Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

SEArch, una infraestructura de ejecución de software basado en servicios

Tesis de Licenciatura en Ciencias de la Computación

Pablo Montepagano
pablo@montepagano.com.ar

Director: Carlos Gustavo López Pombo

Buenos Aires, 2023

SEArch, una infraestructura de ejecución de software basado en servicios

En las últimas décadas, con la explosión de Internet, las APIs (*Application Programming Interfaces*) se impusieron como el mecanismo ubicuo para hacer disponible una pieza de software como un servicio que puede ser consumido por otro sistema de software. Sin embargo, en la mayoría de los casos las APIs no están documentadas adecuadamente y solo se describen el formato de los datos necesarios para invocarlas y requerimientos básicos de los protocolos de invocación. Para escribir software que utilice un servicio externo a través de una API, se requiere escritura manual de código que utilice dicha API, la cual es específica a un proveedor en particular.

En este trabajo implementamos una infraestructura experimental donde se cuenta con un repositorio global de contratos de provisión de servicios. Dichos contratos son descripciones formales de APIs. De esta manera, para desarrollar software que utilice un servicio externo, en lugar de escribir código específico para la API de un proveedor en particular, uno puede utilizar un contrato de requerimiento para describir qué espera del servicio externo. Dicho contrato se envía a un *Service Broker* que sabe determinar cuáles proveedores del repositorio global satisfacen el requerimiento. De esa manera, la infraestructura se ocupa de manera automática de hallar el mejor servicio disponible *en tiempo de ejecución* sin necesidad de depender exclusivamente de un proveedor en particular.

Palabras clave: SOA, contrato, CFSM, infraestructura

SEArch, a service-based infrastructure for software execution

In the last decades, with the explosion of the Internet, Application Programming Interfaces (APIs) have emerged as the ubiquitous mechanism for making a piece of software available as a service that can be consumed by another software system. However, in most cases, APIs are not adequately documented, and only the data format needed to invoke them and basic invocation protocol requirements are described. To write software that uses an external service through an API, manual code writing that utilizes that specific API, which is provider-specific, is required.

In this work, we implemented an experimental infrastructure where there is a global repository of services and their corresponding contracts. These contracts are formal descriptions of APIs. This way, to develop software that uses an external service, instead of writing specific code for a particular provider's API, one can write a requirement contract to describe what is expected from the external service. This contract is sent to a Service Broker that can determine which providers in the global repository satisfy the requirement. In this manner, the infrastructure automatically handles finding the best available service at runtime without the need to rely exclusively on a particular provider.

Keywords: SOA, contrato, CFSM, infraestructura

Agradecimientos

En primer lugar, agradezco a mi mamá y mi papá infinitamente. Siempre alentaron que estudie y me dedique a lo que me gusta. Sin su empuje y apoyo nunca hubiera llegado a terminar mis estudios. A mis hermanos, primos y tíos que son de fierro.

Especialmente agradezco a Charlie, mi director de tesis, que me aguantó durante más tiempo del que me gustaría admitir y siempre puso su tiempo y cabeza a disposición. Es una persona infinitamente generosa y con quien se disfruta enormemente trabajar y conversar.

A Nacho, Emilio y Juan Pablo, mis jurados, por leer y evaluar mi tesis a pesar de la agenda apretada. Les estoy enormemente agradecido.

A la Universidad pública, gratuita y de calidad, que me formó y me permitió conocer a tanta gente valiosa en mi vida. En particular, a Exactas UBA, donde estudié, trabajé y se convirtió en mi segundo hogar. A mis compañeros de La Mella, con quienes sigo aprendiendo: Axel, Florci, Eze, Lupi, Agus, Tavo, Cani, Santi, Maia, Caro, Pablito y muchos otros. A mis compañeros de trabajo en Exactas, tanto de la Biblioteca como de la UTI, que siempre me empujaron para que estudie y me reciba: Gaby, Diego, Martín DM, Emilio, Martín W, Ana, Vero, Kari y tantos otros. Al grupo del LAFHIS, que en el último tramo de la carrera me dio un espacio de pertenencia que me ayudó para dar el tirón final (Diego, JP, Agus, Iván, Sebas, ¡gracias!).

A Flor, que me banca en todas y que sacrificó un montón de su tiempo para que yo pueda estudiar y terminar esta tesis. ¡Gracias!

Índice

1. Introducción 8
1.1. Fundamentos formales de SEArch 12
1.2. Estructura de la tesis 12
2. Definiciones preliminares 13
2.1. Semántica operacional 13
2.2. Contratos comportamentales 18
3. Diseño 26
3.1. Interfaz que el Middleware expone a Service Clients/Providers 27
3.2. Registro de Service Provider ante el Service Repository 29
3.3. Ejemplo de implementación: Ping-Pong 31
3.4. Establecimiento de un canal 35
3.5. Abstracción sobre los contratos 39
3.6. Verificación de compatibilidad entre contratos 40
3.7. Limitaciones 41
4. Implementación 44
4.1. Elección del protocolo de comunicación y del lenguaje de programación 44
4.2. Organización del repositorio de código fuente 45
4.3. Cálculo y almacenamiento de resultados de compatibilidad 46
4.4. Conexión entre middlewares 47
4.5. Tests de integración 47
5. Caso de estudio 48
6. Conclusiones y trabajo futuro 59
6.1. Monitoreo de corrección de los protocolos 59
6.2. Ranking de proveedores 60
6.3. Logging distribuído 61
6.4. Tipos complejos de datos para los mensajes 61
Bibliografía 63

1. Introducción

Los nuevos paradigmas de computación Cloud/Fog, como así también la Internet de las cosas (IoT por su sigla en inglés), han impulsado enormemente lo que hoy se denomina economía de las APIs[1][2]. La idea que subyace en la economía de las APIs es la posibilidad de construir nuevos servicios utilizando APIs provistas por terceras partes y, a su vez, hacer disponibles estos nuevos servicios publicando sus propias APIs. Al mismo tiempo, la economía de las APIs está revolucionando la forma en la que el software se desarrolla, implementa y usa. Las aplicaciones y los dipositivos están cada vez más interconectados para poder asistir a las personas, por lo cual requerimientos tales como *self-adaptiveness* y reconfiguración dinámica transparente son requerimientos esenciales de las aplicaciones actuales. En este nuevo modelo de cómputo basado en APIs, se vuelven cada vez más presentes los microservicios, donde se desplaza el desarrollo tradicional de software monolítico para dar lugar al uso de interfaces que permiten acceder a servicios y recursos distribuidos. De esta manera, las APIs posibilitan a los proveedores de software controlar cuáles son los aspectos de sus plataformas que desean hacer públicos y cuáles privados, posibilitando la interoperabilidad de componentes a más alto nivel.

En la industria actual, las APIs representan el último escalón en la interoperabilidad y ponen de relieve la necesidad de descripciones precisas como forma preponderante de documentación. Sin embargo, muchas APIs no están documentadas adecuadamente[3]. En la mayoría de los casos, las APIs solo describen el formato de los datos necesarios para invocarlas y requerimientos básicos de los protocolos de invocación, pero los aspectos más importantes de su comportamiento son documentados informalmente, como es el caso de lenguajes adoptados por la industria como WS-BPEL[4], desarrollado y promovido por OASIS[5]. La documentación informal dificulta validar el software que se obtiene componiendo servicios a través de sus APIs, establecer sus propiedades y mantener aplicaciones[6]. En consecuencia, describir formalmente el comportamiento de las APIs de forma que éste provea garantías a sus clientes constituye un desafío técnico clave en este contexto.

Este proyecto de tesis parte de identificar tres elementos centrales concernientes a la descripción de APIs: 1) las condiciones de interoperabilidad entre servicios (que expresan la forma en que las componentes se deben comunicar intercambiando mensajes); 2) los requerimientos funcionales, expresados como condiciones sobre los valores intercambiados durante la comunicación (expresando las condiciones que permiten definir a un servicio a partir de una caracterización formal de qué es lo que este hace [9]); y 3) los requerimientos no-funcionales, expresados como condiciones sobre variables cuantificables que reflejen métricas de calidad de servicio – QoS.

El presente proyecto asume una perspectiva en la que el ideal detrás de la ejecución de aplicaciones basadas en APIs se lleva a cabo sobre una infraestructura de comunicación y cómputo

[1] `http://ibm.com/apieconomy`. Reach new customers with the API economy. Accedido el 3 de abril de 2017.

[2] `http://nordicapis.com/ebook-release-api-economy-disruption-business-apis/`. The API Economy – Disruption and the Business of APIs. Nodic APIs (nordicapis.com). Publicado el 6 de mayo de 2016.

[3] `http://www.computerworld.com/article/2593623/app-development/application-programming-interface.html`. Application Programming Interface. Computer World. Publicado el 10 de enero de 2000.

[4] `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`

[5] `https://www.oasis-open.org`

[6] `https://alm.parasoft.com/api-testing-myths`. Testing in the API Economy Top 5 Myths.

ubicua y preexistente y en la que un middleware es capaz de solicitar a un service broker la búsqueda de un servicio al que, sujeto a una negociación de nivel servicios (SLA por su sigla en inglés, Service Level Agreements) [8], pueda vincularse en forma completamente automática y transparente, para que colectivamente sea posible alcanzar cierto objetivo de negocios.

La estructura de los sistemas de software es cada vez más dinámica y compleja a medida que las aplicaciones necesitan responder y adaptarse a los cambios en el entorno en el que operan. El paradigma de *Service-Oriented Computing* (SOC) [12] surge alrededor del año 2000 en un contexto en el que Internet se masifica. SOC propone una nueva generación de software que ejecuta sobre una infraestructura de cómputo global e interconectada, en la cual las aplicaciones utilizan servicios externos de manera dinámica. La premisa se basa en desarrollar aplicaciones distribuídas que utilicen servicios a través de la red. Al componer distintos servicios, las aplicaciones pueden proveer funcionalidades más complejas y con mayor valor agregado. No hay una única definición de lo que es SOC, sino que más bien se trata de una serie de principios de diseño de software a gran escala. La *arquitectura orientada a servicios* (SOA, por sus siglas en inglés) tiene tres componentes principales: un proveedor, un consumidor y un registro.

Además, en SOA se propone que el desarrollo de estos sistemas no implique una tarea manual de integración entre distintos sistemas a ser implementada por equipos de desarrollo de software. En cambio, en SOA el descubrimiento y uso de servicios externos se realiza a través de un *middleware*. Los proveedores publican sus servicios en el registro, donde los consumidores los pueden descubrir para luego utilizarlos.

El objetivo central del presente trabajo es la implementación de una infraestructura que posibilite la ejecución distribuída de servicios en la que el proceso de *discovery*, y su posterior *binding* se realicen en tiempo de ejecución gracias a la intervención de las componentes activas de la misma. A esta infraestructura la denominamos «SEArch» (*Service Execution Architecture*) y propone los siguientes principios:

- pensar los servicios como procesos que ejecutan concurrentemente y se comunican asincrónicamente a partir del envío y recepción de mensajes en un orden preestablecido, análogo a un protocolo de comunicación,
- dotar a la infraestructura de ejecución de la capacidad de realizar *discovery* automático de servicios a partir de contratos de requerimientos y provisión, y
- posibilitar el *binding* transparente del servicio seleccionado.

Para poder llevar esto a cabo, contamos con las siguientes componentes activas:

- ***Service Provider***: se trata del servicio que se disponibiliza para el consumo por parte de otros servicios o aplicaciones. Cuenta con un contrato de provisión que describe su comportamiento.
- ***Service Client***: es el artefacto de software que consume o utiliza el servicio del *Service Provider*. También cuenta con un contrato que describe el comportamiento de la interacción con el o los servicios que requiere utilizar (contrato de requerimiento).
- ***Service Repository***: servicio centralizado donde los *Service Providers* registran sus contratos de provisión junto con la información necesaria para ser «descubiertos» por los *Service Clients* a través del *Service Broker*.
- ***Service Broker***: se trata de un servicio centralizado al cual acceden los *Service Clients* cada vez que necesitan utilizar algún servicio del ecosistema. El *broker*, dado un contrato de requerimiento, puede determinar cuáles de todos los servicios disponibles en el *Service*

Repository son compatibles con el *Service Client* que lo solicita, y además los puede ordenar según otros criterios no funcionales como precio, latencia, disponibilidad, reputación, etc.

- ***Middleware***: tanto *Service Clients* como *Service Providers* utilizan un *middleware* para gestionar las conexiones con el *broker* y con los otros servicios del ecosistema. De esa manera, los autores de los distintos servicios pueden abstraerse de los detalles relativos a la resolución de dependencias y al *binding*.
- ***Certification Authority***: se trata de una componente opcional del ecosistema que no implementaremos en el marco de esta tesis. Su rol es el de validar que la implementación de un servicio satisface el contrato de provisión que éste declara antes de registrar el mismo en el repositorio. Como veremos más adelante, en lugar de utilizar una *Certification Authority*, en el marco de este trabajo permitiremos la registración de servicios sin certificación y en cambio llevaremos adelante un mecanismo de monitoreo de validez en tiempo de ejecución que se realiza en los *middlewares*.

Para que un *Service Client* pueda «descubrir» un servicio proveedor, antes deben suceder dos procesos que se realizan en forma *offline*, que pueden observarse en la Figura 1:

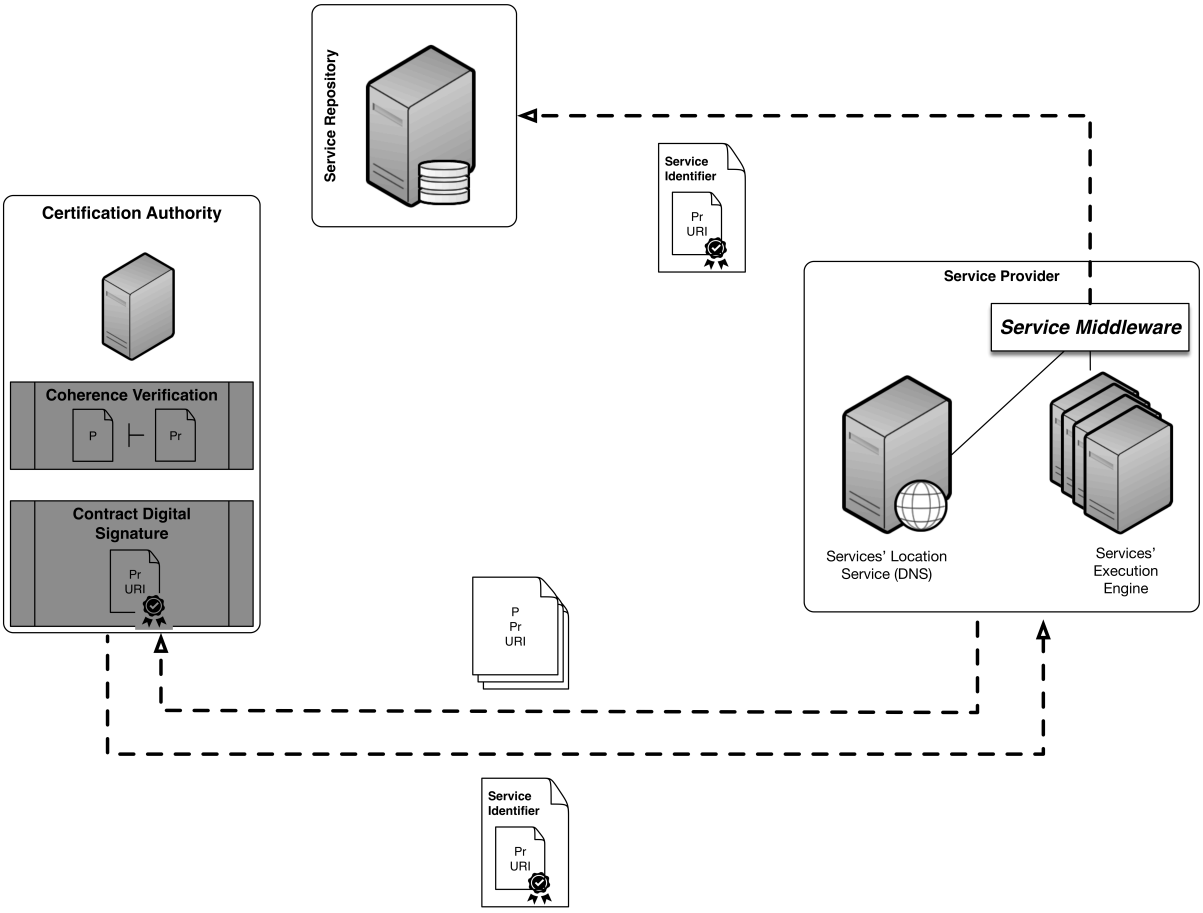


Figura 1: Visión esquemática del proceso de registración en SEArch.

- cuando un *Service Provider* desea proveer un servicio, es necesario que una *Certification Authority* analice si la implementación P propuesta satisface los contratos Pr que serán publicados en el repositorio; en caso de satisfacerlos, la autoridad asigna al servicio un *URI* (*Uniform Resource Identifier*) y firma el contrato, y
- el *Service Provider* puede publicar en un *Service Repository* los contratos de un servicio, acompañados del *URI* correspondiente, firmado por una *Certification Authority*.

En segundo lugar, como lo muestra la Figura 2, la ejecución en este modelo ocurre de la siguiente forma:

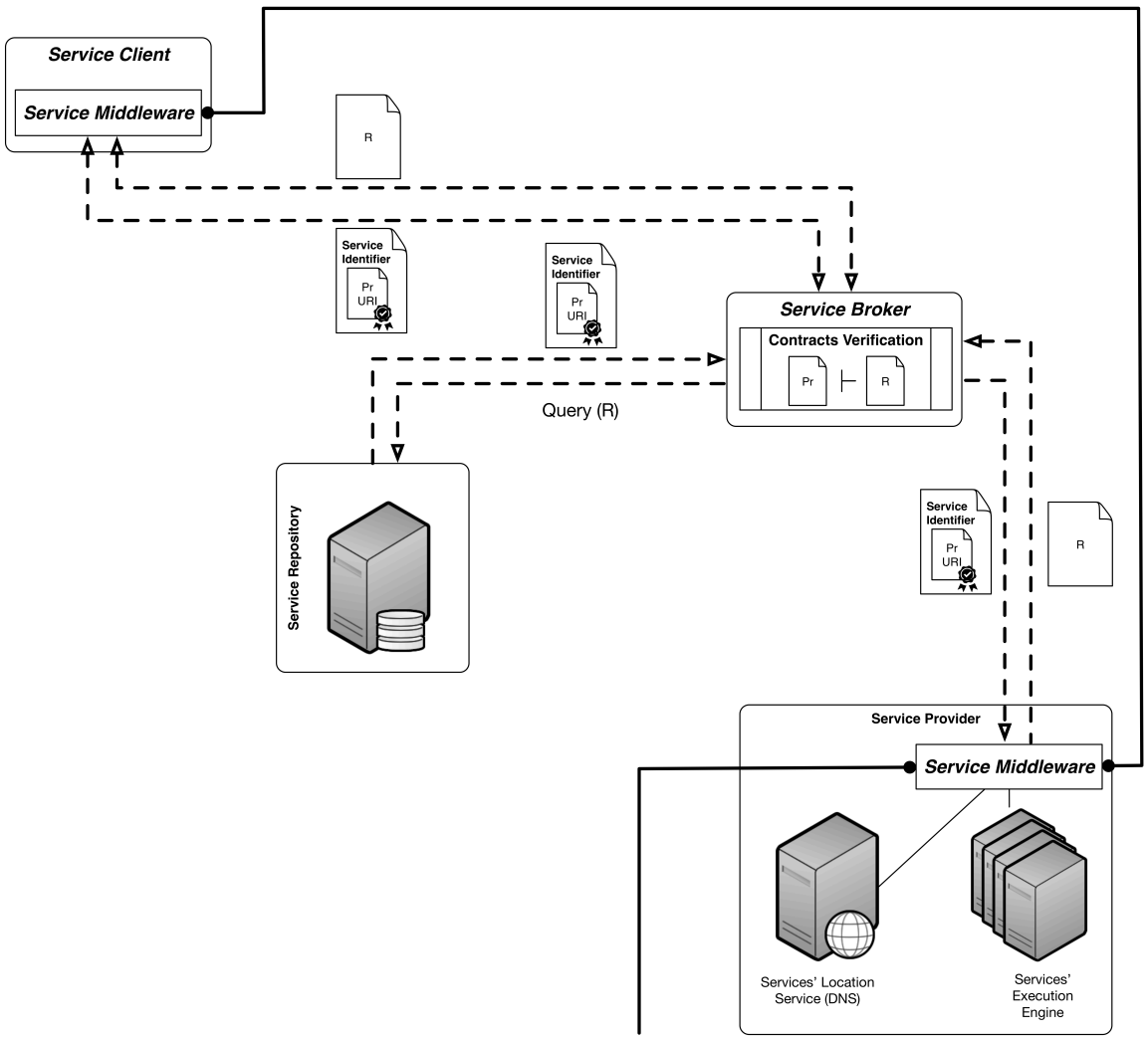


Figura 2: Visión esquemática del proceso de ejecución en SEArch.

- un proceso, que diremos que actúa como *Service Client*, realiza un envío de un mensaje (respectivamente espera un mensaje) sobre un canal de comunicación que no se encuentra conectado,
- el *Middleware* al que este proceso se encuentra conectado detecta este envío (respectivamente espera), lo captura y se envía una solicitud a un *Service Broker* adjuntando los contratos de requerimientos R asociados a ese canal de comunicación,
- el *Service Broker* que recibió la solicitud, consulta los *Service Repository* a los que tiene acceso a fin de evaluar cuáles de los servicios registrados en ellos poseen contratos de provisión Pr que satisfagan los contratos de requerimiento R,
- una vez encontrado el candidato óptimo (bajo las políticas implementadas por el *Service Broker*), este retorna al *Middleware* que disparó la búsqueda el *URI* del servicio seleccionado,
- el *Middleware* que disparó la búsqueda utiliza el *URI* para establecer una conexión con el *Middleware* a través del cual se accede al servicio identificado por dicho *URI*,
- la ejecución continúa hasta que algún proceso produce una situación como la expresada en 1) (vale mencionar que cualquier proceso participante podrá actuar como *Service Client* disparando dicha situación).

1.1. Fundamentos formales de SEArch

Existen dos elementos centrales sobre los que se basa este trabajo. Por un lado, existe un modelo formal de cómputo que otorga una semántica precisa a los artefactos de software ejecutados en este modelo. Este modelo se denomina *Asynchronous Relational Nets* (ARN). En el mismo, se modelan a los *procesos* (unidades de cómputo) y a los *canales de comunicación* como hiperejes de un hipergrafo. Los nodos, a su vez, se denominan *puertos* y son los puntos en los que los procesos intercambian mensajes con los canales de comunicación. En las ARNs no hay procesos adyacentes entre sí, como tampoco hay canales de comunicación adyacentes entre sí. De esta manera se modela la comunicación entre procesos a través de canales de comunicación. Los puertos que solamente son adyacentes a hiperejes de proceso se denominan *provides-points*, mientras que aquellos que solo son adyacentes a hiperejes de comunicación se denominan *requires-points*. El *provides-point* de un servicio es la interfaz a través de la cual el mismo exporta su funcionalidad, mientras que un *requires-point* describe la interfaz a través de la cual un proceso espera comunicarse con un servicio externo (su contrato de requerimiento). De esta manera, la composición de ARNs se realiza «fusionando» *requires-points* y *provides-points* que sean compatibles entre sí.

