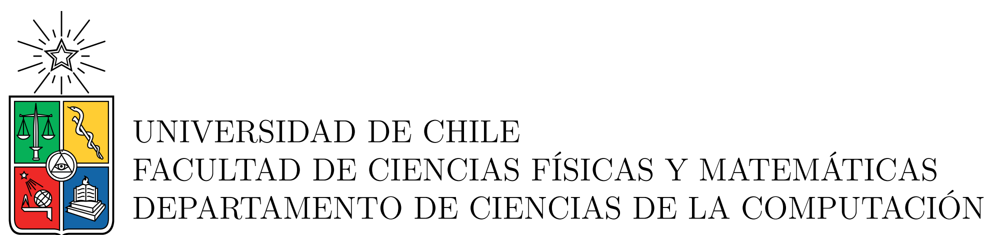


UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DESARROLLO DE UNA HERRAMIENTA DE ARBITRAJE PARA CRIPTOMONEDAS BASADO EN MOQUI

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

MATÍAS ALONSO CORDÓN ROJAS

PROFESOR GUÍA:
Andrés Muñoz Órdenes

MIEMBROS DE LA COMISIÓN:
Benjamín Bustos Cárdenas
Eduardo Riveros Roca

SANTIAGO DE CHILE
2022

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERÍA CIVIL
EN COMPUTACIÓN
POR: MATÍAS ALONSO CORDÓN ROJAS
FECHA: 2022
PROF. GUÍA: ANDRÉS MUÑOZ ÓRDENES

DESARROLLO DE UNA HERRAMIENTA DE ARBITRAJE PARA CRIPTOMONEDAS BASADO EN MOQUI

El presente trabajo tiene por objetivo describir el proceso realizado para elaborar una herramienta de arbitraje de criptomonedas, utilizando el *framework* Moqui. Para lograr el objetivo anterior, se revisan las principales características de los diferentes sistemas de arbitraje que existen, las formas de llevar a cabo su implementación, junto con identificar sus principales ventajas y desventajas.

Por otra parte, se revisarán los principales conceptos relacionados al estudio de las criptomonedas, cómo operan en los *exchangers*, sus formas de intercambio, y finalmente, como este amplio mundo de transacciones digitales se pueden comunicar con la herramienta desarrollada en Moqui.

En síntesis, la metodología utilizada para llevar a cabo el trabajo fue la siguiente: primero se comenzó con el planteamiento de los objetivos generales y específicos. Estos buscan concretar un sistema de arbitraje robusto que permita las transacciones con criptomonedas en Moqui. Seguido de esto, se planteó un estudio de otros sistemas de arbitrajes que habían disponibles en el mercado para obtener sus mejores cualidades y replicarlas en este trabajo. Posterior a ello se realizó una investigación acerca de Moqui y las tecnologías necesarias para llevar a cabo un sistema de arbitraje. Con el estudio terminado, se plantearon los diseños de la aplicación, desde arquitectura, base de datos, los programas usados y las interfaces que el usuario iba a interactuar. Una vez el diseño estaba completo, se prosiguió con la implementación. Esta comprende desde la instalación de Moqui, hasta los servicios levantados en el *framework* y otras tecnologías externas a él. Luego se realizó la evaluación del sistema, realizando pruebas de las principales transacciones ofrecidas por el sistema; compra de criptodivisa, arbitraje simple, arbitraje triangular en un mismo *exchanger* y arbitraje triangular en diversos *exchangers*. Para finalizar, se discutieron los resultados y se llegó a una conclusión.

Finalmente el trabajo abordará los principales desafíos de desarrollar esta herramienta, cuáles son sus fortalezas, así como identificar sus debilidades, para concluir si es viable o no desarrollar una herramienta exitosa que permita efectivamente a los usuarios sacar provecho de un sistema de arbitraje.

Para mi Padre, Madre y Hermano,
los pilares fundamentales de mi vida.

Agradecimientos

Me gustaría agradecer a mis Padres, que desde un principio se preocuparon por hacer de mi un hombre hecho y derecho, y siempre velaron porque nunca me faltara nada. También quiero agradecer a mi Hermano, quien es mi apoyo más grande y modelo a seguir. Gracias por todo lo que me han dado y enseñado, el fruto de esta memoria y de la persona que soy hoy en día es gracias a ustedes. Los amo.

Por último, me gustaría agradecer a todas las personas que me han dado fuerzas y apoyo para seguir este largo recorrido, gracias por confiar en mí y apoyarme siempre que lo he necesitado.

Tabla de Contenido

1. Introducción **1**
1.1. Antecedentes Generales . . . 1
1.1.1. Sistema de arbitraje . . . 2
1.1.2. Moit y el framework Moqui . . . 3
1.2. Objetivos . . . 3
1.2.1. Objetivos Generales . . . 3
1.2.2. Objetivos Específicos . . . 3

2. Preparación **5**
2.1. Definición de requerimientos . . . 5
2.2. Investigación de otros sistemas de arbitraje . . . 6
2.3. Estudio Moqui Framework . . . 6
2.4. Estudio de Tecnologías Propias del Proyecto . . . 7
2.4.1. Testnet . . . 7
2.4.2. Entorno de pruebas, sandbox y elección . . . 8

3. Diseño **10**
3.1. Arquitectura . . . 10
3.2. Base de Datos . . . 11
3.3. Programas . . . 12
3.3.1. Screens . . . 12
3.3.2. Servicios . . . 13
3.3.3. APIs . . . 14
3.3.4. Python CGI . . . 14
3.4. Interfaces . . . 15
3.4.1. Interfaces de arbitraje . . . 16
Arbitraje simple . . . 16
Arbitraje triangular múltiple . . . 17
Arbitraje triangular simple . . . 18
Comprar criptomoneda . . . 19
3.4.2. Interfaces de información . . . 20

4. Implementación **23**
4.1. Instalación inicial Moqui Framework . . . 23
4.2. Instalación de otros servicios . . . 23
4.2.1. Consideraciones con python . . . 24
4.3. Servicios y Módulos de Moqui . . . 25
4.3.1. Servicio propio de Moqui . . . 25

4.3.2. Servicios python 27

5. Evaluación **29**
5.1. Presentar Resultados 29
5.1.1. Comprar Criptodivisa 29
5.1.2. Arbitraje Simple 31
5.1.3. Arbitraje Triangular en un mismo Exchanger 35
5.1.4. Arbitraje Triangular en diversos Exchangers 38
5.2. Discusión de los resultados expuestos 41
5.3. Análisis de cumplimiento de los objetivos 42

6. Conclusión **43**
6.1. Retrospectiva 43
6.2. Trabajo Futuro y conclusiones finales 43

Bibliografía **45**

Anexo A. Apéndice Código **47**
A.1. Servicios Moqui 47
A.2. Python 48

Anexo B. Resultados **52**

Índice de Tablas

B.1. Tabla de 10 resultados compra criptomoneda. 52
B.2. Tabla de 10 resultados arbitraje simple. 52
B.3. Tabla de 10 resultados arbitraje triangular en un mismo exchanger. 53
B.4. Tabla de 10 resultados arbitraje triangular en distintos exchangers. 53

Índice de Ilustraciones

3.1. Diseño de Arquitectura de la solución 11
3.2. Entidades base de datos 12
3.3. Inicio de aplicación 15
3.4. CryptoDashboard 16
3.5. Arbitraje Simple 17
3.6. Arbitraje Triangular Múltiple 18
3.7. Arbitraje Triangular Simple 19
3.8. Comprar Crypto, vista de la interfaz una vez se intenta realizar la compra. . . 20
3.9. Pestaña Billetera. 20
3.10. Historial de transacciones. 21
3.11. Historial de transacciones realizadas en Moqui. 21
5.1. Primera compra BTC, vista en Moqui. 30
5.2. Registro historial Exchanger y base de datos. 30
5.3. Segunda compra BTC, vista en Moqui. 31
5.4. Registro historial Exchanger y base de datos. 31
5.5. Registro Moqui, arbitraje simple. 32
5.6. Registro base de datos e historial de Swyftx y Gemini respectivamente. 32
5.7. Vista segundo arbitraje simple en Moqui 33
5.8. Historial de Gemini y Coinbase respectivamente. 33
5.9. Base de datos después del segundo arbitraje simple. 33
5.10. Interfaz en Moqui después de una transacción fallida. 34
5.11. Historial exchangers, primero historial Gemini y segundo Swyftx. 34
5.12. Base de datos, después de que fallara la transacción en Gemini. 35
5.13. Vista de Moqui cuando no detecta beneficio para el usuario al arbitrar. 35
5.14. Base de datos después del intento fallido de arbitraje simple. 35
5.15. Interfaz Moqui sobre arbitraje triangular comprando ETH en un exchanger. . 36
5.16. Base de datos e historial Swyftx luego de realizar arbitraje triangular. 36
5.17. Interfaz Moqui sobre arbitraje triangular comprando BTC en un exchanger . . 37
5.18. Historial Swyftx una vez se realiza el arbitraje triangular 37
5.19. Información base de datos después de realizar arbitraje triangular en Swyftx. . 38
5.20. Interfaz de Moqui al arbitrar con éxito. 38
5.21. Historial de Gemini. 39
5.22. Historial de Swyftx. 39
5.23. Historial de Coinbase. 39
5.24. Base de datos luego de la transacción. 39
5.25. Transacción fallida en el paso final. 40
5.26. Base de datos en la venta fallida. 40
5.27. Fallo desde el inicio en Moqui. 40

5.28. Base de datos con la transacción fallida desde el inicio. 41

Capítulo 1

Introducción

A modo de resumen, el siguiente trabajo de memoria describe la forma en que se abordó el desarrollo de un sistema de arbitraje de criptomonedas basado en Moqui, presentando sus desafíos, dificultades, objetivos y resultados.

En este primer capítulo se explicará el contexto general del presente trabajo, explicando qué es el arbitraje y los posibles riesgos que envuelve este mecanismo. Además, se hablará sobre Moit, la empresa que postuló este tema de memoria, y de Moqui, el *framework* con el que se desarrolló este sistema. Finalmente se exponen los objetivos de este trabajo.

1.1. Antecedentes Generales

Las criptomonedas, o también conocidas como criptodivisas, son activos digitales de alcance global [1] que se han popularizado en la última década. Se caracterizan por proporcionar un sistema de pago seguro debido a su cifrado criptográfico. Además, estas funcionan y operan en un sistema descentralizado, sin la intervención de instituciones financieras, entidades fiscalizadoras, o de control.

Los criptoactivos no existen de forma física: se almacenan en una cartera digital, o también conocidas como: E-Wallets. Estas últimas permiten a los usuarios administrar dinero de una forma virtual y realizar transacciones de una forma simple en cualquier momento y desde cualquier lugar.

Si bien una transacción formal de criptomonedas puede resultar un poco complejo de comprender debido a su origen criptográfico, para efectos de este trabajo se puede entender como el intercambio que realizan dos usuarios, donde uno entrega una cantidad específica de una criptomoneda de la que es dueño, y el otro a cambio entrega dinero “real” u otro criptoactivo.

Cabe recalcar que la plataforma en dónde se realizan la mayoría de las transacciones de criptodivisas son los *exchangers*, que funcionan como el punto de encuentro de miles de usuarios interesados en realizar estos intercambios. Resultan muy relevantes los *exchangers*, puesto que permiten al usuario, además de realizar pagos y otras funciones, sacar ventajas del mercado y lograr ganancias en función de la volatilidad del precio de los criptoactivos.

A modo de ejemplo, una transacción común en un *exchanger* se da cuando un usuario A, ofrece una cantidad específica de una criptomoneda de la que es dueño a un precio determinado. Si el usuario B, acepta esa oferta, la cantidad de criptomoneda ofrecida se transferiría a este último, y se pagaría el precio estipulado desde un principio al usuario A.

1.1.1. Sistema de arbitraje

Un sistema de arbitraje está presente en todos los activos económicos que buscan sacar ventaja de las situaciones de ineficiencia de los mercados, y no sólo está presente en las criptodivisas. El objetivo es aprovechar las diferencias de valores que tasan en la misma bolsa, o en bolsas diferentes. También existe arbitraje en el mercado de derivados, en el de renta fija, entre el mercado primario y secundario. Dicho de otro modo, el fin del arbitrajista es vender el activo más caro de lo que lo compró, y en especial para el caso de las criptomonedas, se busca realizar esta operación en un reducido espacio de tiempo para disminuir los riesgos de pérdida que pueden ocasionarse por la alta volatilidad de este tipo de activo. [2]

Para que exista arbitraje se debe cumplir al menos una de las siguientes condiciones [3]:

- El mismo activo no se transe al mismo precio en distintos mercados
- Dos o más activos que producen el mismo flujo de efectivo no se transen al mismo precio.

Para el mercado convencional existen más dificultades a la hora de encontrar un buen arbitraje, puesto que hay un ente que regula los precios de los activos y los mantiene en un margen, es decir, les centran límites y restricciones. Por el contrario, para un sistema descentralizado como el de las criptomonedas, resulta más fácil porque no existe tal ente que regule los precios de los activos de manera estricta, y es más probable que haya una diferencia en éstos que se pueda explotar, lo que da valor a un sistema de arbitraje.

Si bien existen muchas clases de arbitrajes, cuando se habla de arbitraje de criptomonedas existen dos tipos que suelen ser lo más frecuentes, y que además se verán en este proyecto de memoria:

- Arbitraje de *exchanger* o simple: Este es el tipo más común. Se produce cuándo se compra el mismo activo en un *exchanger* y se vende en otro. La idea es encontrar una diferencia de precios entre los *exchangers* para sacar provecho.
- Arbitraje triangular: Este ocurre cuando hay una discrepancia de precio entre tres monedas diferentes, que se intercambian entre sí en una especie de bucle. Por ejemplo, si se compra BTC (Bitcoin) con BNB (Binance Coin), luego se compra ETH (Ether) con ese mismo BTC, para finalmente volver a comprar BNB con ETH. El objetivo sería sacar provecho del valor relativos entre ETH y BTC, si es que este no coincide con el valor que tienen cada una de estas criptodivisas con BNB, existe la oportunidad de arbitraje.

Aunque el sistema de arbitraje de criptomonedas tenga como principio ser de bajo riesgo [4], no significa que esté exento de ellos. Uno de los principales enemigos que tiene el sistema son las tarifas u honorarios que existen al retirar dinero o realizar transacciones. En muchas ocasiones una compra o venta puede ser potencialmente beneficiosa, pero a causa de los impuestos se puede volver una pérdida de dinero. Junto con ésto, también puede haber

problemas con los impuestos propios de cada país, que al retirar el dinero de la criptodivisa y convertirlo en moneda local puede también significar una reducción de las ganancias totales. También se pueden encontrar complicaciones menores en la misma volatilidad del precio de las criptomonedas y en la baja liquidez de los activos, que provoca que no se pueda ejecutar la transacción rápido al precio deseado.

1.1.2. Moit y el framework Moqui

Por otro lado, Moit es una empresa chilena que busca dar soporte tecnológico a diversas organizaciones, ofreciendo aplicaciones de tipo *SaaS* (Software as a Service), sistemas *On Premise*, entre otros. Dentro de su oferta, poseen servicios de *ERP* (enterpise Resource Planning), *WMS* (Warehouse Mangement), *CRM* (Customer Relationship Management), entre otras [5] y lo hacen a través de Moqui.

Moqui [6] es un *framework* open source basado en XML y Groovy que cuenta con diversos módulos de apoyo para diversos servicios de aplicaciones empresariales, y entrega soporte a una amplia variedad de aplicaciones, desde sitios web pequeños a complejos sistemas ERP. Esto lo logra a través de herramientas que manejan interfaces conectadas entre sí, uso de bases de datos, lógica de back y front-end y entre otros.

Algo que vale la pena mencionar, es que la solución de este trabajo, comprende de varias tecnologías aparte de Moqui, y que serán integradas a este *framework*. La empresa chilena también buscaba demostrar lo versátil que podía ser Moqui junto a otras tecnologías.

Moit, junto con buscar la integración del sistema de arbitraje de criptomonedas, también quiere agregar valor al *framework* de Moqui, puesto que la solución que explora esta memoria será útil para la comunidad de desarrolladores que busquen implementar un sistema similar de arbitraje u otro sistema con manejo de criptomonedas.

Aprovechar el increíble crecimiento que están teniendo las criptomonedas en la actualidad puede presentar un interesante nicho de negocio, ya que cada vez más mercados implementan su uso en la actualidad [7]. Es en razón de lo anterior, que Moit presentó este tema de memoria para alumnos de la Universidad de Chile, y a su vez ha sido elegida para ser desarrollado en este trabajo.

1.2. Objetivos

1.2.1. Objetivos Generales

El objetivo de esta memoria es implementar un sistema de arbitraje de criptomonedas, que en primer lugar, guíe al usuario para informarle acerca de potenciales oportunidades de arbitraje entre los *exchangers*, y al mismo tiempo, que sea capaz de efectuar tanto el arbitraje simple como el arbitraje triangular entre los *exchangers* utilizados.

