Miscelánea Matemática **56** (2013) 113–144 SMM

Computabilidad distribuida

Maurice Herlihy
Department of Computer Science,
Brown University, USA
mph@cs.brown.edu

Sergio Rajsbaum
Instituto de Matemáticas, UNAM,
México City, D.F. 04510, México
rajsbaum@math.unam.mx

Michel Raynal
Institut Universitaire de France,
IRISA, Univ. Rennes 1, 35042 Rennes, France
raynal@irisa.fr

Resumen

La computabilidad clásica se entiende a través de la tesis de *Church-Turing*: todo lo que puede ser computado, puede ser computado por una máquina de Turing. Más precisamente, en su artículo original, Turing introdujo lo que ahora se conoce como máquina de Turing, para formalizar la noción de *funciones* computables. Esta tesis se cumple inclusive cuando hay paralelismo en el sistema: es posible incrementar la velocidad, pero no la clase de funciones computables. Estos resultados son bien conocidos. Mucho más reciente, y menos conocido, es que cuando puede haber fallas en algunos de los componentes de un sistema, cambia el carácter de la computabilidad. Las entradas y salidas de la función están distribuidas entre los procesos del sistema, y al problema que define cuales son las salidas legales para cada entrada se le llama *tarea*, el equivalente de una función en el contexto de computabilidad distribuida.

*Se agradece el apoyo del Proyecto UNAM-PAPIIT IN104711 y LAISLA.

La computabilidad distribuida de una tarea depende de las propiedades del sistema: el tipo de comunicación, las velocidades relativas de los procesos, y las fallas que pueden ocurrir en los procesos y en el medio de comunicación. Sin embargo, es posible describir los fundamentos de computabilidad distribuida utilizando un modelo canónico, el llamado IWS, que se describe en este trabajo. Se presentan ejemplos de tareas y el teorema que caracteriza las tareas que tienen solución en el modelo IWS. Se explica el rol fundamental de la topología en la teoría de la computabilidad distribuida.

1. Introducción

En 1936, cuando tenía 24 años de edad, Alan Turing hizo un descubrimiento fundamental. Probó algo muy extraño: existen funciones que pueden ser descritas, y sin embargo no pueden ser computadas mediante ninguna secuencia de pasos bien definida. Esto sorprendió a todos, especialmente a los matemáticos de la época, que estaban buscando algoritmos para calcular polinomios. Concretamente, el famoso décimo problema de Hilbert [**42**], pedía encontrar un método mediante el cual se pudiera decidir, en un número finito de pasos, si un polinomio con coeficientes enteros tiene solución en enteros. El resultado de Turing implicaba que no existe tal algoritmo. Para obtener esta conclusión, negativa, Turing tuvo que definir una noción formal de "método", de "secuencia de pasos bien definida". Es esta noción la que hoy se conoce como *máquina de Turing*, y la cual nos lleva a la *tesis de Church-Turing:*

> Todo lo que puede ser computado, puede ser computado por una máquina de Turing.

Computación distribuida. En 1993 se anuncia en el Congreso Anual de Teoría de la Computación de la ACM [**10**, **31**, **48**] el descubrimiento de un segundo fenómeno extraño. Mientras que en los sistemas secuenciales la computabilidad se entiende a través de la tesis de Church-Turing, en los sistemas distribuidos cuyos componentes pueden fallar, en los que el cómputo requiere de coordinación entre participantes, los aspectos de computabilidad adquieren una faceta distinta. Aquí también hay muchos problemas que no son computables; no obstante, estos límites a la computabilidad reflejan la dificultad de tomar decisiones frente a la imposibilidad de conocer el estado global del sistema, y en realidad tienen muy poco que ver con la capacidad computacional inherente de cada uno de los participantes (que

son procesos secuenciales). Más aun, los argumentos de imposibilidad de cómputo distribuido tienen que ver con argumentos geométricos, de tipo topológico, en contraste con los argumentos de imposibilidad de cómputo secuencial, que tienen que ver con argumentos de conteo y diagonalización.

Si los participantes se pudieran comunicar de manera confiable entre sí, cada uno podría ensamblar el estado completo del sistema y el cómputo podría seguir de manera secuencial. En efecto, los resultados clásicos de computabilidad nos dicen que un modelo de cómputo paralelo puede resolver problemas más rápidamente que uno secuencial, pero no puede resolver más problemas. Una máquina de Turing de k cintas puede ser simulada por la usual de una cinta, e.g. [**49**]. Así mismo, el modelo paralelo que fue el originalmente más estudiado, el PRAM, e.g. [**24**] no tiene más poder de computabilidad que el secuencial, RAM, y ambos son equivalentes en cuanto a los problemas que son computables, a una máquina de Turing. No obstante, en cualquier modelo realista de computación distribuida, cada participante conoce al inicio solo una parte del estado global del sistema; las incertidumbres ocasionadas por fallas, así como la incapacidad de predecir la velocidad de procesamiento y comunicación de los componentes del sistema, limitan a cada participante, proporcionándoles una imagen incompleta del estado global del sistema, que continúa más allá del inicio del cómputo. Algunos libros de texto de computación distribuida son [**7**, **32**, **41**, **47**, **50**].

Problemas secuenciales y distribuidos. En computación secuencial, el estudio de las *funciones* es una de las grandes preocupaciones. Una máquina de Turing inicia con una sola entrada, computa por una duración finita y se para con una sola salida. No cualquier función es computada por una máquina de Turing; de hecho, desde el artículo original de Turing [**51**], se sabe que son mucho más las funciones no computables que las computables: la cardinalidad del primer conjunto es no numerable, mientras que la del segundo sí lo es.

En computación distribuida, el análogo de una función es llamado *tarea*. En este caso, la entrada se divide entre los procesos: de manera inicial, cada proceso conoce su propio valor de entrada, sin conocer los demás valores. Cuando cada proceso computa, se comunica con los otros y finalmente se para con su propio valor de salida. (Podemos pensar que cada proceso tiene su propia cinta, que contiene una parte de la entrada al inicio, sirve de almacenamiento, y finalmente contiene su salida). De manera colectiva, los valores de salida individuales forman la salida de la tarea. En la figura 1, para una tarea para tres procesos, se representa

una entrada que tiene tres partes, la parte x_i le toca al proceso p_i, para $i = 1, 2, 3$. La salida representada también tiene tres partes, el proceso p_i produce la salida y_i. A diferencia de una función, la cuál de manera determinista transporta un solo valor de entrada hacia un solo valor de salida, las especificaciones de tareas interesantes son típicamente no deterministas, a fin de dar cabida al no-determinismo introducido por las fallas y la asincronía. En el ejemplo de la figura, para la entrada x_1, x_2, x_3, la salida y_1, y_2, y_3 es válida, pero podría ser que hubiera otras salidas que también son válidas para esa entrada.

El ejemplo más importante de tarea es la del *consenso*. Cada proceso comienza con una entrada, que es cualquiera de los elementos de un conjunto fijo de posibles valores de entrada. Cada proceso "sabe" su valor de entrada, pero no el de los otros procesos. Cada proceso debe producir como salida un valor, tal que todos los procesos producen el mismo valor. Además, el valor decidido debe ser igual al valor de entrada de uno de los procesos. La tarea es no determinista: cuando los procesos tienen dos valores diferentes de entrada, ambos son valores de salida legales. En un sistema sin fallas, la solución es simple: cada proceso envía su valor de entrada a todos, y espera a recibir los valores de entrada de los demás procesos. Una vez que ya recolectó todos los valores de entrada, produce como salida el menor de ellos. En un sistema en el cual puede haber fallas de comunicación y asincronía, la situación se vuelve interesante. Cuando dos procesos intentan recolectar los valores de entrada de otros procesos, es posible que escuchen de distintos grupos de procesos, debido a fallas, y no coincidan en el menor valor de entrada. De hecho, se puede demostrar que cuando cada proceso corre a distinta velocidad, impredecible, y los procesos pueden fallar, la tarea del consenso no se puede resolver [**17**]. Uno de los objetivos de este trabajo es explicar por que no se puede resolver la tarea de consenso en algunos sistemas distribuidos y sí en otros, y más en general, presentar un panorama de la variedad de tareas que se han estudiado y su computabilidad distribuida.

Modelos distribuidos. El objetivo de un modelo distribuido, tal y como ocurre con la máquina de Turing, no es tratar de presentar todos los detalles de la forma en la que funciona un sistema real. En lugar de eso, al iniciar con una abstracción limpia y básica, podemos concentrarnos en las propiedades esenciales de la computación distribuida.

Un sistema distribuido es un conjunto de máquinas secuenciales llamadas *procesos*. No debemos suponer que cada proceso es una máquina de Turing. Los límites inherentes a la computabilidad distribuida no

cambian, incluso si cada proceso fuese un autómata con un número infinito de estados, capaz de computar funciones no Turing-computables. Un *protocolo* también llamado *algoritmo distribuido* consiste de una colección de algoritmos secuenciales, uno para cada proceso. Cada algoritmo secuencial, además de las instrucciones usuales, incluye instrucciones para comunicarse con otros procesos.

Un proceso puede *fallar*. Únicamente consideramos las *fallas por paro*: El proceso con falla simplemente se detiene y no avanza. En otros modelos que no estamos considerando en este trabajo, los procesos pueden presentar incluso fallas bizantinas [**38**], en las que pueden presentar conductas arbitrarias, maliciosas. Supondremos que tenemos un modelo en donde cada subconjunto del conjunto de procesos puede sufrir una falla de paro en su ejecución. Es decir, cualquiera de los procesos puede fallar, de forma independiente. En situaciones como esta, donde cualquier número de procesos puede fallar, se dice que el modelo es *sin espera*, ya que no tiene sentido que un proceso espere a recibir información de otro, debido a que siempre es posible que el proceso falle y la información nunca llegue. Como veremos, hay muchas tareas interesantes que se pueden resolver bajo estas condiciones, tan difíciles. Los algoritmos distribuidos sin espera, tienden a ser más complicados, pero tienen la ventaja de que cada proceso corre a su máxima velocidad, la cuál nunca es afectada por la velocidad de procesos lentos.

