Universidad Nacional Autónoma de México

Posgrado en Ciencias e Ingeniería De la Computación
Instituto de Investigaciones en Matemáticas Aplicadas
y Sistemas

Algoritmos de Edición Colaborativa con diferentes garantías de consistencia

T E S I S

QUE PARA OPTAR POR EL GRADO DE:

Maestra en Ciencia e Ingeniería de la Computación

PRESENTA:

Diana Olivia Montes Aguilar

TUTOR:
Dr. Armando Castañeda Rojano
Instituto de Matemáticas, UNAM

Ciudad Universitaria, CDMX, mayo, 2022

Universidad Nacional Autónoma de México

Posgrado en Ciencias e Ingeniería De la Computación
Instituto de Investigaciones en Matemáticas Aplicadas
y Sistemas

Algoritmos de Edición Colaborativa con diferentes garantías de consistencia

T E S I S

QUE PARA OPTAR POR EL GRADO DE:

Maestra en Ciencia e Ingeniería de la Computación

PRESENTA:

Diana Olivia Montes Aguilar

TUTOR:
Dr. Armando Castañeda Rojano
Instituto de Matemáticas, UNAM

Ciudad Universitaria, CDMX, mayo, 2022

Agradecimientos

A Dios.
A mi tutor, el Dr. Castañeda.
A mis sinodales, el Dr. David Flores, el Dr. Sergio Rajsbaum, el Dr. Joel Trejo y el Dr. Jorge Luis Ortega, por su tiempo y los comentarios que enriquecieron mi trabajo.
A mi familia por su ejemplo.
A mis amigos por su comprensión.
A todas las personas que han enseñado a ser mejor.
A CONACYT por su apoyo en mi educación.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Los grandes sistemas computacionales como *Amazon*, *Google Workspace*, *Visa* o las nubes, por mencionar algunos, serían imposibles de reproducir en una arquitectura centralizada. Los sistemas distribuidos permiten atender peticiones de clientes en todo el mundo con baja latencia, aumentar la tolerancia a fallos, procesar y almacenar grandes volúmenes de datos, entre otras ventajas, pero también representan grandes retos. El teorema de CAP [8] establece las limitaciones de los sistemas distribuidos. *CAP* surge de *Consistencia (Consistency)*, *Disponibilidad (Availability)* y *Tolerancia a particiones (Partition Tolerance)*. A continuación definimos estás propiedades someramente:

1. *Consistencia*: significa que cada operación de lectura recupera el último récord escrito, es decir, se garantiza que toda la información está actualizada.

2. *Disponibilidad*: Aún si algunos nodos están apagados, se garantiza que el sistema estará disponible a través de otros nodos.

3. *Tolerancia a particiones*: implícitamente significa que cada nodo funciona independientemente de los demás.

El teorema estable que sólo es posible asegurar 2 de estas 3. Los múltiples factores que intervienen en el funcionamiento de un nodo en un sistema distribuido, hacen casi imposible evitar los comportamientos defectuosos, por lo que siempre es deseable tener la propiedad de tolerancia a particiones. Cabe aclarar que la consistencia a la que se hace referencia en el teorema CAP, es consistencia fuerte, el nivel supremo de esta propiedad.

Con el propósito de ofrecer alternativas a las limitantes delineadas por el teorema CAP, trabajos posteriores [7, 8, 27] han propuesto relajar el grado de consistencia que garantiza un sistema, en pos de la disponibilidad para procesar peticiones. Cada una de estas alternativas conlleva también, un nivel de dificultad y consumo recursos diferentes. El presente trabajo toma como guía una interesante alternativa a Bitcoin, para analizar el problema de la edición colaborativa y proporcionar 3 implementaciones que juegan con el balance entre consistencia y eficiencia. Dos de ellas son novedosas en el sentido en que ofrecen mayor grado de consistencia que las soluciones actuales. Las 3 ofrecen disponibilidad y tolerancia a particiones.

En los siguientes párrafos ahondaremos en el contexto del problema de la transferencia descentralizada de dinero, Bitcoin, su solución emblemática, y el artículo que propuso una alternativa efectiva sin *prueba-de-trabajo*. El artículo mencionado muestra además, una metodología para analizar los problemas distribuidos, misma que será nuestra guía en esta tesis. Dado que el problema analizar es la edición colaborativa, en la siguiente sección también incluimos el estado del arte en esta área.

1.1. Contexto

1.1.1. Transferencia descentralizada de dinero

Con el advenimiento de los pagos electrónicos, la privacidad ofrecida por el efectivo, desapareció.

Grupos anarquistas, como los *Cypherpunks*, en los 90's, discutieron ampliamente sobre la importancia de la privacidad. En 1993, uno de sus fundadores, *Eric Hughes* [15] publicó un manifiesto dónde sienta las bases del movimiento y su postura con respecto a la privacidad y su defensa.

> "... La privacidad en una sociedad abierta requiere de sistemas de transacciones anónimas. Hasta ahora el efectivo ha sido el principal sistema de este tipo.
>
> ... La privacidad en una sociedad abierta también requiere de la criptografía. Nosotros, los Cypherpunks estamos dedicados a construir sistemas anónimos. Defenderemos nuestra privacidad con criptografía, con sistemas anónimos de correo electrónico, con firmas digitales y con dinero electrónico."
>
> Eric Hughes, 9 Marzo 1993

Prueba-de-trabajo es un mecanismo fundamental en las criptomonedas actuales, pero originalmente surgió con el objetivo de frenar el abuso en el uso de recursos digitales. Su primera concepción fue hecha en 1992 por Dwork & Naor [13]. En ella se obligaba a los clientes a resolver una función matemática moderadamente complicada, antes de tener acceso a un recurso o mandar un mensaje. Posteriormente en 1997, sin el conocimiento de la publicación previa, Adam Back propuso *hashcash* [6], un mecanismo similar, que se basa en obtener colisiones parciales de hashes, ya que son complicadas de encontrar, pero fáciles de verificar, ahora conocidos como *acertijos criptográficos*. En 2004, Hal Fenney fue mas allá al desarrollar un sistema reutilizable de prueba-de-trabajo que era público y con el cuál los usuario podían validar la integridad del código y su funcionamiento (*servidor transparente*).

A finales de los 90's, surgieron las primeras propuestas de monedas digitales:

- *b-money*: En 1998, Wei Dai envió un mensaje al foro de los *Cypherpunks*, en él describía un sistema de transferencia de dinero. Las cuentas estaban representadas por llaves públicas, como actualmente se hace.
- *BitGold* fue diseñado por Nick Szabo también en 1998. Nunca fue implementado, pero logró reunir muchas ideas que componen las monedas digitales actuales. Los participantes debían resolver un acertijo criptográfico, utilizaba una cadena de hashes para garantizar inmutabilidad (posteriormente llamados blockchains) y era descentralizado con el objetivo de erradicar la necesidad de confiar en un

tercero. Sin embargo no logró resolver el problema del doble gasto. Este problema está presente cuando se permite que un *token* digital pueda ser gastado en más de una transacción.

En 2008, nació Bitcoin con la publicación de su protocolo [25], un año después su implementación fue liberada. Bitcoin consiste en una cadena de bloques distribuida, pública e inmutable. A esta estructura se le ha denominado *Blockchain*, y cada uno de los bloques contiene un conjunto de transacciones entre cuentas que son representadas por llaves públicas. Los bloques son introducidos uno a uno y dado que no existe un ente regulador, en cada ronda se elije un líder para que decida cuál es el siguiente bloque. La elección del líder se hace mediante el desafío computacional de resolver un acertijo criptográfico, que como expusimos anteriormente es conocido como *prueba-de-trabajo*. En términos del cómputo distribuido, esto es un mecanismo de consenso, si bien no en toda su expresión, ya que está teóricamente demostrado que no se puede tener consenso en un ambiente asíncrono con actores propensos a fallos, accidentales o intencionales [16]. Pero las posibilidades de tener un error en el sistema bajo este mecanismo, disminuyen significativamente cuando la mayoría de los participantes son correctos. Por lo que podemos decir que *Bitcoin* es seguro, es decir evita el doble gasto con muy altas probabilidades.

En los años de vida de Bitcoin han surgido inconvenientes con su mecanismo de consenso. A continuación enumeramos algunos:

- *Baja escalabilidad*. La cantidad de transacciones pendientes ha ido en aumento mientras que la capacidad para procesarlas se ha mantenido, provocando un cuello de botella en el sistema.
- *Desempeño poco competitivo* con respecto a los sistemas centralizados como Visa.
- *Centralización*. Los participantes con equipos de cómputo más poderosos tienen mayor probabilidad de ser líderes y con ello mayor poder de decisión sobre los bloques que se colocan en la cadena.
- *Contaminación*. Se estima que la producción de CO_2, debido a la energía que consume minar Bitcoin, anual es equivalente a la emitida por países como Sri Lanka o Jordania [30].

Ante estos inconvenientes, los académicos y la industria han respondido con alternativas como las siguientes:

- Una amplia gama de algoritmos de consenso [5, 10].
- DAG's. Modificación en la estructura de los bloques, en lugar de ser una cadena, es una gráfica dirigida acíclica [26].
- No basar la corrección del sistema en el consenso. Un trabajo publicado en 2019 demostró que el problema de la transferencia de dinero, es fácil comparado con los

problemas que el consenso resuelve [17], además en un artículo posterior, explican la implementación de un sistema descentralizado, escalable, seguro, robusto y auditable que procesa tantas transacciones por minuto como los sistemas centralizados [11]. A este sistema lo denominan *Astro* y está basado en los resultados teóricos del primer artículo.

Esta última alternativa, además de proveer resultados novedosos, muestra una interesante y sólida metodología. Comienza por plantear el problema de la transferencia de dinero, tal como lo definió S. Nakamoto, en un objeto concurrente. La implementación de este objeto, pone de manifiesto, que el nivel de sincronización, según la escala del número de consenso [18], es el más básico, contrastando con la idea actual de la necesidad del consenso, el nivel superior de sincronización. En la segunda parte del artículo, se propone una implementación del objeto en el modelo de paso de mensajes asíncronos con actores bizantinos y se demuestra que no es vulnerable al ataque de doble gasto.

1.1.2. Edición colaborativa

En la sección anterior abordamos lo concerniente a las criptomonedas, continuación discutiremos nuestra otra área de interés, la *Edición Colaborativa*.

Edición colaborativa es la modificación de un mismo contenido por varias personas posiblemente desde diferentes lugares. Típicamente cada una de las partes cuenta con una copia local, llamada réplica, y las actualizaciones locales sobre cada réplica pueden realizarse libremente sin esperar respuesta o confirmación, es decir de manera optimista. En un momento posterior, cuando las actualizaciones locales se publican a las demás réplicas, pueden aparecer los conflictos. Dependiendo del contenido y su función, los conflictos pueden resolverse con intervención del usuario, como en el caso de Git o Mercurial, o son delegados a la aplicación, como el caso de Google Docs u Overleaf. En nuestro caso nos centraremos en las aplicaciones que hacen transparente al usuario la presencia de conflictos.

Concretamente definimos el problema de la edición colaborativa como la modificación de un estado común a varios participantes, sin que ellos tengan que preocuparse de los conflictos por sus actualizaciones, realizadas de manera optimista y garantizando que los estados de cada réplica lleguen a ser los mismos, es decir, converjan.

En la teoría y en la práctica existen principalmente dos enfoques que se ocupan de este problema:

- *Transformación Operacional (Operational Transformation, OT)*: surgió a finales de la década de los 80's [14]. Este enfoque consiste en resolver los conflictos entre las actualizaciones, transformándolas en operaciones equivalentes que no produzcan conflicto respetando dos propiedades, *TP1* y *TP2*. En su concepción, no requiere de la presencia de una entidad centralizada que tenga la vista global del estado y aplique las transformaciones, pero todos los algoritmos descentralizados, salvo uno [24] de implementación muy complicada, han resultado con errores e

incapaces de alcanzar la consistencia en las réplicas bajo ciertas circunstancias. Esto ha llevado a que los algoritmos exitosos usen un servidor centralizado, como en el caso de Google Docs. Sin embargo este es el enfoque más usado en las aplicaciones de edición colaborativa.

- *Tipos de datos replicados libres de conflictos (CRDT)*: surgió en 2011 [29] y consiste en construir objetos cuyas operaciones sean conmutativas, de modo que, sin importar el orden en que se apliquen en las diferentes réplicas, el estado será el mismo. Estos objetos y sus propiedades dieron paso al concepto de consistencia eventual fuerte. Es importante observar que no todas las aplicaciones de edición colaborativa pueden modelarse de esta manera, pues no en todos los casos se logra construir objetos con operaciones conmutativas. Si bien este enfoque ha tomado fuerza en la academia y en la industria, ninguna plataforma de gran relevancia las usa actualmente.