1.2.2. Objetivos Específicos

1. Realizar un levantamiento de los requisitos funcionales para un sistema de arbitraje.

2. Implementar a través del uso de APIs [8], un sistema que entregue los valores de compra y venta de Bitcoin en al menos dos *exchangers* y que estos valores puedan ser vistos a través de Moqui.
3. Escalar el sistema anterior para no sólo tener valores de Bitcoin en dólares estadounidenses (USD), sino además, de tener valores de Etheureum y valores de cambio entre Ethereum y Bitcoin.
4. Establecer un programa que sea capaz de realizar transacciones simples de criptomonedas, es decir, que sea capaz de comprar y vender una criptodivisa a cambio de USD.
5. Crear un programa que simule un arbitraje simple entre dos *exchangers*.
6. Crear un programa que sea capaz de simular un arbitraje triangular en un mismo *exchanger*.
7. Crear un programa que sea capaz de simular un arbitraje triangular entre distintos *exchangers*.
8. A través de APIs, implementar una vista que permita al usuario revisar el dinero que tenga en sus E-Wallets de los distintos *exchangers*.
9. Implementar un sistema que permita ver el historial de transacciones realizadas en los distintos exchagners.
10. Realizar una evaluación de la solución implementada, con la cual se pueda determinar el cumplimiento de los requisitos levantados.

Capítulo 2

Preparación

En el siguiente capítulo se discutirá acerca de los requerimientos del sistema de arbitraje de criptomonedas en Moqui, además de hacer un estudio de otros sistemas de arbitraje que están presentes en el mercado. También se revisará el estudio que se hizo sobre Moqui y de las tecnologías propias del trabajo.

2.1. Definición de requerimientos

Antes de comenzar con la elaboración del sistema de arbitraje, y con ayuda del profesor guía, se plantearon los requerimientos principales para poder abordar el problema, estos son de alta importancia para el sistema y se deben cumplir para llevar a cabo un sistema de arbitraje en Moqui.

En primer lugar, se planteó la importancia de que el sistema fuese escalable, dicho de otro modo, se espera que en caso de que fuese necesario, que fuese fácil agregar información de otro *exchanger* o de otra criptodivisa, puesto que el sistema de arbitraje se beneficia de tener muchas fuentes de información, mientras más criptomonedas se estudien y se arbitren, mayores posibilidades hay de encontrar un potencial arbitraje beneficioso.

Otra funcionalidad crucial es llevar un registro de las transacciones que se llevan a cabo con el sistema, es de vital importancia que haya un reflejo de lo que se hace o no con este sistema, y además permite llevar una cuenta clara de lo que se gana o no.

Asimismo, se veía importante la necesidad de que Moqui fuese un "punto en común" para todos los *exchangers* que se trabajen en el sistema, es decir, que la información de los *exchangers*, ya sean las criptomonedas que están presentes de las E-Wallets, las transacciones realizadas y los precios de las criptomonedas, estuvieran presentes en Moqui, para que el usuario que use el sistema de arbitraje tenga todo en un mismo lugar.

El sistema debe ser capaz de simular tanto el arbitraje simple, como el arbitraje triangular.

Por último, una vez se realice una transacción, el sistema debe ser capaz de identificar si es que hubo ganancia o no, y mostrarlo en el *front-end*, para que el usuario tenga conocimiento inmediato.

2.2. Investigación de otros sistemas de arbitraje

Una vez planteados los requerimientos, se buscaron diversas alternativas de sistemas de arbitrajes que habían en el mercado, y a raíz de esta búsqueda, nacieron características importantes que pueden ser de alta relevancia y que pueden estar presentes en la implementación de este sistema. Las características más destacables que están presentes en los otros sistemas son:

- Un usuario no debe ser experimentado ni especializado en el mundo de las criptomonedas para poder utilizar el programa, el sistema debe ser auto explicativo e intuitivo.
- Mientras más criptodivisas se manejen en el sistema, habrán mayores oportunidades de ganancias.
- Del mismo modo, mientras más *exchangers* hayan en el sistema, mayor variación de precios entre las criptomonedas habrán, por lo que nuevamente se abren más oportunidades que aprovechar.
- Soporte para varios tipos de arbitraje, mientras más estrategias hayan disponibles, más fácil es generar ganancias.
- El historial de las transacciones que se realicen es sumamente importante para dejar en claro los movimientos que el usuario va haciendo.
- Cálculos de arbitrajes en tiempo real para las distintas monedas ofrecidas, en donde se diga al usuario si es que se ganará o no dinero al realizar el arbitraje.
- Cualquier tipo de cargo comercial de retiro de dinero o de impuestos para transacciones va incluido en el cálculo del beneficio final del arbitraje.

Se escogieron estas características puesto que entregan un gran valor agregado al sistema y se repetían en muchos arbitrajes que el mercado actual ofrece.

2.3. Estudio Moqui Framework

Instanciado el ambiente de desarrollo, comienza la familiarización con Moqui como tal. Para comenzar se presentan los módulos de Moqui, que por sí solos muestran varias funcionalidades y que además pueden utilizar funcionalidades de otros módulos si es que así se requiere.

Dentro de todos los módulos que Moqui puede ofrecer, existen tres que son los principales para el funcionamiento del sistema, que son:

- mantle-usl: Este módulo tiene las funcionalidades básicas del *framework*. Tiene la base de los servicios básicos que Moqui entrega.
- mantle-udm: Provee el modelo de datos para Moqui y la carga de datos. Este módulo permite funcionar diferentes motores de bases de datos, específicamente permite el funcionamiento con PostgreSQL, y en ambiente de desarrollo usa un motor autocontenido llamado H2.

- SimpleScreens: Módulo que provee *templates* que son usados en el desarrollo de front-end.

Junto con lo anterior, el equipo de Moit está desarrollando sus propios módulos:

- Moquichile: Es un módulo que está encargado de tener funcionalidades básicas de transformación de datos al contexto chileno, como por ejemplo cambiar un tipo de moneda a peso chileno (que potencialmente será muy útil para el contexto de este trabajo), creación de documentos de servicios de impuestos internos de Chile o traducciones de idiomas al español.
- moit-utils: Componente que es utilizado para reunir herramientas útiles y comunes entre distintos proyectos.

Por otro lado, cada módulo del *framework* tiene una estructura peculiar dividida en directorios, que indican dónde se debe implementar cada funcionalidad de la aplicación. La estructura y nombre de las carpetas es la siguiente:

- *data*: En esta carpeta está la carga inicial de datos y datos de prueba. Con estos datos se puede levantar el módulo y poder realizar pruebas básicas de la aplicación. Algo importante a recalcar es que la lectura y ejecución de los archivos se hacen en orden alfabético, por lo que los archivos de carga inicial deben estar antes (en orden alfabético) que los de prueba.
- *entity*: Se encuentran todas las entidades del módulo, las cuales son cargadas durante el proceso de ejecución de Moqui. Acá está contenido la estructura de datos y modelos.
- *screen*: En este directorio se encuentra todo lo relacionado a las interfaces de la aplicación y su front-end.
- *service*: En esta carpeta se involucra todo lo que relaciona al desarrollo back-end.

Finalmente, se estudia en profundidad la forma de integrar el uso de Javascript y python-cgi con Moqui, y uso de API's. Esto último se explicará más en detalle en la fase siguiente y también en la fase de implementación.

2.4. Estudio de Tecnologías Propias del Proyecto

2.4.1. Testnet

La *testnet* [9] es un bloque de cadena (*blockchain*) que está separado de los comunes (*conocidos como mainnet*), en los cuales los desarrolladores pueden realizar ensayos con criptomonedas de prueba (cuyo valor es casi nulo) para simular transacciones o simulaciones de movimientos con criptodivisas. De este modo, se evitan modificaciones en el *blockchain* original, la pérdida de información, o en el caso de las criptomonedas, de dinero. Sin lugar a dudas, el *testnet* ofrece una increíble oportunidad para el desarrollo de esta memoria, permitiendo realizar pruebas y evaluaciones del sistema sin poner en juego dinero real.

Un punto a tener en consideración, es que todas las criptomonedas bajo la *testnet* no pueden ser transferidas a un *blockchain* original (ni tampoco al revés), de este modo los desarrolladores se aseguran la correcta separación entre los *blockchains* de la *testnet* y los de la *mainnet*, evitando cualquier tipo de acción fraudulenta que pueda alterar la información. Por otro lado, el minado de criptomonedas [10] en un *testnet* es mucho más sencillo [11], es por ello que existe una altísima cantidad de estas monedas de prueba circulando y además también explica su bajísimo valor.

Aparte de estas diferencias generales, y algún otro pequeño detalle, el resto de los elementos de la criptomoneda de un *testnet* y de una *mainnet* son prácticamente idénticas, y más relevante aún, siempre se puede asegurar que algún movimiento realizado en la *testnet* se puede hacer perfectamente en la *mainnet*.

2.4.2. Entorno de pruebas, sandbox y elección

Del mismo modo, todos los cambios que se hacen en la *mainnet* se realizan primero en la *testnet* para no estropear nada, en otras palabras, es un ambiente de pruebas para evitar fallos a futuro.

Y así como la *testnet* es un ambiente de prueba, también existen ambientes de pruebas en los *exchangers*. Los *exchangers* ofrecen "*sandbox*" para que un usuario pueda acceder a él y realizar transacciones con criptomonedas sin poner en riesgo su dinero, y mejor aún, estas cuentas en los *sandbox* vienen con criptomonedas (*de la testnet*) para que se pueda ir probando tranquilamente. Aún más novedoso, muchos *exchangers* ofrecen sus APIs tanto como para el *sandbox* como para su ambiente real, por lo que si se desea probar el uso de las APIs, también se puede a través de *testnet*. Si bien el valor de las criptomonedas en la *testnet* es bajo, se simulan a un precio ficticio similar (aunque nunca igual) al que está en el *exchanger* real, por lo que las pruebas suelen ser bastante robustas y verídicas.

La búsqueda de un buen *sandbox* consta de cuatro factores principales:

- Soporte de ETH y BTC (ambas de *testnet* y además son de las criptomonedas más negociadas [12])
- Buen flujo de transacciones (muchas ofertas de compra y venta permiten mayor dinamismo)
- Precios más o menos ajustados a la realidad (para que sea lo más verídico posible)
- Una API robusta que permita hacer buenas pruebas con las criptomonedas.

Para esta tarea, se escogieron tres *sandboxs* que cumplían con estas características: *Gemini Sandbox, Coinbase Sandbox y Swyftx Sandbox.*

Por último, resulta relevante mencionar que los *sandboxes* de Gemini y Coinbase, tienen un comportamiento en común. El usuario a través de las APIs podrá comprar y vender criptodivisas a un precio específico por una cantidad específica de criptomoneda. Explicado de otra forma, si es que se intenta comprar 1 BTC por un valor 10 USD, el sistema efectúa la transacción si es que este precio está disponible para esa cantidad, y si es que no lo está,

simplemente rechaza la transacción. Pero por otro lado, la API de Swyftx no soporta de buena manera este tipo de operación, por lo que para el caso específico de este *exchanger*, si es que el precio que se busca operar en Swyftx no está disponible, en vez de rechazar la transacción como en el caso de los otros dos *exchangers*, realizará la transacción de igual manera por la cantidad que el usuario quiere, pero con el mejor precio posible del mercado en ese momento.

Capítulo 3

Diseño

En este capítulo se explicará el diseño seguido para la implementación del sistema de arbitraje. Se verá desde la arquitectura, los programas que los componen, uso de API's, python-cgi y por último las interfaces.

3.1. Arquitectura

Una vez terminado el estudio de Moqui, se trabajó en la conexión entre Moqui y los *exchangers* través de las APIs que estos mismo ofrecen. En un principio se iba a realizar todo a través de JavaScript, pero debido a la complejidad de algunas tareas y por problemas de *CORS* [13], la conexión por completo entre Moqui y los *exchangers* tuvo que ser realizada a través de un servicio propio elaborado en python.

De este modo, la comunicación entre el servidor python y Moqui fue llevada a cabo gracias al módulo python-cgi. Este servidor era el encargado de enviar las *HTTP request* a las APIs ofrecidas por los *exchangers* y así devolver una *response* para que esta pueda ser leída a través de JavaScript o servicios propios creados en Moqui, usando Groovy y xml.

Finalmente, la línea de acción común del proyecto, consiste es que desde el *front-end* (Moqui) se llama a uno de estos servicios elaborados en python, para que el servidor en python ejecute consultas a las APIs y devuelva información a Moqui, que es leída a través de AJAX (en caso de usar JavaScript) o leída a través de algún servicio creado en Moqui. También, y en caso de que se realice una transacción, el servidor de python será el encargado de guardar la información de la misma en la base de datos.

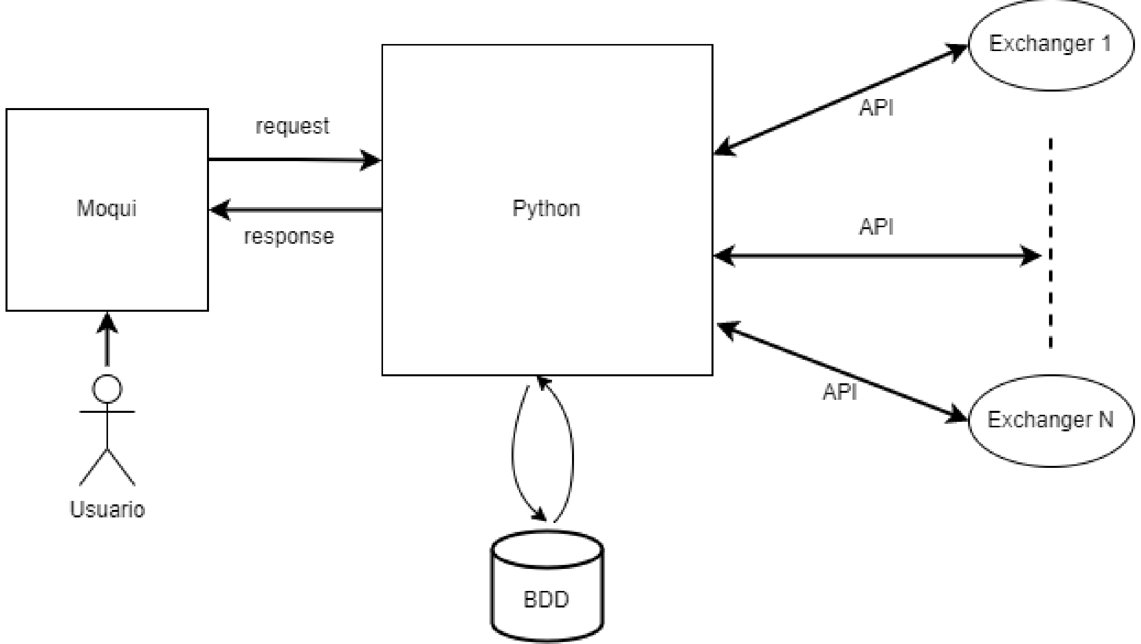


Figura 3.1: Diseño de Arquitectura de la solución

3.2. Base de Datos

La mayor parte de almacenamiento de datos y despliegue de estos en la aplicación será a través del uso de las mismas APIs de los *exchangers*. Es decir, si se desea consultar los saldos de las E-Wallets, o revisar el historial de transacciones realizadas en un específico *exchanger*, bastará con hacer la consulta a la API correspondiente y desplegar los datos.

Cabe señalar, que fue importante realizar un sondeo de las transacciones realizadas a través de Moqui y que estás pudiesen ser mostradas al usuario. Para ello fue necesario que se almacenaran en una base de datos a través de una base de datos gestionada con MySQL.

Como es importante almacenar la información de las transacciones, las entidades que comprenden a la base de datos llevan el nombre de las operaciones que se pueden realizar en Moqui. Las operaciones disponibles en el sistema son: compra (o venta) de criptomonedas, arbitraje simple, arbitraje triangular unitario y arbitraje triangular múltiple. Es por esto que el nombre de las entidades de la base de datos son:

- comprarCrypto
- arbitrajeSimple
- arbitrajeTriangularUnitario
- arbitrajeTriangularMultiple

Estas entidades, tienen como objetivo contener la información más relevante de cada transacción, como por ejemplo el *exchanger* en que se realizó, la cantidad de criptodivisa en que se hizo la operación, precio, símbolo (nombre), la operación realizada (compra o venta),

el camino escogido (opción que escoge el usuario para arbitrar) y fecha en que se realiza el intento de arbitraje.

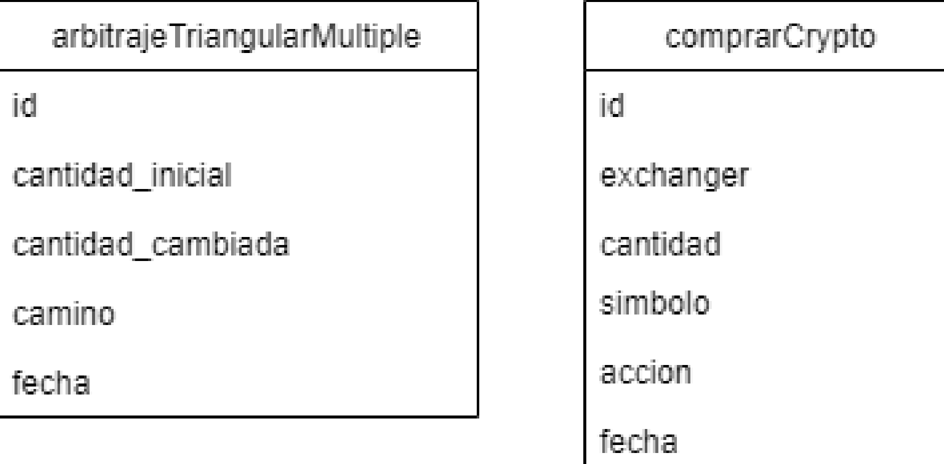


Figura 3.2: Entidades base de datos

3.3. Programas

En esta sección se plantearán las partes más importantes y relevantes para la construcción del sistema de arbitraje.