La semántica de las ARNs se define en función de secuencias infinitas de conjuntos de acciones. En su primera definición [7], la descripción del comportamiento de las mismas se dio utilizando fórmulas de *Linear Temporal Logic*. En una redefinición posterior de ARNs ([13]), en la que se basa este trabajo, se utilizan, en cambio, autómatas de Muller para describir tanto a los hiperejes de proceso como a los de comunicación. En el primer caso, el autómata formaliza el cómputo llevado adelante por dicho servicio. En el segundo, los autómatas representan un orquestador que coordina a los participantes de la comunicación (los canales de comunicación pueden tener más de dos participantes ya que se trata de un hipergrafo). El comportamiento global del sistema se obtiene componiendo los distintos autómatas (de ambos tipos de hipereje).

El segundo elemento central sobre el que se basa este trabajo es la adopción de *Communicating Finite State Machines* [3] para exponer la interfaz de los servicios que se encuentran disponibles en el *Service Repository*. Esto permite al *Service Broker* determinar automáticamente la compatibilidad entre un *requires-point* y un *provides-point*, y de esa manera garantizar la interoperabilidad entre *Service Client* y *Service Provider* [26].

1.2. Estructura de la tesis

En el Capítulo 2 damos las definiciones formales que dan sustento teórico a este trabajo. En el Capítulo 3 abordamos el diseño general de la solución implementada, con foco principal en la interfaz que se expone a los desarrolladores que utilicen la infraestructura desarrollada. En el Capítulo 4 mostramos detalles sobre la implementación. En el Capítulo 5 se muestra un ejemplo detallado que ilustra cómo utilizar la infraestructura para construir servicios sobre la misma. Finalmente, en el Capítulo 6 presentamos algunas conclusiones del presente trabajo y detallamos algunas líneas de trabajo futuro relacionado.

2. Definiciones preliminares

A continuación presentaremos los conceptos formales sobre los que se apoya la propuesta de infraestructura de cómputo distribuido SEArch, presentada informalmente en la Capítulo 1.

2.1. Semántica operacional

Fiadeiro y Lopes presentan en [7] las *Asynchronous Relational Networks* (ARNs) con la intención de formalizar elementos de una teoría de interfaces[1] para software orientado a servicios. En particular, al hablar de interfaces de servicios, Fiadeiro y Lopes se refieren a una descripción de las propiedades que un servicio provee (de manera tal que pueda ser «descubierto») así como también de las que un servicio requiere de otros servicios externos (de manera tal que el middleware pueda seleccionar un proveedor apropiado).

En [7] (2011) los *procesos* están interconectados por *canales*. A diferencia de otros modelos para describir coreografías u orquestación de servicios donde la comunicación es sincrónica, aquí la comunicación es asincrónica y se puede modelar utilizando *communicating finite-state machines* (CFSMs) [3]. Los mensajes de un proceso se organizan en conjuntos denominados *puertos*. Cada proceso contiene una colección (finita) de puertos mutuamente excluyentes. Es decir, cada mensaje que puede ser intercambiado por un proceso pertenece a uno solo de sus puertos. Cada mensaje asociado a un puerto tiene una *polaridad*: $^-$ si es un mensaje saliente (es decir, que se publica en ese puerto) y $^+$ si es entrante (se recibe en ese puerto). Los procesos, además, contienen un conjunto de fórmulas de *Linear Temporal Logic* (LTL) que describen el comportamiento del proceso con respecto al envío y recepción de mensajes a través de sus puertos. Los *canales* conectan procesos a través de los puertos de éstos, apareando mensajes con polaridades opuestas en cada puerto.

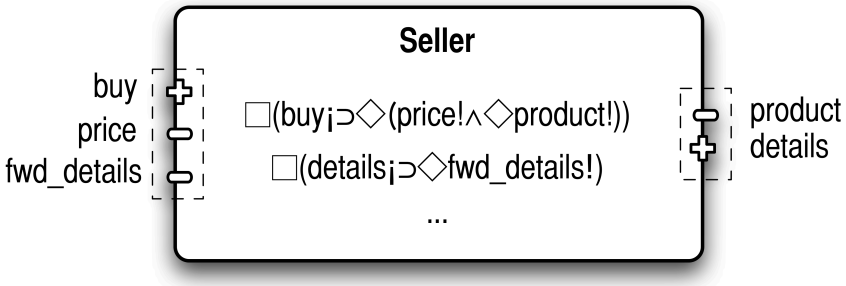


Figura 3: Ejemplo de un proceso con dos puertos, tomado de [7].

En [13] (2015) Fiadeiro y Tutu redefinen a la ARN como un hipergrafo[7] donde los nodos representan puertos y los hiperejes representan procesos o canales de comunicación. Los puertos que solo pertenecen a hiperejes de proceso se llaman *requires-points*, mientras que aquellos que solo pertenecen a ejes de comunicación se llaman *provides-points*. Esto tiene que ver con lo que éstos representan: un *provides-point* es la interfaz a través de la cual un servicio exporta

[7]Un hipergrafo es un grafo en el cual los ejes, en vez de ser «líneas» que conectan dos nodos, son conjuntos que conectan una cantidad finita de nodos (no nula). Gráficamente, en los hipergrafos los nodos suelen representarse como puntos (al igual que en los grafos), pero los ejes se representan como un contorno que encierra a los nodos que interconecta. La redefinición de ARNs utilizando hipergrafos permite definir canales donde intervienen más de dos procesos.

su funcionalidad. Mientras que un *requires-point* es la interfaz a través de la cual un proceso espera que otro servicio le provea cierta funcionalidad. La composición de ARNs se obtiene al «fusionar» *provides-points* con *requires-points* siempre y cuando se verifique la interoperabilidad de los contratos asociados a cada uno.

Hay distintos formalismos e interpretaciones para dar semántica a las ARNs. En [7], la descripción del comportamiento se da utilizando LTL. En cambio, en [13], se utilizan autómatas de Muller donde las cadenas de entrada son secuencias de conjuntos de acciones. Tanto los ejes que representan procesos como los ejes que representan canales de comunicación están etiquetados con autómatas de Muller. En el caso de los procesos, los autómatas formalizan el cómputo realizado por el proceso, mientras que en los canales, los autómatas representan la orquestación de los participantes que se comunican a través del canal.

Las *acciones* de enviar (publicar) o recibir un mensaje m se denotan $m!$ y $m¡$, respectivamente.

Definición 1 (Autómata de Muller). La categoría $\mathbb{MA}$ de autómatas de Muller sobre $\mathbb{A}$ se define como los pares $\langle A, \Lambda \rangle$ donde A es un conjunto de acciones y Λ un autómata de Muller donde $\Lambda = \langle Q, P(\mathbb{A}), \Delta, I, F \rangle$ sobre el alfabeto $P(\mathbb{A})$[8] donde:

- Q es el conjunto de estados de Λ
- $\Delta \subseteq Q \times P(\mathbb{A}) \times Q$ es la relación de transición de Λ
- $I \subseteq Q$ es el conjunto de estados iniciales de Λ, y
- $F \subseteq 2^Q$ es el conjunto de conjuntos finales de Λ.

El conjunto F define el *criterio de aceptación* del autómata. Los autómatas de Muller toman cadenas infinitas como entrada. Una cadena c que, al ser consumida por el autómata Λ, atraviesa infinitas veces al conjunto de estados F_c es aceptada si y solo si $F_c \in F$.

En [25] se propone una semántica operacional para artefactos de software orientado a servicios utilizando *Asynchronous Relational Nets*. Como mencionamos, en ellas los hiperejes se dividen en dos particiones: los de comunicación y los de cómputo. Los hiperejes de cómputo representan procesos, mientras que los de comunicación representan canales. Los nodos del hipergrafo se encuentran etiquetados con *puertos*, es decir, conjuntos de mensajes con su polaridad (M^+ para los entrantes, M^- para los salientes). Los hiperejes, a su vez, se encuentran etiquetados con autómatas de Muller. Por lo tanto, los procesos y los canales de comunicación poseen una descripción de su comportamiento dada por el autómata correspondiente a su eje, e interactúan entre sí a través de los mensajes definidos en los puertos (nodos) en los que se conectan.

Definición 2 (Proceso). Un proceso $\langle \gamma, \Lambda \rangle$ está compuesto por un conjunto γ de puertos disjuntos par a par (no hay mensajes repetidos en distintos puertos) y un autómata de Muller Λ definido sobre el conjunto de acciones $A_\gamma = \bigcup_{M \in \gamma} A_M$, donde $A_M = \{m! \mid m \in M^-\} \cup \{m¡ \mid m \in M^+\}$.

Por ejemplo, en la Figura 4 se ilustra el proceso $\langle \gamma_{\mathrm{TA}}, \Lambda_{\mathrm{TA}} \rangle$, donde el conjunto de puertos es $\gamma_{\mathrm{TA}} = \{\mathrm{TA}_0, \mathrm{TA}_1, \mathrm{TA}_2\}$ (Figura 4 (a)), y Λ_{TA} es el autómata de la Figura 4 (b). El propósito del *Travel Agent* es proveer reservas de hoteles y/o vuelos en la moneda local del comprador. Para ello, requiere interactuar con dos servicios externos: por un lado con un proveedor de vuelos y hoteles, y por el otro, con un proveedor de conversión de monedas.

[8] El alfabeto de los autómatas de Muller sobre $\mathbb{A}$ se define sobre el alfabeto de $P(\mathbb{A})$ (de tamaño $2^{|\mathbb{A}|}$) porque en cada transición puede haber no solo una acción sino un conjunto de acciones. Cuando hay más de una acción en una transición entre dos estados, interpretamos que esas acciones suceden al mismo tiempo (o con tan poca diferencia de tiempo que resulta indistinguible).

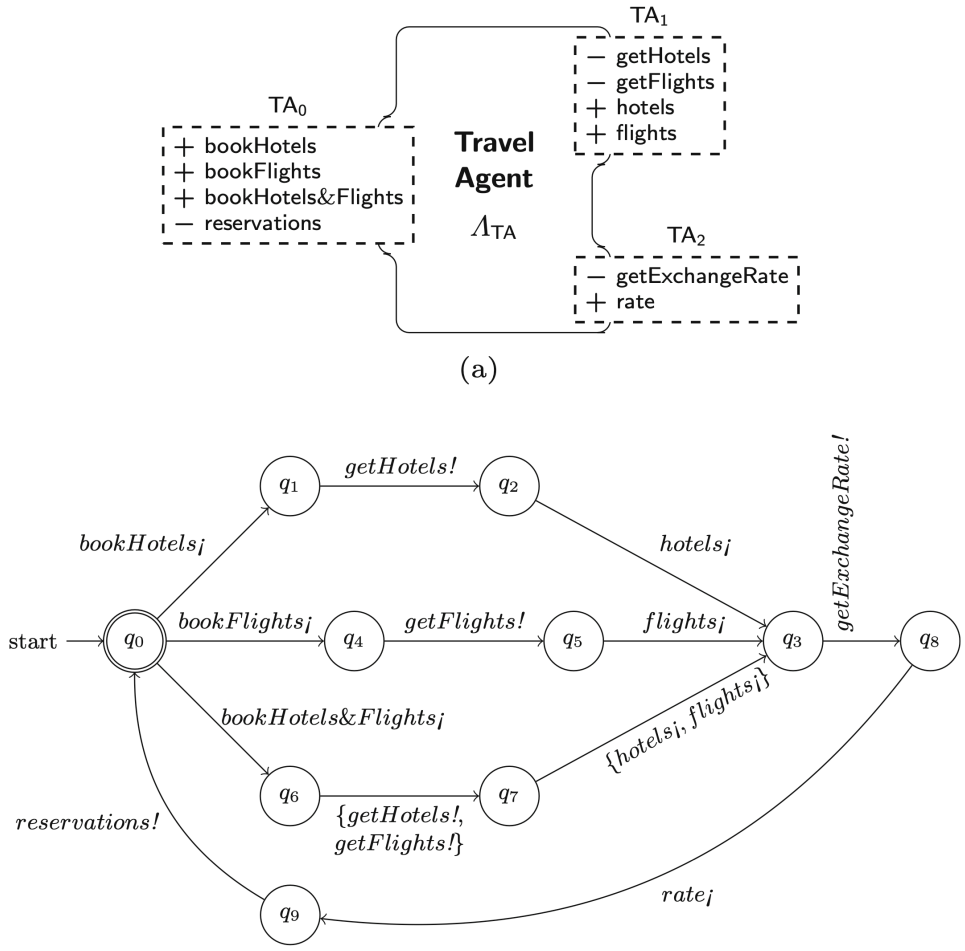


Figura 4: El proceso TravelAgent (a) junto a su autómata Λ_{TA} (b), tomado de [25].

Definición 3 (Conexión (en ARN)). Sea γ un conjunto de puertos disjuntos par a par. Una conexión $\langle M, \mu, \Lambda \rangle$ entre los puertos de γ está compuesta por un conjunto M de mensajes, una función inyectiva parcial $\mu_i : M \rightharpoonup M_i$ para cada puerto $M_i \in \gamma$, y un autómata de Muller Λ sobre $A_M = \{m! \mid m \in M\} \cup \{m¡ \mid m \in M\}$ tal que

$$(a)\ \ M = \bigcup_{M_i \in \gamma} \mathrm{dom}(\mu_i) \quad \text{y } (b)\ \ \mu_i^{-1}(M_i^{\mp}) \subseteq \bigcup_{M_j \in \gamma \setminus M_i} \mu_j^{-1}\left(M_j^{\pm}\right)$$

En la Fig. 5 (a) se ilustra la conexión C_0. El conjunto de mensajes es la unión de los mensajes de los puertos $\{\mathrm{TA}_1, H_0, F_0\}$. La función μ se define trivialmente asignando cada mensaje a su puerto correspondiente. En la Fig. 5 se ilustra el autómata de Muller Λ_{C_0} que describe el comportamiento de la conexión.

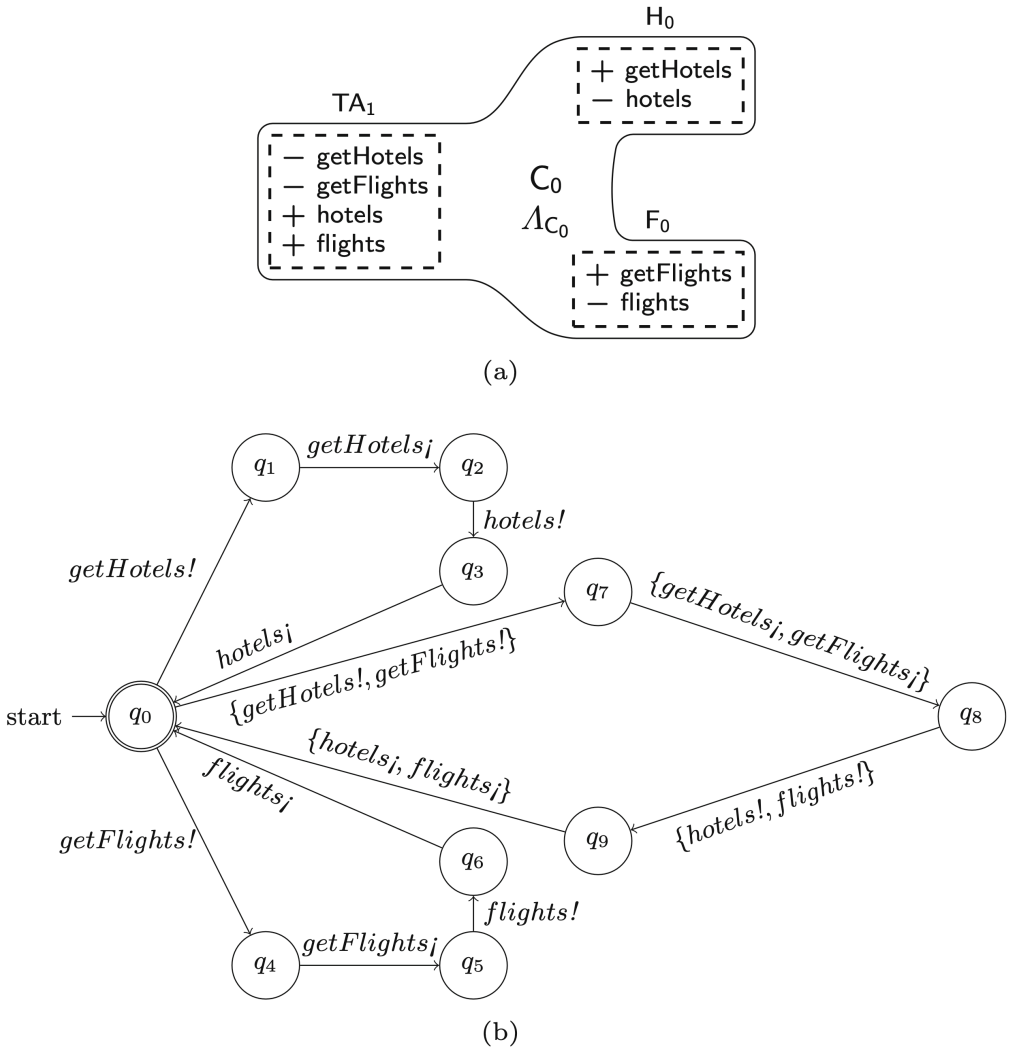


Figura 5: La conexión C_0 (a) junto a su autómata Λ_{C_0} (b), tomado de [25].

Definición 4 (Asyncronous Relational Net). Una ARN $\alpha = \langle X, P, C, \gamma, M, \mu, \Lambda \rangle$ está compuesta de:

- un hipergrafo $\langle X, E \rangle$, donde X es un conjunto finito de nodos y $E = P \cup C$ es un conjunto de hiperejes (conjuntos no vacíos de X) que está particionado en *hiperejes de cómputo* $p \in P$ e *hiperejes de comunicación* $c \in C$ tales que ningún hipereje es adyacente a otro de la misma partición; y
- tres funciones de etiquetado que asignan:
  - un puerto M_x a cada nodo $x \in X$
  - un proceso $\langle \gamma_p, \Lambda_p \rangle$ a cada hipereje $p \in P$, y
  - una conexión $\langle M_c, \mu_c, \Lambda_c \rangle$ a cada hipereje $c \in C$.

En otras palabras, una ARN es un hipergrafo en el cual los hiperejes que son *procesos* formalizan el cómputo con autómatas de Muller que se comunican con otros procesos a través de puertos (los nodos del hipergrafo) utilizando un lenguaje (fijo) de acciones. La comunicación entre procesos no es directa, sino que está mediada por los hiperejes de comunicación, que también tienen su comportamiento descripto con autómatas de Muller sobre el mismo lenguaje.

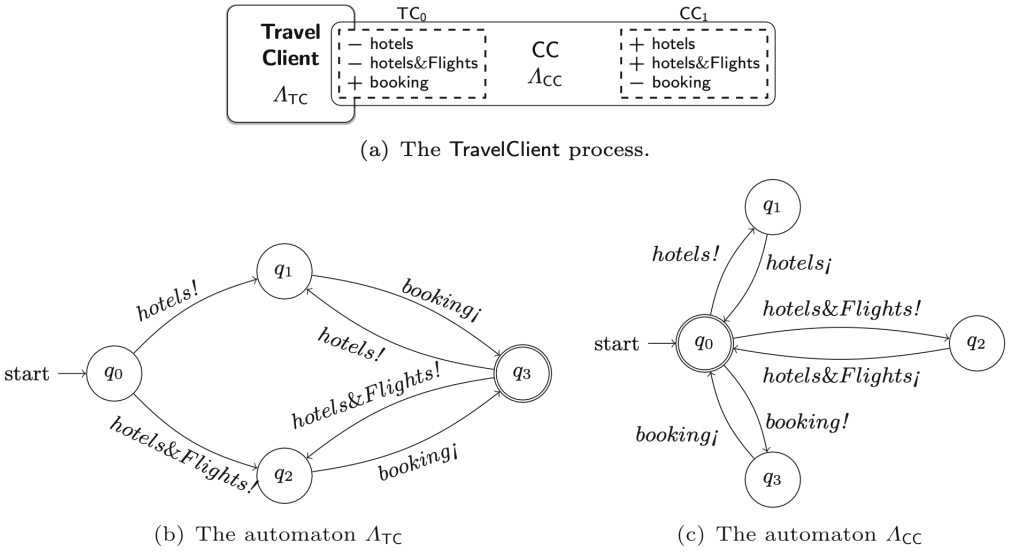
Para utilizar estos conceptos para hablar de servicios, repositorios y actividades, nos interesa poner el foco en dos tipos de nodos (puertos): aquellos que no son adyacentes al mismo tiempo a hiperejes de comunicación e hiperejes de cómputo.

Definición 5 (Puertos de requerimiento y provisión). Un *puerto de requerimiento* de una ARN es un nodo que es adyacente solamente a un hipereje de comunicación. Un *puerto de provisión* de una ARN es un nodo que solamente es adyacente a un hipereje de cómputo.

Definición 6 (Repositorio de servicios). Un repositorio de servicios es un conjunto $\mathbb{R}$ de módulos de servicio, es decir, triplas de $\langle P, \alpha, R\rangle$, también denotadas $P \underset{\alpha}{\longleftarrow} R$, donde α es una ARN, P es un *puerto de provisión* de α, y R es un conjunto finito de *puertos de requerimiento* de α.

Definición 7 (Actividad). Una *actividad* es una tupla $\langle \alpha, R\rangle$ tal que α es una ARN y R es un conjunto finito de *puertos de requerimiento* de α. Una *actividad* solo tiene puertos de requerimiento, no tiene ningún puerto de provisión.

Las definiciones anteriores formalizan la idea un artefacto de software orientado a servicios como una *actividad* cuyos requerimientos computacionales son modelados por sus conexiones «abiertas» (como puntas «desconectadas»); y que no proveen ningún servicio a otras unidades de cómputo, lo cual es modelado con la ausencia de *puertos de provisión*. La Figura 6 ilustra a la *actividad* TravelClient, que tiene un único *puerto de requerimiento* a través del cual solicita reservas de solamente hoteles o reservas de hoteles y vuelos en conjunto. Los *puertos de requerimiento* se fusionan con los *puertos de provisión* de algún servicio compatible del *repositorio de servicios* para satisfacer dichos requerimientos y conformar una ARN más grande que compone a ambas.



(a) The TravelClient process.

(b) The automaton Λ_{TC}

(c) The automaton Λ_{CC}

Figura 6: La actividad *TravelClient, tomado de [25].*

Para transformar un proceso en un *servicio* que pueda ponerse a disposición de otros procesos y poder ser descubierto en el *repositorio de servicios*, es necesario declarar el proceso a través del cual se provee el servicio (el hipereje de cómputo que tiene el *puerto de provisión*, en nuestro ejemplo: Λ_{TA}, en la Figura 4 b). Asimismo, si los tuviera, también los canales de comunicación necesarios para conectarse a otros (terceros) servicios. En el caso del TravelAgent (Figura 7), éste requiere tres servicios para ejecutar, que se acceden a través de dos canales de comunicación distintos. En uno de ellos, el proceso interactúa con proveedores de reservas de alojamiento y

proveedores de reservas de vuelos. En el otro canal, solo se comunica con un servicio de tasas de cambio entre monedas para poder ofrecer al usuario las reservas en su moneda local. En cierto sentido, el servicio TravelAgent provee la capacidad de combinar estos otros tres servicios para ofrecer una experiencia de usuario más rica.

