez por segundo, que no tienen ningún contenido definido. Se toma nota de las cuatro marcas de tiempo de envío y recepción (ver detalles en la Sección 3.1) para luego calcular la diferencia de tiempo entre ellos. Las marcas de tiempo se persisten temporalmente hasta llegar a tener un minuto completo de información.

Al finalizar el minuto, se envía un paquete "largo" al servidor con un contenido *(payload)* definido. Aquí se incluyen todas las mediciones de tiempo para luego ser procesadas. Existe un algoritmo cuya responsabilidad es confirmar que esta información llegó correctamente al servidor, y en caso contrario realiza hasta 5 reintentos de envío de mediciones. Apenas se recibe la confirmación de recepción exitosa, se detienen los reintentos.

Existe la opción de persistir los timestamps localmente, de modo permanente. En la ventana de Preferencias, al tildar la caja de "Guardar logs de forma local", se generará a partir de ese instante un archivo por minuto con la información en bytes. Se usará el directorio /tix-client-logs dentro de la carpeta *home* de cada usuario.

En cuanto a tecnologías utilizadas, la clase Reporter emplea Netty [26] para el envío y recepción de paquetes. Netty es un *framework* cliente-servidor para Java que es asincrónico y no bloqueante.

3.4. El Subsistema de Presentación y Administración de Datos

El subsistema de presentación de datos se puede dividir en dos servicios, los cuales funcionan de manera totalmente autónoma, el sistema encargado de guardar los datos y entregarlos a quien lo solicite (la API) y la web, cuyo fin es presentar los datos de forma concisa al usuario. Los repositorios para dichos componentes son https://github.com/TiX-measurements/tix-api y https://github.com/TiX-measurements/tix-web, respectivamente.

La web fue desarrollada principalmente en React.js[22], pero utiliza otras bibliotecas, tales como Material-ui[50], Redux[51] y Redux-forms[52] con el objetivo de minimizar la cantidad de código escrito y presentar la información de manera consistente.

React.js es una biblioteca de código abierto bajo la licencia MIT, la cual fue desarrollada por Facebook INC en el año 2013 para el desarrollo de aplicaciones frontend que manejan grandes volúmenes de información, de manera que no fuera necesaria la recarga del sitio. Asimismo busca generar simplicidad, escalabilidad y velocidad a la aplicación que se está desarrollando. Cabe destacar que React solo se encarga de generar la visualización del sitio.

Luego del lanzamiento de ReactJs, la comunidad comenzó a desarrollar diversas bibliotecas para compensar la falta de modelos y almacenamiento para la aplicación, es as como en 2015 un grupo de desarrolladores creó Redux, un contenedor del estado de la aplicación altamente predecible que ayuda a mantener el estado de la aplicación consistente durante el tiempo, ya que utiliza una única fuente de verdad (*source of truth*).

Para el desarrollo de la plataforma web, se utilizó un código base creado por la comunidad, React-boilerplate, el uso del mismo garantiza la utilización de las últimas tecnologías y estándares, así como también ayudar a mantener la calidad del código y bajar el el costo inicial de creación del aplicativo. Este codigo también se encuentra bajo licencia MIT, cuenta con más de 100 contribuyentes y cerca de 1000 commits.

Estos componentes conforman la web a la cual se puede acceder mediante la URL https://tix.innova-red.net.net

Por otro lado, la API se encuentra desarrollada principalmente en NodeJs[53], y diversas librerias basadas en Javascript, tales como ExpressJs[54], PassportJs[55], Knex[56], Bookshelf[57] y RamdaJs[58]. Finalmente, esta parte del proyecto tiene como dependencia IP2AS, el cual dada una IP, se encarga de resolver el AS que le corresponde

NodeJs es un entorno de ejecución de código abierto (bajo licencia MIT), asincrónico, con E/S de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google[59]. Dada su estructura, es especialmente útil para

micro-arquitecturas como la propuesta para este proyecto. Su manejador de paquetes, npm[60], es considerado el ecosistema más grande de bibliotecas open source existente hoy en dia.

Una de las partes más importantes del servicio es el framework ExpressJs, este se describe como un framework web minimalista, que permite con mucha facilidad agregar nuevas rutas y modificar las existentes. Se desarrollo bajo licencia MIT y cuenta con más de 5000 commits y 200 contribuyentes.

Por otro lado, el servidor utiliza MySQL para guardar/hacer queries sobre la información, se decidió utilizar un sistema basado en SQL, ya que se busca priorizar la consistencia de los datos. Para conectar nuestro servidor a la base de datos, se utilizaron dos bibliotecas, Knex, un framework para la generación de queries de forma simple y Bookshelf.js, un ORM que corre sobre Knex, ambos de código abierto bajo licencia MIT.

Finalmente, se utilizó PassportJs para realizar la autenticación de los usuarios. Este framework utiliza el concepto de «estrategias» para cumplir su rol. Lo que permitiría, por ejemplo, implementar un método de autenticación vía Facebook simplemente agregando una nueva «estrategia».

La conjunción de estos frameworks/bibliotecas conforman la API.

3.4.1. Despliegue y puesta en producción

De la misma manera que el Subsistema de Ingesta y Procesamiento, la API utiliza el sistema de docker-compose para su proceso de despliegue, bajo este compose, se encuentra también la base de datos (MySQL) e IP2AS. De esta forma, una vez que se corre el comando de ejecución, la API ya cuenta con todas las dependencias que necesita para correr correctamente.

Por otro lado, el despliegue de la web se realiza mediante TravisCI, una vez finalizada la etapa do construcción del paquete, en donde se compila y minifica el código, este los copia a la carpeta correspondiente para su ejecución.

En el repositorio https://github.com/TiX-measurements/tix-api-deploy se encuentra un archivo YAML como ejemplo para desplegar la API.

3.4.2 Definición de la API

Como fue mencionado anteriormente, la API expone una interfaz en la cual cualquier cliente se puede conectar con facilidad y acceder a los datos del sistema, esta se encuentra bajo la URL: https://tix.innova-red.net/api y expone 4 contratos:

Contrato de Usuario:

- username: Nombre de usuario

- role: Rol que cumple el usuario, puede ser usuario o administrador
- id: ID del usuario
- enabled: Boolean que indica si el usuario está activo o no.

Contrato de Instalación:

- id: ID de la instalación
- name: Nombre de la instalación
- publickey: Clave pública correspondiente a la instalación
- providers: (opcional) Lista de proveedores asociados a la instalación

Contrato de Proveedores:

- id: ID del proveedor
- name: Nombre del proveedor

Contrato de Métrica:

- upUsage: Métrica de uso de la coneccion de subida
- downUsage: Métrica de uso de la coneccion de bajada
- upQuality: Métrica de calidad de la coneccion de subida
- downQuality: Métrica de calidad de la coneccion de bajada
- timestamp: Hora del reporte
- location_id: ID de la ubicación donde fue generado el reporte
- provider_id: ID del proveedor donde fue generado el reporte
- user_id: ID del usuario que genero el reporte

También expone los siguientes métodos:
Método: GET /api

- Uso: Conocer si el servicio está corriendo actualmente o no
- Respuesta esperada: 200 OK

Método: POST /api/register

- Uso: Permite al cliente generar un nuevo usuario dentro del sistema.
- Cuerpo requerido:
  - captcharesponse: string que devuelve el reto de Re-Captcha
  - username: Nombre de usuario que se quiere crear
  - password1: Contraseña del usuario
  - password2: verificación de la contraseña
- Respuesta esperada: 200 OK con el contrato del usuario recién creado.