3.3.1. Screens

Así como se comentó en el capítulo 2, las *screen* de Moqui comprenden las interfaces de la aplicación y corresponden al *front-end*. En particular para este proyecto fue necesario plantear *screens* que tuvieran la lógica para poder realizar una transacción o un arbitraje, y también *screens* que entregarán información importante para el usuario. Estas *screens* corresponden a:

- billeteras.xml: corresponde a la vista que permitirá al usuario consultar el dinero que tiene en las E-wallets de los distintos *exchangers*.
- historialExchangers.xml: corresponde a la vista que permite al usuario revisar las distintas transacciones realizadas en los distintos *exchangers*.
- historialTransacciones.xml: comprende a las transacciones realizadas en Moqui. Extrae la información directamente de la base de datos y las muestra al usuario.
- simpleArbitrage.xml: es la *screen* encargada de tener la vista e interfaz del arbitraje simple. Acá se realiza el arbitraje simple.
- triangularOne.xml y triangularMultiple.xml: son las vistas que contienen la interfaz de arbitrajes triangualres. El primer *screen* comprende a un arbitraje triangular realizado en un mismo *exchanger*. Por su parte, la segunda *screen* tiene la vista de un arbitraje triangular realizada en distintos *exchangers*. En estas *screen* se llevan a cabo los arbitrajes triangulares del sistema.
- comprarCrypto.xml: esta última *screen* tiene la vista que permite al usuario comprar (con USD) alguna criptomoneda que esté disponible.

Por último, hay algunas *screens* que contienen la lógica interna de Moqui y el sistema de arbitraje:

- cryptoDashboard.xml: esta *screen* permite al usuario navegar a través de las demás partes de la aplicación, vale decir, corresponde al punto en común de todas las demás *screens* que están disponibles.
- cryptoarbitragestatic.xml: *screen* estándar y necesaria para el correcto funcionamiento del módulo creado en Moqui.

3.3.2. Servicios

Si bien, la mayoría de los servicios usados fueron confeccionados en python (que será explicado más en detalle en unas secciones más adelante), se crearon servicios nativos en Moqui para desplegar información de una forma mucho más fácil y sencilla, aprovechando el lenguaje xml y groovy que son nativos en Moqui.

El servicio básico creado en Moqui correspondía a un *getter*, es decir, se encargaba de obtener la información de un servicio/api (en este caso, obtenía valores del servicio levantado con python-cgi) para luego hacer un *map* de los valores obtenidos en la respuesta al servicio y desplegarlos fácilmente en parámetros previamente definidos. Estos servicios obtenían la información sobre las transacciones realizadas en los *exchangers* y también las criptomonedas que el usuario tiene en sus E-wallets.

La estructura acerca del servicio era un archivo de extensión *XML* llamado *cryptoAPI.xml* que contenía los distintos servicios encapsulados:

- cryptoAPI.xml
  - get#Billeteras: encargado de obtener la cantidad de criptodivisas de las E-wallets.

- get#Historial: obtiene las transacciones realizadas en los distintos *exchangers*.
- get#HistorialTransacciones: obtiene información de las transacciones realizadas en Moqui desde la base de datos.

3.3.3. APIs

Además de los servicios y APIs implementados para el correcto desarrollo del sistema de arbitraje, también se consumieron APIs que ofrecían los distintos *exchangers*. En total, se consumieron tres APIs para realizar pruebas, una de cada *exchanger* que ofrecía *sandbox*. El objetivo de estas APIs era poder concretar las transacciones con las criptomonedas; obtener datos importantes como valores de las criptodivisas; obtener las E-wallets; y el historial de transacciones.

Las APIs usadas fueron:

- Gemini API [14]
- Coinbase API [15]
- Swyftx API [16]

Si bien los *endpoints* usados en cada API son diferentes, las funcionalidades de estos *endpoints* son bastante similares, siendo las más usadas aquellas que permiten obtener los precios de las criptomonedas, obtener las E-wallets, obtener las transacciones realizadas y colocar una orden de compra (que es precisamente realizar una transacción con alguna criptomoneda).

3.3.4. Python CGI

El rol principal del módulo python-cgi es encargarse de la comunicación entre todo el sistema. Se encarga de construir una comunicación dinámica entre el cliente y servidor. En este sistema de arbitraje, parte de la información que entrega el usuario a través del cliente es necesaria para poder comunicarse correctamente con las APIs que eran ofrecidas por los *exchangers*. Esa información recibida era ejecutada en scripts de python-cgi.

Los *scripts* de python están divididos por funcionalidades: *scripts* de transacciones, *scripts* de funciones auxiliares y *scripts* que obtienen información de los *exchangers*. Los de transacciones tienen justamente como rol concretar una transacción:

- arbitrajeSimple.py: crea una transacción que concreta un arbitraje simple.
- arbitrajeTriangularMultiple.py: contiene la lógica y concreta un arbitraje triangular entre varios *exchangers*.
- arbitrajeTriangularSwyftx.py: es el encargado de realizar un arbitraje triangular en un mismo *exchanger*, específicamente en Swyftx.
- comprarCrypto.py: compra criptomonedas a cambio de USD en un *exchanger*.

Algo importante a recalcar, es que cada vez que el sistema efectúa una transacción exitosa, se guarda la información de la transacción en la base de datos.

Por otro lado, los *script* que comprenden las funciones auxiliares son:

- transacciones.py: contiene todas las funciones necesarias para llamar a las APIs y generar las acciones correspondientes.
- db.py: contiene todas las funciones necesarias para llamar guardar y obtener valores de la base de datos.

Por otro lado, los *scripts* que obtienen información de los *exchangers* son:

- wallets.py: llama a la API de los *exchangers* arbitrados y arroja todas las E-wallets asociadas a ellos.
- getHistory.py: llama a la API de los *exchangers* arbitrados y arroja las últimas transacciones realizadas en ellos.

Por último, si bien se usó el ambiente de pruebas (*sandbox* API) para todos estos los *exchangers* al llamar a las APIs, la ventaja de la implementación es que sólo bastaría con cambiar la URL del ambiente de pruebas a la URL real de la API del *exchanger* para que se puedan hacer transacciones reales.

3.4. Interfaces

La primera interfaz destacable de la aplicación comprende la vista de todos los módulos que hay en Moqui. Básicamente es un menú que permite ir al módulo que el usuario escoja.

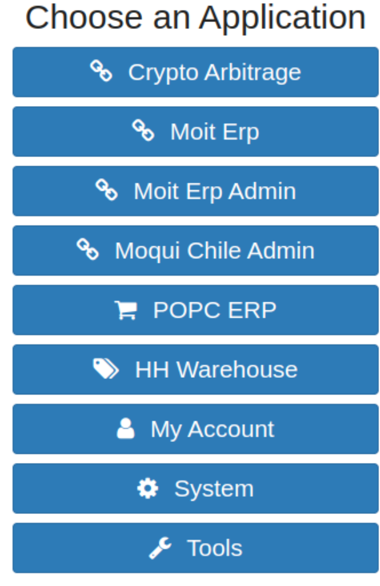


Figura 3.3: Inicio de aplicación

Por supuesto, para acceder al módulo del sistema de arbitraje, el usuario debe apretar el botón denominado “Crypto Arbitrage”. Una vez seleccionada esa opción, se muestra la *screen* que corresponde a *cryptoDashboard*:

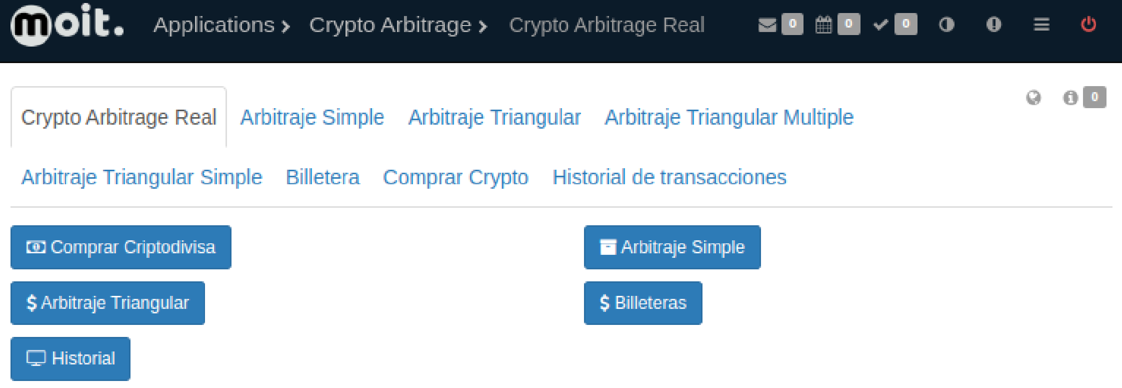


Figura 3.4: CryptoDashboard

En la figura 3.3, se pueden apreciar tres características principales: el *navbar* o menú de navegación situado en la parte superior; las pestañas que están situadas en medio; y finalmente los botones de la parte inferior. El *navbar* permite volver al inicio (cliqueando en el ícono de moit), navegar entre los módulos (cliqueando en *applications*) y además permite siempre volver al *cryptoDasboard*, al hacer clic en "Crypto Arbitrage". Por otro lado, las pestañas permiten acceder directamente a una *screen* en particular y los botones de la parte inferior permiten al usuario dirigirse a la sección que desee.

Se mostrarán las interfaces dividiéndolas entre aquellas que permiten arbitrar las criptomonedas y las interfaces que muestran información relevante para el usuario.

3.4.1. Interfaces de arbitraje

Arbitraje simple

En primer lugar, si desde la figura 3.4 el usuario hace clic en la pestaña o botón que dice "Arbitraje Simple" y realiza un arbitraje, se tendrá la siguiente interfaz:

Figura 3.5: Arbitraje Simple

En esta vista, podemos ver el arbitraje simple realizado entre el *exchanger* Gemini y Coinbase. En segundo lugar, se puede ver que debajo del nombre de los *exchangers* están los precios de compra y venta de las criptomonedas, de este modo el usuario logra identificar los precios a que están sujetos los arbitrajes. Cabe recalcar que el sistema identifica la transacción más beneficiosa para el usuario y es en la que se efectúa si se decide arbitrar, por esta razón no se considera el *exchanger* Swyftx en la figura 3.5.

Por debajo de los valores hay dos botones: el botón de “Refrescar” que actualiza los valores de precio y compra, y está el botón de “Calculo arbitraje” que despliega el texto por debajo de él, dando información de si es conveniente o no proceder con el arbitraje.

Por último, abajo existe un input en donde el usuario puede ingresar la cantidad de la criptomoneda que desee arbitrar (en este caso BTC), y una vez que aprieta el botón de “Arbitrar” se podría generar o no la transacción. Debajo del texto de “Información” aparece el resumen del arbitraje.

Arbitraje triangular múltiple

Por otro lado, si el usuario se dirige a la pestaña de “Arbitraje Triangular Multiple” y realiza un arbitraje, se verá la siguiente interfaz:

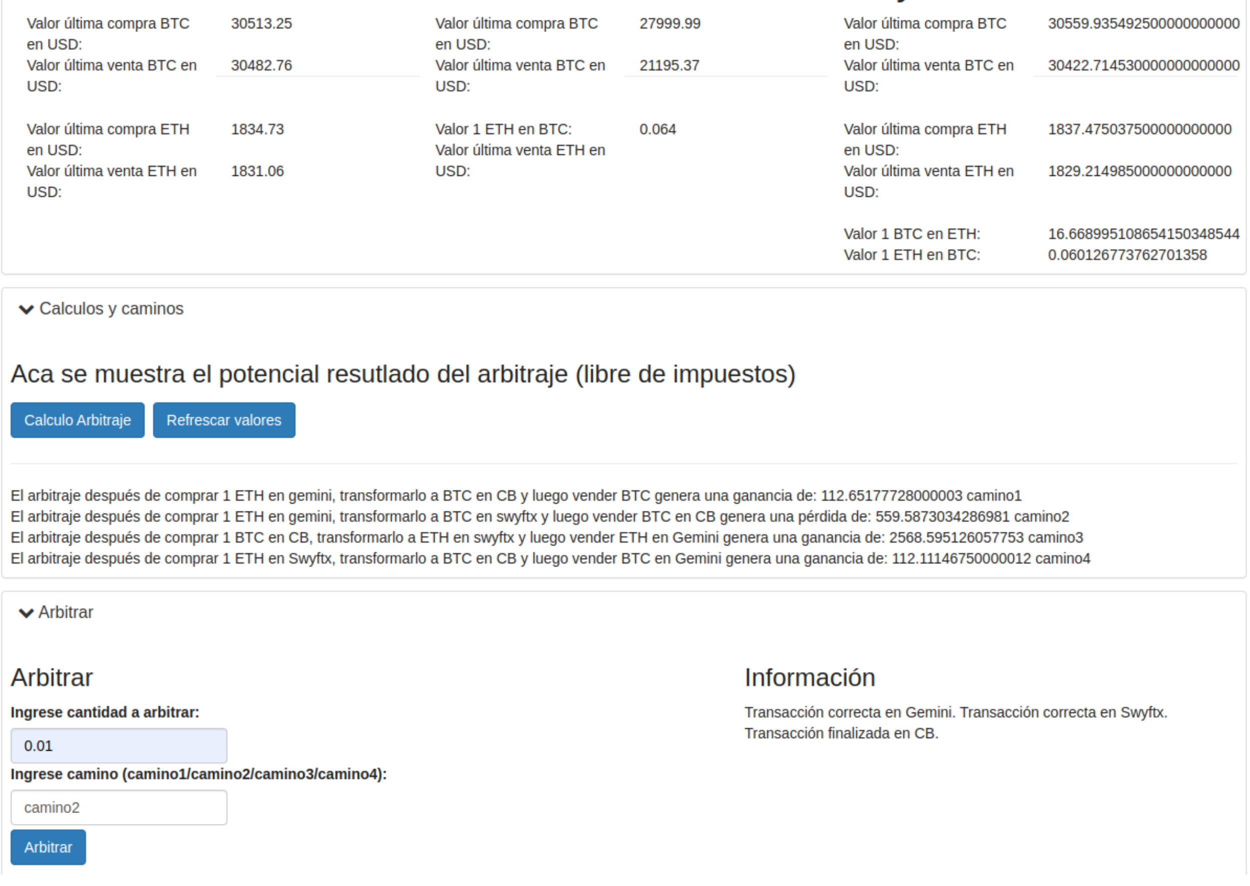


Figura 3.6: Arbitraje Triangular Múltiple

Como se puede notar, tiene muchas similitudes con la vista anterior: se muestra el precio de igual manera (esta vez mostrando más criptomonedas); están los botones para calcular los arbitrajes, mostrando al usuario si conviene o no; está el botón para refrescar los valores de las criptodivisas en los distintos *exchangers*. Nuevamente se tiene que ingresar una cantidad de moneda a arbitrar y también hay una caja de información sobre la transacción realizada. La diferencia radica que el usuario tiene que escoger el "camino" que más le convenga. Esto se debe a que pueden haber potenciales ganancias en más de un camino y quizá prefiera optar por un camino sobre otro (por ejemplo, el usuario podría preferir un camino con menor ganancia, pero que tenga mayor cantidad de ese activo, así puede hacer una transacción con más volumen de moneda). Se optó por la opción de "escoger camino" puesto que se da un poco más de versatilidad al usuario, sin perjuicio de que en parte se sacrifica un poco de velocidad en la transacción, puesto que el usuario debe elegir que camino escoger.

Arbitraje triangular simple

A continuación, se puede señalar que el "Arbitraje Triangular Simple" comprende prácticamente la misma vista, sólo que los cálculos de los "caminos" son en un mismo *exchanger*.