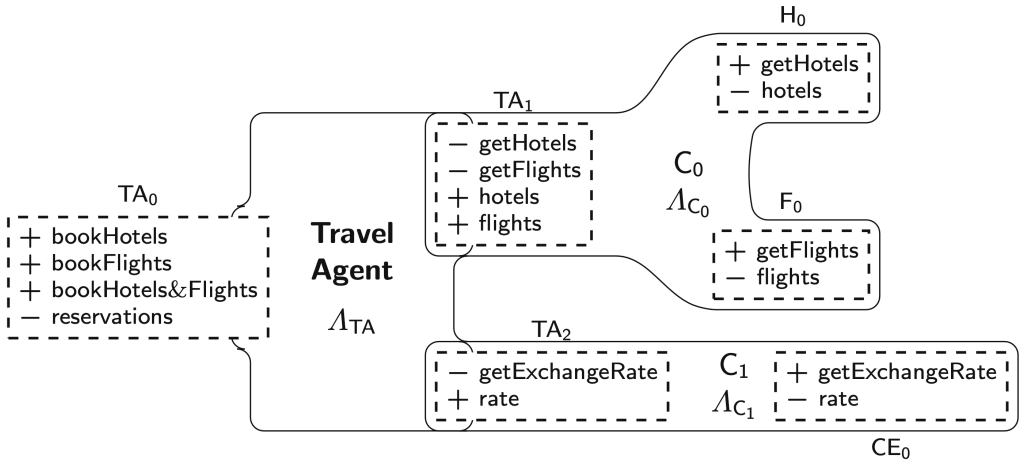


Figura 7: El servicio *TravelAgent, tomado de [25].*

Este encuadre permite en [25] dar una semántica operacional a los procesos y a los canales de comunicación de manera integrada. También permite representar la reconfiguración dinámica de ARNs (cuando se obtiene un servicio del repositorio y se fusiona un *puerto de requerimiento* de una *actividad* con el *puerto de provisión* de un servicio). Esto permite también distinguir entre el no-determinismo causado por un comportamiento no-determinístico de un componente de la ARN, y aquel no-determinismo que surge del hecho de que el «descubrimiento» de servicios es no-determinístico, ya que el conjunto de servicios del repositorio puede ir modificándose a lo largo del tiempo, y el criterio para el *descubrimiento* puede también variar[9].

2.2. Contratos comportamentales

Para poder componer ARNs, en [25] y [27] se utilizan descripciones en *linear temporal logic* (LTL)[23] sobre los *puertos de requerimiento y de provisión*, sobre las que se puede utilizar cualquier algoritmo de decisión sobre LTL (por ejemplo, *model-checking* [2]). Esto presenta dos desventajas:

1. El uso de LTL para verificar que un *puerto de provisión* satisface lo que necesita un *puerto de requerimiento* establece una relación asimétrica ya que formaliza la noción de trazas. Esta asimetría puede conducir a situaciones indeseables. Por ejemplo, supongamos que el proceso TravelAgent utilizado anteriormente también requiere utilizar un tercer servicio para procesar pagos electrónicos. Supongamos que el *puerto de requerimiento* está etiquetado con un contrato que establece que el resultado de la ejecución es que se *acepta* o se *rechaza* el pago, y se formaliza con la siguiente fórmula LTL:

   $$\Diamond((-\text{aceptar} \vee -\text{rechazar}) \wedge \neg(-\text{aceptar} \wedge -\text{rechazar}))$$

   Asimismo, uno podría especificar un servicio con un *puerto de provisión* compatible que siempre rechaza los pagos y nunca los acepta utilizando la siguiente fórmula LTL para su contrato:

[9]Por ejemplo, si el repositorio contiene dos servicios compatibles con el requerimiento A, el *broker* puede en distintos momentos resolver uno u otro según criterios no funcionales como precio o latencia.

$$\diamondsuit(-\text{rechazar} \wedge \neg -\text{aceptar})$$

Es sencillo demostrar que la segunda fórmula satisface a la primera. Esto habilitaría la *fusión* entre la actividad que requiere un sistema que acepte o rechace pagos con un servicio que siempre los rechaza, lo cual no es deseable.

2. La inclusión de los autómatas que orquestan la comunicación en los canales (los autómatas de los hiperejes de comunicación) junto con los autómatas de procesos aumenta considerablemente el tamaño del autómata resultante de la composición, lo cual hace que el análisis de dicho autómata sea más costoso.

En [26], Vissani, López Pombo y Tuosto proponen las *Communicating Relational Networks* (CRNs) como una variante de las ARNs en las que el *puerto de provisión* está etiquetado con una *Communicating Finite State Machine* (CFSM) [3] que describe el protocolo de comunicación del servicio exportado, mientras que los hiperejes de comunicación, en cambio, se encuentran etiquetados con *Global Graphs* [5] que especifican el comportamiento del canal.

Las *Communicating Finite State Machines* fueron presentadas por primera vez en [3] para modelar y estudiar protocolos de comunicación utilizando máquinas de estados finitas para representar procesos y colas de mensajes para representar canales de comunicación entre procesos. Las colas de mensajes tienen una capacidad ilimitada, lo que permite representar protocolos que admiten una cantidad arbitraria de mensajes en tránsito. Cada par de procesos dentro de un mismo sistema o protocolo está interconectado por un canal en cada sentido de la comunicación, sin errores y que preserva el orden de los mensajes (FIFO)[10].

Definición 8 (Communicating Finite State Machine). Dado un conjunto finito de mensajes M y un conjunto finito de participantes P, una CFSM es una tupla $\langle Q, C, q_0, M, \delta \rangle$, donde:

- Q es un conjunto finito de estados;
- $C = \{pq \in P^2 \mid p \neq q\}$ es un conjunto de canales de comunicación
- $q_0 \in Q$ es un estado inicial;
- $\delta \subseteq Q \times (C \times \{!,?\} \times M) \times Q$ es un conjunto finito de transiciones.

Un *sistema de CFSMs* es una función S que asigna una CFSM $S(p)$ a cada $p \in P$ (es decir, cada participante es una CFSM). Decimos que $q \in S(p)$ cuando q es un estado de la máquina $S(p)$, del mismo modo que decimos que $t \in S(p)$ es una transición de $S(p)$.[11]

Los *Global Graphs* son representaciones gráficas de coreografías similares a BPMN o UML que resultan apropiadas para representar especificaciones de comunicación entre múltiples participantes. Los *Global Graphs* cuentan con mecanismos expresivos para representar bifurcaciones y uniones que permiten modelar protocolos de comunicación de aplicaciones.

Definición 9 (Global Graph). Dado un conjunto finito de mensajes M y un conjunto finito de participantes P, un *global graph* sobre P y M es un grafo dirigido finito cuyos nodos están etiquetados con algún elemento del conjunto $L = \{\bigcirc, \circledcirc, \diamondsuit, \boxminus\} \cup \{s \to r : m \mid s, r \in P \wedge m \in M\}$. Formalmente, un *global graph* es una tupla $\langle V, A, \Lambda \rangle$ donde V es el conjunto de vértices del grafo, $A \subseteq V \times V$ es el conjunto de aristas y $\Lambda : V \to L$ es la función de etiquetado tal que $\Lambda_{-1}(\bigcirc)$ es un único elemento (el estado inicial del grafo), y para cada $v \in V$:

1. si $\Lambda(v)$ es de la forma $s \to r : m$ entonces v tiene una única arista incidente y una única arista saliente,

[10] Un canal no ideal se puede representar utilizando una máquina de estados adicional que corrompe o pierde mensajes que pasan a través de ella.

[11] En [3] los sistemas de CFSMs son llamados «protocolos» y los canales se definen de manera implícita.

2. si $\Lambda(v) \in \{\diamondsuit, \boxminus\}$ entonces v tiene al menos una arista incidente y una arista saliente,
3. si $\Lambda(v) = \circledcirc$ entonces v no tiene ninguna arista saliente.

Una etiqueta de la forma $s \to r : m$ representa una interacción en la que la máquina s le envía el mensaje m a la máquina r. Un vértice con la etiqueta $\bigcirc$ representa el estado inicial del *global graph*, $\circledcirc$ representa la terminación de una rama o hilo de ejecución, $\boxminus$ representa la bifurcación o la unión de hilos de ejecución, y $\diamondsuit$ representa puntos en los que se abren nuevos hilos de ejecución, se fusionan hilos o marcan puntos de entrada a ciclos.

Existe un algoritmo de proyección que permite, dado un *Global Graph*, obtener una CFSM para cada uno de sus participantes. [14]

Las *Communicating Relational Networks* (CRNs), entonces, se definen del mismo modo que las ARNs pero con una definición distinta de *Conexión* basada en *global graphs*. Dado un conjunto de puertos, el conjunto de mensajes será la unión de los mensajes de los puertos y los participantes serán los mismos puertos.

Definición 10 (Conexión (en CRNs)). Dado γ, un conjunto de puertos disjuntos par a par (no comparten mensajes) y M, el conjunto de los mensajes en γ, decimos que $\langle M, \mu, \Gamma \rangle$ es una *conexión* sobre γ, donde Γ es un *global graph* donde los participantes son los puertos de γ que se intercambian mensajes de M tal que:

$$\mu_\pi^{-1}(\pi^-) \subseteq \bigcup_{\varphi \in \gamma \setminus \{\pi\}} \mu_\varphi^{-1}(\varphi^+)$$

$$y$$

$$\mu_\pi^{-1}(\pi^+) \subseteq \bigcup_{\varphi \in \gamma \setminus \{\pi\}} \mu_\varphi^{-1}(\varphi^-)$$

Definición 11 (Communicating Relational Network (CRN)). Una CRN es una tupla $\langle X, P, C, \gamma, M, \mu, \Lambda \rangle$ donde:

- $\langle X, P, C \rangle$ conforman un hipergrafo $\langle X, E \rangle$, donde X es un conjunto finito de nodos y $E = P \cup C$ es un conjunto de hiperejes (conjuntos no vacíos de X) que está particionado en *hiperejes de cómputo* $p \in P$ e *hiperejes de comunicación* $c \in C$ tales que ningún hipereje es adyacente a otro de la misma partición; y
- tres funciones de etiquetado que asignan:
  - un puerto M_x a cada nodo $x \in X$
  - un proceso $\langle \gamma_p, \Lambda_p \rangle$ a cada hipereje $p \in P$, y
  - una conexión $\langle M_c, \mu_c, \Lambda_c \rangle$ a cada hipereje $c \in C$.

En relación a la propuesta de reemplazar el uso de ARNs por CRNs presentada en [26], el lector notará que existen consideraciones que merecen su explicación. Por la forma en la que están definidas las ARNs, la semántica del sistema resultante de una serie de reconfiguraciones está definida como la composición de los autómatas de Muller con los que están etiquetados los hiperarcos tal como se detalló en la Sección 2.1. Ahora bien, al eliminar el rol de orquestador que ocupan los autómatas de Muller correspondientes a los hiperarcos de comunicación, los autómatas de Muller correspondientes a los hiperarcos de proceso ya no pueden interactuar como se espera pues han desaparecido los eventos de sincronización de sus etiquetas. Más aun, al reemplazar el mecanismo a través del cual interactúan los procesos, pasando de un modelo orquestado a una forma de coreografía de caracter asincrónica, el uso de autómatas de Muller se vuelve insuficiante puesto que lo que sea que usemos para describir el comportamiento de los procesos deberá implementar las acciones necesarias para representar la comunicación, es decir,

del envío y recepción de mensajes. En [4] Davidovich Caballero introdujo una nueva clase de autómatas finitos con la capacidad de dar soporte para el tipo de composición necesaria una vez que se reemplazan el mecanismo de comunicación sincrónico de las ARNs por un mecanismo asincrónico como el que expresan las CFSMs. A conticuación presentaremos las definiciones relativas a este lenguaje formal llamado *Autómata Finito de Comunicación Asincrónica.*

Definición 12 (Cola). [4] Una cola es un tipo de estructura de datos caracterizada por ser una secuencia de elementos first in-first out (FIFO) debido a que el primer elemento en entrar es el primer elemento en salir. Para manejarla se definen dos operaciones, una para agregar y otra para sacar elementos de la cola. Para los usos de este trabajo vamos a definir colas en las cuales se depositan y de las cuales se retiran mensajes, con las operaciones:

- Cola vacía: denotado [ ]
- Encolar mensaje: denotado como $b \ll m$, si b es una cola y m es un mensaje.
- Desencolar mensaje: denotado como $b \gg m$, si b es una cola y m es un mensaje.
- Sea una cola vacía $b = [\,]$ al aplicarle la operación $b \ll m$ queda $b = [m]$
- Sea una cola no vacía $b = [m_1, m_2, ..., m_n]$ al aplicarle la operación $b \ll m$ queda $b = [m_1, m_2, ..., m_n, m]$ del mismo modo al aplicarle $b \gg m_1$ queda $b = [m_2, ..., m_n, m]$

Definición 13 (Autómata finito de comunicación asincrónica). [4] Sea $\mathbb{P}$ un conjunto de participantes y $\mathbb{M}$ un conjunto de mensajes. Un *autómata finito de comunicación asincrónica* es una estructura $A_{\mathbb{P}} = \langle Q, B, \mathbb{C}, \Sigma, \delta, q_0, F \rangle$ tal que:

- Q es un conjunto finito de estados,
- $B \subseteq \left\{ b_{pq_n} \mid pq \in \mathbb{P}^2, n \in N, p \neq q \right\}$ es un conjunto finito de buffers (i.e. colas),
- $\mathbb{C} \subseteq \left\{ pq_n \mid pq \in \mathbb{P}^2, n \in N, p \neq q \right\}$ es un conjunto de canales tales que $B \cap \mathbb{C} = \emptyset$, y hay al menos un canal en cada sentido entre cada par de participantes,
- $\Sigma = \{\Sigma_{\text{Int}} \cup \Sigma_{\text{Ex}} \cup \Sigma_{\text{Buff}}\}$, $\Sigma \cap \mathbb{M} = \emptyset$ es el conjunto de etiquetas del autómata, siendo
  1. Σ_{Int} las acciones internas del autómata
  2. Σ_{Ex} un conjunto de etiquetas de la forma $\text{In}(c, m), \text{Out}(c, m)$ dónde $c \in \mathbb{C}$ es el canal a través del cual se resuelve la misma y $m \in \mathbb{M}$ es el mensaje.
  3. Σ_{Buff} es el conjunto de etiquetas de las acciones sobre los buffers de la forma $b \ll m$ o $b \gg m$, dónde $b \in B$ y $m \in \mathbb{M}$.
- $\delta = (\delta_{\text{Int}} \cup \delta_{\text{Ex}} \cup \delta_{\text{Buff}})$ siendo:
  1. $\delta_{\text{Int}} \subseteq Q \times \Sigma_{\text{Int}} \times Q$ es la relación de transición por acciones internas de $A_{\mathbb{P}}$,
  2. $\delta_{\text{Ex}} \subseteq Q \times \Sigma_{\text{Ex}} \times Q$ es la relación de transición de comunicación externa de $A_{\mathbb{P}}$,
  3. $\delta_{\text{Buff}} \subseteq Q \times \Sigma_{\text{Buff}} \times Q$ es la relación de transición de comunicación interna de $A_{\mathbb{P}}$
- $q_0 \in Q$ es el estado inicial, y
- $F \subseteq Q$ es el conjunto de estados finales.

De esta forma, es posible dar una nueva definición de CRN donde se reemplaza el uso de los autómatas de Muller como lenguaje formal para caracterizar el comportamiento de los procesos, por el de los Autómatas Finitos de Comunicación Asincrónica, recuperando así una noción bien fundada de composición y, con ella, una semántica operacional precisa.

La intuicion detrás de las componentes de un AFCA es la siguiente:

- Σ es el conjunto de todas las etiquetas del autómata y lo dividimos en tres subconjuntos disjuntos de la forma Σ_{Int}, Σ_{Ex} y Σ_{Buff} que corresponden a las acciones de procesamiento interno, las de comunicación externa con otros participantes y las de comunicación interna vía buffers.

- Σ_{Ex} es el conjunto de etiquetas correspondientes a la comunicación externa del autómata. Como la comunicación es dirigida punto a punto, cada etiqueta incluye a dos participantes $p_1, p_2 \in \mathbb{P}$ (emisor y receptor) y un canal $c \in \mathbb{C}$. Necesariamente en todas las etiquetas vale que p_1 y p_2 son distintos y entre cada par de participantes hay al menos un canal en cada sentido de la comunicación.
- Σ_{Buff} es el conjunto de etiquetas de acciones de comunicación interna vía buffers. Se especifican las acciones de encolar y desencolar como $b \ll m$, $b \gg m$, respectivamente, donde $b \in B$ y m es el mensaje que se inserta/extrae del mismo.
- δ es el conjunto de transciciones que se compone de δ_{Int}, las acciones internas, δ_{Ex} las transiciones de comunicación con otros agentes y δ_{Buf} las acciones de buffer.
- δ_{Ex} se compone de dos tipos de acciones que representan el intercambio de mensajes $m \in \mathbb{M}$ entre dos autómatas a través de un canal de comunicación $c \in \mathbb{C}$. Las acciones de entrada o input representan la recepción de un mensaje de algún proceso externo al correspondiente al autómata, denotadas $\text{In}(c, m)$. Las acciones de output o salida que representan el envío de un mensaje y se denotan $\text{Out}(c, m)$. Estas acciones son la parte externa de la comunicaión asincrónica. Como los canales son unidireccionales decimos que para todo par de participantes existen una cantidad finita de pares de canales.
- δ_{Buff} es la relación de transición de comunicación interna mediante buffers.
- Formalmente, $F \subseteq Q$. Es el conjunto de estados finales, los posibles estados a los que una ejecución llega y es aceptada como válida. Para los autómatas asincrónicos pedimos también que al llegar a este estado final los buffers se encuentren vacíos. Para esto definimos lo siguiente:

Definición 14 (Configuración instantánea). [4] Dados $\mathcal{P}$ un conjunto de participantes y $\mathcal{M}$ un conjunto de mensajes. A un autómata con comunicación asincrónica $A_{\mathcal{P}} = \langle Q, B, \mathcal{C}, \Sigma, \delta, q_0, F \rangle$, definimos:

Una configuración instantánea del autómata como $\langle q, \Omega \rangle \in Q \times \{(m_b)_{b \in B} \mid m_b \in \mathcal{M}^*\}$ donde:

- q es el estado actual
- Ω es el conjunto de buffers (con sus respectivos contenidos hasta el momento). Decimos que Ω es de la forma $\omega_{b_1}, \omega_{b_2}, ..., \omega_{b_n}$ con $b_i \in B$.
- Decimos que una configuración es inicial si $q = q_0$ y $\Omega = \{[\,], ..., [\,]\}$
- Decimos que una configuración es final si $q \in F$ y $\Omega = \{[\,], ..., [\,]\}$

Definición 15 (Relación de transición entre configuraciones). [4] Sean $A_{\mathcal{P}} = \langle Q, B, \mathcal{C}, \Sigma, \delta, q_0, F \rangle$, $q_1, q_2 \in Q$, $m \in \mathcal{M}$ definimos $\vdash$, relación de transición entre configuraciones, como:

1. $\langle q_1, \Omega \rangle \vdash \langle q_2, \Omega \rangle \Leftrightarrow \langle q_1, r, q_2 \rangle \in \delta \wedge r \in \Sigma \setminus \Sigma_{Buff}$
2. $\langle q_1, \{\omega_{b_1}, ..., \omega_{b_i}, ..., \omega_{b_n}\} \rangle \vdash \langle q_2, \{\omega_{b_1}, ..., m : \omega_{b_i}, ..., \omega_{b_n}\} \rangle \Leftrightarrow \langle q_1, b_i \ll m, q_2 \rangle \in \delta$
3. $\langle q_1, \{\omega_{b_1}, ..., \omega_{b_i} : m, ..., \omega_{b_n}\} \rangle \vdash \langle q_2, \{\omega_{b_1}, ..., \omega_{b_i}, ..., \omega_{b_n}\} \rangle \Leftrightarrow \langle q_1, b_i \gg m, q_2 \rangle \in \delta$

Definición 16 (Traza de un AFCA). [4] Llamamos traza a una secuencia posible de acciones de un autómata. Se define como una secuencia finita de etiquetas de estado y transición alternadas, que comienza y termina con un estado. Dado un autómata $A_{\mathcal{P}} = \langle Q, B, \mathcal{C}, \Sigma, \delta, q_0, F \rangle$, una traza tiene la forma $[q_0, \sigma_1, q_1, ..., q_{n-1}, \sigma_n, q_n]$ donde:

- q_0 es el estado inicial del autómata
- $q_i \in Q$,
- $\sigma_i \in \Sigma$ y
- $\langle q_{i-1}, \sigma_i, q_i \rangle \in \delta$

El comportamiento de un AFCA es el conjunto de todas las trazas posibles.

Definición 17 (Ejecución de un AFCA). [4] Una ejecución es una secuencia $\langle q, \Omega \rangle \vdash \langle q_1, \Omega_1 \rangle \vdash ... \vdash \langle q_n, \Omega_n \rangle$ donde $\langle q, \Omega \rangle$ es la configuración inicial, $\langle q_n, \Omega_n \rangle$ es una configuración final y $\forall i \in [1, ..., n]$, $\langle q_i, \Omega_i \rangle$ es una configuración válida (una configuración que no es de deadlock, mensajes huérfanos ni recepción no especificada).

Ejemplo de un Autómata Finito de Comunicación Asincrónica.

Considere el AFCA $S = \langle \{q_0, .., q_5\}, \{b_0\}, \{sr_1, sr_2, rs_1\}, \Sigma_A, \delta_A, q_0, \{q_4\} \rangle$, donde

$\Sigma = \{wait, b_0 \ll m_1, b_0 \gg m_1, Out(sr_1, a), Out(sr_2, c), In(rs_1, b), In(rs_2, d)\}$

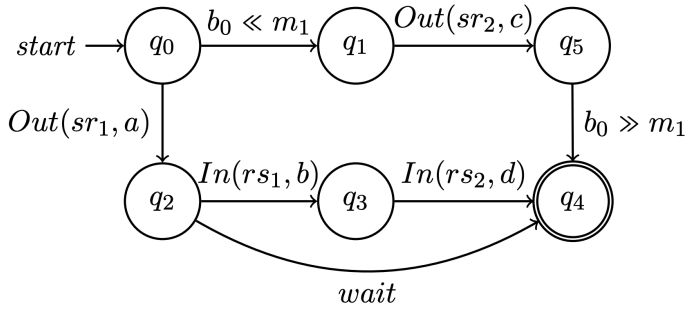


Figura 8: AFCA del ejemplo, tomado de [4].

En el ejemplo podemos ver los tres tipos de transiciones que los AFCA pueden realizar.

- Transiciones de comunicación interna: $b_0 \ll m_1$ y $b_0 \gg m_1$
- Transiciones de comunicación externa: $Out(sr_1, a), Out(sr_2, c), In(rs_1, b)$ y $In(rs_2, d)$
- Transiciones internas: $wait$

Dado que los AFCA representan procesos, o servicios, que interactúan entre sí como parte de un sistema, necesitamos definir la operación de composición.

Definición 18 (Compatibilidad). [4] Dados $\mathcal{P}$ un conjunto de participantes, $\mathcal{M}$ un conjunto de mensajes y $A_{\mathcal{P}} = \{A_p \mid p \in \mathcal{P}\}$ un conjunto finito de autómatas finitos de comunicación asincrónica $A_{p_i} = \langle Q_{p_i}, B_{p_i}, \mathcal{C}_{p_i}, \Sigma_{p_i}, \delta_{p_i}, q_{0p_i}, F_{p_i} \rangle$, diremos que $A_{\mathcal{P}}$ es *compatible* si se satisface que para todos $p_i, p_j \in \mathcal{P}$, $\Sigma_{p_i} \cap \Sigma_{p_j} = \emptyset$ y $B_{p_i} \cap B_{p_j} = \emptyset$.