1.2. Motivación

Después de estudiar detenidamente el artículo *Consensus number of a cryptocurrency* [17] y la sencilla solución que ofrece, consecuencia del análisis teórico previamente realizado, nos preguntamos *¿es posible utilizar esa solución en otro problema, brindando beneficios adicionales a las soluciones de facto?* Al revisar las soluciones ofrecidas para el problema de la edición colaborativa vimos que las dos clases de consistencia que ofrecen son las siguientes [29]:

1. *Consistencia eventual*: Tiene las siguientes propiedades:

   - *Entrega eventual*: Una actualización recibida por una réplica correcta, eventualmente será recibida por todas las réplicas correctas.
   - *Convergencia*: Réplicas correctas que han recibido las mismas actualizaciones eventualmente alcanzan estados equivalentes.
   - *Terminación*: Todas las ejecuciones terminan.

   De modo intuitivo, garantiza que, si hay un periodo en el que no se hagan actualizaciones, entonces todos los nodos alcanzarán el mismo estado. Este modelo permite actualizaciones concurrentes que pueden causar conflictos, por ello requiere un mecanismo para resolverlos, en varios casos, esto significará realizar *rollback* y recalcular las modificaciones.

2. *Consistencia eventual fuerte*: Cumple con:

   - *Consistencia eventual*.
   - *Convergencia fuerte*: Réplicas correctas que han recibido las mismas actualizaciones tienen estados equivalentes.

Este modelo garantiza que siempre que dos nodos hayan recibido el mismo conjunto de actualizaciones, posiblemente en diferente orden, su vista del estado compartido es idéntica.

La solución propuesta por nuestro artículo base [17], ofrece un grado mayor de consistencia sin perder la característica de ser responsivo, deseable en un sistema de edición colaborativa. Es por ello que decidimos seguir su metodología para tratar el problema de la edición colaborativa, no sin aplicar algunos ajustes requeridos para este problema.

1.3. Objetivos y contribución

El objetivo de este trabajo es exponer un análisis teórico de la edición colaborativa del cuál surgen, de manera natural, tres propuestas de implementación. Dichas implementaciones basan su efectividad, ampliamente, en la conmutatividad de las operaciones, como en el caso de los CRDT's, pero en dos de ellas proveen un grado adicional de consistencia. La primera respeta el orden por fuente, es decir, las réplicas aplican las modificaciones remotas en el orden que fueron realizadas por su emisor. La segunda respeta el orden de tiempo real, es decir, las dependencias de las modificaciones entre los diferentes actores. Se podría argumentar que los sistemas de edición colaborativa actuales, como Google Docs, ya proveen soluciones funcionales, pero recordemos que su correcto funcionamiento necesitan un servidor que haga el trabajo de ordenamiento correcto de las operaciones. En cambio estas soluciones son puramente distribuidas.

1.4. Justificación

Los modelos de concurrencia y paso de mensajes se han vuelto los más relevantes en los sistemas actuales. Sus usarios esperan una respuesta cada vez más rápida o quieren almacenar cada vez más información que posteriormente desean procesar y consultar eficientemente. Si bien ya hay muchas herramientas que facilitan la implementación bajo estos modelos, el desconocimiento de conceptos fundamentales puede llevar a la elección erronea de la herramienta, con respecto a los requerimientos del sistema. Por el otro lado, el acercamiento académico, en México particularmente, en estás áreas no es abundante ni exahustivo lo que evita la promoción de investigación en estas áreas.

Este trabajo expone un problema conocido y fácil de plantear, sobre el cuál se razona con un enfoque téorico haciendo énfasis en dos aspectos determinantes de un sistema: Consistencia y disponiblidad. También se ofrecen protocolos alternativos para la solución de este problema. El desarrollo de las pruebas que sustentan los protocolos, es un ejercicio poco realizado con exhaustividad. En la mayoría de la bibliografía que hemos revisados, las pruebas ofrecidas, si bien no son incorrectas, obvian muchos de los detalles de linealizabilidad y concurrencia. Las pruebas presentadas en este trabajo

exponen con mejor detalle los puntos críticos que se deben tomar en cuenta cuando se realizan este tipo de demostraciones.

Además la contribución al problema de la edición colaborativa, se considera que este trabajo puede abonar al repertorio disponible que se consulta cuando se desea demostrar protocolos de esta naturaleza.

1.5. Organización de la Tesis

Para inicial la construcción teórica de nuestro objeto de edición colaborativa, presentamos las bases de los conceptos de cómputo concurrente en el capítulo 2. En el capítulo 3 ofrecemos la discusión teórica del problema de la edición colaborativa y como resultado llegamos a un objeto concurrente de edición colaborativa, cuyo estado contiene todas modificaciones sobre el contenido, y semánticamente su valor actual, que corresponde a un subconjunto de esas modificaciones. Si bien la edición colaborativa no es exclusiva de documentos de texto, en este trabajo lo enfocamos de esa manera, aunque observamos que con algunas simples modificaciones puede agregarse otro significado al contenido del estado. Las operaciones de este objeto concurrente son, por simplicidad, leer el valor del documento, insertar o borrar un carácter. La corrección de estas operaciones está basada en la existencia de un conjunto denso de identificadores para cada carácter insertado. El Capítulo 4 propone un conjunto de identificadores con esta característica. El capítulo 5 propone las implementaciones en un ambiente distribuido para la edición colaborativa. Se incluyen las pruebas de corrección de las implementaciones. Finalmente el capítulo 6 tiene las conclusiones de este trabajo.

Capítulo 2

Concurrencia en Memoria Compartida

La primera aproximación al problema de la edición colaborativa será en el modelo concurrente con comunicación por memoria compartida. Esto permitirá dejar de lado los inconvenientes propios de la comunicación por paso de mensajes, pero expondrá los puntos finos inherentes a la concurrencia.

Este capítulo está dedicado a exponer el modelo concurrente en memoria compartida. Empezaremos con sus antecedentes, el modelo secuencial, posteriormente daremos su definición formal y las características de sus integrantes, objetos y procesos. Ahondaremos en la definición de los objetos concurrentes, sus implementaciones y los criterios de corrección de las mismas. Finalmente presentaremos abstracciones básicas de memoria compartida, como son los registros y el *Snapshot Atómico*.

2.1. Modelo Secuencial

Este modelo es herencia de la arquitectura propuesta por John Von Neumann en 1945 [32]. Esta arquitectura cuenta con memoria para almacenar datos e instrucciones. Las instrucciones son ejecutadas, *una a una*, por la unidad de procesamiento según la unidad de control lo indique. A su vez, esta arquitectura también comparte la característica del procesamiento secuencial de instrucciones con la máquina abstracta de cómputo propuesta por Alan Turing en 1936 [31].

En el *modelo secuencial* existen *objetos* que son modificados por un *proceso* mediante operaciones atómicas. Como consecuencia, las operaciones se realizan una tras otra, sin traslaparse.

2.1.1. Objetos Secuenciales

Un *objeto* es una estructura que almacena *datos* y provee medios para recuperarlos o modificarlos, conocidos como *métodos*. La descripción de los datos del objeto entre invocaciones a métodos −cuando no se está manipulando el objeto− es llamado el *estado* del objeto.

Los objetos pertenecen a un *tipo* $\mathcal{T}$, que es la descripción general de los métodos y sus datos. Ya que los métodos son la interfaz para interactuar con el objeto, resulta natural definirlos en términos de *precondiciones*, el estado del objeto antes de invocarlos, y *postcondiciones*, el estado resultante del objeto al término del método y el valor de respuesta. A las precondiciones y postcondiciones de los métodos se les llama *especificación secuencial*.

Formalmente un tipo $\mathcal{T}$ es una *5-tupla* (Q, q_0, M, R, Δ), donde

- Q es el conjunto de todos los posibles estados que el objeto puede alcanzar.
- q_0 es el estado inicial.
- M, el conjunto de métodos.
- R, el conjunto de todos los posibles valores de respuesta de los métodos.

- Δ es una función $Q \times M \to Q \times R$, que establece las precondiciones y postcondiciones de cada método. $\Delta(q, m) = (q', r)$ significa que el objeto estaba en el estado q antes de la invocación del método m. q' es el estado resultante después de la ejecución de m y r es su respuesta. Si no hay un único estado resultante, sino un conjunto de estados, decimos que el método es *no determinista*.

Una implementación es la especificación imperativa de un tipo. Los datos están contenidos en estructuras de datos y los métodos son expresados como un algoritmo que detalla las operaciones atómicas −manipulaciones sobre las estructuras de datos y cálculos para computar la respuesta− que un proceso debe realizar para cumplir con la especificación secuencial. Una *ejecución* de una implementación es una secuencia de llamadas a métodos. Una *llamada a método* es un intervalo que comienza con un evento de *invocación* y termina con un evento de *respuesta*.

Para ilustrar los conceptos, considere un objeto de tipo *registro de lectura-escritura* S, donde S es el conjunto de valores que el registro puede guardar:

- $Q := S \cup \{null\}$. *null* es un valor especial que indica que no se han hecho escrituras sobre el registro.
- q_0 es *null*.
- $M := \{write(x) : x \in S, read()\}$
- $R := S \cup \{null\} \cup \{true\}$
- Δ consiste de las siguientes reglas donde $q \in Q$ es algún estado del objeto:
  - $\Delta(q, read()) = (q, q)$
  - $\Delta(q, write(x)) = (x, true)$

Una implementación secuencial del registro de escritura-lectura es la siguiente:

Algoritmo 1: Implementación de un registro de lectura y escritura de tipo S.

Datos:
S valor, inicialmente *null*

```
1 Metodo read():
2     return valor
3 Metodo write(nuevoValor):
4     valor = nuevoValor
5     return true
```

Finalmente presentamos una ejecución de un registro R:

Las líneas punteadas representan el tiempo. Los intervalos representan llamadas a métodos; empiezan con un evento de invocación, $R.método$, y terminan con un evento de respuesta, $R : respuesta$. Sólo hay un proceso interactuando con el registro.

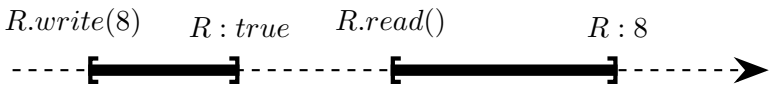

Figura 2.1: Ejecución de la implementación de un registro de lectura-escritura

Las definiciones de arriba tienen perfecto sentido cuando el contexto es secuencial, es decir, sólo un proceso manipula el objeto simultáneamente. Cuando hay más de un proceso modificando el objeto, las llamadas a método pueden traslaparse, perdiendo su orden secuencial. Consecuentemente, las definiciones de estado, precondiciones y post-condiciones ya no son claras. Las siguientes secciones proveen criterios para describir y evaluar el comportamiento de objetos concurrentes.

2.2. Modelo concurrente

En el *modelo concurrente* hay un conjunto $\Pi = \{p_1, p_2, ..., p_n\}$ de *procesos asíncronos*, que se comunican a través de *objetos* en memoria compartida con operaciones atómicas. Los procesos son secuenciales, es decir, no realizan una invocación nueva sin antes haber recibido respuesta de la previa. El retardo que presentan los procesos mientras realizan una invocación es arbitrario y pueden presentar fallas de tipo paro −detienen su ejecución de manera permanente− en cualquier momento.

2.2.1. Implementaciones y sincronización

Si bien las implementaciones de los métodos constan de operaciones atómicas, estos (los métodos) no lo son. Esto da lugar, en el modelo concurrente, a que las llamadas a métodos se traslapen. Continuando con el ejemplo de los registros de lectura y escritura, observemos cómo se comporta la implementación 1 en un ambiente concurrente. La figura 2.2 muestra una ejecución de un registro compartido por tres procesos, p, q y r.

Las escrituras de los procesos p y r son concurrentes, de modo que no es claro qué valor prevalece en el registro cuando el proceso q realiza la lectura, por lo tanto ¿bajo qué criterio se podría concluir si la implementación es correcta o no en el modelo concurrente? Como vemos, la especificación secuencial por si misma no contesta esta pregunta, pues nos enfrentamos a una ejecución no secuencial. Se requiere agregar a la implementación algún tipo de coordinación o sincronización de los procesos para ordenar las invocaciones concurrentes. Hay dos tipos de sincronización, la *sincronización bloqueante* y la *no-bloqueante*.