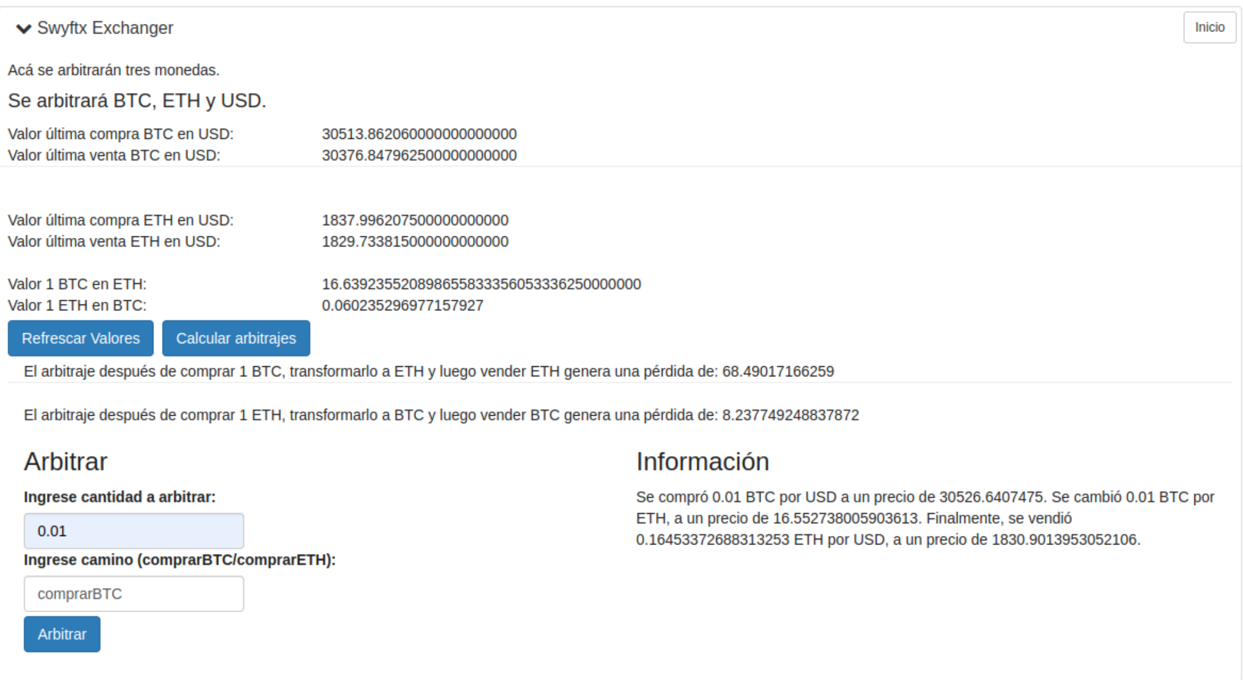


Figura 3.7: Arbitraje Triangular Simple

Comprar criptomoneda

Finalmente, la vista de "Comprar Crypto" es similar a la de "Arbitraje Simple", ya que cuenta con un botón que refresca valores de los precios, y un botón que dice "Operar transacción" para efectuar la misma. En esta interfaz uno puede comprar o vender la criptomoneda a cambio de USD:

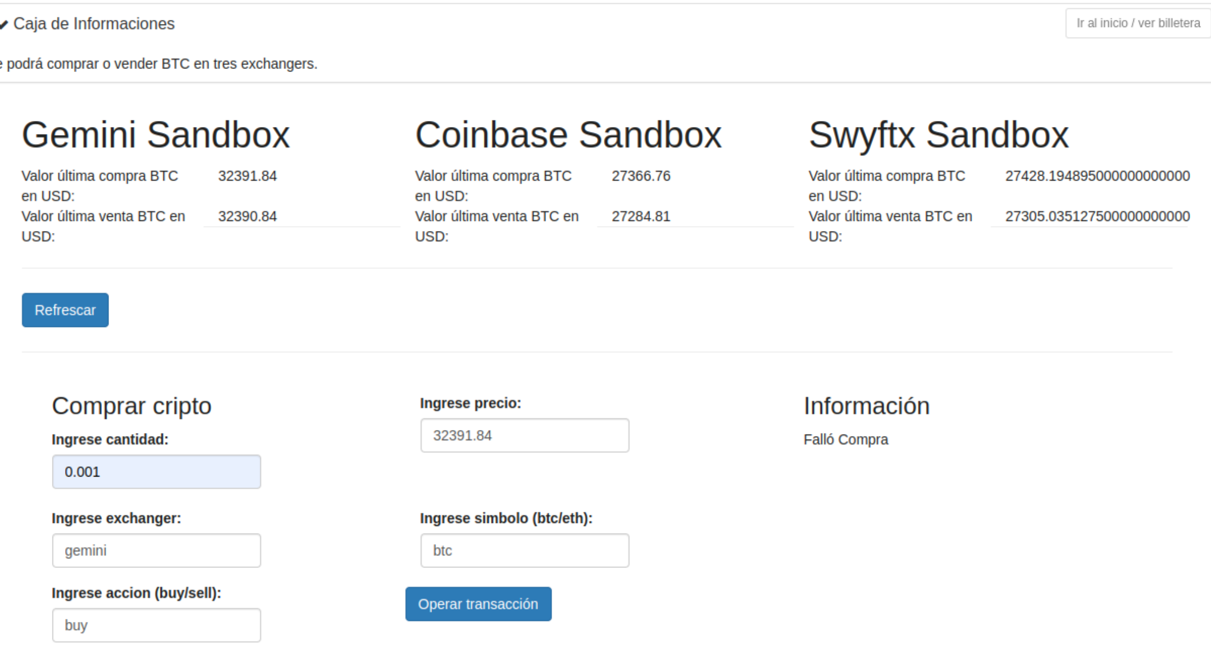
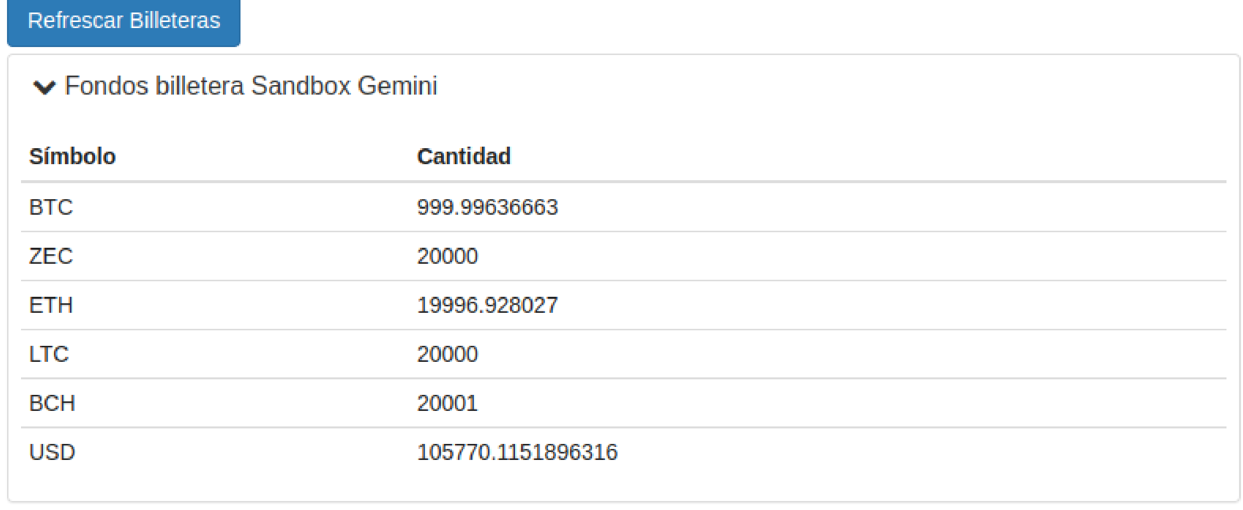


Figura 3.8: Comprar Crypto, vista de la interfaz una vez se intenta realizar la compra.

3.4.2. Interfaces de información

En esta sección de interfaces, se muestran las criptomonedas que hay las E-wallets de los distintos *exchangers*, las transacciones que se realizan en ellos; y además las transacciones realizadas en Moqui.

Cuando el usuario se dirige a la pestaña de “Billetera” puede ver:



Símbolo	Cantidad
BTC	999.99636663
ZEC	20000
ETH	19996.928027
LTC	20000
BCH	20001
USD	105770.1151896316

Figura 3.9: Pestaña Billetera.

El botón "Refrescar Billeteras" actualiza los valores de todas las billeteras de los *exchangers*. Cabe señalar que en la figura 3.9 sólo se muestra un extracto de la interfaz, puesto que las demás billeteras tienen la misma estructura.

Por otro lado, la *screen* que corresponde a *historialExchangers.xml* permite al usuario revisar sus últimas transacciones realizadas en los distintos *exchangers*:

Refrescar Historial

Historial Sandbox Coinbase

Símbolo	Cantidad	Acción	Fecha
BTC-USD	0.00996393	buy	2022-06-02T21:53:42.518437Z
BTC-USD	0.00003607	buy	2022-06-02T21:53:42.518437Z
BTC-USD	0.00034976	sell	2022-05-31T20:11:24.617411Z
BTC-USD	0.01000000	buy	2022-05-31T19:59:05.446951Z
BTC-USD	0.00100000	buy	2022-05-30T19:42:09.772047Z
BTC-USD	0.00065509	buy	2022-05-30T19:40:01.845617Z
BTC-USD	0.00409531	buy	2022-05-30T06:47:27.031339Z

Figura 3.10: Historial de transacciones.

Nuevamente, el botón de refresco actualiza las transacciones y también se muestra únicamente un extracto puesto que el historial se muestra de igual forma para el resto de los *exchangers*.

Finalmente, la vista que corresponde a la información guardada en la base de datos, es decir, las transacciones realizadas en Moqui, tiene la siguiente forma:

Historial de transacciones hechas en Moqui

Refrescar Historial

Historial Compras Criptomoneda

ID	Exchanger	Cantidad	Precio	Símbolo	Acción	Fecha
1	Gemini	0.001	31345.58	btcusd	buy	2022-06-06 20:59:34.034416
2	Gemini	0.001	31421.26	btcusd	sell	2022-06-06 21:18:42.907767
3	Gemini	0.01	31415.93	btcusd	buy	2022-06-07 19:22:12.962615
4	Gemini	0.001	31389.55	btcusd	buy	2022-06-07 22:11:58.025746

Figura 3.11: Historial de transacciones realizadas en Moqui.

En la figura anterior, se puede apreciar la información acerca de las compras de criptomonedas realizadas en Moqui. El resto de las transacciones siguen el mismo formato de tabla,

con su información correspondiente. Por último, el botón de “Refrescar Historial” sigue el mismo comportamiento que los demás botones de refresco.

Capítulo 4

Implementación

En el siguiente capítulo se explicará la implementación de la herramienta de arbitraje utilizando Moqui y otras tecnologías.

4.1. Instalación inicial Moqui Framework

Para instalar Moqui se debía contar con JDK 11, IDE InteliJ IDEA y una cuenta de GitLab para el versionamiento de código. Se decidió trabajar en un sistema operativo basado en Linux, Ubuntu, para facilitar el trabajo de desarrollo.

Junto con lo instalado, se utilizó la cuenta GitLab de Moit, para poder administrar y versionar todo código creado, y más importante aún, acceder a repositorios de trabajos realizados con Moqui para craer una instancia de proyecto basado en algunos repositorios previamente creados por el equipo de Moit.

Después de tener todo lo necesario, se procede con la instalación de Moqui. Para ello, primero se clona el repositorio *moqui-framework*, el que contiene la mayor cantidad de módulos y componentes necesarios para el funcionamiento del *framework*. Luego, se procede a correr el comando *gradlew GetComponents*, que se encarga de reunir los componentes necesarios para correr Moqui, para posteriormente hacer un *checkout* con *./gradlew* a las ramas, de la más general a la más específica. Respecto a esto último, se partió desde la rama master, luego la rama moit y finalmente la rama de este proyecto. A modo de ejemplo, se usó: *./gradlew gitCheckoutAll -Pbranch=master*, para hacer *checkout* de la rama maestra.

Concluido el paso anterior, se procede a hacer el *build* de la aplicación y la carga de los datos. El proceso para lograr esto es nuevamente con Gradlew y su comando *build*, que obtiene el archivo *.war*. Con el archivo listo, se procede a hacer la carga inicial de datos con el comando *gradlew load*. Finalmente se corre el archivo *.war* con el comando *gradlew run*, el cual levanta el servidor y Moqui por completo en la máquina local.

4.2. Instalación de otros servicios

Como se mencionó en secciones anteriores, aparte de Moqui, se hace uso del módulo python-cgi para lograr la comunicación entre Moqui y python. La razón detrás del uso de

este módulo de python es porque las peticiones a través de JavaScript están sujetas a *CORS policy enforcement* [13], que restringe la información que es entregada al resto de los *websites* que las consultan. A raíz de esto último, todos los métodos HTTP tipo *PUT, POST y DELETE* hacia las APIs que ofrecen los distintos *exchangers*, devuelven un error de *CORS*, puesto que no permiten que cualquier sitio acceda a su información más delicada ni tampoco permiten peticiones con *API-Keys* o *Tokens* de seguridad en sus peticiones en browsers.

Además de esto, habían cálculos complejos y *encriptaciones* necesarias a la información que se envía a través de las *request* hacia los *exchangers*, que en python resultaban más fáciles de realizar y ejecutar. De este modo, se decidió que Moqui se iba a comunicar con un servidor de python a través del módulo python-cgi, y una vez con la información que entregaba Moqui, python se encargara de lo demás.

4.2.1. Consideraciones con python

Cabe destacar que para usar el módulo de python-cgi en un puerto específico (este caso 8000) y en *localhost*, primero se tiene que levantar un *HTTP server* de python, y para esto se utiliza en consola:

Código 4.1: Correr servidor python de forma local usando el puerto 8000.

```
python -m http.server --bind localhost --cgi 8000
```

También fue necesario el uso de la librería *mysql.connector*, que permitió conectar la base de datos MySQL con los códigos del módulo python-cgi.

Una vez con el servidor corriendo, para encontrar el correcto funcionamiento de la conexión entre Moqui y python-cgi se debió tener presente un par de consideraciones. En primer lugar, se tenía que controlar el problema de *CORS*, desde Moqui a python-cgi (puesto que al estar corriendo en puertos distintos Moqui y python-cgi arroja error de *CORS* igualmente). En segundo lugar, se tuvo que idear una forma de entregar datos entre Moqui y python-cgi, es decir, que el intercambio de información entre ambas partes fuera correcta. Para estas consideraciones se plantea el siguiente esquema en todo archivo python:

Código 4.2: Ejemplo esquema python-cgi.

```
#!/usr/bin/python3
import cgi
import ...

print('Access-Control-Allow-Origin: *')
print("Content-type: application/json")
print("")

form = cgi.FieldStorage()

value = form.getvalue('value')

(...)

result = json.dumps(requests)
```

```
16 print(result)
```

La prímera línea de código alude al *shebang*, que es la ruta en dónde está el intérprete, que en este caso es python. Por otro lado, la línea 5 y 6, se encargan de primero, atajar el problema de *CORS*, permitiendo recibir llamadas de cualquier lugar, y segundo, aclara qué tipo será la respuesta que entregue de vuelta.

Además de esto, el objeto *cgi.FieldStorage()* será el encargado de contener toda la información que se recibirá desde Moqui (con ayuda de un llamado AJAX). Por su parte, el método *.getvalue()* será el utilizado para extraer la información del objeto *cgi.FieldStorage()*, y por último, se entrega el resultado de los llamados y de la construcción del mensaje final en un mensaje de tipo JSON *string*.

4.3. Servicios y Módulos de Moqui

4.3.1. Servicio propio de Moqui

Si bien se mencionaron los módulos correspondientes a las *screens* y *service* de Moqui, es relevante explicar el módulo de *service* en profundidad. Acá se crearon servicios que permitían solicitudes de tipo GET hacia los mismos python-cgi para obtener la información y mostrarla en las *screens*. Se usó este tipo de servicio en específico para aprovechar la facilidad con que Moqui podía leer la *response* y asociar cada dato a un campo en específico de una *screen*, para que se muestre fácilmente en el *front-end*.

Los tres *services* son bastante similares y se componen de tres secciones, por ejemplo, para el caso del servicio que entrega la información de las E-wallets de los *exchangers*:

Código 4.3: Primera sección servicio getBilleteras de Moqui.

```
1 <service verb="get" noun="Billeteras">
2       <out-parameters>
3          <parameter name="billeteraGemini"/>
4          <parameter name="billeteraCoinbase"/>
5          <parameter name="billeteraSwyftx"/>
6       </out-parameters>
```

Primero se debe definir el servicio con un *verb* y *noun*, que serán las palabras claves para después ser llamados en una *screen*. Además, se deben definir los *out-parameters* que son precisamente el nombre de los campos en las *screens* los cuales se mostrará la información que recibe el servicio. La segunda sección del servicio se ve así:

Código 4.4: Segunda sección servicio getBilleteras de Moqui.

```
1 <actions>
2          <set field="requestUrl" value="http://127.0.0.1:10000/cgi-bin/wallets.py"/>
3          <set field="contentType" value="application/json"/>
4          <script><![CDATA[
5             try {
6                restClient = new
   ↪ org.moqui.util.RestClient().method('GET').contentType(contentType).uri(requestUrl)
```

```
                response = restClient.call()
                responsePayload = response.text()
                jsonSlurper = new groovy.json.JsonSlurperClassic()
                responseMap = jsonSlurper.parseText(responsePayload)
                responsePrint = groovy.json.JsonOutput.prettyPrint(responsePayload)
                billeteraGemini = responseMap.gemini
                billeteraCoinbase = responseMap.coinbase
                billeteraSwyftx = responseMap.swyftx
            } catch (Exception e) {
                failReason = e.toString()
            }
        ]]></script>
```

En esta sección, primero se define la URL a la cual se consultará, además de definir el *contentType* en que se recibirá la respuesta. Seguido de esto, se crea un llamado a través de un cliente en Java, con un llamado tipo *"GET"* asociado con la URL y *contentType*. Cabe recalcar que el objeto *responseMap* se encarga de tener la información del llamado, para luego poder filtrarla, por ejemplo, *responseMap.gemini* obtiene todo el valor del JSON cuya llave es *"gemini"*. Y por último, la tercera sección:

Código 4.5: Última sección servicio getBilleteras de Moqui.

```
]]></script>
        <if condition="failReason">
            <log level="error" message="failReason: ${failReason}"/>
        </if>
    </actions>
</service>
```

Finalmente, si es que hay algún error se entrega este como respuesta de salida y se notifica al usuario.