Método: POST /api/login

- Uso: Permite al cliente generar un token para autentificarse con la plataforma
- Cuerpo requerido:
  - username: Nombre de usuario
  - password: Contraseña del usuario
- Respuesta esperada:
  - token: JWT token utilizado para autenticar al usuario.
  - El resto del contrato de usuario.

Método: POST /api/recover

- Uso: Permite al cliente generar el token de recupero de contraseña el cual es enviado al email del cliente.
- Cuerpo requerido:
  - email: Email del usuario a recuperar
- Respuesta esperada: 200 OK

Método: POST /api/recover/code

- Uso: Permite al usuario modificar su contraseña junto al token enviado anteriormente.
- Cuerpo requerido:
  - email: Email del usuario a modificar
  - code: Código de verificación enviado en el email anteriormente
  - password: Contraseña a setear en el usuario seleccionado

- Respuesta esperada: 200 OK

Método: GET /api/user/:id

- Uso: Devuelve la información básica del usuario.
- Parámetros:
  - id: ID del usuario
- Respuesta esperada: 200 OK con el contrato del usuario

Método: PUT /api/user/:id

- Uso: Editar la información del usuario
- Parámetros:
  - id: ID del usuario
- Respuesta esperada: 200 OK con el contrato del usuario

Método: GET /api/user/:id/installation

- Uso: Devuelve la lista de instalaciones pertenecientes al usuario
- Parámetros:
  - id: ID del usuario
- Respuesta esperada: 200 OK con una lista de contratos de instalaciones.

Método: GET /api/user/:id/installation/:installationId

- Uso: Devuelve una instalación en particular
- Parámetros:
  - id: ID del usuario
  - installationId: ID de la instalación deseada
- Respuesta esperada: 200 OK con el contrato de la instalación

Método: PUT /api/user/:id/installation/:installationId

- Uso: Editar una instalación
- Parámetros:

        - id: ID del usuario
        - installationId: ID de la instalación deseada
    - Cuerpo requerido:
        - name: Nombre de la instalación
    - Respuesta: 200 OK con el contrato de la nueva instalación

Método: DELETE /api/user/:id/installation/:installationId

- Uso: Borrar la instalación deseada
- Parámetros:
    - id: ID del usuario
    - installationId: ID de la instalación deseada
- Respuesta esperada: 200 OK

Método: GET /api/user/:id/provider

- Uso: Devuelve la lista de proveedores asociados al usuario
- Parámetros:
    - id: ID del usuario
- Respuesta esperada: 200 OK con una lista de contratos de proveedores

Método: GET /api/user/:id/provider/:providerId

- Uso: Devuelve el proveedor deseado
- Parámetros:
    - id: ID del usuario
    - providerId: ID del proovedor
- Respuesta esperada: 200 OK con el contrato de proveedor

Método: GET /api/users/:id/reports

- Uso: Devuelve los reportes asociados al usuario
- Parámetros:
    - id: ID del usuario

- Respuesta esperada: 200 OK con una lista de contratos de reportes

Método: POST /api/user/:id/installation/:installationId/reports

- Uso: Crear un nuevo reporte
- Parámetros:
  - id: ID del usuario
  - installationId: ID de la instalación
- Cuerpo requerido:
  - ip: IP donde se generó el reporte
  - upUsage: Metrica de utilizacion de subida
  - downUsage: Metrica de utilizacion de bajada
  - upQuality: Metrica de calidad de subida
  - downQuality: Metrica de calidad de bajada
  - timestamp: Momento en el que se genero el reporte
- Respuesta esperada: 200 OK con el contrato de reporte

Método: GET /api/admin/users

- Uso: Devuelve todos los usuarios actuales en el sistema
- Respuesta esperada: 200 OK con una lista de contratos de usuario

Método: GET /api/admin/reports

- Uso: Devuelve todos los reportes según el criterio establecido
- Parámetros opcionales:
  - startDate: Fecha de inicio
  - endDate: Fecha de fin
  - providerId: ID del proveedor
- Respuesta esperada: 200 OK con una lista de contratos de reportes.

4. Ensayos

4.1. Prueba de Sistema

4.1.1. Objetivos

La Prueba de Sistema está apuntada a corroborar el correcto funcionamiento del Servicio de TiX en su totalidad en condiciones normales. Esto significa, que se debe probar de punta a punta el Sistema, con una carga normal, para una persona determinada. La idea de esto es asegurar una calidad mínima de servicio en base a parámetros preestablecidos.

4.1.2. Indicadores propuestos

Las métricas sobre las cuales se va a determinar que el sistema funciona correctamente son:

1. Tiempo de funcionamiento, entendiéndose como la cantidad de tiempo que estuvo vivo el sistema durante la prueba, para lo cual pudo recibir mediciones del Subsistema Cliente y mostrar el progreso de la prueba a través del Subsistema de Presentación.
2. Tiempo medio de demora en obtenerse una métrica, entendiéndose como el promedio de los tiempos que tarda el Subsistema de Procesamiento en entregar el resultado de un conjunto de mediciones más el Subsistema de Presentación en mostrarlo.
3. La media del valor absoluto del error de la métrica de Uso de Ancho de Banda, entendiéndose como el promedio de los valores absolutos de las diferencias entre el valor mostrado y el valor real para cada punto de la métrica calculado durante la prueba.

4.1.3. Consideraciones

Para esta prueba se tienen lo siguientes considerandos y supuestos:

- Se considera que el sistema es robusto y que el Tiempo de funcionamiento a lo largo de la prueba es una muestra lo suficientemente representativa del mismo en todo momento.
- Se considera que la prueba es lo suficientemente compleja para que su automatización total no sea relevante ni un punto a considerar para la presente disertación.
- Se considera una métrica aquella que puede ser visualizada a través del servicio tix-web del Subsistema de Presentación y Administración.