2.2.1.1. Sincronización bloqueante

E. Dijkstra observó la necesidad de coordinar procesos secuenciales que compartían variables. En su artículo [12] propuso la idea de *sección crítica*, que es una parte del

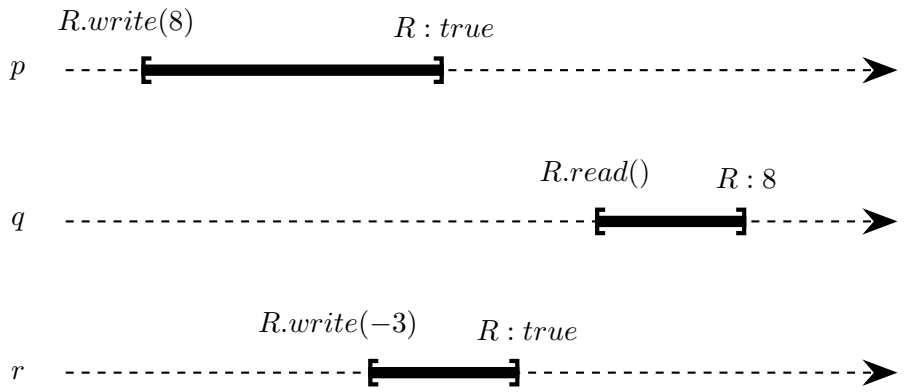

Figura 2.2: Ejecución concurrente de un registro de lectura-escritura

código en la que se accede a, o se manipula la memoria compartida.

La idea intuitiva detrás de este tipo de solución es que cada sección crítica tiene un candado que permanece abierto mientras no haya nadie dentro de ella. Cuando un proceso quiere entrar, primero debe obtener el candado y cerrarlo. No es posible que ningún otro proceso entre hasta que el proceso salga y libere el candado.

La sección crítica debe contar con las siguientes características:

- *Exclusión mutua.* Las ejecuciones de las secciones críticas de los procesos no se traslapan.
- *Libre de Deadlock.* Si algún proceso intenta adquirir el candado, entonces algún proceso adquirirá el candado. Es decir, se garantiza el progreso en el sistema, pero no en cada proceso.

Identificamos estas soluciones como bloqueantes porque retrasos inesperados en un proceso pueden evitar que todos los demas avancen, por ejemplo si un proceso dentro de las sección crítica se detiene. Aunque el objeto es compartido por varios procesos, con este tipo de soluciones se elimina la concurrencia, es decir, los accesos a los objetos compartidos se vuelven secuenciales. Al respecto, la *ley de Amdahl* expone el detrimento en la velocidad de ejecución cuando hay trabajo necesariamente secuencial [2]. Sin embargo, al ser secuencial la ejecución, puede evaluarse con la especificación secuencial.

2.2.1.2. Sincronización no-bloqueante

Cuando el retraso arbitrario e inesperado de un proceso no evita que los otros procesos continúen su ejecución, decimos que la sincronización es *no-bloqueante*.

En 1991, *M. Herlihy*, publicó un artículo en el que jerarquiza las primitivas de sincronización no-bloqueantes a partir de su poder para resolver el problema del consenso [18]. Al número de procesos para los que la primitiva es capaz de alcanzar consenso, le llamó *número de consenso*. Podría pensarse que siempre es bueno realizar las

implementaciones con la primitiva más poderosa sin importar los requerimientos del problema, pero se debe observar que las primitivas más poderosas son más costosas de implementar y en la práctica demandan más recursos. Por otro lado si se usa una primitiva con menor poder del necesario, la implementación no cumplirá con su propósito. La siguiente tabla muestra la jerarquía de las primitivas de sincronización según su número de consenso.

Número de Consenso	Objeto
1	registros de lectura/escritura
2	test&set, swap, fetch&add, cola, pila
⋮	⋮
$2n-2$	asignación simultánea de n-registros
⋮	⋮
∞	memoria-a-memoria move y swap, cola aumentada, compare&swap, fetch&cons, sticky byte

Tabla 2.1: Jerarquía de las primitivas de sincronización no bloqueantes.

Probar la corrección de este tipo de implementaciones es más laborioso, pero garantiza mayor desempeño al conservar la concurrencia. Podemos observar que la sincronización, a diferencia de la técnica anterior, no radica en eliminar la concurrencia, sino en utilizar un objeto que provea el poder de sincronización necesario. En este caso, la especificación secuencial no es el criterio adecuado para evaluar la corrección de una implementación. Las secciones sucesivas proveen los conceptos necesarios para entender y aplicar los criterios de corrección para este tipo de implementaciones. Nos interesa abordar ese tema porque en el siguiente capítulo tomaremos el problema de la edición colaborativa, lo analizaremos y propondremos un objeto con el poder de sincronización necesario y suficiente para resolverlo.

2.2.2. Conceptos generales

Los siguientes conceptos fueron tomados del libro *The Art of Multiprocessor Programming* de M. Herlihy y N. Shavit [20]:

Un *sistema concurrente* es una colección de procesos secuenciales y asíncronos que se comunican mediante la implementación de objetos compartidos. Una *ejecución* de un sistema concurrente es la interacción de los procesos con los objetos compartidos, es decir, una secuencia de eventos: invocaciones a métodos, operaciones atómicas sobre los objetos y respuestas a métodos. La ejecución está modelada por una *historia*, H,

que es la secuencia finita de los eventos de invocación y respuestas de métodos. Una *subhistoria* de H es una subsecuencia de eventos de H. Escribimos una invocación a método como $\langle A.m(args^*)\ p\rangle$ donde A es un objeto, m un método, $args^*$ una secuencia de argumentos y p un proceso. $\langle A : t(res^*)\ p\rangle$ representa un evento de respuesta, donde t es `OK` ó una excepción, res^* una secuencia de valores resultantes.

Una respuesta *corresponde* a una invocación si llevan el mismo objeto y proceso y no hay entre ellas otra invocación o respuesta con el mismo objeto y proceso. Una *llamada a método* en una historia H es una pareja formada por una invocación y su correspondiente respuesta en H. Necesitamos distinguir llamadas que concluyen de aquellas que no: Una invocación *pendiente* en H es aquella a la que no le corresponde ninguna respuesta posterior. Una *extensión* de H es una historia construida al agregar respuestas a cero o más invocaciones pendientes. A veces se ignoran las invocaciones pendientes: *completa(H)* es la subhistoria de H que consiste de todas las invocaciones y sus respuestas correspondientes.

En algunas historias las llamadas a métodos no se traslapan: Una historia H es *secuencial* si el primer evento es una invocación y cada invocación, excepto tal vez la última, es seguida inmediatamente por su respuesta correspondiente.

Una *subhistoria de un proceso p* en H, $H|p$, es una subsecuencia de eventos en H cuyo nombre de proceso es p. Una *subhistoria de un objeto A*, $H|A$, se define de manera similiar pero usando las etiquetas del nombre del objeto. Al final, lo que importa es como cada proceso ve lo que sucede: dos historias H y H' son *equivalentes* si para cada proceso p, $H|p = H'|p$. Finalmente necesitamos identificar historias sin sentido: En una historia *bien formada* las subhistorias de procesos son secuenciales, a esa secuencia se le conoce como *orden de programa*; el orden en que cada proceso realizó las llamadas. Todas las historias que consideremos aquí estarán bien formadas. Se puede notar que las subhistorias de procesos de historias bien formadas, son siempre secuenciales.

Sea $H|A = m_1, m_2, m_3, ..., m_n$ una historia secuencial del objeto A de tipo $\mathcal{T}_A = (Q_A, qA_0, M_A, R_A, \Delta_A)$. qA_0 es el estado del objeto antes de m_1. En general qA_{i-1} es el estado del objeto antes de la llamada al método m_i y qA_i es el estado del objeto al terminar la llamada a método m_i. Es posible calcular qA_i, pues se conoce la especificación secuencial de A. $H|A$ es *legal* si para cada llamada a método, m_i, se tiene que: $(qA_{i-1}, nom(m_i), qA_i, resp(m_i)) \in \Delta_A$. Donde $nom(m_i)$ y $resp(m_i)$ obtienen la información de la invocación −el nombre del método y los argumentos− y la respuesta de la llamada m_i, respectivamente.

Decimos que una llamada a método m_0 *precede* a otra llamada a método m_1 en la historia H, $m_0 \prec_H m_1$, si m_0 termina antes de que m_1 empiece: la respuesta de m_0 sucede antes de la invocación de m_1. Notemos que $\prec_H$ establece un *orden parcial* (relación irreflexiva y transitiva) en invocaciones de H, pues en *llamadas con concurrentes* no se puede hablar de precedencia. A esta relación también se le conoce como *orden de tiempo real*.

2.2.3. Propiedades de las implementaciones de objetos concurrentes

Para evaluar el comportamiento de los objetos concurrentes se deben tomar en cuenta sus propiedades de *corrección* y *progreso*.

2.2.3.1. Corrección

La propiedad de corrección establece que: *nada malo pasa*. Esta definición puede parecer un poco vaga, pero esencialmente dice que la implementación del objeto no debe desviarse de la especificación. Ya que la especificación secuencial ya se conoce y está bien definida, todas las nociones de corrección para objetos concurrentes y sus implementaciones están basadas en alguna definición de equivalencia con ella. La condición de corrección en la que nos enfocaremos es *linealizabilidad* [21].

Para demostrar que una implementación es *linealizable* −tiene la propiedad de linealizabilidad− se debe demostrar que la historia H de cualquier ejecución cuenta con una extensión, H', y una historia secuencial legal L, tal que:

1. $completa(H')$ es equivalente a L, es decir, para todo proceso p, $completa(H')|p = L|p$.
2. $\prec_H \subseteq \prec_L$, es decir L respeta el orden de tiempo real de H.

Nos referimos a L como una linealización de H. Puede haber múltiples linealizaciones de H.

En términos llanos, lo que dice la definición de linealizabilidad es que se puede obtener una historia secuencial L a partir de H conservando el mismo orden entre las invocaciones que no son concurrentes, y permitiendo reordenar, a conveniencia, las llamadas a métodos concurrentes. Si L resulta en una ejecución legal, de acuerdo a la especificación secuencial del objeto, entonces, la implementación es linealizable.

En la ejecución concurrente de la figura 2.2, ejemplificaremos los conceptos antes definidos.

A continuación escribimos la historia asociada a la ejecución, H. Cómo puede observarse es concurrente y no tiene invocaciones pendientes.

```
<R.write(8) p>
<R.write(-3) r>
<R:OK(true) p>
<R:OK(true) r>
<R.read() q>
<R:OK(8) q>
```

Hay 3 llamadas a método, `M1, M2, M3`:

```
M1 = (<R.write(8) p>, <R:OK(true) p>)
M2 = (<R.write(-3) r>, <R:OK(true) r>)
M3 = (<R.read() q>, <R:OK(8) q>)
```

El orden de tiempo real, $\prec_H$, es el siguiente:

$$\texttt{M1} \prec_H \texttt{M3}$$
$$\texttt{M2} \prec_H \texttt{M3}$$

`M1` y `M2` son concurrentes, por lo que pueden reordenarse.

A continuación se proponen dos linealizaciones, las cuales deben ser legales, respetar el orden de programa y de tiempo real.

- `L1: M1, M2, M3`

  1. Para comprobar si es legal es necesario reproducir la secuencia de llamadas a método empezando desde el estado inicial, *null*:

     ```
     (null,write(8)) -> (8,true),
     (8,write(-3)) -> (-3, true),
     (-3,read()) -> (8, true)
     ```

     Como podemos observar el último paso no cumple con Δ pues en una lectura está regresando un valor diferente al del estado. Por lo tanto `L1` no es legal y no puede ser una linealización de H.

- `L2: M2, M1, M3`

  1. Procedemos de la misma manera para revisar la legalidad

     ```
     (null,write(-3)) -> (-3,true),
     (-3,write(8)) -> (8, true),
     (8,read()) -> (8, true)
     ```

     Todos los pasos cumplen con Δ. Por lo tanto es legal.
  2. Es sencillo verificar que respeta el orden de programa, pues la historia es muy sencilla y cada proceso sólo realiza una llamada.
  3. También respeta el orden en tiempo real pues `M3` aparece al último.

  Por lo tanto `L2` sí es una linealización de H.

Para demostrar que una implementación es linealizable se debe tomar una ejecución arbitraria y demostrar que su historia asociada tiene una linealización.

La siguiente figura muestra una ejecución no linealizable.

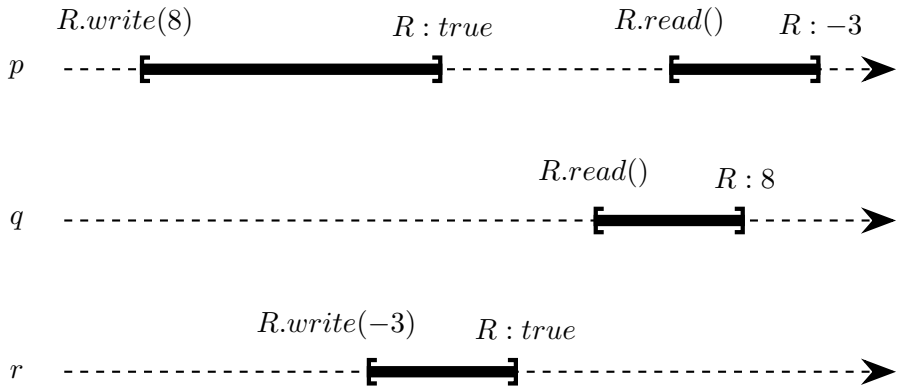

Figura 2.3: Ejecución concurrente no linealizable.