Existen muchos modelos de comunicación posibles para la computación distribuida. En este trabajo, suponemos que los procesos se comunican de la manera más sencilla, leyendo y escribiendo en una memoria compartida. Otros modelos populares como el de paso de mensajes o varios modelos de redes que limitan la conectividad directa de proceso a proceso, son equivalentes o menos poderosos que los de memoria compartida. Elegimos el modelo de comunicación de memoria compartida porque su sencillez hace resaltar las propiedades fundamentales de la computación distribuida. De hecho, los sistemas modernos multiprocesadores usualmente ofrecen algún tipo de abstracción de memoria compartida (aunque nuestro modelo abstrae muchas detalles importantes de un modelo real). Por otro lado, se pueden obtener resultados sobre otros modelos, que representan a sistemas reales que proveen de objetos compartidos más poderosos que de lectura y escritura, a partir de este modelo, mediante reducciones y simulaciones: [**12**, **29**].

Existen varios modelos de temporización. En un modelo *síncrono* los procesos ejecutan sus operaciones a la misma duración; mientras que en el modelo *asíncrono*, las velocidades relativas de ejecución son arbitrarias. Los modelos semi-síncronos, donde existen cotas sobre estas

velocidades, y sobre el tiempo de transmisión de los mensajes, es el más realista. En este trabajo utilizamos el modelo asíncrono, ya que es el más fácil de analizar, y los resultados sirven de base para analizar los otros modelos.

Es preciso hacer notar que las fallas y la asincronía interactúan entre sí: si un proceso falla y no recibe un mensaje de otro proceso, puede ser que el otro proceso haya fallado, o tal vez se deba a que el proceso emisor es demasiado lento. Esta ambigüedad es la raíz de los problemas de computabilidad en la computación distribuida.

En resumen, aunque existen muchos modelos de sistemas distribuidos, es posible presentar los resultados básicos de computabilidad distribuida, utilizando un modelo canónico, llamado IWS. Este modelo fundamental de cómputo distribuido consiste de un conjunto de procesos secuenciales, asíncronos, de los cuales cualquier número puede fallar (deteniéndose), y los cuales se comunican mediante operaciones de lectura y escritura sobre una memoria compartida. El análisis se simplifica si suponemos que el modelo es *iterado* [**44**], de manera que la memoria esta organizada en filas, y los procesos pueden leer y escribir una sola vez sobre cada fila, y en el mismo orden. Ver libros de texto como [**7**, **32**, **41**], o artículos como [**6**, **18**, **31**] para obtener más información sobre este modelo y los modelos relacionados.

Propiedades del cómputo distribuido La teoría de computación distribuida estudia tareas, que pueden ser computables incluso en presencia de fallas y retrasos en la comunicación. Los modelos que considera producen una estructura matemática rica, relacionada intrínsecamente con las nociones de topología combinatoria. De manera específica:

1. Hay problemas sencillos, computables secuencialmente que no son computables por un sistema distribuido incluso en presencia de una sola falla. Por ejemplo, resolver la tarea de consenso resulta trivial si no hay fallas; sin embargo es imposible de resolver si un proceso puede fallar.

2. La pregunta de si una tarea es computable por dos procesos, puede reducirse a un problema de conectividad en gráficas y por lo tanto es decidible [**8**, **17**]. Esta situación es similar a lo que ocurre cuando en un sistema de varios procesos puede fallar solamente uno (cualquiera de ellos, pero solo uno).

3. Sin embargo, la pregunta de si una tarea es computable en presencia de cualquier número de fallas, se tiene que ver con la pregun-

ta de si una estructura geométrica asociada, llamada complejo simplicial tiene "agujeros" de cierta dimensión; en este caso el problema es indecidible [**22**, **33**].

4. Finalmente, de manera similar a los oráculos de la computabilidad clásica, existen tareas que son computables solo cuando se da acceso a un oráculo distribuido para otras tareas, llevando a jerarquías de tareas infinitas, e.g. [**18**, **26**, **39**].

En este trabajo solamente se describe el resultado fundamental de computabilidad distribuida, que caracteriza las tareas que tienen solución en el modelo IWS, como un problema de topología. Sin embargo, las ideas presentadas son la base para obtener otros resultados, como los arriba enumerados.

Organización. El resto de este artículo está organizado de la siguiente manera: La sección 2 describe nociones básicas, de notación y de modelos de cómputo distribuido. La sección 3, introduce la noción de *tarea*, la unidad básica de la computación concurrente. El modelo IWS, simple, y a la vez fundamental en el sentido de que incluye las propiedades comunes a otros modelos, se describe en la sección 4. La sección 5 introduce conceptos de la topología combinatoria que revelan el poder de cómputo del modelo fundamental. Se concluye el trabajo en la sección 6.

2. Fundamentos

Consideramos $n + 1$ procesos secuenciales, denotados por $p_0, \ldots, p_n$, que se comunican por medio de una memoria compartida que provee de operaciones de lectura y escritura. Cada una de las ubicaciones en la memoria, también llamadas *registros atómicos* [**37**], son escritas por un solo proceso y son leídas por todos los procesos. Los procesos pueden fallar por *paro*: el proceso se detiene y no ejecuta ningún paso más. Un proceso *sin falla* es aquél donde no hay paro. Si el proceso entra en paro antes de ejecutar algún paso, decimos que no participa en la ejecución. Un *protocolo* define un programa secuencial para cada proceso, en el que cada uno inicia con una entrada privada, se comunica de manera repetida con los demás y se detiene con una salida. Un protocolo es *sin espera* si todos los procesos sin falla terminan en un número de operaciones que es independiente de las velocidades relativas o fallas de los procesos [**25**]. En la sección 4 se define en más detalle el modelo que utilizaremos.

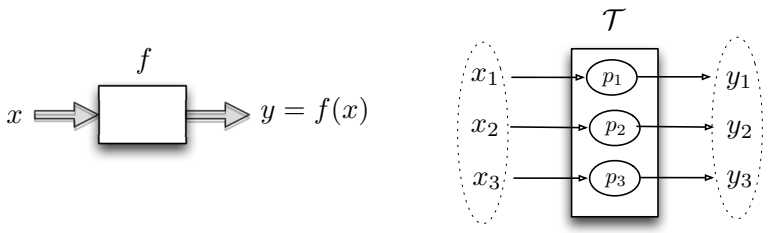

Figura 1: Función f vs. tarea $\mathcal{T}$ para tres procesos p_1, p_2, p_3.

Como ya se discutió en la Introducción, se utiliza este modelo por que es básicamente equivalente a otros modelos comunes, por ejemplo, en los cuales la comunicación es mediante mensajes, en lugar de mediante una memoria compartida. También, otros modelos más poderosos (con menos posibilidad de fallas o mecanismos de comunicación más fuertes), se pueden analizar mediante reducciones a nuestro modelo.

Aun cuando los procesos pueden fallar, suponemos que la memoria es confiable. En otros trabajos [**23**, **47**] se han considerado modelos en los que es necesario construir una memoria confiable a partir de componentes poco confiables.

El problema básico de la computación distribuida es llamado *tarea*, y cumple con el mismo papel que una función en la computación secuencial. En una tarea $\mathcal{T}$, se le asigna de manera inicial a cada uno de los $n+1$ procesos un valor de entrada y cada uno de los procesos (que no falle) debe elegir un valor de salida, de manera irrevocable. La especificación de la tarea determina cuales salidas están permitidas para cada entrada. Al inicio, cada proceso desconoce los valores de entrada de los demás. Nuestra definición formal de tarea se presenta en la sección 3, utilizando las siguientes nociones elementales tomadas de topología, que se definen en [**36**] en un contexto matemático, y también en [**6**, **18**, **31**] donde se usan en un contexto de computación distribuida.

Consideremos una ejecución en la que cada proceso p_i tenga un valor de entrada x_i. Podemos definir al estado inicial de cada proceso mediante una pareja, (p_i, x_i), a la que llamaremos *vértice*. Si solo los procesos $p_{i_0}, \ldots, p_{i_k}$ participan en una ejecución, entonces el estado inicial del sistema es caracterizado por el conjunto de parejas iniciales de los procesos de participación:

$$s = \{(p_{i_0}, x_{i_0}), \ldots, (p_{i_k}, x_{i_k})\}.$$

A este conjunto de parejas lo llamamos *simplejo*. (Notar que los valores iniciales asignados a los procesos no participantes son irrelevantes, ya que no tienen efecto en los valores elegidos por los procesos participan-

tes). Denotamos $nombres(s)$ al conjunto de procesos en s, y $vistas(s)$ al conjunto de sus valores. El estado local de un proceso en un momento dado define lo que éste "ve" de la ejecución. Observar que si s es un simplejo que representa el inicio de una ejecución, también lo es cualquier $s' \subseteq s$, ya que s' describe el inicio de una ejecución en donde participan menos procesos. Un *complejo simplicial* (o *complejo*) es aquel donde $\mathcal{K}$ es un conjunto de simplejos cerrados bajo inclusión: Si $s \in \mathcal{K}$ y $s' \subseteq s$, entonces $s' \in \mathcal{K}$. El conjunto de todos los posibles valores de entrada asignados a una tarea, forman un complejo. En computación distribuida, cada vértice v de un simplejo es etiquetado con un proceso distinto p_i. Decimos que un complejo formado por tales simplejos, es *coloreado* con esos nombres de proceso.

La *dimensión* de un simplejo s es $|s| - 1$, uno menos que el tamaño del conjunto. La *dimensión* de un complejo es la mayor dimensión de cualquiera de sus simplejos; y un complejo es *puro* si todos los simplejos maximales en ese complejo tienen la misma dimensión. Un simplejo de dimensión d es llamado *d-simplejo*, y lo mismo ocurre con los complejos. Un simplejo de dimensión 1 es llamado *arista*, de dimensión dos es llamado *triángulo*, y de dimensión tres es un *tetraedro*. Un complejo simplicial que solo contiene vértices y aristas, es llamado *gráfica*.

La subdivisión de un complejo A geométrico se logra al "dividir" los simplejos de A en simplejos más pequeños. Ver figura 2.

Un *mapeo simplicial* de un complejo $\mathcal{K}$ a uno $\mathcal{L}$ es una función de los vértices de $\mathcal{K}$ a los de $\mathcal{L}$ que preserva simplejos. Si además preserva nombres, se dice que el mapeo simplicial es cromático. Notemos que en el caso de que los complejos sean puros de dimensión 1, es decir, gráficas, el mapeo simplicial es un homomorfismo que permite mapear aristas en aristas o en vértices, y en caso de ser cromático, debe mapear aristas en aristas. La importancia de los mapeos simpliciales en topología y en la teoría de computación distribuida, radica en que son versiones discretas de funciones continuas. Por ejemplo, si $\mathcal{K}$ representa una gráfica conexa, entonces la imagen de $\mathcal{K}$ bajo un mapeo simplicial es conexa.

