Universidad Nacional Autónoma de México
Posgrado en Ciencia e Ingeniería de la Computación

Pilas y colas concurrentes de larga vida desde la perspectiva topológica

T E S I S

Que para optar por el grado de:
Maestro en Ciencia e Ingeniería de la Computación

Presenta:
Pablo Enrique Zenil Rivas

Director de tesis:
Dr. Armando Castañeda Rojano
Instituto de Matemáticas

Ciudad Universitaria, Cd. Mx., 19 de noviembre de 2019

Agradecimientos

Quiero agradecer a mi asesor, el Dr. Armando Castañeda, por su apoyo y paciencia a lo largo de la maestría y en el desarrollo de este trabajo de tesis. Gracias a mis sinodales: el Dr. Manuel Alcántara, el Dr. David Flores, el Dr. Sergio Rajsbaum y el Dr. Carlos Velarde, por sus observaciones y comentarios que enriquecieron este trabajo.

Agradezco a CONACYT por el apoyo económico durante mis estudios de maestría. Gracias a la UNAM por permitirme ser parte de esta gran institución. Gracias a mi familia y amigos por ese apoyo y amor incondicional.

Resumen

Los objetos secuenciales y las tareas distribuidas son la forma predominante de especificar problemas distribuidos. Un *objeto secuencial* es definido por una *especificación secuencial* y puede ser invocado múltiples veces por cada proceso. En contraste, una *tarea* puede ser invocada una sola vez por cada proceso y determina, para cada conjunto de procesos y para cada posible asignación de valores de entrada, los valores de salida válidos.

Un *objeto concurrente* es un objeto definido por una especificación secuencial, al que los procesos pueden acceder de manera concurrente. Un algoritmo distribuido *implementa* a un objeto concurrente si satisface su especificación secuencial.

Las tareas son utilizadas en el estudio de la computabilidad distribuida y pueden ser modeladas a través de espacios topológicos combinatorios llamados *complejos simpliciales*. Por otro lado, los *protocolos* son la manera de especificar algoritmos distribuidos considerando un modelo de cómputo determinado, y también pueden ser modelados a través de complejos simpliciales.

Con todo esto, utilizando técnicas topológicas, podemos verificar que una tarea puede ser resuelta por un protocolo si la representación topológica del protocolo puede ser mapeada, bajo algunas condiciones específicas, a la representación topológica de la tarea. Esto implica que puede haber una implementación que satisfaga la especificación de la tarea, en el modelo considerado por el protocolo.

En este trabajo demostramos la imposibilidad de la implementación de pilas y colas concurrentes de larga vida, en el *modelo de escritura-lectura por capas*.

Primero, dado que las tareas carecen de poder expresivo para representar pilas y colas concurrentes, utilizamos a las *tareas refinadas*, una extensión de las tareas con mayor poder expresivo, para dar una representación de estos objetos. Después, utilizando argumentos topológicos, demostramos que es imposible mapear la representación topológica de cada tarea a la representación topológica del protocolo correspondiente, para el modelo de escritura-lectura por capas.

Índice general

Introducción IX

1. Preliminares **1**
1.1. Sistema distribuido 1
1.2. Linealizabilidad 2
1.2.1. Definiciones formales 4
1.3. Topología combinatoria 10
1.3.1. Complejos simpliciales abstractos 10
1.3.2. Mapeos simpliciales 11
1.3.3. Mapeos portadores 13
1.3.4. Complejos simpliciales cromáticos 14
1.3.5. Subdivisión cromática estándar 15
1.3.6. Definiciones formales 17
1.3.7. Operación Join 24
1.4. Enfoque topológico del cómputo distribuido 24
1.5. Resumen 25

2. Especificación de objetos concurrentes **27**
2.1. Tareas 27
2.1.1. Tarea Consenso binario 28
2.1.2. Tarea Inmediate-snapshot 29
2.1.3. Definiciones formales 31
2.1.4. Satisfacción de una tarea 32
2.1.5. Limitaciones de las tareas 33
2.2. Tareas refinadas 35
2.2.1. Pilas concurrentes 36
2.2.2. Definiciones formales 42
2.2.3. Satisfacción de una tarea refinada 43

2.3. Resumen 47

3. Especificación de algoritmos distribuidos **49**
3.1. Modelo de cómputo 49
3.2. Protocolos 51
3.2.1. Protocolos immediate-snapshot de una capa 52
3.2.2. Protocolos immediate-snapshot multicapa 55
3.2.3. Resolución de una tarea 56
3.2.4. Definiciones formales 57
3.2.5. Protocolos multi-shot 59
3.3. Resumen 67

4. Resultados **69**
4.1. Una invariante del protocolo 69
4.2. Resultados de imposibilidad 77
4.3. Resumen 82

5. Conclusiones **85**

Bibliografía **87**

Introducción

Un *sistema distribuido* [1, 2, 5] es un conjunto de procesos que se comunican entre sí para resolver un *problema distribuido*.

Cada *proceso* ejecuta un *protocolo* o algoritmo para resolver un problema. Los procesos inician el protocolo en un estado inicial y ejecutan una secuencia de pasos concurrentes, determinados por el mismo protocolo. En cada paso concurrente, cada proceso se comunica con los demás, a través de un ambiente de comunicación determinado por el sistema, y realiza un cómputo local. Al completar el protocolo, cada proceso decide un valor de salida.

Los *objetos secuenciales* [2, 5] y las *tareas distribuidas* [1, 5] son las dos formas predominantes de especificar un problema distribuido.

Un objeto provee un conjunto de métodos que permiten a un conjunto de procesos, modificar su estado. Un objeto secuencial está determinado por una *especificación secuencial*. Las llamadas a los métodos del objeto, hechas por los distintos procesos, son secuenciales.

Un *objeto concurrente* es un objeto compartido al que los procesos pueden acceder de manera concurrente y usualmente es definido por una especificación secuencial. Un algoritmo distribuido *implementa* a un objeto concurrente si satisface su especificación secuencial. La *linealizabilidad* [2, 3] provee una caracterización formal del comportamiento concurrente y correcto de estos objetos, en términos de especificaciones secuenciales equivalentes. Una implementación es *linealizable* si cada una de sus ejecuciones son *linealizables*.

Por otro lado, las tareas distribuidas (o simplemente tareas) son utilizadas en la especificación de problemas distribuidos que permiten una sola invocación a un método, por cada proceso. Las tareas representan un problema en términos de asignaciones de valores de entrada y de salida válidos. Una tarea define para cada conjunto de procesos que pueden correr concurrentemente y cada asignación de valores de entrada a los procesos, los valores de salida válidos de los procesos.

Los objetos secuenciales y las tareas son estilos de especificaciones muy diferentes: mientras que las tareas determinan qué debe de pasar cuando un conjunto de

procesos corren concurrentemente para resolver un problema distribuido, los objetos secuenciales solamente especifican qué pasa cuando los procesos corren secuencialmente.

Hay problemas distribuidos que no pueden ser expresados como objetos secuenciales, por ejemplo, el *immediate-snapshot* [1, 2]. Otros no pueden ser expresados como tareas, por ejemplo, las pilas y colas concurrentes. Además, hay problemas distribuidos que son expresados con mayor naturalidad como tareas. Usualmente, las tareas especifican problemas en los que cada proceso invoca una sola vez a un método, mientras que los objetos especificados secuencialmente son de larga vida, es decir, un proceso puede invocar muchas veces a un método. El consenso, un problema fundamental del cómputo distribuido, puede ser expresado con ambos formalismos.

Las tareas son utilizadas en el estudio de la computabilidad distribuida, lo que ha derivado en el desarrollo de la conexión entre el cómputo distribuido y la topología. Las tareas pueden ser modeladas a través de espacios topológicos combinatorios llamados *complejos simpliciales* [1, 4].

Mientras que las tareas especifican un problema distribuido, los *protocolos* [1, 4] son la manera de especificar algoritmos distribuidos, considerando un modelo de cómputo determinado. Un protocolo representa a un algoritmo distribuido en términos de configuraciones iniciales y finales, y también puede ser modelado a través de complejos simpliciales.

Todo esto permite aplicar técnicas topológicas en el estudio de la solubilidad de las tareas en algún modelo de cómputo determinado. Si la representación topológica de un protocolo puede ser mapeada, bajo algunas condiciones específicas, a la representación topológica de una tarea, entonces la tarea puede ser resuelta por el protocolo. Esto implica que puede haber una implementación que satisfaga la especificación de la tarea, en el modelo considerado por el protocolo.

Esta metodología no se ha aplicado a objetos secuenciales en general, y mucho menos en casos particulares como las colas y pilas concurrentes.

En esta tesis estudiaremos, en el marco de la topología combinatoria, el caso de la imposibilidad de la implementación de pilas y colas concurrentes en el *modelo de escritura-lectura por capas* [1].

Estos resultados son bien conocidos en el contexto operacional y se obtienen utilizando *números de consenso* [2]. El aporte principal de este trabajo es la obtención de estos resultados aplicando el método topológico por primera vez a pilas y colas de larga vida.

En el capítulo 1, estableceremos las bases para la representación formal de las pilas y colas concurrentes en el marco operacional vía la linealizabilidad. También estableceremos los conceptos básicos (complejos simpliciales, simplejos, mapeos portadores

y simpliciales) que nos permitirán trabajar en el marco de la topología combinatoria.

En el capítulo 2 veremos cómo las tareas carecen del poder expresivo para representar a las pilas y las colas concurrentes. Utilizaremos a las *tareas refinadas* [5], una extensión de las tareas con mayor poder expresivo, para dar una representación de las pilas y colas concurrentes. Daremos una noción de satisfacción por una especificación secuencial, para ambos tipos de tarea.

En el capítulo 3 daremos la definición del modelo de cómputo de escritura-lectura por capas y la definición combinatoria de los protocolos para este modelo. Extenderemos la definición de protocolo, considerando a las tareas refinadas. Daremos también, en términos topológicos, una noción de resolución de una tarea por un protocolo. Esto establece las condiciones necesarias para que un objeto concurrente pueda ser implementado en el modelo de cómputo considerado.

En el capítulo 4, utilizando argumentos topológicos, veremos que es imposible que exista una implementación de las pilas y colas concurrentes, en el modelo de escritura-lectura por capas.

Finalmente, en el capítulo 5 presentamos las conclusiones de este trabajo.

En cada capítulo, cada sección introduce los conceptos de manera informal, con el objetivo de familiarizar al lector y facilitarle la lectura. Todo esto es seguido de una subsección con las definiciones y resultados formales correspondientes a esa sección.

Capítulo 1

Preliminares

En este capítulo estableceremos las bases teóricas para poder estudiar las pilas y colas concurrentes de larga vida, desde el *enfoque topológico del cómputo distribuido*. Por el lado operacional, veremos cómo las *especificaciones secuenciales* proveen una caracterización formal de *objetos secuenciales* [2]; y cómo la *linealizabilidad* [2, 3] provee una caracterización formal del comportamiento concurrente y correcto de *objetos concurrentes*, en términos de especificaciones secuenciales equivalentes. Después, definiremos los elementos necesarios (*complejos simpliciales*, *mapeos portadores* y *mapeos simpliciales*) [1] que nos permitirán representar y estudiar estos objetos en el marco de la topología combinatoria.

1.1. Sistema distribuido

Un *sistema distribuido* [1, 2] es un conjunto de máquinas de estado llamadas *procesos* que se comunican entre sí a través de un ambiente de comunicación, para resolver un problema distribuido. Es conveniente pensar a un proceso como un autómata determinista con un conjunto de estados posiblemente infinito. Cada transición está determinada por el estado actual del proceso y el estado del ambiente de comunicación.

Cada proceso ejecuta un *protocolo*, comienza en un estado inicial y realiza un paso a la vez hasta que falla, se para sin realizar ningún paso adicional; o termina, completando el protocolo. Usualmente, en cada paso, cada proceso realiza un cómputo local y se comunica con otros procesos a través del ambiente de comunicación que provee el modelo. Los procesos corren de forma *concurrente*, es decir, pueden realizar pasos que se entrelazan en el tiempo de manera no determinista.

El estado del protocolo está determinado por los estados de los procesos que no

fallan y el estado del ambiente de comunicación. Una *ejecución* del sistema, es una secuencia de transiciones de estado de proceso. Una ejecución lleva el sistema de un estado a otro y está determinada por los procesos que realizaron algún paso y los eventos de comunicación que ocurrieron.

Para este trabajo, vamos a considerar un modelo de cómputo en el que los procesos se comunican escribiendo y leyendo de una memoria compartida. Los procesos serán capaces de combinar una escritura y la lectura de una secuencia arbitraria de palabras contiguas, en un sólo paso atómico llamado *immediate-snapshot* [2].

Los procesos se ejecutan de manera *asíncrona*. Cada proceso corre a una velocidad arbitraria que puede variar con el tiempo y no depende de otros procesos. En este modelo, las fallas son indetectables, es decir, un proceso que no responde puede haber fallado o ser muy lento y no hay forma de que otro proceso sepa cuál es el caso.

Cada proceso debe completar el protocolo en un número acotado de pasos y no podrá esperar a ningún otro proceso. Por esto decimos que el modelo es *libre de espera* o *wait-free* [2].

La formalización de estos conceptos la veremos en la sección 1.2 (lo que respecta a procesos y objetos) y en el capítulo 3 (el modelo de comunicación y los protocolos).

1.2. Linealizabilidad

Un *objeto* provee un conjunto de métodos que permiten manipular el estado interno del mismo. Para un *objeto secuencial*, cada *método* puede ser descrito por una pareja que consta de una *precondición* y una *postcondición*. La precondición describe el estado del objeto antes de invocar el método, y la postcondición, describe el estado del objeto al regresar del método junto con el valor regresado. Sin embargo, para un *objeto concurrente*, las llamadas a los métodos se pueden traslapar en el tiempo por lo que no tiene sentido caracterizar a cada método en términos de precondiciones y postcondiciones.

La *linealizabilidad* [2, 3], es una condición de corrección para implementaciones (algoritmos) de objetos concurrentes. Esta condición caracteriza el comportamiento concurrente de un objeto en términos de un comportamiento secuencial equivalente. De manera intuitiva, cada llamada a un método tiene efecto de manera instantánea en algún punto entre su invocación y su respuesta, esto permite describir el estado del objeto antes y después de dicho instante.

La linealizabilidad tiene dos propiedades formales muy útiles:

- *Non-blocking*: no se requiere que un proceso espere que otro termine una llamada en curso a un método.

- *Local*: un objeto compuesto por objetos linealizables, es también linealizable.

La ejecución de un sistema concurrente es modelada por una *historia*, una secuencia finita de (eventos) *invocaciones* a métodos y de *respuestas* de los mismos. Una *llamada a un método* en una historia, es un par que consiste en una invocación y la siguiente respuesta correspondiente, hechas por el mismo proceso. Una historia es *secuencial* si el primer evento es una invocación, y cada invocación, es seguida inmediatamente por su respuesta correspondiente. Una historia es *linealizable* si, para cada llamada a un método, es posible encontrar un punto entre la invocación y la respuesta tal que esos puntos inducen una historia secuencial, en algún sentido equivalente, que es válida, de acuerdo a la especificación del objeto. La implementación de un objeto es linealizable si todas las historias de sus ejecuciones son linealizables. Cabe destacar que todos estos conceptos son estándares en la literatura de cómputo distribuido ([1, 2]).

En adelante nos enfocaremos en pilas (LIFO) concurrentes, pensando en que todo lo que digamos acerca de estos objetos también aplica para las colas (FIFO) concurrentes, a menos que se indique lo contrario.

Por ejemplo, supongamos que tres procesos a, b y c, utilizan una pila concurrente q. La ejecución de la figura 1.1, es linealizable para q.

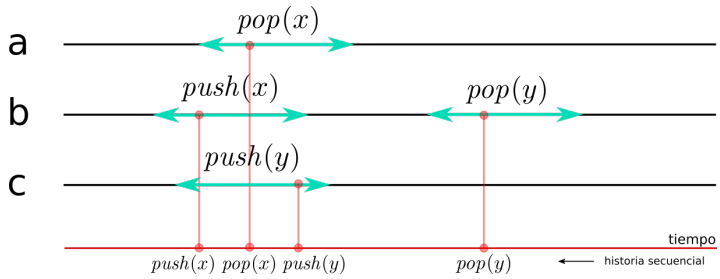

Figura 1.1: Una ejecución linealizable de q.

Cada intervalo (línea verde con flechas) representa la llamada a un método, los puntos (rojos) de linealización se proyectan hacia la línea del tiempo (abajo roja), estos determinan una historia secuencial. De manera informal, podemos *elegir* cada punto de linealización tal que la historia secuencial inducida, sea válida de acuerdo a la especificación secuencial de q.

La ejecución de la figura 1.2, no es linealizable para q, pues no hay manera de elegir los puntos de linealización tal que la historia secuencial sea válida.

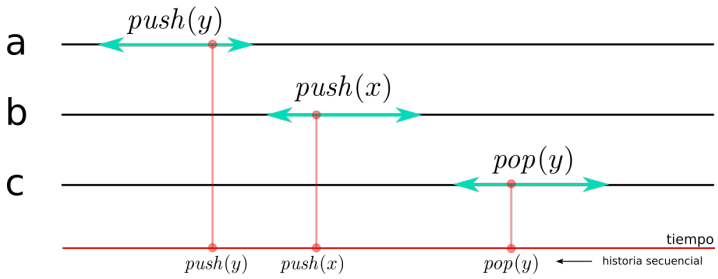

Figura 1.2: Una ejecución no linealizable de q.

Si q es una implementación linealizable, está garantizado que la ejecución de la figura 1.2 no puede darse.

1.2.1. Definiciones formales

En un sistema distribuido hay n procesos, cada uno con un nombre o identificador (id) único, tomado de un conjunto de nombres Π. [1] Usualmente $\Pi = [n]$, donde $[n] = \{1, \ldots, n\}$. Al proceso con id $j \in \Pi$, le decimos "el proceso j" o "el j-ésimo proceso".

El j-ésimo proceso es un autómata (en el sentido usual) con un conjunto de estados Q_j, el cual incluye un conjunto de *estados iniciales* Q_j^i y un conjunto de *estados finales* Q_j^f. Cada proceso conoce su propio nombre, aunque no conoce el nombre de los demás procesos *a priori*. Cada estado s de un proceso, incluye una componente inmutable *name* (el nombre o id del proceso), con un valor tomado de Π y denotada por $name(s)$. Es decir, si en una ejecución de un proceso, éste va del estado s al estado s', entonces $name(s) = name(s')$. De manera similar, cada estado s, incluye una componente mutable *view* (la vista del proceso), denotada por view(s) y que puede cambiar de estado a estado en una ejecución del proceso. Un estado s está definido por su nombre y su vista, con lo que s puede ser descrito por una tupla (j, v), donde $name(s) = j$ y $view(s) = v$.

Definición 1.1. Una *configuración* C del sistema distribuido es un conjunto de estados de procesos, correspondiente al *estado del sistema*, en un momento en el tiempo.

Cada proceso aparece a lo más una vez en cada configuración. Es decir, si s_0 y s_1 son estados distintos en una configuración C, entonces $name(s_0) \neq name(s_1)$.

[1] También hay sistemas distribuidos en los que los procesos son anónimos, es decir, no tienen un identificador. Sin embargo, para este trabajo, nuestro interés sólo recae en sistemas con identificadores de proceso.

Definición 1.2. Una *configuración inicial* es una configuración en donde todos los estados son estados iniciales de algún proceso. Análogamente, una *configuración final* es una configuración en donde todos los estados son finales.

Los autores en [1] definen una ejecución del sistema distribuido, en términos de configuraciones, tal como en la definición 1.3.

Definición 1.3. [Sección 4.1.3 de [1]] Una *ejecución* de un sistema distribuido, es una secuencia alternante de configuraciones y conjuntos de nombres de procesos:

$$C_0, S_0, C_1, S_1, \ldots, S_{r-1}, C_r$$

tales que:

- C_0 es la configuración inicial, y
- S_k es el conjunto de nombres de proceso para los cuales hay un cambio de estado entre la configuración C_i y la siguiente C_{i+1}.

Nos referimos a cada tripleta C_i, S_i, C_{i+1}, como un *paso concurrente*. Si $j \in S_i$, decimos que j *ejecuta un paso*.

Como lo veremos en los capítulos 2 y 3, la manera de especificar un problema distribuido, en el marco de la topología combinatoria, es a través de la caracterización, utilizando espacios topológicos, de configuraciones iniciales y finales. Por esto es que pensar en una ejecución en términos de configuraciones nos será de gran utilidad. Sin embargo, no utilizaremos la definición 1.3 de [1] y optaremos por una definición de ejecución distinta. Aunque no dejaremos de lado las configuraciones, pensar en una ejecución en términos de objetos y la invocación de sus métodos por los procesos, se adecúa mejor a nuestros intereses.

La definición 1.4 de una ejecución, tal y como se utiliza en [2], se define en términos de objetos y procesos. Denotamos a la invocación de un método con $\langle x.m(a^*), A\rangle$, donde x es un objeto, m el nombre del método, a^* una secuencia de parámetros y A un proceso o hilo. La respuesta a un método la denotamos con $\langle x : t(r^*), A\rangle$, donde t es el valor "ok" o el nombre de una excepción y r^* es una secuencia de valores de respuesta.

Definición 1.4. [Sección 3.6 de [2]] Una *ejecución* o *historia* H de un sistema concurrente, es una secuencia de eventos (invocaciones a métodos y respuestas).

La definición 1.4, aunque menos general que la definición 1.3, permite tener una noción de implementación precisa para los objetos concurrentes, a través de la linealizabilidad y las especificaciones secuenciales.

En la figura 1.3 podemos ver un ejemplo de una ejecución H en la que dos procesos a y b, invocan a los métodos *push* y *pop* de dos pilas p y q.

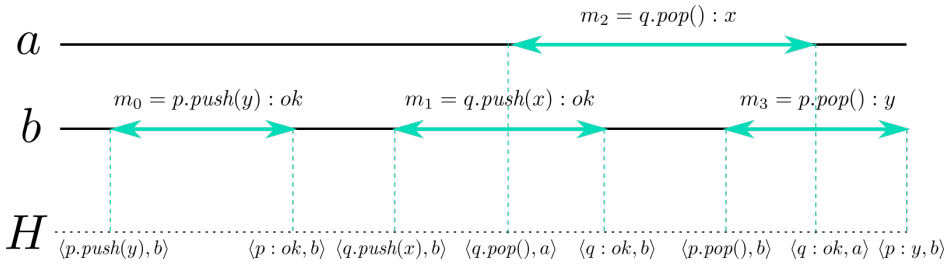

Figura 1.3: Una ejecución con dos procesos a y b y dos pilas p y q.

A veces nos fijaremos en sólo una parte de toda la ejecución.

Definición 1.5. Una *subejecución* o *subhistoria* de H, es una subsecuencia de eventos de H.

Cada subejecución sigue siendo, por si misma, una ejecución. Además, para cada invocación en H, se espera una respuesta que tenga sentido.

Definición 1.6. Decimos que una respuesta *corresponde* a una invocación, si ambas son realizadas por el mismo proceso y sobre el mismo objeto.

Definición 1.7. Una *llamada* a un método en una historia H, es un par que consiste de una invocación y la siguiente respuesta correspondiente en H.

Sin embargo, no siempre se tiene una respuesta y tenemos que distinguir entre las invocaciones con respuesta y sin ella.

Definición 1.8. Decimos que una invocación está *pendiente* en una historia H, si no hay una respuesta correspondiente (en H).

A veces nos conviene ignorar o no tener invocaciones pendientes.

Definición 1.9. Una *extensión* de H, es una historia construida al concatenar respuestas a cero o más invocaciones pendientes de H.

Definición 1.10. Denotamos con $complete(H)$ a la subejecución de H que no contiene ninguna invocación pendiente.

En el ejemplo de la figura 1.3, $complete(H) = H$. A veces nos fijaremos en ejecuciones hechas por un solo proceso o sobre un determinado objeto.

Definición 1.11. Denotamos con H_α a la subejecución en H donde las invocaciones y respuestas son hechas por o sobre α. Donde α puede ser un proceso o un objeto.

Por ejemplo, considerando la figura 1.3, la ejecución del proceso a en H es:

$$H_a = \langle q.pop(), a\rangle \ \langle q : ok, a\rangle$$

mientras que la ejecución sobre el objeto q en H es:

$$H_q = \langle q.push(x), b\rangle \ \langle q.pop(), a\rangle \ \langle q : ok, b\rangle \ \langle q : ok, a\rangle$$

Las ejecuciones en las que ninguna llamada a método ocurre mientras alguna otra está pendiente (que no se traslapan en el tiempo), son relevantes y tenemos que distinguirlas.

Definición 1.12. Una ejecución H es *secuencial*, si el primer evento es una invocación y cada invocación, excepto posiblemente la última, es seguida inmediatamente por su respuesta correspondiente.

Considerando la figura 1.3, H no es secuencial ya que la invocación $\langle q.push(x), b\rangle$, es seguida por la invocación $\langle q.pop(), a\rangle$. De la misma manera, H_q no es secuencial mientras que H_a, H_b y H_p, sí lo son.

Con lo anterior, podemos dar un criterio de equivalencia para ejecuciones.

Definición 1.13. Dos ejecuciones H y H', son *equivalentes* ($H \simeq H'$) si para cada proceso a, $H_a = H'_a$.

Tenemos que fijarnos en las ejecuciones que tienen sentido.

Definición 1.14. Una ejecución H está *bien formada* si para cada proceso a, H_a es secuencial.

En nuestro ejemplo, H está bien formada ya que H_a y H_b, son secuenciales. Las subejecuciones de una ejecución bien formada y realizadas por un solo proceso, siempre son secuenciales, pero las subejecuciones sobre un objeto, no necesariamente lo son.