- Se considera negligible el efecto que tiene en el Uso de Ancho de Banda la carga de páginas web, la descarga esporádica de archivos con tamaño inferior a 1 MB y el uso en segundo plano de la conexión por parte de servicios como los son las redes sociales (e.g.: Facebook, Instagram, Twitter), medios de comunicación instantánea (e.g.: Slack, Whatsapp, Discord), portales web y agentes de correo electróncio (e.g.: GMail, Outlook, Outlook Express, Thunderbird).
- No se considera negligible el efecto que tiene en el Uso de Ancho de Banda la descarga constante de archivos o de archivos con tamaño superior a 1 MB, el uso de servicios o portales web de flujo de datos, sea de video (e.g.: YouTube, Netflix), audio (e.g.: iTunes, Spotify), u otro tipos de aplicaciones en tiempo real (i.e.: video juegos on-line, proyectos de cómputo distribuido, etc).
- Si bien en general los Proveedores de Servicio de Internet no entregan velocidades simétricas, lo que podría suponer una diferencia en la calidad de las conexiones de subida y bajada, se considera que:
  - Como método para medir el Uso de Ancho de Banda es el mismo tanto para subida como para la bajada.
  - Y que el método es robusto a esto ante la diferencia de calidad.
- Por el considerando anterior, sólo se medirá el Uso de Ancho de Banda de bajada.
- No se medirá la Calidad de la conexión debido a los muchos factores externos que juegan con ella.
- Se considera que una vez insertada la métrica en la base de datos del Subsistema de Presentación, está lista para ser mostrada. Con lo que si no puede ser mostrada es que está caído el Subsistema de Presentación en sí.

4.1.4. Metodología

Se crea un usuario en el Sistema TiX a través del Subsistema de Presentación, por medio de la página web. Luego se descarga y se instala el cliente.

Se instala un cliente de bittorrent en la computadora que va a realizar la prueba. La idea es saturar el enlace de forma programada para así poder tener valores de medición contra los cuales comparar. A su vez se debe asegurar que durante el periodo de la prueba ningún otro servicio salvo el cliente de bittorrent, el cliente de TiX y aquellos considerados negligibles para la saturación del ancho de banda, mantengan conexiones con internet. Durante las pruebas se utilizó el cliente Vuze [61]. Este permite programar de forma muy granular las velocidades máximas de transferencia de carga y descarga. Para la gran mayoría de las pruebas se utilizó la configuración del programador de velocidades de Vuze que se muestran en el Cuadro 1.

```
daily pause_all from 00:00 to 01:00 # No downloads
daily unlimited from 01:00 to 02:00 # Download at 100%
daily 80_pct_speed from 02:00 to 03:00 # Download at 80%
daily 75_pct_speed from 03:00 to 04:00 # Download at 75%
daily half_speed from 04:00 to 05:00 Download at 50%
daily 25_pct_speed from 05:00 to 06:00 # Download at 25%
daily pause_all from 06:00 to 23:59 # No downloads
```

Cuadro 1: Programación de las velocidades de descarga de Vuze

Para saturar la conexión se usaron los torrents de varias distribuciones de linux, ya que esto asegura descargas de gran tamaño por el tiempo prolongado que dura la prueba. Una vez iniciado el cliente de bittorrent, se procede a iniciar el cliente de TiX y se los deja corriendo durante todo el tiempo de la prueba.

Tras la finalización de la mismas se ejecutan las consultas que se encuentran en el repositorio de la prueba de sistema sobre la base de datos del Subsistema de Presentación. Estas arrojan los resultados de las métricas que nos interesan tener para verificar el éxito de la prueba.

El código para poder replicar parte de esta prueba junto con instrucciones, se encuentra en el repositorio https://github.com/TiX-measurements/tix-time-system-test.

4.1.5. Resultados

Tras correr algunas pruebas los resultados obtenidos fueron.

- Tiempo de funcionamiento: 95.0 %
- Tiempo medio de demora en obtenerse una métrica: $2' 30''$
- Media del error total de la métrica de Uso de Ancho de Banda: 0.12

Observaciones

- Aunque el tiempo de funcionamiento es menor al esperado, hay que tener en cuenta que se usa una única locación al momento de hacer las métricas, lo cual puede confundir el problema de un funcionamiento subpar con una caída del servicio desde el lado del cliente. Ejemplo de esto son las imágenes 16 y 18, donde para experimentos similares existen muchos menos puntos de ciertos periodos de la prueba.
- El error medio relativo puede dar más que el esperado debido a que el método utilizado, saturación mediante la descarga de torrents. La saturación del canal en este caso, no depende de la máquina desde donde se están haciendo las pruebas, sino de los pares y las semillas del torrent en sí.

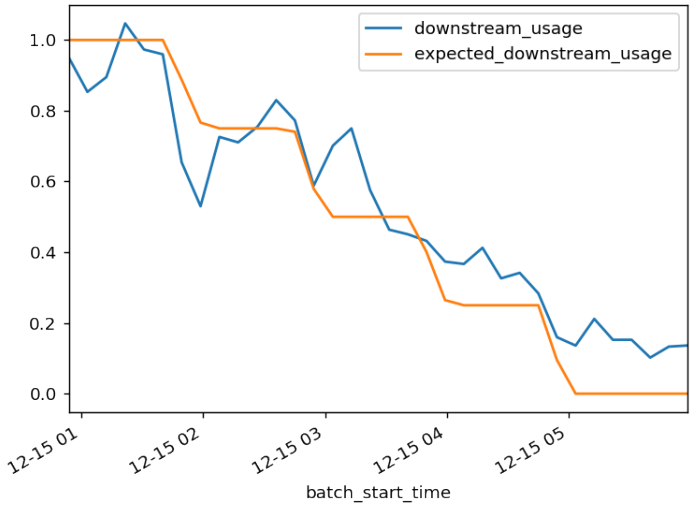

Figura 16: Medición contra valor esperado

- Como se puede apreciar desde las imágenes 16, 18, el error no es homogéneo en todas las mediciones. Hay mayor error cuanto menor es el uso de la conexión. Esto se hace evidente también en la figura 17, que además muestra que los puntos de mayor error son aquellos donde existen saltos bruscos de uso de datos.

4.2. Prueba de Carga

4.2.1. Objetivos

La Prueba de Carga está apuntada a medir la carga máxima del Sistema TiX y encontrar los cuellos de botella del mismo. Además, sirve como experimento para entender cómo es que se comporta un Sistema Complejo como esto y cuales son sus posibles mejoras.

Debido a la dinámica operativa de TiX, el servicio genera un gran nivel de carga, la cual, si bien es constante, es mucho más alta de lo que suele recibir un servidor web con la misma cantidad de usuarios. En un escenario ideal, el servicio tendría que ejecutarse en un espacio el cual permita el libre escalamiento de recursos, para cubrir eventuales picos de tráfico e infinitos usuarios. Lamentablemente esto es financieramente inviable, por lo que es necesario conocer con gran precisión cuál es el máximo número de usuarios que la plataforma puede soportar bajo la infraestructura en la cual se encuentra corriendo el servicio.