3. Tareas

Formalmente, una tarea $\mathcal{T}$ para $(n+1)$ procesos es definida por una tripleta, $\mathcal{T} = (\mathcal{I}, \mathcal{O}, \Delta)$, donde $\mathcal{I}$ es un complejo puro n-dimensional que define las asignaciones de valor de entrada posibles: Si $\{(p_{i_0}, x_{i_1}), \ldots, (p_{i_k}, x_{i_k})\}$ es un simplejo de $\mathcal{I}$, entonces es posible que la ejecución inicie con cada p_{i_j} con valor asignado de entrada x_{i_j}, para $0 \leq j \leq k$. De manera similar, $\mathcal{O}$ es un complejo puro n-dimensional que define las

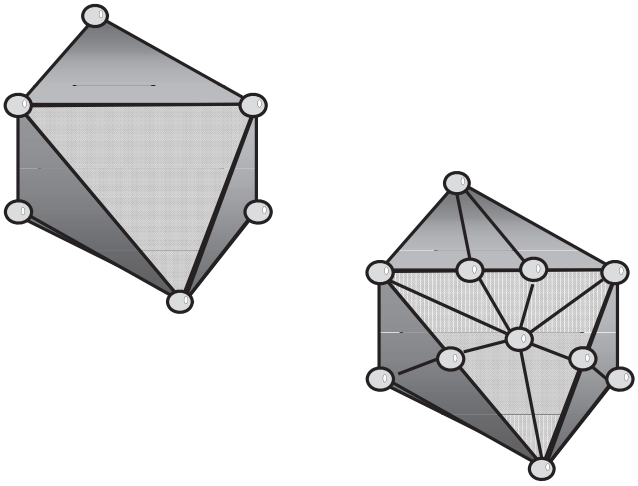

Figura 2: complejo (arriba) y complejo subdividido (abajo).

opciones posibles de los valores de salida: Si $\{(p_{i_1}, y_{i_1}), \ldots, (p_{i_k}, y_{i_k})\}$ es un simplejo de $\mathcal{O}$, entonces es posible que una ejecución termine con cada p_{i_j}, habiendo elegido el valor de salida y_{i_j}, para $0 \leq j \leq k$. Finalmente, Δ es un mapeo que asigna cada simplejo de entrada s en $\mathcal{I}$ un subcomplejo $\Delta(s) \subseteq \mathcal{O}$, con la siguiente interpretación: si el sistema inicia en el estado $s \in \mathcal{I}$, entonces, cada ejecución deberá terminar en algún estado $t \in \Delta(s)$. Por ejemplo, en la figura 1, se representa el simplejo de entrada $\{(p_1, x_1), (p_2, x_2), (p_3, x_3)\}$, y el simplejo de salida $\{(p_1, y_1), (p_2, y_2), (p_3, y_3)\}$.

El mapeo Δ debe satisfacer las siguientes propiedades formales. Cada proceso que termina en una ejecución debe haber iniciado; por lo que Δ debe *preservar el color*: para cada $t \in \Delta(s), \text{names}(t) \subseteq \text{names}(s)$. Considerar una ejecución en la que los procesos en $s \in \mathcal{I}$ participan, pero un subconjunto $s' \subset s$ termina antes de que el resto inicie. Los procesos que inician tarde deberán elegir valores compatibles con los valores ya elegidos por los procesos que empiezan antes. Esto se expresa en el siguiente requisito de *mapeo de transporte:*

$$\text{si } s' \subseteq s, \text{ entonces } \Delta(s') \subseteq \Delta(s).$$

Un protocolo *resuelve* la tarea $(\mathcal{I}, \mathcal{O}, \Delta)$ si para cada simplejo de entrada $s \in \mathcal{I}$, y cada ejecución del protocolo iniciando a partir de s,

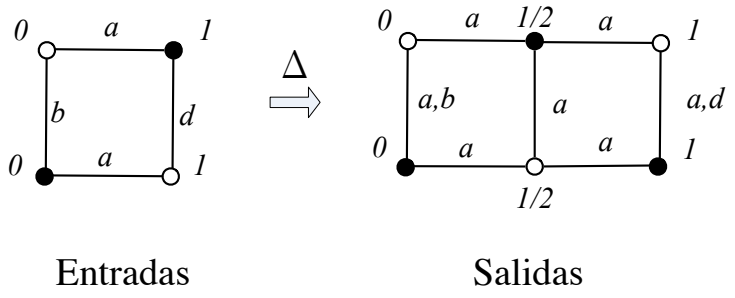

Figura 3: Tarea de acuerdo aproximado para dos procesos y $\epsilon = 1/2$.

cada proceso sin falla elige un valor de salida y el simplejo de salida es definido por esas elecciones.

3.1. Algunas tareas

La tarea de ϵ-*acuerdo aproximado* [**16**] ha jugado un papel central en la computación distribuida. Desde el lado práctico, permite resolver el problema de consenso con una aproximación arbitrariamente buena, mientras la solución perfecta no existe (ver sección 5.2). Desde el lado teórico, el acuerdo aproximado permite probar (uno de los dos sentidos) el teorema 5.1 de caracterización de la computabilidad distribuida.

En el caso binario, la tarea de acuerdo aproximado se define de la manera siguiente. Cada proceso inicia con un valor de $\{0, 1\}$. Los procesos deben decidir valores que se encuentren a distancia a lo más ϵ, $\epsilon > 0$, uno del otro y si todos empiezan con el mismo valor, deben decidir dicho valor. Consideremos el caso donde $\epsilon = \frac{1}{2^x}$, para un entero $x > 0$. La figura 3 (de [**18**]) ejemplifica la tarea de acuerdo aproximado binario [**16**] para dos procesos, donde $\epsilon = 1/2$. El conjunto de posibles valores de entrada es $\{0, 1\}$. Un vértice $(p_i, j), i, j \in \{0, 1\}$, es representado en la figura como un círculo, blanco para p_0 y negro para p_1, etiquetado con el valor j. El complejo de entrada está del lado izquierdo y el complejo de salida está a la derecha. Cada valor de entrada es 0 ó 1. Si ambos inician con el mismo valor, deberán decidir ese valor. En la figura, el requisito corresponde a un arista etiquetada con b ó d. Las dos aristas en las que inician los procesos con distintos valores, tienen la etiqueta a. En este caso, Δ permite que el proceso decida cuál utilizar $\left\{0, \frac{1}{2}, 1\right\}$, siempre y cuando las diferencias en los valores decididos sean a lo más 1/2. En particular Δ de un arista etiquetada a incluye cualquier salida etiquetada con a. Las ejecuciones en las que solo participa un proceso, son capturadas por la definición $\Delta(p_i, j) = (p_i, j)$,

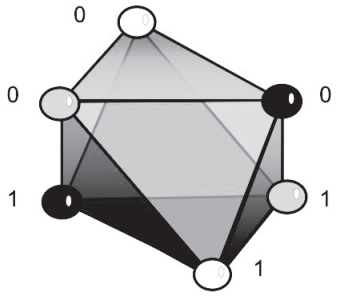

Figura 4: Complejo de entrada binaria para 3 procesos.

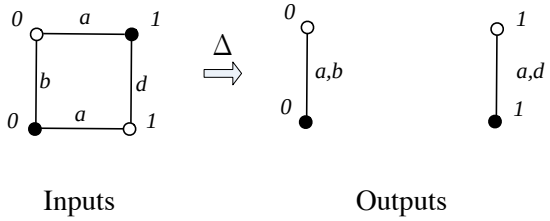

Figura 5: Tarea de consenso para dos procesos.

para $i, j \in \{0, 1\}$.

El complejo de entrada para tres procesos, aparece ilustrado en la figura 4. Cada simplejo de entrada para 3 procesos corresponde a un triángulo, en el que los procesos inician con 0 ó 1. La tarea del acuerdo aproximado tiene un protocolo sin espera, para cualquier $\epsilon > 0$, como se describe en la sección 4.3.

Aun cuando los procesos pueden acordar sobre valores arbitrariamente cercanos, con un protocolo sin espera no pueden llegar a alcanzar un acuerdo perfecto. Cuando ϵ es cero, el problema se llama *consenso* (figura 5) y en este modelo no existe un protocolo sin espera [**17**, **40**]. La tarea del consenso puede relajarse, para obtener la tarea k-*acuerdo*, en donde cada proceso elige un valor de entrada, pero se pueden elegir tantos valores distintos como k [**15**]. Esta tarea puede resolverse si y solo si el número de procesos que pueden fallar es menor a k [**10**, **31**, **48**]. En particular, el consenso, que es el 1-acuerdo tiene protocolo solo cuando ningún proceso puede fallar.

Las tareas que *preservan localidad* [**18**] tienen una naturaleza di-

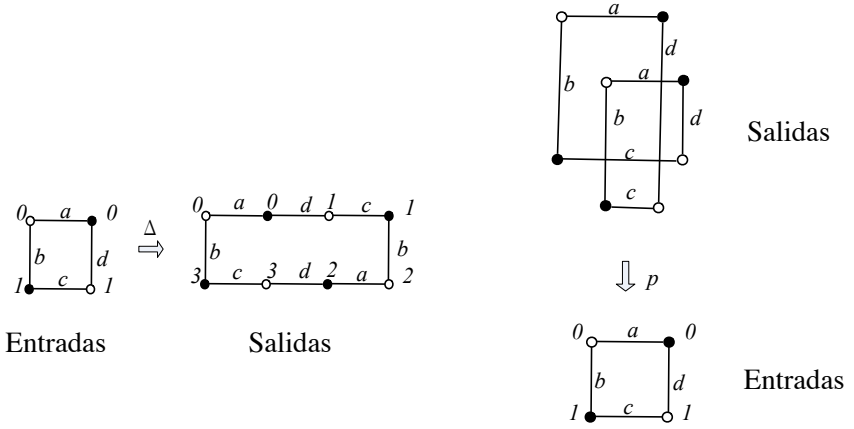

Figura 6: Una tarea de cubrimiento doble para dos procesos.