Definición 1.15. Una *especificación secuencial* de un objeto, es el conjunto de todas sus ejecuciones secuenciales, cerrado bajo prefijos.

Si una ejecución X está en la especificación, también lo está cualquier prefijo de X. Decimos que un objeto es *secuencial* si tiene una especificación secuencial.

Vamos a suponer que tenemos una forma efectiva de reconocer si la historia de un objeto secuencial α es o no *legal* o *válida* para la clase de α.

Por ejemplo, podemos caracterizar la clase de una pila secuencial, caracterizando al conjunto de ejecuciones secuenciales válidas para la pila. Esto lo logramos especificando cómo se modifica el estado al ejecutar cada uno de sus métodos: después de $push(x)$, la pila tendrá en la "cabeza" el valor x y después de $pop()$ la pila regresa el valor que tenga en la cabeza o un valor nulo si está vacía. De esta manera, podemos tomar una historia de una pila secuencial y verificar de alguna forma, que cada llamada a método cumple con esta caracterización.

Definición 1.16. Una ejecución secuencial H es *legal* o *válida* si toda subejecución sobre un objeto α, es legal para α.

Dado que cada llamada a método ocurre en un intervalo de tiempo, podemos determinar un orden entre ellas, comparando el tiempo en el que ocurre cada invocación o cada respuesta. Para esto, como herramienta formal de estudio, utilizaremos algunos resultados de relaciones sobre conjuntos.

Definición 1.17. Un *orden parcial estricto* R para un conjunto X, es una relación

- no reflexiva: $(x, x) \notin R$
- transitiva: $(x, y), (y, z) \in R \Rightarrow (x, z) \in R$.

donde $x, y, z \in X$.

Definición 1.18. Un orden *total* estricto R (para X) es un orden parcial estricto tal que para cada par distinto x, y: $(x, y) \in R$ o $(y, x) \in R$.

En un orden parcial hay elementos que no se pueden comparar, es decir, no están en la relación. En un orden total, cualesquiera dos elementos pueden ser comparados. Cada orden parcial guarda alguna relación con un orden total.

Lema 1.1 (Hecho 3.6.1 de [2])**.** *Si $\prec$ es un orden parcial en X, entonces existe un orden total $<$ en X, tal que si $x \prec y$, entonces $x < y$.*

Con esto podemos describir el orden en el tiempo entre llamadas a métodos en una ejecución.

Definición 1.19. La llamada a un método m_0 *precede* a la llamada a m_1, en una ejecución H, si m_0 termina antes de que m_1 empiece, y lo denotamos con $m_0 \prec m_1$.

En el ejemplo de la figura 1.3, $m_0 \prec m_1, m_0 \prec m_2$ y $m_0 \prec m_3$. Sin embargo, m_1 y m_2 no están en la relación $\prec$. Las llamadas que se traslapan en el tiempo no están en $\prec$. Por esto, $\prec$ es un orden parcial y si H es secuencial, entonces $\prec$ es total y lo denotamos con $<$.

Con lo anterior, tenemos lo necesario para definir la noción de linealizabilidad, la cual, caracteriza una ejecución concurrente de un objeto, en términos de una ejecución secuencial equivalente.

Definición 1.20. Una ejecución H es *linealizable* si tiene una extensión H' y existe una ejecución secuencial legal S, tales que:

- $complete(H')$ es equivalente a S, y
- si $m_0 \prec_H m_1$ en H, entonces $m_0 \prec_S m_1$ en S.

Decimos que S es una *linealización* de H (H puede tener más de una linealización). De manera intuitiva, la extensión de H a H' captura la idea de que cada llamada pendiente ha hecho efecto. La existencia de S nos dice que cada llamada a un método tiene efecto de manera instantánea en algún punto entre su invocación y su respuesta. Esto permite determinar un orden total, es decir, describir una ejecución secuencial. La segunda condición asegura que el orden existente entre llamadas, se preserve en S.

La linealizabilidad tiene una propiedad importante con respecto a cada prefijo de una ejecución.

Lema 1.2 (Teorema 3.6.1 de [2])**.** *Si H es linealizable, entonces todo prefijo H' de H, es linealizable.*

Si bien la linealizabilidad permite hablar del estado de un objeto, hay problemas distribuidos que no pueden ser expresados como objetos secuenciales, por ejemplo, el immediate-snapshot. Además, hay problemas distribuidos que son expresados con mayor naturalidad en términos de entradas y salidas (configuraciones iniciales y finales), conocidos como *tareas* y que no son objetos secuenciales. Las tareas permiten, de manera relativamente sencilla, utilizar herramientas topológicas para su estudio. En la siguiente sección, establecemos las bases para hacer uso de dichas herramientas y poder extenderlas para el estudio de objetos secuenciales y linealizables.

1.3. Topología combinatoria

La *topología* es una rama de las matemáticas, encargada de la distinción de las propiedades *esenciales* y *no esenciales* de los espacios. Las propiedades esenciales son aquellas que persisten cuando el espacio es sujeto a transformaciones continuas.

Las distintas ramas de la topología difieren (de alguna manera) en cómo se representan los espacios y las transformaciones continuas que preservan sus propiedades esenciales.

La *topología combinatoria* provee una representación de espacios para los que las propiedades esenciales, se caracterizan contando los elementos simples que los forman. A estos elementos simples los llamamos *vértices*.

1.3.1. Complejos simpliciales abstractos

La noción de *complejo simplicial abstracto* [1], o simplemente complejo simplicial, provee un vínculo entre la combinatoria y la topología. Cada complejo simplicial puede verse como un objeto continuo: un espacio topológico. También puede verse como un objeto puramente combinatorio: una colección de conjuntos cerrados bajo contención; ya que sus propiedades topológicas, a menudo, se relacionan con propiedades combinatorias. Los complejos simpliciales son elementos centrales para el estudio de sistemas distribuidos desde la topología combinatoria.

Un complejo simplicial A es un conjunto cerrado bajo contención, de subconjuntos de un conjunto finito de vértices S. Es decir, cada elemento X de A es un subconjunto de S y cada subconjunto de X también está en A.

A cada subconjunto de A cerrado bajo contención, lo llamamos *subcomplejo simplicial*. Un subcomplejo simplicial es también un complejo simplicial.

A cada elemento de A lo llamamos *simplejo*. Hay que destacar que A es un conjunto de subconjuntos de S, mientras que un simplejo de A es un subconjunto de S. Un simplejo $\sigma \in A$ tiene dimensión $|\sigma| - 1$. A un simplejo de dimensión n lo llamamos *n-simplejo* o simplejo *n-dimensional*.

Un complejo simplical es *n-dimensional*, si la dimensión más grande de sus simplejos es igual a n, y es *puro n-dimensional*, si todos sus simplejos de mayor dimensión son n-dimensionales.

Usualmente, nos referiremos a los 0-simplejos como vértices, a los 1-simplejos como aristas, a los 2-simplejos como triángulos y a los 3-simplejos como tetraedros. Esta nomenclatura está asociada la representación gráfica de los distintos n-simplejos, ilustrada en la figura 1.4.

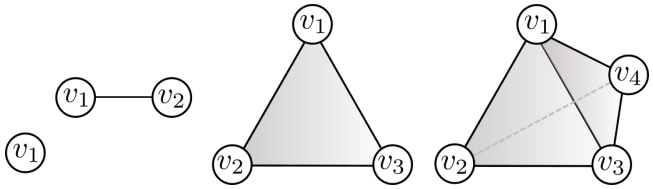

Figura 1.4: Un 0-simplejo, 1-simplejo, 2-simplejo y 3-simplejo, respectivamente.

Decimos que un simplejo τ es una *cara* de un simplejo σ, si τ es un subconjunto de σ. Es decir, cada simplejo es una cara de los simplejos de mayor dimensión a los que pertenece. Por ejemplo, en la figura 1.5, cada vértice es una cara del triángulo y de las aristas a las que pertenece. Así mismo, cada arista es una cara del triángulo.

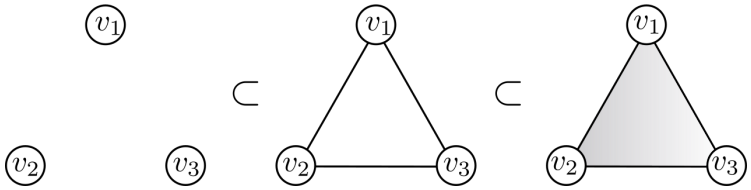

Figura 1.5: Cerradura bajo contención de simplejos de un complejo simplicial. Un simplejo es una cara de los simplejos de mayor dimensión a los que pertenece.

Utilizaremos letras minúsculas (x, y, z) para denotar vértices, letras mayúsculas (A, B, C) para denotar complejos simpliciales y letras griegas (σ, α, τ) para denotar simplejos.

Como lo veremos en los capítulos 2 y 3, los complejos simpliciales son centrales en el estudio de sistemas distribuidos. A groso modo, los vértices representarán procesos. Los simplejos de mayor dimensión, representarán ejecuciones entre procesos o dicho de otro modo, configuraciones de ejecuciones de un sistema distribuido.

Por otro lado, los complejos simpliciales son representaciones de espacios topológicos con propiedades esenciales. Podemos definir transformaciones entre distintos complejos simpliciales buscando preservar ciertas propiedades.

1.3.2. Mapeos simpliciales

Los *mapeos simpliciales* [1] nos permiten establecer una equivalencia entre complejos simpliciales a través de la estructura de sus simplejos. Un mapeo simplicial entre dos complejos simpliciales A y B, lleva cada vértice de A a un vértice de B. Los mapeos simpliciales preservan estructura, es decir, la imagen de los vértices de

un simplejo en A, forman un simplejo en B. La imagen 1.6 ilustra una parte de un mapeo simplicial φ de A a B.

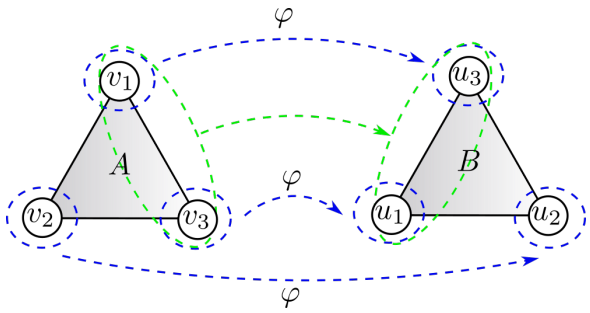

Figura 1.6: Un mapeo simplicial φ de un complejo A, a un complejo B, con $\varphi(v_1) = u_3$, $\varphi(v_2) = u_2$ y $\varphi(v_3) = u_1$. El mapeo de la arista v_1v_3 forma la arista u_1u_3.

Los vértices v_1, v_2 y v_3 de A, son mapeados a los vértices u_3, u_2 y u_1 de B, respectivamente. Para cada par de vértices (que forman una arista) en A, sus mapeos correspondientes, forman una arista en B. De la misma manera, el mapeo de los tres vértices en A, forman un triángulo en B. Dos simplejos son, en algún sentido, equivalentes, si podemos definir un mapeo simplicial entre ellos. Es importante aclarar que un mapeo simplicial, no necesariamente preserva la dimensión de los simplejos. En la figura 1.7, el mapeo simplicial φ, no preserva la dimensión de A.

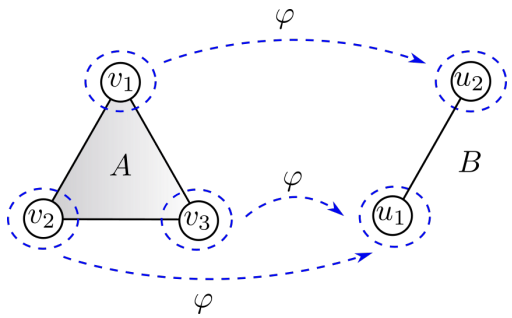

Figura 1.7: Un mapeo simplicial φ de un complejo A, a un complejo B, con $\varphi(v_1) = u_2$ y $\varphi(v_2) = \varphi(v_3) = u_1$.

El vértice v_1 es mapeado al vértice u_2, mientras que los vértices v_2 y v_3 son mapeados al vértice u_1. El mapeo φ sigue conservando la estructura. Para cada par de vértices (que forman una arista) en A, sus mapeos correspondientes, forman un simplejo en B. Por ejemplo, la arista v_2v_3 es mapeada al vértice u_1, mientras que el

triángulo es mapeado a la arista u_1u_2. Decimos que el mapeo φ, *colapsa* el triángulo en una arista.

Los mapeos simpliciales son la contraparte de los mapeos continuos en la topología clásica. De alguna manera, son transformaciones continuas entre complejos simpliciales. También nos interesa utilizar transformaciones más generales.

1.3.3. Mapeos portadores

Los *mapeos portadores* [1] nos ayudan a definir transformaciones más generales entre complejos simpliciales y son importantes para la aplicación de la topología en el estudio de sistemas distribuidos. Un mapeo portador Φ, entre dos complejos simpliciales A y B, lleva cada simplejo de A, a un subcomplejo de B, conservando los patrones de contención de los simplejos de A en los subcomplejos de B. En la figura 1.8, podemos ver un ejemplo de un mapeo portador Φ, entre dos complejos simpliciales A y B.

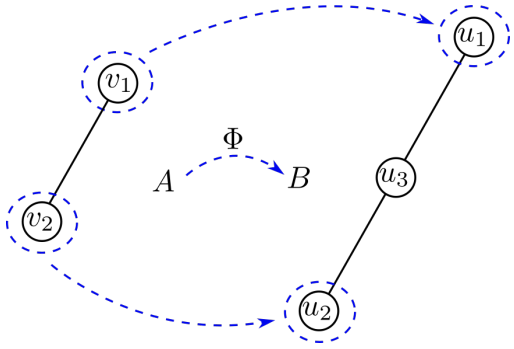

Figura 1.8: Un mapeo portador Φ entre dos complejos A y B. Φ mapea los vértices v_1 y v_2 de A, en cada vértice u_1 y u_2 de B, respectivamente. También mapea el simplejo 1-dimensional de A en B.

El mapeo Φ lleva cada vértice de A a un vértice (visto como subcomplejo) de los extremos de B. La arista en A es mapeada directamente a B. Las caras de la arista v_1v_2 son los subcomplejos $\Phi(v_1)$ y $\Phi(v_2)$ de B, conservando el patrón de contención.

Los simplejos representarán configuraciones, inciales y finales, de una ejecución de un sistema para un problema distribuido. Los mapeos portadores determinarán qué configuraciones finales son alcanzables en una ejecución, desde cada configuración inicial.

Por ejemplo, la figura 1.9 muestra una representación de un problema distribuido. La única configuración inicial, representada por el 1-simplejo de A, está dada por los

estados iniciales $(a, 1)$ y $(b, 0)$, de los procesos a y b, respectivamente. De la misma manera, la única configuración final, representada por el 1-simplejo de B, está dada por los estados finales $(a, 1)$ y $(b, 1)$, de los procesos a y b, respectivamente.

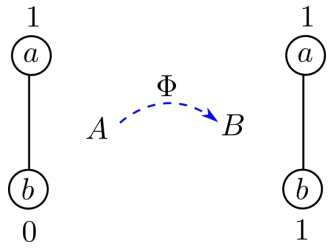

Figura 1.9: Representación de la configuración inicial y la configuración final de un problema distribuido. El id de proceso de cada estado se representa en un vértice y la vista correspondiente, en una etiqueta a su lado.

Los estados de cada configuración son representados por un vértice con el id del proceso y una etiqueta con la vista. La configuración inicial nos dice que el proceso a inicia el cómputo con una vista (valor inicial) igual a 1 y el proceso b con una vista igual a 0. El mapeo Φ especifica que la configuración final en donde ambos procesos tienen el valor 1 en su vista, es alcanzable al final de la ejecución.

Recordemos que en cada configuración, sólo puede haber un estado por cada proceso. Por ejemplo, no puede haber una configuración con los estados $(a, 0)$ y $(a, 1)$. Sin embargo, como son dos estados distintos, al verlos como vértices, no hay una restricción que nos impida formar con ellos, una arista. Dicho de otro modo, no cualquier simplejo puede representar una configuración.

1.3.4. Complejos simpliciales cromáticos

Una *m-coloración*, o simplemente *coloración*, de un complejo simplicial A, es una función inyectiva que mapea cada vértice de A, en un elemento de un conjunto M, de cardinalidad m. Por ser inyectiva, dos vértices distintos son mapeados a dos elementos distintos. Una coloración corresponde a un etiquetado de los vértices de un complejo simplicial, tal que dos vértices vecinos (vértices conectados por una arista) no tienen la misma etiqueta o color.

Un *complejo simplicial cromático* [1] es una pareja (A, χ), donde A es un complejo simplicial y χ una coloración. A cada simplejo de un complejo simplicial cromático, le decimos *simplejo cromático*.

Como queremos representar configuraciones de ejecuciones de un sistema distribuido, la imagen de χ será un subconjunto de un conjunto de nombres de proceso Π

(sección 1.2.1). De esta manera, al representar cada estado de proceso con un vértice, el id corresponde a un color.

En adelante, sólo consideraremos complejos simpliciales cromáticos y siempre que quede claro, a cada complejo simplicial cromático (A, χ) le diremos simplemente, complejo simplicial A.

Notemos que los complejos simpliciales de la figura 1.9, son cromáticos. La figura 1.10 muestra varios ejemplos de complejos simpliciales cromáticos.

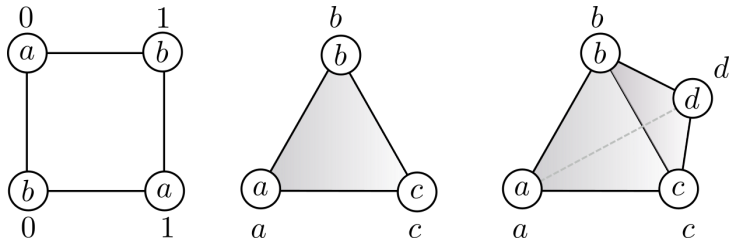

Figura 1.10: Representación de distintos complejos simpliciales cromáticos. La vista de cada proceso se representa con una etiqueta al lado cada vértice correspondiente.

Debemos asegurarnos que las transformaciones produzcan complejos simpliciales cromáticos. Un mapeo simplicial es *cromático* si a cada vértice lo mapea a un vértice con el mismo color. Análogamente, un mapeo portador Φ es *cromático*, si para cada simplejo σ, el conjunto de colores de los vértices de σ, es igual al conjunto de colores de los vértices de $\Phi(\sigma)$. Por ejemplo, en la figura 1.9, Φ es cromático. En adelante, solamente consideraremos mapeos simpliciales y portadores, cromáticos.

1.3.5. Subdivisión cromática estándar

Una *subdivisión* es una transformación entre dos complejos simpliciales A y B, que se obtiene al "dividir" los simplejos de A en simplejos más pequeños, obteniendo al complejo simplicial B.

La *subdivisión cromática estándar* [1], es un elemento central para el análisis en la solubilidad de problemas en el modelo computacional de escritura-lectura, el cual estudiaremos a detalle en el capítulo 3. Dado un complejo simplicial cromático A, denotamos a su subdivisión cromática con $\mathrm{Ch}(A)$. Ch es un mapeo portador cromático entre los complejos simpliciales A y $\mathrm{Ch}(A)$, que subdivide a cada simplejo de A de la siguiente manera:

- Los 0-simplejos se mapean en 0-simplejos, conservando el color.

- A cada 1-simplejo σ, se le agregan dos vértices, de manera que se formen tres nuevos 1-simplejos cromáticos, con el mismo conjunto de colores de σ. El producto también es un complejo cromático puro 1-dimensional. La figura 1.11 muestra la subdivisión cromática estándar de un 1-simplejo.

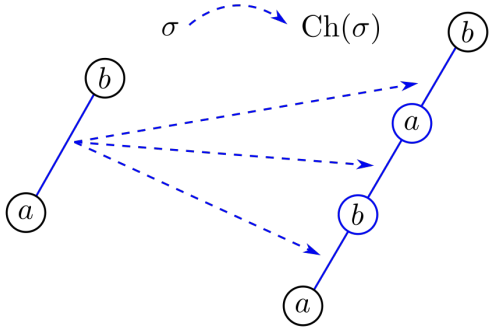

Figura 1.11: Subdivisión cromática estándar de un 1-simplejo σ.

- Para subdividir a cada 2-simplejo σ, primero se subdividen sus aristas y después se agrega un 2-simplejo con los mismos colores que σ, los vértices introducidos en las aristas, se unen cromáticamente al nuevo triángulo introducido. El producto también es un complejo cromático puro 2-dimensional. La figura 1.12 muestra la subdivisión cromática estándar de un 2-simplejo.

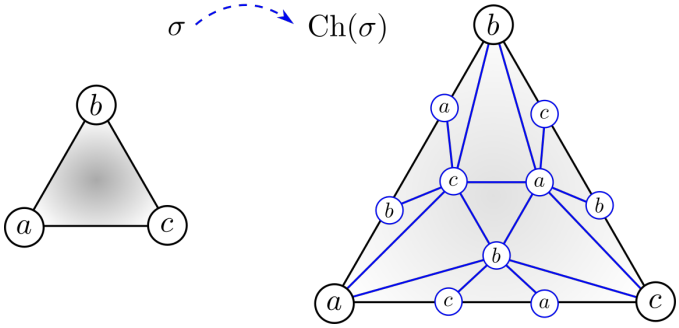

Figura 1.12: Subdivisión cromática estándar de un 2-simplejo σ.

- En general, para subdividir un n-simplejo σ, de manera inductiva, sus caras son subdivididas y unidas cromáticamente, a un nuevo n-simplejo agregado al interior. El producto es un complejo cromático puro n-dimensional.

Como lo veremos en el capítulo 3, la subdivisión cromática estándar caracteriza a todas las posibles ejecuciones concurrentes de immediate-snapshot, fundamentales para el estudio de los problemas distribuidos en el modelo de escritura-lectura por capas.

1.3.6. Definiciones formales

Dado un conjunto S y una familia A de subconjuntos finitos de S.

Definición 1.21. Decimos que A es un *complejo simplicial abstracto* (CSA) en S, si se satisface lo siguiente:

- Si $X \in A$, y $Y \subseteq X$, entonces $Y \in A$.
- $\{v\} \in A$ para todo $v \in S$.

La primera condición es la cerradura bajo contención de los elementos de A. Un elemento de S es un *vértice* y un elemento de A (subconjunto de S) es un *simplejo*.

Definición 1.22. Al conjunto de todos los vértices de A lo denotamos con $V(A)$.

Definición 1.23. Un simplejo $\sigma \in A$ se dice que tiene *dimensión* $|\sigma| - 1$.

Definición 1.24. Un simplejo σ de dimensión n, es un *n-simplejo* y lo denotamos con σ^n.

En particular, a cada 0-simplejo de A, también le llamamos vértice.

Definición 1.25. Un simplejo τ es una *cara* del simplejo σ, si $\tau \subseteq \sigma$.

Todo simplejo es una cara de los simplejos a los que pertenece, incluyéndose.

Definición 1.26. Un simplejo τ es una *cara propia* del simplejo σ, si $\tau \subset \sigma$.

Todo simplejo es una cara propia de los simplejos de mayor dimensión a los que pertenece. Las caras y las caras propias de un simplejo σ, corresponden a subconjuntos y subconjuntos propios de σ, respectivamente.

Definición 1.27. Un simplejo σ de un complejo A, es una *faceta* si no es una cara propia de algún otro simplejo de A.

Dicho de otro modo, cada faceta es un simplejo que no está contenido en otro simplejo de A. La dimensión de un complejo simplicial está determinada por la dimensión de sus facetas.

Definición 1.28. La *dimensión* de un complejo es la dimensión más grande de sus facetas.

Definición 1.29. Un complejo es *puro* si todas sus facetas tienen la misma dimensión.

Si cada faceta tiene dimensión n, decimos que es puro n-dimensional. La figura 1.13 muestra un complejo A puro 1-dimensional, un complejo B puro 2-dimensional, y un complejo C 2-dimensional que no es puro.

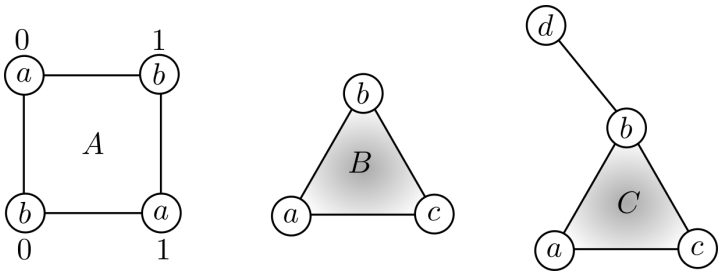

Figura 1.13: Un complejo A puro 1-dimensional, un complejo B puro 2-dimensional, y un complejo C 2-dimensional que no es puro.

Definición 1.30. Un complejo B es un *subcomplejo* de un complejo simplicial A, si todo simplejo de B también es un simplejo de A.

Todo subcomplejo también es un complejo simplicial.

Caracterizar un camino, en el sentido usual, entre dos vértices, nos será de gran utilidad en el capítulo 4.

Definición 1.31. Un *camino* (de aristas) entre dos vértices u y v, de un complejo simplicial A, es una secuencia de vértices $u = v_0, v_1, \ldots, v_k = v$, tal que cada par $\{v_i, v_{i+1}\}$ es una arista de A, con $0 \leq i < k$.

Un camino es *simple* si todos sus vértices son distintos. En la figura 1.13, los vértices de B forman un camino, mientras que los vértices a, b y d de C, forman un camino simple.

Mapeos simpliciales

Definición 1.32. Dados dos complejos simpliciales A y B, un *mapeo de vértices* $\mu : V(A) \to V(B)$, mapea cada vértice de A a un vértice de B.

En topología nos interesan aquellos mapeos que preservan estructura.