Intuitivamente, la razón por la que no es linealizable es porque las escrituras de diferentes valores toman efecto estrictamente antes que las lecturas. Es decir, cuando empiecen las lecturas hay un estado determinado y no importa el número de lecturas posteriores, todas tienen que ver lo mismo, pero como se muestra en la ejecución, las lecturas entregan valores diferentes.

2.2.3.2. Progreso

La propiedad de *progreso* establece que *algo bueno eventualmente pasa*. Sin ser muy específica la definición de esta propiedad, deja ver que su objetivo es garantizar dinamismo en el sistema. Existen dos tipos de condiciones de progreso, las *bloqueantes* y las *no bloqueantes*, las bloqueantes garantizan que algún proceso tendrá exito, y la no bloqueantes garantizan que todos los proceso tendrán éxito. Sin embargo, hay que hacer énfasis en cuánto a cuáles son *todos* los procesos a los que nos referimos. Recordemos que puede haber procesos que detienen su ejecución, es decir, son fallidos. Por ello con *todos* nos referemos a los procesos correctos. Los procesos correctos son aquellos, que en una ejecución infinita, producen una secuencia infinita de eventos.

De las diversas condiciones de progreso [19], nos centraremos en la condición de *libre de espera*, una condición no bloqueante. Intuitivamente, garantiza que cada invocación hecha por un proceso correcto termina en un número finito de pasos, es decir, retrasos arbitrarios e inesperados de algunos procesos no evitan que los demás continúen su ejecución.

Formalmente decimos que una implementación es *libre de espera* si en cada una de sus ejecuciones infinitas pasa que: todo proceso correcto p, completa un número infinito de invocaciones a métodos.

2.3. Modelos de Memoria Compartida

En la sección anterior estudiamos el modelo concurrente en memoria compartida y utilizamos como ejemplo los registros. Los registros son el modelo concurrente de memoria compartida más simple, pero a partir de ellos se construyen modelos mas robustos, como el *Snapshot*. Estos dos modelos serán el material de exposición en lo que resta del capítulo pues nos basaremos en ellos para proponer nuestro objeto de edición colaborativa en el siguiente capítulo.

2.3.1. Registros

En una sección anterior ahondamos con mayor detalle en el tipo *registro*. Por ahora basta recordar que son objetos que encapsulan un valor que puede ser observado con el método *read* y modificado por el método *write*. Estos objetos son los más simples considerados en el cómputo concurrente y son compartidos por procesos secuenciales y asíncronos. Hay una gama de registros que toma en cuenta los siguientes criterios:

1. El conjunto de valores que pueden albergar (Booleanos, enteros,...), *binarios* o *multivaluados*.

2. La cardinalidad, uno o muchos, de los procesos que realizan lecturas o escrituras sobre un registro.

3. Grado de consistencia que proveen. Cuando un registro es accedido secuencialmente, el comportamiento es fácil de definir: una invocación de lectura regresa el último valor escrito. Cuando se accede de manera concurrente hay, principalmente, 3 variantes consideradas:

   a) *Seguridad*. Únicamente garantiza que una lectura que no es concurrente con una escritura regresa el último valor escrito. Si la escritura es concurrente con la lectura, el valor de retorno puede ser cualquier valor en el rango de valores del registro, incluyendo un valor que nunca ha sido escrito. Por tanto, solo soporta un escritor.

   b) *Regularidad*. Un registro *regular* es un registro seguro y además garantiza que una lectura concurrente con varias escrituras regresa cualquiera de los valores escritos concurrentemente con esta lectura o el anterior a las escrituras concurrentes. En este contexto puede suceder un fenómeno llamado *inversión antiguo/nuevo* que consiste en que dos lecturas secuenciales que son concurrentes a varias escrituras pueden regresar primero algún valor nuevo y la siguiente lectura el valor anterior a las escrituras.

   c) *Atomicidad*. Un registro atómico asegura linealizabilidad, es decir, además de ser seguro y regular, garantiza que la *inversión antiguo/nuevo* no sucede, la segunda lectura regresa el mismo valor o un valor más nuevo.

Los registros más poderosos son los registros atómicos MRMW multivaluados y los menos poderosos son los registros seguros SRSW booleanos.

En cuanto a sus implementaciones se puede demostrar [20] que los registros más poderosos pueden ser construidos a partir de los más sencillo.

2.3.2. Snapshots

Además de poder leer un sólo valor almacenado en un registro, resulta útil que el lector pueda adquirir, en una sola operación, el vector de los últimos valores escritos por cada proceso. La abstracción basada en registros atómicos MRSW con estas características, es conocida como *Snapshot Atómico*. A cada uno de los procesos que interactúan con el *Snapshot* se les asigna un registro. Las operaciones que ofrece el *Snapshot* son [1]:

- *scan*(): provee una captura del vector de registros garantizando que no hay modificaciones mientras se toma la captura, es decir, son los últimos valores escritos por cada proceso.
- *update*(d): El proceso escribe el valor d en su registro. Ningún proceso puede modificar un registro diferente al que le fue asignado.

Se ha demostrado que el *Snapshot Atómico* tiene una implementación libre de espera y linealizable [1] con los registros atómicos MRSW.

Capítulo 3

Tipo Documento Colaborativo

A continuación daremos la intuición de un documento colaborativo, su definición formal como un objeto concurrente, los inconvenientes que resultan de la concurrencia y cómo abordarlos.

3.1. Documento Colaborativo

Pensemos en un documento sencillo como el siguiente:

```
dos
```

Lógicamente está compuesto por elementos indivisibles −en este caso caracteres− tomados de un conjunto determinado, llamado alfabeto, que tienen asociado un orden:

```
(1,d), (2,o), (3,s)
```

Pero ese orden es dinámico; con la inserción o borrado de caracteres puede cambiar:

```
uno dos
```

El nuevo orden es:

```
(1,u), (2,n), (3,o), (4, ), (5,d), (6,o), (7,s)
```

Sin embargo, cuando se edita un documento, no se piensa en modificar posiciones, sino en agregar o quitar caracteres, por lo que resulta más acercado a la semántica de un documento, asignar identificadores a los caracteres de modo que aunque se mueva la posición, sigan siendo accesibles mediante su identificador. Además de la presencia de un carácter en el documento, también se tiene que preservar la intención del editor al ponerlo. Escogeremos el conjunto de identificadores de tal manera que, además de identificar, conserven el orden. Pensando en las maneras posibles de insertar un carácter, nuestro conjunto de identificadores debe cumplir lo siguiente:

- *Infinito.* No resulta útil un documento con una cantidad limitada de elementos.
- *Ordenado.* Si el conjunto no tiene un orden total, no reflejará la intención con que fue insertado el carácter al que representa.
- *Denso.* Es común que un carácter se inserte entre otros que ya estaban, por lo que resulta necesario que entre cuales quiera dos identificadores siempre pueda obtenerse otro que esté entre esos dos.
- *Abierto.* Si en el conjunto no hay máximo ni mínimo se garantiza que las inserciones al principio y al final siempre sean posibles.

No está de más resaltar la importancia de que los identificadores sean únicos en un documento, de lo contrario, no sería claro a cuál carácter se están aplicando las modificaciones.

Recordemos que nos interesa que el documento sea modificado por varios editores, un documento colaborativo. En la siguientes secciones, se discutirán los retos que impone la concurrencia sobre el documento.

Resumiendo, un documento es una colección de elementos a los que llamaremos átomos, por ser la unidad más pequeña que lo constituye. Nuevos átomos pueden ser agregados y los ya existentes pueden ser modificados, para lo cual es necesario que cada uno cuente con un identificador que permita su manipulación y preserve la intención de su inserción.

3.2. Tipo Documento Colaborativo

Formalmente un objeto de *tipo documento*, $\mathcal{T}$, es una 5-tupla (Q, q_0, M, R, Δ), dónde:

- Q es el conjunto de todos los posibles *estados* en los que puede estar el documento. Sea Σ el conjunto finito de *átomos* posibles que pueden constituir un documento, $\mathcal{C}$ el conjunto de *identificadores* para los átomos, es necesario que este conjunto tenga un orden total y sea denso, es decir, entre cada dos elementos, siempre se puede encontrar otro diferente. Por simplicidad en la exposición, los átomos sólo tendrán dos estados; insertado y borrado, 1 y 0, respectivamente.

  $Q = \mathcal{P}(\Sigma \times \mathcal{C} \times \{0, 1\})$, dónde $\mathcal{P}$ indica el conjunto potencia.

- q_0 es el *estado inicial* del documento y siempre es vacío, $\emptyset$.

- M, es el conjunto de operaciones que provee el objeto.

$$\begin{aligned} M = &\{estado()\} \\ &\cup \{lee(x) : x \in \mathcal{C}\} \\ &\cup \{agrega(a, x) : a \in \Sigma, x \in \mathcal{C}\} \\ &\cup \{borra(x) : x \in \Sigma\} \end{aligned}$$

- R es el conjunto de las respuestas que devuelven las operaciones antes mencionadas.

$$\begin{aligned} R = &\{true, false\} \\ &\cup Q \\ &\cup \Sigma \times \mathcal{C} \times \{0, 1\} \end{aligned}$$

- Δ es una función de $Q \times M \to Q \times R$. A continuación se enumeran las reglas que la definen, dónde $\forall q \in Q, \forall x \in \mathcal{C}, a, b \in \Sigma, y \in \{0, 1\}$:

1. $\Delta(q, estado()) = (q, q)$
2. $\Delta(q, lee(x)) = \begin{cases} (q, (a, x, y)), & \exists a, y, (a, x, y) \in q \\ (q, false), & \forall a, y, (a, x, y) \notin q \end{cases}$
3. $\forall a, \Delta(q, agrega(a, x)) = \begin{cases} (q, false), & \exists b, y, (b, x, y) \in q \\ (q \cup \{(a, x, 1)\}, true), & \forall b, y, (b, x, y) \notin q \end{cases}$
4. $\Delta(q, borra(x)) = \begin{cases} (q \setminus \{(a, x, y)\} \cup \{(a, x, 0)\}, true), & \exists a, y, (a, x, y) \in q \\ (q, false), & \forall a, y, (a, x, y) \notin q \end{cases}$

A continuación mostramos un ejemplo del tipo y de sus operaciones. Tomemos $\Sigma = \{a, b, ..., z\}$ y $\mathcal{C} = \mathbb{R}$. En la parte superior de las flechas se encuentra el estado del objeto, la flecha simboliza la ejecución del método que aparece a la derecha, junto con su respuesta. El estado resultante de aplicar la operación se encuentra en la punta de la flecha, junto a él, en azul, está el documento al que representa.

$$Q = \emptyset$$
$$\downarrow \; < agrega(h, 0.3) : true >$$
$$\{(h, 0.3, 1)\} \quad \texttt{h}$$
$$\downarrow \; < agrega(p, 0.5) : true >$$
$$\{(h, 0.3, 1), (p, 0.5, 1)\} \quad \texttt{hp}$$
$$\downarrow \; < agrega(m, 0.6) : true >$$
$$\{(h, 0.3, 1), (p, 0.5, 1)(m, 0.6, 1)\} \quad \texttt{hpm}$$
$$\downarrow \; < agrega(o, 0.7) : true >$$
$$\{(h, 0.3, 1), (p, 0.5, 1), (m, 0.6, 1)(o, 0.7, 1)\} \quad \texttt{hpmo}$$
$$\downarrow \; < borra(0.5) : true >$$
$$\{(h, 0.3, 1), (m, 0.6, 1), (o, 0.7, 1), (p, 0.5, 0)\} \quad \texttt{hmo}$$
$$\downarrow \; < agrega(o, 0.4) : true >$$
$$\{(h, 0.3, 1), (o, 0.4, 1), (m, 0.6, 1), (o, 0.7, 1), (p, 0.5, 0)\} \quad \texttt{homo}$$

Figura 3.1: Objeto de tipo Documento

3.3. Concurrencia

Como se puede ver en la definición de Δ, los métodos que modifican el estado son *borra* y *agrega*. Las llamadas concurrentes de estos métodos pueden causar un estado inconsistente, por ellos analizaremos los casos en que se pueden combinar estas llamadas: $borra(x) - borra(y)$, $agrega(a, x) - borra(y)$ y $agrega(a, x) - agrega(b, y)$.

Cuando las operaciones concurrentes difieren en el identificador, son conmutativas. Es decir, sin importar el orden de ejecución se alcanza el mismo estado, esta es una propiedad de conmutatividad del objeto.

Los casos en los que los identificadores de las llamadas concurrentes son diferentes no causan problemas pues están modificando secciones ajenas.

Los casos más interesantes son aquellos dónde los identificadores son iguales, $x = y$, pues más de un proceso está modificando el mismo átomo. Revisemos cada uno:

1. $borra(x) - borra(x)$

   Observando la *regla 4* de la definición de Δ notamos que el método *borra* es idempotente, pues sin importar cuántas veces se realice, se tiene el mismo resultado que si sólo se borrara una vez. Esto garantiza que llamadas concurrente de borrado sobre un mismo átomo no causen problemas.