ferente. Se sabe que en nuestro modelo no tienen protocolos sin espera [**31**]. La figura 6 muestra un ejemplo con dos procesos. Los procesos inician con entradas binarias. Del lado izquierdo de la figura, la especificación de la tarea dice que si ambos inician con el mismo valor, deben decidir el mismo valor: Si empiezan con 0, es preciso decidir 0 ó 2; si inician con 1, deben decidir 1 ó 3. Si inician con diferentes valores, las salidas válidas están definidas en la figura, mediante etiquetas de arista b ó d. La relación Δ también define las salidas posibles cuando solo un proceso opera: por ejemplo, cuando el proceso blanco inicia con 0, puede decidir 0 ó 2; mientras que si el proceso inicia con 1, puede decidir 1 ó 3. Noten que $\mathcal{O}$, un ciclo de longitud 8, localmente se ve como $\mathcal{I}$, un ciclo de longitud 4, en el sentido de que la 1-vecindad de cada vértice v en $\mathcal{I}$ es igual a la 1-vecindad del vértice correspondiente en $\Delta(v)$. El lado derecho de la figura muestra como cubre $\mathcal{I}$, envolviéndolo 2 veces; donde p identifica las aristas con la misma etiqueta. De manera informal, $\mathcal{O}$ *cubre* $\mathcal{I}$; es decir que hay un mapeo p de $\mathcal{O}$ a $\mathcal{I}$, tal que por cada vértice v en $\mathcal{I}$, si consideramos los vértices w_i con $p(w_i) = v$, las vecindades alrededor de cada w_i se ven idénticas a la vecindad alrededor de w. En efecto, el análisis de la computabilidad distribuida de estas tareas esta íntimamente relacionado con un tema importante de topología: *espacios de cubrimientos.*

Otras tareas incluyen acuerdos de ciclo [**26**] (ver la siguiente descripción), consenso simultáneo [**2**], tareas que preservan localidad [**18**], renombrarado (renaming) [**3**] (ver [**13**] para una introducción), rompimiento de simetría débil [**31**, **21**]; tareas de rompimiento de simetría generalizada [**34**] (una familia de tareas que incluye renombrado y rompimiento de simetría débil) .

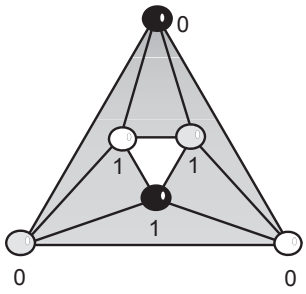

Figura 7: Complejo de salida para dos procesos de la tarea de rompimiento de simetría débil.

Algunas tareas inducen jerarquías, en donde tareas son computables únicamente cuando se les da acceso a un oráculo distribuido para otras tareas: tareas de preservación de localidad [**18**] y de acuerdo de ciclos [**26**, **39**].

3.2. Tareas coloreadas vs. tareas incoloras

Muchas de las tareas estudiadas anteriormente, tales como el consenso, el k-acuerdo, el acuerdo aproximado y el acuerdo de ciclo son *incoloras* en el sentido de que pueden ser definidas solo en términos de conjuntos de posibles valores de entrada y de salida, sin tener que especificar cuál proceso tiene que valor asignado como entrada o como salida. En una tarea incolora, un proceso puede adoptar el valor de entrada o de salida de otro proceso, sin violar la especificación de la tarea.

Un ejemplo de una tarea que *no* es incolora, es la famosa tarea de *renombrado* [**3**]. En esta tarea, los procesos inician con nombres únicos tomados de un espacio de nombres grande y deben decidir nombres únicos tomados de un espacio de nombre más pequeño. En este caso, un proceso no puede adoptar el nombre de salida de otro proceso, porque los nombres de salida tal vez no serían únicos. Se puede decir algo similar de una tarea de rompimiento de simetría débil, la cual requiere que si todos los procesos participan, al menos uno decida 0 y al menos uno decida 1.

La figura 7 muestra el complejo de salida para tres procesos de rompimiento de simetría (el triángulo blanco es prohibido, ya que ahí todos los procesos eligen el mismo valor de salida).

Una tarea incolora $(\tilde{\mathcal{I}}, \tilde{\mathcal{O}}, \tilde{\Delta})$ se define en términos de un complejo en el cual los vértices no están coloreados con nombres de procesos y cuyos simplejos pueden ser de cualquier dimensión, sin relación con el número de procesos. En realidad, una tarea incolora específica una

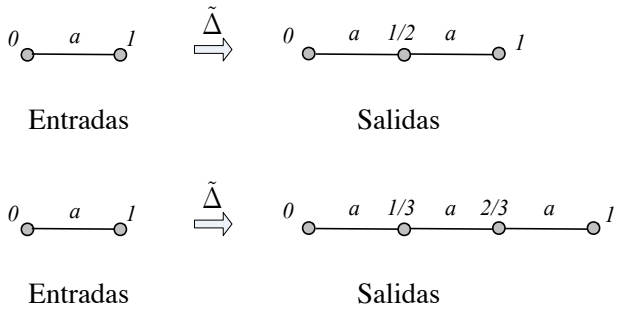

Figura 8: Dos tareas de ϵ-acuerdo incoloro para dos procesos y $\epsilon = 1/2, 1/3$.

familia de tareas, una para cada número de procesos $n > 1$.

Definición 1. *Una* tarea incolora $\tilde{T}$ *es una tripleta* $(\tilde{\mathcal{I}}, \tilde{\mathcal{O}}, \tilde{\Delta})$ *donde* $\tilde{\mathcal{I}}$ *y* $\tilde{\mathcal{O}}$ *son complejos y* $\tilde{\Delta}$ *es un mapeo de transporte de* $\tilde{\mathcal{I}}$ *a* $\tilde{\mathcal{O}}$.

Cada vértice de $\tilde{\mathcal{I}}$ y $\tilde{\mathcal{O}}$ es solo un valor (no incluye un nombre de proceso), el valor de la pareja se asignará igual que se asigna en el caso que la tarea sí tiene color. Debido a que $\tilde{\Delta}$ es el mapeo de transporte (como se define al inicio de la sección 3), $\tilde{\Delta}(\tilde{s'}) \subseteq \tilde{\Delta}(\tilde{s})$ por cada $\tilde{s'}, \tilde{s} \in \tilde{\mathcal{I}}$, de tal forma que $\tilde{s'} \subseteq \tilde{s}$.

Si $\tilde{s}$ es un simplejo de $\tilde{\mathcal{I}}$, entonces hay un estado inicial del sistema en donde s es el conjunto de valores de entrada. Simétricamente, un conjunto de valores de salida es un simplejo en $\tilde{\mathcal{O}}$ si hay un estado de sistema final en donde $\tilde{t}$ es el conjunto de los valores de salida. A nivel operativo, los procesos son asignados vértices de manera inicial (no necesariamente distintos) de un simplejo $\tilde{s}$ en $\tilde{\mathcal{I}}$, y se detienen después de elegir vértices (no necesariamente distintos) de un simplejo $\tilde{t}$ *en* $\tilde{\mathcal{O}}$, donde $\tilde{t} \in \tilde{\Delta}(\tilde{s})$.

Formalmente, por cada n, una tarea incolora $\tilde{T} = (\tilde{\mathcal{I}}, \tilde{\mathcal{O}}, \tilde{\Delta})$ induce una tarea $T = (\mathcal{I}, \mathcal{O}, \Delta)$ para n procesos, de la manera siguiente. Un simplejo de entrada s está en $\mathcal{I}$ si los nombres en s son distintos y el conjunto de valores de s forma un simplejo $\tilde{s} \in \tilde{\mathcal{I}}$. De manera similar, para un simplejo de salida t en $\mathcal{O}$, $\tilde{t} \in \tilde{O}$ con $\tilde{O} \in \Delta(\tilde{s})$. Decimos que $\tilde{s}$ se obtiene de s proyectando los nombres. Ahora, $t \in \Delta(s)$ si (s, t) es una pareja entrada-salida y hay proyecciones $\tilde{s}$, $\tilde{t}$, con $\tilde{t} \in \tilde{\Delta}(\tilde{s})$.

La tarea de acuerdo aproximado con $\epsilon = 1/2$ de la figura 3 está representada de manera más sucinta como una tarea incolora. En la figura 8, también se ejemplifica el caso $\epsilon = 1/3$. Notemos que se implican las tareas para un cierto número de procesos. Es decir, los procesos inician con un valor de entrada 0 ó 1. Si todos empiezan con el mismo

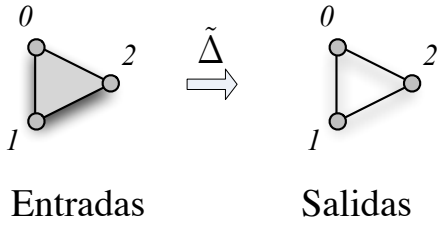

Figura 9: Tarea 2-Acuerdo.

valor, todos decidirán el mismo valor. De otra manera, todos decidirán valores a lo más ϵ de cada uno: valores que forman un simplejo de salida (un vértice o un arista en el complejo de salida).

Una tarea 2-acuerdo se representa de manera sucinta en como tarea incolora en la figura 9.

Cualquier número de procesos, inicia con valores del conjunto $\{0, 1, 2\}$ y se deciden los valores del mismo conjunto. Los valores decididos deben pertenecer a los valores propuestos en la ejecución. A lo más dos valores diferentes pueden decidirse. Por lo tanto, el complejo de entrada es inducido por un triángulo de incoloro sólido, mientras que el complejo de salida es inducido por un triángulo incoloro sin el interior. De manera formal $\tilde{\Delta}$ se define así:

- Si $\tilde{s} = \{v\}$ entonces $\tilde{\Delta}(\tilde{s}) = \{v\}$;
- Si $\tilde{s} = \{u, v\}$ entonces $\tilde{\Delta}(\tilde{s}) = \{\{u, v\}, \{u\}, \{v\}\}$;
- Si $\tilde{s} = \{u, v, w\}$ entonces $\tilde{\Delta}(\tilde{s}) = \{\{u, v\}, \{u, w\}, \{v, w\}, \{u\}, \{v\}, \{w\}\}$.

Intuitivamente, es más fácil resolver una tarea incolora, ya que el protocolo puede ser *anónimo*. Debido a que la tarea es definida por los posibles conjuntos de valores de entrada y salida sin especificar nombres de procesos, un proceso resolviendo una tarea incolora no tendría necesidad de utilizar su identidad.