Definición 1.33. Para dos complejos simpliciales A y B, un mapeo de vértices μ es llamado *mapeo simplicial*, si mapea simplejos en simplejos; esto es, si $\{s_0, \ldots, s_n\}$ es un simplejo de A, entonces $\{\mu(s_0), \ldots, \mu(s_n)\}$ es un simplejo de B.

El simplejo $\mu(\sigma)$ puede tener menor dimensión que σ. Extendemos de manera natural un mapeo simplicial μ sobre un simplejo $\sigma \in A$ tal que $\sigma = \{s_0, \ldots, s_n\}$, con $\mu(\sigma) = \{\mu(s_0), \ldots, \mu(s_n)\}$. Con los mapeos simpliciales podemos definir una equivalencia entre complejos simpliciales.

Definición 1.34. Dos complejos simpliciales A y B son *isomorfos* y los denotamos con $A \cong B$, si existen dos mapeos simpliciales $\varphi : A \to B$ y $\psi : B \to A$ tales que para cada vértice a de A, $a = \psi(\varphi(a))$ y para cada vértice b de B, $b = \varphi(\psi(b))$.

Es importante notar que dos complejos simpliciales isomorfos tienen estructuras idénticas.

La siguiente proposición será de gran utilidad en el capítulo 4.

Proposición 1.1. *Dos caminos simples son isomorfos si tienen el mismo número de vértices.*

Demostración. Sea el camino c_1 con lo vértices $v_1, \ldots, v_k$ y el camino c_2 con los vértices $u_1, \ldots, u_k$. Basta con definir, para $1 \leq i \leq k$, los mapeos simpliciales:

- φ de c_1 a c_2 tal que $\varphi(v_i) = u_i$
- ψ de c_2 a c_1 tal que $\psi(u_i) = v_i$.

Con lo que tenemos $v_i = \psi(\varphi(v_i))$ y $u_i = \varphi(\psi(u_i))$.

□

También nos interesan los mapeos simpliciales que preservan dimensión.

Definición 1.35. Dados dos complejos simpliciales A y B. Un mapeo simplicial $\varphi : A \to B$, es *rígido* si para cada $\sigma \in A$, $|\varphi(\sigma)| = |\sigma|$.

Es decir, la imagen de cada simplejo σ tiene la misma dimensión que σ.

Por último, es importante notar que la composición de mapeos simpliciales también es un mapeo simplicial, y si los mapeos son rígidos, también lo es su composición [1][Secc. 3.2.1].

Mapeos portadores

Definición 1.36. Dados dos complejos simpliciales A y B. Un *mapeo portador* $\Phi : A \to B$, mapea cada simplejo de σ de A a subcomplejos $\Phi(\sigma)$ de B, tal que para todo $\sigma, \tau \in A$, si $\sigma \subseteq \tau$, entonces $\Phi(\sigma) \subseteq \Phi(\tau)$.

Dado que un mapeo portador Φ mapea simplejos de A, a subcomplejos de B, lo denotamos con $\Phi : A \to 2^B$.

Definición 1.37. Un mapeo portador Φ es *monotónico* si se cumple:

$$\Phi(\sigma \cap \tau) \subseteq \Phi(\sigma) \cap \Phi(\tau)$$

para todo $\sigma, \tau \in A$.

La monotonicidad nos asegura que el patrón de inclusión de cada simplejo de $\sigma \in A$ (se preserva) es igual al patrón de inclusión de los subcomplejos de $\Phi(\sigma)$.

Podemos extender el mapeo Φ sobre un subcomplejo $K \subseteq A$ de la siguiente manera: $\Phi(K) := \cup_{\sigma \in K} \Phi(\sigma)$.

Las siguientes son algunas propiedades, que nos interesan, de los mapeos portadores. Hay mapeos portadores que preservan dimensión.

Supongamos que tenemos dos complejos simpliciales A y B y un mapeo portador $\Phi : A \to 2^B$.

Definición 1.38. El mapeo Φ es *rígido* si para cada simplejo $\sigma \in A$ de dimensión d, el subcomplejo $\Phi(\sigma)$ es puro de dimensión d.

Definición 1.39. El mapeo Φ es *estricto* si se da la igualdad para la ecuación de la definición 1.37:

$$\Phi(\sigma \cap \tau) = \Phi(\sigma) \cap \Phi(\tau)$$

para todo $\sigma, \tau \in A$.

Esta propiedad nos asegura la existencia del simplejo σ de la definición 1.40.

Definición 1.40. Dado un mapeo portador estricto $\Phi : A \to 2^B$. Para cada simplejo $\tau \in \Phi(A)$, hay un único simplejo $\sigma \in A$ de mínima dimensión, tal que $\tau \in \Phi(\sigma)$. A este simplejo σ, lo llamamos el *Portador* de τ o $\text{Carr}(\tau, \Phi(\sigma))$.

Si es claro del contexto, omitimos $\Phi(\sigma)$ en $\text{Carr}(\tau, \Phi(\sigma))$ y escribimos solamente $\text{Carr}(\tau)$.

Definición 1.41. Dados dos mapeos portadores $\Phi : A \to 2^B$ y $\Psi : A \to 2^B$ y un mapeo simplicial $\varphi : A \to B$, decimos que:

1. Ψ es *portado* por Φ, si $\Psi(\sigma) \subseteq \Phi(\sigma)$ para todo $\sigma \in A$ y lo denotamos con $\Psi \subseteq \Phi$.

2. φ es *portado* por Φ, si $\varphi(\sigma) \in \Phi(\sigma)$ para todo $\sigma \in A$.

Podemos componer mapeos portadores con mapeos simpliciales.

Definición 1.42. Dados los complejos simpliciales A, B, C y un mapeo portador Φ de A a B:

1. Si $\varphi : C \to A$ es un mapeo simplicial, definimos el mapeo portador $\Phi \circ \varphi$ de C a B de la manera usual, $(\Phi \circ \varphi)(\sigma) = \Phi(\varphi(\sigma))$, para todo $\sigma \in C$.

2. Si $\varphi : B \to C$ es un mapeo simplicial, definimos el mapeo portador $\varphi \circ \Phi$ de A a C de la manera usual, $(\varphi \circ \Phi)(\sigma) = \varphi(\Phi(\sigma))$, para todo $\sigma \in A$.

Notemos que $\varphi(\Phi(\sigma)) = \cup_{\tau \in \Phi(\sigma)} \varphi(\tau)$.

Componer un mapeo simplicial rígido con un mapeo portador rígido (por cualquier lado), produce un mapeo portador rígido [1][Secc. 3.4].

También podemos componer mapeos portadores entre sí.

Definición 1.43. Dados dos mapeos portadores $\Phi : A \to 2^B$ y $\Psi : B \to 2^C$ definimos la composición $\Psi \circ \Phi : A \to 2^C$ como $(\Psi \circ \Phi)(\sigma) = \Psi(\Phi(\sigma))$ para todo $\sigma \in A$, con $\Psi(\Phi(\sigma)) = \cup_{\tau \in \Phi(\sigma)} \Psi(\tau)$.

De esta composición se obtiene el siguiente resultado.

Proposición 1.2 (Proposición 3.4.6 de [1])**.** *Dados dos mapeos portadores $\Phi : A \to 2^B$ y $\Psi : B \to 2^C$:*

1. *Si Φ y Ψ son rígidos, también lo es $\Psi \circ \Phi$.*

2. *Si Φ y Ψ son estrictos, también lo es $\Psi \circ \Phi$.*

Complejos simpliciales cromáticos

Definición 1.44. Un *m-etiquetado* de un complejo A, es un mapeo que manda cada vértice de A, a un elemento de algún dominio de cardinalidad m. Es decir, es un mapeo $\varphi : V(A) \to M$, donde $|M| = m$.

Normalmente, $M = \{1, \ldots, m\}$. Podemos extender el etiquetado φ sobre un simplejo $\sigma \in A$ de manera natural sobre los vértices de σ con $\varphi(\sigma) = \{m | m = \varphi(v), v \in\}$.

Este etiquetado determina una coloración.

Definición 1.45. Una *m-coloración* de un complejo n-dimensional A, es un m-etiquetado $\chi : V(A) \to M$ tal que χ es inyectiva sobre los vértices de todo simplejo de A, es decir, para $v_0, v_1 \in \sigma$ distintos, $\chi(v_0) \neq \chi(v_1)$.

Utilizaremos al conjunto de nombres de procesos Π, en un sistema distribuido, como el conjunto de colores para χ.

Definición 1.46. A un complejo simplicial A junto con una coloración χ, lo llamamos *complejo simplicial cromático* y lo denotamos con (A, χ).

Siempre que sea claro, no referiremos a un complejo simplicial cromático (A, χ), sólo con A. Una nueva propiedad de los mapeos simpliciales, surge cuando se consideran coloraciones.

Definición 1.47. Dados dos complejos simpliciales m-cromáticos (A, χ_A) y (B, χ_B), decimos que un mapeo simplicial $\varphi : A \to B$ es *cromático*, si para todo vértice $v \in A$, $\chi_A(v) = \chi_B(\varphi(v))$.

En otras palabras, φ es cromático si mapea a cada vértice de A a un vértice de B con el mismo color. También decimos que φ *preserva colores* o *etiquetas*. A menos que lo especifiquemos, en adelante, todos los mapeos simpliciales serán cromáticos.

De manera análoga para los mapeos portadores.

Definición 1.48. Dados dos complejos simpliciales cromáticos A, B y un mapeo portador $\Phi : A \to 2^B$, decimos que Φ es *cromático* si es rígido y para todo simplejo $\sigma \in A$, se cumple:

$$\chi_A(\sigma) = \chi_B(\Phi(\sigma)), \text{ donde } \chi_B(\Phi(\sigma)) = \{\chi_B(v) | v \in V(\Phi(\sigma))\}$$

Es decir, el conjunto de colores de los vértices de σ es igual al conjunto de colores del subcomplejo correspondiente $\Phi(\sigma)$. Además, como χ_A y χ_B son coloraciones y Φ es rígido, cada faceta de $\Phi(\sigma)$ contiene un vértice, por cada vértice de σ, con el mismo color.

Subdivisión cromática estándar

Definición 1.49. Sea (A, χ) un complejo simplicial cromático. Su *subdivisión cromática estándar* $\mathrm{Ch}(A)$, es el complejo simplicial para el cual, los vértices son de la forma (i, σ_i), con $i \in [n]$, σ_i una cara no vacía del simplejo σ, e $i \in [\chi(\sigma)]$, para todo $\sigma \in A$.

Siendo $[\chi(\sigma)]$ el conjunto de colores de los vértices de σ.

Una $(k+1)$-tupla $(\sigma_0, \ldots, \sigma_k)$, es un simplejo de $\mathrm{Ch}(A)$ si y sólo si:

- La tupla puede indexarse de tal forma que $\sigma_0 \subseteq \cdots \subseteq \sigma_k$.
- Para $0 \leq i, j \leq n$, si $i \in [\chi(\sigma_j)]$, entonces $\sigma_i \subseteq \sigma_j$.

Para que $\mathrm{Ch}(A)$ sea cromático, extendemos la coloración χ sobre $\mathrm{Ch}(A)$, con $\chi((i, \sigma)) = i$.

Lema 1.3 (Sección 3.6 de [1])**.** *Ch es un mapeo portador cromático estricto.*

Definición 1.50. La *subdivisión cromática estándar iterada* de A, para $k > 0$, es la composición de Ch consigo misma, k veces. Esto es:

$$\mathrm{Ch}^k(A) = \underbrace{\mathrm{Ch} \circ \cdots \circ \mathrm{Ch}}_{k\text{-veces}}(A)$$

La figura 1.14 ilustra la subdivisión cromática iterada de un 1-simplejo.

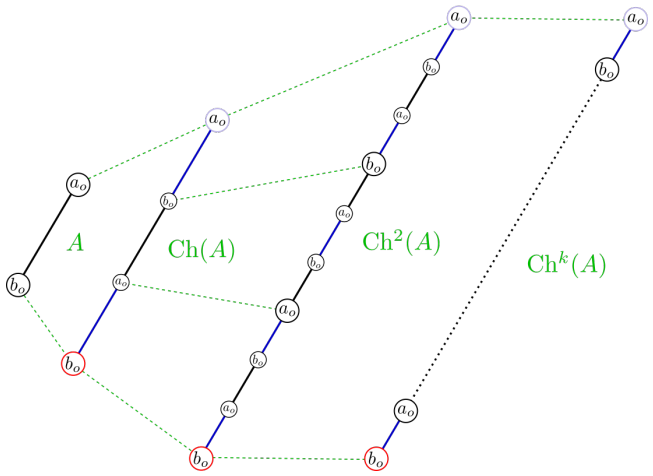

Figura 1.14: La subdivisión cromática iterada de un 1-simplejo.

Podemos construir un complejo simplicial a partir de otro complejo a través de las transformaciones determinadas por los mapeos portadores y simpliciales. También podemos construir un complejo simplicial, combinando más de un complejo, con algunas consideraciones. Las llamadas *Construcciones Estándar* [1] nos permiten construir, a partir de dos o más complejos simpliciales existentes, un nuevo complejo simplicial. Hay al menos tres construcciones relevantes: la Estrella, el Vínculo y la operación Join (distinguiéndola de la unión de conjuntos). Sin embargo, sólo nos interesa una.

1.3.7. Operación Join

La operación *Join* de complejos simpliciales, extiende la unión de conjuntos de forma natural. Definiremos esta operación utilizando el término en inglés para dejar claro que no es la unión usual de conjuntos, aunque en adelante nos referiremos a ella como la unión de complejos simpliciales. Dados dos complejos simpliciales A y B con sus conjuntos de vértices $V(A)$ y $V(B)$ disjuntos.

Definición 1.51. La operación *join* de A y B, denotada con $A * B$, es el complejo simplicial con el conjunto de vértices $V(A) \cup V(B)$ y cuyos simplejos son todas las uniones $\alpha \cup \beta$, para todo simplejo $\alpha \in A$ y todo simplejo $\beta \in B$.

Los simplejos α o β pueden ser vacíos. En particular, A y B son subcomplejos de $A * B$.

La figura 1.15 muestra la unión de algunos complejos simpliciales.

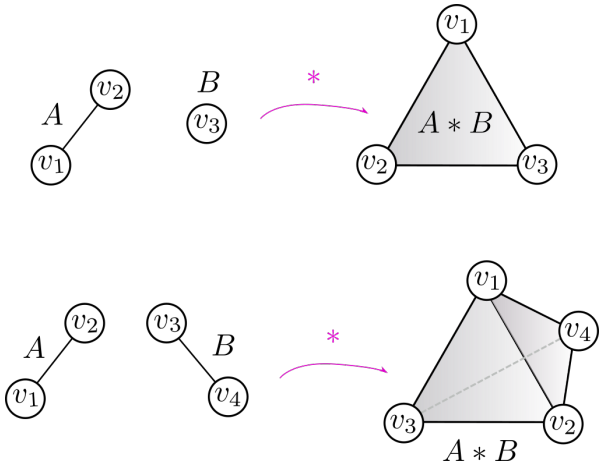

Figura 1.15: La unión de complejos. El primer ejemplo es la unión de un 1-simplejo con un 0-simplejo. El segundo ejemplo, un 2-simplejo con otro 2-simplejo.

Proposición 1.3 (Secc. 3.3.3 de [1])**.** *La operación Join es conmutativa y asociativa.*

Con lo anterior, tenemos las bases para caracterizar y estudiar a las pilas y colas concurrentes desde el enfoque topológico del cómputo distribuido.

1.4. Enfoque topológico del cómputo distribuido

El enfoque topológico en el computo distribuido se utiliza para obtener resultados de imposibilidad. La estrategia que se suele seguir, en general, consta de los siguientes

puntos:

1. Representar el problema como objetos topológicos: Especificar el problema en términos de un complejo simplicial cromático de entrada I, un complejo simplicial cromático de salida O, y un mapeo portador Δ de I a O, que especifica el problema.

2. Representar algoritmos como objetos topológicos: Tomar en cuenta el modelo de cómputo distribuido y especificar el protocolo en términos del complejo simplicial cromático de entrada I, un complejo simplicial cromático de protocolo P y un mapeo portador Ξ de I a P, que determina todas las ejecuciones.

3. Demostrar que los espacios P y O son incompatibles: Si el problema en cuestión puede solucionarse en el modelo considerado, entonces existe un mapeo simplicial cromático δ de P a O, con ciertas características. Utilizando resultados y herramientas topológicas, para demostrar P y O son incompatibles, basta ver que δ no existe.

Este enfoque topológico no se ha aplicado, en general, al estudio de objetos secuenciales, y mucho menos en casos particulares como las pilas y colas.

1.5. Resumen

Un objeto provee un conjunto de métodos que permiten manipular el estado interno del mismo. Un objeto secuencial, a diferencia de un objeto concurrente, se puede caracterizar formalmente en términos de precondiciones y postcondiciones de sus métodos. La linealizabilidad caracteriza el comportamiento concurrente y correcto de un objeto en términos de un comportamiento secuencial equivalente.

Hay problemas distribuidos que no pueden ser expresados como objetos secuenciales. También hay problemas distribuidos que son expresados con mayor naturalidad en términos de configuraciones iniciales y finales. Los complejos simpliciales y los mapeos simpliciales y portadores, permiten la especificación de problemas distribuidos, en el marco de la topología combinatoria. Esto abre la posibilidad de aplicar técnicas topológicas al estudio de objetos secuenciales. Este enfoque topológico no se ha aplicado en general a objetos secuenciales, y mucho menos en casos particulares como las pilas y colas.

Capítulo 2

Especificación de objetos concurrentes

Los objetos concurrentes usualmente se especifican en términos de objetos secuenciales. Las las *tareas distribuidas* [1, 4, 5] son una forma de representar un problema distribuido de manera estática, en términos de asignaciones de valores de entrada y salida válidos.

En este capítulo veremos cómo representar tareas en el marco de la topología combinatoria, mediante complejos simpliciales. Sin embargo, las tareas carecen del poder expresivo para representar a las pilas y las colas concurrentes. Veremos cómo las *tareas refinadas* [5] son una extensión de las tareas, con mayor poder expresivo, que nos permitirán representar a estos objetos. Además, daremos una noción de satisfacción por una especificación secuencial, para ambos tipos de tarea.

2.1. Tareas

Una *tarea* [1, 5, 4], es una manera estática de especificar un problema distribuido con una operación que puede ser invocada sólo una vez (*one-shot*), por cada proceso. De forma más concreta, una tarea es el equivalente, en forma distribuida, a una función definida por un conjunto de entradas a los procesos V_i y un conjunto de salidas legales V_o.

Las tareas son una descripción estática combinatoria, de un problema distribuido, expresada en términos de relaciones entre espacios topológicos que representan asignaciones de valores de entrada y salida.

En una tarea, cada proceso empieza con un valor de entrada en V_i, privado; y eventualmente tiene que decidir un valor de salida en V_o. Un proceso inicialmente,

no está enterado de los valores de entrada de otros procesos.

Consideremos una ejecución donde k procesos participan. Una familia de subconjuntos de un conjunto de vértices $\{(i_1, x_1), \ldots, (i_k, x_k)\}$, determina un *complejo simplicial de entrada* I, en donde cada $x_j \in V_i$ denota el *valor de entrada* del proceso con identificador i_j, esto es, el vértice (i_j, x_j) representa el estado inicial de i_j. A cada simplejo de I, le decimos *simplejo de entrada*.

De la misma forma, una familia de subconjuntos de un conjunto de vértices $\{(i_1, y_1), \ldots, (i_k, y_k)\}$, determina un *complejo simplicial de salida* O, en donde cada $y_j \in V_o$ denota el *valor de salida* del proceso con identificador i_j, esto es, el vértice (i_j, y_j) representa la decisión, o estado final, de i_j. A cada simplejo de O, le decimos *simplejo de salida*.

Decimos que cada simplejo de entrada representa una *configuración de entrada* y cada simplejo de salida representa una *configuración de salida*.

Todas las posibles configuraciones de salida que pueden ser alcanzadas, después de una ejecución, para alguna configuración de entrada determinada, están dadas por un mapeo portador cromático Δ. Dicho de otro modo, Δ determina todos los valores de salida válidos, de cada proceso, para una configuración de entrada determinada.

Una tarea es una tupla $\langle I, O, \Delta \rangle$ que representa de manera estática, la ejecución concurrente en la que cada proceso i_j ejecuta la operación $task(x_j)$, si el vértice (i_j, x_j) pertenece a I. Además al terminar la ejecución, el proceso i_j tiene el valor de salida y_j, si el vértice (i_j, y_j) pertenece a O.

Veamos dos ejemplos de problemas distribuidos que pueden ser modelados con tareas. El primero, el *Consenso binario*, es un problema fundamental en el cómputo distribuido. El segundo, el *Inmediate-snapshot*, es un elemento central en el modelo de cómputo de *escritura-lectura por capas* que veremos en el capítulo 3.

2.1.1. Tarea Consenso binario

En el consenso binario, n procesos tienen que ponerse de acuerdo en un valor en el conjunto de entradas $V_i = \{0, 1\}$, propuesto por alguno de ellos. Es decir, tienen que elegir un mismo valor de salida en $V_o = \{0, 1\}$. La figura 2.1 ilustra la tarea para el consenso binario entre dos procesos p y q.

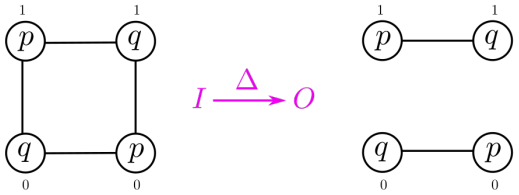

Figura 2.1: Tarea para el consenso binario entre los procesos p y q.

Tanto I como O, son cromáticos puros 1-dimensionales. En el complejo I, cada proceso puede iniciar con 0 o 1. En el complejo O, sólo hay dos configuraciones de salida válidas, una por cada valor que ambos procesos tienen que decidir.

La figura 2.2 ilustra cómo el mapeo Δ captura las salidas válidas para dos configuraciones (simplejos) de entrada: la primera, si p corrió con un valor inicial 1 y no recibió información de q, entonces está obligado a decidir 1; la segunda, si ambos procesos propusieron el valor 0, ambos deciden 0.

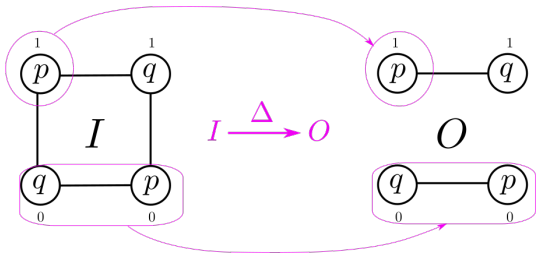

Figura 2.2: Dos asignaciones váidas para Δ. El mapeo portador Δ especifica, para cada asignación de valores de entrada, que asignación de valores de salida pueden escogerse.

En el caso de que ambos procesos propongan un valor distinto, las asignaciones de salida válidas, representan a todo el complejo O. Es importante notar, que no hay una configuración de salida (un simplejo) en la que ambos procesos deciden un valor distinto.

2.1.2. Tarea Inmediate-snapshot

En el *Inmediate-snapshot* [2, 5], n procesos escriben y leen de una memoria compartida, de manera *atómica* [2], en dos pasos contiguos. En el primer paso, un proceso escribe su vista en la memoria, posiblemente de forma concurrente con otros procesos. En el siguiente paso, toma un *snapshot* de toda la memoria (inmediatamente después del paso anterior), posiblemente de forma concurrente con otros procesos.

La figura2.3 ilustra la especificación de una tarea Inmediate-snapshot con tres procesos.

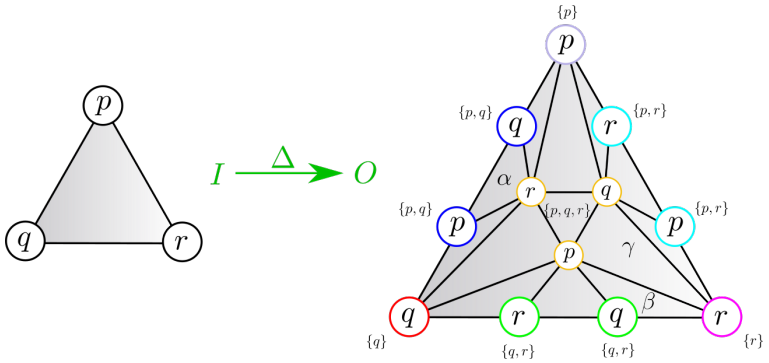

Figura 2.3: Representación de la tarea para el objeto *Inmediate-snapshot* y tres procesos p, q y r. Los vértices, en O, con la misma vista, indican que esos procesos corrieron de manera concurrente.

En el complejo de entrada I, cada vértice puede verse como un par $\{(i_j, i_j)\}$, es decir, el valor de entrada (implícito) de cada proceso es igual a su identificador, con lo que el conjunto de entradas válidas es $V_i = \{p, q, r\}$. I es un complejo simplicial cromático puro 2-dimensional.

Por otro lado, el complejo de salida O, también es un complejo simplicial cromático puro 2-dimensional. La coloración sigue correspondiendo a los identificadores de los procesos (y no al color de los vértices en la imagen). Cada vértice puede verse como un par $\{(i_j, \mathit{view}_j)\}$, en donde la componente view_j es la vista que corresponde a el contenido de la memoria del *snapshot* realizado por el proceso i_j. Por ejemplo, el vértice $\{(p, \{p, q\})\}$ nos dice que el proceso p, al ejecutar el *inmediate-snapshot*, escribió el valor p y leyó los valores p y q. El conjunto de valores de salida V_o, es el conjunto potencia, menos el vacío, del conjunto $\{p, q, r\}$.

Todas las posibles ejecuciones entre p, q y r, están codificadas, de manera estática, en O. Es decir, O captura las configuraciones de salida legales para cuando los procesos:

- Corren solos sin saber de los demás (esquinas del complejo).
- Corren sabiendo sólo de algún otro proceso (simplejos de la frontera de O).
- Corren sabiendo de los otros dos procesos (triángulos internos) o pueden correr dos de forma concurrente y después el tercero (o viceversa).