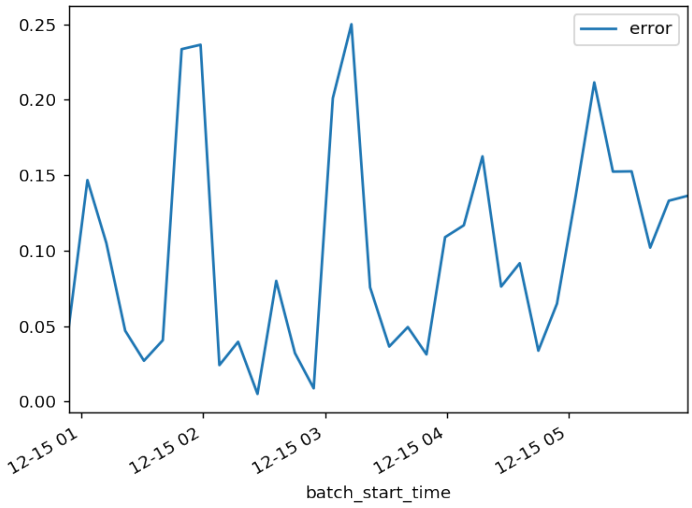

Figura 17: Error de la medición de la Figura 16

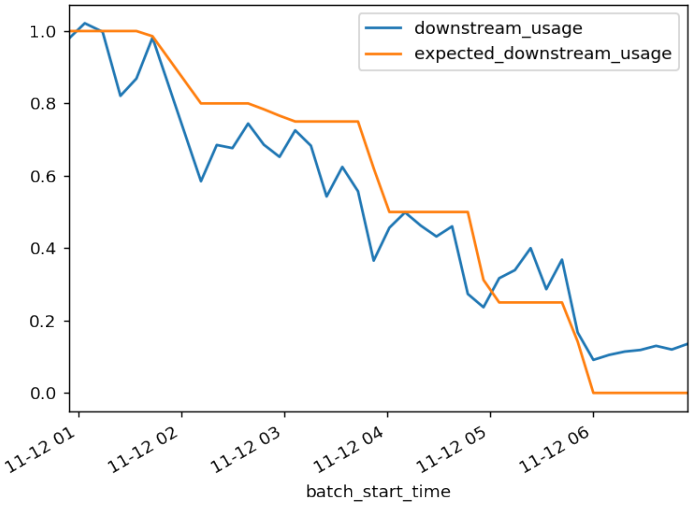

Figura 18: Otra medición contra su valor esperado

4.2.2. Indicadores propuestos

Para esta Prueba se considera una única

1. Cantidad máxima de usuarios con el sistema saturado.

Si bien existen muchas otras métricas que podrían resultar interesantes, como tiempo de funcionamiento para el sistema saturado, o degradación de la calidad del servicio con el sistema saturado, se entiende que el objetivo del presente Proyecto no es hacer una auditoría exhaustiva del Sistema, sino asegurar métricas mínimas de funcionamiento.

4.2.3. Consideraciones

Para esta prueba se tienen los siguientes considerandos y supuestos:

- La infraestructura sobre la que corre el sistema está compuesta por un servidor con las especificaciones descritas en el Cuadro 2.
- Dado el funcionamiento del Subsistema de Ingesta y Procesamiento, se considera que para probar la carga del sistema no es necesario usar múltiples usuarios, pero sí múltiples instalaciones.
- Se considera que para probar la carga del sistema, los resultados de las métricas no influyen. Es decir, el procesamiento de las mediciones y los resultados que estos arrojen no cambian el resultado de la Prueba de Carga. Por lo que los clientes pueden ser desplegados en cualquier infraestructura, desde cualquier lugar del mundo.

4.2.4. Metodología

La metodología para realizar la prueba de carga consta de dos partes: por un lado la generación de clientes para realizar la prueba, y por el otro el análisis de los resultados para saber si la prueba fue exitosa.

Las configuraciones y algunos de los requerimientos básicos para correr esta prueba, junto con las instrucciones para hacerla, se encuentran en el repositorio https://github.com/TiX-measurements/tix-time-stress-test.

Generación de Clientes

Para la Generación de clientes, se utiliza una versión modificada del cliente actual, la cual cumple el mismo rol que el cliente estándar (autenticarse al servicio, crear una instalación y generar reportes). Esto permite correr múltiples instancias del cliente en el mismo dispositivo, con un consumo de recursos menor.

```
System:
    Host: tix.innova-red.net
    Kernel: 4.4.0-83-generic x86_64 (64 bit gcc: 5.4.0)
    Console: tty 1
    Distro: Ubuntu 16.04 xenial
Machine:
    System: Dell product: PowerEdge R210 II serial: BQK2TW1
    Mobo: Dell model: 03X6X0 v: A02 serial: ..CN708212BR2HYQ.
    Bios: Dell v: 2.2.3 date: 10/25/2012
CPU:
    Quad core Intel Xeon E31220 (-HT-MCP-) cache: 8192 KB flags:
(lm nx sse sse2 sse3 sse4_1 sse4_2 ssse3 vmx) bmips: 24744
    clock speeds: max: 3400 MHz 1: 1652 MHz 2: 1600 MHz 3: 1686
MHz 4: 1600 MHz
Graphics:
    Card: Matrox Systems MGA G200eW WPCM450 bus-ID: 03:03.0
    Display Server: N/A driver: N/A tty size: 237x56
    Advanced Data: N/A for root out of X
Network:
    Card-1: Broadcom NetXtreme II BCM5716 Gigabit Ethernet
driver: bnx2 v: 2.2.6 bus-ID: 02:00.0
        IF: eno1 state: up speed: 1000 Mbps duplex: full mac:
d4:ae:52:c7:83:53
    Card-2: Broadcom NetXtreme II BCM5716 Gigabit Ethernet
driver: bnx2 v: 2.2.6 bus-ID: 02:00.1
        IF: eno2 state: down mac: d4:ae:52:c7:83:54
Drives:
    HDD Total Size: 499.6GB (9.8% used) ID-1: /dev/sda model:
    Virtual_Disk size: 499.6GB temp: 0C
Partition:
    ID-1: / size: 19G used: 13G (75%) fs: ext4 dev: /dev/sda7
    ID-2: swap-1 size: 4.09GB used: 0.87GB (21%) fs: swap dev:
/dev/sda6
RAID: No RAID devices: /proc/mdstat, md_mod kernel module
present
Sensors:
    System Temperatures: cpu: 29.8C mobo: N/A
    Fan Speeds (in rpm): cpu: N/A
Info:
    Processes: 203
    Uptime: 144 days
    Memory: 1166.0/3921.2MB
    Init: systemd runlevel: 5 Gcc sys: 5.4.0
    Client: Shell (bash 4.3.481) inxi: 2.2.35
```

Cuadro 2: Especificaciones del Servidor de acuerdo a la salida del script inxi [62]

Por otro lado se utiliza JMeter [63], una herramienta especializada en generación de pruebas de carga. Esta permite crear múltiples instancias del cliente, las cuales son ejecutadas en simultáneo y genera un reporte basado en el resultado de su ejecución.

Finalmente, se necesita un servidor para poder correr la prueba de stress. Dada la carga generada y la necesidad de mantener el servicio en funcionamiento por al menos 24 horas, es recomendable que se realice en un servidor donde se pueda monitorear su funcionamiento. Aunque también puede ser realizado en cualquier PC.

Encontramos dos indicadores clave para conocer si la prueba podía ser corrida de manera exitosa:

1. Cantidad de operaciones de E/S en el cliente.
2. Estado del consumo de CPU en el servidor de los clientes.

El primero es necesario para asegurarse que la prueba está efectivamente corriendo al ratio que le fue solicitado. Por ejemplo, si se encuentra corriendo aproximadamente 1000 instancias, entonces tendrían que haber aproximadamente 1000 escrituras en archivo por segundo.