Definición 19 (Composición). [4] Dados $\mathcal{P}$ un conjunto de participantes, $\mathcal{M}$ un conjunto de mensajes y $A_{\mathcal{P}} = \{A_p \mid p \in \mathcal{P}\}$ un conjunto finito de autómatas finitos de comunicación asincrónica compatibles $A_{p_i} = \langle Q_{p_i}, B_{p_i}, \mathcal{C}_{p_i}, \Sigma_{p_i}, \delta_{p_i}, q_{0p_i}, F_{p_i} \rangle$. Luego, definimos la composición $A_{\mathcal{P}} = \|_{1..n} A_{p_i}$ como $A = \langle Q, B, \mathcal{C}, \Sigma, \delta, q_0, F \rangle$ tal que:

- $Q_{\mathcal{P}} = \Pi_{p_i \in \mathcal{P}} Q_{p_i}$ (i.e. el conjunto de estados de la composición es el producto cartesiano de los estados de los autómatas componentes),
- $B_{\mathcal{P}} = \bigcup_{p_i \in \mathcal{P}} B_{p_i} \cup \{\omega_c \mid c \in \mathcal{C}_{p_i \to p_j}\}$, dónde $\mathcal{C}_{p_i \to p_j} = \{c \in \mathcal{C}_{p_i} \cap \mathcal{C}_{p_j}\}$ (i.e. el conjunto de nombres de buffers del autómata resultante se compone de los buffers de cada autómata

participante en la composición, junto con uno nuevo por cada canal compartido entre cada par de autómatas, dichos canales son aquellos mediante los cuales los autómatas a componer intercambian mensajes entre sí),

- $\mathcal{C}_{\mathcal{P}} = \bigcup_{p_i \in \mathcal{P}} \mathcal{C}_{p_i} \setminus \mathcal{C}_{p_i \to p_j}$,
- $\Sigma_{\mathcal{P}} = \Sigma_{\mathcal{P}Int} \cup \Sigma_{\mathcal{P}Ex} \cup \Sigma_{\mathcal{P}Buff}$ tal que:

  1) $\Sigma_{\mathcal{P}Int} = \bigcup_{p_i \in \mathcal{P}} \Sigma_{p_i Int}$,

  2) $\Sigma_{\mathcal{P}Ex} = \bigcup_{p_i \in \mathcal{P}} \Sigma_{p_i Ex} \setminus \Sigma_{p_i \mapsto p_j}$ y

  3) $\Sigma_{\mathcal{P}Buff} = \bigcup_{p_i \in \mathcal{P}} \Sigma_{p_i Buff} \cup \Sigma_{p_i \mapsto p_j}$,

  donde $\Sigma_{p_i \mapsto p_j} = \{\langle p_i, p_j, c\rangle \mid p_i, p_j \in \mathcal{P}, c \in \mathcal{C}_{p_i \to p_j}\}$
- $\delta_{\mathcal{P}} = \delta_{\mathcal{P}Int} \cup \delta_{\mathcal{P}Ex} \cup \delta_{\mathcal{P}Buff}$ tal que:

  1. $\delta_{\mathcal{P}Int} \subseteq Q_{\mathcal{P}} \times \Sigma_{\mathcal{P}Int} \times Q_{\mathcal{P}}$ es la relación de transición interna, tal que que se satisface la siguiente fórmula:

  $$(\forall \langle q, \sigma, q'\rangle \in \delta_{\mathcal{P}Int})(\exists p_i \in \mathcal{P})\Big(\exists q_{p_i}, {q'}_{p_i} \in Q_{p_i}, \sigma \in \Sigma_{p_i}\Big)\Big(\langle q_{p_i}, \sigma, {q'}_{p_i}\rangle \in \delta_{p_i Int}\Big)$$

  2. $\delta_{\mathcal{P}Ex} \subseteq Q_{\mathcal{P}} \times \Sigma_{\mathcal{P}Ex} \times Q_{\mathcal{P}}$ es la relación de transición de comunicación externa, tal que que se satisface la siguiente fórmula:

  $$(\forall \langle q, \sigma, q'\rangle \in \delta_{\mathcal{P}Ex})(\exists p_i \in \mathcal{P})\Big(\exists q_{p_i}, {q'}_{p_i} \in Q_{p_i}, \sigma \in \Sigma_{p_i}\Big)\Big(\langle q_{p_i}, \sigma, {q'}_{p_i}\rangle \in \delta_{p_i Ex}\Big)$$

  3. $\delta_{\mathcal{P}Buff} \subseteq Q_{\mathcal{P}} \times \Sigma_{\mathcal{P}Buff} \times Q_{\mathcal{P}}$ es la relación de transición de comunicación interna tal que se satisface la siguiente fórmula:

  $$\begin{aligned}
  \langle q, \sigma, q'\rangle \in \delta_{\mathcal{P}Buff} \ \text{ sii } (q = \langle q_k\rangle_{p_k \in \mathcal{P}} \wedge q' = \langle {q'}_{k'}\rangle_{p_{k'} \in \mathcal{P}} \wedge \\
  ((\exists p_i \in \mathcal{P})((\forall p_k \in \mathcal{P})(p_k \neq p_i \Longrightarrow q_k = {q'}_k) \wedge \\
  \Big(\exists \omega_b \in B_{p_i}, \omega_b\{\gg, \ll\}m \in \Sigma_{p_i Buff}\Big)\Big(\langle q_i, \omega_b\{\gg, \ll\}m, {q'}_i\rangle \in \delta_{p_i Buff}\Big) \wedge \\
  \sigma = \omega_b\{\gg, \ll\}m) \vee \\
  (\exists p_i, p_j \in \mathcal{P})(p_i \neq p_j \wedge (\forall p_k \in \mathcal{P})\big((p_k \neq p_i \wedge p_k \neq p_j) \Longrightarrow q_k = {q'}_k\big) \wedge \\
  (\Big(\exists Out\Big(c_{p_i,p_{j_n}}, m\Big) \in \Sigma_{p_i Ex}\Big)(\langle q_i, Out\Big(c_{p_i,p_{j_n}}, m\Big), {q'}_i\rangle \in \delta_{p_i Ex} \wedge \\
  \sigma = \omega_{c_{p_i p_{j_n}}} \ll m) \vee \\
  \Big(\exists In\Big(c_{p_j,p_{i_n}}, m\Big) \in \Sigma_{p_i Ex}\Big)(\langle q_i, In\Big(c_{p_j,p_{i_n}}, m\Big), {q'}_i\rangle \in \delta_{p_i Ex} \wedge \\
  \sigma = \omega_{c_{p_j,p_{i_n}}} \gg m))))
  \end{aligned}$$

  Además vale que $\forall q_i, q_j \in Q_{\mathcal{P}}, m \in \mathcal{M} \mid \langle q_i, [\omega_b]_{b \in B_{\mathcal{P}}} \gg m, q_j\rangle$

  $\langle q_i, [\omega_b]_{b \in B_{\mathcal{P}}} \gg m, q_j\rangle \in \delta_{\mathcal{P}Buff} \Leftrightarrow \exists {q'}_i, {q'}_j \in Q_{\mathcal{P}}$ tal que
  $\langle {q'}_i, [\omega_b]_{b \in B_{\mathcal{P}}} \ll m, {q'}_j\rangle \in \delta_{\mathcal{P}Buff}$ y se cumple una de las siguientes condiciones:

  1. ${q'}_j = q_i$
  2. $\exists \sigma \in \Sigma_{\mathcal{P}}$ tal que $\langle {q'}_j, \sigma, q_i\rangle \in \delta_{\mathcal{P}} \wedge \sigma \neq [\omega_b]_{b \in B_{\mathcal{P}}} \gg m$

3. $\exists s = \left[q'_j, ..., q_i\right]$ donde s es una secuencia finita de estados tales que $s[0] = q'_j, s[n-1] = q_i$ y sea $0 \ll x \ll$ n-1,$s[x] \in Q_{\mathcal{P}}$ y $\forall s[x], s[x+1], \exists \sigma \in \Sigma_{\mathcal{P}} \wedge \langle s[x], \sigma, s[x+1] \rangle \in \delta_{\mathcal{P}}$

- $q_0 = \langle q_{0_{p_1}}, ..., q_{0_{p_n}} \rangle$, y
- $F_{\mathcal{P}} = \bigcup_{1..n} F_{p_i}$.

3. Diseño

En este capítulo abordaremos distintas decisiones que se tomaron a la hora de implementar la infraestructura necesaria para SEArch: el *broker*, el *middleware* y las bibliotecas necesarias para desarrollar *Service Clients* y *Service Providers*.

En la primera sección describiremos la interfaz expuesta a quienes desarrollen software que utiliza la infraestructura de SEArch como clientes (*Service Clients*). En la segunda, describiremos cómo implementar proveedores para esta misma infraestructura (*Service Providers*). Luego, la tercera sección ilustra ambos escenarios a través de un ejemplo simple en el cual implementamos un *Service Provider* y un *Service Client* que se comunican entre sí. El ejemplo es un simple «ping-pong» de mensajes, tomado de ChorGram [15]. La cuarta sección describe el protocolo de comunicación interna entre las distintas componentes implementadas (quienes implementan software para esta infraestructura no necesitan conocer estos detalles). La quinta sección trata sobre los contratos de interoperabilidad implementados.

Es importante señalar que algunas decisiones de diseño no son apropiadas para una implementación que pueda satisfacer atributos de calidad necesarios para un entorno de producción, sino que se trata de una prueba de concepto. La última sección del capítulo identifica algunas limitaciones en el diseño y sugiere posibles maneras de resolverlas.

En la Figura 9 se ilustran las componentes de la infraestructura del Service Client y del Service Provider cuando hay un canal establecido. Ilustramos allí el caso más simple posible: un canal de solo dos participantes (un Service Client y un Service Provider). Para establecer el canal se require también de la participación del Service Broker, lo cual no se ilustra en este diagrama. A lo largo de este capítulo y el próximo haremos foco sobre distintas componentes de este diagrama.

Quienes implementen software que utilice esta infraestructura, lo único que deberán hacer es:

1. descargar el *middleware* y ejecutarlo,
2. descargar el cliente (*stub*) del middleware correspondiente al lenguaje de programación en el cual desarrollen su *Service Client* o *Service Provider*, y utilizarlo para declarar servicios y canales, y para enviar y recibir mensajes.

El *middleware* se encargará de establecer las conexiones necesarias con el *broker* y con otros *middlewares* para descubrir, concetarse a y comunicarse con los demás participantes del canal.

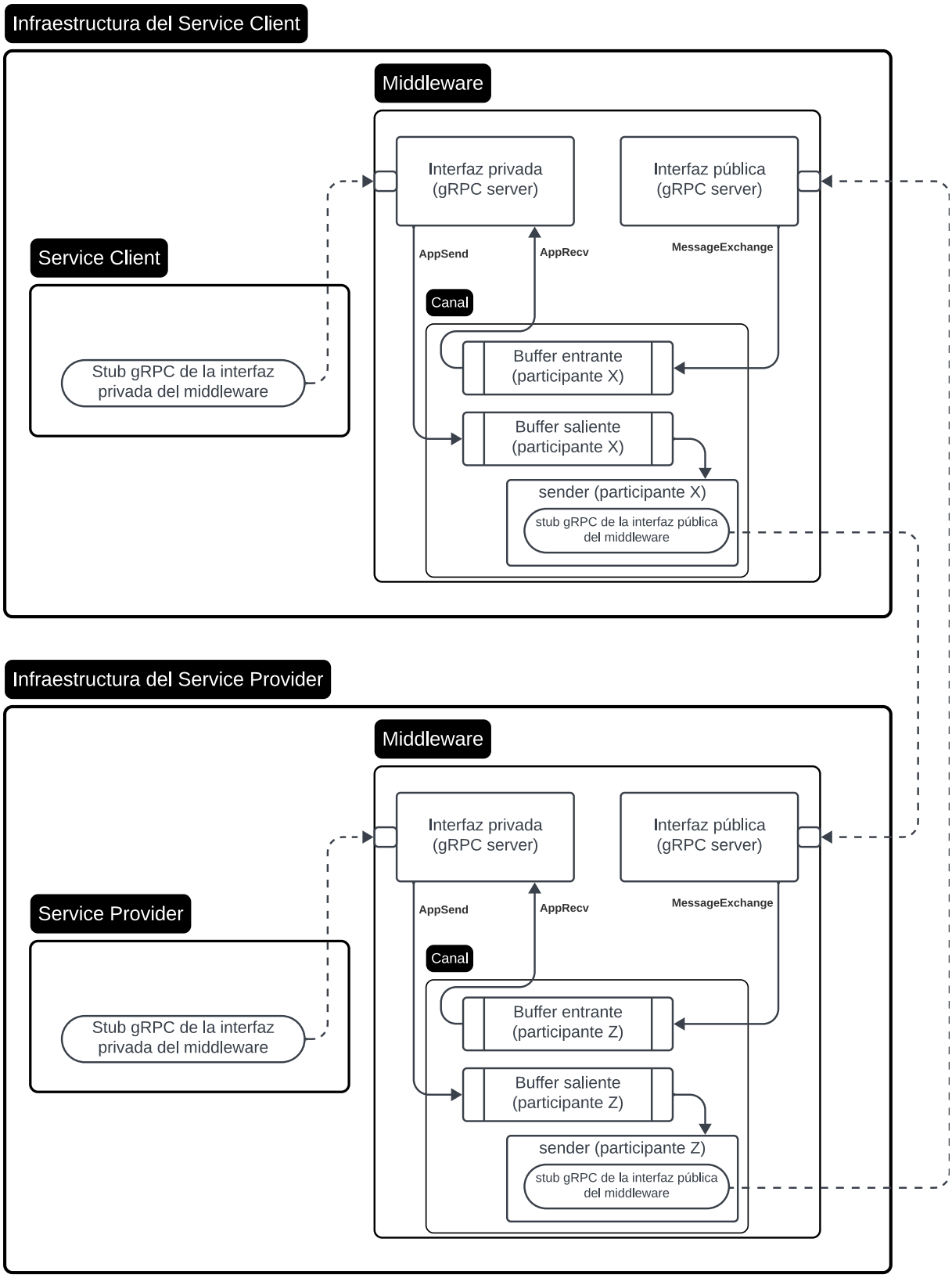


Figura 9: Diagrama de componentes sin el Broker cuando un canal simple está establecido.

3.1. Interfaz que el Middleware expone a Service Clients/Providers

Diseñamos el *middleware* como un servicio que tanto quienes implementen *Service Clients* como *Service Providers* deben ejecutar en su propia infraestructura. Cada *middleware* posee dos interfaces: una pública, donde debe aceptar conexiones entrantes del *broker* y de otros *middle-*

wares, y una interfaz privada, donde solamente debe aceptar conexiones de parte de los *Service Clients/Providers* «propios»[12]. Cada interfaz escucha en un puerto TCP distinto.

Para desarrollar software que funcione en el ecosistema de SEArch, además de la escritura del contrato, es necesario interactuar con el *middleware* para declarar servicios y canales, y para realizar envíos y recepciones de mensajes. En esta sección describimos en detalle la interfaz privada que el *middleware* expone a quienes desarrollen software para el ecosistema SEArch.

Se puso especial énfasis en diseñar una interfaz simple de alto nivel para quienes implementen *Service Clients* y *Service Providers*, y delegar en la infraestructura las complejidades asociadas al *brokering* y al manejo de sesiones, fallos, establecimiento de conexiones, etc.

La interfaz privada que el *middleware* ofrece a *Service Clients* y *Service Providers* está definida utilizando Protocol Buffers [16] de la siguiente manera:

```
service PrivateMiddlewareService {
  rpc RegisterChannel(RegisterChannelRequest) returns(RegisterChannelResponse);
  rpc RegisterApp(RegisterAppRequest) returns (stream RegisterAppResponse);
  rpc CloseChannel(CloseChannelRequest) returns (CloseChannelResponse);
  rpc AppSend(AppSendRequest) returns (AppSendResponse);
  rpc AppRecv(AppRecvRequest) returns (AppRecvResponse);
}
```

Fragmento de código 1: Definición de la interfaz privada del Middleware.

Describimos a continuación cada una de las RPCs (*Remote Procedure Calls*) que se utilizan para implementar un *Service Client*. *AppSend*, *AppRecv* y *CloseChannel* son utilizadas tanto por clientes como proveedores una vez se encuentra establecido un canal de comunicación. *RegisterChannel* y *RegisterApp* son utilizadas únicamente por *Service Clients* para registrar canales y servicios, respectivamente. Describiremos esta última en la siguiente sección.

Registro de un canal

`RegisterChannel` es utilizada por los *Service Clients* para declarar un canal de comunicación. Para ello, se envía el siguiente tipo de mensaje al *middleware*:

```
message RegisterChannelRequest {
    GlobalContract requirements_contract = 1;
    map<string, RemoteParticipant> preset_participants = 2;
}
```

Fragmento de código 2: Definición del mensaje que el Service Client envía a su middleware para declarar un canal de comunicación.

Además del contrato de requerimiento (un contrato global que describe al canal – es decir, tiene una definición de cómo se comporta cada participante que se comunica en el canal), se envía un dato opcional llamado `preset_participants` que permite al *Service Client* excluir a ciertos participantes del *brokering*, y de esa manera designar explícitamente qué *Service Provider* cumple el papel de alguno(s) de los participantes del canal de comunicación. Para esto se envía un *map* en el cual la clave es el nombre del participante según el contrato global, y el valor asociado es del tipo `RemoteParticipant` que contiene la información de conexión de dicho proveedor.

El *middleware* responde a esta solicitud con un identificador del canal. El *Service Client* debe luego hacer referencia a este identificador al utilizar los restantes métodos para utilizar el canal (enviar un mensaje, recibir un mensaje y cerrar el canal).

[12]No se implementó autenticación en el marco de esta tesis. Para hacerlo, hay soporte de múltiples mecanismos en las bibliotecas de gRPC utilizadas (mTLS, OAuth2 y otros). `https://grpc.io/docs/guides/auth/`

Envío y recepción de mensajes

`AppSend` y `AppRecv` son utilizadas tanto por *Service Clients* como por *Service Providers*. Ambas solicitudes reciben como parámetros un identificador de canal y un nombre de participante. `AppSend` recibe, además, el tipo de mensaje a enviar y su contenido.

La recepción se bloquea si el buffer está vacío (es decir, no hay mensajes en la cola sobre la que se llamó el `AppRecv`). El envío se bloquea si el buffer está lleno. Por defecto, cada cola de mensajes[13] tiene una capacidad de 100 mensajes, pero este valor se puede modificar en la configuración del *middleware*. Para evitar bloquearse indefinidamente, al realizar un `AppSend` o `AppRecv` se debe establecer un *deadline* a la invocación.

No se garantiza orden entre llamadas concurrentes a `AppSend` o `AppRecv`: si el *Service Client/ Provider* utiliza distintos hilos de ejecución y necesita garantizar un orden entre envíos o recepciones, es responsabilidad del mismo sincronizar dichas llamadas. Sin embargo, una vez que los mensajes son encolados para su envío, se garantiza su entrega al participante remoto en el mismo orden en que fueron encolados[14].

Cierre de un canal

`CloseChannel` es utilizada tanto por *Service Clients* como por *Service Providers* para indicar el cierre del canal. Esto hace que el *middleware* envíe a los demás participantes los mensajes que tenga en buffer saliente y luego envíe una notificación de cierre a los demás participantes con los cuales tenía establecida una conexión. Si el *middleware* tiene algún mensaje en *buffer* dirigido al participante que está intentando cerrar el canal, devuelve un error.

3.2. Registro de Service Provider ante el Service Repository

La RPC `RegisterApp` expuesta en la interfaz privada del *middleware* es utilizada por los *Service Providers* para registrarse ante el *broker* y ante su propio *middleware*. El *Service Provider* envía a su *middleware* el contrato de provisión. El *middleware*, a su vez, reenvía este contrato al *broker* para registrar el servicio en el repositorio. Una vez registrado el servicio, el *middleware* responde al *Service Provider* con una confirmación (ACK). Esta RPC se mantiene abierta entre el *Service Provider* y el *middleware*, ya que a través de este *stream* el *middleware* notifica al *Service Provider* cada vez que inicia un canal que lo involucra.

En el desarrollo de esta tesis no se implementó el Service Repository como un servicio independiente, sino que sus funciones son implementadas por el Service Broker. En la solución implementada, para registrar un Service Provider, éste debe enviar un mensaje al Service Broker a través de su middleware.

[13] Cada canal tiene dos colas por cada participante remoto: una para mensajes salientes y otra para mensajes entrantes.

[14] Para garantizar esto se utilizan Go channels como buffers y un *stream* gRPC para la comunicación entre middlewares, ver Sección 4.4.

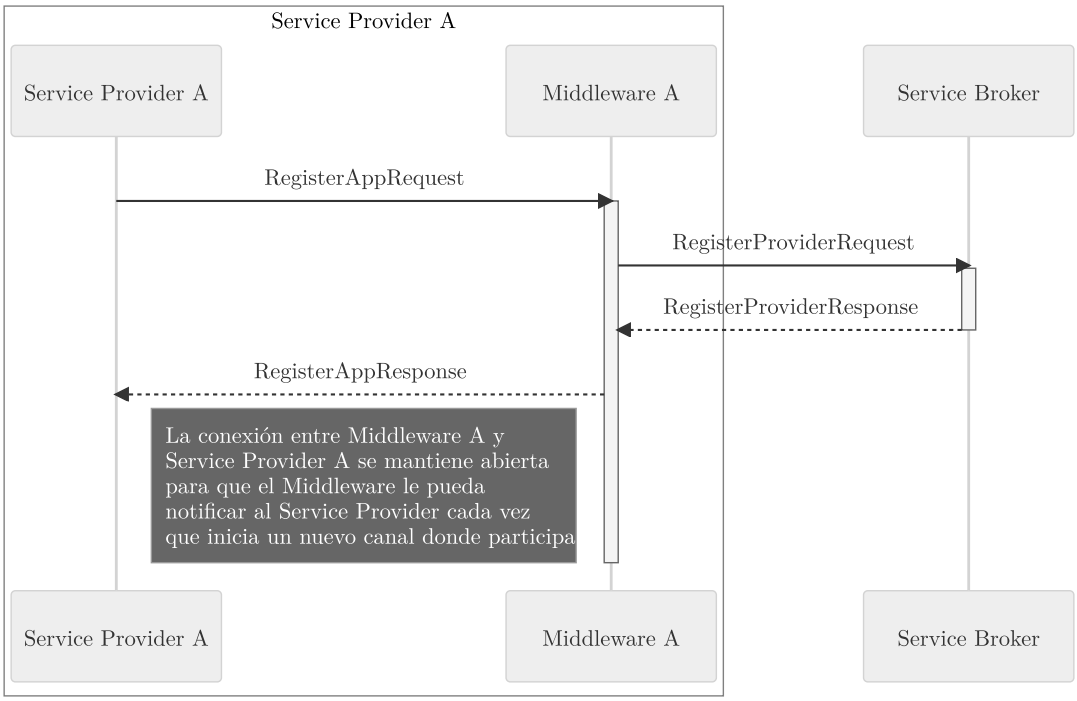


Figura 10: Diagrama de secuencia del proceso de registro de un Service Provider.

La Figura 10 ilustra el proceso de registro de un Service Provider ante el Service Broker. El Service Provider envía un mensaje del tipo `RegisterAppRequest` a su middleware utilizando la RPC `RegisterApp` que el middleware expone en su interfaz privada. Éste, a su vez, se comunica con el Service Broker con la RPC llamada `RegisterProvider`. El middleware envía un único mensaje del tipo `RegisterProviderRequest` que está definido así:

```
message RegisterProviderRequest {
    LocalContract contract = 1;
    string url = 2;
}
```

Fragmento de código 3: Definición del mensaje que el Middleware envía al Broker para registrar un Service Provider.

El Broker, a su vez, responde asignando un UUID al Service Provider. Dicho UUID identifica al Service Provider en todo el ecosistema de SEArch. El middleware, luego de recibir este UUID, se lo reenvía al Service Provider como primer mensaje del *server-side stream* que se establece entre el middleware y el Service Provider. Los mensajes del stream tienen esta definición en Protobuf:

```
message RegisterAppResponse {
    oneof ack_or_new {
        string app_id = 1;
        InitChannelNotification notification = 2;
    }
}
```

Fragmento de código 4: Definición de los mensajes que el Middleware envía al Service Provider para notificar sobre un nuevo canal o para confirmar el registro ante el Broker.