Por ejemplo, el simplejo α representa la ejecución en el orden p y q (concurrentemente) y después r. El simplejo β representa la ejecución secuencial en el orden

r, q, p. El simplejo γ representa el orden r y después p y q (concurrentemente). Por último, el triángulo interior representa la ejecución en que los tres procesos corrieron concurrentemente.

La figura 2.4 ilustra cómo Δ determina el conjunto de decisiones para dos configuraciones (simplejos) de entrada. La salida correspondiente a la ejecución en solitario del proceso p con valor de entrada p, es el proceso p con valor de salida $\{p\}$. Es decir, Δ mapea el simplejo $\{(p, p)\} \in I$, en el subcomplejo de O que contiene solamente al simplejo $\{(p, \{p\})\}$:

$$\Delta(\{(p, p)\}) = \{\{(p, \{p\})\}\}$$

El simplejo $\{(q, q), (r, r)\}$ es mapeado en el subcomplejo de O que contiene a los vértices $(q, \{q\}), (r, \{r\}), (q, \{q, r\})$ y $(r, \{q, r\})$.

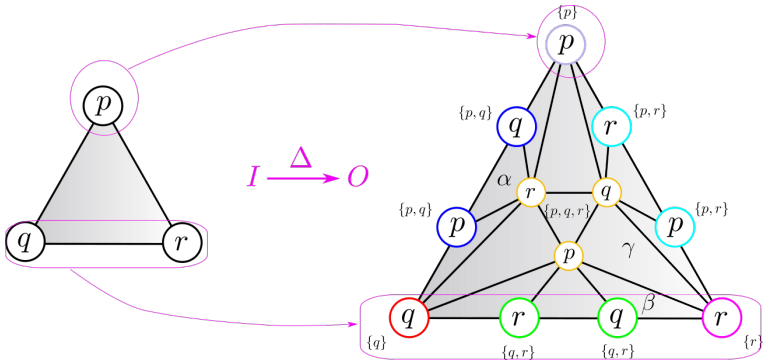

Figura 2.4: Dos asignaciones válidas para Δ. El mapeo portador Δ determina todos los valores de salida válidos de cada proceso, para una configuración de entrada determinada.

Es importante notar que O corresponde a la subdivisión cromática estándar de I. Como lo veremos en el capítulo 3, la subdivisión cromática estándar es un elemento central en el análisis del poder de solución de las tareas, en el modelo computacional de escritura-lectura, donde los procesos se comunican escribiendo y leyendo de una memoria compartida utilizando Immediate-snapshot.

2.1.3. Definiciones formales

Sean Π un conjunto de n nombres de procesos, V_i un dominio de valores de entrada y V_o un dominio de valores de salida.

Definición 2.1. Una *tarea* para n procesos, es una tupla $\langle I, O, \Delta \rangle$, donde:

- I es un *complejo simplicial de entrada* puro cromático $(n-1)$-dimensional, con una coloración $\chi_I : V(I) \to \Pi$ y un etiquetado $\phi_I : V(I) \to V_i$, tal que cada vértice es unívocamente identificado por su color y su etiqueta;

- O es un *complejo simplicial de salida* puro cromático $(n-1)$-dimensional, con una coloración $\chi_O : V(O) \to \Pi$ y un etiquetado $\phi_O : V(O) \to V_o$, tal que cada vértice es unívocamente identificado por su color y su etiqueta;

- Δ es un mapeo portador cromático, de I a O.

Normalmente, haremos implícito el etiquetado y representaremos a cada vértice de I y O, como una pareja (*id*, *valor*). Por ejemplo, en la tarea del consenso binario, a los vértices de I y O los denotamos con las parejas $(p,1),(p,0),(q,1),(q,0)$.

2.1.4. Satisfacción de una tarea

Las tareas son un método para especificar un problema distribuido. Las ejecuciones en la que cada proceso invoca una tarea, determina un conjunto de historias para ese problema. Necesitamos una manera de garantizar que en cualquier ejecución, cada proceso se comporta de acuerdo a la especificación de la tarea.

Sea E una historia arbitraria, donde cada proceso invoca a una tarea $T = \langle I, O, \Delta \rangle$ una vez.

Definición 2.2. El simplejo de entrada σ_E se define como: (id_i, x_i) está en σ_E, si y sólo si, en E hay una invocación de $task(x_i)$ por el proceso id_i.

σ_E corresponde a la configuración de entrada de la ejecución E.

Definición 2.3. El simplejo de salida τ_E se define como: (id_i, y_i) está en τ_E, si y sólo si hay una respuesta y_i a un proceso id_i en E.

Análogamente, τ_E corresponde a la configuración de salida de la ejecución E.

Definición 2.4. Decimos que E *satisface* T si para cada prefijo E' de E, se tiene que $\tau_{E'} \in \Delta(\sigma_{E'})$.

El requerimiento del prefijo nos asegura que cada proceso se comporte de acuerdo a la especificación de la tarea, durante toda la ejecución.

2.1.5. Limitaciones de las tareas

Contamos con dos métodos de especificar problemas distribuidos, las tareas y los objetos secuenciales. En el marco de las especificaciones secuenciales, es fácil representar una pila. Sin embargo, en esta sección veremos cómo, en el marco de las tareas, representar una pila es imposible.

De manera intuitiva, una especificación de una tarea T, captura un conjunto de historias válidas para el problema distribuido que representa T. El conjunto de historias que satisfacen la tarea T, es el conjunto $E(T)$ de historias válidas. Por otro lado, dada la especificación secuencial S de un objeto, el conjunto de las historias linealizables con respecto a S, es el conjunto $E(S)$ de historias válidas para el problema distribuido que representa S. Para las pilas con especificación secuencial P, no existe una tarea T, tal que $E(T) = E(P)$. Es decir, no existe una tarea que pueda capturar todas las historias válidas que captura P.

Sea C una especificación secuencial de una pila, restringida a una sola invocación de sus métodos *push* y *pop*, por cada proceso.

Lema 2.1. *No existe una tarea T_C tal que toda historia E es linealizable con respecto a C si y sólo si E satisface T_C.*

Demostración. Por contradicción, supongamos que existe una tarea $T_C = \langle I, O, \Delta \rangle$ tal que, para cada historia E, E es linealizable con respecto a C si y sólo si E satisface T_C.

Consideremos tres procesos p, q y r. La invocación a alguno de los métodos (*push* o *pop*) de una pila se hará a través del parámetro *método* de la invocación a *task*(*método*).

Es decir, el proceso p invocará $task(push(x))$, el proceso q invocará $task(push(y))$ y el proceso r invocará $task(pop())$.

Con esto, el conjunto de valores de entrada es

$$V_i = \{pop(), push(x), push(y)\}$$

el conjunto de valores de salida debe ser

$$V_o = \{ok, x, y, \perp\}$$

el valor $\perp$ representa la respuesta a una llamada a *pop* cuando la pila está vacía, y x o y en otro caso. El valor *ok*, es la respuesta a *push*.

El complejo de entrada I debe contener un vértice para cada posible operación de un proceso: $(p, push(x)), (q, push(y)), (r, pop())$. El complejo de salida O debe contener un vértice para cada posible respuesta a un proceso: $(r, y), (r, x), (r, \perp), (p, ok)$, (q, ok).

La representación de T_C está ilustrada en la figura 2.5.

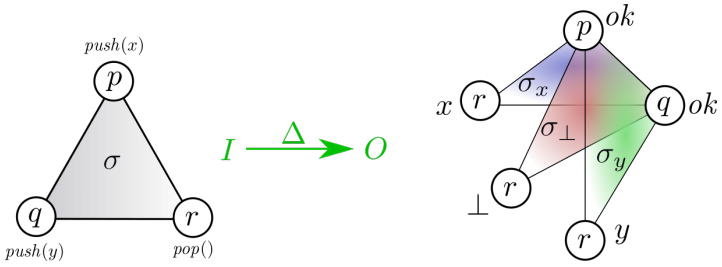

Figura 2.5: Tarea para una pila y tres procesos p, q y r. p y q ejecutan el método *push* con x y y como parámetros y tienen como respuesta el valor *ok*. Por otro lado, r ejecuta *pop* y puede obtener como respuesta x, y o $\perp$, dependiendo del orden de las llamadas a métodos.

Cada vértice de la forma (r, z), con $z \in \{x, y, \perp\}$, determina a un simplejo $\sigma_z = \{(p, ok), (q, ok), (r, z)\}$ que corresponde a cada uno de los posibles valores del método *pop* ejecutado por r, de manera concurrente con p y q. Cabe destacar que I y O, son puros cromáticos 2-dimensionales.

Ahora consideremos las ejecuciones α_1, α_2 y α_3 de la figura 2.6.

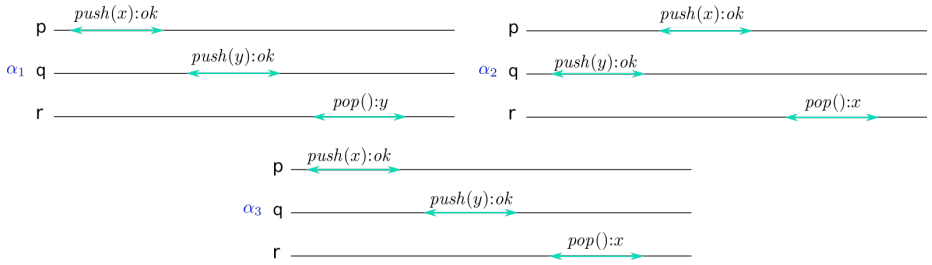

Figura 2.6: Tres ejecuciones de una pila. α_1 y α_2 son ejecuciones linealizables y α_3 no lo es.

Las ejecuciones α_1, α_2 son linealizables con respecto a C y además son válidas para T_C.

Para α_1, es claro que:

$$\{(p, ok)\} = \Delta(\{(p, push(x))\})$$
$$\{(p, ok), (q, ok)\} \in \Delta(\{(p, push(x)), (q, push(y))\})$$
$$\sigma_y = \{(p, ok), (q, ok), (r, y)\} \in \Delta(\{(p, push(x)), (q, push(y)), (r, pop())\}) = \Delta(\sigma)$$

De forma similar (nos fijamos sólo en el ultimo simplejo), para α_2 se tiene que:

$$\sigma_x = \{(p, ok), (q, ok), (r, x)\} \in \Delta(\sigma)$$

Ahora consideremos α_3, esta secuencia no es linealizable con respecto a C, sin embargo tenemos que es válida para T_C, ya que los conjuntos de entrada y salida son los mismos que para α_2:

$$\sigma_x = \{(p, ok), (q, ok), (r, x)\} \in \Delta(\sigma)$$

Con lo que α_3 es parte del conjunto de historias válidas para T_C y no está en el conjunto de historias válidas para C. Esto contradice nuestra hipótesis y por lo tanto no existe T_C.

□

Las tareas carecen del poder expresivo para representar incluso una versión restringida de una pila concurrente. Notemos que en la demostración del lema 2.1, las ejecuciones α_2 y α_3 son indistinguibles, pues la especificación de T_C no tiene la información necesaria para poder distinguir el orden en que pueden correr p y q.

2.2. Tareas refinadas

Las tareas carecen de poder expresivo al tener un mecanismo limitado para codificar los patrones entrelazados de las llamadas a métodos en una ejecución, o visto de otra forma, el tiempo. Las *tareas refinadas* [5], son una extensión de las tareas que nos permiten representar justo ese patrón de llamadas. En [5], existen dos versiones de las tareas refinadas:

- *one-shot*: permiten una sola invocación por proceso y
- *multi-shot*: permiten una o más invocaciones por cada proceso.

Ambas especificaciones difieren solamente en un valor extra codificado en los vértices de salida, por lo que no representa una diferencia técnica significativa. Además, la primera es un caso particular de la segunda, por lo tanto, trabajaremos desde un principio con la variante *multi-shot*.

Al igual que en una tarea regular, en una tarea refinada $T = \langle I, O, \Delta \rangle$, cada proceso empieza con un valor de entrada en V_i, privado; y eventualmente tiene que decidir un valor de salida en V_o.

Consideremos una ejecución donde k procesos participan. Cada vértice de entrada en I, sigue siendo una pareja (i_j, x_j) (con $x_j \in V_i$) que representa el estado inicial del proceso i_j. A diferencia de las tareas regulares, ahora ambos valores i_j, x_j, determinan el color de cada vértice.

El valor x_j representará, en el caso de una tarea para un objeto O, un método de O. Por ejemplo, en la tarea para una pila, el vértice de entrada $(a, \mathit{push}(x))$, representa la ejecución del método $\mathit{push}(x)$ por el proceso a. Dado que ambas componentes determinan el color del vértice, el vértice $(a, \mathit{push}(y))$ tiene un color distinto y por lo tanto la arista $\{(a, \mathit{push}(x)), (a, \mathit{push}(y))\}$ es un simplejo cromático y representa una configuración de entrada válida, en la que el proceso a ejecuta dos veces el método *push* con parámetros distintos. De esta manera, las tareas refinadas permiten representar más de una invocación por cada proceso.

Cada vértice de salida en O, es una tupla $(i_j, x_i, y_j, \omega_j)$ donde i_j sigue siendo el identificador de proceso y y_i sigue siendo el valor de salida. El valor de entrada x_i se mantiene en los vétices de salida. Esto permite mantener el color del vértice, determinado por ambos valores i_j, x_j y además, codifica información extra. En el caso de la representación de objetos, sabemos que el método x_j, produce la salida y_j. La componente ω_j es un simplejo de entrada (en I), llamado la *vista* de i_j. Este simplejo representa a todas las invocaciones y el orden en que preceden a la respuesta y_j, en una ejecución.

El mapeo Δ sigue siendo un mapeo portador cromático considerando la nueva coloración determinada por las primeras dos componentes de cada vértice. Además satisface que para cada simplejo $\sigma \in I$, la vista τ de cada vértice de $\Delta(\sigma)$, es un simplejo tal que $\tau \subseteq \sigma$. El simplejo τ codifica el orden de las invocaciones que pueden darse con la configuración de entrada σ. Esto quedará mas claro con los ejemplos de las tareas refinadas.

2.2.1. Pilas concurrentes

Una pila y tres procesos

La tarea refinada $T_1 = \langle I, O, \Delta \rangle$ de la figura 2.7, representa tres procesos p, q y r, donde p y q ejecutan los métodos $\mathit{push}(x)$ y $\mathit{push}(y)$, respectivamente, y el proceso r ejecuta el método $\mathit{pop}()$. Esta tarea sí representa la ejecución que la tarea del lema 2.1 pretendía representar.

El conjunto de valores de entrada (o métodos) es

$$V_i = \{\mathit{pop}(), \mathit{push}(x), \mathit{push}(y)\}$$

el conjunto de valores de salida es

$$V_o = \{ok, x, y, \perp\}$$

el valor $\perp$ representa la respuesta a una llamada a *pop* cuando la pila está vacía, y x o y en otro caso. El valor *ok*, es la respuesta a *push*.

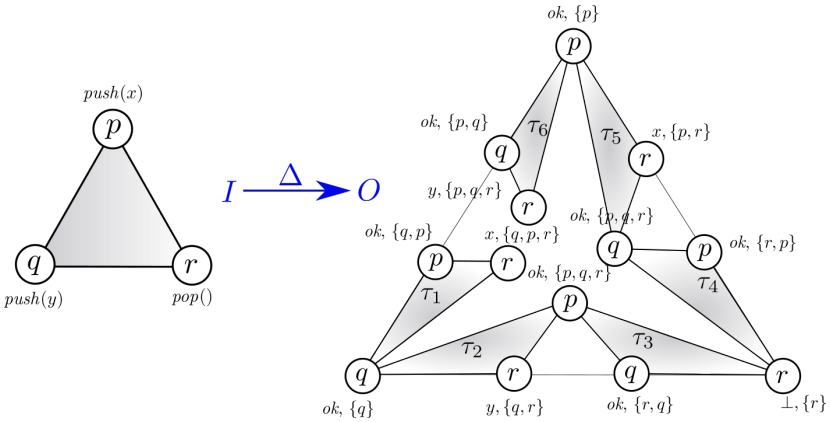

Figura 2.7: Tarea refinada T_1, para una pila y tres procesos p, q y r. El método en los vértices de salida está implícito y en la vista de cada vértice de salida, representa un simplejo de entrada. Los procesos p y q ejecutan el método *push* con x y y como parámetros y tienen como respuesta el valor *ok*. Por otro lado r, ejecuta *pop* y puede obtener como respuesta x, y o $\perp$.

Todas las posibles ejecuciones válidas entre p, q y r, están codificadas en cada uno de los triángulos τ_i. El orden de contención de las vistas de los vértices de un triángulo, determina el orden de ejecución. Por ejemplo, en el triángulo τ_1 las vistas pueden ordenarse por contención:

$$\{q\} \subseteq \{q, p\} \subseteq \{q, p, r\}$$

y representan la ejecución secuencial en el orden q, p y r.

Comparando los complejos de salida de las figuras 2.7 y 2.8, podemos apreciar toda la información codificada en la tarea refinada y que no existe en la tarea regular.

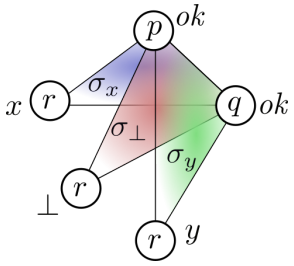

Figura 2.8: Complejo de salida O que tendría una tarea (regular) para una pila y tres procesos p, q y r.

Los procesos p y q, en la figura 2.7, aunque siguen teniendo las misma respuesta (ok), forman distintos simplejos, dependiendo de el orden de las invocaciones en sus vistas.

La figura 2.9 ilustra cómo Δ determina el conjunto de decisiones para dos configuraciones (simplejos) de entrada.

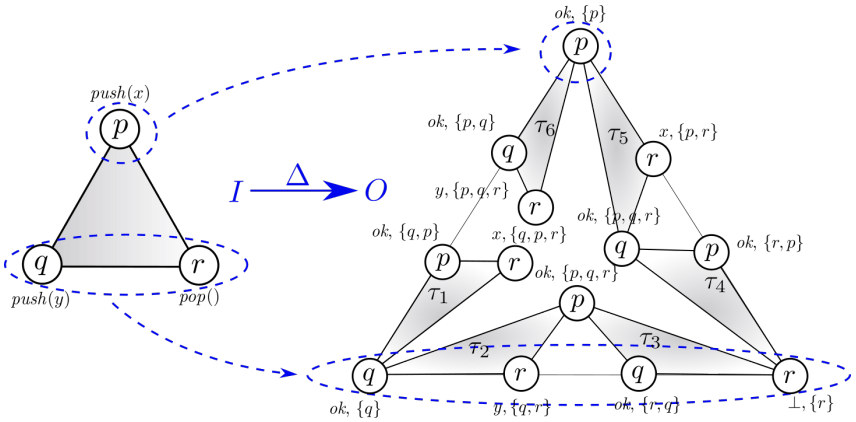

Figura 2.9: Dos asignaciones válidas para Δ. El mapeo portador Δ determina todos los valores de salida válidos de cada proceso, para una configuración de entrada determinada.

Las vistas de los vértices de los procesos q y r, de los triángulos τ_2 y τ_3, son caras de la arista formada por los los vértices de los procesos q y r de I.

Dos procesos, dos *pop* y un *push*

La figura 2.10 representa a una tarea refinada $T_2 = \langle I, O, \Delta \rangle$ para dos procesos a y b, donde a ejecuta los métodos $push(x)$ y $pop()$, y el proceso b ejecuta el método $pop()$.

El conjunto de valores de entrada (o métodos) es

$$V_i = \{pop(), push(x)\}$$

el conjunto de valores de salida es

$$V_o = \{ok, x, \perp\}$$

Para simplificar la notación, codificaremos el color (las primeras dos componentes) de cada vértice con el id del proceso (a o b) y un subíndice: o si la operación es *pop* y u si la operación es *push*. A los simplejos de mayor dimensión los codificaremos con secuencias de estos colores. Por ejemplo, a_o representará al vértice $(a, pop())$. La secuencia $a_o a_u b_u$, representará al simplejo formado por los vértices $(a, pop()), (a, push(x))$ y $(b, push(y))$. Notemos que $a_o \subset a_o a_u b_u$.

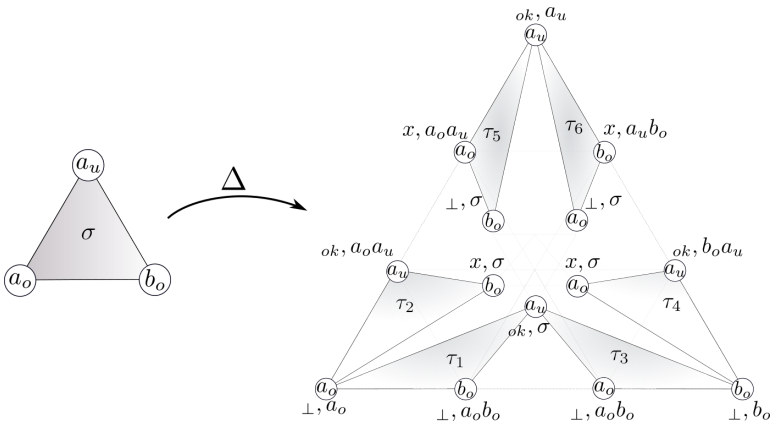

Figura 2.10: Representación de la tarea T_2, para una pila y dos procesos a y b. $a_o = (a, pop())$, $b_o = (b, pop())$ y $a_u = (a, push(x))$. Cada vértice de salida está etiquetado con el valor de salida y la vista.

El simplejo σ es el triángulo de entrada. Las esquinas del complejo de salida siguen representando la ejecución en solitario de cada proceso. No existe un 1-simplejo entre los vértices a_o y a_u que contengan la misma vista, es decir, que hayan ejecutado de manera concurrente sus operaciones. Las ejecuciones entre a_o y a_u son siempre secuenciales.

Una vez más, el orden por contención de las vistas de los vértices de cualquier triángulo τ_i determina el orden de ejecución. Por ejemplo, las vistas del triángulo τ_3 se pueden ordenar por contención, $b_o \subseteq a_o b_o \subseteq \sigma$, y representa la ejecución secuencial en el orden b_o, a_o, a_u.

Dos procesos, dos *push* y un *pop*

La tarea refinada $T_3 = \langle I, O, \Delta \rangle$ de la figura 2.11, representa dos procesos a y b, donde a ejecuta los métodos $push(x)$ y $pop()$, y el proceso b ejecuta el método $pop()$.

El conjunto de valores de entrada (métodos) es

$$V_i = \{pop(), push(x), push(y)\}$$

el conjunto de valores de salida es

$$V_o = \{ok, x, y, \perp\}$$

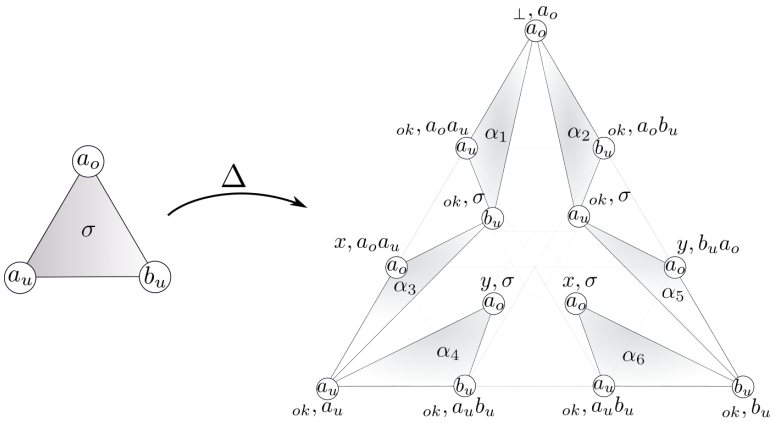

Figura 2.11: Representación de la tarea T_3 con el simplejo de entrada $\sigma = a_o a_u b_u$.

Notemos que los complejos de salida de T_3 y T_1 modelan las mismas configuraciones de salida. Esto es porque todos los triángulos de ambos complejos, modelan ejecuciones secuenciales.

Dos procesos y dos llamadas por proceso

Al aumentar el número de operaciones y/o procesos, aumenta la dimensión de los complejos de entrada y salida.

La tarea refinada $T_4 = \langle I, O, \Delta \rangle$, representa dos procesos a y b, donde ambos ejecutan un $push()$ y un $pop()$.

El conjunto de valores de entrada (métodos) es

$$V_i = \{pop(), push(x), push(x)\}$$

el conjunto de valores de salida es

$$V_o = \{ok, x, y, \perp\}$$

En este caso tenemos 4 distintas ejecuciones por lo que los complejos de salida y entrada aunmentan de dimensión.

El complejo de entrada en la figura 2.12 es puro 3-dimensional.

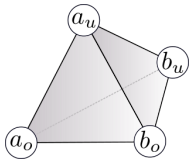

Figura 2.12: Complejo de entrada de T_4.

El complejo de salida en la figura 2.13 es puro 3-dimensional.

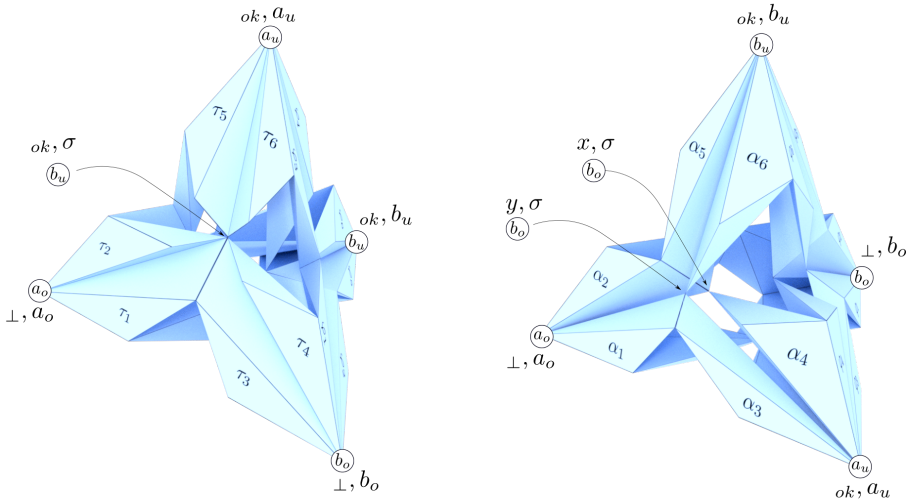

Figura 2.13: Complejo de salida de T_4, visto desde dos de sus caras.