2. $agrega(a, x) - borra(x)$

   El resultado esperado sería que primero tomara efecto la inserción y posteriormente el borrado. Pues un proceso no puede borrar un elemento que aún no conoce.

3. $agrega(b, x) - agrega(a, x)$

   Cuando dos procesos insertan un carácter en la misma posición, conciden en el identificador, lo que provoca repetición de identificadores en Q. En la siguiente sección formalizamos el problema de la generación de identificadores, propondremos una solución y demostraremos que funciona.

3.4. Generación de Identificadores

Para evitar un estado inconsistente en el objeto, cuando hay inserciones concurrentes en la misma posición, es necesario que los procesos se sincronicen. Dado que nos interesa la eficiencia en el sistema, utilizaremos una solución no bloqueante. Tampoco recurriremos a primitivas de sincronización poderosas, como *CompareAndSwap,CAS*, pues si bien es libre de espera, demanda muchos recursos computacionales. El enfoque que tomaremos está inspirado en el algoritmo de la *panadería* [22]. A continuación explicamos de manera general la solución que proponemos.

Cuando se desea insertar un átomo tomaremos en cuenta a sus vecinos inmediatos, el de la izquierda y el de la derecha, para que el nuevo identificador dependa de ellos y

conserve la posición de inserción. Al insertar en la frontera del documento sólo hay un vecino −al inicio, sólo el de la derecha, al final, sólo el de la izquierda−, representaremos el límite inferior con $\perp$ y superior con $\top$. El primer carácter que se inserte en el documento no tendrá vecinos. El nuevo identificador también dependerá del proceso que lo está insertando; él marca la diferencia cuando, concurrentemente, se están insertando átomos en una misma posición.

Formalmente, para generar identificadores de átomos, necesitamos una función $f : \mathcal{I} \cup \{\top, \perp\} \times \mathcal{I} \cup \{\top, \perp\} \times \Pi \to \mathcal{I}$ dónde:

- Π: conjunto finito y discreto de los identificadores de procesos.
- $\mathcal{I}$: Es un conjunto infinito y denso con un orden total que está construido a partir de $\mathcal{C}$ (conjunto denso y abierto) y Π. Mas adelante detallo la construcción de este conjunto.
- $\{\top, \perp\}$: Símbolos que representan la frontera del documento. Aunque no están en el conjunto de identificadores, los relacionaremos con ellos de la siguiente manera $\forall x \in \mathcal{I}, \perp < x < \top$.

Para garantizar que no haya repeticiones de identificadores y que siempre hay un identificador sin importar dónde se inserte, f debe cumplir con las siguientes propiedades, $\forall(x, x', y, y' \in \mathcal{I} \cup \{\top, \perp\}), \forall(p, q \in \Pi)$:

1. $x < y \implies x < f(x, y, p) < y$. Siempre se puede encontrar un identificador en medio de cualesquiera dos elementos. $(<)$ representa el orden total en $\mathcal{I} \cup \{\perp, \top\}$, más adelante ahondamos en ello.
2. $p \neq q \implies f(x, y, p) \neq f(x, y, q)$. Si procesos diferentes desean insertar en la misma posición, se podrá encontrar un identificador diferente para cada átomo.
3. $f(x, y, p) = f(x', y', p) \implies x = x', y = y'$. No hay repetición de identificadores en posiciones diferentes.

3.4.1. Función generadora de identificadores

$\mathcal{C}$ es un conjunto abierto −sin mínimo ni máximo−, denso y con un orden total $<_{\mathcal{C}}$, por lo tanto podemos garantizar que existe la siguiente función $g : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ tal que:

$$\forall(x, y, \in \mathcal{C}) : x < y \implies x < g(x, y) < y$$

Esta función también debe ser capaz de encontrar un elemento diferente en $\mathcal{C}$ cuando no hay forzosamente dos elementos de referencia, usaremos también $\{\top, \perp\}$ para indicar la ausencia de una referencia. Haremos una extensión del conjunto $\mathcal{C}$ y de la función g para incluir esos elementos.

Definición 1 (Extensión de $\mathcal{C}$, $\mathcal{C}'$)**.** *Definimos $\mathcal{C}'$ como $\mathcal{C} \cup \{\perp, \top\}$ con un orden total $<_{\mathcal{C}'}$. Decimos que $x <_{\mathcal{C}'} y, x \neq y$, si sucede una de las siguientes condiciones:*

- $x, y \in \mathcal{C}$, $x <_{\mathcal{C}} y$
- $x = \bot$ *o* $y = \top$

Usando el conjunto anterior, se puede definir la función que usaremos para encontrar un nuevo elemento en $\mathcal{C}$, $g' : \mathcal{C}' \times \mathcal{C}' \to \mathcal{C}$. Gracias a que $\mathcal{C}$ es abierto, siempre se puede garantizar que $g'(\bot, x)$ puede regresar un valor más pequeño que x, de igual manera en la frontera superior, $g'(y, \top)$, puede regresar un valor más grande que y.

El conjunto de identificadores que usaremos para nuestros átomos es el siguiente:

Definición 2 (Conjunto de identificadores, $\mathcal{I}$)**.** *Sea $\mathcal{C}$ un conjunto infinito, abierto, denso y con orden total $<_{\mathcal{C}}$, Π un conjunto finito con un orden total $<_{\Pi}$ de identificadores de proceso. Definimos $\mathcal{I}$ como $(\mathcal{C}\Pi)^+$, es decir, cadenas de longitud arbitraria, pero par, intercalando elementos en $\mathcal{C}$ con elementos en $\mathcal{I}$. Una cadena s está en $\mathcal{I}$ si:*

$$s = c_1 p_1 ... c_n p_n \text{ dónde } c_i \in \mathcal{C}, p_i \in \Pi, i \in \{1, ..., n\}$$

Podemos notar que la cadena vacía, ϵ, no está en $\mathcal{I}$, lo cuál no resulta ser un problema, pues no nos interesan los identificadores vacíos.

Representaremos la concatenación con un punto $(\cdot)$ o simplemente juntando las cadenas o caracteres.

Podemos pensar una cadena en $\mathcal{I}$ como una sucesión de elementos en el producto cartesiano $\mathcal{C} \times \Pi$. Como cada uno de esos conjuntos tiene un orden total, el producto cartesiano puede ser ordenado por el orden lexicográfico y por tanto es un orden total. Ahora bien, el orden en $\mathcal{I}$ es también un orden lexicográfico de cadenas pensando como caracteres cada uno de los pares $-$subcadenas formadas por un elemento en $\mathcal{C}$ seguido de otro elemento en $\Pi-$ que la conforman. A continuación lo describimos formalmente.

Definición 3 (Orden en $\mathcal{I}$, $<_{\mathcal{I}}$)**.** *Sean $s, r \in \mathcal{I}; w, z, z' \in \mathcal{I} \cup \{\epsilon\}; c_i, c_i' \in \mathcal{C}; p_i, p_i' \in \Pi$, tales que $s = w c_i p_i z, r = w c_i' p_i' z'$. Observemos que w es un prefijo común de s y r, posiblemente vacío.*

Decimos que $s <_{\mathcal{I}} r$ si sucede una de las siguientes condiciones:

- $c_i = c_i'$ *y* $p_i <_{\Pi} p_i'$
- $c_i <_{\mathcal{C}} c_i'$
- $c_i p_i z = \epsilon$

Como se puede apreciar, se definió el orden para cualesquiera dos elementos en $\mathcal{I}$, por lo tanto, el orden $<_{\mathcal{I}}$ es total.

Como observamos, no se están incluyendo los símbolos que representan la frontera del documento. Como hicimos con $\mathcal{C}$, hacemos una extensión de $\mathcal{I}$ para incluirlos

Definición 4 (Extensión de $\mathcal{I}$, $\mathcal{I}'$)**.** *Definimos $\mathcal{I}'$ como $\mathcal{I} \cup \{\bot, \top\}$ con un orden total $<_{\mathcal{I}'}$. Decimos que $x <_{\mathcal{I}'} y, x \neq y$, si sucede una de las siguientes condiciones:*

- $x, y \in \mathcal{I}$, $x <_{\mathcal{I}} y$
- $x = \bot$
- $y = \top$

Ahora si estamos listos para definir la función generadora de identificadores.

Definición 5 (Función $f : \mathcal{I}' \times \mathcal{I}' \times \Pi \to \mathcal{I}$)**.** *Sean $r, s \in \mathcal{I}'$, $w, z, z' \in \mathcal{I} \cup \{\epsilon\}, c_i, c_i' \in \mathcal{C}', p_i, p_i' \in \Pi$, tales que $s <_{\mathcal{I}'} r; s = wc_ip_iz, r = wc_i'p_i'z'$.*

$$f(s, r, p) = f(wc_ip_iz, wc_i'p_i'z', p) = \begin{cases} s \cdot g'(\bot, \top) \cdot p, & c_i = c_i' \ y \ p_i <_{\Pi} p_i' \\ w \cdot g'(c_i, c_i') \cdot p, & c_i <_{\mathcal{C}'} c_i' \end{cases}$$

El ejemplo al final del capítulo ayuda a ejemplificar mejor los conceptos descritos anteriormente.

Ahora, demostraremos que la definición propuesta de f cumple con las propiedades definidas anteriormente.

Lema 1. *Para todo $x, y \in \mathcal{I}', p \in \Pi$, la definición 5 cumple con:*

$x <_{\mathcal{I}'} y \implies x <_{\mathcal{I}'} f(x, y, p) <_{\mathcal{I}'} y.$

Demostración. Como podemos observar en la definición 5, debemos explorar dos casos: Sean $w, z, z' \in \mathcal{I}$ tales que $x = waqz, y = wbrz', a, b \in \mathcal{C}', q, r \in \Pi$, $q <_{\Pi} r$.

- Caso 1: $a = b, f(x, y, p) = x \cdot g'(\bot, \top) \cdot p$

  Sabemos que:

  $$x = waqz <_{\mathcal{I}'} waqz \cdot g'(\bot, \top) \cdot p = f(x, y, p) <_{\mathcal{I}'} war <_{\mathcal{I}'} warz' = y$$

  Por lo tanto:

  $$x <_{\mathcal{I}'} f(x, y, p) <_{\mathcal{I}'} y$$

- Caso 2: $a <_{\mathcal{C}'} b, f(x,y,p) = w \cdot g'(a,b) \cdot p$

  Sabemos que:

  $$a <_{\mathcal{C}'} g'(a,b) <_{\mathcal{C}'} b$$

  tenemos:

  $$x = waqz <_{\mathcal{J}'} w \cdot g'(a,b) \cdot p = f(x,y,p) <_{\mathcal{J}'} wbr <_{\mathcal{J}'} wbrz' = y$$

  Por lo tanto:

  $$x <_{\mathcal{J}'} f(x,y,p) <_{\mathcal{J}'} y$$

■

Lema 2. *Para todo $x, y \in \mathcal{I}', p \in \Pi$, la definición 5 cumple con:*

$p \neq q \implies f(x,y,p) \neq f(x,y,q)$

Demostración. Esto se puede validar trivialmente, ya que cada proceso, p y q, concatena al final su identificador de proceso. Entonces al menos $f(x,y,p)$ y $f(x,y,q)$ difieren en el último carácter, por lo tanto $f(x,y,p) \neq f(x,y,q)$. ■

Lema 3. *Para todo $x, x', y, y' \in \mathcal{I}', p \in \Pi$, la definición 5 cumple con:*

$f(x,y,p) = f(x',y',p) \implies x = x', y = y'.$

Demostración. Podemos observar que hay tres casos distintos, cuando x, y y x', y' definen dos intervalos ajenos, cuando comparten el extremo izquierdo y cuando comparten el extremo derecho.

- Caso 1 (Por contradicción): Supongamos que $x \neq x', y \neq y'$ y s.p.g $x <_{\mathcal{J}'} y <_{\mathcal{J}'} x' <_{\mathcal{J}'} y'$. No puede pasar que $x' <_{\mathcal{J}'} y'$ pues una posición está definida entre dos identificadores contiguos, en ese caso serían x y x', pues x' aparece antes que y.

  Sabemos que:

  $$f(x,y,p) <_{\mathcal{J}'} y \implies f(x',y',p) <_{\mathcal{J}'} y \implies x' <_{\mathcal{J}'} f(x',y',p) <_{\mathcal{J}'} y \implies x' <_{\mathcal{J}'} y$$

  Llegamos a una contradicción, $x' <_{\mathcal{J}'} y$, que vino de suponer que $x \neq x'$ y $y \neq y'$, por tanto $x = x'$ y $y = y'$.