En segundo lugar, el estado de consumo de CPU es un factor clave para conocer si el servidor donde están corriendo los clientes se encuentra al límite de procesamiento. Al momento en que este se quede sin recursos, los clientes se volverán más lentos.

Si alguno de estos indicadores muestra algún problema, entonces prueba no está corriendo correctamente y sus resultados no pueden ser fiables.

Analisis de Resultados

El análisis de resultados comprende todas aquellas herramientas que pueden ser utilizadas para conocer cómo está funcionando internamente el Servicio. Esto incluye:

- Análisis de logs.
- Análisis de carga del sistema.
- Análisis del estado de las colas de procesamiento.

Siendo este último en particularmente crítico para conocer si el sistema está funcionando correctamente o se encuentra saturado.

El Análisis de logs se hace utilizando Docker Compose y su comando para visualizar logs. La principal idea de esto es detectar problemas en la y excepciones arrojadas por los servicios. También es entender el funcionamiento de cada uno de los servicios bajo estrés de manera más específica.

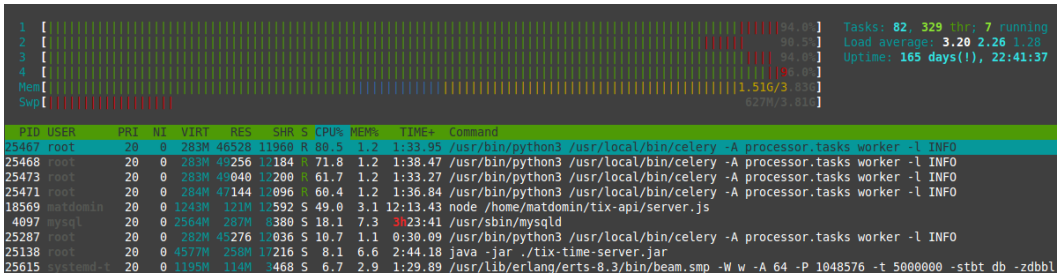

Figura 19: Captura de Pantalla mostrando la salida de HTOP al momento en que hay un pico de consumo de recursos en el Servidor del Sistema

El Análisis de carga del sistema incluye medir y monitorear el uso de recursos, como CPU, memoria y operaciones de entrada y salida. La finalidad de esto es ver cuál es el cuello de botella general del Sistema. Si es intensivo en capacidad de cómputo, en memoria o tráfico de red y operaciones de lecto-escritura. Para este análisis se usaron herramientas como htop, atop e iotop.

Finalmente, Analizar el estado de la cola de procesamiento es el factor determinante para entender el resultado de la prueba. En términos generales, dada la característica de TiX en donde el tráfico es constante, si la cantidad de paquetes que se están encolando para procesamiento es mayor que la capacidad de procesarlos, el sistema se va a ir acercando lentamente al colapso. Esto se puede visualizar analizando el gráfico que provee RabbitMQ sobre el estado de la cola.

4.2.5. Resultados

Luego de correr la prueba de carga múltiples veces, la conclusión es que el sistema, en las condiciones actuales, es capaz de soportar 2200 usuarios concurrentes.

Durante las distintas pruebas se pudo ver si bien la carga del servidor aumenta conforme aumenta la cantidad de clientes, el servidor no está todo el tiempo consumiendo al máximo sus recursos. Sino que por el contrario, que durante el tiempo de máxima carga hay momentos de actividad baja y momentos de saturación.

El momento de mayor consumo de recursos o de saturación es cuando se están procesando las métricas, como se puede ver en la figura 19. Esto se debe a que además del procesamiento, la API está recibiendo las métricas generadas e insertándolas en la base de datos.

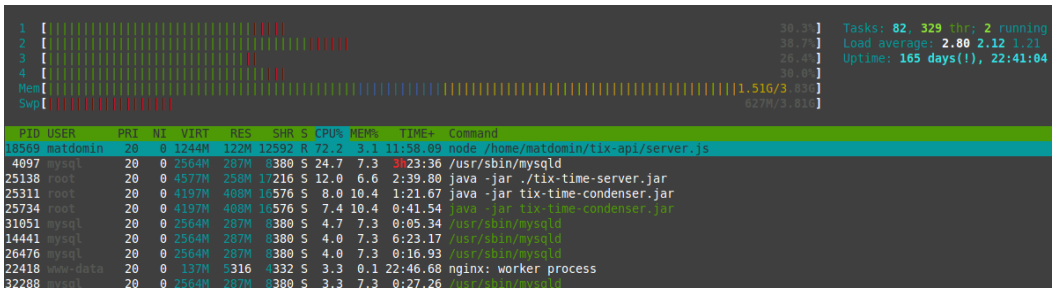

Figura 20: Captura de Pantalla mostrando la salida de HTOP al momento en que hay un consumo moderado a bajo de recursos en el Servidor del Sistema

Luego, el momento de baja actividad corresponde con el régimen regular del Sistema de Ingesta, como se muestra en la figura 20. Esto es, la toma de paquetes desde el tix-time-server, el procesamiento de los datos en el tix-time-condenser y los consultados generadas contra la tix-api.

Lo lleva a la conclusión que existen tres servicios bien diferenciados con distintos niveles de requerimientos. Por un lado la tix-api, debido a su conexión con la base de datos. Por otro el tix-time-server y el tix-time-condenser, que realizan un trabajo más rutinario y monótono con una carga constante del sistema. Y finalmente el tix-time-processor, que realiza un trabajo de procesamiento por lotes con un consumo fuerte en cortos periodos, y un consumo nulo el tiempo restante.

También, a partir de estas imágenes podemos ver que el Sistema es intensivo en Procesamiento, pero no en memoria. Ocupando en momentos de máxima saturación el 34 % de la memoria disponible con una configuración de una instancia de cada servicio, como se ve en la figura 21. Para esta configuración es que el sistema logró procesar 1900 usuarios concurrentes.

El sistema entero no satura los recursos del servidor, sino que el tiempo de respuesta para atender los mensajes que llegan en la cola no es el suficiente.

La primera cola en saturarse, como se puede ver en las imágenes 22 y 23 es la que conecta el tix-time-server con el tix-time-condenser. Agregar más instancias de este último servicio permite mejorar el rendimiento general del sistema. Pero cada instancia nueva lo mejora marginalmente. Las razones de esto no fueron profundizadas, pero las hipótesis que surgieron son:

- Un problema de configuración del mismo servicio tix-time-condenser, que no le permite tomar una mayor cantidad de mensajes de la cola.
- Un problema de configuración de alguno de los otros servicios con los que interactúa (RabbitMQ y tix-api), que no le permite escalar linealmente.