En el complejo de salida, los vértices de las esquinas siguen representando las ejecuciones en solitario de cada proceso. El subcomplejo 2-dimensional (frontera de O) con esquinas a_u, a_o, b_o, es el mismo complejo de salida de la tarea T_2. Ambos están formados por los triángulos τ_i. El mismo caso se tiene para el subcomplejo (frontera de O) formado con esquinas a_u, a_o, b_u y el complejo de salida de la tarea T_3. Ambos están formados por los triángulos α_i.

La cerradura bajo contención de los simplejos, captura la noción de que si un proceso es muy lento o falla desde el principio, los demás procesos correrán y se comportarán como si el proceso faltante no existiera. Cada uno de los simplejos τ_i y α_j, son la cara de un 3-simplejo (al interior del complejo de salida), que se forma con el vértice de color restante.

En general, para el caso de n procesos, cada uno ejecutando k operaciones, el complejo de entrada I tendrá nk vértices distintos ($(nk-1)$-dimensional) y por tanto $\Omega(2^{nk})$ simplejos distintos.

Para un objeto de larga vida, el conjunto de operaciones que puede realizar un proceso es potencialmente infinito. Por lo que I sería un complejo de dimensión infinita con simplejos de dimensión finita.

2.2.2. Definiciones formales

Sean Π un conjunto de n nombres de procesos, V_i un dominio de valores de entrada y V_o un dominio de valores de salida.

Definición 2.5. Una *tarea refinada* para n procesos es una tupla $\langle I, O, \Delta \rangle$, donde:

1. I es un *complejo simplicial de entrada* puro, cromático, $(n-1)$-dimensional, con una coloración $\chi_I : V(I) \to \Pi \times V_i$, tal que cada vértice es unívocamente identificado por su color;

2. O es un *complejo simplicial de salida* puro, cromático, $(n-1)$-dimensional, con una coloración $\chi_O : V(O) \to \Pi \times V_i$, y dos etiquetados $\phi_{O_1} : V(O) \to V_o$ y $\phi_{O_2} : V(O) \to I$, tal que cada vértice es unívocamente identificado por su color y sus etiquetas;

3. Δ es un mapeo portador cromático, de I a O, que además, satisface lo siguiente:

   - para cada simplejo $\sigma \in I$, para todo vértice $(i_j, x_j, y_j, \omega_j) \in \Delta(\sigma)$, se tiene que $\omega_j \subseteq \sigma$.

Una vez más, haremos implícito el etiquetado y representaremos a cada vértice de I, como una pareja (*id*, *invocación a método*), y a cada vértice de O, como una tupla (*id*, *invocación a método*, *valor de salida*, *vista*).

Al igual que con las tareas regulares, necesitamos una manera de garantizar que en cualquier ejecución, cada proceso se comporta de acuerdo a la especificación de la tarea.

2.2.3. Satisfacción de una tarea refinada

Sea H una historia arbitraria donde cada proceso invoca a una tarea refinada $T = \langle I, O, \Delta \rangle$.

Las definiciones 2.6 y 2.7, son análogas a las definiciones 2.2 y 2.3, respectivamente.

Definición 2.6. σ_H es el simplejo con todas las invocaciones en H.

Definición 2.7. τ_H es el simplejo con todas las respuestas en H y cada respuesta con su correspondiente vista en H.

Por ejemplo, consideremos la tarea T_2 de la figura 2.10 y una historia H tal que:

$$H = \langle push(x), a\rangle\langle ok, a\rangle\langle pop(), b\rangle\langle x, b\rangle$$

entonces, $\sigma_H = a_u b_o$ y $\tau_H = (a_u, ok, a_u)(b_o, x, a_u b_o)$. La vista de cada vértice de respuesta en τ_H, es el simplejo formado por las invocaciones que le preceden en H, a esa respuesta.

La definición 2.8 es análoga a la definición 2.4, tomando en cuenta las vistas de los vértices de salida.

Definición 2.8. Decimos que una historia H *satisface* T, si para todo prefijo E de H, hay un simplejo $\lambda \in \Delta(\sigma_E)$ tal que:

- $\text{Ids}(\tau_E) \subseteq \text{Ids}(\lambda)$ y
- Para todo $(id, op, y, \omega) \in \tau_E$, existe $(id, op, y, \omega') \in \lambda$ tal que $\omega' \subseteq \omega$.

Donde $\text{Ids}(\sigma)$ denota al conjunto de colores de los vértices en el simplejo σ.

De manera intuitiva, hay un simplejo λ en el que el orden por contención de sus vistas, codifica el orden de las invocaciones en H.

Por ejemplo, consideremos la tarea T de la figura 2.14. Consideremos la ejecución α_1 de la figura 2.6:

$$\alpha_1 = \langle push(x), p\rangle\langle ok, p\rangle\langle push(y), q\rangle\langle ok, q\rangle\langle pop(), r\rangle\langle y, r\rangle$$

Tomemos un prefijo E de α_1

$$E = \langle push(x), p\rangle\langle ok, p\rangle\langle push(y), q\rangle\langle ok, q\rangle$$

La figura 2.14 ilustra los simplejos σ_E, τ_E y λ junto con el subcomplejo $\Delta(\sigma_E)$.

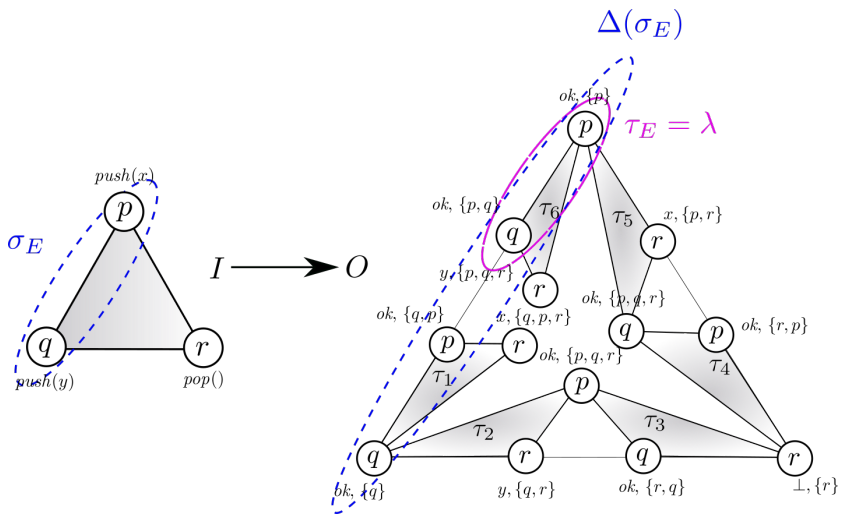

Figura 2.14: Los simplejos σ_E, τ_E y λ junto con el subcomplejo $\Delta(\sigma_E)$ para un prefijo E de una historia H.

Lo anterior sugiere que para que una tarea pueda ser satisfecha, sus vistas deben de estar bien definidas, en algún sentido.

Definición 2.9. Una tarea está bien definida si

- Todo vértice de salida (id, op, y, ω) está en $\Delta(\omega)$.
- Para todo simplejo $\sigma \in I$, las vistas de cada simplejo $\tau \in \Delta(\sigma)$ son tales que:
  - Para cada $(id, op, y, \omega) \in \tau$, ω contiene la invocación a op del proceso id.
  - El conjunto de todas las vistas en τ pueden ordenarse bajo contención. Es decir, pueden indexarse como $\omega_1, \omega_2, \ldots, \omega_m$, tales que $\omega_1 \subset \omega_2 \subset \cdots \subset \omega_m$.

Es fácil verificar que las tareas anteriores están bien definidas. Consideremos la tarea T_2. Si nos fijamos en el simplejo τ_1:

- la vista de cada uno de sus vértices está en $\Delta(\sigma)$ cumpliendo la primera condición.
- Para la segunda condición es claro que en τ_1, cada vértice contiene un vértice con el mismo color.
- Las vistas de los vértices de τ_1, se pueden ordenar bajo contención, es decir, $a_o \subset a_o b_o \subset \sigma$,

además, para cada simplejo de τ_1 (aristas y vértices), también se cumplen estas condiciones. Con un análisis similar, podemos ver que todos los simplejos τ_i cumplen con estas condiciones y por lo tanto la tarea T_2, está bien definida. De manera similar, podemos verificar que T_1, T_3 y T_4 están bien definidas.

En general una tarea T, definida para n distintas tuplas de la forma (i, o) en la que el proceso i invoca al método o, representará todas las posibles ejecuciones secuenciales entre los n distintos procesos (i, o), si satisface la definición 2.9.

Aqui cabe destacar que, en una tarea para una pila, podemos representar que un proceso a invoque más de una vez al método *pop*, haciendo que el método reciba un parámetro de la misma manera que el método *push*. Así, cada invocación al mismo método por el mismo proceso determina un color distinto ya que se puede distinguir por su parámetro.

Sea $T = \langle I, O, \Delta \rangle$ una tarea refinada en la que n distintos procesos pueden invocar un número arbitrario de veces, cualquiera de los dos métodos, *pop* y *push*, de una pila; y que además satisface la definición 2.9.

Sea C una especificación secuencial de una pila (sin restricciones). Sea H una historia donde cada proceso invoca a la tarea T.

Teorema 2.1. *H es linealizable si y sólo si, H satisface la tarea T.*

Demostración.

Sea E un prefijo de H.

- $(\rightarrow)$. Supongamos que H es una ejecución linealizable. Sabemos que E es linealizable.

  Como E es linealizable, existe una linealización S de E tal que:

  - S es una ejecución secuencial válida para C.
  - Existe una extensión completa E' de E tal que E' es equivalente a S.

  Notemos que $\sigma_E = \sigma_{E'} = \sigma_S$, pues E, E' y S tienen las mismas invocaciones. Además, $\tau_E \subseteq \tau_{E'}$, pues E' es extensión de E.

  P.D. que H satisface T. es decir, hay un simplejo $\lambda \in \Delta(\sigma_E)$ tal que:

  - $\text{Ids}(\tau_E) \subseteq \text{Ids}(\lambda)$ y
  - Para todo $(id, op, y, \omega) \in \tau_E$, existe $(id, op, y, \omega') \in \lambda$ tal que $\omega' \subseteq \omega$.

  Proposición: $\lambda = \tau_S$.

Dado que S es secuencial, τ_S es un simplejo de O con las respuestas a las invocaciones en σ_S y por tanto $\tau_S \in \Delta(\sigma_S) = \Delta(\sigma_E)$.

Como E' y S son equivalentes, $\text{Ids}(\tau_{E'}) = \text{Ids}(\tau_S)$ y por lo tanto $\text{Ids}(\tau_E) \subseteq \text{Ids}(\tau_S)$, pues $\tau_E \subseteq \tau_{E'}$.

Ahora tomemos un vértice $(id, op, y, \omega) \in \tau_E$, por lo anterior, sabemos que existe un vértice $(id, op, y', \omega') \in \tau_S$. Como S es una linealización de E, la respuesta a la invocación de op hecha por el proceso id, es la misma, es decir, $y = y'$.

Falta demostrar que $\omega' \subseteq \omega$. Por contradicción, supongamos que $\omega \subset \omega'$, es decir, existe una invocación $i \in \omega' \setminus \omega$. Además, Δ garantiza que $\omega' \subseteq \sigma_E$ (pues $\sigma_E = \sigma_S$), es decir, que las invocaciones representadas en ω', están en E.

Por lo tanto y aparece antes de la invocación i en E. Pero S respeta el orden de las llamadas en E, por lo que y aparece antes de la invocación i en S también, por lo tanto $i \notin \omega'$, lo que es una contradicción. Por lo anterior, $\omega' \subseteq \omega$.

- ($\leftarrow$). Supongamos que H satisface a T. Sea σ_E el simplejo con las invocaciones en E y sea τ_E el simplejo con las respuestas en E.

  Como H satisface a T, hay un simplejo $\lambda \in \Delta(\sigma_E)$ tal que:

  - $\text{Ids}(\tau_E) \subseteq \text{Ids}(\lambda)$ y
  - Para todo $(id, op, y, \omega) \in \tau_E$, existe $(id, op, y, \omega') \in \lambda$ tal que $\omega' \subseteq \omega$.

  Por construcción de T, λ representa una ejecución secuencial S. Sea σ_S el simplejo con las invocaciones en S y sea $\tau_S = \lambda$ el simplejo con las respuestas en S. Es fácil ver que $\tau_E \subseteq \tau_S$ pues las respuestas en S son las mismas representadas en λ. Además, las invocaciones en λ son las mismas que las de σ_E por lo que $\sigma_S = \sigma_E$.

  Por lo tanto, en S se tienen las misma invocaciones que en E y las respuestas en E son un subconjunto de las respuestas en S.

  Para que S sea una linealización de E, falta demostrar que S mantiene el orden de las llamadas en E.

  Por contradicción, supongamos que en E hay una respuesta y_i a una invocación (i, op_i) antes de una invocación $I_j = (j, op_j)$. Además, supongamos que en S, I_j ocurre primero que y_i.

  Sean (i, op_i, y_i, ω) y (i, op_i, y_i, ω') en τ_E y τ_S, respectivamente.

Entonces $I_j \notin \omega$ e $I_j \in \omega'$, entonces $\omega' \not\subseteq \omega$, lo que contradice que H satisface T.

□

2.3. Resumen

Una tarea es una manera estática de especificar un problema distribuido con una operación que puede ser invocada sólo una vez por cada proceso.

Una limitación importante de las tareas es que carecen del poder expresivo para representar patrones entrelazados de llamadas a métodos en una ejecución. Las tareas refinadas son una extensión de las tareas, con un mayor poder expresivo, que nos permiten representar justo ese patrón de llamadas.

Utilizando una noción de satisfacción de una tarea por una especificación secuencial, establecimos la representación combinatoria de las pilas concurrentes de larga vida.

Estudiamos el caso de las pilas concurrentes, teniendo en mente que el razonamiento para las colas concurrentes es el mismo.

Capítulo 3

Especificación de algoritmos distribuidos

Las tareas son una forma de representar un problema distribuido de manera estática, en términos de asignaciones de valores de entrada y salida válidos. Éstas no toman en cuenta cómo se comunican los procesos entre sí, son independientes del modelo de cómputo distribuido.

Los *protocolos* [1] son la manera de especificar un algoritmo distribuido, considerando un modelo de cómputo determinado y se representan de manera estática en términos de configuraciones iniciales y finales.

En este capítulo veremos cómo representar protocolos en el marco de la topología combinatoria, mediante complejos simpliciales, para el modelo de cómputo de nuestro interés: el *modelo de escritura-lectura por capas* [1] en el que los procesos se comunican a través de una memoria global compartida.

También veremos cómo la solubilidad de una tarea en el modelo, depende de la existencia de alguna relación topológica (mapeos simpliciales) entre los complejos simpliciales de la tarea y los complejos simpliciales de un protocolo asociado a ella.

3.1. Modelo de cómputo

En el *modelo de escritura-lectura por capas* [1], n procesos se comunican al leer y escribir en una memoria compartida. Para resolver un problema, cada proceso ejecuta un algoritmo llamado *protocolo immediate-snapshot por capas* [1, 2]. Un proceso empieza la ejecución en un estado inicial y puede fallar sin haber ejecutado un solo paso o *capa*; o ejecuta un número de capas hasta parar por haber completado el protocolo. En cada capa realiza un cómputo local y se comunica con otros procesos

a través del entorno que provee el modelo. Todos los procesos ejecutan cada capa de manera asíncrona.

En una ejecución de un protocolo de L capas, la memoria se organiza como un arreglo $mem[L][n]$ de $L \times n$ elementos. En cada capa l, el proceso j realiza un *immediate-snapshot* escribiendo su estado en $mem[l][j]$ y obteniendo el estado de los demás procesos en las entradas $mem[l][*]$. Al valor obtenido por el snapshot en una capa k, por un proceso i, le llamamos la *vista* de i en la capa k. Los protocolos son de *información completa*, es decir, un proceso le comunica a los demás su estado completo. Esto lo podemos considerar porque sólo estamos interesados en estudiar la solubilidad de las tareas y no cuestiones de implementación/optimización. Dado que ningún proceso espera a otro para actuar, este modelo es *wait-free* [2].

Un *protocolo* para una tarea T, es simplemente, un algoritmo que resuelve T. En particular, estamos interesados en un tipo particular de protocolo, el cual divide el cómputo en una parte que es independiente de T y otra que depende de ella. La parte independiente es de comunicación: cada proceso i, en cada capa, comunica su vista completa a los demás procesos y recibe la vista de los otros a cambio (cada vista asociada al identificador del proceso al que pertenece); después, i actualiza su estado, reflejando lo que ha aprendido. La parte dependiente de T es de decisión: cuando han pasado suficientes capas de comunicación, cada proceso escoge un valor de salida válido, de acuerdo a la especificación de T, aplicando un mapeo de decisión δ, a su *vista final*.

Cualquier algoritmo que implemente un protocolo immediate-snapshot por capas, se puede representar con el pseudo-código de la figura 3.1.

```
1: shared mem: array[L][n]
2: i: P_i                                                  ▷ id del proceso

3: protocol layered()
4:     value: input                                        ▷ input es el valor inicial
5:     for l : 1 to L do
6:         immediate
7:             mem[l][i]: value
8:             snap: snapshot(mem[l][*])
9:         value: updateState(snap)
10:    return δ(value)                                     ▷ aplica decisión δ al estado final
11: end protocol
```

Figura 3.1: Pseudocódigo de un protocolo de L capas que divide el cómputo en dos partes: comunicación y decisión.

3.2. Protocolos

En un protocolo, la parte independiente de la tarea la podemos representar de manera estática mediante complejos simpliciales. Cada proceso empieza con un valor inicial en un dominio de valores iniciales V_i. Eventualmente, después de suficientes capas de comunicación, obtiene un valor final en un dominio de valores finales válidos V_p. En la parte dependiente de la tarea, aplica un mapeo de decisión sobre el valor final obtenido.

Consideremos una ejecución en donde k procesos participan. En un protocolo, una familia de subconjuntos de un conjunto de vértices $\{(i_1, x_1), \ldots, (i_n, x_n)\}$, determina un un complejo simplicial de entrada I, puro cromático $(n-1)$-dimensional, en donde cada $x_j \in V_i$ denota el *valor inicial* del proceso con identificador i_j, esto es, el vértice (i_j, x_j) representa el estado inicial de i_j. Cada simplejo de entrada representa una configuración inicial.

Una familia de subconjuntos de un conjunto de vértices $\{(i_1, y_1), \ldots, (i_n, y_n)\}$, determina un *complejo simplicial del protocolo* P, puro cromático $(n-1)$-dimensional, en donde cada $y_j \in V_p$ denota la *vista final* del proceso con identificador i_j, esto es, el vértice (i_j, y_j) representa el estado final, de i_j. A cada simplejo de P, le decimos *simplejo de protocolo* y representa una configuración final que corresponde a una ejecución del protocolo.

La relación entre una configuración de entrada y su correspondiente conjunto de

configuraciones finales válidas, la denotamos mediante un mapeo portador cromático Ξ, llamado *mapeo portador de ejecución*. El mapeo Ξ determina todos los estados finales válidos de cada proceso, para una configuración de entrada determinada. Dicho de otro modo, Ξ especifica, para cada asignación de valores de entrada, qué asignaciones de valores para las vistas pueden escogerse, al final de las capas de comunicación.

Para cada configuración de entrada, puede haber más de una configuración final que depende de la interacción concurrente de los procesos, razón por la cual, Ξ mapea un simplejo de I en un subcomplejo de P.

Un protocolo que resuelve una tarea $T = \langle I, O, \Delta \rangle$, es una tupla $\langle I, P, \Xi \rangle$, en donde el complejo de entrada I de T es también el complejo de entrada del protocolo. El complejo del protocolo P, se relaciona con el complejo de salida O, mediante un mapeo simplicial δ llamado el *mapeo de decisión*, el cual manda cada vértice del complejo del protocolo P, a un vértice del complejo de salida O, respetando la coloración.

3.2.1. Protocolos immediate-snapshot de una capa

La figura 3.2 representa al protocolo immediate-snapshot de una capa para dos procesos p y q. Cada proceso tiene implícito como valor inicial su identificador.

El dominio de valores iniciales es

$$V_i = \{p, q\}$$

Al escribir en la memoria el proceso i, escribe el valor i en la entrada mem[0][i]. Al hacer el snapshot, guardará el conjunto de valores obtenidos, cada uno como una tupla (j, j), de todos los procesos que hicieron snapshot antes o concurrentemente con i, incluyendo lo que él mismo escribió $((i, i))$.

El dominio de valores finales es

$$V_p = \{\{(p,p)\}, \{(q,q)\}, \{(p,p),(q,q)\}\} = \mathcal{P}(\{(p,p),(q,q)\}) \setminus \emptyset$$

siendo $\mathcal{P}(A)$ el conjunto potencia del conjunto A.

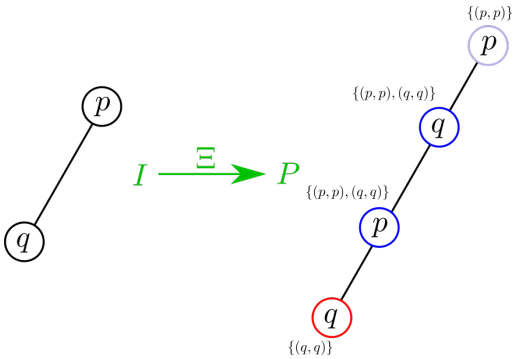

Figura 3.2: Representación del protocolo immediate-snapshot de una capa para dos procesos p y q. Los valores de entrada están implícitos y son iguales al identificador del proceso.

Tanto I como P, son cromáticos puros 1-dimensionales. Todas las posibles ejecuciones entre p y q, están codificadas de manera estática, en P.

Los extremos de P, representan las ejecuciones en solitario por lo que un proceso sólo ve el valor que él mismo escribió. Las dos aristas conectadas a los extremos, representan ejecuciones secuenciales, en la que el primer proceso en ejecutar la capa, ve sólo su valor y el otro proceso puede ver ambos. La arista central representa una ejecución concurrente, ambos procesos ven ambos valores escritos por cada uno.

El complejo simplicial P es la subdivisión cromática estándar de I, $\mathrm{Ch}(I)$.

La figura 3.2 representa al protocolo immediate-snapshot de una capa para tres procesos p, q y r.

El dominio de valores iniciales es

$$V_i = \{p, q, r\}$$

El proceso p al hacer snapshot puede obtener uno de los siguientes conjuntos de valores:

- $\{(p, p)\}$, si corrió en solitario o antes que q y r.
- $\{(p, p), (j, j)\}$ si corrió después o de manera concurrente con el proceso j (con $j \in \{q, r\}$).
- $\{(p, p), (q, q), (r, r)\}$ si corrió después o de manera concurrente con ambos procesos.

En otras palabras, dado que p, q y r son procesos distintos y pueden correr de forma concurrente, el conjunto potencia (menos el vacío) de $\{(p, p), (q, q), (r, r)\}$, contiene

todas las posibles combinaciones en las que los procesos pueden realizar el immediate-snapshot. El proceso i puede obtener como valor, cualquier elemento de ese conjunto que contenga la tupla (i, i) (un proceso siempre ve lo que el mismo escribe).

El dominio de valores de finales es

$$V_p = \mathcal{P}(\{(p,p),(q,q),(r,r)\}) \setminus \emptyset$$

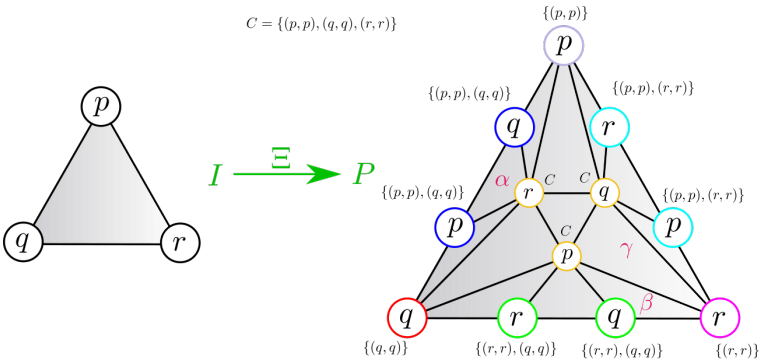

Figura 3.3: Representación del protocolo immediate-snapshot de una capa para tres procesos p, q y r. Los valores de entrada están implícitos y son iguales al identificador del proceso.

Tanto I como P, son cromáticos puros 2-dimensionales. Todas las posibles ejecuciones entre p, q y r, están codificadas, de manera estática, en P.

Las esquinas de P, representan las ejecuciones en solitario de cada proceso, un proceso sólo ve el valor que él mismo escribió. El simplejo α representa la ejecución en la que p y q, corrieron concurrentemente antes que r. El simplejo γ representa la ejecución en la que r corrió primero y después p y q, concurrentemente (el orden inverso a α). El simplejo β representa la ejecución secuencial en el orden r, q, p. El triángulo interior representa la ejecución concurrente de los tres procesos.