Luego de la recepción del primer mensaje del stream donde se confirma el registro del servicio en el *Service Repository*, el Service Provider deberá esperar a recibir mensajes del tipo `InitChannelNotification` de parte de su *middleware*. Estos mensajes solamente contienen un UUID que identifica al canal. El Service Provider deberá utilizar este UUID para identificar el

canal en el que participará en los subsiguientes mensajes que intercambie con el Middleware (`Recv(participant)` y `Send(participant, msg)`).

3.3. Ejemplo de implementación: Ping-Pong

Para ilustrar el uso de la interfaz expuesta, en esta subsección mostramos cómo implementar dos servicios que se comporten de acuerdo a la siguiente coreografía global, inspirada en un ejemplo de la documentación de ChorGram[15]:

```
repeat Ping {             ... después de al menos un intercambio de ping-pongs
  Ping -> Pong: ping ;
  Pong -> Ping: pong
};
sel {                     ... el cliente cierra la sesión de alguna de las dos maneras:
  Ping -> Pong: finished  ... sin esperar respuesta del servidor
  +
  Ping -> Pong: bye;      ... o requiriendo una respuesta del servidor
  Pong -> Ping: bye
}
```

Fragmento de código 5: Global Graph del protocolo ping-pong.

En este ejemplo, tenemos dos participantes: `Ping` y `Pong`. Ping cumple el rol de cliente. Ping envía un mensaje de tipo `ping` y espera una respuesta de tipo `pong` de parte del participante Pong. Esto lo realiza al menos una vez y lo puede repetir una cantidad indefinida de veces. Luego, puede cerrar la sesión de dos formas: enviando un mensaje `finished` sin esperar respuesta, o enviando un mensaje `bye`, tras lo cual espera la respuesta `bye` de Pong.

Este contrato lo convertimos a dos CFSMs utilizando la herramienta `gc2fsa` de ChorGram, que produce el siguiente *output* en formato FSA (*Finite State Automata*):

```
.outputs Ping
.state graph
0 1 ! ping 4
2 1 ! bye 5
2 1 ! finished 1
3 1 ! *<1 0
3 1 ! >*1 2
4 1 ? pong 3
5 1 ? bye 1
.marking 0
.end

.outputs Pong
.state graph
0 0 ? ping 4
2 0 ? bye 5
2 0 ? finished 1
3 0 ? *<1 0
3 0 ? >*1 2
4 0 ! pong 3
5 0 ! bye 1
.marking 0
.end
```

Fragmento de código 6: CFSMs obtenidas a partir del Global Graph del protocolo ping-pong en formato FSA.

El formato FSA, que representa CFSMs, es muy simple: las líneas `.outputs [string]` marcan el comienzo de una CFSM, siendo `[string]` el nombre de la misma. Esa línea es seguida por una línea `.state graph`, y luego se suceden las transiciones entre estados de la CFSM. Cada

[15]Se puede consultar la sintaxis de este formato en `https://bitbucket.org/eMgssi/stable_chorgram/wiki/syntax.md`

transición representa un envío o recepción de mensaje. En FSA, cada transición es una línea con cinco elementos:

- el estado origen,
- el nombre de la CFSM a la cual se le envía o recibe un mensaje (o el índice numérico en el que la CFSM aparece en el archivo, empezando por 0),
- un caracter `?` si se trata de una recepción, un caracter `!` si se trata de un envío,
- el nombre del tipo de mensaje,
- el estado destino.

Finalmente, la línea `.marking <state>` indica cuál es el estado inicial de la CFSM, y se marca el fin de la definición de la CFSM con la línea `.end`.

Estas dos CFSMs se representan gráficamente de la siguiente manera:

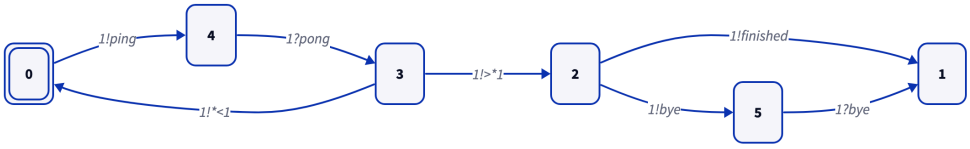


Figura 11: La CFSM de Ping.

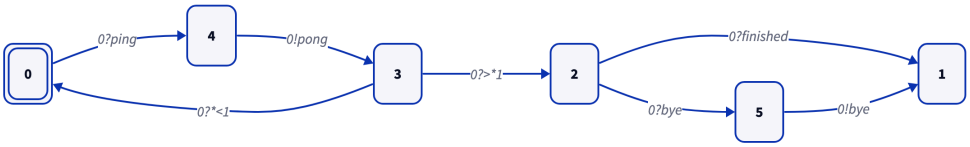


Figura 12: La CFSM de Pong.

Como se puede apreciar en este ejemplo, `gc2fsa` introdujo los mensajes `*<1` y `*>1` para representar los mensajes de sincronización que hacen que Ping le comunique a Pong si decidió iniciar una nueva iteración del ciclo o salir del mismo, respectivamente.

Para los ejemplos utilizaremos fragmentos de código Go, pero en el siguiente capítulo mostraremos un ejemplo más completo utilizando distintos lenguajes. Se excluyen de los fragmentos algunos detalles sobre parámetros de conexión al middleware y manejo de errores para simplificar la lectura. Se asume que ambos servicios tienen acceso (a través de una conexión de red) a una instancia propia del *middleware*. En el repositorio de código de esta tesis se puede hallar este ejemplo completo con todos sus detalles escrito como un test de integración (archivo `internal/middleware/middleware_test.go`, test `TestPingPongFullExample`).

Service Provider: Pong

Primero el Service Provider establece una conexión con su *middleware* y obtiene un *stub* (cliente) gRPC a partir de esa conexión. El *stub* es el objeto que se utiliza para invocar las RPCs expuestas por el *middleware*. En el Fragmento de código 7, el objeto *client* tiene un método para cada una de las RPCs detalladas en Fragmento de código 1. Además, el módulo `pb` importado contiene las definiciones de todos los mensajes que se utilizan en dichas RPCs, de manera tal que facilita la escritura de los parámetros de las RPCs y la lectura de las repuestas.

Existen *stubs* para distintos lenguajes de programación, los cuales son generados de forma automática en función de las definiciones de interfaz del *middleware*. Asumiremos en esta sección que dichos *stubs* ya fueron descargados por quien desarrolla los ejemplos. En el siguiente capítulo se ilustrará con un ejemplo más detallado cómo descargar y utilizar estos *stubs* en distintos lenguajes.

```go
import pb "github.com/pmontepagano/search/gen/go/search/v1"
conn, _ := grpc.Dial(middlewareURL, []grpc.DialOption...)
client := pb.NewPrivateMiddlewareServiceClient(conn)
```

Fragmento de código 7: Conexión del Service Provider hacia su Middleware.

Luego, el Service Provider registra el contrato de provisión, y al hacerlo, abre un *stream* a través del cual el middleware notificará al Service Provider cada vez que se inicia un canal que lo involucra. Como contrato de provisión utilizamos la CFSM obtenida anteriormente.

```go
req := pb.RegisterAppRequest{
  ProviderContract: &pb.LocalContract{
    Contract: []byte(`
        .outputs Pong
        .state graph
        0 Ping ? ping 4
        2 Ping ? bye 5
        2 Ping ? finished 1
        3 Ping ? *<1 0
        3 Ping ? >*1 2
        4 Ping ! pong 3
        5 Ping ! bye 1
        .marking 0
        .end
    `),
    Format:   pb.LocalContractFormat_LOCAL_CONTRACT_FORMAT_FSA,
  },
}
stream, _ := client.RegisterApp(context.Background(), &req)
ack, err := stream.Recv()
if err != nil {
  t.Error("Could not receive ACK from RegisterApp")
}
```

Fragmento de código 8: Registro del Service Provider.

Después de la primera recepción, en la que el Service Provider recibe confirmación de haber sido registrado en el *Service Registry* (Fragmento de código 8, línea 20), el Service Provider solamente debe recibir sobre ese stream nuevos mensajes que representan el inicio de un nuevo canal en el cual participa dicho Service Provider (Fragmento de código 9, línea 2).

```go
for {
  newSess, err := stream.Recv()
  channelID := newSess.regappResp.GetNotification().GetChannelId()
  go func(channelID string, client pb.PrivateMiddlewareServiceClient) {
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()
    for {
      recvResponse, err := client.AppRecv(ctx, &pb.AppRecvRequest{
        ChannelId:   channelID,
        Participant: "Other",
      })
      sendResponse, err := client.AppSend(ctx, &pb.AppSendRequest{
        ChannelId: channelID,
        Recipient: "Other",
        Message: &pb.AppMessage{
          Body: recvResponse.Message.Body,
          Type: "pong",
        },
      })
      // Continúa con el manejo de "bye" y "finished", se omite aquí.
    }
  }(channelID, client)
}
```

Fragmento de código 9: El Service Provider espera el inicio de un nuevo canal de comunicación y luego interactúa con el mismo.

Como se ve en el ejemplo, una vez recibida la notificación de inicio de un canal nuevo, el Service Provider solo necesita invocar los métodos `AppRecv` y `AppSend` utilizando el identificador del canal y el nombre del participante con el cual necesita comunicarse.

Service Client: Ping

La API a utilizar por un Service Client es casi idéntica a la que utilizan los Service Providers. La conexión con su middleware es la misma (ver Fragmento de código 7).

El *Service Client* declara el canal enviando el contrato de requerimiento a través de la RPC `RegisterChannel` (línea 31)[16]. Como parte del mensaje, el *Service Client* indica cuál de las distintas CFSMs del contrato global es la que lo representa a él (campo `InitiatorName`, línea 27).

```
1  req := pb.RegisterChannelRequest{
2    RequirementsContract: &pb.GlobalContract{
3      Contract:      []byte(`
4          .outputs Ping
5          .state graph
6          0 1 ! ping 4
7          2 1 ! bye 5
8          2 1 ! finished 1
9          3 1 ! *<1 0
10         3 1 ! >*1 2
11         4 1 ? pong 3
12         5 1 ? bye 1
13         .marking 0
14         .end
15
16         .outputs Pong
17         .state graph
18         0 0 ? ping 4
19         2 0 ? bye 5
20         2 0 ? finished 1
21         3 0 ? *<1 0
22         3 0 ? >*1 2
23         4 0 ! pong 3
24         5 0 ! bye 1
25         .marking 0
26         .end`),
27     InitiatorName: "Ping",
28     Format:        pb.GlobalContractFormat_GLOBAL_CONTRACT_FORMAT_FSA,
29   },
30 }
31 regResult, err := client.RegisterChannel(context.Background(), &req)
32 if err != nil {
33   t.Error("Received error from RegisterChannel")
34 }
35 channelID := regResult.ChannelId
```

Fragmento de código 10: Declaración de un canal por parte de un Service Client con un sistema de CFSMs como contrato.

Al igual que en el ejemplo del *Service Provider*, una vez obtenido el identificador del canal, se pueden realizar envíos y recepciones con `AppSend` y `AppRecv`:

[16]El primer parámetro (`context.Background()`) en Go devuelve un contexto vacío. Se utiliza el paquete `context` en Go para propagar señales de cancelación, *deadlines* y otros valores asociados a un contexto de ejecución (normalmente dentro de una misma sesión o *request*).

```go
sendResponse, err := client.AppSend(ctx, &pb.AppSendRequest{
  ChannelId: channelID,
  Recipient: "Pong",
  Message: &pb.AppMessage{
    Body: []byte("hello"),
    Type: "ping",
  },
})
if err != nil || sendResponse.Result != pb.AppSendResponse_RESULT_OK {
  t.Errorf("Failed to AppSend")
}

recvResponse, err := client.AppRecv(ctx, &pb.AppRecvRequest{
  ChannelId:   channelID,
  Participant: "Pong",
})
```

Fragmento de código 11: Envío y recepción de mensajes en el Service Client.

3.4. Establecimiento de un canal

Una vez que ya hay *Service Providers* registrados en el *Service Repository* (que en esta tesis es una componente interna del *Service Broker*), un *Service Client* puede utilizarlos enviando un requerimiento al Broker.

La interfaz que el *Service Broker* expone tiene solo dos RPCs (Fragmento de código 12): además de la RPC para registrar un proveedor (Sección 3.2), cuenta con otra RPC para recibir el requerimiento de un canal por parte de *Service Clients* (`BrokerChannel`). El formato del mensaje que espera es idéntico al que el *middleware* espera de parte del *Service Client*: el contrato global del canal y un mapa de participantes predeterminados con sus URIs. En este caso, siempre se espera que al menos haya un participante en ese mapa, que es el propio *Service Client* que está solicitando el canal (este dato lo agrega el *middleware*).

```protobuf
service BrokerService {
    rpc BrokerChannel (BrokerChannelRequest) returns (BrokerChannelResponse) {}
    rpc RegisterProvider (RegisterProviderRequest) returns (RegisterProviderResponse) {}
}
message BrokerChannelRequest {
    GlobalContract contract = 1;
    map<string, RemoteParticipant> preset_participants = 2;
}
```

Fragmento de código 12: Interfaz del Service Broker.

La Figura 13 es un diagrama de secuencia que ilustra cómo funciona el establecimiento de un canal para el caso más sencillo (dos participantes). En el mismo se asume que el *Service Provider* ya se encuentra registrado ante el *Service Broker*. Al comenzar, el *Service Client* envía a su *middleware* el contrato global del canal (1). El *middleware*, a su vez, genera un identificador único para el canal y se lo envía como respuesta al *Service Client* (2).

El *middleware* posterga el proceso de *brokerage* hasta que el *Service Client* haga uso efectivo del canal a través de algún `AppSend` o `AppRecv`. Esto es una decisión de diseño que busca evitar el *brokerage* innecesario del canal en caso de que el *Service Client* no haga uso de él. Se consideró la posibilidad de agregar un parámetro opcional a `RegisterChannelRequest` para indicar que el *brokerage* debe ser iniciado inmediatamente, pero se dejó fuera de alcance de esta tesis. Esto podría ayudar en casos de uso donde el *Service Client* sabe que siempre necesitará utilizar el canal, o también en casos en los que se desea minimizar la latencia en el primer uso del canal. Otra variante posible que se evaluó y dejó fuera de alcance es que el *middleware* realice *brokerage* parcial del canal, obteniendo cada *Service Provider* del *Broker* a medida que el uso del

canal requiera de cada participante (dependiendo de cómo es la coreografía del canal, no todas las trazas necesitan de la participación de todos los *Service Providers*). El *brokerage parcial* introduciría mayor complejidad al diseño del protocolo de establecimiento de un canal, ya que, según la coreografía global del canal, el participante que toma la decisión de interactuar con algún *Service Provider* que no haya aún participado en la comunicación la puede tomar cualquier participante, no solo el *Service Client*. Y, por lo tanto, para permitir *brokerage parcial*, deberíamos admitir que los *middlewares* de los *Service Providers* puedan solicitar el *brokerage* de nuevos participantes (ya sea enviando un mensaje directamente al broker o a través del *middleware* del *Service Client*).[17]

[17]Para el *Service Broker* es importante saber a quién atribuir cada *brokerage*, ya que, en un contexto comercial, se podría cobrar por el servicio de *brokerage*. En caso de que se admitiera delegar en otros participantes del canal la solicitud de *brokerage* de algún participante aún no conectado al canal, habría que diseñar algún mecanismo de delegación de confianza (por ejemplo, firmando un permiso con cierto vencimiento).

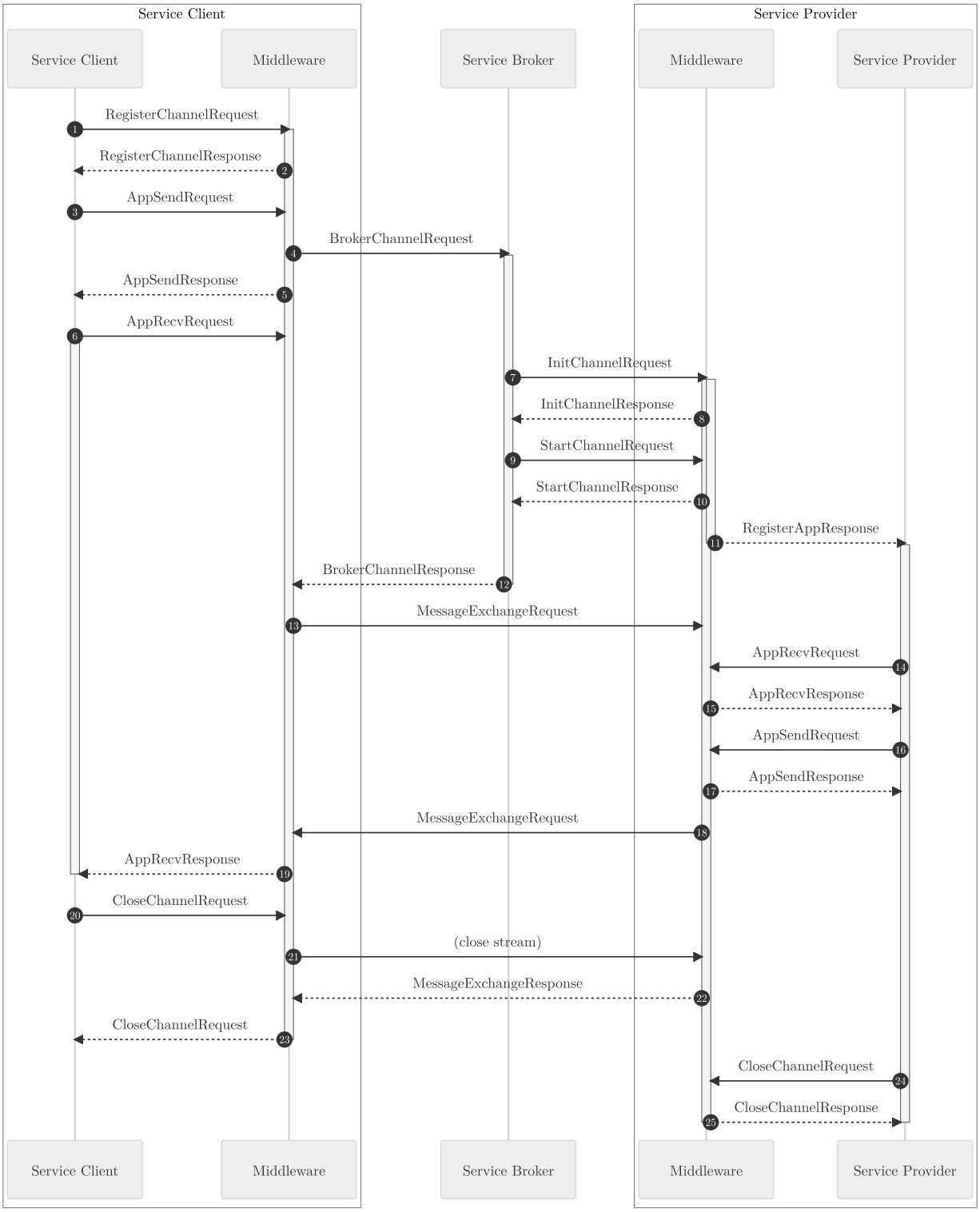


Figura 13: Diagrama de secuencia del brokering.

En la Figura 13 el primer uso del canal por parte del *Service Client* es un envío (`AppSend`) (3). En ese caso, el *middleware* encola el mensaje saliente e inmediatamente responde un ACK al *Service Client* (5). Concurrentemente, el *middleware*, al haber detectado la primera interacción sobre el canal declarado, inicia el proceso de *brokerage*, enviando el contrato global al *Service Broker* (4). Si, en cambio, el *Service Client* intentara recibir un mensaje con `AppRecv` como primera acción luego de registrar el canal, en ese caso la llamada se bloquearía mientras el *middleware*

obtiene un proveedor apropiado del *Broker* y efectivamente recibe el mensaje esperado de parte del otro participante.

El *broker*, al recibir el contrato global, proyecta el contrato local correspondiente para cada participante (excepto aquellos que hayan sido excluídos) y calcula, para cada uno, cuáles proveedores del *Service Registry* son compatibles. En el caso de los contratos globales como sistemas de CFSMs, la proyección es simplemente la CFSM correspondiente. En el ejemplo de la Figura 13, solo hay un participante a resolver. Para esto, el *broker* cuenta con una base de datos de resultados precalculados. Sin embargo, si el contrato de requerimiento es nuevo para el *broker*, debe realizar el cómputo en el momento. Profundizaremos en algunos detalles sobre esto en la Sección 4.3.

Una vez seleccionado el mejor candidato proveedor para cada participante del contrato global, el *broker* realiza dos rondas sucesivas en las que envía un mensaje a cada *Service Provider*. En la primera ronda, envía un mensaje de tipo `InitChannelRequest` (7) (Fragmento de código 13). En este mensaje se indica al *middleware* de cada proveedor que se está dando inicio a un canal que involucra al *Service Provider* que ese middleware representa, y se indican las URIs de los demás participantes del canal. Al mismo tiempo, este mensaje le permite al *broker* verificar que el proveedor se encuentre efectivamente en línea.[18] A partir de la recepción de este mensaje, los *middlewares* deberán aceptar mensajes entrantes para este canal y encolarlos para su eventual recepción por parte del *Service Provider*[19].

```
message InitChannelRequest {
    string channel_id = 1;
    string app_id = 2; // Para identificar al Service Provider
    map<string, RemoteParticipant> participants = 3;
}
```

Fragmento de código 13: Primer mensaje que recibe el middleware de cada Service Provider durante el proceso de brokerage.

Si todos los *Service Providers* responden satisfactoriamente al envío del mensaje `InitChannelRequest` de la primera ronda, el *Service Broker* les envía un segundo mensaje para indicar que se debe comenzar el envío de mensajes en el canal (según el contrato correspondiente). Este mensaje es de tipo `StartChannelRequest`.

El *middleware* del *Service Provider*, luego de recibir ambos mensajes de inicialización del canal, envía a su *Service Provider* un mensaje de tipo `RegisterAppResponse` (11) que contiene un mensaje de tipo `InitChannelNotification` con el UUID del nuevo canal. El *Service Provider* debe, entonces, comenzar a enviar y recibir mensajes en este canal de acuerdo a su contrato.

Los *middlewares* exponen las siguientes RPCs en su interfaz pública (Fragmento de código 14):

[18] En el diseño inicial se consideró que si el envío de este mensaje falla, el *broker* puede marcar a ese proveedor como fuera de línea y reintentar con algún otro proveedor compatible del *Service Registry*. Dicha funcionalidad no se implementó en el contexto de esta tesis, pero las distintas funciones auxiliares del código fuente del *broker* y la estructura del código contemplan el desarrollo de esta funcionalidad en el futuro. En el código se pueden encontrar referencias a esto en el `chan string unresponsiveParticipants` y el parámetro `denylistedProviders` de la función `getBestCandidates`.

[19] Si así no lo hicieran y esperaran a recibir el segundo mensaje, podrían darse condiciones de carrera en las que un *Service Provider* ya recibió ambos mensajes de inicialización del canal, envía un mensaje a otro participante, y éste no lo acepta por no haber aún recibido el segundo mensaje de inicalización del canal.

```
service PublicMiddlewareService {
  rpc InitChannel(InitChannelRequest) returns(InitChannelResponse);
  rpc StartChannel(StartChannelRequest) returns(StartChannelResponse);
  rpc MessageExchange (stream MessageExchangeRequest) returns (MessageExchangeResponse);
}

message MessageExchangeRequest {
    string channel_id = 1;
    string sender_id = 2;
    string recipient_id = 3;
    AppMessage content = 4;
}
message AppMessage {
    string type = 1;
    bytes body = 2;
}
```

Fragmento de código 14: Interfaz pública del Middleware.