En otro orden de cosas, y para recibir la información de este servicio, desde un *screen* se llama al servicio y se invoca los campos respectivos para que se vea en el *front-end*:

Código 4.6: Consumo servicio en una screen de Moqui.

```
<actions>
    <service-call name="cryptoApi.get#Billeteras" in-map="context" out-map="context"/>
</actions>

<form-list name="ListaFormulario" list="billeteraCoinbase">
    <field name="currency">
        <default-field title="Símbolo">
            <display/>
        </default-field>
    </field>
    <field name="balance">
        <default-field title="Cantidad">
            <display/>
        </default-field>
    </field>
```

```
16 </form-list>
```

En donde, primero que todo se llama al servicio, usando su *verb* y *noun* previamente definidos en el servicio, y luego se crean los campos correspondientes en donde irá la información contenida en el JSON.

4.3.2. Servicios python

Los servicios clave que ofrecen los archivos python son sin duda los que se encargan de generar una transacción de arbitraje. A mayor abundamiento, estos servicios se construyen en base a funciones que generan órdenes de compra o venta en los *exchangers*. Por ejemplo, en el caso particular del *exchanger* Gemini, para realizar una transacción se usa:

Código 4.7: Función que crea una transacción en Gemini.

```
1  def transaccion_gemini(simbolo,cantidad,precio,accion):
2
3      base_url = "https://api.sandbox.gemini.com"
4      endpoint = "/v1/order/new"
5      url = base_url + endpoint
6
7      gemini_api_key = "os.environ['GEMINI-APIKEY']"
8      gemini_api_secret = "os.environ['GEMINI-APISECRET']".encode()
9
10     t = datetime.datetime.now()
11     payload_nonce =  str(int(time.mktime(t.timetuple())*1000))
12
13     payload = {
14     "request": "/v1/order/new",
15         "nonce": payload_nonce,
16         "symbol": simbolo,
17         "amount": cantidad,
18         "price": precio,
19         "side": accion,
20         "type": "exchange limit",
21         "options": ["immediate-or-cancel"]
22     }
23
24     encoded_payload = json.dumps(payload).encode()
25     b64 = base64.b64encode(encoded_payload)
26     signature = hmac.new(gemini_api_secret, b64, hashlib.sha384).hexdigest()
27
28     request_headers = { 'Content-Type': "text/plain",
29                         'Content-Length': "0",
30                         'X-GEMINI-APIKEY': gemini_api_key,
31                         'X-GEMINI-PAYLOAD': b64,
32                         'X-GEMINI-SIGNATURE': signature,
33                         'Cache-Control': "no-cache" }
34
35     response = requests.post(url,
36                             data=None,
37                             headers=request_headers)
```

```
38
39    new_order = response.json()
40    return json.dumps(new_order)
```

Revisar esta función en específico es valioso por tres motivos: siguen un esquema similar casi a todos los métodos que se usan en *transacciones.py*; se revisan los campos que el usuario envía a través de Moqui; y por último, se revisa el proceso de autenticación que requieren las *requests* enviadas a los *exchangers*.

Los campos que el usuario envía desde Moqui en esta función son *simbolo, cantidad, precio* y *accion*:

- *simbolo:* representa la moneda a intercambiar, por ejemplo, el par *btcusd* representa Bitcoin a cambio de dólar.
- *cantidad:* representa justamente el número de criptomonedas que se operará en la transacción.
- *precio:* indica el valor monetario en que se va a efectuar la operación.
- *accion:* precisa si es que se quiere comprar o vender la criptodivisa.

En síntesis, son los campos precisos para que el usuario decida comprar o vender una cantidad específica de criptomoneda, en un precio dado.

Para finalizar, se debe mencionar que por temas de seguridad todo *exchanger* precisa no solo de una API-KEY, sino también de una autenticidad extra, como lo es "'firmar" cada mensaje que se envíe. Esto es así para corroborar la identidad de quién envía la *request* y proteger la información que es enviada.

Toda firma va junto con un *nonce*, que es un número que nunca se repite y que además va en aumento entre cada *request* hecha. La idea detrás de este *nonce* es prevenir que un potencial atacante que haya capturado alguna *request* vuelva a repetir el mismo llamado copiando y pegando la *request* que obtuvo. El *nonce* se construye a partir de la fecha y hora actual (como siempre va en incremento es útil para el cálculo del número). Por su parte, la "firma" que exige Gemini, consta de un MAC entre la API-SECRET y la *payload* (la información que se envía en la *request*) codificada en una base 64 junto a un *hash* de 384bits. Todo esto debe ser entregado en una base hexadecimal. Una vez creada la firma, se entrega en los *headers* de la *request*.

Capítulo 5

Evaluación

En este capítulo se evaluarán las distintas transacciones realizadas en Moqui, desde comprar la criptomoneda, hasta realizar un arbitraje simple, un arbitraje triangular en un mismo *exchanger* y un arbitraje triangular entre varios *exchangers*. Una vez realizada la transacción, se contrastará la información arrojada en Moqui y lo almacenado en la base de datos contra la información que ofrecen los *exchangers*. Algo interesante a tener en cuenta es que la fecha y hora en que se realiza transacción es "independiente" a la fecha y hora que los *exchangers* arrojan en sus transacciones, por lo que será un dato interesante a tener en cuenta para revisar qué tan rápido se hacen las peticiones desde python hacia los *exchangers*.

5.1. Presentar Resultados

5.1.1. Comprar Criptodivisa

Se efectuará compra y venta de criptomoneda y se contrastará con lo arrojado en el *exchanger*. Se comenzará por efectuar dos compras en Gemini, por una cantidad de 0.01 BTC al precio que arroja el sistema y otra compra por la misma cantidad, pero a un precio bajísimo (un precio muy bajo provocará un fallo, puesto que no habrán ofertas de compra tan bajas). Para el primer caso, una vez realizada la transacción, en Moqui se tiene:

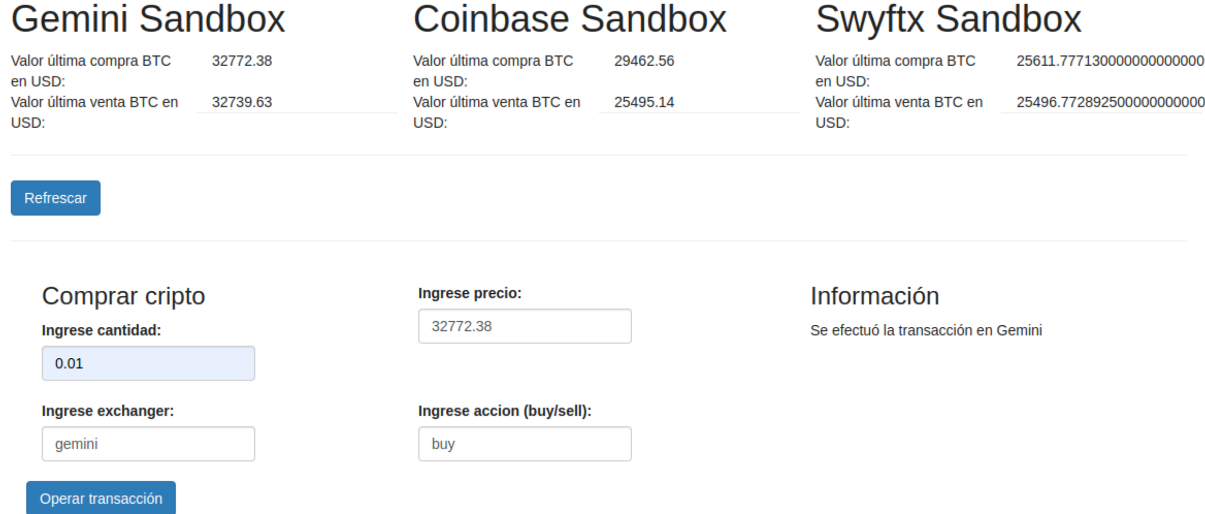


Figura 5.1: Primera compra BTC, vista en Moqui.

Y por otro lado, desde el historial de compras desde el mismo *exchanger* Gemini y lo registrado en la base de datos se tiene:

```
mysql> select * from comprarCrypto WHERE id = 22;
+----+-----------+----------+----------+---------+--------+----------------------------+
| id | exchanger | cantidad | precio   | simbolo | accion | fecha                      |
+----+-----------+----------+----------+---------+--------+----------------------------+
| 22 | gemini    | 0.01     | 32772.38 | btcusd  | buy    | 2022-06-12 23:57:53.468440 |
+----+-----------+----------+----------+---------+--------+----------------------------+
```

Date	Currency pair	Side	Order type	Crypto price	Quantity	Total	Status
06/12/2022 23:57:53	BTCUSD	Buy	Limit	$32,772.38 USD	0.01 BTC	$327.72 USD	Filled

Figura 5.2: Registro historial Exchanger y base de datos.

Se puede ver que en la parte superior está la consulta a la base de datos que arroja la información de la transacción y justo por debajo está el historial que entrega el *exchanger* acerca de la compra. El campo "fecha" desde la base de datos y "Date" en el historial del *exchanger* demuestran que la información entre que se realiza la transacción y que esta es guardada en la base de datos es prácticamente el mismo.

Por último, se procede a la compra de 0.01 BTC pero a un precio de $15.000, cosa que provocará un error puesto que no se podrá realizar la transacción:

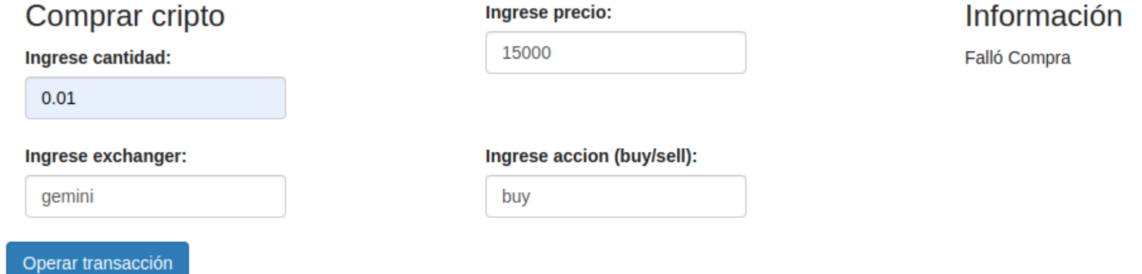
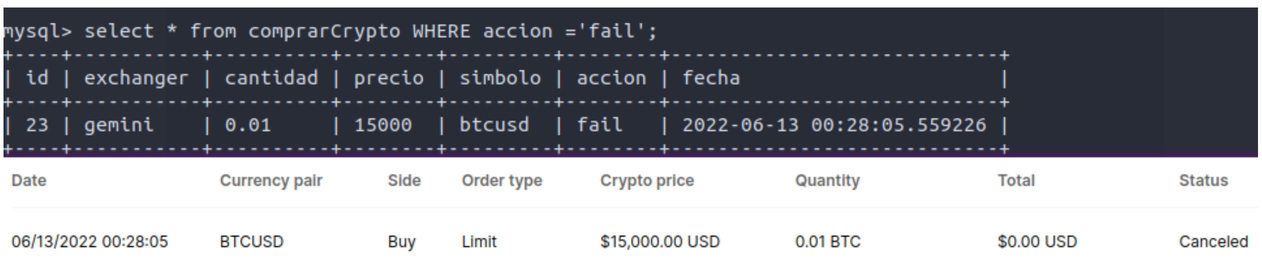


Figura 5.3: Segunda compra BTC, vista en Moqui.



```
mysql> select * from comprarCrypto WHERE accion ='fail';
+----+-----------+----------+--------+---------+--------+----------------------------+
| id | exchanger | cantidad | precio | simbolo | accion | fecha                      |
+----+-----------+----------+--------+---------+--------+----------------------------+
| 23 | gemini    | 0.01     | 15000  | btcusd  | fail   | 2022-06-13 00:28:05.559226 |
+----+-----------+----------+--------+---------+--------+----------------------------+
```

Date	Currency pair	Side	Order type	Crypto price	Quantity	Total	Status
06/13/2022 00:28:05	BTCUSD	Buy	Limit	$15,000.00 USD	0.01 BTC	$0.00 USD	Canceled

Figura 5.4: Registro historial Exchanger y base de datos.

Se puede apreciar en este caso que el “status” del historial ofrecido por el *exchanger* arroja que fue cancelado y al mismo tiempo en la base de datos se puede apreciar que la columna “accion” arroja *fail*, que precisamente alude a que no realizó la transacción.

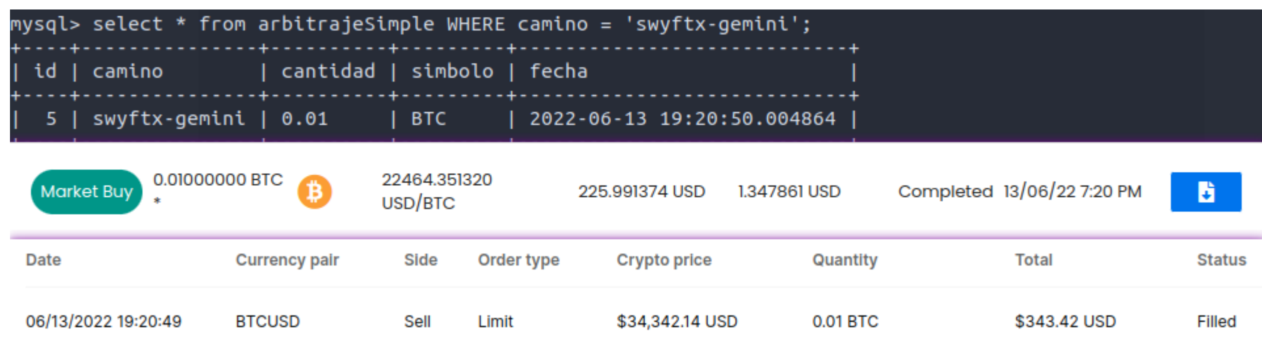
5.1.2. Arbitraje Simple

Para comenzar con la evaluación de este tipo de arbitraje, se realizaron dos transacciones exitosas. Una transacción exitosa se verifica cuando se realiza correctamente la compra de una criptomoneda en un *exchanger* y luego se realiza su venta en otro. Podemos visualizar la primera prueba de transacción de la siguiente manera:

Figura 5.5: Registro Moqui, arbitraje simple.

En primer lugar, se puede apreciar que la transacción se realizó con éxito, pero que el precio de compra de Swyftx no es el mismo que aparece en la caja de información. Si bien esto se debe a las limitaciones que tiene su API, también se debe a que el entorno de pruebas de Swyftx es el que más usuarios activos tiene y en el que más transacciones se realizan en tiempo real. Desde el momento que se cargó el precio y se realizó el arbitraje el precio cambió radicalmente debido al alto flujo de transacciones realizadas por el resto de los usuarios, sin embargo, el sistema igualmente tomó el mejor precio posible para realizar la transacción.

Por otro lado, de un contraste de la información de la base de datos y los historiales de ambos *exchangers*, se puede apreciar:



```
mysql> select * from arbitrajeSimple WHERE camino = 'swyftx-gemini';
+----+---------------+----------+---------+----------------------------+
| id | camino        | cantidad | simbolo | fecha                      |
+----+---------------+----------+---------+----------------------------+
|  5 | swyftx-gemini | 0.01     | BTC     | 2022-06-13 19:20:50.004864 |
```

Market Buy	0.01000000 BTC *	22464.351320 USD/BTC	225.991374 USD	1.347861 USD	Completed	13/06/22 7:20 PM

Date	Currency pair	Side	Order type	Crypto price	Quantity	Total	Status
06/13/2022 19:20:49	BTCUSD	Sell	Limit	$34,342.14 USD	0.01 BTC	$343.42 USD	Filled

Figura 5.6: Registro base de datos e historial de Swyftx y Gemini respectivamente.

En la figura anterior se aprecia que en la base de datos se arroja el camino correcto, puesto que “swyftx-gemini” alude que la primera transacción, que es de compra, fue realizada en Swyftx y luego la segunda, que es de venta, fue realizada en Gemini. Más importante aún, los datos que arrojó Moqui al realizar la transacción estaban correctos y la fecha de la transacción está correcta tanto en los dos *exchangers* como en la base de datos.

En la segunda prueba, se realiza el siguiente arbitraje simple en Moqui:

Figura 5.7: Vista segundo arbitraje simple en Moqui

Podemos ver que se realiza una compra de 0.01 BTC en Gemini y posteriormente se vende esa misma cifra en Coinbase. Si revisamos el historial que ofrece Gemini y Coinbase respectivamente:

Date	Currency pair	Side	Order type	Crypto price	Quantity	Total	Status
06/14/2022 18:12:15	BTCUSD	Buy	Limit	$21,577.97 USD	0.01 BTC	$215.78 USD	Filled

Side	Market	Size	Filled	Filled Price	Fee	Date
Sel	BTC/U...	0.01000000 BTC	0.01000000 BTC	$24,308.57	$0.97	Jun 14, 2022 - 06:12:15 PM...