El complejo simplicial P es la subdivisión cromática estándar de I, $\mathrm{Ch}(I)$. El complejo del protocolo de la figura 3.2 es un subcomplejo de P (la frontera entre las esquinas p y q). Esto es claro ya que P codifica todas las posibles ejecuciones, en particular las que se dan entre p y q solamente (cuando r no participa).

Muchas veces, una sola capa de comunicación no será suficiente para resolver un problema concurrente. Sin embargo, podremos construir protocolos multicapa usando la *composición de protocolos* en la que el complejo de protocolo de uno se vuelve el complejo de entrada de otro y los mapeos portadores siguen la composición usual de funciones. De esta manera, podemos construir protocolos con un número arbitrario de capas.

3.2.2. Protocolos immediate-snapshot multicapa

Podemos construir un protocolo immediate-snapshot de L capas tomando como entrada de cada capa, al complejo de protocolo de la capa anterior. El complejo para un protocolo immediate-snapshot de L capas, con un complejo de entrada I, construido de esta manera, es la subdivisión cromática estándar iterada, $\mathrm{Ch}^L(I)$.

La figura 3.2 representa dos complejos de protocolo immediate-snapshot para las capas 1 y 2, para dos procesos p y q.

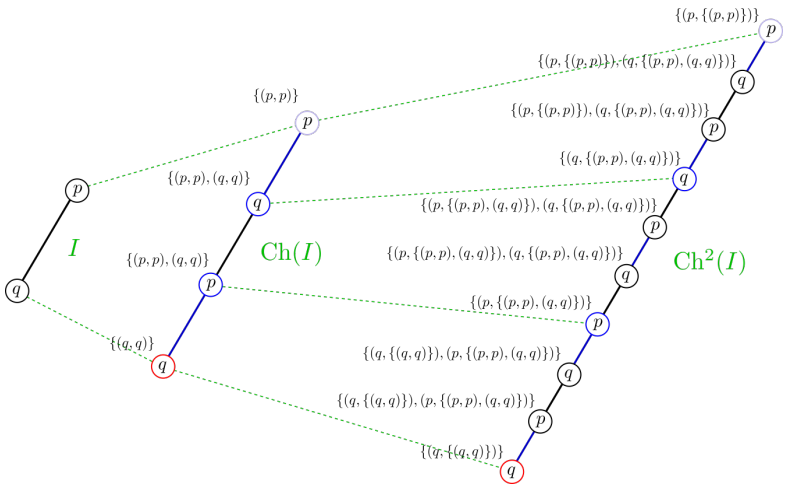

Figura 3.4: Complejo de entrada I y las subdivisiones cromáticas estándar, $\mathrm{Ch}(I)$ y $\mathrm{Ch}^2(I)$ que corresponden a las capas 1 y 2, resp., del protocolo immediate-snapshot.

La vista en un vértice de $\mathrm{Ch}^2(I)$ se sigue codificando como una tupla con el identificador del proceso y el snapshot de la memoria en la capa 2. Cada proceso escribe en la memoria de la capa 2, lo que obtuvo como valor al hacer el snapshot en la capa 1.

La figura 3.5 representa el complejo del protocolo immediate-snapshot para la primera capa y un subcomplejo del protocolo para la segunda capa. Este subcomplejo es la subdivisión cromática estándar de dos simplejos α y β que comparten una arista, en el complejo de la capa anterior.

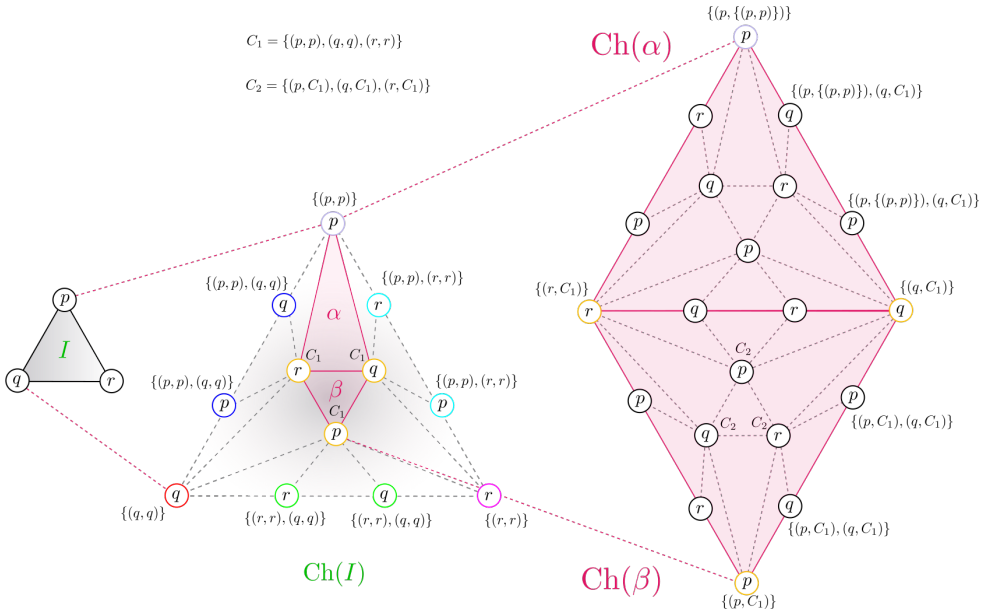

Figura 3.5: Protocolo immediate-snapshot para tres procesos. Dos subcomplejos de la segunda capa, están formados por la subdivisión cromática estándar de los simplejos α y β de la capa anterior.

Los simplejos α y β comparten una arista con los colores r y q. En la siguiente capa, la subdivisión cromática de esta arista, sigue siento compartida por las correspondientes subdivisiones cromáticas de α y β. La subdivisión cromática estándar iterada no genera huecos, es decir, si dos simplejos comparten una cara c, sus correspondientes subdivisiones cromáticas estándar compartirán las caras de la subdivisión de c.

Notemos que el subcomplejo de la frontera de $\mathrm{Ch}(I)$ entre las esquinas p y q, es el complejo de protocolo de la primera capa en la figura 3.2. Lo mismo sucede para de la frontera de $\mathrm{Ch}^2(I)$ (que no está totalmente representado en la figura 3.5) y el complejo de protocolo de la segunda capa en la figura 3.2. Esto tiene sentido pues los complejos de protocolo para tres procesos codifican todas las posibles ejecuciones concurrentes entre ellos, en particular, las ejecuciones en las que uno de los procesos no participa.

3.2.3. Resolución de una tarea

Una tarea $\langle I, O, \Delta\rangle$ y un protocolo $\langle I, P, \Xi\rangle$ están vinculados mediante el complejo simplicial de entrada I y el mapeo de decisión $\delta : O \to P$, este último es utilizado por los procesos para escoger sus valores de salida, mapeando cada vista final a un

valor de salida. Decimos que un proceso i *escoge* o *decide* la salida u con vista final v si $\delta((i,v)) = (i,u)$. Decimos que un protocolo *resuelve* una tarea si existe un mapeo de decisión entre P y O.

La tarea immediate-snapshot (figura 2.3) es resuelta trivialmente por el protocolo immediate-snapshot de una capa (figura 3.3).

Es claro (figura 3.6) que cada simplejo de P tiene una correspondencia uno a uno con cada simplejo de O.

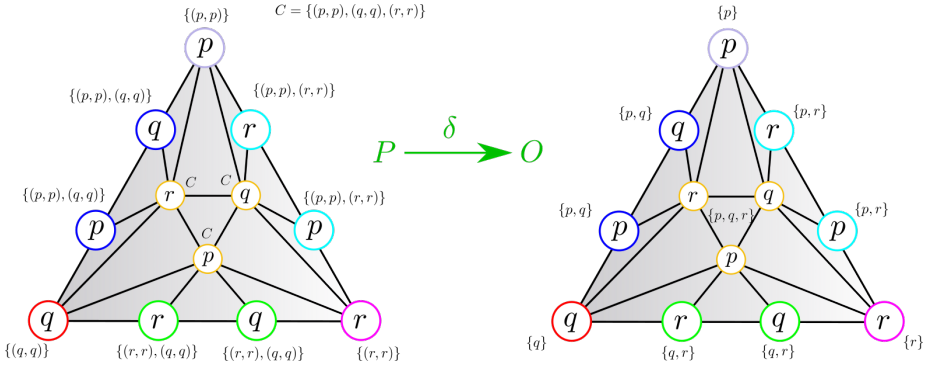

Figura 3.6: Los simplejos de P tienen una correspondencia uno a uno con los simplejos de O.

3.2.4. Definiciones formales

Sean Π un conjunto de n nombres de procesos, usualmente $\Pi = [n]$, V_i un dominio de valores iniciales y V_p un dominio de valores finales.

Definición 3.1. Un *protocolo* para n procesos, es una tupla $\langle I, P, \Xi \rangle$, donde:

- I es un *complejo simplicial de entrada* puro, cromático, $(n-1)$-dimensional, con una coloración $\chi_I : V(I) \to \Pi$ y un etiquetado $\phi_I : V(I) \to V_i$, tal que cada vértice es unívocamente identificado por su color y su etiqueta;
- P es un *complejo simplicial de protocolo* puro, cromático, $(n-1)$-dimensional, con una coloración $\chi_P : V(P) \to \Pi$ y un etiquetado $\phi_P : V(P) \to V_p$, tal que cada vértice es unívocamente identificado por su color y su etiqueta;
- $\Xi : I \to 2^P$ es un mapeo portador cromático estricto tal que $P = \cup_{\sigma \in I} \Xi(\sigma)$.

Al igual que con las tareas, haremos implícito el etiquetado y representaremos a cada vértice de I, como una pareja (*id*, *valor*), y a cada vértice de P, como una pareja (*id*, *vista*).

Definición 3.2. El *protocolo immediate-snapshot de una capa* $\langle I, P, \Xi \rangle$, para n procesos, es:

- El complejo de entrada I puede ser cualquier complejo simplicial puro cromático n-dimensional, coloreado con valores de Π y etiquetado con valores V_i.
- El mapeo portador $\Xi : I \to 2^P$ manda cada simplejo $\sigma \in I$ al subcomplejo de configuraciones finales de ejecuciones immediate-snapshot de un sola capa en donde todos y solamente, los procesos en σ, participan.
- El complejo de protocolo P es la unión de $\Xi(\sigma)$, sobre todo $\sigma \in I$.

Lema 3.1 (Lema 8.4.4 - [1])**.** *El protocolo immediate-snapshot de una capa cumple con la definición 3.1 de protocolo.*

En el caso en que tenemos n distintos procesos que realizan una sola operación (single-shot), tenemos que $P = \mathrm{Ch}(I)$, y además $\Xi = \mathrm{Ch}$. Por lo tanto, el protocolo immediate-snapshot equivale a la tupla $\langle I, \mathrm{Ch}(I), \mathrm{Ch} \rangle$. Sin embargo, en general esto no se cumple. En la sección 3.2.5 veremos cómo al considerar la representación de un proceso que puede realizar distintas operaciones (multi-shot) el complejo del protocolo P no coincide con $\mathrm{Ch}(I)$.

Utilizaremos la composición de protocolos para construir protocolos multicapa.

Definición 3.3 (Composición de protocolos)**.** Supongamos que tenemos dos protocolos $\langle I, P, \Xi \rangle$ y $\langle I', P', \Xi' \rangle$, donde $P \subseteq I'$. Su *composición* es el protocolo $\langle I, P'', \Xi'' \rangle$, donde Ξ'' es la composición de Ξ y Ξ', $(\Xi' \circ \Xi)(\sigma) = \Xi'(\Xi(\sigma))$, para $\sigma \in I$ y $P = \Xi''(I)$.

La composición de protocolos sigue siendo un protocolo ya que la composición de mapeos portadores estrictos, sigue siendo un mapeo portador estricto. En particular, nos interesa construir protocolos immediate-snapshot multicapa.

Definición 3.4. El *protocolo immediate-snapshot de L capas*, para n procesos, es la tupla $\langle I, \mathrm{Ch}^L(I), \mathrm{Ch}^L \rangle$.

Resolución de una tarea

Supongamos que tenemos una tarea $\langle I, O, \Delta \rangle$ y un protocolo $\langle I, P, \Xi \rangle$, ambos para n procesos.

Definición 3.5. Decimos que el protocolo *resuelve* la tarea si existe un mapeo simplicial cromático $\delta : P \to O$, llamado *mapeo de decisión*, tal que $\delta \circ \Xi$ es portado por Δ.

La razón por la que requerimos que δ se un mapeo simplicial cromático es porque cada simplejo en el complejo de protocolo P es una configuración final, esto es, el conjunto de estados finales que puede ser alcanzado en alguna ejecución. Las configuraciones de salida de la tarea son simplejos de O. Si δ mapeara alguna configuración final a un conjunto de vértices que no forman un simplejo en O, quiere decir que esa configuración es el estado final en donde los procesos escogen un valor de una asignación de salida no válida de una ejecución.

Por un lado, I representa el conjunto de asignaciones de entrada de la tarea y por otro el conjunto de configuraciones iniciales del protocolo. En cambio, P representa al conjunto de final de configuraciones y O al conjunto de asignaciones de salida, ambos relacionados por el mapeo de decisión δ.

Con los protocolos podemos estudiar la solubilidad de alguna tarea (one-shot) en el modelo escritura-lectura por capas. Sin embargo, el poder expresivo de estos, no alcanza para estudiar a las tareas refinadas multi-shot. Para para poder hacer compatibles a los protocolos con estas tareas, extenderemos su definición, de la misma forma que lo hicimos con las tareas regulares.

3.2.5. Protocolos multi-shot

Vamos a extender la definición de protocolo de manera que nos permita representar el que un proceso pueda realizar más de una operación. Esta extensión no representa una diferencia técnica significativa y se sigue naturalmente de la definición de una tarea refinada multi-shot. Esto es, los vértices del complejo de entrada I, ahora son coloreados por el identificador del proceso y el valor de entrada (un método u operación). Los vértices del complejo de protocolo P, también son coloreados por el identificador del proceso y una componente extra que contiene el valor de entrada (un método u operación) que le corresponde según el mapeo portador Ξ. El mapeo portador Ξ sigue siendo cromático estricto, considerando la nueva coloración.

Definición 3.6. Un *protocolo* para n procesos, es una tupla $\langle I, P, \Xi \rangle$, donde:

- I es un *complejo simplicial de entrada* puro, cromático, $(n-1)$-dimensional, con una coloración $\chi_I : V(I) \to \Pi \times V_i$, tal que cada vértice es unívocamente identificado por su color;
- P es un *complejo simplicial de protocolo* puro, cromático, $(n-1)$-dimensional, con una coloración $\chi_P : V(P) \to \Pi \times V_i$, y un etiquetado $\phi_P : V(P) \to V_p$, tal que cada vértice es unívocamente identificado por su color y su etiqueta;
- $\Xi : I \to 2^P$ es un mapeo portador cromático estricto tal que $P = \cup_{\sigma \in I} \Xi(\sigma)$.

Podemos componer protocolos multi-shot aplicando directamente la definición 3.3.

La manera en la que escribe un proceso su vista en la memoria necesita un pequeño ajuste considerando esta nueva coloración. Supongamos que al vértice (i, o) del proceso i con operación o, le corresponde el color i_o. Este color lo podemos pensar como un índice tal que en la capa l, el proceso i realiza un immediate-snapshot escribiendo su estado en la memoria $mem[l][i_o]$.

La figura 3.7 representa el protocolo immediate-snapshot de una capa dos procesos a y b que pueden realizar cada uno una de las dos operaciones distintas o y u.

El dominio de valores iniciales es

$$V_i = \{o, u\}$$

Dado que a y b son procesos distintos y pueden correr de forma concurrente, el dominio de valores finales son todas las posibles combinaciones en las que los procesos pueden realizar el immediate-snapshot.

$$V_p = \mathcal{P}(\{(a, o), (b, u)\}) \setminus \emptyset$$

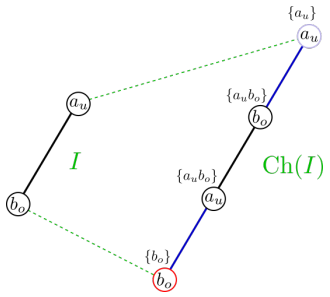

Figura 3.7: Complejos del protocolo immediate-snapshot de una capa, para dos procesos a y b; y dos operaciones u y o. Los símbolos a_u, b_o denotan las tuplas $(a, u), (b, o)$, respectivamente.

Simplificamos la notación codificando la tupla (i, v) con el símbolo i_v donde i es el identificador del proceso y v su operación.

La figura 3.8 representa dos complejos de protocolo immediate-snapshot para las capas 1 y 2, para dos procesos a y b que pueden realizar cada uno una de las dos operaciones distintas o y u.

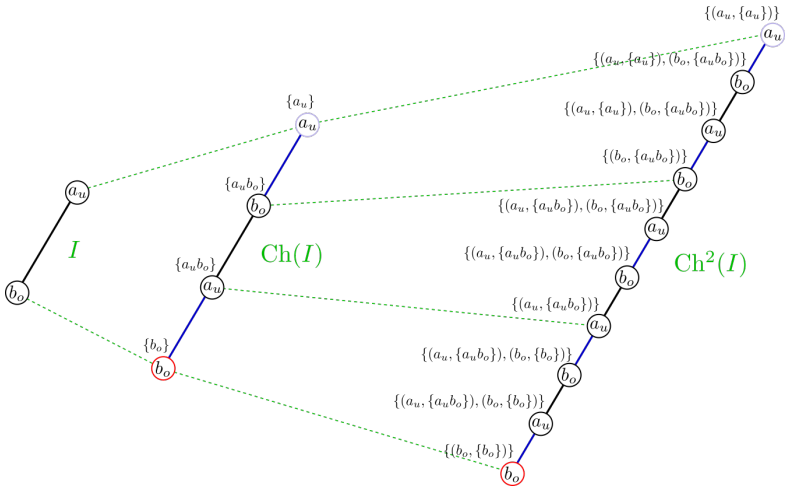

Figura 3.8: Complejos del protocolo de una y dos capas, para dos procesos a y b; y dos operaciones u y o. Los símbolos a_u, b_o denotan las tuplas $(a, u), (b, o)$, respectivamente.

La coloración sobre los valores (id, op) de los vértices, no cambia la representación de las posibles ejecuciones concurrentes de distintos procesos. La subdivisión cromática estándar de I sigue codificando de manera estática, todas las posibles ejecuciones entre procesos distintos. Notemos que los complejos de protocolo siguen teniendo la misma estructura que los de la figura 3.2. Ahora podemos representar un proceso con más de una operación.

La figura 3.9 representa dos complejos de protocolo immediate-snapshot para las capas 1 y 2, para un proceso a y dos operaciones distintas o y u.

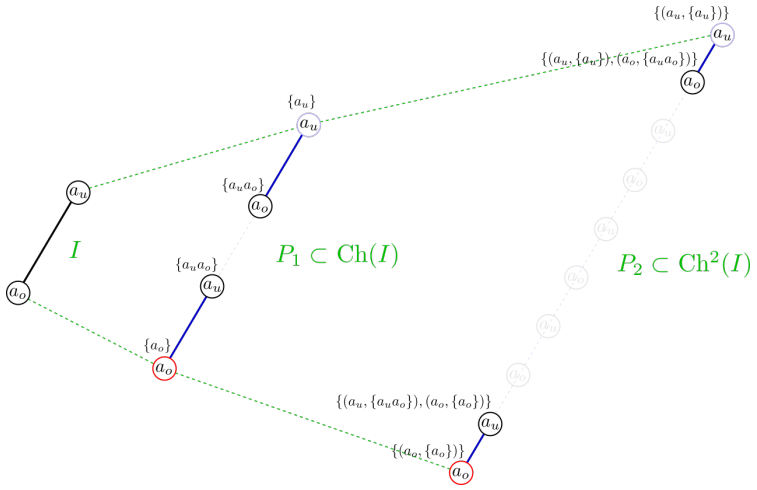

Figura 3.9: Complejos del protocolo de una y dos capas, para un proceso a, y dos operaciones u y o.

Los complejos del protocolo no forman una subdivisión cromática estándar del complejo de entrada. Esto es porque no puede haber ejecuciones concurrentes de un mismo proceso sin importar que operaciones realice. En general cualquier simplejo de protocolo que pudiera representar una ejecución "concurrente" del mismo proceso, no estará presente. Sin embargo, dado que la subdivisión cromática estándar codifica todas las posibles configuraciones finales válidas, el complejo de protocolo deberá ser un subcomplejo de ésta. Dicho de otro modo, para un protocolo de una capa, si σ es un simplejo de entrada, $\Xi(\sigma) \subseteq \mathrm{Ch}(\sigma)$. Además, $\Xi(\sigma) = \mathrm{Ch}(\sigma)$ si todos los id de procesos en σ son distintos.

La figura 3.10 muestra una ejecución concurrente entre dos procesos. En cada capa, cada proceso puede hacer el snapshot antes, después o concurrentemente con el otro proceso. Este orden determina la vista de cada proceso en cada capa.

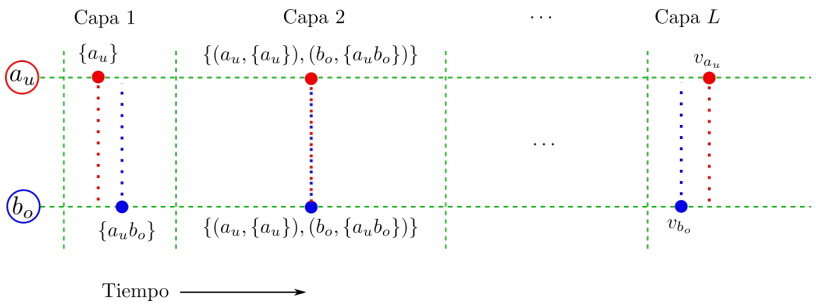

Figura 3.10: Ejecución de L capas del protocolo immediate-snapshot, por dos procesos a y b y dos operaciones distintas u y o. Cada punto representa el immediate-snapshot atómico hecho por cada proceso en la capa correspondiente. La etiqueta de cada punto es la vista del proceso para esa capa.

Cuando la ejecución es de un solo proceso y más de una operación, el proceso debe de ejecutar todas las capas por cada operación distinta. La figura 3.11 muestra la ejecución de un proceso.

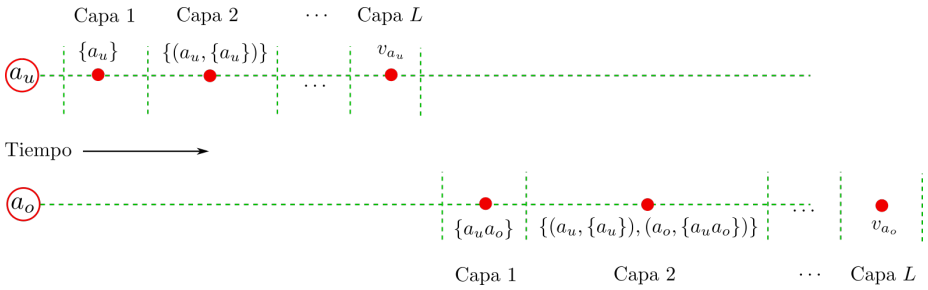

Figura 3.11: Ejecución de L capas del protocolo immediate-snapshot, por un proceso a y dos operaciones distintas o y u. Cada punto representa el immediate-snapshot atómico hecho por el proceso en una capa. La etiqueta de cada punto es la vista del proceso para esa capa.

El proceso a_u (proceso a con operación u) corre solo (o concurrentemente con otros procesos distintos) y en cada capa, la vista no tiene información del proceso a_o. Por otro lado, el proceso a_o, en todas las capas, tiene información de a_u (que corrió primero). En general, en cualquier ejecución del protocolo, un proceso p con dos operaciones distintas i y j, la vista de p_i, en todas las capas o en ninguna, tiene información de p_j.

La figura 3.12 representa el protocolo immediate-snapshot de una capa para dos procesos a y b y dos operaciones u y o. El proceso a puede realizar ambas operaciones y b sólo la operación o.

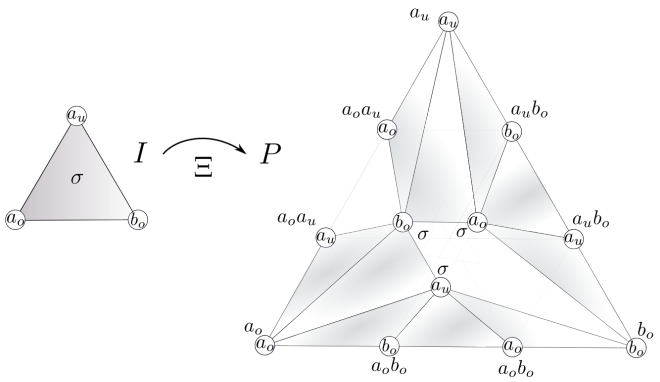

Figura 3.12: Complejos del protocolo de una capa para dos procesos a y b y dos operaciones u y o.

Los simplejos que representan operaciones concurrentes entre a_o y a_u no son configuraciones finales válidas y por lo tanto, no están presentes en el complejo del protocolo.

La figura 3.13 representa el protocolo immediate-snapshot de dos capas para dos procesos a y b y dos operaciones u y o. El proceso a puede realizar ambas operaciones y b sólo la operación o.