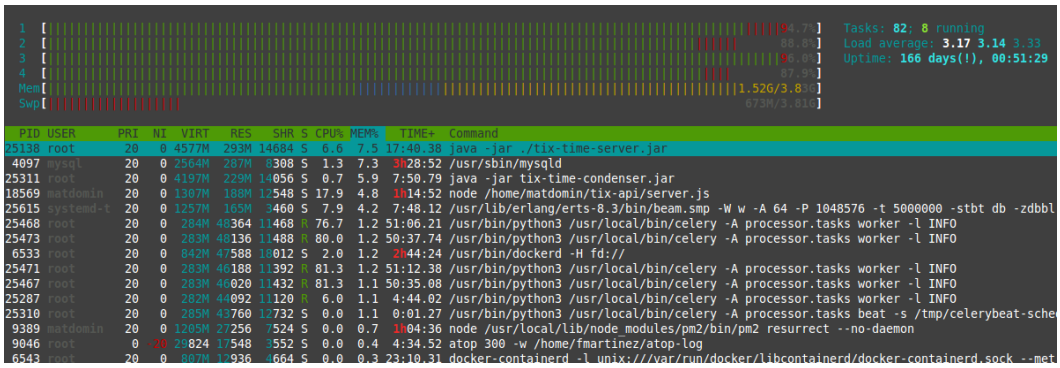

Figura 21: Captura de Pantalla mostrando la salida de HTOP al momento en que hay un pico de consumo de recursos en el Servidor del Sistema con los procesor ordenados por consumo de memoria

server	condenser	processor (BEAT)	processor (WORKER)[1]	API	Usuarios
1	1	1	1 (1 min)	1	1800
1	1	1	2 (1 min)	1	1800
1	1	1	1 (5 min)	1	1800
1	2	1	1 (5 min)	1	1900
1	3	1	1 (5 min)	1	2200

Cuadro 3: Configuraciones usadas para la prueba de carga

- Un problema de infraestructura, consecuencia directa de estar corriendo todo en un mismo servidor. Es decir, la misma infraestructura es el cuello de botella.

Para poder hacer esto se debe hacer un perfilado más específico de estos servicios con herramientas como NewRelic, Pingdom Server Monitor o DataDog.

Como se menciona, se probaron varias configuraciones con distintos resultados, mostradas en la tabla 3.

Figura 22: Panel de Control de RabbitMQ mostrando la cola que conecta el tix-time-server con el tix-time-condenser saturada

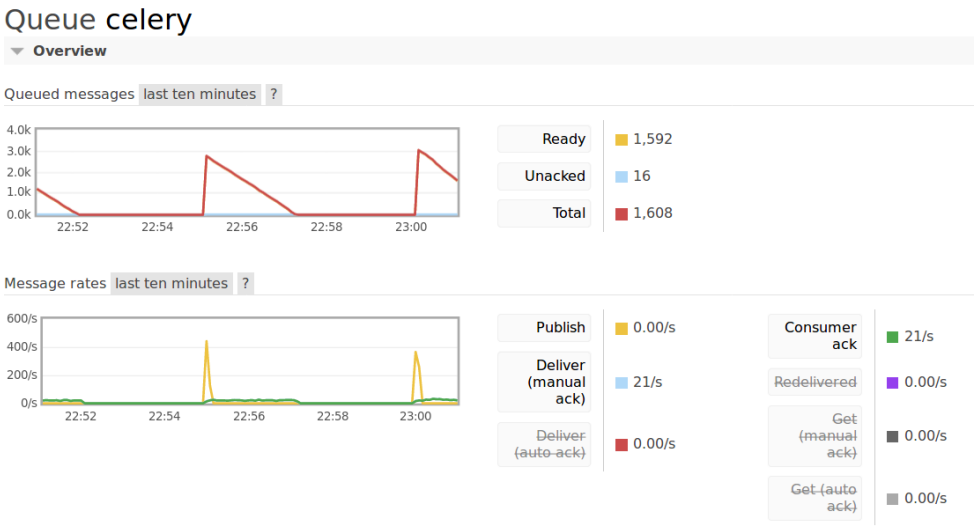

Figura 23: Panel de Control de RabbitMQ mostrando la cola que usa Celery al momento de saturación del sistema

5. Conclusiones

5.1. Aprendizajes y resultados

El proyecto TiX lleva ya unos años en desarrollo y ha tenido unos cuantos colaboradores a lo largo del tiempo. Antes de comenzar esta iteración, han pasado dos equipos de desarrolladores cuyos objetivos eran distintos, principalmente crear el sistema y sumar funcionalidades. Cada uno de ellos se concentró en resolver los problemas propuestos y lograr que el proyecto siga funcionando. Tres años más tarde, el principal desafío fue llevar adelante el desarrollo de un código que estaba en un estado más cercano a una prueba de concepto que a un código apto para ser puesto en producción. Al revisar el material disponible, faltaba documentación de varias funcionalidades (o la misma estaba desactualizada), el sistema no estaba diseñado para ser usado por más de una centena de usuarios, se observaban problemas en distintos módulos y existían incompatibilidades con ciertas versiones de sistemas operativos populares.

Tal como se comentada en la Introducción, esta iteración se centró en que TiX sea un sistema estable y escalable, permitiendo contar con cinco mil usuarios concurrentes. Alcanzar esta meta permitiría abrir el proyecto a una cantidad razonable de usuarios, siempre y cuando quede como un proyecto de uso dentro de la comunidad académica y no de uso masivo.

Gran parte del paso por el proyecto consistió en analizar el estado actual del código, del sistema en total y de la forma óptima para lograr alcanzar nuestro objetivo de escalabilidad. Luego de muchas discusiones, se acordó que era necesario llevar a cabo importantes cambios. Se replanteó el diseño del sistema, las tecnologías usadas en varios módulos, las interacciones entre ellos, etc. Todas estas modificaciones llevaron a retrabajar el alcance y los plazos que se habían acordado los primeros días de esta iteración.

Después de mucho trabajo se logró cumplir con las metas propuestas. El sistema es efectivamente escalable y se observa un funcionamiento estable con casi dos mil usuarios concurrentes. Esto está avalado por los distintos ensayos que se presentan en el capítulo dedicado a ellos.

Las pruebas de carga señalan que el sistema actualmente puede brindar servicio a unos 2000 usuarios al mismo tiempo, lo cual es un gran avance comparado con la iteración anterior. Cabe destacar que no todos los usuarios de un sistema suelen utilizarlo (correr la aplicación) al mismo tiempo - se asume que una fracción de los clientes permanecerán inactivos en cualquier instante. Esto lleva a creer que el sistema hoy en día puede funcionar con más de dos mil usuarios registrados.

El limitante actual de este número es el servidor de TiX, con lo cual habría que probar el sistema con una infraestructura más avanzada y llevar a cabo nuevas pruebas de carga. Más allá de los posibles resultados que arrojen, es importante

señalar que todas las partes del sistema han sido planteadas bajo la premisa de escalabilidad y estabilidad bajo un gran caudal de usuarios.