Una vez inicializado un canal, los *middlewares* establecen un stream unidireccional con los demás *middlewares* para enviar mensajes. En el ejemplo de la Figura 13, el *middleware* del Service Client establece un stream con el otro *middleware* para enviar el primer mensaje (13). A su vez, el *middleware* del Service Provider establece un stream en el sentido opuesto para enviar la respuesta (18).

Finalmente, el *Service Client* decide cerrar el canal, y para eso envía un mensaje de tipo `CloseChannelRequest` (20) a su *middleware*. Éste, a su vez, cierra el stream que tiene abierto con el otro *middleware*.

3.5. Abstracción sobre los contratos

En este trabajo implementamos solamente un modelo de contratos: las *Communicating Finite State Machines*. Para los contratos de provisión utilizamos una sola CFSM, mientras que para los contratos de requerimiento, utilizamos un *sistema* de CFSMs; es decir, dos o más CFSMs que se comunican entre sí.

Se exploró la posibilidad de dar soporte, también, para describir los contratos de requerimiento utilizando *Global Graphs*. La ventaja de utilizar *Global Graphs* para describir un canal de comunicación es que garantiza que los contratos locales que se obtienen al proyectar un *Global Graph* cumplen una condición llamada *generalised multiparty compatibility* (GMC). Dicha propiedad establece la compatibilidad entre los contratos locales de cada participante. Se analizó la posibilidad de incorporar al *middleware* la herramienta `gc2fsa`[20] de ChorGram[21], desarrollada por Lange, Tuosto y Yoshida [15]. Sin embargo, se descartó esta opción porque `gc2fsa` agrega nuevos mensajes (de tipos nuevos, no existentes en el *Global Graph*) a los contratos locales cada vez que un participante toma la decisión de salir de un ciclo o de continuar en una nueva iteración del mismo. Esto se hace para asegurar que la decisión sobre qué rama se toma en el grafo la tome un único participante. Dado que es necesario que quien implemente un *Service Client* que utilice dicho contrato de requerimiento envíe explícitamente los mensajes de control que `gc2fsa` agrega, no es razonable admitir contratos de requerimiento en formato *Global Graph*.

No deja de ser importante destacar que Chorgram es una herramienta súmamente útil a la hora de diseñar un contrato de requerimiento, ya que permite diseñar protocolos correctos sin deadlocks [10]. Esperamos que quienes implementen *Service Clients* diseñen los protocolos de

[20]Como `gc2fsa` está escrita en Haskell, y uno de los atributos deseables del *middleware* es la portabilidad, se compiló `gc2fsa` a WASI (WebAssembly) para así poder obtener un ejecutable portable.

[21]https://bitbucket.org/eMgssi/stable_chorgram/

aplicación utilizando *Global Graphs* y luego obtengan la proyección a CFSM (utilizando ChorGram) para implementar el canal de comunicación. Un posible desarrollo futuro relacionado es un generador de código que, dada una CFSM, genere el esqueleto de código del *Service Client/ Provider* que implemente dicho contrato.

Es importante también señalar que GMC garantiza interoperabilidad a nivel del protocolo de aplicación, pero no provee ninguna garantía con respecto a requisitos funcionales (pre y post-condiciones), ni tampoco con respecto a requisitos no funcionales (restricciones de tiempo, consumo de recursos, costo, etc.). Para resolver esto, se puede en un futuro extender la noción de contrato para incorporar también estos aspectos, por ejemplo aspectos de calidad de servicio (QoS) [18,19].

Si bien, entonces, en el marco de este trabajo solo se utilizan CFSMs para representar contratos, se implementaron interfaces para contratos *globales* (de requerimiento) y contratos *locales* (de provisión), de manera tal de permitir la extensión futura de SEArch a otros tipos de contratos de interoperabilidad, como por ejemplo *Session types*[22] [11].

- Métodos compartidos por ambos tipos de contratos:
  - `GetContractID() string`: devuelve un identificador único para el contrato. Dos contratos iguales deberían devolver el mismo identificador. La igualdad la tomamos a nivel semántico. Podría pasar que dos contratos que devuelven el mismo identificador tengan distinta representación en bytes (`GetBytesRepr()`), pero no puede pasar que dos contratos que tienen la misma representación en bytes devuelvan distintos identificadores. Si dos contratos devuelven los mismos valores para todos los métodos salvo `GetBytesRepr()`, sería deseable que devuelvan el mismo identificador.
  - `GetRemoteParticipantNames() []string`: devuelve la lista de nombres de los participantes «remotos» del contrato; es decir, de todos los participantes que no son el *Service Client* o el *Service Provider* cuyo contrato estamos describiendo. Si bien los contratos *globales* en abstracto describen a todo el canal, en esta representación de contratos siempre hay un participante distinguido (el *Service Client*, que es quien es «dueño» o «iniciador» del canal).
  - `GetBytesRepr() []byte`: devuelve la secuencia de *bytes* que representa al contrato. Esto es lo que utilizamos como serialización para la transmisión de contratos a través de la red.
- Métodos específicos de los contratos globales:
  - `GetParticipants() []string`: Devuelve lo mismo que `GetRemoteParticipantNames()` pero también devuelve el nombre del *Service Client*.
  - `GetLocalParticipantName() string`: Devuelve el nombre de participante del *Service Client*.
  - `GetProjection(string) LocalContract`: Devuelve la proyección del contrato global sobre el nombre de participante recibido por parámetro.

Para la implementación concreta de contratos locales y globales como CFSMs y sistemas de CFSMs, respectivamente, se realizó un *fork* de la biblioteca `cfsm` de Go de Nicholas Ng [20], utilizada en [14].

3.6. Verificación de compatibilidad entre contratos

Al momento de iniciar esta tesis nos concentramos en un enfoque que se apoyaba sobre la idea de que existe una ontología preexistente y universal de mensajes. Es decir, asumimos que

[22] El código correspondiente se encuentra en el archivo `contract/contract.go`.

cuando en un contrato de requerimiento o provisión se hace referencia a un tipo de mensaje en particular (por ejemplo «Ping»), su representación como tipo de datos básico o complejo es conocida y compartida por todos los involucrados. Y por lo tanto, la noción de compatibilidad se apoya sobre la igualdad de los nombres de mensajes, porque esos mensajes estánt tomados de esa ontología que debe ser utilizada por todos los participantes involucrados en un mismo canal.

En una tesis reciente [24] se estudió un algoritmo de bisimilaridad para CFSMs que no requiere de la existencia de esa ontología sino que, al momento de chequear bisimilaridad computa el matching de nombres que permite igualar el lenguaje de un participante con el lenguaje de otro.

El Service Broker implementado en el marco de esta tesis espera una función de chequeo de compatibilidad del siguiente tipo (Fragmento de código 15)[23]:

```
type ContractCompatibilityFunc func(
  ctx context.Context,
  req contract.LocalContract,
  prov contract.LocalContract)
  (bool, map[string]string, error)
```

Fragmento de código 15: Signatura de la función de chequeo de compatibilidad utilizada

Como mencionamos anteriormente, en el marco de esta tesis no se desarrolló ningún algoritmo para verificación de compatibilidad entre contratos. Para poder utilizar la infraestructura desarrollada, basta con escribir dicho algoritmo respetando la signatura de `ContractCompatibilityFunc`. Los valores de retorno representan (en orden):

- si los dos contratos recibidos como entrada son compatibles entre sí (tipo `bool`),
- un diccionario con la traducción de nombres de participantes entre los contratos, donde las claves son los nombres de los participantes según el contrato de provisión (`prov`) y los valores son los nombres correspondientes según el contrato de requerimiento (`prov`)
- un error (en caso de haberlo, si no, es `nil`).

El los distintos tests de integración lo que se hizo fue escribir una función de compatibilidad *ad-hoc* para cada test, ya que se conocen los contratos a priori en ese caso.

Para poder implementar el algoritmo de [24] que también contempla traducciones entre nombres de tipos de mensajes, necesitaríamos modificar la signatura de la función de compatibilidad de manera tal que devuelva un segundo diccionario con la traducción de nombres de tipos de mensajes. Asimismo, también la base de datos de resultados de chequeo de compatibilidad también debería ser modificada para almacenar esta nueva traducción (ver Sección 4.3).

3.7. Limitaciones

3.7.1. Uso de un único stream bidireccional para intercambio de mensajes entre middlewares

En vez de establecer un stream unidireccional (`MessageExchange`) entre *middlewares*, se podría utilizar un único stream bidireccional. Esto ahorraría establecer una conexión gRPC adicional, lo cual, utilizando cifrado TLS del canal, puede tomar al menos tres *round-trips*. En el contexto de esta tesis se decidió por una implementación más sencilla utilizando un stream unidireccional en cada sentido. Sin embargo, es posible mejorar este diseño sin modificar las interfaces provistas a los *Service Clients* y *Service Providers*.

[23] `https://github.com/pmontepagano/search/blob/519f1756b92f35667e8c457a02c3a77a09951847/internal/broker/broker.go#L64`

3.7.2. Cifrado del canal

En todos los tests desarrollados en el marco de este trabajo, no se utilizó cifrado del canal para la comunicación entre las distintas componentes. Sin embargo, gRPC, al funcionar sobre HTTP/2, provee soporte de cifrado TLS. Nuestra implementación del *Service Broker* y del *middleware* ya contempla parámetros de invocación que permiten al usuario indicar la clave x509 pública y privada a utilizar. Sin embargo, ninguna de las conexiones que se inician desde estas componentes utiliza cifrado, y en un contexto productivo deberían ser modificadas[24].

Cabe destacar que, si bien el *Service Broker* y los *middlewares* de *Service Providers* tendrían URLs conocidas para las cuales se pueden obtener certificados TLS válidos de la WebPKI, no sucede lo mismo para los *Service Clients*. Este problema puede ser resuelto si se modifica el diseño de manera tal que las conexiones entre el *middleware* del *Service Client* y otros *middlewares* sean siempre iniciadas por el primero.

3.7.3. Registro de Service Clients

Es esperable que un servicio como el que provee el *Service Broker* no sea gratuito, ya que el cómputo de compatibilidad entre contratos es costoso. En un contexto comercial, sería razonable que para poder utilizar el servicio del *Broker* los *Service Clients* deban primero registrarse, aceptar términos y condiciones del servicio, etc. Es habitual que este tipo de servicio se cobre por cantidad de invocaciones. Por lo tanto, el *Broker* necesita saber qué cliente (comercial) es el que invoca las llamadas a la RPC `BrokerChannel`. Este aspecto no tiene nada novedoso y es muy habitual que para utilizar una API, quién la invoca envíe alguna clave para identificarse (*API key*). Este aspecto se dejó intencionalmente fuera del alcance del presente trabajo.

3.7.4. Registro *offline* de Service Providers

El diseño actual del proceso de registro de *Service Providers* presenta algunos déficits: en primer lugar, es razonable esperar que los *Service Providers* tengan *uptime* medido en días o semanas, y por lo tanto es probable que el requisito de mantener una conexión abierta entre *middleware* y *Service Provider* durante todo el tiempo de vida del servicio cause problemas de estabilidad, ya que es común que las conexiones TCP se interrumpan ante inestabilidades en la red, especialmente si ambos servicios no se encuentran dentro de la misma red local. En segundo lugar, al atar las notificaciones de inicio de un nuevo canal a la recepción del mensaje `InitChannelNotification`, esto implica que quien implemente un *Service Provider* debe escribir cierta lógica para tener un proceso que constantemente intente recibir nuevos mensajes del *stream* para disparar un nuevo proceso que atienda al nuevo canal. Por último, este diseño complica el escalamiento horizontal de *Service Providers*, ya que si quisiéramos correr más de una instancia del *Service Provider*, con este esquema de registro terminaríamos registrando N veces al mismo *Service Provider* ante el *Broker*.

Para resolver estos problemas, se esbozó un rediseño de este aspecto de la arquitectura pero no se llegó a implementar[25]. Las ideas principales de este rediseño son:

- Realizar el registro de los *Service Providers* de manera *offline*. Es decir, el administrador de un *Service Provider* deberá invocar manualmente la RPC `RegisterProvider` del Service Broker. En un contexto comercial/productivo este paso podría realizarse también a través de una interfaz web con un formulario.

[24] Éstas se encuentran marcadas con comentarios en el código fuente que indican `TODO: use tls`.

[25] Los detalles se pueden encontrar en GitHub en `https://github.com/pmontepagano/search/issues/4`.

- Eliminar la RPC `RegisterApp` del *middleware*. En su lugar, para que un *middleware* preste servicio a un *Service Provider*, se deberá hacer a través de un archivo de configuración del *middleware* donde se indique el contrato del proveedor y la(s) URL(s) donde el *middleware* puede hallarlo. Puede haber más de una URL de manera tal que el *middleware* pueda distribuir la carga entre distintas instancias del mismo *Service Provider* (y tener redundancia en caso de que alguna de ellas no esté disponible).
- Crear una nueva definición de servicio utilizando Protocol Buffers y gRPC que será lo que deban implementar los *Service Providers*. Esto simplificaría el desarrollo de *Service Providers*, ya que bastaría con que implementen un *handler* para esta RPC. Este handler es invocado por el *middleware* cada vez que se inicia un nuevo canal con este participante. Dentro del cuerpo del *handler* el *Service Provider* usará el canal tal como lo hace en el diseño actual.

3.7.5. Escalamiento horizontal del *middleware*

Otro aspecto que puede representar una limitación para el diseño actual es que no está contemplado ningún mecanismo de balanceo de carga para los *middlewares*. Es decir, el *middleware* del Service Provider se vuelve un «single point of failure». En principio esto debería poder ser resuelto sin cambios en la arquitectura utilizando un *proxy load balancer*[26].

[26] `https://grpc.io/blog/grpc-load-balancing/`

4. Implementación

En este capítulo se abordan distintos aspectos puntuales de la implementación. Todo el código desarrollado se puede encontrar en el repositorio del proyecto[27].

4.1. Elección del protocolo de comunicación y del lenguaje de programación

Uno de los objetivos es ofrecer una abstracción que facilite integrar sistemas de *software* heterogéneos con ciertas garantías de interoperabilidad. Las interacciones entre los distintos sistemas de *software* en SEArch están mediadas por el *middleware*. Por lo tanto, necesitamos seleccionar un protocolo de comunicación entre el *middleware* y los *Service Clients* y *Service Providers* que tenga soporte en una amplia variedad de lenguajes de programación ya que queremos proporcionar bibliotecas (SDKs) para facilitar la escritura o adaptación de software para el ecosistema de SEArch.

La primera opción que se evaluó fue utilizar HTTP como protocolo de comunicación, ya que hoy en día es el protocolo más ampliamente utilizado y conocido. Sin embargo, HTTP presenta algunas limitaciones: hasta HTTP/2 solo se admite un esquema de comunicación *request-response* y no tiene soporte nativo para definir mensajes tipados (*schemas*).

Se decidió utilizar Protocol Buffers [16] para definir el esquema de los mensajes y gRPC como protocolo de comunicación. gRPC es un framework RPC de alto rendimiento que permite utilizar Protocol Buffers como lenguaje de descripción de interfaces. Protocol Buffers es un formato de serialización de paquetes de datos tipado y estructurado. Existen compiladores que interpretan las definiciones de mensajes y servicios de archivos `.proto`, y generan código en una amplia variedad de lenguajes de programación para manipular los mensajes con clases y tipos nativos, e invocar o implementar servicios gRPC. De esta manera, con definir en un único lenguaje (Protocol Buffers) los tipos de mensajes y servicios necesarios para interactuar con SEArch, podemos generar código interoperable para una amplia variedad de lenguajes[28]. Una posible desventaja de gRPC es que no es apto para ser utilizado desde un navegador web (ya que desde éstos solo es posible realizar requests HTTP). Dicha limitación puede ser resuelta fácilmente adoptando el protocolo Connect[29], que está definido sobre HTTP/1.1.

Con respecto a la elección de lenguaje de programación para implementar la infraestructura (*middleware* y *broker*), se seleccionó Go[30][6] porque es un lenguaje diseñado con foco en *systems programming*, es decir, para desarrollar *software* que provee servicio a otros sistemas de *software* y no a usuarios finales. Go tiene soporte nativo para concurrencia (*goroutines*) y comunicación entre procesos (*channels*). Además, Go tiene buen soporte para gRPC, que es el protocolo de comunicación que se eligió para la comunicación entre las distintas componentes de SEArch. Otra ventaja de Go frente a otros lenguajes es la facilidad para compilar binarios

[27] `https://github.com/pmontepagano/search`

[28] En Julio de 2023 hay soporte oficial para C# / .NET, C++, Dart, Go, Java, Kotlin, Node, Objective-C, PHP, Python y Ruby, pero también existen *plugins* para muchos otros lenguajes de programación. `https://grpc.io/docs/languages/`

[29] `https://connect.build/docs/introduction`

[30] `https://go.dev/`

que funcionen en una multiplicidad de sistemas operativos y arquitecturas de procesador[31]. Esto es particularmente útil para el *middleware*, ya que es la pieza de software de SEArch que se espera sea adoptada por quienes implementen y operen *Service Clients* y *Service Providers*. Con la simple modificación de los parámetros `GOOS` y `GOARCH` del compilador, podemos generar un binario compilado estáticamente que funcione en una enorme variedad de sistemas operativos y arquitecturas de procesador.

Otro motivo que influyó en la elección de Go es que, al ser un lenguaje sencillo y muy popular, facilita la adopción de la tecnología por parte de terceros y las contribuciones de la comunidad *open-source*. Además, se trata de un lenguaje con fuertes garantías de compatibilidad hacia atrás, lo que debería permitir que el código desarollado continúe funcionando por muchos años sin modificaciones a medida que se publiquen nuevas versiones del compilador[32].

4.2. Organización del repositorio de código fuente

Dentro del repositorio de código de SEArch[33] el código se organiza de la siguiente manera:

- `proto`: Contiene los archivos Protocol Buffer con las definiciones de los tipos de mensajes y servicios gRPC. Estos archivos se compilan utilizando la herramienta `buf`[34]. El archivo de configuración que determina los lenguajes destino de compilación es `buf.gen.yaml` (en la raíz del repositorio).
- `gen`: Aquí se guardan los archivos generados por `buf` al compilar los `.proto`. El *middleware* y el *broker* utilizan el código generado en Go, pero almacenamos también aquí el código generado en otros lenguajes.
- `cfsm`: Contiene el *fork* de la biblioteca `nickng/cfsm` que implementa Communicating Finite State Machines. Los principales cambios con respecto a la original son: el agregado de un parser para leer archivos FSA, unos cambios de estructuras de datos para forzar a que la serialización de las CFSMs sea determinística (se utilizaban estructuras sin orden definido), y el agregado del uso de nombres de CFSMs en vez de índices numéricos para hacer más legibles los archivos FSA generados.
- `cmd`: Contiene el código para los ejecutables del *middleware* y el *broker*.
- `contract`: Aquí están definidas las interfaces de contrato global, contrato local y las impementaciones concretas de ambos con CFSMs (Sección 3.5).
- `ent`: Contiene el código que gestiona la base de datos del *broker* (ORM). Las definiciones de las entidades y relaciones se encuentran en el subdirectorio `schema`. El resto del código es generado a partir de éstas utilizando la herramienta `entgo`[35].
- `mocks`: Mocks de los contratos, para facilitar el testing del *broker*. Generados con la herramienta `mockery`[36].
- `internal`: Aquí se encuentra el código del *middleware* y del *broker*, junto con sus tests.

[31] El *target* de compilación de Go se controla a través de las variables de entorno `GOOS` (sistema operativo) y `GOARCH` (arquitectura). Al momento de la elaboración de esta tesis, Go 1.21 soporta catorce valores distintos para la variable `GOOS` y doce valores distintos para la variable `GOARCH`. No todas las combinaciones de ambas variables son válidas. Fuente: `https://go.dev/doc/install/source#environment`

[32] `https://go.dev/blog/compat`

[33] `https://github.com/pmontepagano/search`

[34] `https://github.com/bufbuild/buf`

[35] `https://entgo.io/`

[36] `https://github.com/vektra/mockery`

4.3. Cálculo y almacenamiento de resultados de compatibilidad

El algoritmo de chequeo de compatibilidad entre CFSMs es costoso. Por lo tanto, no es esperable que el *Service Broker* ejecute ese algoritmo ante cada solicitud. Además, a medida que la cantidad de proveedores del *Service Registry* aumenta, mayor será la cantidad de contratos a chequear.

Por ello, se implementó una base de datos del *Service Broker* que almacena los distintos contratos y los resultados del chequeo de compatibilidad entre ellos. En la Figura 14 se puede ver el diagrama de entidades y relaciones desarrollado.

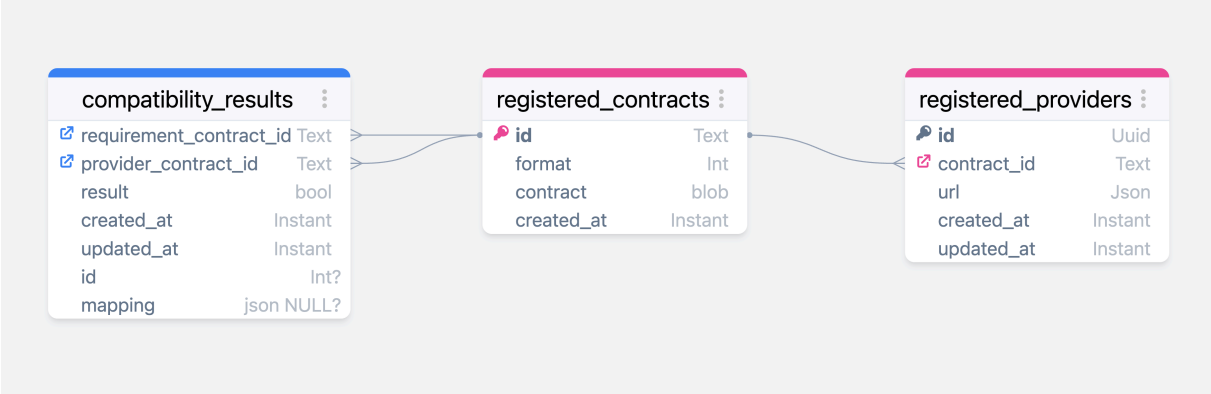


Figura 14: Diagrama de entidades y relaciones de la base de datos del Service Broker.

Cada vez que el *Service Broker* recibe un contrato de provisión o de requerimiento, lo guarda en la base de datos (si no existe ya en la misma). Todos los contratos almacenados son contratos *locales*, ya que la función que chequea la compatibilidad entre contratos lo hace siempre entre dos contratos *locales*. Al recibir un contrato global de requerimiento, el *Service Broker* almacena cada una de las proyecciones.

Al recibir un contrato de requerimiento a través de la RPC `BrokerChannel`, el *Service Broker* invoca la función `getBestCandidates`, que, a su vez, invoca concurrentemente a la función `getBestCandidate` para cada uno de los participantes del contrato global (excepto el *Service Client*). Esta última función primero consulta la tabla de resultados de compatibilidad para encontrar candidatos para los que ya se haya determinado que son compatibles con el contrato de requerimiento. En caso de hallar alguno, se devuelve uno de ellos como resultado[37]. Queda fuera del alcance de esta tesis la elaboración de un ranking entre distintos proveedores de manera tal que permita al *Service Broker* seleccionar, de entre todos los *Service Providers* compatibles, al «mejor» candidato según el ranking.

Si bien se puso especial cuidado en el manejo adecuado de las *goroutines* que ejecutan concurrentemente los chequeos de compatibilidad[38], el diseño implementado solo escala *verticalmente*. Es decir, se aprovecha al máximo un servidor multi-core y multi CPU, pero no se desarrolló soporte para escalar horizontalmente el cómputo de compatibilidad entre contratos. Esta mejora queda fuera del alcance de la tesis, ya que no presenta ningún desafío interesante. Es habitual lidiar con este tipo de problemas con una cola de trabajos y una granja de servidores que los procesa.