3.3. Tareas incoloras como problemas de convergencia

Podemos pensar en una tarea incolora $(\tilde{\mathcal{I}}, \tilde{\mathcal{O}}, \tilde{\Delta})$ como la especificación de un problema de convergencia de robots o agentes. Los procesos juegan el papel de los robots, el complejo $\tilde{\mathcal{O}}$ juega el papel del espacio donde navegan los robots. Estos deben "reunirse" en puntos que se encuentren cerca uno de los otros, es decir, en vértices de un simplejo de

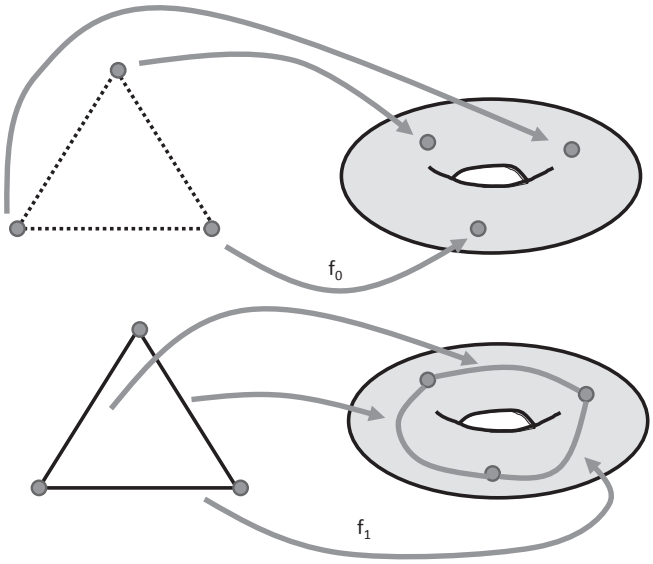

Figura 10: Una tarea de acuerdo de ciclo.

$\tilde{\mathcal{O}}$. Los simplejos tienen la función de representar puntos cercanos en el espacio $\tilde{\mathcal{O}}$ donde se mueven los robots.

Por ejemplo, consideremos la siguiente tarea de acuerdo de ciclo [**26**], que muestra la figura 10. Un *ciclo de aristas* $K = \vec{k}_0, \ldots \vec{k}_k$ es una secuencia de vértices, de forma tal que $\vec{k}_i$ y $\vec{k}_{i+1}$ forman una arista de $\mathcal{K}$, y todos los vértices son distintos, excepto $\vec{k}_0 = \vec{k}_k$, donde $\mathcal{K}$ es un complejo simplicial de 2-dimensión finito (sin color); K un ciclo simple de $\mathcal{K}$, y k_0, k_1, k_2 tres vértices distinguidos en $\mathcal{K}$. Para i, j, y k, distintos permitiremos que K_{ij} sea una sub-trayectoria de K uniendo a k_i con k_j, sin pasar por k_k. Cada uno de los $n+1$ procesos tiene un valor de entrada en $\{0, 1, 2\}$. Para cada simplejo de entrada s, se define $\Delta(s)$ de la siguiente manera:

$vals(s)$	$\Delta(s)$,
$\{i\}$	Todos deciden $\vec{k}_i$,
$\{i, j\}$	Vértices abarcan simplejos en K_{ij},
$\{0, 1, 2\}$	Vértices abarcan simplejos en $\mathcal{K}$.

En otras palabras, los procesos convergen en un simplejo de $\mathcal{K}$. Si todos los procesos tienen el mismo valor de entrada, convergen en el vértice correspondiente. Si solo tienen dos vértices de entrada distintos, convergen en algún simplejo que se encuentre en la trayectoria que

conecta los vértices correspondientes. Finalmente, si los procesos tienen tres vértices de entrada, todos convergen en cualquier simplejo de $\mathcal{K}$.

Por ejemplo, el 2-acuerdo es especificado por la tarea de ciclo cuando $\mathcal{K}$ consiste en los tres lados del triángulo con los vértices k_0, k_1, k_2, y está "hueco", es decir, sin simplejos de 2-dimensiones. En esta versión del acuerdo, los procesos deciden sobre dos de los valores k_0, k_1, k_2, como máximo.

4. El modelo iterado de lectura/escritura

Con frecuencia es conveniente estructurar los cómputos asíncronos en ejecuciones basadas en rondas [**14**, **30**, **43**]. En cada ronda, los procesos se comunican a través de una memoria compartida que solo a la cual pueden acceder en esa ronda. En el caso más sencillo, la memoria compartida para cada ronda consiste en un arreglo de registros, cada uno de los cuales puede ser escrito por un sólo proceso, y leído por todos. En cada ronda, cada proceso escribe en su registro y lee uno por uno todos los registros. La restricción de que cada memoria se puede acceder en una sola ronda induce modelos *iterados.*

Se puede obtener un modelo iterado más estructurado si la memoria compartida en cada ronda se puede leer mediante una sola operación, que devuelve una fotografía instantánea del contenido de la memoria. A esta operación se le llama *lectura instantánea.* En esta sección se define el modelo de cómputo distribuido con el que trabajaremos en el resto del artículo, y se denotará como modelo IWS (de su nombre en inglés, *iterated write-snapshot*).

4.1. La abstracción de lectura instantánea

Un objeto de lectura instantánea proporciona al programador una abstracción de memoria compartida de alto nivel, pero no ofrece ninguna capacidad adicional computacional [**1**].

Una lectura instantánea transforma un arreglo X de registros en la memoria que se pueden leer o escribir de manera individual, con una entrada por proceso, en un arreglo que proporciona una operación adicional: X.snapshot(). La operación X.snapshot() regresa el valor actual de todo el arreglo X. Las implementaciones más eficientes de lectura instantánea sin espera tienen una complejidad de tiempo $O(n \log n)$ [**5**]. Construcciones para el caso de lectura instantánea parcial se han propuesto en [**4**, **35**].

4.2. El modelo iterado con lectura instantánea (IWS)

Un objeto de lectura instantánea (que se puede accesar una sola vez) [**9**] es un arreglo $WS[0..n]$, inicializado en $[\perp, \ldots, \perp]$, al cual se puede tener acceso mediante una sola operación write_snapshot() que cada proceso invoca una sola vez. Intuitivamente, cuando un proceso p_i invoca write_snapshot(v), es como si de manera instantánea ejecuta una operación de escribir $WS[i] \leftarrow v$, seguida de una operación WS.snapshot(). Si se ejecutan varias operaciones WS.write_snapshot() de manera simultánea, sus escrituras correspondientes serán ejecutadas de manera concurrente y sus lecturas instantáneas correspondientes también serán ejecutadas de manera concurrente. Cada lectura instantánea concurrente ve los valores escritos de manera concurrente.

Una operación write_snapshot() invocada por p_i se comporta de la siguiente manera. Sea v_i es el valor escrito por p_i, y sm_i el valor (o vista -view) que p_i recibe de la operación. Una vista sm_i es un conjunto de parejas (k, v_k), donde v_k corresponde al valor de la entrada de p_k en el arreglo. Si $WS[k] = \perp$, la pareja $(k, \perp)$ no está en sm_i. Asimismo, suponemos que $sm_i = \perp$, si p_i nunca invoca WS.write_snapshot(). Cada invocación de WS.write_snapshot() hecha por un proceso que no falla, regresa con un valor.

Definición 2. *Un* objeto write_snapshot *es un objeto cuya única operación* write_snapshot() *es tal que si un proceso invoca* write_snapshot(v_i)*, obtiene un* sm_i *que satisface las siguientes propiedades:*

- Auto-inclusión. $\forall\, i : (i, v_i) \in sm_i$.
- Contención. $\forall\, i, j : sm_i \subseteq sm_j \ \vee \ sm_j \subseteq sm_i$.
- Inmediación. $\forall\, i, j : [(i, v_i) \in sm_j \ \wedge \ (j, v_j) \in sm_i] \ \Rightarrow \ (sm_i = sm_j)$.

La propiedad de auto-inclusión establece que un proceso ve lo que escribe, mientras que las propiedades de contención estipulan que las vistas obtenidas por los procesos pueden ser ordenadas. Finalmente, la propiedad de inmediación estipula que si dos procesos "se ven entre sí", obtienen la misma vista, cuyo tamaño de la cual corresponde al grado de *concurrencia* de las invocaciones correspondientes write_snapshot() y extiende los snapshots a *snapshots inmediatos*, también llamados *ejecuciones en bloque* [**10**, **48**]. (Las implementaciones están descritas en [**9**, **13**, **47**].)

El modelo que se denota IWS, está formado de un número sin límite de objetos write_snapshot. Estos objetos, denotados como $WS[1]$, $WS[2]$, ..., son accedidos de manera secuencial y asincronía por cada proceso, de acuerdo con el patrón basado en rondas de la figura 11.

```
v_i ← input; r_i ← 0;
loop forever r_i ← r_i + 1;
             sm_i ← WS[r_i].write_snapshot(v_i);
             v_i ← sm_i;
end loop.
```

Figura 11: Protocolo genérico para el modelo iterado

Al inicio, p_i almacena su entrada en su variable local v_i. En cada ronda r_i, p_i escribe su valor actual v_i en el objeto write_snapshot r_i, almacenando el resultado en sm_i. Éste es el valor v_i (llamado su *vista*) que escribirá la siguiente iteración. Por lo tanto, el proceso escribe todo lo "que conoce" en cada iteración. Esto es debido a que estamos interesados en resultados de computabilidad. Si la eficiencia es una preocupación, p_i podría escribir en la siguiente ronda un valor computado a partir de sm_i.

Resolviendo tareas. Para resolver una tarea $T = (\mathcal{I}, \mathcal{O}, \Delta)$ en el modelo IWS, dividimos el protocolo en dos partes. En la primera parte, cada proceso escribe repetitivamente su vista en una memoria compartida y después construye una nueva vista al leer la memoria, como ocurre con el protocolo genérico. Esta parte es genérica, en el sentido de que podría ser parte de cualquier protocolo para cualquier tarea. En la segunda parte, cada proceso decide cuántas iteraciones ejecutar y después aplica un mapeo de decisión específico a cada tarea, a su vista final, para determinar su valor de decisión (irrevocable). El número de iteraciones y el mapeo de decisión dependen de la tarea que se esté resolviendo. Para una tarea $\mathcal{T}$, las vistas iniciales de los procesos forman un simplejo de entrada $s \in \mathcal{I}$, y los valores de decisión forman un simplejo de salida t, tal que $t \in \Delta(s)$.