Figura 5.8: Historial de Gemini y Coinbase respectivamente.

Se puede observar que la información entregada por los *exchangers* es idéntica a la de Moqui, por lo que se puede concluir que el arbitraje fue un éxito. Además de los *exchangers*, se puede ver que lo guardado en la base de datos coincide correctamente con el camino escogido y también calza con la fecha perfectamente:

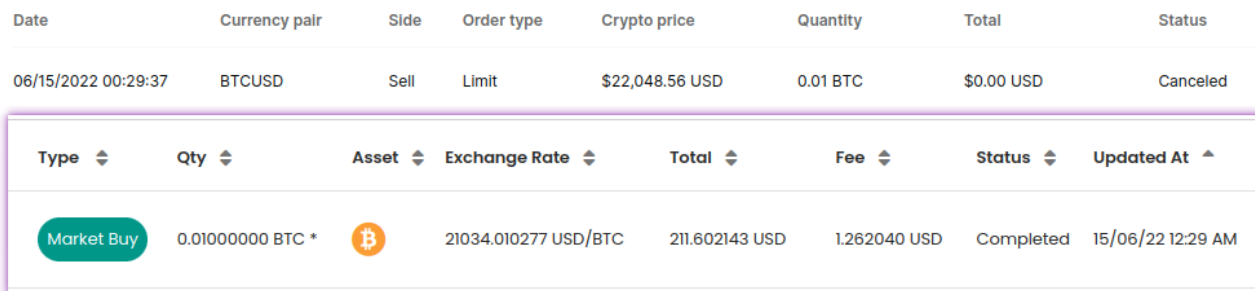
```
mysql> select * from arbitrajeSimple where camino LIKE 'gemini-%';
+----+-----------------+----------+---------+----------------------------+
| id | camino          | cantidad | simbolo | fecha                      |
+----+-----------------+----------+---------+----------------------------+
| 15 | gemini-coinbase | 0.01     | BTC     | 2022-06-14 18:12:15.787627 |
+----+-----------------+----------+---------+----------------------------+
1 row in set (0,00 sec)
```

Figura 5.9: Base de datos después del segundo arbitraje simple.

Aparte de los casos de éxito, también se soportan errores, es decir, al obtener algún fallo, o simplemente alguna transacción que no pueda ser ejecutada, el programa lo arroja inmediatamente. Por ejemplo, si el sistema es capaz de efectuar la compra pero no así la venta del criptoactivo, se tendría lo siguiente en Moqui:

Figura 5.10: Interfaz en Moqui después de una transacción fallida.

Se puede ver que faltó la venta en Gemini. Por otro lado, al revisar la información de ambos *exchangers*:



Date	Currency pair	Side	Order type	Crypto price	Quantity	Total	Status
06/15/2022 00:29:37	BTCUSD	Sell	Limit	$22,048.56 USD	0.01 BTC	$0.00 USD	Canceled

Type	Qty	Asset	Exchange Rate	Total	Fee	Status	Updated At
Market Buy	0.01000000 BTC *		21034.010277 USD/BTC	211.602143 USD	1.262040 USD	Completed	15/06/22 12:29 AM

Figura 5.11: Historial exchangers, primero historial Gemini y segundo Swyftx.

Desde el historial de Gemini se puede apreciar cómo el estado indica que la transacción fue cancelada y desde el historial entregado por Swyftx se puede ver que la transacción fue realizada con éxito. Dicho de otro modo, el precio de venta escogido para Gemini ya no estaba disponible para realizar la transacción, por lo que al intentar realizar la transacción fue rechazada.

Desde el lado de la base de datos se puede apreciar cómo indica que la operación falló en la venta en la columna "operacion":

```
mysql> select * from arbitrajeSimple where id = 47;
+----+---------------+----------+-------------+----------------------------+
| id | camino        | cantidad | simbolo     | fecha                      |
+----+---------------+----------+-------------+----------------------------+
| 47 | swyftx-gemini | 0.01     | failed-sale | 2022-06-15 00:29:37.724896 |
+----+---------------+----------+-------------+----------------------------+
```

Figura 5.12: Base de datos, después de que fallara la transacción en Gemini.

Por último, si el sistema no detecta ningún potencial de beneficio para el arbitraje que realiza, se puede ver en Moqui:

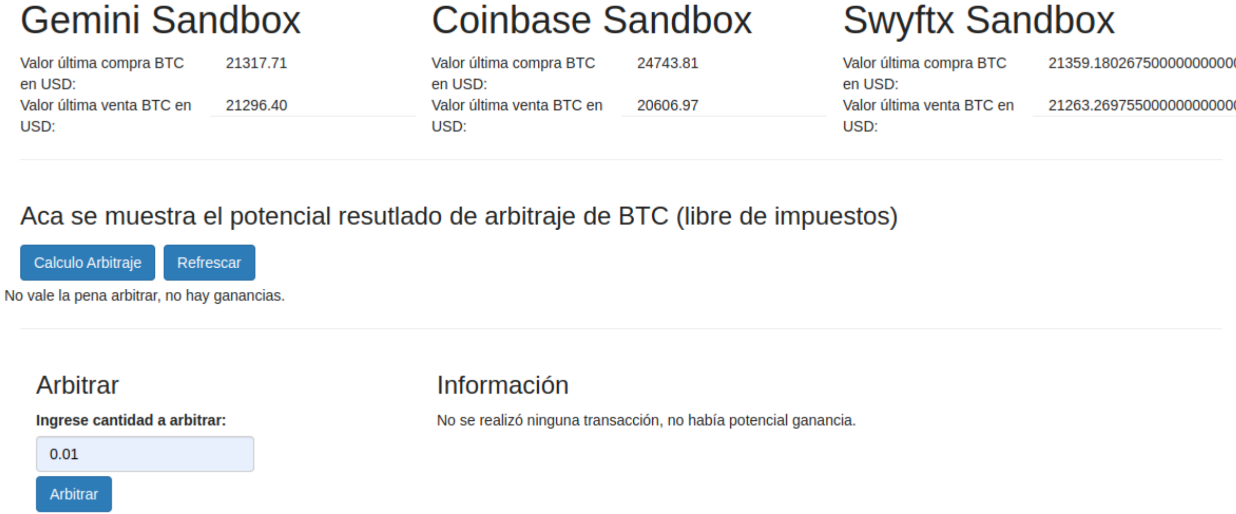


Figura 5.13: Vista de Moqui cuando no detecta beneficio para el usuario al arbitrar.

En este caso, no se realiza ninguna operación en los *exchangers*, y la base de datos almacena este caso fallido junto con la hora:

```
mysql> select * from arbitrajeSimple where camino = 'None';
+----+--------+----------+--------------------+----------------------------+
| id | camino | cantidad | simbolo            | fecha                      |
+----+--------+----------+--------------------+----------------------------+
| 46 | None   | 0.01     | failed-transaction | 2022-06-15 00:04:51.660035 |
+----+--------+----------+--------------------+----------------------------+
1 row in set (0,00 sec)
```

Figura 5.14: Base de datos después del intento fallido de arbitraje simple.

5.1.3. Arbitraje Triangular en un mismo Exchanger

Para esta sección, se realizarán las pruebas usando el *exchanger* Swyftx, puesto que este presenta un buen cambio entre ETH y BTC, cosa que los demás *exchangers* en su entorno de pruebas no ofrecían. Ahora bien, para la primera prueba, en Moqui se tiene:

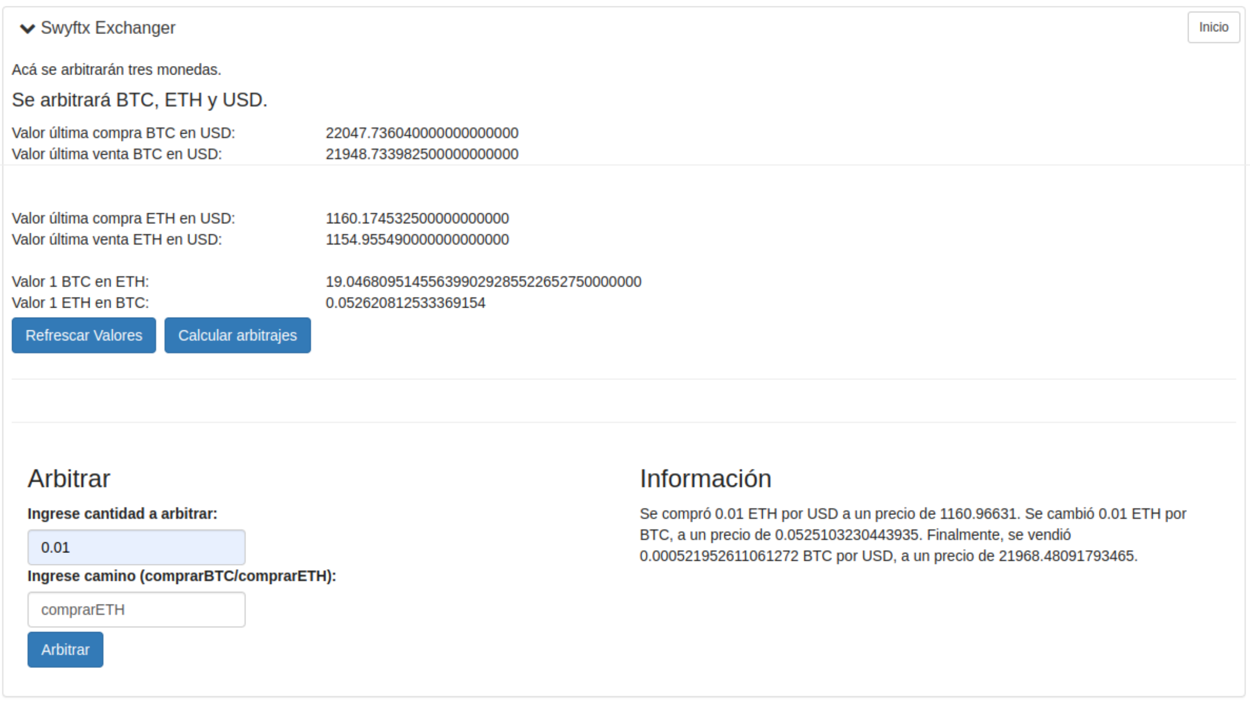


Figura 5.15: Interfaz Moqui sobre arbitraje triangular comprando ETH en un exchanger.

Se puede apreciar la transacción que parte con la compra de ETH en USD, para luego ser cambiado por BTC, para finalizar con una venta de BTC a cambio de USD. Esta vez, la diferencia entre los precios reales de compra y venta son sumamente similares a los entregados en Moqui debido a que la mayor concentración de ventas y compras en el ambiente de pruebas ofrecido por Swyftx se centra en BTC. Dicho de otro modo, los precios de ETH se mantienen un poco más estáticos y fluctúan un poco menos.

Por otro lado, y revisando lo almacenado en la base de datos y contrastando con el historial de transacciones ofrecido por Swiftx:

```
mysql> select * from arbitrajeTriangularUnitario where fecha LIKE '2022-06-14%';
+----+-----------+-----------------+----------------------+------------+----------------------------+
| id | exchanger | cantidad_inicial | cantidad_cambiada    | operacion  | fecha                      |
+----+-----------+-----------------+----------------------+------------+----------------------------+
|  3 | Swyftx    | 0.01            | 0.000521952611061272 | comprarETH | 2022-06-14 01:00:55.763806 |
+----+-----------+-----------------+----------------------+------------+----------------------------+
```

Type	Qty	Asset	Exchange Rate	Total	Fee	Status	Updated At
Market Sell	0.00052195 BTC *		21968.480917 USD/BTC	11.397706 USD	0.068799 USD	Completed	14/06/22 1:00 AM
Market Sell	0.01000 ETH *		0.05251032 BTC/ETH	0.00052195 BTC	0.00000315 BTC	Completed	14/06/22 1:00 AM
Market Buy	0.01000 ETH *		1160.966310 USD/ETH	11.679321 USD	0.069657 USD	Completed	14/06/22 1:00 AM

Figura 5.16: Base de datos e historial Swyftx luego de realizar arbitraje triangular.

Se puede revisar que todos los datos coinciden correctamente, tanto con lo mostrado en Moqui como lo reflejado en el historial y la base de datos. A modo de análisis, se puede ver

que el usuario, si estuviese gastando efectivo real, tiene una tendencia clara a perder dinero. En esta transacción hubo una pérdida de 0.3 USD.

Con respecto a una transacción realizada con el camino de comprar BTC a cambio de USD, para luego cambiar ese BTC por ETH, para finalizar con una venta de ETH, se tiene:

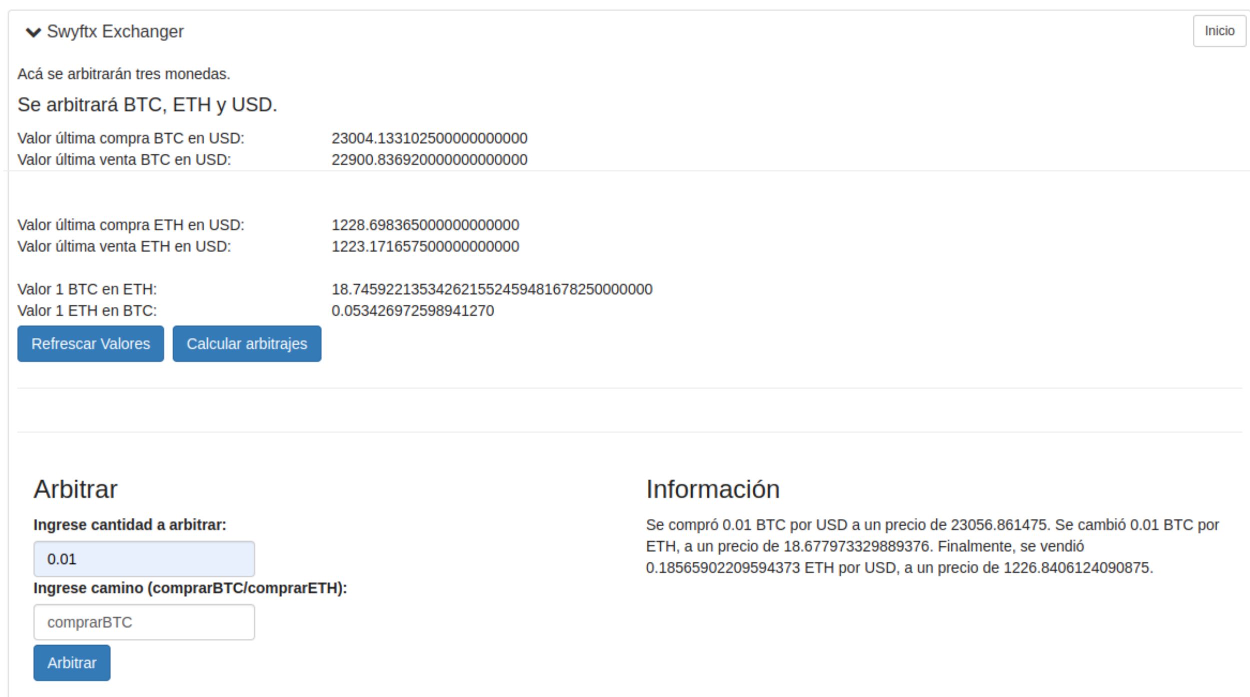


Figura 5.17: Interfaz Moqui sobre arbitraje triangular comprando BTC en un exchanger

Se puede apreciar que esta vez la fluctuación entre el precio de compra de BTC y de venta de ETH que ofrece Moqui, comparado con el resultante de la transacción real parecen bastante similares ($\pm$50USD y $\pm$3USD respectivamente), lo que puede influir de gran manera en los resultados finales de la transacción del arbitraje. Esto se refleja más aún cuando se revisa el historial de la transacción en Swyftx:

Type	Qty	Asset	Exchange Rate	Total	Fee	Status	Updated At
Market Sell	0.18571 ETH *		1226.840612 USD/ETH	226.479796 USD	1.366643 USD	Completed	14/06/22 1:31 AM
Market Sell	0.01000000 BTC *		18.67797 ETH/BTC	0.18565 ETH	0.00112 ETH	Completed	14/06/22 1:31 AM
Market Buy	0.01000000 BTC *		23056.861475 USD/BTC	231.952026 USD	1.383411 USD	Completed	14/06/22 1:31 AM

Figura 5.18: Historial Swyftx una vez se realiza el arbitraje triangular

Revisando los USD con que se partieron y con los que se terminaron, hubo una pérdida de 5 USD. Si bien parece poco, cabe destacar que se operó con 0.01 BTC, así que con mayores cantidades la pérdida sería mucho mayor. Por último, se puede revisar la base de datos que nuevamente guarda correctamente la información:

```
mysql> select * from arbitrajeTriangularUnitario where id = 4;
+----+-----------+-----------------+---------------------+-----------+----------------------------+
| id | exchanger | cantidad_inicial | cantidad_cambiada   | operacion | fecha                      |
+----+-----------+-----------------+---------------------+-----------+----------------------------+
|  4 | Swyftx    | 0.01            | 0.18565902209594373 | comprarBTC | 2022-06-14 01:31:21.475493 |
+----+-----------+-----------------+---------------------+-----------+----------------------------+
1 row in set (0,00 sec)
```

Figura 5.19: Información base de datos después de realizar arbitraje triangular en Swyftx.