En cuanto a las pruebas de sistema, éstas indican que el funcionamiento completo del sistema es el esperado: tanto el Subsistema de Ingesta y Procesamiento, el Subsistema Cliente y el Subsistema de Presentación de Datos están cumpliendo debidamente con sus responsabilidades. Los indicadores observados en estas pruebas coinciden con los valores esperados. Y cualquier usuario puede comprobar por su cuenta que hoy es posible instalar el cliente en su sistema operativo, generar mediciones y analizar los resultados desde el sitio web.

Más allá del objetivo de escalabilidad, se ha trabajado en una serie de mejoras notables. La aplicación cliente es sumamente portable y ha incorporado facilidades para el usuario final tales como actualizaciones automáticas, se ha mejorado la seguridad del sistema mediante la encriptación de datos y comunicaciones entre distintos subsistemas, se presentan diversas mejoras en el Subsistema de Presentación de Datos, etc.

Otro logro importante ha sido preparar el sistema para un mantenimiento sencillo. Por ejemplo, se optó por usar Docker para agrupar partes de código en distintos componentes, y se incluyeron tests para permitir hacer cambios en código sin que peligre su funcionamiento. El resultado es un proyecto de Código Libre de calidad. Esto es de suma importancia en un proyecto de carácter académico ya que permite que las próximas iteraciones de código se hagan sin grandes sobresaltos y hasta posiblemente se sume gente externa.

Sin duda, se han puesto en práctica innumerables conocimientos adquiridos a lo largo de la carrera de Ingeniería Informática: nociones de diseño de sistemas, concurrencia, encriptado, aplicaciones web, aplicaciones de escritorio, programación orientada a objetos, protocolos de comunicación y muchos más. Pero lo más importante ha sido aplicar la capacidad de análisis crítico para concluir que sólo sería posible llegar al objetivo replanteando gran parte del trabajo hecho.

5.2. Trabajo pendiente y posibles mejoras o ampliaciones

A lo largo del proceso de diseño y desarrollo, fueron surgiendo ideas o posibles mejoras para la plataforma, que por no considerarse prioritarias para llegar al objetivo planteado, se dejaron como propuestas a ser resueltas cuando el objetivo sea distinto o se cuente con los recursos para hacerlos realidad.

Cabe destacar, que si bien el objetivo principal del proyecto era soportar hasta 5000 usuarios concurrentes, esto no pudo ser verificado bajo las condiciones actuales dado la infraestructura con la que se cuenta, pero es posible dada la arquitectura del sistema. Este proceso, generó muchas propuestas sobre cómo llegar al requerimiento con la infraestructura actual, pero la complejidad de este trabajo no justifica la necesidad de nueva infraestructura, además de no existir seguridad que dicha mejora funcione.

Finalmente, el grupo plantea las siguientes propuestas:

- Aplicación móvil con un protocolo optimizado.
- Actualizaciones automáticas del Cliente en todo momento y no sólo cuando se inicia.
- Implementar y poner en producción un conjunto de herramientas que permitan realizar el trabajo de operaciones del día a día en el Sistema TiX instalado en el servidor.
- Cambiar el diseño del protocolo para que ya no sea solamente por UDP sino que sea por UDP y HTTP. La idea es que los paquetes UDP sean de PING a un servicio especial para esto y los paquetes de reportes vayan a otro servicio que sea HTTP y sea el nuevo tix-time-server.
- Hacer una biblioteca aparte para el Analyzer y el ReportsHandler en el tix-time-processor, que estén distribuidas por PyPi para una fácil instalación. Esto también permitiría poner los notebooks de análisis en otro repositorio que no sea el del tix-time-processor.
- Mejorar el paquete tix-time-core incluyendo la parte de serialización y deserialización de paquetes no sólo en UDP, sino también en otro formatos como JSON.
- Hacer pruebas con Celery usando Redis en vez de contra RabbitMQ como soporte para analizar el impacto en la escalabilidad y la performance del servicio.
- Analizar la posibilidad de hacer el procesamiento por lotes con herramientas de análisis de grandes cantidades de datos, como lo son Apache Hadoop, Apache Spark y Presto.
- Implementar mejoras a IP2AS que permita hacer queries más eficientemente.
- Implementar cache en la API para reducir el tiempo de respuesta y no consumir datos desde la base de datos.

Referencias

[1] J. Postel, "User datagram protocol," Internet Engineering Task Force, Tech. Rep., aug 1980. [Online]. Available: https://www.ietf.org/rfc/rfc768.txt

[2] D. Mills, "Network time protocol (NTP)," Internet Engineering Task Force, Tech. Rep., sep 1985. [Online]. Available: https://tools.ietf.org/html/rfc958

[3] M. E. Crovella and A. Bestavros, "Self-similarity in world wide web traffic: evidence and possible causes," *IEEE/ACM Transactions on networking*, vol. 5, no. 6, pp. 835–846, 1997.

[4] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson, "On the self-similar nature of ethernet traffic (extended version)," *IEEE/ACM Transactions on networking*, vol. 2, no. 1, pp. 1–15, 1994.

[5] Oracle, "Java Language Documentation." [Online]. Available: https://docs.oracle.com/en/java/

[6] R. Johnson, J. Hoeller, K. Donald, C. Sampaleanu, R. Harrop, A. Arendsen, T. Risberg, D. Davison, D. Kopylenko, M. Pollack, T. Templier, E. Vervaet, P. Tung, B. Hale, A. Colyer, J. Lewis, C. Leau, M. Fisher, S. Brannen, R. Laddad, A. Poutsma, C. Beams, T. Abedrabbo, A. Clement, D. Syer, O. Gierke, and R. Stoyanchev, "Spring Framework Documentation." [Online]. Available: https://docs.spring.io/spring/docs/3.1.1.RELEASE/spring-framework-reference/html/

[7] Red Hat JBoss Middleware, "Hibernate ORM Documentation." [Online]. Available: http://hibernate.org/orm/documentation/

[8] Apache Software Foundation, "Apache Wicket Framework Documentation." [Online]. Available: https://ci.apache.org/projects/wicket/apidocs/1.5.x/index.html

[9] R. Mordani, Oracle, N. Abramson, Apache Software Foundation Art Technology Group Inc.(ATG), K. Avedal, BEA Systems, H. Bergsten, Boeing, Borland Software Corporation, Developmentor, J. Hunter, IBM, InterX PLC, R. Johnson, Lutris Technologies, New Atlanta Communications, LLC, Novell, Inc., Persistence Software, Inc., Pramati Technologies Progress Software, SAS Institute, Inc., Sun Microsystems, Inc., Sybase, and G. Wilkins, "JavaTM Servlet 2.4 Specification," Java Community Process, techreport 154, Nov. 2003. [Online]. Available: https://jcp.org/en/jsr/detail?id=154

[10] Python Software Foundation, "Python 2 Language Documentation." [Online]. Available: https://docs.python.org/2/

[11] R Development Core Team, "The R Language Manuals." [Online]. Available: https://cran.r-project.org/manuals.html

[12] J. Buck and L. Hambley, “Capistrano Documentation.” [Online]. Available: http://capistranorb.com/

[13] Y. Matsumoto, “Ruby Language Documentation.” [Online]. Available: https://www.ruby-lang.org/en/documentation/