Se ha demostrado que el modelo IWS es equivalente al modelo usual de lectura/escritura, en el cual no se restringe a los protocolos a acceder sólo una vez a cada parte de la memoria compartida [**11**, **19**]. El atractivo del modelo IWS surge de que sus ejecuciones tienen una estructura recursiva elegante: la estructura global después de $r+1$ rondas se obtiene de la estructura del global después de r rondas. Esto facilita

```
operation approx_Agreement(x, v):
(01) pairs_i ← SM.write_snapshot(x, v) ;
(02) if (x = 0)
(03)       then out_i ← v
(04)       else  out_i ← approx_Agreement(x − 1, mid(pairs_i))
(05) end if;
(06) return(out_i).
```

Figura 12: Protocolo recursivo de acuerdo aproximado (código para p_i)

el análisis de cómputos asíncronos sin espera para probar resultados de imposibilidad [**28**, **27**]. La naturaleza recursiva del modelo también facilita el diseño y el análisis de protocolos [**13**, **20**].

4.3. Programación en el modelo iterado y recursión distribuida

Presentamos en esta sección un ejemplo de cómo se resuelve un tarea en el modelo IWS. Elegimos la tarea de acuerdo aproximado, que ya hemos utilizado a lo largo del trabajo.

La figura 12 muestra un protocolo que resuelve la tarea de acuerdo aproximado. Este protocolo es recursivo, en el modelo IWS. La idea del protocolo es sencilla. En cada iteración ocurre lo siguiente: Los procesos proponen a lo más dos valores distintos y deciden valores, de tal forma que la diferencia entre ambos se divide entre dos. Por lo tanto, si los procesos inician con valores de entrada en $\{0, 1\}$, después de x iteraciones, $x \geq 0$, los procesos decidirán valores que estén a lo más separados uno del otro en $1/2^x$.

Los procesos se comunican sus valores v, mediante una operación de lectura instantánea. El proceso p_i almacena el resultado de una lectura instantánea en su variable local $pairs_i$. Consideremos la iteración x. En donde si $x = 0$ no se hace nada y la salida de cada proceso es igual a su entrada. Supongamos $x \geq 1$. El proceso p_i ve dos valores de procesos $pairs_i$; los llamaremos p_1, p_2 (no son necesariamente diferentes). Después, p_i invoca recursivamente al protocolo, con $x - 1$, y el valor v igual a $(v_1 + v_2)/2$. Solo existen dos posibilidades: o todos ven ambos valores, o algunos ven un valor, digamos v_1, y los otros ven ambos. Al menos el último proceso en realizar la lectura instantánea ve ambos. Es imposible que algunos procesos vean solo un valor y otros procesos vean solo el otro valor, debido a la operación snapshot. Por lo tanto, en la invocación recursiva se tendrán a lo más dos valores diferentes y estarán separados entre sí por la mitad de los valores originales.

La complejidad del protocolo en pasos es x veces la complejidad de la operación de lectura instantánea.

5. Computabilidad distribuida

Hemos definido los problemas que nos interesa resolver, tareas, y hemos definido el modelo de cómputo, IWS, en el cual nos interesa resolver los problemas. Ahora nos interesa saber cuáles tareas tienen solución y cuáles no, en el modelo IWS.

5.1. El complejo del protocolo y su estructura recursiva

Consideremos el protocolo genérico de la figura 11, y las posibles vistas v_i, de cada iteración, r, $r \geq 0$. La vista v_i de p_i en r es denotada $view_i^r$. El caracterizar las posibles vistas da como resultado una caracterización de las tareas que pueden ser computadas en el modelo IWS.

Supongamos que se utiliza el protocolo genérico para resolver una tarea T definida por el triplete $(\mathcal{I}, \mathcal{O}, \Delta)$. Recordemos que cada simplejo de entrada I en $\mathcal{I}$, es un conjunto de pares

$$I = \{(p_i, x_i)\}$$

para algunos subconjuntos de los procesos, $\{p_i\}$, cada uno con un valor de entrada x_i. Por lo tanto, cada simplejo de entrada corresponde a un conjunto posible de vistas iniciales, en donde $view_i^0 = x_i$. El conjunto de todas las vistas iniciales posibles es por lo tanto igual al complejo de entrada $\mathcal{I}$.

En general, el conjunto de vistas en la ronda r, siempre es un complejo, $\mathcal{K}^r$, llamado el *complejo del protocolo*, porque si s es un conjunto de vistas de algunos procesos en ronda r, entonces cualquier subconjunto de s también es un conjunto de vistas en r. De manera más precisa, un simplejo $s \in \mathcal{K}^r$ consiste en un conjunto de parejas $\{(p_i, v_i)\}$, que corresponden a un subconjunto de los procesos, así como una ejecución del protocolo genérico, en donde cada v_i es una vista que p_i puede obtener en la iteración r de la ejecución.

¿Cuál es el complejo de vistas en la ronda $r = 1$? Está definido por las propiedades de la definición 2: Auto-inclusión, Contención, e Inmediación. Todas las vistas posibles en la primera ronda forman una subdivisión del simplejo de entrada. En la figura 13 se muestra el caso en el que $\mathcal{I}$ es el complejo de entrada binaria del consenso o del acuerdo

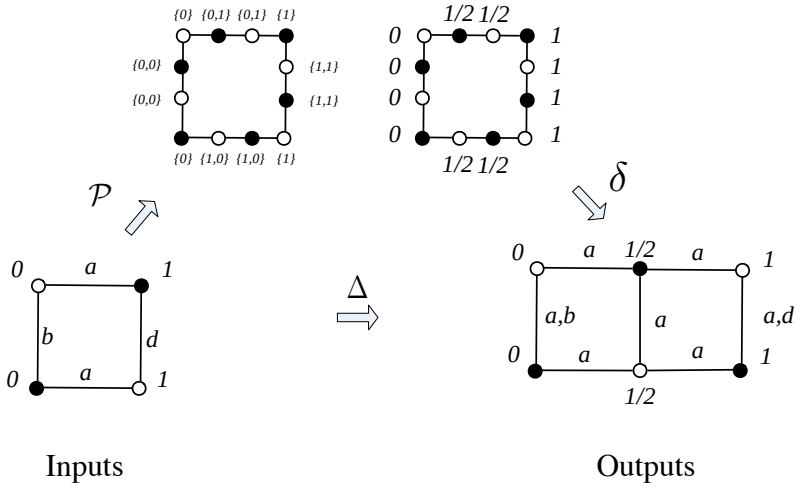

Figura 13: Tarea de acuerdo aproximado para dos procesos. El complejo de protocolo después de una ronda se encuentra en la parte superior (vistas del lado izquierdo, decisiones del lado derecho).

aproximado. Cada vértice de un proceso p_i está etiquetado con su vista v_i. Es preciso notar que los 4 vértices de las esquinas del cuadrado corresponden a las ejecuciones en las que un proceso solo se ve a sí mismo. Los otros vértices corresponden a las vistas en las que un proceso ve tanto su propio valor, como el valor de los otros procesos.

El protocolo corresponde a un mapeo de transporte $\mathcal{P}$ que manda cada simplejo del complejo de entrada a un subcomplejo del complejo del protocolo. Cada vértice de $\mathcal{I}$ es enviado al vértice que representa la ejecución en la que participa únicamente ese proceso. En general, $\mathcal{P}$ manda cada simplejo de $\mathcal{I}$ al subcomplejo de todas las posibles ejecuciones, iniciando con esas entradas, en la que sólo los correspondientes procesos participan.

Estructura Recursiva. El modelo iterado tiene la ventaja de que cada ronda es independiente de rondas anteriores. Las salidas de la ronda r son las entradas para la ronda $r+1$. Esta es la razón por la que es posible razonar de manera recursiva en este modelo, y el complejo en la ronda $r+1$ se obtiene al reemplazar cada simplejo por el mismo complejo.

En el caso de dos procesos, consideremos la figura 13. Cada arista en el complejo de entrada (complejo en la ronda 0) es reemplazada por una trayectoria de tres aristas en el complejo en la ronda 1. Notemos que siempre que haya un vértice contenido en dos aristas, en la ronda 1, se

presenta la situación en la que el proceso que corresponde a ese vértice no distingue entre dos ejecuciones, las cuales están representadas por los dos aristas.

En general, el complejo en la ronda $r + 1$ se obtiene al reemplazar cada arista del complejo en la ronda r por una trayectoria de tres aristas en el complejo en la ronda r. Pensemos en el protocolo como si "estiráramos" cada arista de entrada hacia una trayectoria, al insertar nuevos vértices entre los puntos finales. El caso de los tres procesos, representada en la figura 14, se explica a continuación.

Pensemos en el mapeo de decisión transportando esta trayectoria estirada a la trayectoria correspondiente en la gráfica de salida, de manera que respete Δ. Esta idea se utiliza para probar el siguiente resultado, acerca de la preservación de conexidad de ronda a ronda. Nos referimos a conexidad en el sentido de gráficas. Es decir, decimos que un complejo es *conexo* si la gráfica formada por sus vértices y sus aristas es conexa.

Lema 5.1. *Sea r cualquier ronda. Si el complejo de entrada $\mathcal{I}$ $(= \mathcal{K}^0)$ para el protocolo iterado es conexo, entonces el complejo de protocolo, $\mathcal{K}^r$, formado por las vistas en la ronda r, también es conexo.*

Este lema es un caso especial de propiedades más generales que el complejo de protocolo preserva, de ronda en ronda. La más sencilla de probar es la de ser una variedad combinatoria [**21**]. Decimos que un complejo puro n-dimensional es una *variedad* si cada uno de sus simplejos $(n-1)$-dimensionales está contenido en uno o dos simplejos n-dimensionales. De hecho se cumple la propiedad más general, de preservar la conectividad topológica [**28**], ya que el complejo de protocolo en una ronda es una subdivisión del complejo de la ronda anterior.