[37]En este caso, además, si se detecta que hay contratos en la base de datos contra los cuales no se verificó compatibilidad, se encolan trabajos para realizar ese cálculo y dejar el resultado almacenado.

[38]Se utilizó la biblioteca `conc` para prevenir *leaks* de *goroutines* (rutinas que quedan ejecutando infinitamente). `https://github.com/sourcegraph/conc`. Además, se utilizó la biblioteca `goleak` para detectar leaks de *goroutines* en los tests. `https://github.com/uber-go/goleak`.

4.4. Conexión entre middlewares

Hay una traducción muy directa entre las CFSMs y su implementación en Go, que resultó muy idiomática. Para representar los *buffers* de mensajes entrantes y salientes de cada CFSM utilizamos *channels* de Go. Los *channels* son primitivas del lenguaje similares a los *pipes* de Unix. Básicamente son colas *thread-safe*. Esto nos permitió utilizar un *channel* para cada *buffer* entrante y saliente (uno para cada participante del canal). Una rutina llamada *sender* es la encargada de procesar cada mensaje encolado en los *buffers* salientes y enviarlos al participante remoto. Por otro lado, los handlers de `MessageExchange` y `AppRecv` encolan y desencolan mensajes de los buffers entrantes, respectivamente.

Por otro lado, para respetar el orden de los mensajes en tránsito, se utilizó una RPC de tipo *streaming* (`MessageExchange`), que está garantizado que preserva el orden.

4.5. Tests de integración

El código desarrollado contiene distintos tests de integración (en los archivos cuyo nombre termina con `_test.go`). Se obtuvo un *coverage* del 82% al momento de la entrega de este documento. La medición del mismo se realizó utilizando la herramienta oficial de Go, que mide cobertura de líneas ejecutadas del código fuente (no es exatamente *statement coverage*). La herramienta reescribe el código fuente del programa bajo test antes de compilarlo, e introduce un contador en cada bloque de código. Esto provee un mecanismo barato para medir coverage con un *overhead* de 3% [22]. El archivo `README.md` del repositorio de código fuente contiene instrucciones para obtener un reporte de *coverage*.

Los tests de integración que cubren a todas las componentes desarrolladas se encuentran en el archivo `internal/middleware/middleware_test.go`. Por ejemplo, el test llamado `TestPingPongFullExample` realiza las siguientes acciones:

1. Inicia un Broker.
2. Configura una función de chequeo de compatibilidad de contratos para el broker. Esta función es un *mock*, en el sentido de que devuelve *true* solo para los contratos específicos que se utilizan en el test que sabemos que son compatibles entre sí.
3. Inicia dos instancias del *middleware*, uno para el Service Provider «Pong» y uno para el Service Client «Ping».
4. Ejecuta funciones de test para tomar el rol de los programas «Pong» y «Ping». Este último envía un mensaje al primero, luego espera la respuesta, y luego cierra la sesión enviando otro mensaje del cual también espera la respuesta, de manera similar al ejemplo descripto en la Sección 3.3. Estas funciones contienen distintas aserciones para verificar el comportamiento esperado.

5. Caso de estudio

En este capítulo realizaremos una explicación detallada de cómo implementar *Service Clients* y *Service Providers* a través de un ejemplo en el que intervienen tres participantes. El objetivo es que el lector comprenda cuál es la experiencia de usuario de quienes utilicen esta infraestructura para desarrollar software.

El ejemplo que utilizaremos es el de un flujo de pago con tarjeta de crédito *online*. Tendremos tres participantes: una aplicación cliente desarrollada en Java, un servidor de *backend* desarrollado en Python y un servicio de procesamiento de pagos de tarjeta de crédito desarrollado en Go.

El código fuente de todo este ejemplo se puede encontrar en el subdirectorio `examples/credit-card-payments` del repositorio. La Figura 15 muestra las CFSMs para los tres servicios.

El contrato global del canal (a partir del cual se generó la representación gráfica de la Figura 15) puede leerse en el Fragmento de código 16, el cual también se encuentra en el repositorio de código[39].

```
.outputs ClientApp
.state graph
q0 Srv ! PurchaseRequest q1
q1 Srv ? TotalAmount q2
q2 PPS ! CardDetailsWithTotalAmount q3
q3 PPS ? PaymentNonce q4
q4 Srv ! PurchaseWithPaymentNonce q5
q5 Srv ? PurchaseOK q6
q5 Srv ? PurchaseFail q7
.marking q0
.end

.outputs Srv
.state graph
q0 ClientApp ? PurchaseRequest q1
q1 ClientApp ! TotalAmount q2
q2 ClientApp ? PurchaseWithPaymentNonce q3
q3 PPS ! RequestChargeWithNonce q4
q4 PPS ? ChargeOK q5
q4 PPS ? ChargeFail q6
q5 ClientApp ! PurchaseOK q7
q6 ClientApp ! PurchaseFail q8
.marking q0
.end

.outputs PPS
.state graph
q0 ClientApp ? CardDetailsWithTotalAmount q1
q1 ClientApp ! PaymentNonce q2
q2 Srv ? RequestChargeWithNonce q3
q3 Srv ! ChargeOK q4
q3 Srv ! ChargeFail q5
.marking q0
.end
```

Fragmento de código 16: Contrato global del canal del caso de estudio en formato FSA.

[39] `https://github.com/pmontepagano/search/blob/main/examples/credit-card-payments/contract.fsa`

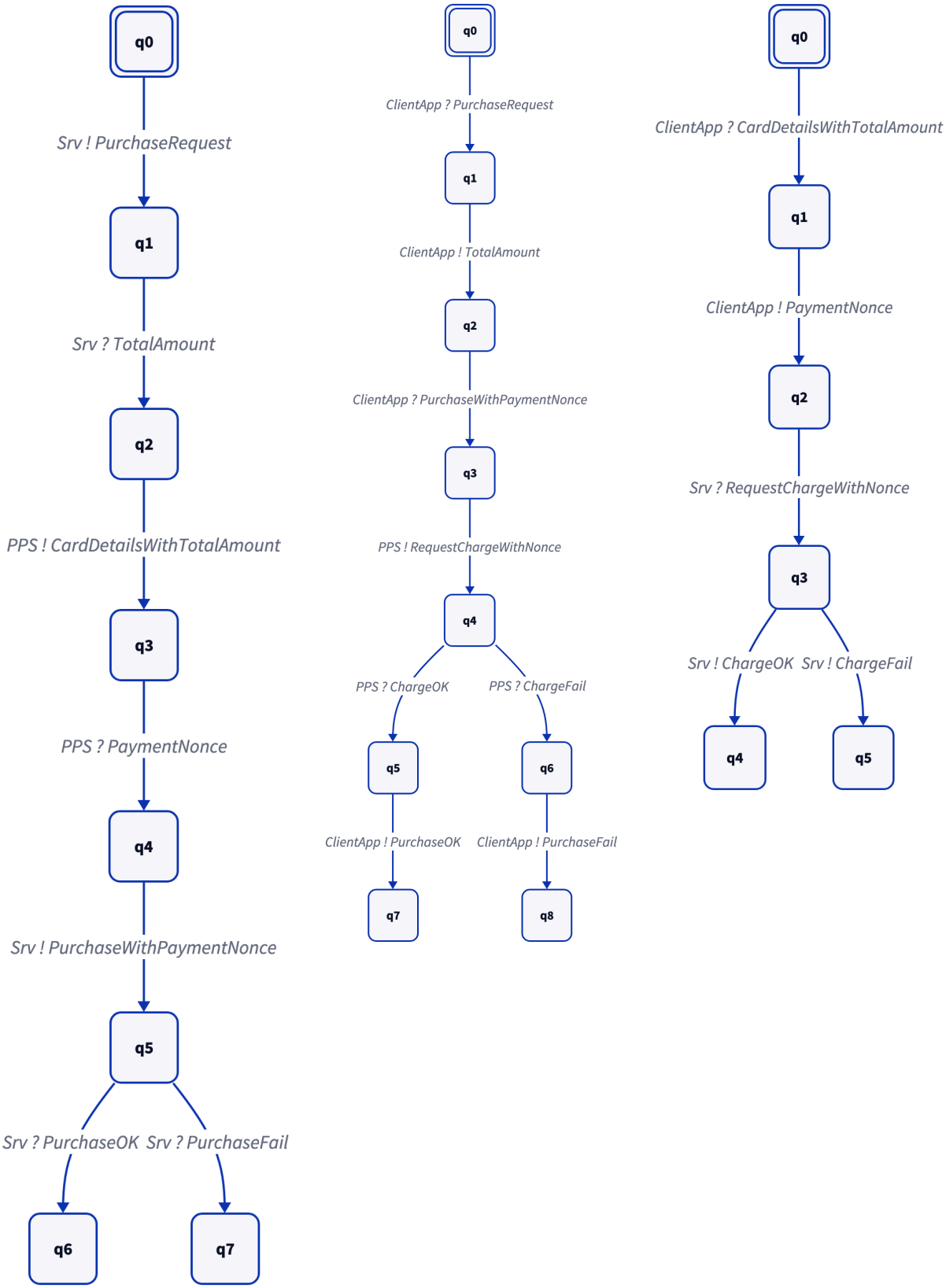


Figura 15: CFSMs para el cliente (izquierda, identificado como `ClientApp`), para el servidor (centro, identificado como `Srv`) y para el servicio de procesamiento de pagos (derecha, identificado como `PPS`).

El ejemplo está basado en interacciones reales típicas de procesadores de pago de tarjetas de crédito. La idea central es que los datos de la tarjeta de crédito no se envían al vendedor en ningún momento, sino que el usuario los envía directamente al servicio de procesamiento de

pagos junto con el monto total a pagar. Una vez que el servicio de procesamiento de pagos valida la información de la tarjeta, le devuelve al cliente un identificador aleatorio (*nonce*). Éste, a su vez, envía esta información al servicio del vendedor, que la reenvía al servicio de procesamiento de pagos para hacer efectiva la transacción. De esta manera, el servicio del vendedor evita lidiar con la información de la tarjeta de crédito, ya que para ello requeriría atravesar auditorías de seguridad de la información impuestas por las empresas emisoras de tarjetas.

Comenzaremos analizando el Service Client implementado en Java. El código del mismo se encuentra en el directorio `examples/credit-card-payments/client`[40] del repositorio. Creamos archivo `pom.xml` para allí definir dependencias del proyecto a ser descargadas y compiladas por Maven[41]. Agregamos las siguientes dependencias a `pom.xml`:

```xml
<dependencies>
  <dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-core</artifactId>
    <version>1.58.0</version>
  </dependency>
  <dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-protobuf</artifactId>
    <version>1.58.0</version>
  </dependency>
  <dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-stub</artifactId>
    <version>1.58.0</version>
  </dependency>
  <dependency> <!-- necessary for Java 9+ -->
    <groupId>org.apache.tomcat</groupId>
    <artifactId>annotations-api</artifactId>
    <version>6.0.53</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

Fragmento de código 17: Dependencias gRPC para el Service Client implementado en Java.

Además, se creó un *symlink* del subdirectorio `lib` del proyecto del Service Client que apunta a los *stubs* Java para SEArch (ubicados en `gen/java` dentro del mismo repositorio). Esto se hizo para poder importar y utilizar los *stubs* en el proyecto del Service Client. Recordemos que los *stubs* son código generado con la herramienta `buf` a partir de las definiciones en Protocol Buffers de mensajes e interfaces que exponen los *middlewares* y el *broker*. En un contexto de desarrollo más maduro del ecosistema SEArch, los *stubs* generados para distintos lenguajes estarían publicados en un repositorio de Maven. Esto permitiría a quien desarrolle un *Service Client* o *Service Provider* utilizar los *stubs* simplemente agregándolos como dependencia al archivo `pom.xml` del mismo modo que se hizo con las dependencias de gRPC. La misma idea aplica a los *stubs* generados para cualquier otro lenguaje de programación, utilizando los repositorios propios de cada ecosistema o lenguaje de programación (*PyPi* para Python, *npm* para TypeScript/Javascript, etc.).

De este modo, podemos importar en Java las distintas clases generadas automáticamente a partir de los archivos `.proto` como se muestra en el Fragmento de código 18 e iniciar una conexión al middleware propio.

[40] `https://github.com/pmontepagano/search/blob/main/examples/credit-card-payments/client`

[41] Maven es la herramienta de gestión de proyectos y dependencias más utilizada en el ecosistema de Java. `https://maven.apache.org/`

```java
import io.grpc.ManagedChannelBuilder;
import ar.com.montepagano.search.v1.PrivateMiddlewareServiceGrpc;

// Get the stub to communicate with the middleware
ManagedChannel channel = ManagedChannelBuilder.forTarget(
  "middleware-client:11000").usePlaintext().build();
PrivateMiddlewareServiceGrpc.PrivateMiddlewareServiceBlockingStub stub =
  PrivateMiddlewareServiceGrpc.newBlockingStub(channel);
```

Fragmento de código 18: Conexión a la interfaz privada del middleware desde Java.

Una vez instanciado el *stub* para comunicarse con el *middleware*, registramos el canal. Dado que en el marco de esta tesis no contamos con un algoritmo para validar la compatibilidad entre contratos, utilizaremos el parámetro opcional `preset_participants` (ver Fragmento de código 2) para indicarle al *broker* que queremos utilizar dos servicios específicos para que cumplan los roles del *backend* (con nombre «Srv» en el contrato utilizado) y del procesador de pagos («PPS»). Otra manera de realizar esto hubiese sido compilar una versión específica del *broker* en la que la función de chequeo de compatibilidad entre contratos responda afirmativamente para aquellos contratos que son idénticos, y devuelva una función de *mapping* de nombres de participantes que sea la función identidad.[42] Como en este ejemplo el registro de los proveedores ante el *broker* se realiza en cada ejecución, no sabemos el AppID asignado a cada uno hasta que ese proceso finaliza. Por ello, en este ejemplo, el Service Client solicita al usuario que provea los identificadores de ambos servicios. En un contexto de uso real, estos identificadores serían fijos, o bien no se utilizaría la funcionalidad de `preset_participants`, sino que se delegaría la selección de los proveedores al *broker*.

```java
RemoteParticipant pps = RemoteParticipant.newBuilder().setUrl(ppsHostname).setAppId(ppsAppId).build();
RemoteParticipant srv = RemoteParticipant.newBuilder().setUrl(srvHostname).setAppId(srvAppId).build();
RegisterChannelRequest request = RegisterChannelRequest.newBuilder().setRequirementsContract(
  contract).putPresetParticipants("PPS", pps).putPresetParticipants("Srv", srv).build();
RegisterChannelResponse response = stub.registerChannel(request);
var channelId = response.getChannelId();
```

Fragmento de código 19: Registro del canal en Java utilizando preset_participants.

En el Fragmento de código 19 en las líneas 1 y 2, las variables `ppsHostname` y `srvHostname` tienen la URI de la interfaz pública del *middleware* correspondiente a cada Service Provider. Del mismo modo, las variables `ppsAppId` y `srvAppId` contienen los identificadores de ambos Service Providers provistos por el *broker*. Al ejecutar el ejemplo, el usuario debe extraer esos valores de los *logs* y proveerlos al programa cliente.

Hasta este punto de la ejecución, el *middleware* del Service Client aún no se ha comunicado con el *broker*. Esto se realizará en la próxima interacción, cuando el cliente envíe el primer mensaje al canal (Fragmento de código 20).

[42]La funcionalidad de `preset_participants` requiere un rediseño para contemplar casos en los que los servicios predefinidos no comparten el mismo nombramiento de participantes que el Service Client. Ver `https://github.com/pmontepagano/search/issues/12` .

```
1  // Send PurchaseRequest with each item quantities and the shipping address
2  Map<String, Integer> items = new HashMap<>();
3  for (int selectedBook : selectedBooks) {
4      String bookTitle = bookTitles[selectedBook - 1];
5      if (items.containsKey(bookTitle)) {
6          items.put(bookTitle, items.get(bookTitle) + 1);
7      } else {
8          items.put(bookTitle, 1);
9      }
10 }
11 var body = ByteString.copyFromUtf8(String.format(
12   "{\"items\": %s, \"shippingAddress\": \"%s\"}",
13   gson.toJson(items), shippingAddress));
14 var msg = AppMessage.newBuilder().setType("PurchaseRequest").setBody(body).build();
15 var sendreq = AppSendRequest.newBuilder().setChannelId(
16   channelId).setRecipient("Srv").setMessage(msg).build();
17 var sendresp = stub.appSend(sendreq);
18 if (sendresp.getResult() != Middleware.AppSendResponse.Result.RESULT_OK) {
19     System.out.println("Error sending PurchaseRequest. Exiting...");
20     System.exit(1);
21 }
```

Fragmento de código 20: Envío del primer mensaje al canal desde Java.

Como vemos, para enviar un mensaje al *backend* el Service Client indica el tipo de mensaje (`PurchaseRequest`) y el contenido en *bytes* del mensaje en la línea 14. Como retomaremos en la Sección 6.4, en esta versión de la infraestructura el tipo de un mensaje no impone ninguna restricción al contenido. En el contexto de este ejemplo utilizamos JSON para codificar mensajes que contienen más de un dato, y un string simple para codificar tipos de datos básicos. Tanto el emisor y el receptor de un mensaje a través de un canal saben el tipo del mensaje, pero deberán serializar y deserializar el mismo a partir de los *bytes* del cuerpo del mensaje. La infraestructura no provee (por ahora) mecanismos para facilitar esto.

En las líneas 15 y 16 del Fragmento de código 20 se termina de conformar el *request* con el nombre del destinatario del mensaje según el contrato («Srv») y el mensaje. En la línea 17 se invoca a `stub.appSend` para enviar el mismo. Luego de realizar el envío, el Service Client verifica que el *middleware* haya respondido de manera exitosa. Esto podría no ser así (por ejemplo, si el *middleware* tiene su *buffer* lleno), y por lo tanto es importante verificarlo.

En este punto, el *middleware* del Service Client inicia el proceso de *brokering* enviando un mensaje al *Service Broker*, quien establece el canal enviando una notificación a los dos *Service Providers*. Una vez confirmada esta notificación por parte de los *Service Providers*, el *broker* indica a todos los participantes (incluso el *Service Client*) que la comunicación puede comenzar a través del canal. En este ejemplo, el *middleware* del *Service Client*, como ya tiene un mensaje en cola para enviar al participante *Srv*, establece una conexión directa con el *middleware* de dicho participante y envía el mensaje.

Luego de enviar el primer mensaje al *backend*, el cliente espera una respuesta de este. En el Fragmento de código 21 vemos cómo se realiza esto.

```
1  var recvreq = Middleware.AppRecvRequest.newBuilder().setChannelId(
2    channelId).setParticipant("Srv").build();
3  var recvresp = stub.appRecv(recvreq);
4  if (!recvresp.getMessage().getType().equals("TotalAmount")) {
5      System.out.println("Error receiving TotalAmount. Exiting...");
6      System.exit(1);
7  }
8  var total_amount = gson.fromJson(recvresp.getMessage().getBody().toStringUtf8(), double.class);
9  System.out.println("Total amount: " + total_amount);
```

Fragmento de código 21: Recepción de un mensaje en Java.

Para recibir un mensaje, basta con indicar el identificador del canal y el participante del cual queremos recibir un mensaje (línea 1). Luego, el mensaje se recibe invocando a `stub.appRecv` (línea 2). En las líneas 4-7 se verifica que el tipo de mensaje recibido sea el esperado. En este caso el contrato solo contempla un tipo posible de mensaje. De implementarse el monitoreo de corrección de contratos en los *middlewares* como se propone en la Sección 6.1, esta validación sería innecesaria cuando existe un único tipo posible de mensaje a recibir de parte de un participante, ya que ese chequeo lo haría directamente el *middleware*. Finalmente, vemos en la línea 8 cómo se deserializa el mensaje a un tipo de dato nativo (un `double` que representa el monto total a abonar).

El resto de la implementación del cliente en Java continúa de manera similar, realizando envíos y recepciones de mensajes utilizando el mismo identificador de canal. No detallaremos los siguientes pasos en esta sección. Remitimos al lector a leer en detalle el ejemplo completo en el repositorio de código de este trabajo.

La implementación del *backend* en Python se encuentra en el directorio `examples/credit-card-payments/client` del repositorio de código. Al igual que como se hizo en el ejemplo en Java, se creó un *symlink* del subdirectorio `lib` que apunta al directorio `gen/python` donde se encuentra el código generado a partir de las definiciones en Protocol Buffers.

Al tratarse de un *Service Provider*, en este caso registramos el servicio ante el *broker* a través de la invocación a la RPC `RegisterApp` del *middleware*. Como se ve en el Fragmento de código 22, una vez establecida la conexión del el *middleware* propio (cosa que se realiza fuera del fragmento, pero cuya referencia se almacena en la variable `grpc_channel`), se crea un *stub* de una manera similar a la que se vio en Java. Con dicho *stub*, luego, se invoca la RPC `RegisterApp`. Como se trata de una *server-streaming RPC*, se itera sobre las respuestas que el middleware envía al Service Provider (líneas 5-12). Se espera que el primer mensaje sea un *acknowledgement* con el AppId asignado por el *broker* (líneas 17-20). Luego, todos los siguientes mensajes serán notificaciones de inicio de un nuevo canal en el que este proveedor debe participar (líneas 13-16). Vemos en la línea 16 que se lanza una tarea concurrente de la función `session` (Fragmento de código 23). Este manejo de sesiones manual se puede evitar si se realiza el rediseño discutido en la Sección 3.7.4.

```python
from lib.search import v1 as search

async def main(grpc_channel):
    stub = search.PrivateMiddlewareServiceStub(grpc_channel)
    registered = False
    logger.info("Connected to middleware. Waiting for registration...")
    async for r in stub.register_app(
        search.RegisterAppRequest(
            provider_contract=search.LocalContract(
                format=search.LocalContractFormat.LOCAL_CONTRACT_FORMAT_FSA,
                contract=PROVIDER_CONTRACT,
            )
        )
    ):
        if registered and r.notification:
            logger.info(f"Notification received: {r.notification}")
            # Start a new session for this channel.
            asyncio.create_task(session(grpc_channel, r.notification))
        elif not registered and r.app_id:
            # This should only happen once, in the first iteration.
            registered = True
            logger.info(f"App registered with id {r.app_id}")
        else:
            logger.error(f"Unexpected response: {r}. Exiting.")
            break

    grpc_channel.close()
```

Fragmento de código 22: Registro de contrato de provisión y espera de notificaciones de inicio de canales en Python.

Una vez establecido un canal, la interacción que el *Service Provider* realiza con el canal es igual que la que realiza un *Service Client*. Podemos ver en las líneas 7-12 del Fragmento de código 23 cómo el *backend* recibe un mensaje de «ClientApp» y verifica (líneas 13-15) que el tipo del mensaje sea el esperado (`PurchaseRequest`). En las línea 16 se deserializa el cuerpo del mensaje, que contiene cada item de la compra, la cantidad de cada uno y la dirección postal del envío. Con esta información, el *backend* calcula el monto total de la compra (líneas 20-23) y lo envía a la aplicación cliente con el mensaje de tipo `TotalAmount` (líneas 25-39). El resto de la función `session` no se incluye aquí, ya que es una implementación lineal que sigue la descripción del contrato del Fragmento de código 16.