- Caso 2: (Por contradicción): Supongamos que $x = x', y \neq y'$ y s.p.g $x <_{\mathcal{I}'} y <_{\mathcal{I}'} y'$ Primero se tuvo que haber calculado $f(x, y', p)$, ya que de lo contrario, x y y' no estarían definiendo una posición, pues estaría y en medio. No puede pasar que $f(x, y', p) <_{\mathcal{I}'} y$ porque $f(x, y, p)$ no estaría definiendo una posición válida, pues entre x y y estaría $f(x, y, p)$, por lo tanto tenemos que:

$$x <_{\mathcal{I}'} f(x, y, p) <_{\mathcal{I}'} y <_{\mathcal{I}'} f(x, y', p) <_{\mathcal{I}'} y'$$

  Pero como $f(x, y, p) = f(x, y', p)$, llegamos a una contradicción pues y está entre dos valores iguales. La contradicción vino de suponer que $y \neq y'$. Por tanto $y = y'$.

- Caso 3: es simétrico al caso 2.

■

Teorema 1. *f, definida en la definición 5, cumple con ser una función generadora de identificadores.*

Demostración. Esto se puede verificar con los lemas anteriores: lema 1, lema 2 y lema 3. ■

Para clarificar los conceptos antes definidos, mostramos un ejemplo:

`Definimos los elementos que se necesitan:`

- $\Pi = \{p_1, p_2, p_3\}$
- $\mathcal{C} = (0, 1) \subset \mathbb{R}$
- $\mathcal{C}' = \mathcal{C} \cup \{\bot, \top\}$
- $\mathcal{I} = (\mathcal{C}\Pi)^+$, `representaremos los elementos de` $\mathcal{I}$ `como listas de elementos, por ejemplo` $[0.5, p_3]$.
- $\mathcal{I}' = \mathcal{I} \cup \{\bot, \top\}$

- $g'(x,y) = \begin{cases} 0.5, & x = \bot; y = \top \\ y/2, & x = \bot; y \in \mathcal{C} \\ x + (1-x)/2, & x \in \mathcal{C}; y = \top \\ (x+y)/2 & x, y \in \mathcal{C} \end{cases}$

Iniciamos con el documento vacío. En él dos procesos, de manera concurrente, realizan una inserción.

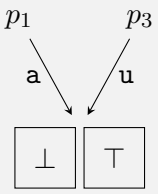

Figura 3.2: Inserciones concurrentes en documento vacío

La función f genera dos identificadores usando la segunda regla de su definición, dónde $w = \epsilon$ y $c_i = \bot, c'_i = \top$:

- $f(\bot, \top, p_1) = \epsilon \cdot g'(\bot, \top) \cdot p_1 = [0.5, p1]$
- $f(\bot, \top, p_3) = \epsilon \cdot g'(\bot, \top) \cdot p_3 = [0.5, p3]$

Como podemos observar, aún cuando los proceso insertan de manera concurrente en el mismo lugar, f genera identificadores diferentes.

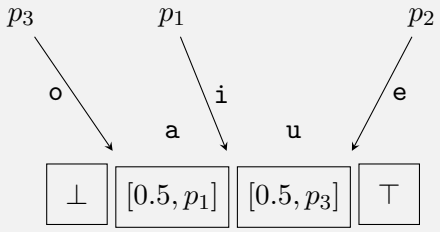

Figura 3.3: Inserciones concurrentes en diferentes lugares

La función f genera tres identificadores:

- $f(\bot, [0.5, p_1], p_3) = \epsilon \cdot g'(\bot, 0.5) \cdot p_3 = [0.25, p3]$
- $f([0.5, p_1], [0.5, p_3], p_1) = [0.5, p_1] \cdot g'(\bot, \top) \cdot p_1 = [0.5, p_1, 0.5, p_1]$
- $f([0.5, p_1], \top, p_2) = \epsilon \cdot g'(0.5, \top) \cdot p_2 = [0.75, p2]$

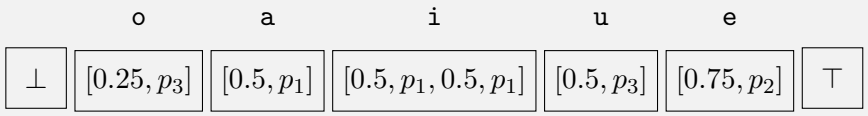

Figura 3.4: Documento Colaborativo

Como podemos observar, rápidamente crecen las cadenas de los identificadores. La implementación debe tener en cuenta este hecho. Existen trabajos en los que se proponen maneras diferentes de crear conjuntos de identificadores conservando las propiedades de densidad y cuidando no sobrepasar las limitaciones de espacio [28].

Capítulo 4

Implementación en Memoria Compartida

A partir de la definición del objeto de tipo *documento colaborativo*, proponemos una implementación linealizable y libre de espera en memoria compartida de ese objeto. Primero presentaremos el algoritmo de la implementación y posteriormente las demostraciones de sus propiedades de progreso y corrección.

4.1. Implementación: Algoritmo y Estructuras de Datos

La implementación está basada en un *Snapshot Atómico*. A cada proceso p_i del conjunto Π se le asigna la entrada *i-ésima*, dónde escribe las modificaciones que ha aplicado al documento. El estado del documento está guardado en el *Snapshot* y es la unión de las todas las modificaciones hechas por los procesos. Además, cada proceso guarda localmente una variable persistente (prevalece entre cada invocación) en la que lleva cuenta de sus modificaciones. Se puede garantizar que la variable local de cada proceso es consistente con su entrada en el *Snapshot*, pues sólo el puede modificar ambas. Recordemos que las modificaciones son tuplas en $\Sigma \times \mathcal{I} \times \{0, 1\}$. $\mathcal{I}$ es el conjunto de cadenas que se abordó en el capítulo anterior.

Antes de presentar la implementación, describiremos las siguientes funciones auxiliares de consulta sobre la captura S del *Snapshot*:

- `actual(id,S)`: devuelve la tupla encontrada en S que refleja el estado del átomo con identificador *id*.
- `existe(id,S)`: regresa `true` si el identificador *id* está presente en S. En otro caso `false`.
- `ids(S)`: obtiene de S el conjunto de todos los identificadores. Es importante notar que este conjunto es finito, pues hay una cantidad finita de átomos en el documento.

Los métodos que constituyen el algoritmo son los siguientes:

- Para leer un identificador, el proceso realiza una captura del *Snapshot*, busca el identificador en la captura, si lo encuentra regresa la tupla correspondiente, en caso contrario `false`.
- Cuando un proceso desea agregar un carácter con su identificador, primero se asegura que el identificador no esté presente, posteriormente agrega la tupla de la modificación a su copia local y finalmente hace la actualización sobre el *Snapshot*. Si el identificador ya está presente, el método termina regresando `false`.
- Para borrar un átomo, el proceso busca el identificador en una captura del *Snapshot*, si lo encuentra, integra en su copia local de las modificaciones la tupla de borrado y luego actualiza su entrada en el *Snapshot*. En caso de que no encuentre el identificador, termina y regresa `false`.

- El estado del documento consta de las tuplas que reflejan cada átomo con su modificación más reciente.

A continuación se muestra el código del algoritmo:

Algoritmo 2: Implementación en memoria compartida del objeto documento colaborativo. El código es ejecutado por cada proceso p.

Variables:
Variables compartidas:
AS, Atomic Snapshot, inicialmente cada entrada con $\emptyset$.
Variables locales persistentes:
$mod \subset \Sigma \times \mathcal{I} \times \{0,1\}$, inicialmente $\emptyset$.

```
1  Function lee(x):
2      S = AS.scan()
3      if existe(x,S) then
4          return actual(x,S)
5      return false
6  Function agrega(a,x):
7      S = AS.scan()
8      if existe(x,S) then
9          return false
10     mod = mod ∪ {(a, x, 1)}
11     AS.update_p(mod)
12     return true
13 Function borra(x):
14     S = AS.scan()
15     if existe(x,S) then
16         a = elemento(x,S)
17         mod = mod ∪ {(a, x, 0)}
18         AS.update_p(mod)
19         return true
20     return false
21 Function estado():
22     S = AS.scan()
23     IDS = ids(X)
24     E = {}
25     foreach id in IDS do  E = E ∪ {actual(id,S)}
26     return E
```

4.1.1. Generación de identificadores

En el algoritmo 2 no aparece la manera en la que se generan los identificadores. La función de insertar ya obtiene el identificador nuevo. Esto es porque hay un paso previo, en el que el editor solicita la inserción en una posición determinada y se calcula cuál es el nuevo identificador, para posteriormente mandar los parámetros completos a `agrega(a,x)`. Pero en la sección anterior fue relevante abordar ampliamente este tema para garantizar que sí hay un conjunto de identificadores como el que se necesita para soportar concurrencia y densidad.

4.2. Demostraciones de las propiedades de progreso y corrección

Es importante validar que la especificación secuencial del documento se respete y también que se asegure que los procesos no se quedarán esperando indefinidamente por una respuesta cuando invoquen una operación. A la implementación propuesta le pediremos *linealizabilidad* [21] como propiedad de corrección y ser *libre de espera* [22] como propiedad de progreso.

Lema 4. *El Algoritmo 2 es libre de espera.*

Demostración. Primero observemos que hay una implementación *libre de espera* del *Snapshot* atómico [1]. Posteriormente notemos que las operaciones del algoritmo ejecutan un número finito de instrucciones, incluso el ciclo de la línea 25, pues IDS es un conjunto finito.

A partir de las dos observaciones anteriores podemos concluir que todas las operaciones invocadas por un proceso terminan un número finito de instrucciones, sin importar la velocidad de otros procesos, por lo tanto el algoritmo es *libre de espera.* ■

Lema 5. *El Algoritmo 2 es una implementación linealizable del objeto de tipo documento colaborativo.*

Demostración. Para demostrar que el algoritmo 1 correctamente implementa el objeto de tipo documento, demostraremos que la implementación es linealizable.

Sea H una historia asociada a una ejecución finita, E, del algoritmo 1, a partir de la cuál obtendremos $\bar{H}$, la compleción de H, de la siguiente manera:

- Si en H hay invocaciones pendientes de lectura (*estado* y *lee*), en $\bar{H}$ ya no aparecen esas invocaciones.

- Si en H hay invocaciones pendientes de modificación (*borra* y *agrega*), pero que ya tomaron efecto en el *Snapshot*, es decir, completaron la instrucción $AS.update_p$ de su código. Entonces en $\bar{H}$ se agrega $true$ como respuesta a esas invocaciones pendientes.

- El resto de las invocaciones pendientes de *borra* y *agrega* en H, las que aun no toman efecto en el *Snapshot*, ya no se integran a $\bar{H}$.

Ya teniendo una historia completa, $\bar{H}$, se obtendrá la historia secuencial L con los siguientes puntos de linealización:

- Si la operación es *lee* o *estado*, sus puntos de linealización están en las líneas 2 y 22, respectivamente y corresponden con el punto de linealización de $AS.scan$.

- Si la operación es *agrega* o *borra* y fue fallida, su puntos de linealización están en las líneas 7 y 14, respectivamente y corresponde con el punto de linealización de $AS.scan$.

- Si la operación es *agrega* o *borra* y fue exitosa, sus puntos de linealización están en las línea 11 y 18, respectivamente, y corresponden con el puntos de linealización de $AS.update_p$.

A continuación demostraremos que L es una historia secuencial legal, es decir que respeta las reglas definidas por Δ. La prueba se hará por inducción sobre el tamaño de los prefijos de L.

Sea L_k el prefijo de L de tamaño k, las primeras k invocaciones con su respuestas (como L es una historia secuencial, ya hay un orden total y se sabe exactamente cuáles son las k primeras invocaciones). Cuando k es 0, L_k es el prefijo vacío y cumple con Δ por vacuidad. Supongamos que L_m es legal, demostremos que L_{m+1} es legal analizando los casos de inv, la *m+1-ésima* invocación. Por hipótesis de inducción L_m es una historia

legal y la captura S refleja fielmente los cambios hechos anteriormente, por tanto los resultados de las funciones `existe(x,S)`, `actual(x,S)` e `ids(S)` serán correctos.

- *inv* es $lee(x)$. No modifica el *Snapshot*, por lo tanto tampoco el estado del objeto y su respuesta es correcta pues utiliza funciones correctas, `existe` y `actual`, para calcularla. Primero busca el identificador, si no lo encuentra regresa $false$, si lo encuentra regresa la tupla correspondiente del átomo x.

- *inv* es *estado*(). No modifica el *Snapshot*, por lo tanto tampoco el estado del objeto y su respuesta es correcta pues utiliza funciones correctas, `ids` y `actual` para calcularla. Primero obtiene todos los identificadores de los átomos y posteriormente la tupla actual de cada uno de ellos, formando el estado del objeto

- *inv* es $borra(x)$ exitoso. Hay una alteración al *Snapshot*, por tanto al estado del objeto, se agrega una nueva tupla $(a, x, 0)$ que refleja la nueva modificación, es decir, a partir de ese punto, cualquier nueva captura S' proveerá que `actual(x,S')` es $(a, x, 0)$. Fue correcto hacer esa modificación pues primero se aseguró la existencia del átomo, como pide Δ.