[14] The Open Group and IEEE Std, “crontab,” The Open Group, techreport, 2001. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html

[15] R. T. Fielding, “Architectural styles and the design of network-based software architectures,” phdthesis, University of California, 2000. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[16] Travis-CI Community, “Travis-CI.” [Online]. Available: http://travis-ci.org/

[17] Docker, Inc., “Docker.” [Online]. Available: https://www.docker.com/

[18] ——, “Dockerhub.” [Online]. Available: https://hub.docker.com/

[19] ——, “Docker Compose Documentation.” [Online]. Available: https://docs.docker.com/compose/

[20] Apache Software Foundation, “Apache Maven.” [Online]. Available: https://maven.apache.org/

[21] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*, A.-W. Professional, Ed. Addison-Wesley Professional, 2014. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se8/html/index.html

[22] Facebook, Instagram, and Community, “React framework (reactjs).” [Online]. Available: https://reactjs.org/

[23] T. Koppers, S. Larkin, J. Ewald, J. Vepsäläinen, K. Kluskens, and W. contributors, “Webpack.” [Online]. Available: https://webpack.js.org/

[24] M. Jones, J. Bradley, and N. Sakimura, “JSON web token (JWT),” IETF, Tech. Rep., may 2015. [Online]. Available: https://tools.ietf.org/html/rfc7519

[25] E. R. Fielding and E. J. Reschke, “Hypertext transfer protocol (HTTP/1.1): Authentication,” IETF, Tech. Rep., jun 2014. [Online]. Available: https://tools.ietf.org/html/rfc7235

[26] Netty Project Community, “Netty.” [Online]. Available: https://netty.io/

[27] P. Webb, D. Syer, J. Long, S. Nicoll, R. Winch, A. Wilkinson, M. Overdijk, C. Dupuis, S. Deleuze, and M. Simons, “Spring Boot Documentation,” 2017. [Online]. Available: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/

[28] The Scipy Community, "NumPy Reference Guide." [Online]. Available: https://docs.scipy.org/doc/numpy-1.13.0/reference/

[29] ——, "Scipy reference guide." [Online]. Available: https://docs.scipy.org/doc/scipy-1.0.0/reference/

[30] J. D. Hunter, M. Droettboom, T. A. Caswell, and The Matplotlib Community, "Matplotlib user's guide." [Online]. Available: http://matplotlib.org/users/index.html

[31] The scikit-learn Community, "scikit-learn documentation." [Online]. Available: http://scikit-learn.org/stable/documentation.html

[32] Jupyter Team, "Jupyter documentation." [Online]. Available: https://jupyter.readthedocs.io/en/latest/contents.html

[33] A. Solem, "Celery user manual." [Online]. Available: http://docs.celeryproject.org/en/latest/

[34] Pivotal, "Rabbitmq documentation." [Online]. Available: https://www.rabbitmq.com/documentation.html

[35] J. Postel, "Transmission control protocol," DARPA Internet Program, Tech. Rep., sep 1981. [Online]. Available: https://tools.ietf.org/html/rfc793

[36] J. Nagle, "Congestion control in IP/TCP internetworks," Internet Engineering Task Force, Tech. Rep., jan 1984. [Online]. Available: https://tools.ietf.org/html/rfc896

[37] F. Yergeau, "UTF-8, a transformation format of ISO 10646," Internet Engineering Task Force, Tech. Rep., nov 2003. [Online]. Available: https://tools.ietf.org/html/rfc3629

[38] R. L. Rivest, A. Shamir, and L. M. Adleman, "Cryptographic communications system and method," United States Patent Patent 4,405,829, 1983. [Online]. Available: http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=%2Fnetahtml%2FPTO%2Fsrchnum.htm&r=1&f=G&l=50&s1=4405829.PN.&OS=PN/4405829&RS=PN/4405829

[39] M. Cooper, Y. Dzambasow, P. Hesse, S. Joseph, and R. Nicholas, "Internet x.509 public key infrastructure: Certification path building," Internet Engineering Task Force, Tech. Rep., sep 2005. [Online]. Available: https://tools.ietf.org/html/rfc4158

[40] Q. H. Dang, "Secure hash standard," National Institute Of Standards and Technology, Tech. Rep., jul 2015. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[41] B. Kaliski, "PKCS #1: RSA encryption version 1.5," Internet Engineering Task Force, Tech. Rep., mar 1998. [Online]. Available: https://tools.ietf.org/html/rfc2437

[42] S. Aiyagari, A. Richardson, M. Arrott, M. Ritchie, M. Atwell, S. Sadjadi, J. Brome, R. Schloming, A. Conway, S. Shaw, R. Godfrey, M. Sustrik, R. Greig, C. Trieloff, P. Hintjens, K. van der Riet, J. O'Hara, S. Vinoski, and M. Radestock, "Advanced message queuing protocol," , techreport, 2008. [Online]. Available: https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf

[43] T. Bray, "The JavaScript object notation (JSON) data interchange format," Internet Engineering Task Force, Tech. Rep., mar 2014. [Online]. Available: https://tools.ietf.org/html/rfc7159

[44] F. Wasilewski, "Pywavelets documentation." [Online]. Available: https://pywavelets.readthedocs.io/en/latest/#license

[45] Oracle, "Javafx." [Online]. Available: http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html

[46] ——, "Swing." [Online]. Available: https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html

[47] ——, "Fxml." [Online]. Available: https://docs.oracle.com/javafx/2/api/javafx/fxml/doc-files/introduction_to_fxml.html

[48] ——, "Java preferences api." [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/preferences/index.html

[49] E. Syse and F. contributors, "Fxlauncher." [Online]. Available: https://github.com/edvin/fxlauncher

[50] Call-Em-All, "Material-ui." [Online]. Available: http://www.material-ui.com

[51] E. Rasmussen, "Redux." [Online]. Available: https://redux.js.org/

[52] ——, "Redux-form." [Online]. Available: https://redux-form.com

[53] J. y. c. Linux Foundation, "Node.js." [Online]. Available: https://nodejs.org

[54] I. y. c. StrongLoop, "Expressjs." [Online]. Available: https://expressjs.com/

[55] A. y colaboradores, "Passportjs." [Online]. Available: http://www.passportjs.org/

[56] T. G. y colaboradores, "Knexjs." [Online]. Available: http://knexjs.org/

[57] B. y colaboradores, "Bookshelfjs." [Online]. Available: http://bookshelfjs.org/

[58] R. y colaboradores, “Ramdajs.” [Online]. Available: http://ramdajs.com/

[59] Google, “Chrome v8.” [Online]. Available: https://developers.google.com/v8/

[60] I. Z. S. y colaboradores, “Npm.” [Online]. Available: https://www.npmjs.com/

[61] Azureus Software, Inc., “Vuze bittorrent client web page.” [Online]. Available: https://www.vuze.com/

[62] H. Hope, “inxi documentation.” [Online]. Available: https://smxi.org/docs/inxi-man.htm

[63] Apache Software Foundation, “Apache JMeter Documentation.” [Online]. Available: http://jmeter.apache.org/usermanual/index.html