```python
async def session(grpc_channel, notification):
    logger.info("Starting new session...")
    stub = search.PrivateMiddlewareServiceStub(grpc_channel)
    channel_id = notification.channel_id

    # Receive purchase request from ClientApp.
    r = await stub.app_recv(
        search.AppRecvRequest(
            channel_id=channel_id,
            participant="ClientApp",
        )
    )
    if r.message.type != "PurchaseRequest":
        logger.error(f"Unexpected message type: {r.message.type}. Exiting.")
        return
    purchase_request = json.loads(r.message.body)
    logger.debug(f"Received purchase request: {purchase_request}")

    # Calculate total amount and send it to ClientApp.
    total_amount = decimal.Decimal(0)
    for book, quantity in purchase_request["items"].items():
        total_amount += PRICE_PER_BOOK * quantity
    total_amount += SHIPPING_COST
    logger.debug(f"Total amount: {total_amount}")
    r = await stub.app_send(
        search.AppSendRequest(
            channel_id=channel_id,
            recipient="ClientApp",
            message=search.AppMessage(
                type="TotalAmount",
                body=str(total_amount).encode("utf-8"),
            ),
        )
    )
    if r.result != search.AppSendResponseResult.RESULT_OK:
        logger.error(
            f"Failure sending message. Received AppSendResponse with status: {r.status}. Exiting."
        )
        return
    ...
```

Fragmento de código 23: Fragmento del cuerpo principal que implementa el _backend_ en Python para el caso de estudio.

El segundo Service Provider (el procesador de pagos o «PPS»), implementado en Go, sigue una estructura muy similar. El código para conectarse al *middleware* y registrar el servicio puede verse en el Fragmento de código 24. Solo se muestran algunos de los *imports* para hacer más breve el ejemplo en este documento.

```go
import (
  "google.golang.org/grpc"
  "google.golang.org/grpc/credentials/insecure"
  pb "github.com/pmontepagano/search/gen/go/search/v1"
)
const ppsContract = `
.outputs PPS
.state graph
q0 ClientApp ? CardDetailsWithTotalAmount q1
q1 ClientApp ! PaymentNonce q2
q2 Srv ? RequestChargeWithNonce q3
q3 Srv ! ChargeOK q4
q3 Srv ! ChargeFail q5
.marking q0
.end`
var opts []grpc.DialOption
opts = append(opts, grpc.WithTransportCredentials(insecure.NewCredentials()))
conn, err := grpc.Dial(*middlewareURL, opts...)
if err != nil {
  logger.Fatalf("Error connecting to middleware URL %s", *middlewareURL)
}
defer conn.Close()
stub := pb.NewPrivateMiddlewareServiceClient(conn)

// Register provider contract with registry.
req := pb.RegisterAppRequest{
  ProviderContract: &pb.LocalContract{
    Contract: []byte(ppsContract),
    Format:   pb.LocalContractFormat_LOCAL_CONTRACT_FORMAT_FSA,
  },
}
streamCtx, streamCtxCancel := context.WithCancel(context.Background())
defer streamCtxCancel()
stream, err := stub.RegisterApp(streamCtx, &req)
if err != nil {
  logger.Fatal("Could not Register App")
}
ack, err := stream.Recv()
if err != nil || ack.GetAppId() == "" {
  logger.Fatal("Could not receive ACK from RegisterApp")
}
appID := ack.GetAppId()
logger.Printf("Finished registration. Got AppId %s", appID)
```

Fragmento de código 24: Registro de un Service Provider en Go.

Al igual que en el ejemplo de Python, en Go también el Service Provider debe esperar al mensaje de tipo `RegisterAppResponse` de parte de su *middleware* para saber que debe iniciar la ejecución en un canal (ver Fragmento de código 25). En este caso tenemos una *goroutine* muy simple que espera cada uno de esos mensajes y, al recibirlo, lo reenvía a través de un *Go channel* (líneas 7-20). En paralelo, la *goroutine* principal itera infinitamente y se bloquea esperando un mensaje desde ese canal (líneas 21 y 22). Una vez recibido, lanza una *goroutine* que implementa el cuerpo principal del programa siguiendo el contrato de provisión declarado (líneas 34-65). En este documento mostramos solamente la recepción del primer mensaje. El resto de las interacciones son recepciones y envíos iguales a los ya vistos en secciones anteriores en Go.

```go
// wait on RegisterAppResponse stream to await for new channel
type NewSessionNotification struct {
  regappResp *pb.RegisterAppResponse
  err        error
}
recvChan := make(chan NewSessionNotification)
go func(stream pb.PrivateMiddlewareService_RegisterAppClient, recvChan chan NewSessionNotification) {
  // This goroutine simply waits for any new RegisterAppResponse in the stream and sends it
  // to the recvChan.
  for {
    newResponse, err := stream.Recv()
    recvChan <- NewSessionNotification{
      regappResp: newResponse,
      err:        err,
    }
    if err != nil {
      logger.Fatalf("Error receiving RegisterApp notification: %v", err)
    }
  }
}(stream, recvChan)
for {
  newSess := <-recvChan
  err := newSess.err
  if err == io.EOF {
    logger.Printf("middleware unexpectedly ended connection!")
    break
  }
  if err != nil {
    logger.Fatalf("Error receiving notification from RegisterApp: %v", err)
  }

  channelID := newSess.regappResp.GetNotification().GetChannelId()
  logger.Printf("Received Notification. ChannelID: %s", channelID)
  go func(channelID string, client pb.PrivateMiddlewareServiceClient) {
    // This is the actual program for PPS (the part that implements the contract).
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    // Receive CardDetailsWithTotalAmount from ClientApp.
    logger.Print("Waiting for CardDetailsWithTotalAmount...")
    recvResponse, err := client.AppRecv(ctx, &pb.AppRecvRequest{
      ChannelId:   channelID,
      Participant: "ClientApp",
    })
    if err != nil {
      logger.Fatal("Failed to receive CardDetailsWithTotalAmount from ClientApp.")
    }
    if recvResponse.Message.Type != "CardDetailsWithTotalAmount" {
      logger.Fatalf("Received invalid message of type %v", recvResponse.Message.Type)
    }
    // We don't validate the card details or the amount, this is just a demo.
    type CardDetailsWithTotalAmount struct {
      CardNumber         string  `json:"card_number"`
      CardExpirationDate string  `json:"card_expirationdate"`
      CardCVV            string  `json:"card_cvv"`
      TotalAmount        float64 `json:"total_amount"`
    }
    var cardDetails CardDetailsWithTotalAmount
    err = json.Unmarshal(recvResponse.Message.Body, &cardDetails)
    if err != nil {
      logger.Fatalf("Error unmarshalling CardDetailsWithTotalAmount: %v", err)
    }
    logger.Printf("Received CardDetailsWithTotalAmount: %v", cardDetails)
    ...
  }(channelID, stub)
}
```

Fragmento de código 25: Cuerpo principal del Service Provider implementado en Go.

Para ejecutar todos estos distintos servicios con facilidad, lo cual no incluye solamente a los tres programas en cuyos detalles se profundizó en esta sección, sino también a las tres instancias del

middleware y al *broker*, se utilizó Docker Compose[43], que es una herramienta que permite definir y ejecutar aplicaciones que necesitan múltiples *containers*. En el archivo `examples/credit-card-payments/docker-compose.yaml` se puede ver el detalle de los siete containers definidos. Allí se utilizaron distintas *networks* (redes) para definir explícitamente qué containers pueden comunicarse directamente con cuáles. El Service Client y los Service Providers solo pueden comunicarse directamente con su propia instancia del *middleware*. Mientras tanto, los *middlewares* pueden comunicarse entre sí y con el *broker*. El *broker* puede comunicarse directamente con cualquiera de los *middlewares*. Por último, en el mismo archivo de configuración hacemos referencia a los archivos `Dockerfile` correspondientes a cada container que contienen las instrucciones detalladas de cómo compilar e instalar las dependencias de cada servicio. El archivo `README.md` del mismo directorio contiene instrucciones para ejecutar el ejemplo.

[43] `https://docs.docker.com/compose/` .

6. Conclusiones y trabajo futuro

En el desarrollo de esta tesis hemos cumplido el objetivo central de delineado en la introducción: implementar una infraestructura que posibilite la ejecución distribuída de servicios en la que el proceso de *discovery* y su posterior *binding* se realicen en tiempo de ejecución. Asimismo, se utilizaron *Communicating Finite State Machines* para representar los contratos de interoperabilidad de servicios.

Uno de los logros más destacados de este trabajo radica en la economía de código necesario para alcanzar nuestros objetivos. En este sentido, se ha requerido un número sorprendentemente reducido de líneas de código para implementar la arquitectura de SEArch. [Inserte datos específicos aquí sobre la cantidad de líneas de código, una vez disponibles]. Esta eficiencia en la implementación subraya que la elección del lenguaje de programación y las herramientas de definición de interfaces fueron adecuadas y permitieron maximizar la efectividad y minimizar la complejidad.

Un aspecto igualmente relevante es que, durante la ejecución de este proyecto, no se ha necesitado introducir restricciones o compromisos en relación con nuestros objetivos iniciales. Esto subraya la robustez y la flexibilidad de SEArch como infraestructura para la ejecución distribuida de servicios.

La creación de una infraestructura de esta magnitud, que cumpla con todos los requisitos y estándares de calidad necesarios para ser considerada una plataforma productiva, constituye un proyecto de envergadura que solo puede lograrse a través de un enfoque de desarrollo de software altamente estructurado a nivel industrial y con la colaboración de un equipo de desarrolladores. Por esta razón, nos hemos enfocado en la creación de un prototipo cuyo propósito es demostrar la viabilidad práctica de dicha plataforma. Algunos detalles de implementación, como la resolución de cuestiones relacionadas con la escalabilidad de los componentes activos involucrados en la búsqueda y vinculación de servicios, se han dejado como notas a considerar en futuras etapas del proyecto. En este capítulo describimos distintas mejoras a implementar, sugiriendo, en cada caso, posibles diseños. Cada una de ellas será discutida en la profundidad que fue percibida durante el desarrollo de la tesis.

6.1. Monitoreo de corrección de los protocolos

Los *middlewares*, al tener disponible el contrato del participante al cual le proveen servicio, pueden también verificar, en tiempo de ejecución, que el protocolo del contrato sea respetado. Este monitoreo es útil para detectar *bugs* en la implementación de *Service Clients* y *Service Providers*, y también puede ser de utilidad para los rankings de proveedores que debería mantener el *Service Broker* (si un proveedor viola su contrato frecuentemente, sería deseable penalizarlo en los rankings). Se discuten en esta sección dos posibles maneras de implementar el monitoreo: distribuído o centralizado.

Para llevar a cabo el monitoreo de la corrección del contrato local, el middleware debe mantener actualizado cuál es el estado de la CFSM en el que se encuentra el autómata. Al inicializar un canal, el middleware guarda una referencia al estado inicial de la CFSM. Luego, en respuesta a cada operación `AppRecv` o `AppSend` realizada por su participante local, el middleware actuali-

za el puntero, siguiendo la función de transición definida por la CFSM. Si en algún punto el participante local ejecuta una operación de envío o recepción que no está contemplada en la especificación del contrato, se considera que se ha producido una violación del contrato.

Para la implementación de este monitoreo, se puede desarrollar un monitor con la siguiente interfaz:

```
type LocalContractMonitor interface {
  Init(LocalContract) LocalContractMonitor
  TransitionToNextState(participant string, send bool, msgtype string) error
}
```

Fragmento de código 26: Interfaz de monitor.

Dado un contrato local, con la función `Init` obtenemos una instancia del monitor que deberá ser almacenada como una propiedad del canal dentro del *middleware*. Luego, en los *handlers* de `AppRecv` y `AppSend`, el *middleware* debería llamar a la función `TransitionToNextState` del monitor, pasándole el nombre del participante sobre el cual se hace el envío o recepción y el tipo de mensaje (en el caso de una recepción, el tipo de mensaje será el del primer mensaje en el buffer). Esta función debe devolver un error si se produce una violación del contrato. Una pregunta que surge (y dejamos abierta) es qué acciones llevar a cabo ante una violación de contratos.

El mecanismo descripto sirve para detectar violaciones al contrato del participante local. Es decir, si el participante local envía (o intenta recibir) un mensaje no contemplado por su propio contrato, o si recibe, de otro participante, un mensaje no esperado. Sin embargo, puede ser también deseable que el *middleware* del *Service Client*, al cumplir el rol de *orquestador* del canal, detecte violaciones al contrato global por parte de cualquiera de los participantes (en interacciones en las que el *Service Client* no participa). Para poder tener esta visión global del estado del canal, una posible solución es hacer que los *middlewares* de todos los participantes informen al *middleware* del *Service Client* sobre cada envío y recepción. La desventaja de esta solución es que se agregan mayores mensajes de control que se centralizan en el *middleware* del *Service Client* y no queda claro que ésto aporte un valor que lo justifique. Una segunda opción es proveer un mecanismo de reporte de errores. Es decir, que los *middlewares* reporten al *middleware* que coordina el canal solamente en caso de violaciones de contratos y no todos los envíos y recepciones.

6.2. Ranking de proveedores

Como se mencionó en la Sección 4.3, no se implementó en el marco de esta tesis un *ranking* entre proveedores, sino que nos limitamos a determinar si un proveedor es compatible con un requerimiento o no. Se sugieren brevemente algunos datos que pueden ser tenidos en cuenta por el *Service Broker* a la hora de determinar el *ranking* de un proveedor en particular.

En primer lugar, se propone que si un *Service Provider* no responde al mensaje `InitChannel` durante la inicialización de un canal, no solo se marque a dicho proveedor como fuera de línea temporalmente, sino que también se lo penalice en su *ranking*. Esta medida puede contribuir de manera significativa a reducir la latencia a la hora de establecer un canal.

Adicionalmente, como mencionamos más arriba en la sección sobre monitoreo, es razonable aplicar una penalización a aquellos proveedores que incumplan su propio contrato. Además, se plantea la necesidad de considerar la variable de latencia entre el Service Client que está requiriendo el servicio y cada proveedor como un criterio adicional para la elaboración de rankings, lo que proporcionaría una visión más completa de la calidad de los servicios. Esta última

propiedad es la más difícil de evaluar, ya que depende de la ubicación de ambos participantes en la red, lo cual no es directamente medible por el *Service Broker*.

Es importante resaltar que los rankings son inherentemente dinámicos y deben actualizarse periódicamente a medida que el Broker reciba nueva información sobre el desempeño de los proveedores, por lo que se recomienda la implementación de un sistema de actualización automática para mantener la relevancia y utilidad de los rankings en un entorno en constante evolución.

6.3. Logging distribuído

Uno de los aspectos cruciales que consideramos pero finalmente no implementamos se relaciona con la centralización de logs. La gestión eficiente de registros y eventos en un entorno distribuido es fundamental para el diagnóstico, el monitoreo y la resolución de problemas. Reconociendo esta necesidad, contemplamos la posibilidad de abordar este desafío haciendo que cada *middleware* reporte sus *logs* a un servicio centralizado para cada canak. Para lograr esto, se podría agregar un campo opcional a los mensajes `RegisterChannelRequest`, `BrokerChannelRequest` e `InitChannelRequest` en el que el *Service Client* pueda designar la URI del colector de logs correspondiente al canal.

De esta manera, el iniciador del canal tendría la flexibilidad de seleccionar un servidor de logs adecuado para sus necesidades específicas. Los middleware que intervienen en el canal enviarían sus registros a esta URI de servidor de logs designada, permitiendo la acumulación y el análisis centralizado de los registros generados por todos los componentes del sistema.

Además, consideramos una situación en la que un *Service Provider*, para proveer su servicio, a su vez utiliza un servicio de un tercero también a través de SEArch, en este caso cumpliendo el rol de *Service Client*. En este escenario, podría ser deseable que el servicio de terceros también reporte sus logs al mismo servidor de logs. Si se tratara de servicios de organizaciones distintas, no sería esperable que los logs se recolecten «transitivamente», ya que hacer eso expondría detalles de implementación del *Service Provider* que no deberían ser relevantes para los consumidores de ese servicio. Pero si pensamos este problema dentro del contexto de una única organización que utiliza SEArch internamente como mecanismo de *discovery* y *binding* automático para servicios internos, en ese caso sí sería deseable centralizar los logs y tener trazas en profundidad que abarquen múltiples servicios.

Una opción que exploramos para implementar esta funcionalidad fue el uso de *Traces* siguiendo el estándar de OpenTelemetry [21], que es un estándar de *logging* estructurado que permite conformar un árbol de trazas de ejecución de manera distribuída.

6.4. Tipos complejos de datos para los mensajes

Como vimos en el Capítulo 2, los mensajes que se intercambian las CFSMs tienen un tipo asociado. En el contexto de este trabajo, cada tipo es simplemente una cadena de caracteres. Sería deseable proveer un sistema rico de tipos como los de los lenguajes de programación más populares.

Una primera solución que no requiere cambios en el sistema desarrollado es utilizar estas cadenas de caracteres como URLs de un repositorio de tipos. Se podría utilizar, por ejemplo, la URL de un archivo `.proto` que contenga la descripción del tipo en Protocol Buffers. En [17] se describe una técnica similar. Es importante aclarar que el uso de Protocol Buffers, si bien ofrece un conjunto de herramientas útiles, no resuelve el problema de fondo, ya que no contamos

con un mecanismo de unificación y chequeo de tipos para tipos definidos en Protocol Buffers. Queda abierto el problema de chequeo de compatibilidad entre CFSMs que utilicen un sistema de tipos complejo en sus mensajes.

Bibliografía

[1] Luca de Alfaro y Thomas A. Henzinger. 2001. Interface Theories for Component-Based Design. En *Embedded Software*, 2001. Springer Berlin Heidelberg, Berlin, Heidelberg, 148-165. Recuperado a partir de https://doi.org/10.1007/3-540-45449-7_11

[2] Christel Baier y Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT Press. Recuperado a partir de https://mitpress.mit.edu/9780262026499/principles-of-model-checking/

[3] Daniel Brand y Pitro Zafiropulo. 1983. On Communicating Finite-State Machines. *J. ACM* 30, 2 (abril de 1983), 323. Recuperado a partir de https://doi.org/10.1145/322374.322380

[4] Ezequiel Davidovich Caballero. 2020. Autómatas para la vinculación parcial de servicios. Recuperado a partir de http://gestion.dc.uba.ar/media/academic/grade/thesis/Automata_for_Partial_Binding_of_Services.pdf

[5] Pierre-Malo Deniélou y Nobuko Yoshida. 2012. Multiparty Session Types Meet Communicating Automata. En *Programming Languages and Systems*, 2012. Springer Berlin Heidelberg, Berlin, Heidelberg, 194-213. Recuperado a partir de https://doi.org/10.1007/978-3-642-28869-2_10

[6] Alan A.A. Donovan y Brian W. Kernighan. 2015. *The Go Programming Language* (1st ed.). Addison-Wesley Professional. Recuperado a partir de https://www.gopl.io/

[7] José Luiz Fiadeiro y Antónia Lopes. 2011. An Interface Theory for Service-Oriented Design. En *Fundamental Approaches to Software Engineering*, 2011. Springer Berlin Heidelberg, Berlin, Heidelberg, 18-33. Recuperado a partir de https://doi.org/10.1007/978-3-642-19811-3_3

[8] José Luiz Fiadeiro, Antónia Lopes, y Laura Bocchi. 2011. An abstract model of service discovery and binding. *Formal Aspects of Computing* 23, (2011), 433-463. Recuperado a partir de https://doi.org/10.1007/s00165-010-0166-z

[9] Object Management Group. 2018. The Distributed Ontology, Model, and Specification Language (DOL), Version 1.0. (2018). Recuperado a partir de https://www.omg.org/spec/DOL/1.0/About-DOL

[10] Roberto Guanciale y Emilio Tuosto. 2021. PomCho: A tool chain for choreographic design. *Science of Computer Programming* 202, (2021), 102535. Recuperado a partir de https://www.sciencedirect.com/science/article/pii/S016764232030143X

[11] Kohei Honda, Nobuko Yoshida, y Carbone Marco. 2008. Multiparty Asynchronous Session Types. En *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008. ACM Press, San Francisco, California, USA, 273-284. Recuperado a partir de https://doi.org/10.1145/2827695

[12] M.N. Huhns y M.P. Singh. 2005. Service-oriented computing: key concepts and principles. *IEEE Internet Computing* 9, 1 (2005), 75-81. Recuperado a partir de https://doi.org/10.1109/MIC.2005.21

[13] Ionuţ Ţuţu y Jose Luiz Fiadeiro. 2015. Service-Oriented Logic Programming. *Logical Methods in Computer Science* (agosto de 2015). Recuperado a partir de https://lmcs.episciences.org/1579

[14] Julien Lange, Emilio Tuosto, y Nobuko Yoshida. 2015. From Communicating Machines to Graphical Choreographies. En *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '15*, 2015. ACM Press, Mumbai, India, 221-232. Recuperado 18 de julio de 2020 a partir de http://dl.acm.org/citation.cfm?doid=2676726.2676964

[15] Julien Lange, Emilio Tuosto, y Nobuko Yoshida. 2017. A Tool for Choreography-Based Analysis of Message-Passing Software. *Behavioural Types: from Theory to Tools*. Recuperado a partir de https://kar.kent.ac.uk/62371/

[16] Google LLC. 2016. Recuperado a partir de https://protobuf.dev/programming-guides/proto3/

[17] Google LLC. 2023. Protocol Buffers Documentation: Self-describing messages. Recuperado a partir de https://protobuf.dev/programming-guides/techniques/#self-description

[18] Agustín E. Martinez Suñé y Carlos G. Lopez Pombo. 2019. Automatic Quality-of-Service Evaluation in Service-Oriented Computing. En *Coordination Models and Languages*, 2019. Springer International Publishing, Cham, 221-236. Recuperado a partir de https://doi.org/10.1007/978-3-030-22397-7_13

[19] Agustín Martinez Suñé y Carlos Lopez Pombo. 2020. Quality of Service Ranking by Quantifying Partial Compliance of Requirements. En. 181-189. Recuperado a partir de https://doi.org/10.1007/978-3-030-50029-0_12

[20] Nicholas Ng. 2020. nickng/cfsm. Recuperado 18 de julio de 2020 a partir de https://github.com/nickng/cfsm

[21] OpenTelemetry. 2023. OpenTelemetry Documentation: Traces. Recuperado a partir de https://opentelemetry.io/docs/concepts/signals/traces

[22] Rob Pike. 2013. The cover story. *The Go Blog*. Recuperado 30 de octubre de 2023 a partir de https://go.dev/blog/cover

[23] Amir Pnueli. 1981. The temporal semantics of concurrent programs. *Theoretical Computer Science* 13, 1 (1981), 45-60. Recuperado a partir de https://www.sciencedirect.com/science/article/pii/0304397581901109

[24] Diego Norberto Senarruzza Anabia. 2023. Bisimulación de Data-aware Communicating Finite State Machines con propiedades en las acciones.

[25] Ignacio Vissani, Carlos Gustavo Lopez Pombo, IonuŢ ŢuŢu, y José Luiz Fiadeiro. 2015. A Full Operational Semantics for Asynchronous Relational Networks. En *Recent Trends in Algebraic Development Techniques*, 2015. Springer International Publishing, Cham, 131-150. Recuperado a partir de https://doi.org/10.1007/978-3-319-28114-8_8

[26] Ignacio Vissani, Carlos Gustavo López Pombo, y Emilio Tuosto. 2015. Communicating machines as a dynamic binding mechanism of services. En *Proceedings Eighth International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2015, London, UK, 18th April 2015 (EPTCS)*, 2015. 85-98. Recuperado a partir de https://doi.org/10.4204/EPTCS.203.7

[27] Ionuţ Ţuţu y José Luiz Fiadeiro. 2013. A Logic-Programming Semantics of Services. En *Algebra and Coalgebra in Computer Science*, 2013. Springer Berlin Heidelberg, Berlin, Heidelberg, 299-313. Recuperado a partir de https://doi.org/10.1007/978-3-642-40206-7_22