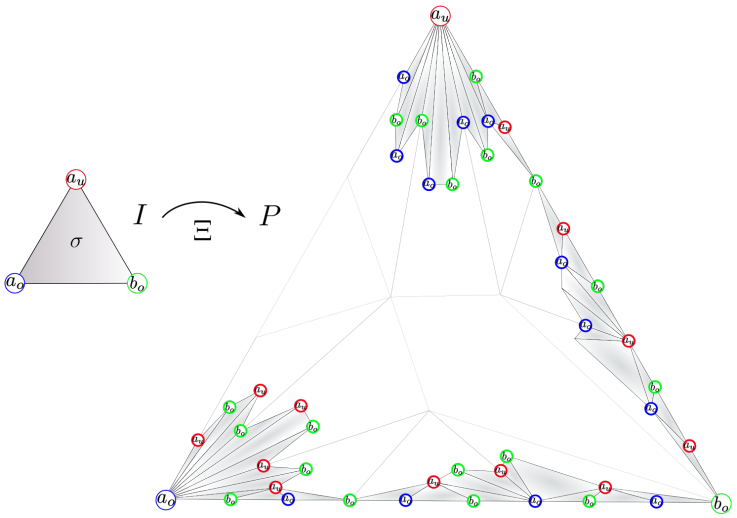

Figura 3.13: Protocolo immediate-snapshot de dos capas para dos procesos a y b y dos operaciones u y o.

El complejo de protocolo P_2 de la figura 3.9 es el subcomplejo $\Xi(a_o a_u)$ que corresponde a la frontera, entre los vértices de las esquinas a_u y a_o, en el complejo de protocolo de la figura 3.13.

Los simplejos 2-dimensionales "anclados" al vértice de la esquina con color a_u (marcados en rojo en la figura 3.14), representan las ejecuciones en solitario de a_u, seguidas de las distintas combinaciones de las ejecuciones en las dos capas, entre a_o y b_o. De la misma manera, los simplejos 2-dimensionales anclados a la esquina a_o (marcados en azul en la figura 3.14), representan las ejecuciones en solitario de a_o, seguidas de las distintas combinaciones de las ejecuciones en las dos capas, entre a_u y b_o.

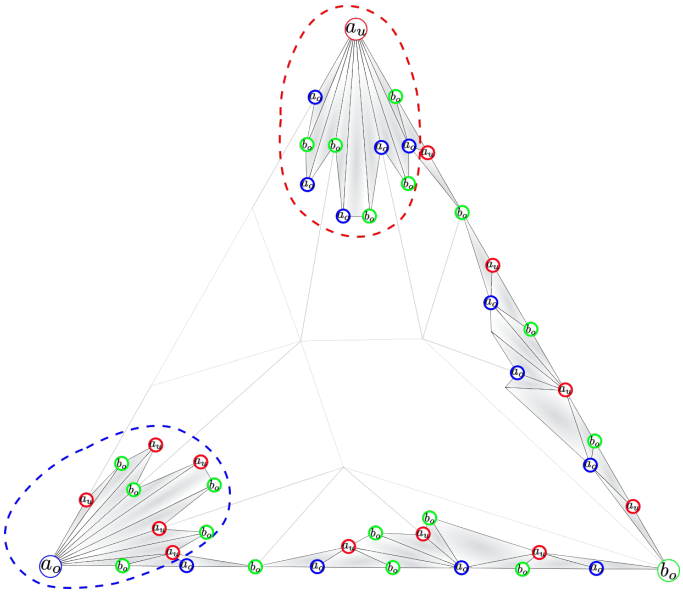

Figura 3.14: Marcadas en rojo, todas las posibles ejecuciones en las que a_u corre en solitario (todas las capas) y antes que a_o y b_o. Marcadas en azul, todas las posibles ejecuciones en las que a_o corre en solitario y antes que a_u y b_o.

Por otro lado, los simplejos 2-dimensionales sobre la frontera, entre las esquinas a_o y b_o (marcados en rojo en la figura 3.15), representan todas las posibles ejecuciones entre a_o y b_o en las dos capas, seguidas de la ejecución de a_u. De la misma manera, los simplejos 2-dimensionales sobre la frontera, entre las esquinas a_u y b_o (marcados en azul en la figura 3.15), representan todas las posibles ejecuciones entre a_u y b_o en las dos capas, seguidas de la ejecución de a_o.

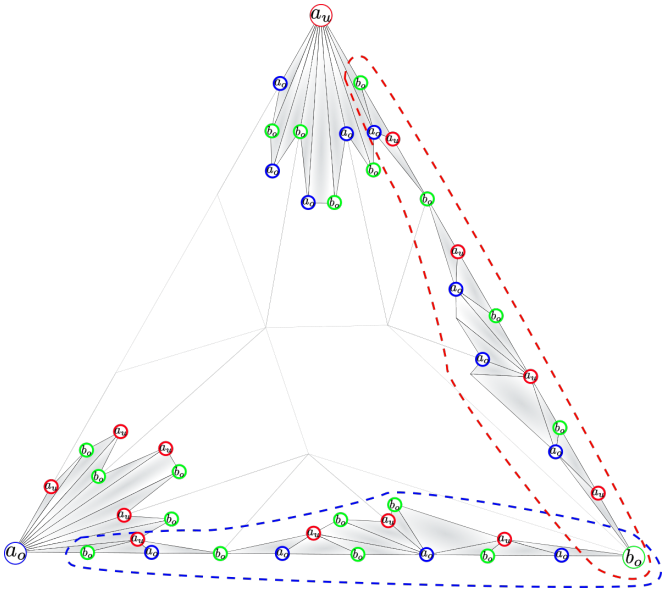

Figura 3.15: Marcadas en rojo, todas las posibles ejecuciones en las que a_u corre en solitario y después que a_o y b_o. Marcadas en azul, todas las posibles ejecuciones en las que a_o corre en solitario y después que a_u y b_o.

En general, el protocolo immediate-snapshot de L capas (figura 3.16) mantiene una estructura similar para $L \geq 2$.

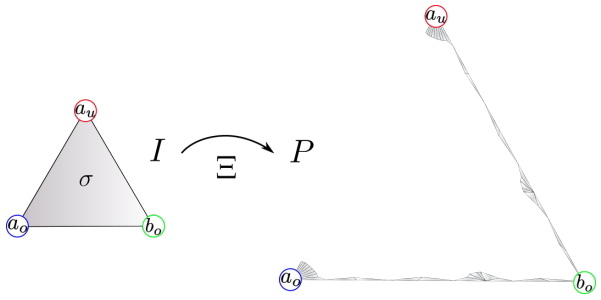

Figura 3.16: Protocolo immediate-snapshot de L capas para dos procesos a y b y dos operaciones u y o.

Esta observación la abordaremos formalmente en el capítulo 4. Con lo anterior podemos extender la noción de resolución para una tarea refinada.

Resolviendo una tarea refinada

Supongamos que tenemos una tarea refinada $\langle I, O, \Delta \rangle$ y un protocolo $\langle I, P, \Xi \rangle$, ambos multi-shot para n procesos.

Definición 3.7. El protocolo resuelve a la tarea si existe un mapeo simplicial cromático $\delta : P \to O$, llamado el mapeo de decisión, tal que:

- $\delta \circ \Xi$ es portado por Δ y
- para todo $\sigma \in I$, para todo vértice $v \in \Xi(\sigma)$, $\text{View}(\delta(v)) \subseteq \text{Carr}(v)$.

Donde $\text{View}(u)$ es la vista del vértice u en el complejo de salida de la tarea refinada. El nuevo requerimiento ($\text{View}(\delta(v)) \subseteq \text{Carr}(v)$) nos garantiza que los valores de salida decididos corresponden al orden de ejecución de los procesos participantes en una ejecución.

Con todo lo anterior, mediante el teorema 3.1 podemos dar una caracterización de la solubilidad de una tarea (regular o refinada) en el modelo de escritura-lectura por capas.

Teorema 3.1 (Teorema 11.2.1 - [1])**.** *Una tarea (regular o refinada) T tiene una implementación wait-free en el modelo de escritura-lectura por capas si existe un entero $K > 0$ tal que el protocolo immediate-snapshot de K capas, resuelve T.*

Usualmente necesitamos encontrar el mapeo decisión entre los complejos de protocolo y de salida para las suficientes K capas de comunicación.

Demostrar que un tarea no puede ser implementada en el modelo de escritura-lectura se reduce a demostrar que el mapeo de decisión no puede existir.

Con esto, en el capítulo 4, podremos establecer la imposibilidad de la implementación de las tareas para las pilas y colas concurrentes de larga vida, en el modelo de escritura-lectura por capas.

3.3. Resumen

Los protocolos son un forma de especificar un algoritmo distribuido para n procesos, considerando un modelo de cómputo determinado. Estos se especifican en términos de complejos simpliciales de entrada y de protocolo, que representan configuraciones iniciales y finales, respectivamente.

Nos interesa estudiar la solubilidad de las tareas en el modelo de escritura-lectura por capas, en el que los procesos se comunican a través de operaciones immediate-snapshot sobre una memoria compartida.

Un protocolo está vinculado a una tarea mediante el complejo de entrada y un mapeo simplicial de decisión, entre el complejo de protocolo y el complejo de salida. Decimos que el protocolo resuelve a la tarea si existe ese mapeo de decisión.

Extendimos la definición de protocolo para poder representar problemas multi-shot. La extensión consiste en considerar cada operación distinta que puede realizar un proceso, como parte de la coloración de los vértices de los complejos de entrada y de protocolo.

En un protocolo immediate-snapshot que resuelve una tarea regular single-shot, el complejo de protocolo es la subdivisión cromática estándar del complejo de entrada. Sin embargo, éste no es el caso para un protocolo que resuelve una tarea refinada multi-shot.

Todo esto nos permite estudiar la solubilidad de una tarea (regular o refinada) en el modelo de escritura-lectura por capas. Con esto podremos, en el capítulo 4, establecer la imposibilidad de la implementación de las pilas y colas concurrentes de larga vida, en este modelo.

Capítulo 4

Resultados

Una tarea tiene una implementación en el modelo escritura-lectura por capas, si un protocolo immediate-snapshot multicapa, la resuelve.

En este capítulo veremos cómo la tarea para una pila concurrente, no tiene una implementación en el modelo escritura-lectura por capas. Para demostrar que es imposible implementar una pila concurrente en este modelo, basta con demostrar que no se puede implementar el siguiente caso: un proceso a puede hacer $pop()$ y $push(x)$ y un proceso b sólo $pop()$.

En una primera instancia, consideraremos una tarea que represente solamente las ejecuciones secuenciales de una pila. Sin embargo, para propósitos de computabilidad, necesitamos considerar todas las posibles ejecuciones concurrentes linealizables de una pila.

Una invariante del protocolo immediate-snapshot multicapa será la clave para poder demostrar que es imposible implementar ambas tareas en el modelo de escritura-lectura por capas.

4.1. Una invariante del protocolo

Consideremos un protocolo immediate-snapshot $\langle I, P, \Xi \rangle$ de L capas, multi-shot. Supongamos que $\Xi = \Xi_L \circ \cdots \circ \Xi_1$, con Ξ_i el mapeo portador del protocolo de la capa i. Recordemos que para un simplejo de entrada en la capa i, cada $\Xi_i(\sigma)$ coincide con $\mathrm{Ch}(\sigma)$, si en σ, todos los id de proceso son distintos.

Una propiedad de Ch^m, ilustrada en la figura 4.1, es que si a y b son dos procesos distintos que realizan las operaciones u y o, el número de aristas de $\mathrm{Ch}^m(a_u b_o)$, es exactamente 3^m y el número de vértices $3^m + 1$. Esto es porque al subdividir cromáticamente cada arista, se introducen dos vértices más, generando tres aristas.

Para esto, basta con que a y b sean distintos y no importa qué operación realiza cada uno. Ya que por cada dos vértices se introducen otros dos, el número de vértices en $\mathrm{Ch}^m(a_u b_o)$ es par y el número de aristas, impar.

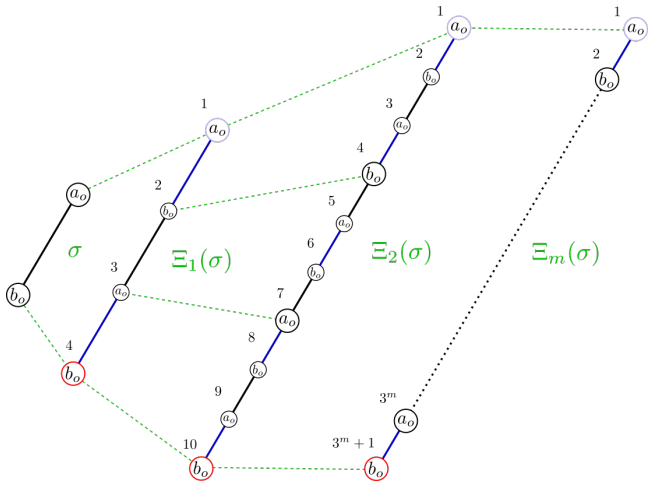

Figura 4.1: El número de vértices de la subdivisión Ch^m para dos procesos distintos, es $3^m + 1$ y el número de 1-simplejos es 3^m.

Por otro lado, una propiedad de Ξ_i, ilustrada en la figura 4.2, es que si p, q, r son tres procesos y σ es un 2-simplejo anclado a la esquina p_o del complejo de entrada de la capa i, $\Xi_i(\sigma)$ introduce tres triángulos α, β y γ, anclados a la esquina p_o, del complejo de protocolo de esa capa. El simplejo α representa la ejecución secuencial en el orden p_o, q_o, r_o, el simplejo β representa la ejecución secuencial en el orden p_o, r_o, q_o y el simplejo γ representa la ejecución de p_o seguida de la ejecución concurrente de r_o y q_o. El simplejo γ estará presente siempre que r y q sean distintos. Si p, q y r son distintos, $\Xi_i(\sigma) = \mathrm{Ch}(\sigma)$.

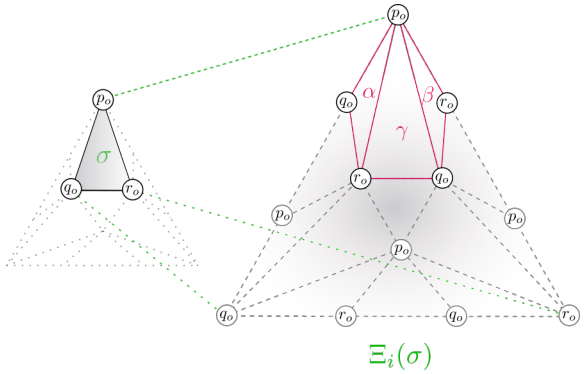

Figura 4.2: Ξ_i introduce tres nuevos triángulos anclados a la esquina p_o, los cuales representan las tres posibles combinaciones en las que p_o corre primero sin saber de los demás en la capa i.

Consideremos el protocolo immediate-snapshot $\langle I, P, \Xi \rangle$ de L capas, de la figura 3.16 (con vértices de entrada a_u, a_o, b_o). Utilizaremos las propiedades anteriores para caracterizar una invariante del protocolo (lema 4.1): un camino l^L sobre los simplejos anclados a la esquina a_u, con colores a_o y b_o, y que es isomorfo a $\mathrm{Ch}^L(a_o b_o)$.

Fijémonos en la región, marcada en la figura 4.3, de los simplejos 2-dimensionales anclados a la esquina del vértice a_u, en el complejo P.

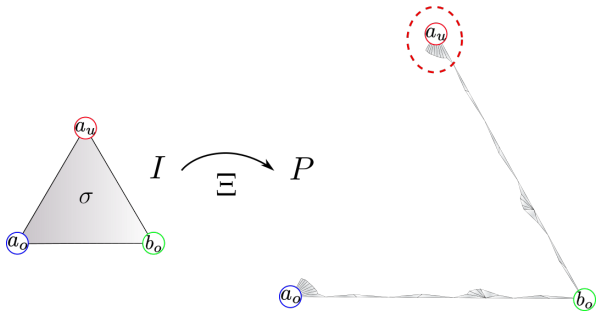

Figura 4.3: Nos fijaremos en la región de los simplejos 2-dimensionales anclados a la esquina a_u.

Todos estos triángulos comparten el mismo vértice con color a_u, son cromáticos con colores a_u, a_o y b_o, y representan ejecuciones en las que corre primero a_u seguido de las distintas combinaciones entre a_o y b_o.

En el protocolo de una capa en la figura 4.4, los simplejos 2-dimensionales anclados al vértice a_u de la esquina de P, representan las tres posibles combinaciones de ejecuciones en las que a_u corre antes que los otros dos. Los simplejos 2-dimensionales, α_1 y β_1, representan las ejecuciones secuenciales en el orden a_u, a_o, b_o y a_u, b_o, a_o, respectivamente. El triángulo entre α_1 y β_1, representa la ejecución de a_u, seguida de

la ejecución concurrente entre a_o y b_o.

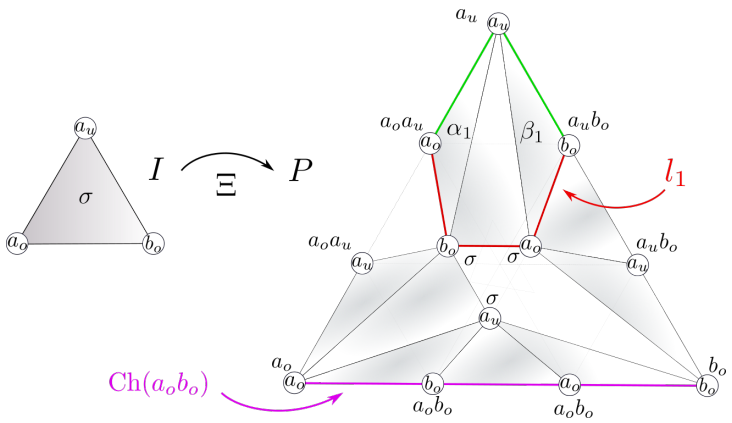

Figura 4.4: El camino $l_1 \cong \mathrm{Ch}(a_o b_o)$, 1 capa.

El camino l_1 que va desde el vértice a_o del simplejo α_1, al vértice b_o del simplejo β_1, es cromático con colores a_o y b_o, y codifica las tres posibles combinaciones del orden de ejecución entre a_o y b_o. De izquierda a derecha en l_1: la primera ejecución es secuencial, a_o seguido de b_o, la segunda concurrente, a_o y b_o, y la tercera también secuencial, b_o seguido de a_o.

El camino l_1 es isomorfo a $\mathrm{Ch}(a_o b_o)$ (proposición 1.1), siendo este último, la frontera inferior (entre las esquinas a_o y b_o) de P.

En el protocolo immediate-snapshot de dos capas en la figura 4.5, los simplejos 2-dimensionales, α_2 y β_2, también representan las ejecuciones secuenciales (ambas capas), en el orden a_u, a_o, b_o y a_u, b_o, a_o, respectivamente.

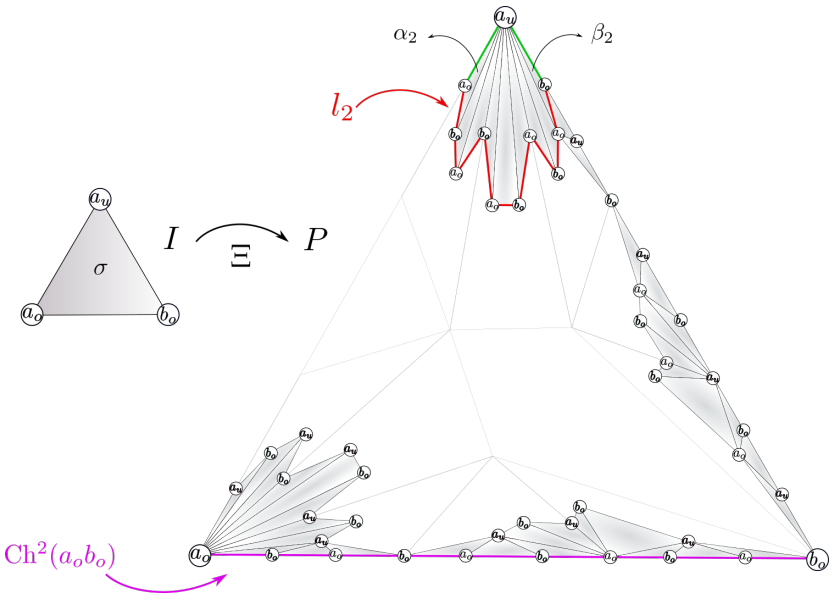

Figura 4.5: El camino $l_2 \cong \mathrm{Ch}^2(a_o b_o)$, 2 capas.

El camino l_2 que va desde el vértice a_o del simplejo α_2, al vértice b_o del simplejo β_2, es cromático con colores a_o y b_o, y codifica todas las posibles combinaciones del orden de ejecución de ambas capas, entre a_o y b_o. Por cada combinación distinta del orden de ejecución entre a_o y b_o en la primera capa (en l_1), tenemos tres distintas combinaciones más en la segunda capa, con un total de 9 distintas combinaciones. Es decir, l_2, tiene 9 simplejos 1-dimensionales y 10 vértices.

Los vértices de l_2 junto con la esquina a_u, son un subcomplejo de las subdivisión cromática estándar de los triángulos con el vértice esquina a_u de la primera capa. El camino l_2 es isomorfo a $\mathrm{Ch}^2(a_o b_o)$ (proposición 1.1).

En general, para el protocolo immediate-snapshot de k capas, con $k \geq 1$, en la figura 4.6, el simplejo 2-dimensional α_k, representa la ejecución secuencial en el orden a_u, a_o, b_o. Es decir, en la que a_u ejecutó las k capas sin saber de los demás procesos, a_o ejecutó las k capas después de a_u y sin saber de b_o, y b_o ejecutó las k capas en último lugar. De manera análoga, el simplejo β_k representa la ejecución secuencial en el orden a_u, b_o, a_o.

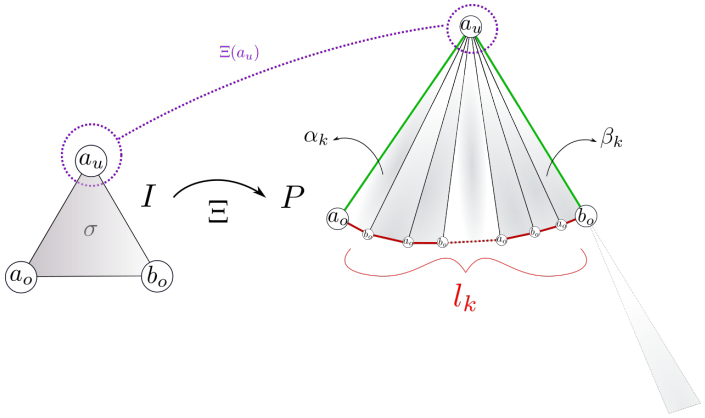

Figura 4.6: El camino $l \cong \mathrm{Ch}^k(a_o b_o)$, k capas.

El camino l_k que va desde el vértice a_o del simplejo α_k, al vértice b_o del simplejo β_k, es cromático con colores a_o y b_o, y codifica todas (3^k) las posibles combinaciones del orden de ejecución de las k capas, entre a_o y b_o. Con el lema 4.1, comprobaremos que l_k existe y es isomorfo a $\mathrm{Ch}^k(a_o b_o)$.

Lema 4.1. *Sea $k \geq 1$, sean α_k y β_k los simplejos 2-dimensionales (figura 4.6) que representan las ejecuciones secuenciales en el orden a_u, a_o, b_o y a_u, b_o, a_o, respectivamente. Del vértice con color a_o del simplejo α_k, al vértice con color b_o del simplejo β_k, existe un camino l_k con exactamente $3^k + 1$ vértices.*

Demostración. Procederemos por inducción sobre el número de capas y sólo consideraremos a los triángulos anclados al vértice esquina a_u.

Para simplificar la notación cambiaremos los colores de los vértices b_o, a_o y a_u, por los colores $0, 1$ y 2, respectivamente, como se ilustra en la figura 4.7. Quitaremos los identificadores de proceso de las vistas para simplificar la notación.

Para el vértice (esquina) 2, w_0 corresponde al valor de entrada. La vista en la capa j la denotaremos con w_j.

Los valores de entrada de 1 y 0 los denotamos con v_1^0 y v_2^0, respectivamente.

Dado que 1 y 0 ejecutan cada capa después de 2, en las vistas de 1 y 0 haremos implícita la información de 2. Por ejemplo, el vértice 1 de α_1, en vez de tener en su vista $w_1 v_1^0$, tendrá sólo v_1^0. De la misma manera, al vértice 0 de α_1, le corresponden los valores v_1^0 y v_2^0.

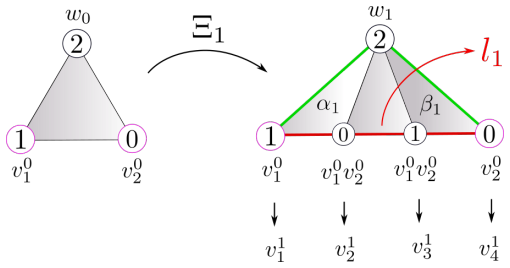

Figura 4.7: El camino l_1 tiene 4 vértices y es cromático con colores $0, 1$.

Las vistas de la capa j tendrán dos formas de codificarse:

1. En términos de los valores de entrada de cada capa, es decir, como combinaciones de valores de la capa anterior. Por ejemplo, en la capa 1, el vértice 0 del simplejo α_1, tiene la vista $v_1^0v_2^0$ (con superíndice 0).

2. Enumerando los vértices de la capa con colores 0 y 1, de modo que si hay m vértices, cada vista será denotada por v_n^j, con $1 \leq n \leq m$. El superíndice de las vistas de 0 y 1, corresponde a la capa de esta numeración. Por ejemplo, las vistas de los 4 vértices con color 0 y 1 de la capa 1 son $v_1^1, v_2^1, v_3^1, v_4^1$.

En la figura 4.8 ilustramos la codificación de las vistas de la segunda capa.

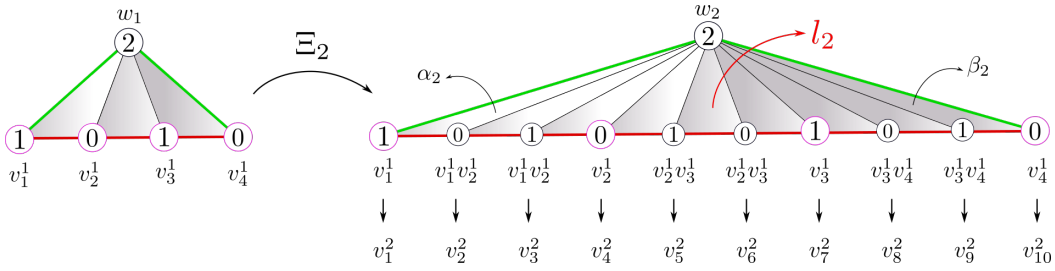

Figura 4.8: El camino l_2 tiene 10 vértices y es cromático con colores $0, 1$.