El caso de los tres procesos está representado en la figura 14 (de [**46**]). El 2-simplejo resaltado del lado izquierdo representa una ejecución donde p_1 y p_3 accesan al objeto de forma concurrente, cada uno ve al otro, pero no así p_2, el cuál accesa al objeto más tarde y observa los 3 valores. Sin embargo p_2 no puede saber el orden en el que p_1 y p_3 accesan al objeto. La ejecución en la que p_1 solo se vio a sí mismo y la ejecución en la que p_3 solo se vio a sí mismo, son indistinguibles para p_2. Estas dos ejecuciones están representadas por los 2-simplejos en las esquinas inferiores de las imágenes de la izquierda. Por lo tanto, los vértices en las esquinas del complejo representan las ejecuciones donde solo un proceso p_i tiene acceso al objeto, y los aristas que conectan dos vértices en la frontera de un simplejo de entrada, representan ejecuciones donde solo dos procesos tienen acceso al objeto. El triángulo en el centro del complejo representa la ejecución donde todos los tres

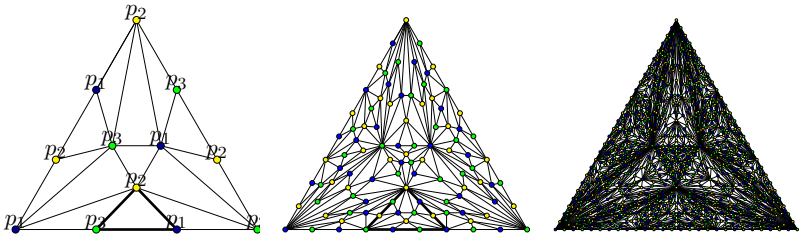

Figura 14: Complejo del protocolo del modelo IWS para tres procesos, para $1, 2$ y 3 rondas.

procesos tienen acceso al objeto de manera concurrente y reciben de regreso la misma vista.

Por lo tanto, el estado de una ejecución después de la primera ronda (con la que está asociado el objeto $WS[1]$), está representado por uno de los triángulos internos de la imagen del lado izquierdo. Entonces, el estado de la ejecución después de la segunda ronda (con la que el objeto $WS[2]$ está asociado), es representado por uno de los pequeños triángulos dentro del triángulo del centro, etc. De manera más general, como se muestra en la figura 14, podemos ver que en el modelo IWS en cada ronda, se construye un nuevo complejo recursivamente, al substituir cada simplejo por un complejo de una ronda. El resultado se resume en el siguiente lema.

Lema 5.2. *Sea $\mathcal{I}$ el complejo de entrada para un protocolo iterado de r rondas. El complejo de protocolo, $\mathcal{K}^r$ es una subdivisión coloreada de $\mathcal{I}$ $(= \mathcal{K}^0)$, y $\mathcal{P}$ es un mapeo de transporte de $\mathcal{I}$ a $\mathcal{K}^r$.*

5.2. Imposibilidad del consenso

Antes de describir el teorema general, que caracteriza las tareas computables en el modelo IWS, veamos el caso de la tarea del consenso, que ha jugado un papel muy importante en la computación distribuida.

Como corolario del lema 5.1, sabemos que la tarea de consenso no se puede resolver en nuestro modelo de cómputo distribuido. El siguiente lema implica que este resultado de imposibilidad se mantiene en el modelo no-iterado [**25**], usando la simulación de [**19**] que transforma un protocolo del modelo no-iterado al modelo IWS. Así mismo, el resultado se mantiene en el caso en que a lo más un proceso puede fallar, usando la simulación de [**12**].

Lema 5.3. *No existe un protocolo que resuelva la tarea de consenso en el modelo IWS.*

Consideremos el caso de dos procesos, para ser concretos, aunque el argumento general es el mismo. Supongamos para propósitos de contradicción que el consenso puede ser resuelto, vamos a decir después de r rondas. Consideremos el complejo de vistas $\mathcal{K}^r$. Es una subdivisión del simplejo de entrada y por lo tanto consiste en una gráfica de cuatro caminos unidos en las esquinas de un cuadrado. Esta gráfica aparece en la parte superior izquierda de la figura 13, para $r = 1$. Estas cuatro esquinas corresponden a las ejecuciones en donde un proceso solo se ve a sí mismo, en cada iteración. En este complejo, cada proceso decide 0 ó 1. La decisión colorea cada vértice con un valor binario. La gráfica con valores binarios se encuentra en la parte superior derecha de la figura 13, para un acuerdo aproximado; en este caso las decisiones de consenso 1/2 son reemplazadas ya sea por 0 ó por 1.

Finalmente, utilicemos el lema de Sperner (para una discusión sencilla y referencias ver [**45**]) para completar la prueba. Consideremos el camino en la parte superior del cuadrado $\mathcal{K}^r$; el que conecta el vértice de la esquina donde inicia el proceso blanco con 0 y el vértice donde el proceso negro inicia con 1. La decisión del proceso blanco en la esquina es 0, mientras que la decisión del proceso negro en la esquina es 1. Debe haber una arista en el camino, cuyos vértices estén coloreados con diferentes valores binarios; debido a que el camino es conexo y sus vértices finales tienen colores distintos. Esta arista corresponde a una ejecución en donde se deciden dos valores distintos, contradiciendo el requerimiento del acuerdo de la tarea de consenso.

La prueba de imposibilidad del consenso en el caso de tres procesos, de igual forma que para dos procesos, se obtiene aplicando el lema de Sperner al lema 5.2. En este caso, el lema de Sperner nos dice que si coloreamos las tres esquinas de un triángulo subdividido con tres colores diferentes, y los vértices en caminos sobre la frontera de la subdivisión, con colores de los vértices de sus esquinas correspondientes, entonces debe haber un triángulo coloreado con tres colores diferentes. Usando esta idea en la figura 14, encontramos un triángulo que corresponde a una ejecución donde los tres procesos deciden tres valores diferentes. Esto demuestra una imposibilidad más fuerte que la de consenso: tres procesos no pueden resolver tampoco la tarea de 2-acuerdo, para dos valores.

5.3. Caracterización de computabilidad

El resultado principal de la teoría de computabilidad distribuida es el teorema que nos dice como saber si una tarea $\mathcal{T} = (\mathcal{I}, \mathcal{O}, \Delta)$ se puede

resolver o no en el modelo IWS. Como lo hemos dicho ya, este teorema es la base de resultados en muchos otros modelos.

La caracterización de computabilidad parte de la siguiente observación. El complejo del protocolo en el modelo IWS está relacionado con el complejo de salida, por el mapeo de decisión $\delta(\cdot)$, un mapeo simplicial que envía cada vértice v del complejo del protocolo a un vértice w en el complejo de salida. El vértice w está etiquetado con el mismo nombre de proceso de v, y con el valor de decisión de éste. Se debe cumplir que $\delta(s)$ es un simplejo de $\mathcal{O}$. Ver la figura 13 para un ejemplo concreto. El valor asociado a w es el valor que el proceso $id(v)$ decide cuando su vista final en la ejecución es $view(v)$. El mapeo δ es *simplicial* porque preserva los simplejos: Si los vértices v_i forman un simplejo en el complejo del protocolo, entonces los vértices $\delta(v_i)$ forman un simplejo en el complejo de salida.

Recordemos el lema 5.2. Un protocolo de r rondas induce una subdivisión coloreada $\mathcal{K}^r$ del complejo de entrada $\mathcal{I}$, con un mapeo de transporte $\mathcal{P}$ que manda cada simplejo de $\mathcal{I}$ a un subcomplejo de $\mathcal{K}^r$. Si el protocolo resuelve la tarea $\mathcal{T} = (\mathcal{I}, \mathcal{O}, \Delta)$, entonces su mapeo de decisión δ, debe *respetar* a Δ, es decir, para cada simplejo $s \in \mathcal{I}$, y cada simplejo $t \in \mathcal{P}(s)$, el simplejo $\delta(t)$ debe estar en $\Delta(s)$. En otras palabras, el protocolo resuelve $\mathcal{T}$ si la composición del mapeo de decisión δ con el mapeo de transporte $\mathcal{P}$ es un mapeo de transporte F, tal que para cada $s \in \mathcal{I}$, $F(s) \subseteq \Delta(s)$. Tenemos el siguiente resultado, ilustrado en la figura 13.

Lema 5.4. *Si la tarea $\mathcal{T} = (\mathcal{I}, \mathcal{O}, \Delta)$ tiene solución, entonces existe una subdivisión coloreada $\mathcal{K}^r$ de $\mathcal{I}$, y un mapeo simplicial δ de $\mathcal{K}^r$ a $\mathcal{O}$ que respeta a Δ.*

Este lema nos da una condición suficiente para la computabilidad de una tarea en el modelo IWS. La caracterización de computabilidad distribuida nos dice que la condición es también necesaria. Para probar este resultado hay que poder obtener un protocolo a partir de *cualquier* subdivisión coloreada de $\mathcal{I}$ y su correspondiente mapeo δ, lo cual no es fácil [**31**].

Teorema 5.1. *Una tarea $\mathcal{T} = (\mathcal{I}, \mathcal{O}, \Delta)$ tiene solución en el modelo IWS si y sólo si existe una subdivisión coloreada $\mathcal{K}$ de $\mathcal{I}$ y un mapeo simplicial cromático δ de $\mathcal{K}$ a $\mathcal{O}$ que respete a Δ.*

6. Conclusión

Al igual que cualquier teoría de computabilidad, una teoría de computabilidad distribuida debe definir sus problemas de interés y un modelo de cómputo para resolverlos. Revisamos el caso en el cual los problemas son *tareas* y el modelo es el IWS. Presentamos una serie de tareas que se han estudiado en el pasado; algunas de ellas se pueden resolver en el modelo IWS y otras no. Describimos el modelo IWS, y explicamos por qué representa una abstracción de un sistema distribuido, con una elegante estructura matemática, cuyo carácter recursivo facilita su análisis. Presentamos, como ejemplo, un protocolo en el modelos IWS para resolver una tarea de acuerdo aproximado, la cual juega un papel importante en la teoría. Finalmente, presentamos el teorema que caracteriza a la tareas que tienen solución en el modelo IWS. Una aplicación concreta, es la famosa prueba de imposibilidad de la tarea de consenso, o más en general, la imposibilidad de resolver la tarea de $(n-1)$-acuerdo, es decir, que n procesos queden de acuerdo ni siquiera en $n-1$ de sus valores propuestos.

El estilo de presentación ha sido informal, e intentando elegir algunas de las ideas más importantes, desde nuestro punto de vista. Sin entrar en detalles, mencionamos que este modelo es equivalente, o forma la base para el análisis de modelos más cercanos a los sistemas reales. Encontrar simulaciones entre modelos de cómputo distribuido, y generalizaciones de los resultados del modelo IWS, es un área de investigación activa. Mencionamos brevemente que, al igual que en la teoría de computación secuencial, las reducciones juegan un papel importante, para organizar a los problemas en jerarquías de acuerdo a su dificultad. En el caso del cómputo distribuido, las reducciones y simulaciones son en sí distribuidas, lo cual ha llevado a la publicación de diversos trabajos de investigación.