- *inv* es $borra(x)$ fallido. No se realiza ninguna modificación al *Snapshot*, lo cuál es correcto pues no existe el identificador x.

- *inv* es $agrega(a, x)$ exitoso. Se modifica el *Snapshot* agregando $(a, x, 1)$, esto es correcto pues antes se determinó que el identificador x no estaba presente y también es importante notar que x no fue insertado por otro proceso en el ínter, pues sería diferente (propiedad de los identificadores: procesos diferentes generan diferentes identificadores).

- *inv* es $agrega(a, x)$ fallido. No se realizan modificaciones al *Snapshot*, lo cual es correcto según Δ, pues primero se verificó que el átomo x ya estaba presente.

Por lo tanto podemos verificar que L es una historia secuencial legal.

Ahora demostremos que para todo proceso $p \in \Pi$, $\bar{H}|_p = L|_p$. De manera similar a la demostración anterior, utilizaremos inducción sobre el tamaño de los prefijos de $\bar{H}|_p$

y demostraremos que cada uno de ellos es igual al prefijo del mismo tamaño de $L|_p$. Denotaremos a $\bar{H}|_p(i)$ como el prefijo de tamaño i de $\bar{H}|_p$, las primeras i llamadas a método de $\bar{H}|_p$. Usaremos una notación equivalente para los prefijos de $L|_p$. El prefijo vacío cumple con la igualdad $\bar{H}|_p(0) = \emptyset = L|_p(0)$. Supongamos que $\bar{H}|_p(i) = L|_p(i)$, demostremos que $\bar{H}|_p(i+1) = L|_p(i+1)$. Por hipótesis de inducción sabemos que las primeras i llamadas a método son idénticas en $\bar{H}|_p$ y en $L|_p$, ahora demostraremos que la llamada a método $i+1$ también es igual en ambas. Recordemos que cada proceso ejecuta una llamada a método una vez que concluyó la anterior, por lo tanto $H|_p$ es una historia secuencial con un orden total. Como los puntos de linealización propuestos están dentro del intervalo definido por la invocación y respuesta de la llamada a método no puede suceder que se altere el orden total antes mencionado. Esto nos lleva a concluir que la llamada a método en la posición *i+1* es la misma en ambas historias secuenciales, en consecuencia $\bar{H}|_p(i+1) = L|_p(i+1)$ y por tanto $\bar{H}|_p = L|_p$.

Finalmente demostremos que $\prec_H \subseteq \prec_L$, es decir, L respeta el orden de *tiempo-real*. Para algunas llamadas a método m_i y m_j, tenemos que $m_i \prec_H m_j$. Supongamos que no pasa que $m_i \prec_L mj$, pero como $\prec_L$ es un orden total y las llamadas son diferentes, sí pasa que $m_j \prec_L mi$. Pero dado que los puntos de linealización que definen L se ubican dentro del intervalo de invocación-respuesta en H, lo anterior significaría que m_i y m_j se traslapan o que m_j sucedió primero que m_i en H, lo cuál es una contradicción. Por ello podemos concluir que $m_i \prec_L mj$. Entonces $\prec_H \subseteq \prec_L$.

A partir de las demostraciones anteriores, ya se pude concluir que el algoritmo 1 es una implementación linealizable del objeto de tipo documento, y por tanto correcta. ■

Teorema 2. *El Algoritmo 2 es una implementación correcta, linealizable y libre de espera, del objeto documento colaborativo.*

Demostración. Esto se sigue de los lema 4 y 5. ■

Capítulo 5

Implementaciones en Paso de mensajes

Cuando presentamos la edición colaborativa, mencionamos que su propósito es habilitar la modificación de un mismo documento desde diferentes dispositivos conectados a una red. Esta red puede ser tan amplia como se desee, el único requisito es que, par a par, todos los dispositivos estén conectados. Hasta este momento, sólo hemos proporcionado una implementación en el modelo de memoria compartida. El objetivo de este capítulo es exponer una gama de implementaciones en el modelo de paso de mensajes, que es dónde la edición colaborativa cobra mayor sentido. Por claridad, las fallas bizantinas se omiten, pero los algoritmos se pueden modificar para considerar ese caso. La gama está compuesta por tres algoritmos que cumplen diferentes compromisos de eficiencia y consistencia. Podremos observar que los más eficientes imponen menos restricciones al incluir operaciones remotas en las réplicas locales de cada proceso, consecuentemente su documento se puede actualizar de formas caprichosas. De manera inversa se relacionan la consistencia y la eficiencia: los algoritmos que fuerzan mayor consistencia en los estados de las réplicas, necesitan mandar mensajes más grandes, por lo que son menos eficientes.

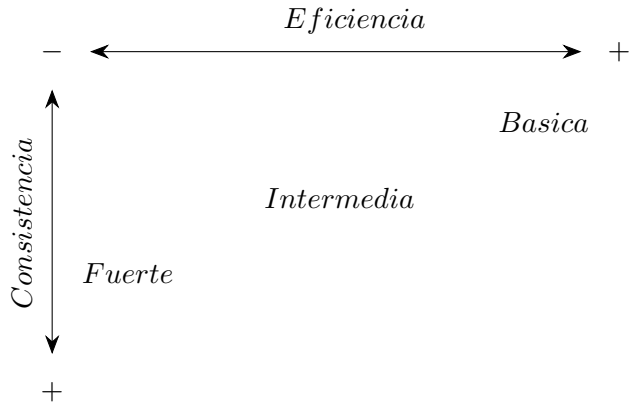

Figura 5.1: Compromisos de consistencia y eficacia de las implementaciones.

Las implementaciones se presentan en orden ascendente de compromiso de consistencia. La primera implementación, *básica*, está basada en el algoritmo 2 expuesto en el capítulo 4. Los mensajes que se mandan únicamente contienen la operación que realiza cada proceso y su procesamiento en las réplicas remotas es no determinista. La implementación *intermedia*, la obtuvimos modificando el algoritmo 2 y usando la idea del artículo *Money Transfer Made Simple* [4], de enriquecer los mensajes con un número de secuencia propio de cada proceso, de modo que se pueden incluir las modificaciones remotas de manera ordenada por proceso. La implementación *fuerte*, ocupa la propuesta del artículo [17] de agregar dependencias a cada modificación. En su caso las dependencias corresponden a un conjunto de transferencias entrantes previas, pero en nuestro caso usaremos los número de secuencia conocidos por el proceso cuando realizó la modificación. Este conocimiento estará codificado en un vector, y para integrar los mensajes usaremos una versión ajustada del algoritmo de los relojes vectoriales [3].

Antes de presentar las implementaciones describiremos el modelo de paso de mensajes.

5.1. Modelo

En el *modelo de paso de mensajes* hay $\Pi = \{p_1, p_2, ..., p_n\}$ procesos secuenciales y asíncronos. Cada uno de esos procesos pueden fallar deteniendo su ejecución prematuramente sin posibilidad de retomarla. A los procesos que fallan, le llamaremos *fallidos*. Si no fallan, les llamaremos *correctos*. Los procesos se comunican mediante el intercambio de mensajes en una red asíncrona, es decir, que tiene un retraso finito, pero arbitrario. Existen múltiples protocolos o primitivas de comunicación en este modelo, nosotros nos centraremos en el *Broadcast*. Es importante destacar que se espera que menos de la mitad de los procesos fallen, de lo contrario no se puede tener una implementación de las primitivas del *Broadcast*. Mas información de las restricciones y características del modelo pueden consultarse en [3, 23].

5.1.1. Broadcast Confiable Regular

En una red, el protocolo de *Broadcast* disemina un mensaje emitido por un proceso a todos los procesos de la red, incluido él mismo. La *API* del protocolo está conformado por:

1. *broadcast(m)*. Función que se encarga de mandar el mensaje m a todos los procesos de red.

2. *deliver(p,m)*. Evento que indica la recepción de un mensaje m enviado por el proceso p.

Existen muchos protocolos de *Broadcast* que varían en cuanto a las propiedades que ofrecen. El protocolo particular que usaremos para nuestras implementaciones es el *Broadcast Confiable Regular*. Este protocolo de comunicación cuenta con las siguientes propiedades: [9]

1. **Validez**. Si un proceso correcto p hace broadcast de un mensaje m, entonces eventualmente recibe m .

2. **No duplicación**. Ningún mensaje es entregado más de una vez para el mismo proceso.

3. **No creación**. Si un proceso recibe un mensaje m con emisor s, entonces m fue emitido por el proceso s.

4. **Acuerdo**. Si un mensaje m es entregado a algún proceso correcto, entonces m será entregado a todos los procesos correctos.

5.1.2. Ejecución de los algoritmos

A continuación explicaremos cómo se deben entender los algoritmos y su ejecución en este modelo de paso de mensajes. Pensemos que tenemos un escenario con 3 procesos, cada uno con una copia local en la que incluye las modificaciones que le solicita su cliente, y también las modificaciones remotas de los otros dos procesos.

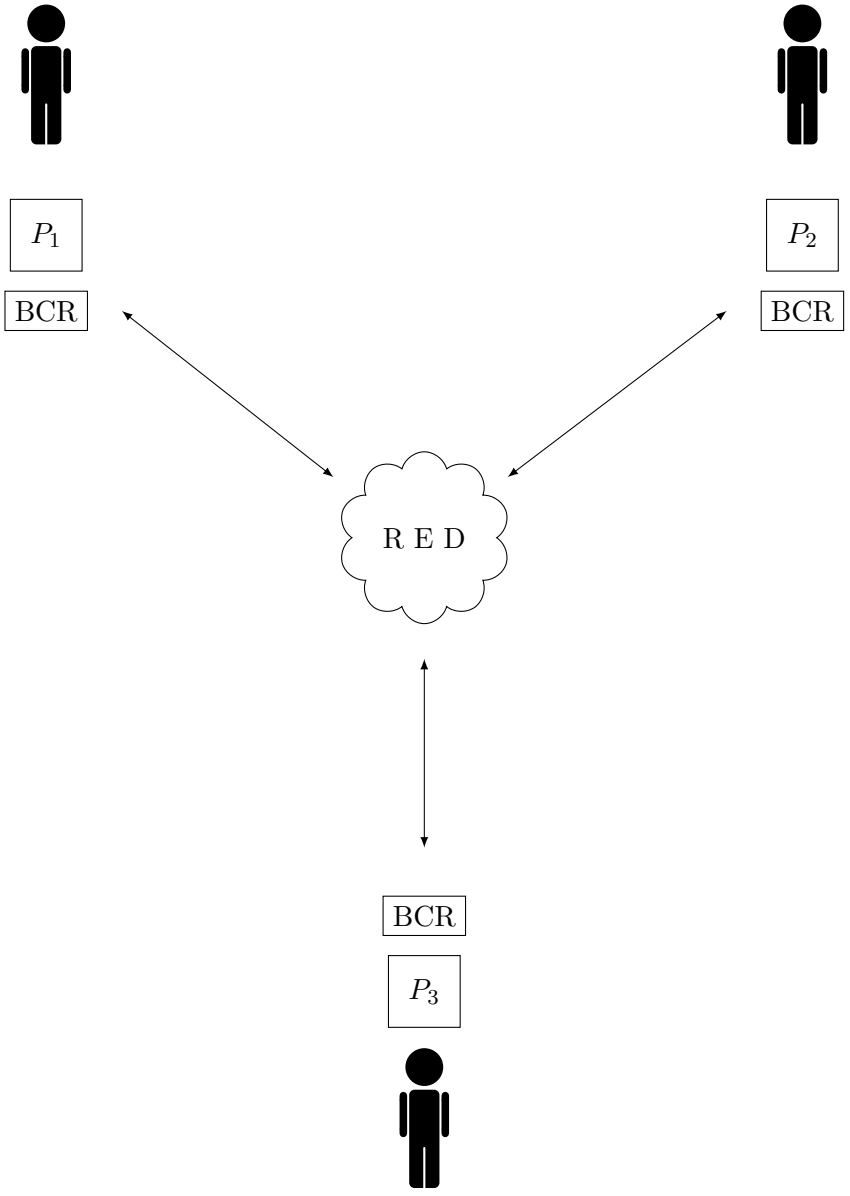

Figura 5.2: Modelo de paso de mensajes

Los mensajes en cada réplica son enviados por el *Broadcast Confiable Regular (BCR)*, a través de la red y también son recibidos por ese mismo protocolo. Ahora nos enfocaremos en lo que sucede en cada réplica.