5.1.4. Arbitraje Triangular en diversos Exchangers

Se vuelve a destacar que la transacción entre cambiar BTC por ETH (o al revés), es decir, comprar BTC pagando con ETH o vender BTC para obtener ETH no está bien soportada en los *exchangers* de Gemini y Coinbase, pero si está bien soportado por Swyftx. En vista de lo anterior el usuario en esta interfaz de arbitraje solamente puede operar usando el camino dos y tres, los cuales corresponden a:

- *camino2:* El arbitraje después de comprar una cantidad de ETH en Gemini, para luego vender ese ETH a cambio de BTC en Swyftx, para posteriormente vender esa cantidad de BTC en Coinbase.
- *camino3:* El arbitraje después de comprar una cantidad de BTC en Coinbase, para luego vender BTC a cambio de ETH en Swyftx y finalizar vendiendo esa cantidad ETH en Gemini.

Para comenzar, se revisa el caso en que un usuario opta por el *camino2*, concluyendo en que la transacción es un éxito. En Moqui se tendría:

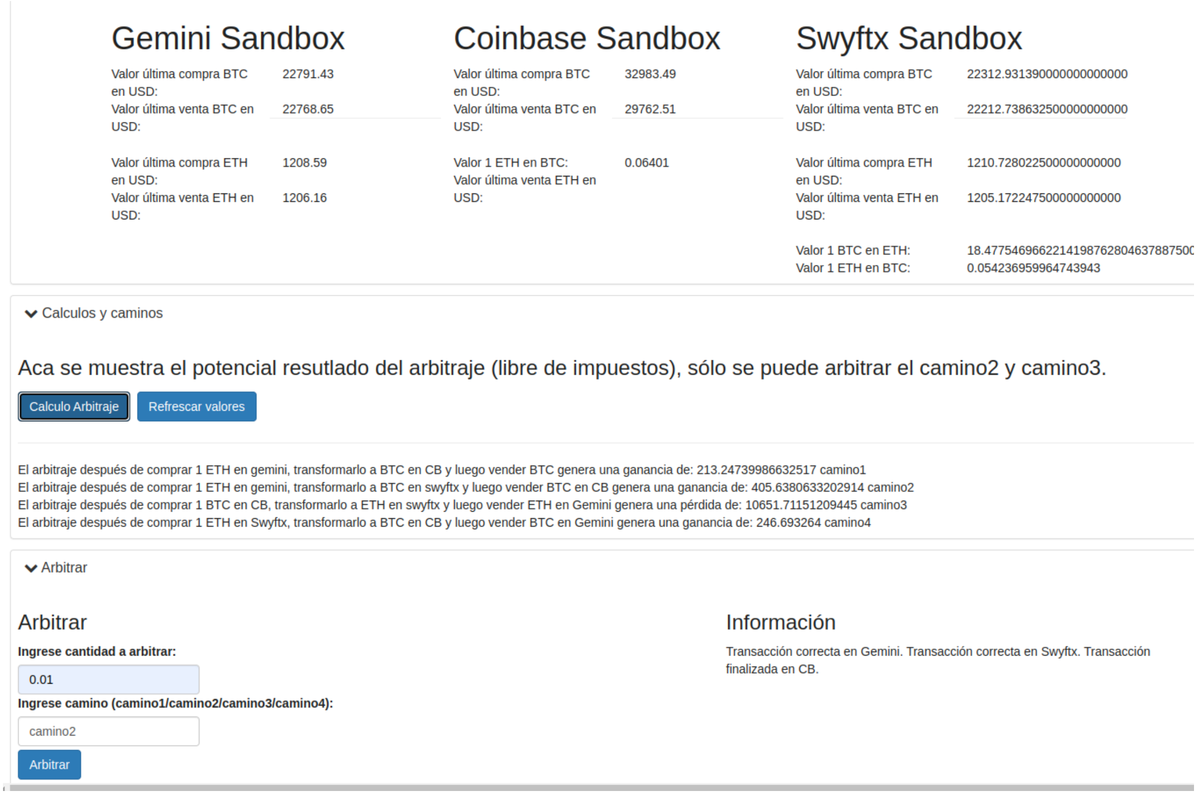


Figura 5.20: Interfaz de Moqui al arbitrar con éxito.

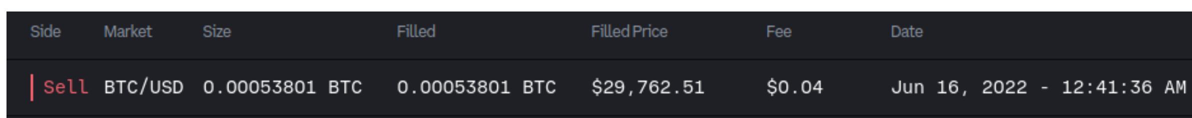
Se le informa al usuario que las transacciones en los tres *exchangers* fueron un éxito, y si se contrasta con la información de los historiales de los *exchangers* se tiene:

Date	Currency pair	Side	Order type	Crypto price	Quantity	Total	Status
06/16/2022 00:41:35	ETHUSD	Buy	Limit	$1,208.59 USD	0.01 ETH	$12.09 USD	Filled

Figura 5.21: Historial de Gemini.

Type	Qty	Asset	Exchange Rate	Total	Fee	Status	Updated At	Action
Market Sell	0.01000 ETH *		0.05412661 BTC/ETH	0.00053801 BTC	0.00000324 BTC	Completed	16/06/22 12:41 AM	

Figura 5.22: Historial de Swyftx.



Side	Market	Size	Filled	Filled Price	Fee	Date
Sell	BTC/USD	0.00053801 BTC	0.00053801 BTC	$29,762.51	$0.04	Jun 16, 2022 - 12:41:36 AM

Figura 5.23: Historial de Coinbase.

Y por último, la información guardada en la base de datos arroja lo siguiente:

```
mysql> select * from arbitrajeTriangularMultiple where id = 11;
+----+-----------------+-------------------+---------+----------------------------+
| id | cantidad_inicial | cantidad_cambiada | camino  | fecha                      |
+----+-----------------+-------------------+---------+----------------------------+
| 11 | 0.01             | 0.00053801        | camino2 | 2022-06-16 00:41:36.696928 |
+----+-----------------+-------------------+---------+----------------------------+
```

Figura 5.24: Base de datos luego de la transacción.

Se puede apreciar que la transacción final (que fue la correspondiente a Coinbase) terminó a un segundo después de la transacción inicial (correspondiente a la de Gemini), que si bien, no es una diferencia de tiempo muy alarmante, queda en claro que no es tan instantáneo como el resto de los arbitrajes.

Adicionalmente, revisando los casos de error, cuando alguna transacción no es completada se avisa al usuario y se guarda en la base de datos el paso en que se falla. A modo de ejemplo, cuando la transacción correspondiente al *camino3* falla en el último paso:

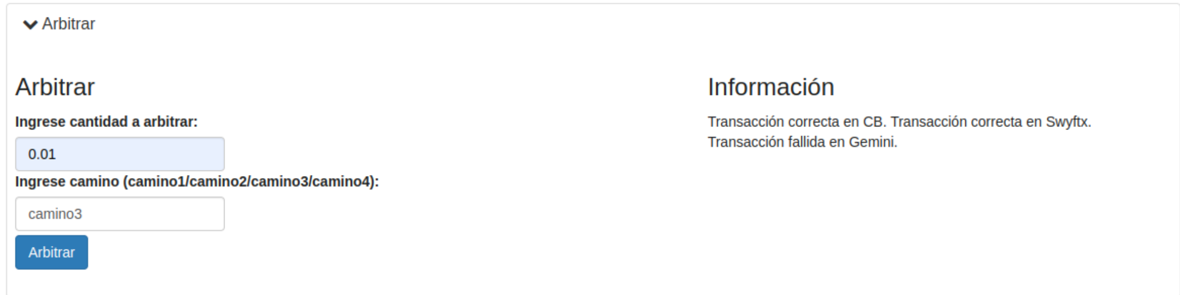


Figura 5.25: Transacción fallida en el paso final.

Como bien se aprecia en la caja de información, la transacción falló en Gemini, que era el *exchanger* encargado de realizar la venta de ETH. En la base de datos, para esta transacción se puede ver:

```
mysql> select * from arbitrajeTriangularMultiple where id = 14;
+----+------------------+-------------------+---------------------+----------------------------+
| id | cantidad_inicial | cantidad_cambiada | camino              | fecha                      |
+----+------------------+-------------------+---------------------+----------------------------+
| 14 | 0.01             | 0.183972          | camino3-failed-sale | 2022-06-16 02:44:37.927033 |
+----+------------------+-------------------+---------------------+----------------------------+
```

Figura 5.26: Base de datos en la venta fallida.

Queda claro que el camino arroja un fallo en la venta, pero aún así la cantidad de criptomoneda inicial está intacta y la cantidad cambiada también, esto se debe a que la primera parte de la transacción, que corresponde a la compra de BTC, si se efectúo por una cantidad de 0.1 y el cambio de criptomoneda, de BTC a ETH también fue un éxito, obteniendo 0.183972 a cambio de ese 0.1 inicial.

Por último, independiente si el usuario escoge el *camino2* o *camino3*, si es que la primera transacción falla, el sistema no permite seguir con el resto de las operaciones y simplemente guarda el fallo en la base de datos. Para ejemplificar este caso, si es que el arbitraje falla al inicio del *camino2*:

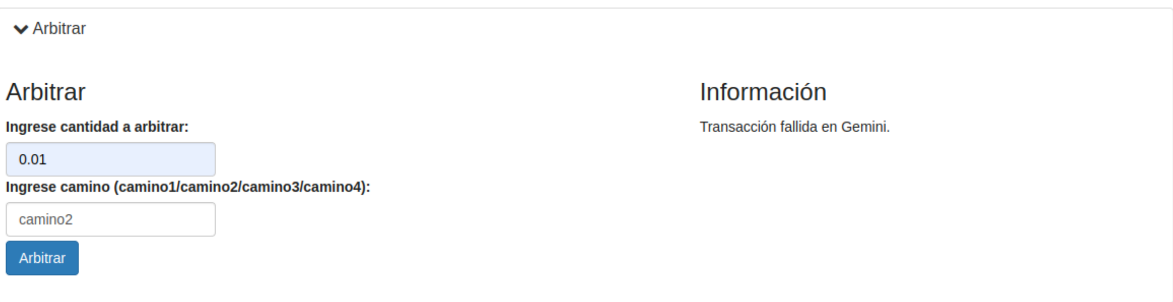


Figura 5.27: Fallo desde el inicio en Moqui.

```
mysql> select * from arbitrajeTriangularMultiple where id = 15;
+----+------------------+-------------------+------------------------+----------------------------+
| id | cantidad_inicial | cantidad_cambiada | camino                 | fecha                      |
+----+------------------+-------------------+------------------------+----------------------------+
| 15 | 0                | 0                 | camino2-failed-purchase | 2022-06-16 02:56:46.731593 |
+----+------------------+-------------------+------------------------+----------------------------+
```

Figura 5.28: Base de datos con la transacción fallida desde el inicio.

Nuevamente el camino arroja en qué parte de la transacción falló, que en este caso sería en la compra, y a diferencia del error anterior, tanto la cantidad inicial como la cambiada están en cero puesto que no hubo ningún tipo de transacción asociadas a esas cantidades.

Más información acerca de resultados pueden ser revisados en el apartado del Anexo B, en dónde se muestra información almacenada en la base de datos acerca de diversas transacciones realizadas. Estos resultados son interesantes de analizar puesto que muchas transacciones resultaron fallidas, y se evidencia el comportamiento del sistema a fallos puntuales. La idea detrás de estas pruebas es obtener fallos y éxitos para revisar el comportamiento de la aplicación e ir probando su robustez.

5.2. Discusión de los resultados expuestos

Con respecto a la compra o venta de criptomoneda se puede apreciar que es un sistema bastante sencillo, pero que entrega una alta utilidad al usuario. Es valioso para él poder comprar activos sin la necesidad de salir de Moqui y tener que dirigirse a un *exchanger*, y por su puesto, es de vital importancia poder comprar o vender estos activos puesto que es la base de todas las transacciones.

En cuanto a los resultados obtenidos se entrega un satisfactorio prontuario para toda compra y venta de criptomoneda que soporta el sistema. Sin embargo, se debe tener alta precaución con el precio de la transacción, puesto que los precios oscilan bastante rápido, y otro usuario puede efectuar una transacción por el precio que uno quiere y quitarnos la oportunidad de operar.

Por otro lado, respecto del arbitraje simple, se pudo concluir que funciona sin problemas entre los *exchangers*, e incluso es posible destacar el tiempo en que se manda la solicitud a los *exchangers* para realizar las transacciones, y el tiempo en que se demora en ejecutarse y luego recibir la respuesta de los *exchangers*. Este periodo es sumamente pequeño, de hecho es prácticamente inmediato. Esto último es algo sumamente relevante para este sistema puesto que el tiempo es vital para realizar las transacciones y mientras menos se demore, mejor serán los resultados obtenidos.

Por su parte, los arbitrajes triangulares, tanto para el caso del arbitraje triangular simple (en un mismo *exchanger*) y el múltiple (en diversos *exchangers*), el tiempo de espera que se mencionaba anteriormente también es casi inmediato, llegando a tardar máximo 1 segundo entre que se realizaba la primera compra hasta la última venta de los criptoactivos. Ahora bien, este tipo de arbitraje es el que más probabilidades de fallo tiene, puesto que se está revisando tres *exchangers* con tres precios distintos, entonces es más probable que el usuario busque una oportunidad a un precio específico, pero este cambie en alguno de los tres *exchangers*. Algo que ayudó a hacer un poco más versátil este arbitraje, fue precisamente

que el *exchanger* Swyftx tomara el precio más cercano (y beneficioso) al que el usuario tiene en Moqui para realizar el arbitraje, esto permitió realizar más transacciones. No obstante lo anterior, esto también tiene una desventaja, puesto que esa diferencia de dinero puede provocar justamente que una potencial ganancia en el arbitraje se convierta en una pérdida para el usuario.

En síntesis, si bien se pudieron realizar arbitrajes en Moqui y también el cálculo de las oportunidades de un potencial arbitraje beneficioso fue correcto (que eran los objetivos principales a concretar), el mayor problema común fue el tiempo y rapidez con que cambian los precios de las criptomonedas en los *exchangers*. En los ambientes de prueba *sandbox* este sistema se comporta de buena manera puesto que no es tan dinámico, pudiendo realizar la mayoría de las transacciones cuando se intentan, pero en un ambiente de producción, en donde hay muchísimas transacciones por segundo [17], es posible que una buena parte de los intentos de arbitrajes sean rechazado por el sistema justamente por algún problema con el precio entregado en Moqui.

5.3. Análisis de cumplimiento de los objetivos

Con respecto al objetivo general, se logra de buena manera un sistema capaz de informar al usuario si es que existe alguna oportunidad de ganancia (o incluso pérdida) al intentar realizar el arbitraje con las criptodivisas entregadas por el programa. También se logró realizar transacciones con criptomonedas a través de Moqui, que mezclado con el cálculo de oportunidades de arbitraje, logró concretar la base de este sistema de arbitraje, y cumplir el objetivo general. Por último, si bien se centró el trabajo en un ambiente de pruebas *(sandbox)*, el paso a producción, es decir, utilizar el sistema real ofrecido por los *exchangers*, sería siguiendo la misma lógica, por lo que da un buen pie a futuro para seguir trabajando con este sistema.

Hilando un poco más fino, la conexión con las APIs ofrecidas por los *exchangers* fue todo un éxito, esto permitió no sólo generar las transacciones anteriormente mencionadas, sino que también permite entregar información necesaria para llevar a cabo mucho de los objetivos específicos. Estos objetivos en particular están relacionados con la entrega de información acerca de las criptomonedas almacenadas en las distintas E-Wallets del usuario, y también está relacionado con los historiales de transacciones realizadas en los distintos *exchangers*. Y por último, esta correcta conexión entre las APIs y Moqui, permitió de buena manera formalizar las transacciones que permitieron generar arbitrajes simples y triangulares en Moqui, que dieron vida al sistema de arbitraje en general realizado en Moqui.

A modo de cierre, tanto objetivos generales como específicos fueron cubiertos de buena manera en este sistema, sólo quedaron oportunidades de mejora que serán comentados en el siguiente capítulo.

Capítulo 6

Conclusión

6.1. Retrospectiva

La creación de un sistema de arbitraje de criptomonedas en Moqui resultó ser un método exitoso para introducir y evaluar el tratamiento de criptodivisas en este *framework*. Lo anterior permitió estudiar el comportamiento o "simbiosis" de Moqui con el ecosistema de los criptoactivos, a saber, su conexión con los *exchangers*, el almacenamiento de información de transacciones, visualización de valores, entre otros.