Esperamos haber logrado mostrar que las relaciones entre la computabilidad distribuida y la topología son sorprendentemente íntimas, a pesar de que no ha sido posible presentar acá todas estas relaciones. Hemos visto que en el modelo IWS, donde cualquier número de procesos pueden fallar, el complejo del protocolo es una subdivisión del complejo de entrada; se requiere de más herramientas, tanto de topología como de computación, describir resultados como el siguiente. Cuando consideramos un sistema en el que menos procesos pueden fallar, el complejo de protocolo tendrá más "hoyos", es decir, homotopía no trivial, y es esto precisamente lo que permite a un sistema más confiable resolver más tareas. En trabajos anteriores se han utilizado con éxito varias ideas y

herramientas centrales de esta importante rama de la matemática moderna, como homología, homotopía, espacios de cubrimiento, teoremas de "shellability", nervios, y aproximación simplicial. Elegimos nuestra presentación de manera que pudieramos exponer ideas importantes requiriendo el mínimo de prerrequisitos del lector tanto de topología como de computación.

Existen otros problemas y modelos de interés en sistemas distribuidos, que representan interesantes líneas de investigación para explorar en el futuro. En la tareas que estudiamos, cada proceso comienza con una sola entrada, y debe decidir una sola salida. En alguna situaciones, cada proceso recibe continuamente entradas, una tras de otra, y debe ir produciendo salidas conforme le van llegando las entradas. En cuanto a modelos de cómputo, hemos considerado solamente las fallas más benignas, y no consideramos el caso en el cual los procesos pueden comportarse arbitrariamente, desplegando incluso comportamientos con la intención explícita de perjudicar al sistema. En cuanto a protocolos, es de interés considerar el poder que da la aleatoriedad, donde se permiten protocolos que fallan con baja probabilidad o que no siempre terminan. Esperamos que las ideas presentadas acá sean de utilidad para estudiar estos problemas.

Bibliografía

1. Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, y N. Shavit, Atomic snapshots of shared memory, *Journal of the ACM* **40(4)** (1993) 873–890.
2. Y. Afek, E. Gafni, S.Rajsbaum, M. Raynal, y C. Travers, The k-simultaneous consensus problem, *Distributed Computing* **22(3)** (2010) 185–195.
3. H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, y R. Reischuk, Renaming in an asynchronous environment, *Journal of the ACM* **37(3)** (1990) 524–548.
4. H. Attiya, R. Guerraoui, y E. Ruppert, Partial snapshot objects, en *Proc. 20th ACM Symposium on Parallel Architectures and Algorithms (SPAA'08)*, ACM Press, 2008, 336–343.
5. H. Attiya y O. Rachman, Atomic snapshots in $o(n \log n)$ operations, *SIAM Journal of Computing* **27(2)** (1998) 319–340.
6. H. Attiya y S. Rajsbaum, The combinatorial structure of wait-free solvable tasks, *SIAM Journal of Computing* **31(4)** (2002) 1286–1313.
7. H. Attiya y J. Welch, *Distributed computing: fundamentls, simulations and advanced topics*, 2 ed., John Wiley and Sons, 2004.
8. O. Biran, S. Moran, y S. Zaks, A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty proces-

sor, en *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing (PODC'88)*, 1988, 263–275.
9. E. Borowsky y E.Gafni, Immediate atomic snapshots and fast renaming, en *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, ACM Press, 1993, 41–51.
10. E. Borowsky y E. Gafni, Generalized flp impossibility result for t-resilient asynchronous computations, en *Proc. 25-th Annual ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, 1993, 91–100.
11. ———, A simple algorithmically reasoned characterization of wait-free computations, en *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, ACM Press, 1997, 189–198.
12. E. Borowsky, E. Gafni, N. Lynch, y S. Rajsbaum, The bg distributed simulation algorithm, *Distributed Computing* **14(3)** (2001) 127–146.
13. A. Castañeda, S. Rajsbaum, y M. Raynal, The renaming problem in shared memory systems: an introduction, *Computer Science Review* **5(3)** (2011) 229–251.
14. B. Charron-Bost y A. Schiper, The heard-of model: Computing in distributed systems with benign faults, *Distributed Computing* **22(1)** (2009) 49–71.
15. S. Chaudhuri, More choices allow more faults: Set consensus problems in totally asynchronous systems, *Information and Computation* **105(1)** (1993) 132–158.
16. D. Dolev, N. Lynch, S. Pinter, E. Stark, y W. Weihl, Reaching approximate agreement in the presence of faults, *Journal of the ACM* **33(3)** (1986) 499–516.
17. M. J. Fischer, N. A. Lynch, y M. S. Paterson, Impossibility of distributed consensus with one faulty process, *Journal of the ACM* **32(2)** (1985) 374–382.
18. P. Fraigniaud, S. Rajsbaum, y C. Travers, Locality and checkability in wait-free computing, en *Proc. 25th Int'l Symposium on Distributed Computing (DISC'11)*, LNCS 6950, Springer, 2011, 333–347.
19. E. Gafni y S. Rajsbaum, Distributed programming with tasks, en *Proc. 14th Int'l Conference on Principles of Distributed Systems (OPODIS'10)*, LNCS 6490, Springer, 2010, 205–218.
20. ———, Recursion in distributed computing, en *Proc. 12th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'10)*, LNCS 6366, 2010, 362–376.
21. E. Gafni, S. Rajsbaum, y M. Herlihy, Subconsensus tasks: Renaming is weaker than set agreement, en *Proc. 20th Int'l Symposium on Distributed Computing (DISC'06)*, LNCS 4167, Springer, 2006, 329–338.
22. E. Gafni y E. Koutsoupias, Three-processor tasks are undecidable, *SIAM J. Comput.* **28(3)** (1999) 970–983.
23. R. Guerraoui y M. Raynal, From unreliable objects to reliable objects:

the case of atomic registers and consensus, en *9th Int'l Conference on Parallel Computing Technologies (PaCT'07)*, LNCS 4671, Springer, 2007, 47–61.

24. T. Harris, A survey of PRAM simulation techniques, *ACM Computing Surveys, ACM New York, NY, USA.* **26(2)** (June 1994) 187–206.
25. M. P. Herlihy, Wait-free synchronization, *ACM Transactions on Programming Languages and Systems* **13(1)** (1991) 124–149.
26. M. P. Herlihy y S. Rajsbaum, A classification of wait-free loop agreement tasks, *Theoretical Computer Science* **291(1)** (2003) 55–77.
27. _____, Concurrent computing and shellable complexes, en *Proc. 24th Int'l Symposium on Distributed Computing (DISC'10)*, LNCS 6343, Springer, 2010, 109–123.
28. _____, The topology of shared memory adversaries, en *Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC'10)*, ACM Press, 2010, 105–113.
29. _____, Simulations and reductions for colorless tasks, en *Proc. 31st ACM Symposium on Principles of Distributed Computing (PODC'12)*, ACM Press, 2012, 253–260.
30. M. P. Herlihy, S. Rajsbaum, y M. Tuttle, Unifying synchronous and asynchronous message-passing models, en *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC'98)*, ACM Press, 1998, 133–142.
31. M. P. Herlihy y N. Shavit, The topological structure of asynchronous computability, versión de congreso en acm stoc'1993, *Journal of the ACM* **46(6)** (1999) 858–923.
32. _____, *The Art of Multiprocessor Programming*, Morgan Kaufman Pub., San Francisco (CA), 508 páginas, 2008.
33. M. Herlihy y S. Rajsbaum, The decidability of distributed decision tasks (extended abstract), en *Proc. 29th annual ACM Symposium on Theory of Computing (STOC'97)*, ACM Press, 1997, 589–598.
34. D. Imbs, S. Rajsbaum, y M. Raynal, The universe of symmetry breaking tasks, en *Proc. 18th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'11)*, NCS 6796, Springer, 2011, 66–77.
35. D. Imbs y M. Raynal, Help when needed, but no more: Efficient read/write partial snapshot, *Journal of Parallel and Distributed Computing* **72(1)** (2012) 1–12.
36. D. Kozlov, *Combinatorial Algebraic Topology*, Algorithms and Computation in Mathematics, Vol. 21, Springer-Verlag, 407 páginas, 2007.
37. L. Lamport, On interprocess communication, part 1: Basic formalism, part ii: Algorithms, *Distributed Computing* **1(2)** (1986) 77–101.
38. L. Lamport, R. Shostak, y M. Pease, The byzantine generals problem, *ACM Transactions Programming Languages Systems* **4(3)** (1982) 382–

401.
39. X. Liu, Z. Xu, y J. Pan, Classifying rendezvous tasks of arbitrary dimension, *Theoretical Computer Science* **410(21-23)** (2009) 2162–2173.
40. M. C. Loui y H. H. Abu-Amara, Memory requirements for agreement among unreliable asynchronous processes, *Parallel and Distributed Computing: vol. 4 of Advances in Computing Research* **4** (1987) 163–183.
41. N. A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Pub., San Francisco (CA), 872 páginas, 1996.
42. Y. Matiyasevich, *Hilbert's 10th Problem*, MIT Press, 1993.
43. Y. Moses y S. Rajsbaum, A layered analysis of consensus, *SIAM Journal Computing* **31(4)** (2002) 989–1021.
44. S. Rajsbaum, Iterated shared memory models, en *Proc. 9th Latin American Symposium Theoretical Informatics (LATIN'10)*, LNCS 6034, Springer, 2010, 407–416.
45. ———, Dos perspectivas del lema de sperner, *Carta Informativa, Sociedad Matemática Mexicana* (junio 1998) 1–3.
46. S. Rajsbaum, M. Raynal, y C. Travers, The iterated restricted immediate snapshot (iris) model, en *14th Int'l Computing and Combinatorics Conference (COCOON'08)*, LNCS 5092, Springer, 2008, 487–496.
47. M. Raynal, *Concurrent Programming: Algorithms, Principles, and Foundations*, Springer, 450 páginas, 2012.
48. M. E. Saks y F. Zaharoglou, Wait-free k-set agreement is impossible: The topology of public knowledge, versión de congreso en acm stoc'1993, *SIAM Journal of Computing* **29(5)** (2000) 1449–1483.
49. M. Sipser, *Introduction to the Theory of Computation*, 3 ed., Course Technology, Junio 2012.
50. G. Taubenfeld, *Synchronization Algorithms and Concurrent Programming*, Pearson Prentice-Hall, 423 páginas, 2006.
51. A. M. Turing, On computable numbers, with an application to the entscheidungsproblem, en *Proceedings of the London Mathematical Society*, tomo 2(42), 1937, 230–65.