Teniendo en cuenta esta notación, demostraremos por inducción que el camino l_k existe y tiene exactamente $3^k + 1$.

P.D. Hay un camino l_k del vértice 1 de α_k, al vértice 0 de β_k, cromático con colores 1 y 0, y que tiene exactamente $3^k + 1$ vértices.

Como ya lo vimos con las figuras 4.4 y 4.7, la base de la inducción (capa 1) se cumple.

Procedemos con el paso inductivo, suponemos que es válido para k y demostramos que se cumple para $k + 1$.

Sea l_k el camino del vértice 1 del simplejo α_k, al vértice 0 del simplejo β_k, representado en la figura 4.9, tal que es cromático, con colores 1 y 0, y tiene exactamente $3^k + 1$ vértices y 3^k aristas.

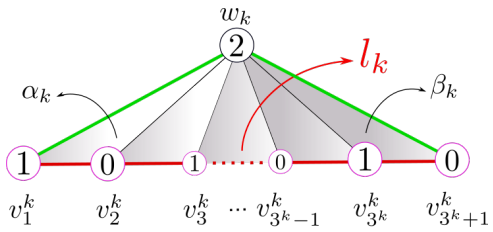

Figura 4.9: El camino l_k tiene $3^k + 1$ vértices y es cromático con colores $0, 1$.

A cada vértice j de l_k, con $1 \leq j \leq 3^k + 1$, le corresponde el id j `mod` 2 y la vista v_j^k, donde `mod` denota la operación de módulo. Cada arista i de l_k, con $1 \leq i \leq 3^k$, tiene la forma:

$$\{(i \texttt{ mod } 2, v_i^k), ((i+1) \texttt{ mod } 2, v_{i+1}^k)\}$$

Sin perder generalidad y para facilitar la notación, podemos suponer que $i \texttt{ mod } 2 = 1$.

En la figura 4.10 podemos apreciar que en la capa $k + 1$, por la propiedad de la figura 4.2, el mapeo Ξ_{k+1}, al subdividir el triángulo que le corresponde a la arista i, introduce los vértices $(1, v_i^k)$, $(0, v_i^k v_{i+1}^k)$, $(1, v_i^k v_{i+1}^k)$, $(0, v_{i+1}^k)$, formando tres nuevas aristas (y sus triángulos correspondientes) cromáticas con colores 1 y 0. De la misma manera, tres nuevas aristas cromáticas son introducidas para los simplejos $i - 1$ e $i + 1$.

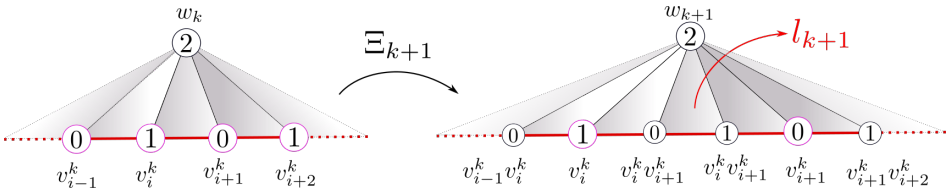

Figura 4.10: El mapeo Ξ_{k+1} del 1-simplejo i, introduce tres nuevas aristas con los vértices $(1, v_i^k), (0, v_i^k v_{i+1}^k), (1, v_i^k v_{i+1}^k), (0, v_{i+1}^k)$. De manera similar lo hace para los 1-simplejos $i-1$ e $i+1$.

Notemos que el vértice $(1, v_i^k)$ pertenece a ambas subdivisiones de los simplejos $i - 1$ e i. De manera similar, el vértice $(1, v_i^{k+1})$ pertenece a ambas subdivisiones de los simplejos i e $i + 1$. Por lo tanto, la conexidad se mantiene en las aristas que pertenecen a los triángulos anclados a 2 en la capa $k + 1$, formando un camino l_{k+1} cromático, con colores 1 y 0.

El número de aristas de l_{k+1} es 3^{k+1}, ya que por cada arista en l_k, se introducen tres nuevas aristas, con lo que el número de vértices es $3^{k+1}+1$.

Como podemos apreciar en la figura 4.11, el triángulo con los vértices $(1, v_1^{k+1})$, $(0, v_2^{k+1})$, $(2, w_{k+1})$, pertenece a la subdivisión de α_k y representa la ejecución secuencial con el orden 2, 1, 0, por lo que lo denotamos con α_{k+1}.

De la misma manera, el triángulo con los vértices $(1, v_{3^{k+1}}^{k+1})$, $(0, v_{3^{k+1}+1}^{k+1})$, $(2, w_{k+1})$, pertenece a la subdivisión de β_k y representa la ejecución secuencial con el orden 2, 0, 1, por lo que lo denotamos con β_{k+1}.

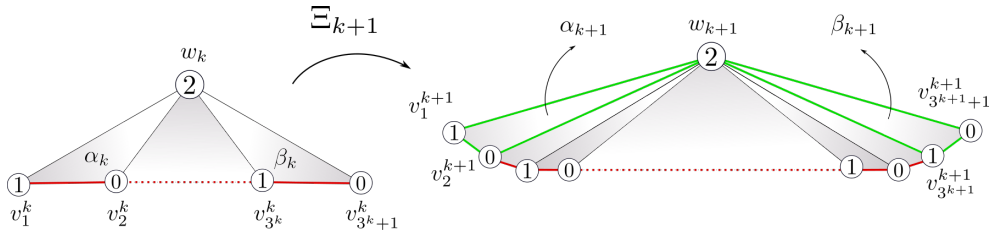

Figura 4.11: Subdivisión de los simplejos α_k y β_k.

Por lo tanto, hay un camino l_{k+1}, del vértice 1 de α_{k+1}, al vértice 0 de β_{k+1}, cromático con colores 0 y 1, y que tiene exactamente $3^{k+1}+1$ vértices.

□

El corolario 4.1 se sigue directamente del lema 4.1 y la proposición 1.1.

Corolario 4.1. *Para $k \geq 1$, el camino l_k es isomorfo a $Ch^m(a_o b_o)$:*

$$l_k \cong \ Ch^k(a_o b_o)$$

El camino l_k nos servirá para demostrar la imposibilidad de implementar una pila concurrente en el modelo de escritura-lectura por capas.

Como una simple observación, $a_u * l_k \cong a_u * \ \mathrm{Ch}^k(a_o b_o)$, aunque no necesitemos este resultado.

4.2. Resultados de imposibilidad

Para demostrar que no es posible implementar una pila en el modelo de escritura-lectura por capas, nos bastará con considerar el caso en el que un proceso a realiza las operaciones $push(x)$ y $pop()$, y un proceso b realiza la operación $pop()$.

Consideremos la tarea $T_{seq} = \langle I, O, \Delta \rangle$, de la figura 4.12, para una pila linealizable en donde el proceso a realiza las operaciones u y o, y el proceso b realiza la operación o. Interpretaremos a la operación u como $push(x)$ y a la operación o como $pop()$.

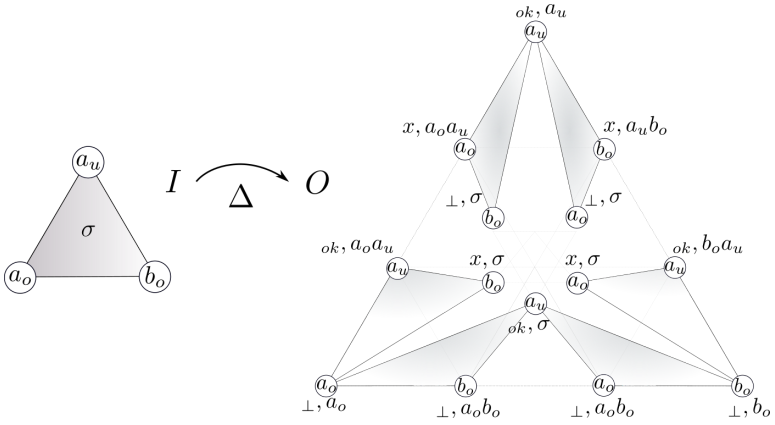

Figura 4.12: Tarea que representa una pila concurrente linealizable con un proceso a realizando las operaciones $u = push(x)$ y $o = pop()$, y un proceso b realizando la operación $o = pop()$.

Teorema 4.1. *La tarea T_{seq} no tiene una implementación wait-free en el modelo de escritura-lectura por capas.*

Demostración por contradicción. Supongamos que T_{seq} tiene una implementación wait-free en el modelo de escritura-lectura por capas. Por el teorema 3.1, existe un entero $k > 0$ tal que el protocolo immediate-snapshot $\langle I, P, \Xi \rangle$ de k capas (multi-shot), con los vértices de entrada a_u, a_o, b_o, resuelve T_{seq}. Es decir, existe un mapeo de decisión $\delta : P \to O$, tal que para cualquier simplejo $\sigma \in I$:

1. $\delta(\Xi(\sigma)) \subseteq \Delta(\sigma)$ y

2. para todo vértice $v \in \Xi(\sigma)$, $\text{View}(\delta(v)) \subseteq \text{Carr}(v)$

Consideremos las figuras 4.12 y 4.13 (donde $\sigma = a_o b_o a_u$ es el 2-simplejo de entrada). La estrategia de la demostración es la siguiente: Veremos que si δ mapea los simplejos $\alpha_k, \beta_k \in P$, a los simplejos $\tau_1, \tau_2 \in O$ (respectivamente), entonces se deriva una contradicción, por lo que δ no puede existir. Teniendo esto, la demostración se reduce a verificar que δ tendría que mapear los simplejos α_k y β_k a los simplejos τ_1 y τ_2.

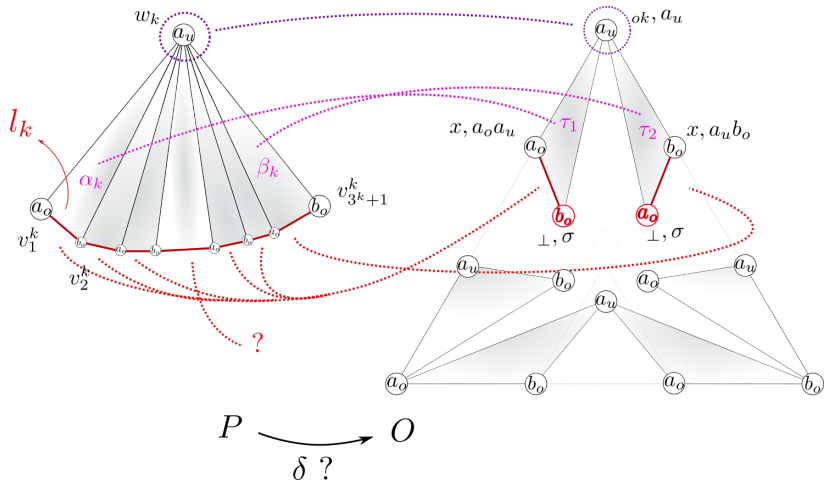

Figura 4.13: δ mapea los simplejos α_k y β_k en los simplejos τ_1 y τ_2, respectivamente.

Supongamos que δ mapea los simplejos α_k y β_k a los simplejos τ_1 y τ_2, respectivamente.

Dado que δ es un mapeo simplicial que preserva colores, hay una arista $\{(b_o, v^k_i), (a_o, v^k_{i+1})\} \in l_k$ (en P), con $1 < i < 3^k$, tal que

$$\delta((b_o, v^k_i)) = (b_o, \perp, \sigma) \text{ y } \delta((a_o, v^k_{i+1})) = (b_o, \perp, \sigma)$$

Notemos que la arista

$$\{\delta((b_o, v^k_i)), \delta((a_o, v^k_{i+1}))\} = \{(b_o, \perp, \sigma), (a_o, \perp, \sigma)\}$$

no existe en O. Esto contradice el hecho de que δ sea un mapeo simplicial.

Con lo anterior, sólo nos queda demostrar que δ tendría que mapear los simplejos α_k y β_k a los simplejos τ_1 y τ_2. Dicho de otro modo, que $\delta(\alpha_k) = \tau_1$ y $\delta(\beta_k) = \tau_2$.

Consideremos el vértice de entrada $a_u \in I$, el vértice (esquina) de salida $(a_u, ok, a_u) \in O$ y el vértice (esquina) de protocolo $(a_u, w_k) \in P$. Por la definición de T_{seq}, $\Delta(a_u) = \{(a_u, ok, a_u)\}$. Por definición del protocolo, $\Xi(a_u) = \{(a_u, w_k)\}$. Por hipótesis, δ es un mapeo de decisión tal que:

- $\delta(\Xi(a_u)) = \delta(\{(a_u, w_k)\}) = \{(a_u, ok, a_u)\} = \Delta(a_u)$
- $\text{View}(\delta((a_u, w_k))) = \text{View}((a_u, ok, a_u)) = a_u = \text{Carr}((a_u, w_k))$

Esto quiere decir que δ mapea el vértice $(a_u, w_k) \in P$, al vértice $(a_u, ok, a_u) \in O$.

Ahora consideremos la arista de entrada $a_o a_u$ y la arista $\{(a_u, w_k), (a_o, v^k_1)\}$ del simplejo α_k. Por la definición de T_{seq}, el subcomplejo $\Delta(a_u a_o)$ es la frontera de O,

entre las esquinas con colores a_u y a_o. Por la definición del protocolo, el subcomplejo $\Xi(a_o a_u)$ es la frontera de P, entre las esquinas con colores a_u y a_o (la esquina a_o no está presente en la figura 4.13).

Por hipótesis:

- $\delta(\{(a_u, w_k), (a_o, v_1^k)\}) \subseteq \Delta(a_o a_u)$
- $\text{View}(\delta((a_u, w_k))) = a_u = \text{Carr}((a_u, w_k))$ y $\text{View}(\delta((a_o, v_1^k))) = a_o a_u = \text{Carr}((a_o, v_1^k))$

Con lo anterior y dado que δ es un mapeo simplicial que preserva colores y además, $\text{View}(\delta((a_o, v_1^k))) = a_o a_u$, entonces

$$\delta((a_o, v_1^k)) = (a_o, x, a_o a_u)$$

y por lo tanto

$$\delta(\{(a_u, w_k), (a_o, v_1^k)\}) = \{(a_u, ok, a_u), (a_o, x, a_o a_u)\}$$

Esto quiere decir que δ mapea la arista con colores a_o y a_u de el simplejo $\alpha_k \in P$, en la arista con colores a_o y a_u de el simplejo $\tau_1 \in O$.

Siguiendo el mismo razonamiento tenemos que

$$\delta(\{(a_u, w_k), (a_o, v_1^k), (b_o, v_2^k)\}) = \{(a_u, ok, a_u), (a_o, x, a_o a_u), (b_o, \perp, \sigma)\}$$

esto quiere decir que $\delta(\alpha_k) = \tau_1$.

Por un razonamiento similar tenemos que $\delta(\beta_k) = \tau_2$

Por lo tanto, δ mapea los simplejos α_k y β_k a los simplejos τ_1 y τ_2.

□

La tarea T_{seq} representa de manera estática a todas las posibles ejecuciones secuenciales de una pila. Por la definición de satisfacción de una tarea, una ejecución concurrente linealizable puede satisfacer T_{seq}. Por esto, en un contexto de mera especificación de un problema distribuido, en T_{seq} nos basta con que estén representadas sólo las ejecuciones secuenciales. Sin embargo, en un contexto de análisis de computabilidad, necesitamos considerar todas las ejecuciones concurrentes válidas.

La tarea $T_{conc} = \langle I, O, \Delta \rangle$, de la figura 4.14, codifica de manera estática todas las posibles ejecuciones concurrentes que son linealizables, de una pila, en donde el proceso a realiza las operaciones $u = push(x)$ y $o = pop()$, y el proceso b realiza la operación $o = pop()$.

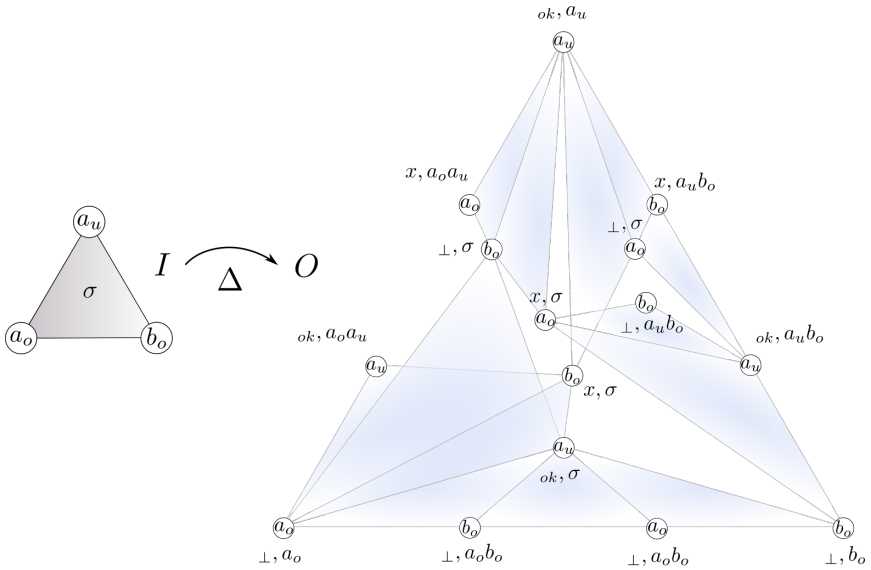

Figura 4.14: Tarea que representa una pila concurrente con un proceso a realizando las operaciones $u = push(x)$ y $o = pop()$, y un proceso b realizando la operación $o = pop()$.

La tarea T_{conc} modela de manera fiel el espacio topológico de todas las ejecuciones concurrentes linealizables de una pila. Con el teorema 4.2 veremos cómo, aún con esta consideración, una pila concurrente sigue siendo imposible de implementar en el modelo de escritura-lectura por capas.

Teorema 4.2. *La tarea T_{conc} no tiene una implementación wait-free en el modelo de escritura-lectura por capas.*

Demostración. La demostración es similar a la del teorema 4.1, considerando al figura 4.15 y utilizando el mismo razonamiento. El mapeo de decisión δ, mapea los simplejos α_k y β_k, a los simplejos τ_1 y τ_2. Además, hay una arista $\{u_1, u_2\} \in l_k$ tal que $\delta(u_1) = (a_o, x, \sigma)$ y $\delta(u_2) = (b_o, x, \sigma)$ pero que $\delta(\{u_1, u_2\}) \notin O$. Todo esto deriva en una contradicción por la suposición de que δ existe y por tanto, de que T_{conc} tiene una implementación en el modelo.

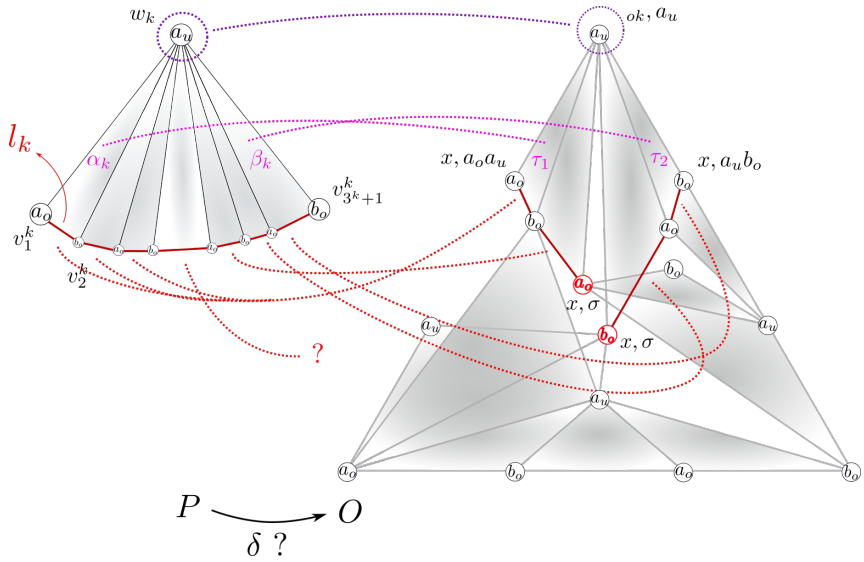

Figura 4.15: δ mapea los simplejos $\alpha_k, \beta_k, \alpha$ y β en los simplejos τ_1, τ_2, τ'_1 y τ'_2.

□

4.3. Resumen

En este capítulo presentamos el resultado principal de este trabajo: es imposible que exista una implementación de las pilas y colas concurrentes, en el modelo de escritura-lectura por capas.

Este resultado es conocido en el contexto operacional y se obtiene utilizando *números de consenso* [2]. Sin embargo, el aporte principal de este trabajo es la obtención de este resultado aplicando el enfoque topológico por primera vez, a pilas y colas concurrentes de larga vida.

Vimos cómo el protocolo immediate-snapshot multicapa, con entradas a_o, a_u, b_o, tiene una propiedad invariante al número de capas.

Utilizamos esta propiedad para demostrar que no es posible que exista un mapeo de decisión entre el complejo del protocolo y los complejos de salida de dos tareas determinadas.

- La primera, una tarea con entradas a_o, a_u, b_o, que representa solamente, todas las ejecuciones secuenciales de una pila. Sin embargo, en un contexto de análisis de computabilidad, necesitamos considerar todas las ejecuciones válidas.

- La segunda, una tarea con entradas a_o, a_u, b_o, que representa todas las posibles ejecuciones concurrentes linealizables de una pila.

Capítulo 5

Conclusiones

En el capítulo 1 vimos cómo la linealizabilidad caracteriza el comportamiento concurrente y correcto de un objeto en términos de un comportamiento secuencial equivalente. Además, definimos los elementos necesarios que permiten la especificación de problemas distribuidos, en el marco de la topología combinatoria.

En el capítulo 2 vimos cómo una tarea es una manera estática de especificar un problema distribuido con una operación que puede ser invocada sólo una vez por cada proceso. Una limitación importante de las tareas es que carecen del poder expresivo para representar pilas y colas concurrentes. Las tareas refinadas son una extensión de las tareas, con un mayor poder expresivo, con las que podemos representar pilas y colas concurrentes. También establecimos una noción de satisfacción para una tarea, por una especificación secuencial.

En el capítulo 3 vimos cómo los protocolos son un forma de especificar un algoritmo distribuido, considerando un modelo de cómputo determinado. Definimos el protocolo immediate-snapshot multicapa, para el modelo de escritura-lectura por capas, en donde los procesos se comunican realizando immediate-snapshots sobre una memoria compartida. Extendimos la definición de protocolo para poder representar problemas multi-shot. Establecimos una noción de resolución de una tarea por un protocolo, en términos topológicos. Esto establece las condiciones necesarias para que un objeto concurrente pueda ser implementado en el modelo de cómputo considerado. A groso modo, una tarea tiene una implementación wait-free en el modelo de escritura-lectura por capas si el protocolo immediate-snapshot multicapa, resuelve la tarea.

En el capítulo 4, utilizando argumentos topológicos, demostramos que es imposible que exista una implementación de las pilas y colas concurrentes, en el modelo de escritura-lectura por capas.

Vimos cómo el protocolo immediate-snapshot multicapa tiene una propiedad invariante al número de capas. Esta propiedad nos permitió demostrar que no es posible que exista un mapeo de decisión entre el complejo de protocolo y el complejo de salida de una tarea que representa a una pila o cola concurrente. Por lo tanto no existe una implementación wait-free, de las pilas y colas, en el modelo de escritura-lectura por capas.

El principal aporte de esta tesis es que no existe un trabajo con el enfoque que seguimos en el estudio, en el marco topológico, de las pilas y colas concurrentes de larga vida.

Un trabajo similar se hace en [7], aunque con un enfoque más de computabilidad. A groso modo, el autor toma un objeto O y construye una tarea T_O para demostrar que existe un algoritmo, en el modelo immediate-snapshot iterado, para O, si y solo si existe un algoritmo para T_O. Además, sólo estudia el caso en el que los procesos invocan una sola operación de O. Mientras que nosotros, tomamos un objeto O y construimos una tarea T_O para demostrar que un algoritmo soluciona O, si y sólo si soluciona T_O. Además, no tenemos una restricción en el número de invocaciones por proceso.

En general, el estudio de objetos concurrentes, en el marco topológico, es un área de estudio poco explorada. La extensión a este trabajo, en un futuro, puede ser la aplicación de esta metodología en el estudio de diferentes objetos concurrentes.

Bibliografía

[1] M. Herlihy, D. Kozlov, S. Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann is an imprint of Elsevier, (2014).

[2] M. Herlihy, N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann is an imprint of Elsevier, (2008).

[3] M. Herlihy. J. M. Wing. *Linearizability: A Correctness Condition for Concurrent Objects*. ACM, (1990).

[4] M. Herlihy, N. Shavit. *The Topological Structure of Asynchronous Computability*. Journal of the ACM, Vol. 46, No. 6, pp. 858 –923, (1999).

[5] A. Castañeda, S. Rajsbaum, M. Raynal. *Unifying Concurrent Objects and Distributed Tasks: Interval-linearizability*. Journal of the ACM (JACM) Volume 65 Issue 6, Article No. 45, (2018).

[6] H. Cárdenas, E. Lluis, F. Raggi, F. Tomas. *Álgebra superior*. Trillas, (2015).

[7] E. Gafni. *Snapshot for Time: The One-Shot Case*. van Glabbeek R.J., On the expressiveness of higher dimensional automata. Theoretical Computer Science, 356(3):265–290, (2014).