Con respecto al estudio de las criptomonedas y los *exchangers*, en general, fue todo un desafío. El mundo de las criptodivisas es increíblemente amplio, en donde podemos encontrar una infinidad de conceptos, variantes, análisis tanto económicos como técnicos, que impactan directamente al momento de desarrollar un sistema de arbitraje.

Por su parte, elaborar un sistema de arbitraje también implicó estudiar a fondo su funcionamiento, las diferentes características entre los distintos tipos de arbitrajes que existen, y lo más fundamental, cómo se relacionan con el peculiar comportamiento de las transacciones de criptomonedas en el mercado y en los *exchangers*. Sin lugar a dudas, un aspecto vital para el desarrollo de lo anterior, fue la utilización de la *testnet*, un *blockchain* que permite realizar pruebas convincentes del sistema de arbitraje en estudio, obteniendo resultados cercanos al ambiente de producción.

Por último, es importante destacar que el estudio del *framework* Moqui fue rico en aprendizaje. Presentó diversas dificultades, entre estas su sistema basado en Groovy y XML que fue una barrera difícil de superar, junto con la escasa documentación disponible diferencia de lo que ocurre con otros *frameworks* trabajados en la actualidad. Sin embargo, la ayuda del profesor guía y del equipo de Moit fue vital para poder completar los objetivos planteados para este sistema de arbitraje.

6.2. Trabajo Futuro y conclusiones finales

Si bien se logró cumplir el objetivo general de esta memoria, existen diversas oportunidades que pueden ser aprovechadas para lograr mejores resultados y tener un sistema aún más robusto y completo.

En primer lugar, se podría utilizar la misma base de datos que ofrece Moqui para almacenar la información relacionada con las transacciones realizadas en el sistema, en vez de utilizar un sistema externo con MySQL. Esto ayudaría primero, a no utilizar herramientas externas a Moqui y también evitaría complejizar aún más la aplicación.

Por otro lado, a pesar de que obtener ganancias no era el objetivo central de la elaboración de este sistema de arbitraje, sin lugar a dudas es una meta futura compleja que necesita mucha más información de la que entrega este proyecto. Por una parte, se debería tener acceso a una mayor cantidad de *exchangers*, esto debido a que se manejarían más precios que permitirán encontrar más posibilidades de arbitrajes que permitan generar potenciales ganancias. De la mano con esto, también es importante tener a disposición más criptomonedas que las presentes en esta memoria, y la razón detrás de esto es exactamente la misma, con un mayor número de monedas habrán más comparaciones de precios y por lo tanto, mayores oportunidades de tener un arbitraje que resulte eficiente para el usuario.

Por último, un desafío sumamente relevante para el sistema sería solucionar el problema evidenciado en los resultados acerca de los precios. Actualmente el sistema arroja el último precio de la criptodivisa en el *exchanger* y se la ofrece al usuario para que este realice una transacción con él, pero en caso de que este precio se actualice justo en el momento que se actualiza en Moqui o si el usuario le toma mucho tiempo en realizar la transacción es probable que esta última falle debido a alguna diferencia entre el precio arrojado en Moqui con el precio real que está en el *exchanger*.

Sin lugar a dudas, tener una comunicación en tiempo real con los *exchangers* cobra vital importancia en este sistema, si se concreta una comunicación de este tipo el usuario en Moqui podría ver reflejado todos los cambios de precio a medida que vayan cambiando en los mismos *exchangers*, y así evitar una potencial transacción fallida. Una forma de implementar esto sería a través de *web sockets*, que precisamente ofrecen una comunicación en tiempo real que facilita al usuario poder revisar la información que él necesite. Y por si fuera poco, las APIs que ofrecen los *exchangers* como Gemini [18] y Coinbase [19] tienen soporte para el uso de *websockets*, por lo que sería un buen punto de partida para cubrir esta funcionalidad.

Bibliografía

[1] "Criptomonedas en el mundo." Disponible en https://www.cnbc.com/2021/08/18/new-cryptocurrency-bitcoin-user-global-map.html, última vez que se visitó: 01/07/22.

[2] "Volatilidad de las criptomonedas." Disponible en https://www.zawya.com/en/special-coverage/the-future-of-cryptos/why-are-bitcoin-other-cryptos-so-volatile-mv36s9zk, última vez que se visitó: 01/07/22.

[3] "Condiciones arbitraje." Disponible en https://www.eleconomista.es/diccionario-de-economia/arbitraje-financiero, última vez que se visitó: 01/07/22.

[4] Coindesk, "Bajo riesgo arbitraje." Disponible en https://www.coindesk.com/learn/crypto-arbitrage-trading-how-to-make-low-risk-gains/, última vez que se visitó: 01/07/22.

[5] Moit, "Servicios moit." Disponible en https://moit.cl/#serv, última vez que se visitó: 01/07/22.

[6] "Moqui." Disponible en https://www.moqui.org/docs/framework/Introduction, última vez que se visitó: 01/07/22.

[7] "Auge de las criptomonedas." Disponible en https://blog.centrodeelearning.com/2021/05/26/auge-de-las-criptomonedas/, última vez que se visitó: 01/07/22.

[8] "¿qué es es una api?." Disponible en https://www.mulesoft.com/es/resources/api/what-is-an-api, última vez que se visitó: 01/07/22.

[9] "Testnets." Disponible en https://academy.bit2me.com/en/que-es-testnet/, última vez que se visitó: 01/07/22.

[10] "Minado de criptomonedas." Disponible en https://www.investopedia.com/tech/how-does-bitcoin-mining-work/, última vez que se visitó: 01/07/22.

[11] "Dificultad del minado." Disponible en https://river.com/learn/terms/t/testnet/, última vez que se visitó: 01/07/22.

[12] "Criptomonedas más negociadas." Disponible en https://www.plus500.com/es/Trading/CryptoCurrencies/What-are-the-Most-Traded-Cryptocurrencies~2, última vez que se visitó: 01/07/22.

[13] "Cross-origin resource sharing (cors)." Disponible en https://developer.mozilla.org/es/docs/Web/HTTP/CORS, última vez que se visitó: 01/07/22.

[14] "Gemini api." Disponible en https://docs.gemini.com/rest-api/, última vez que se visitó: 01/07/22.

[15] "Coinbase api." Disponible en https://docs.cloud.coinbase.com/exchange/docs/sandbox, última vez que se visitó: 01/07/22.

[16] "Swyftx api." Disponible en https://swyftx.docs.apiary.io/, última vez que se visitó:

01/07/22.

[17] "Cantidad de transacciones de btc." Disponible en https://towardsdatascience.com/the-blockchain-scalability-problem-the-race-for-visa-like-transaction-speed-5cce48f9d44, última vez que se visitó: 01/07/22.

[18] Gemini, "Api websockets gemini." Disponible en https://docs.gemini.com/websocket-api/, última vez que se visitó: 01/07/22.

[19] Coinbase, "Api websockets coinbase." Disponible en https://docs.cloud.coinbase.com/prime/docs/websocket-feed, última vez que se visitó: 01/07/22.

Anexo A

Apéndice Código

A.1. Servicios Moqui

Código A.1: Servicio get#Historial.

```

    <service verb="get" noun="Historial">
    <out-parameters>
        <parameter name="historialGemini"/>
        <parameter name="historialCoinbase"/>
        <parameter name="historialSwyftx"/>
    </out-parameters>
    <actions>
        <set field="requestUrl" value="http://127.0.0.1:10000/cgi-bin/getHistory.py"/>
        <set field="contentType" value="application/json"/>
        <script><![CDATA[
            try {
                restClient = new
↪ org.moqui.util.RestClient().method('GET').contentType(contentType).uri(requestUrl)
                response = restClient.call()
                responsePayload = response.text()
                jsonSlurper = new groovy.json.JsonSlurperClassic()
                responseMap = jsonSlurper.parseText(responsePayload)
                responsePrint = groovy.json.JsonOutput.prettyPrint(responsePayload)
                historialGemini = responseMap.gemini
                historialCoinbase = responseMap.coinbase
                historialSwyftx = responseMap.swyftx
            } catch (Exception e) {
                failReason = e.toString()
            }
        ]]></script>
        <if condition="failReason">
            <log level="error" message="failReason: ${failReason}"/>
        </if>
    </actions>
</service>


```

Código A.2: Servicio get#HistorialTransacciones.

```
1
2       <service verb="get" noun="HistorialTransacciones">
3       <out-parameters>
4           <parameter name="comprarCrypto"/>
5           <parameter name="arbitrajeSimple"/>
6           <parameter name="arbitrajeTriangularUnitario"/>
7           <parameter name="arbitrajeTriangularMultiple"/>
8       </out-parameters>
9       <actions>
10          <set field="requestUrl"
    ↪ value="http://127.0.0.1:10000/cgi-bin/getTransacciones.py"/>
11          <set field="contentType" value="application/json"/>
12          <script><![CDATA[
13              try {
14                  restClient = new
    ↪ org.moqui.util.RestClient().method('GET').contentType(contentType).uri(requestUrl)
15                  response = restClient.call()
16                  responsePayload = response.text()
17                  jsonSlurper = new groovy.json.JsonSlurperClassic()
18                  responseMap = jsonSlurper.parseText(responsePayload)
19                  responsePrint = groovy.json.JsonOutput.prettyPrint(responsePayload)
20                  comprarCrypto = responseMap.comprarCrypto
21                  arbitrajeSimple = responseMap.arbitrajeSimple
22                  arbitrajeTriangularUnitario = responseMap.arbitrajeTriangularUnitario
23                  arbitrajeTriangularMultiple = responseMap.arbitrajeTriangularMultiple
24              } catch (Exception e) {
25                  failReason = e.toString()
26              }
27          ]]></script>
28          <if condition="failReason">
29              <log level="error" message="failReason: ${failReason}"/>
30          </if>
31      </actions>
32    </service>
33
34
```

A.2. Python

Código A.3: Clase que autentica al usuario para hacer un llamado a la API de Coinbase.

```
1     class CoinbaseExchangeAuth(AuthBase):
2         def __init__(self, api_key, secret_key, passphrase):
3             self.api_key = api_key
4             self.secret_key = secret_key
5             self.passphrase = passphrase
6
7         def __call__(self, request):
```

```
            timestamp = str(tiempo)
            message = timestamp + request.method + request.path_url + (request.body or
    ↪ b'').decode()
            hmac_key = base64.b64decode(self.secret_key)
            signature = hmac.new(hmac_key, message.encode(), hashlib.sha256)
            signature_b64 = base64.b64encode(signature.digest()).decode()

            request.headers.update({
                'CB-ACCESS-SIGN': signature_b64,
                'CB-ACCESS-TIMESTAMP': timestamp,
                'CB-ACCESS-KEY': self.api_key,
                'CB-ACCESS-PASSPHRASE': self.passphrase,
                "Accept": "application/json",
                'Content-Type': 'application/json'
            })
            return request

```

Código A.4: Función que crea transacción en Coinbase.

```
def transaccion_coinbase(camino, precio, canitdad):

    #sandbox api keys and url
    API_KEY= os.environ[COINBASE-APIKEY]
    API_PASS= os.environ[COINBASE-PASS]
    API_SECRET= os.environ[COINBASE-APISECRET]

    #sandbox url
    api_url='https://api-public.sandbox.exchange.coinbase.com/'
    auth = CoinbaseExchangeAuth(API_KEY, API_SECRET, API_PASS)

    endpoint = 'orders'

    # Place an order
    payload = {
        "type" : "limit",
        "side": camino,
        'price': precio,
        "time_in_force": "IOC",
        "size": canitdad,
        "product_id": "BTC-USD"
    }
    r = requests.post(api_url + endpoint, json=payload, auth=auth)

    return json.dumps(r.json())


```

Código A.5: Función que crea transacción en Swyftx, usando USD como medio de pago.

```python

def swyftx_operar(cantidad, moneda , side):

    operacion = 0
    if(side == 'buy'):
        operacion = 1
    else:
        operacion = 2


    headers = {
    'Content-Type': 'application/json',
    'Authorization': JWT
    }

    url= 'https://api.demo.swyftx.com.au/'
    endpoint = 'orders/'

    values = {
        "primary": "USD",
        "secondary": moneda,
        "quantity": cantidad,
        "assetQuantity": moneda,
        "orderType": operacion
    }

    string_json = json.dumps(values)

    response = requests.post(url+endpoint, data=string_json, headers=headers)

    return json.dumps(response.json())


```

Código A.6: Función que crea transacción en Swyftx, cambiando una criptomoneda a cambio de otra critpomoneda.

```python

def swyftx_operarCoin(cantidad, moneda_conseguir, moneda_vender):

    headers = {
    'Content-Type': 'application/json',
    'Authorization': JWT
    }

    url= 'https://api.demo.swyftx.com.au/'
    endpoint = 'orders/'

    values = {
        "primary": moneda_conseguir,
        "secondary": moneda_vender,
```

```
15            "quantity": cantidad,
16            "assetQuantity": moneda_vender,
17            "orderType": 2
18        }
19
20        string_json = json.dumps(values)
21
22        response = requests.post(url+endpoint, data=string_json, headers=headers)
23
24        return json.dumps(response.json())
25
26
```

Anexo B

Resultados

Se adjuntarán los resultados de diez transacciones diferentes, realizadas en intervalos de tiempo distintos.

Tabla B.1: Tabla de 10 resultados compra criptomoneda.

Exchanger	Cantidad	Precio	Simbolo	Accion	Fecha*
gemini	0.01	32772.38	btcusd	buy	2022-06-12 23:36:29.48
gemini	0.01	32772.38	btcusd	buy	2022-06-12 23:57:53.46
swyftx	0.01	21831.6409175	BTC	buy	2022-06-16 02:37:12.21
gemini	0.1	20297.62	btcusd	buy	2022-06-20 00:27:02.78
swyftx	0.001	20086.6849588	BTC	sell	2022-06-20 00:33:35.19
coinbase	0.01	27325.23	btc	sell	2022-06-20 00:45:30.83
swyftx	0.01	19997.8441375	BTC	buy	2022-06-20 00:54:32.27
gemini	0.01	12000	btcusd	fail	2022-06-20 00:57:59.24
coinbase	0.01	21236.13	btc	fail	2022-06-20 01:00:48.96
swyftx	0.01	19966.9948825	BTC	buy	2022-06-20 01:08:51.00

* Fecha truncada para ajustar tamaño de la tabla.

Tabla B.2: Tabla de 10 resultados arbitraje simple.

Camino	Cantidad	Simbolo	Fecha
swyftx-gemini	0.01	failed-sale	2022-06-15 00:29:37.724896
None	0.01	failed-transaction	2022-06-15 00:04:51.660035
swyftx-gemini	0.01	failed-sale	2022-06-14 23:57:26.771701
swyftx-gemini	0.01	BTC	2022-06-14 23:43:34.642438
swyftx-coinbase	0.012131	BTC	2022-06-14 23:40:02.937114
gemini-coinbase	0.9	failed-sale	2022-06-14 23:35:43.574752
swyftx-coinbase	0.1	BTC	2022-06-14 23:20:42.482435
gemini-coinbase	1	BTC	2022-06-14 20:21:32.754884
coinbase-gemini	0.01	BTC	2022-06-13 20:23:08.960299
swyftx-gemini	0.5	BTC	2022-06-13 20:11:28.290986

Tabla B.3: Tabla de 10 resultados arbitraje triangular en un mismo exchanger.

Exchanger	**Cant_inicial**	**Cant_cambiada***	**Operación**	**Fecha***
Swyftx	0.01	0.168600	comprarBTC	2022-06-06 22:04:35.07
Swyftx	0.01	0.0005751	comprarETH	2022-06-06 22:05:47.36
Swyftx	0.01	0.0005219	comprarETH	2022-06-14 01:00:55.76
Swyftx	0.01	0.185659	comprarBTC	2022-06-14 01:31:21.47
Swyftx	0.3	0.015945	comprarETH	2022-06-20 01:34:46.56
Swyftx	0.1	0.0053257	comprarETH	2022-06-20 01:50:35.42
Swyftx	0.01	0.184468	comprarBTC	2022-06-20 01:50:56.29
Swyftx	0.02	0.001061	comprarETH	2022-06-20 01:57:06.45
Swyftx	0.01	0.184369	comprarBTC	2022-06-20 01:57:30.04
Swyftx	0.013	0.0006842	comprarETH	2022-06-20 02:00:15.83

* Cantidades y Fecha truncadas para ajustar tamaño de la tabla.

Tabla B.4: Tabla de 10 resultados arbitraje triangular en distintos exchangers.

Cant_inicial	**Cant_cambiada**	**camino**	**Fecha**
0.001	5.84218568	camino2	2022-06-06 22:17:54.43134
0.01	0.168104	camino3	2022-06-06 22:23:04.21419
0.1	0.00585142	camino2	2022-06-06 22:24:27.13344
0.01	0.00058514	camino2	2022-06-06 22:29:53.57935
0.01	0.181294	camino3	2022-06-15 00:00:44.66535
0.1	1.824853	camino3	2022-06-16 00:38:26.13485
0.01	0.00053801	camino2	2022-06-16 00:41:36.69692
0.01	0.00053420	camino2	2022-06-16 02:30:42.83832
0.01	0.183972	camino3-failed-sale	2022-06-16 02:44:37.92703
0	0	camino2-failed-purchase	2022-06-16 02:56:46.73159