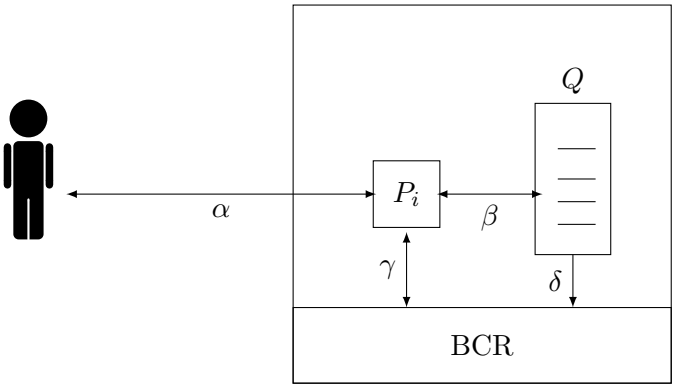

Figura 5.3: Proceso p_i del modelo de paso de mensajes

Como se puede observar en la imagen, el proceso P_i recibe peticiones del cliente y emite respuestas, representadas por α. Las interacciones sobre el estado de la réplica, Q, son realizadas únicamente por P_i; en la imagen: β. Con respecto a los mensajes que llegan a través del BCR, se almacenan en un buffer hasta que P_i los pueda procesar, cuando ese momento llega, procesos independientes se encargan de enviar notifiaciones a P_i, γ, para avisarle que ya alguna condición se ha cumplido y es requerida su acción. δ representa las modificaciones a Q que se dispersan por la red a través de mensajes enviados por el BCR. Ciertamente la invocación de la primitiva de *broadcast* es hecha por P_i, pero lógicamente el origen es Q.

El siguiente pseudocódigo corresponde a un equeleto de los algoritmos que presentaremos en las siguientes secciones:

Algoritmo 3: Psudo código de algoritmo en paso de mensajes

Variables Persistentes:
Q, Estado local

1 **Función** *a(x)*:
2 Código

3 **Upon** *(Condición o Evento)*:
4 Código

La función *a()* corresponde a las acciones que el cliente puede pedir al proceso P_i, como escribir un carácter o leer el estado. La palabra reservada `Upon` indica que se ha cumplido una condición o se ha escuchado un evento, por ejemplo, ha llegado un mensaje remoto (evento *deliver*), y se requiere que el proceso P_i realice alguna acción. Podemos pensarlo como que existen otros procesos que se encargan de verificar cuando

se cumplen las condiciones del `Upon` y notifican a P_i. A su vez, P_i espera a terminar la ejecución de su método actual para atender esas notificaciones. Es importante notar que el único proceso que accede al estado es P_i y que su ejecución es secuencial.

5.2. Implementación básica

Esta implementación es la consecuencia natural del trabajo que hemos realizado hasta ahora. En los capítulos anteriores abordamos el problema de la edición colaborativa desde una perspectiva teórica y planteamos una solución en memoria compartida, 2, de la cuál surge esta propuesta, tomando en cuenta los siguientes puntos:

- En la implementación 2, la comunicación se da mediante las operaciones del *Snapshot atómico*. Los procesos utilizan *update* para anunciar a los demás cuál fue el cambio que realizaron. Todos los procesos tienen acceso a las modificaciones realizadas mediante la operación *scan*.
- En el modelo de paso de mensajes, los proceso no cuentan con una estructura de memoria compartida a través de la cuál comunicarse, por lo que basaremos la comunicación en el *Broadcast Confiable* y en un conjunto local que contenga las modificaciones realizadas, tanto locales como remotas.
- Cuando un proceso realice una modificación, será incluida en su conjunto local, propagada a los demás mediante la operación de *broadcast*, e integrada al conjunto de cada proceso a través del código que acompaña a la primitiva *deliver*.
- Cuando un proceso desee realizar una lectura del documento, la realizará sobre su conjunto local.

5.2.1. Corrección

Para evaluar si la implementación es funcional, primero debemos definir un criterio de corrección. En este caso, nos basaremos en trabajos anteriores de edición colaborativa con enfoque en *CRDT* [27]. Intuitivamente lo que sugieren es que todas las réplicas eventualmente tengan el mismo estado si recibieron el mismo conjunto de modificaciones despreciando el orden. A esta propiedad se le conoce como *Consistencia Eventual Fuerte*:

Definición 6. *Una implementación tiene Consistencia Eventual Fuerte (CEF), si cumple con los siguientes incisos:*

1. Entrega eventual. *Una actualización aplicada a una réplica correcta es eventualmente aplicada a todas las réplicas correctas.*

2. Convergencia. *Si el mismo conjunto de actualizaciones se ha aplicado a dos réplicas (posiblemente en diferente orden), entonces las dos réplicas tiene el mismo estado.*

3. Terminación. *Todas las ejecuciones de métodos terminan.*

5.2.2. Algoritmo

Como mencionamos anteriormente, este algoritmo toma como base la implementación 2, por lo que sólo explicaremos los puntos que se ajustaron para adaptarlo al modelo de paso de mensajes, es decir, las operaciones de *agrega* y *borra*; las lectura son prácticamente las mismas.

Cada proceso cuenta con una variable persistente, Q, es decir, que prevalece entre cada ejecución de función. En ella está el estado local objeto; las modificaciones que ha visto cada proceso. Cuando un proceso inserta o borra un carácter, modifica su copia local, además, crea un mensaje m con la información de su cambio, una tripleta en $\Sigma \times \mathbb{J} \times \{0, 1\}$, y usa *broadcast(m)* para publicarlo a los demás procesos.

Cuando un proceso recibe un mensaje mediante la primitiva *deliver* del *Broadcast Confiable Regular*, lo integra a Q, si la fuente no es él. De otro modo, ya lo había integrado.

A continuación mostramos el algoritmo correspondiente a la implementación.

Algoritmo 4: Implementación en paso de mensajes del objeto documento colaborativo. El código es ejecutado por cada proceso p.

Variables Persistentes:
$Q, \subset \Sigma \times \mathcal{I} \times \{0,1\}$, inicialmente $\emptyset$.

```
1  Función lee(x):
2      if existe(x,Q) then
3          return actual(x,Q)
4      return false
5  Función agrega(a,x):
6      if existe(x,Q) then
7          return false
8      Q = Q ∪ {(a, x, 1)}
9      m = (a, x, 1)
10     broadcast(m)
11     return true
12 Función borra(x):
13     if existe(x,Q) then
14         a = elemento(x,Q)
15         Q = Q ∪ {(a, x, 0)}
16         m = (a, x, 0)
17         broadcast(m)
18         return true
19     return false
20 Función estado():
21     IDS = ids(Q)
22     E = {}
23     foreach id in IDS do  E = E ∪ {actual(id,Q)}
24     return E
25 Upon (deliver(m,q)
26           and q != p ):
27     Q = Q ∪ {m}
```

En los siguientes párrafos demostramos la corrección del algoritmo de acuerdo a la definición que dimos.

Teorema 3. *El algoritmo 4 cumple con la propiedad de* Consistencia Eventual Fuerte.

Demostración. Cuando un proceso, p, invoca el método de *agrega* o *borra*, suceden dos cosas: primero el proceso integra en su documento local la modificación pertinente,

posteriormente difunde un mensaje m, cuyo contenido es la modificación que acaba de realizar. Por las propiedades de *validez* y *acuerdo* del *BCR* se garantiza que todos los demás procesos recibirán el mensaje enviado por p y lo integrarán a su estado, línea 27.

Dado que el estado que mantiene cada réplica es un conjunto, podemos fácilmente verificar que si se han aplicado las mismas modificaciones, sin importar el orden, a diferentes réplicas, entonces tienen el mismo estado. Por lo tanto se cumple la *convergencia*.

Observemos que todas las operaciones del algoritmo ejecutan un número finito de instrucciones. Incluso *estado* que tiene un ciclo, pues itera sobre el conjunto de identificadores y como el documento es finito, se garantiza que hay un número finito de iteraciones y que el ciclo termina. Por lo tanto el algoritmo cumple con *Terminación*.

Con base en los párrafos anteriores podemos concluir que el algoritmo 4 cumple con la propiedad de *Consistencia Eventual Fuerte*. ■

5.2.3. Acotaciones

Dos puntos caracterizan al algoritmo anterior:

1. Cuando recibe un mensaje remoto, salvo una simple validación, lo integra a su estado.
2. Los mensajes son muy pequeños, pues contienen la información mínima que representa la modificación. No se incluye ningún tipo de contexto del emisor.

Como consecuencia de los puntos anteriores, la implementación es eficiente en cuanto al tamaño de los mensajes y a su integración en los procesos remotos. Sin embargo esto provoca estados locales sin ninguna secuencia, que llegarán a ser consistentes, pero en el ínter parece que divergen. Pensemos en el escenario donde un proceso p agrega secuencialmente los siguientes caracteres:

```
hola
```

La *Consistencia Eventual Fuerte* garantiza que cualquier otro proceso q va a recibir los mensajes de p. Supongamos que q realiza 4 lecturas y lo que observó, fue:

```
a
la
hla
```

```
hola
```

Dependiendo de la aplicación en la que se utilice este algoritmo, podría ser irrelevante que exista algún orden en el que se integran los mensajes a las réplicas remotas. En las siguientes implementaciones expondremos mayor información en los mensajes, de modo que las réplicas remotas puedan integrar las actualizaciones con algún orden.

5.3. Implementación intermedia

El propósito de esta versión es mejorar el orden con el que se integran las actualizaciones remotas. Para ello exploramos la solución que proponen en el artículo que ha servido como guía en este trabajo, [17]. Sin embargo esa solución resulta sofisticada para este siguiente paso, pero más adelante la retomaremos. Un año después de [17] se publicó el artículo *Money Transfer made simple* [4], que garantiza una solución para el problema de transferencia de dinero más simple que la anterior. Cuando decimos simple, nos referimos a que la información que acompaña a los mensajes con sus modificaciones, es constante y breve. Por lo tanto, también las validaciones para integrar las modificaciones son ligeras.

Brevemente explicaremos la solución que se propone en [4] y posteriormente expondremos como la adaptamos a nuestro caso.

En [4] se logra resolver el problema del doble gasto en la transferencia de dinero descentralizada entre n procesos, meramente con la imposición de un orden *FIFO* entre cada par de procesos. A grandes rasgos, su implementación consta de los siguientes aspectos:

1. Cada proceso, p_i, enumera las transferencias que realiza. En una variable local, sn_i, se mantiene el número de secuencia de las transferencias realizadas.

2. Cuando un proceso, p_i, realiza una transferencia, manda un mensaje con la información de la misma, pero agrega el número de secuencia que le corresponde a esa transferencia, es decir, el valor de sn_i.

3. Para mantener el control de causalidad que asegura que la historia de transferencias de un proceso sea legal, los procesos que reciben las transferencias remotas, actualizan un arreglo local, $del_i[1, .., n]$, en el que plasman el número de secuencia de la última transferencia que recibieron y aplicaron de cada proceso.

4. Un proceso sólo aplica una transferencia remota si es legal (hay recursos suficientes en la cuenta) y su número de serie coincide con la siguiente trasferencia que se espera de ese proceso. En otro caso se queda pendiente esa transferencia hasta que se cumplan esos requisitos.

5. El intercambio de mensajes se hace mediante unas variantes del *Broadcast Confiable*.

En las siguientes secciones se mostrará a detalle como se integra la solución anteriormente descrita al problema de la edición colaborativa.

5.3.1. Corrección

Tomaremos la propiedad de *Consistencia Eventual Fuerte* como el requisito mínimo de corrección de nuestras implementaciones y ahora agregaremos que, a la vista de todo proceso, se respete el orden en el que los demás procesos realizaron cada una sus modificaciones sobre el documento. De modo que, aún si las modificaciones alcanzan a llegar a los procesos remotos, estos no las van a integrar si no es *su turno*.

Antes de proceder con la definición, daremos el siguiente concepto que utilizaremos más adelante:

Definición 7 (A_{pM})**.** *Sea una ejecución E de un algoritmo y su historia asociada H, $A_{pM}(H)$ es la historia resultante:*

- *al eliminar todas las lecturas que no sean del proceso p. Dicho de otro modo: de los procesos que no son p, sólo conservaremos las operaciones que modifican el estado.*
- *y cuidando que $A_{pM}(H)|p = H|p$. Es decir, respetamos el orden de programa del proceso p.*

Definición 8. *Una implementación del objeto de edición colaborativa cumple la propiedad de* Consistencia Eventual Fuerte *y además, para E, una ejecución finita con historia asociada H, existe una historia secuencial legal S_p para cualquier p, tal que: $S_p|q = A_{pM}(H)|q$ para todo q.*

En palabras simples, lo que dice la definición anteriores es que la implementación debe tener *Consistencia Eventual Fuerte* y cada proceso aplica las transformaciones remotas en el orden que fueron hechas por sus procesos (orden FIFO).

5.3.2. Algoritmo

Siguiendo la idea de [4], cada proceso, p, cuenta con 3 variables persistentes:

1. Q, el conjunto del que ya habíamos hablado en la implementación anterior. Almancena localmente el estado del documento. En el artículo referido, este es un vector que almacena el dinero que tiene cada cuenta.

2. $seq[i]$, es un arreglo de enteros, cuya entrada i almacena el número de secuencia siguiente que se espera del proceso i.

3. sn, es un entero que guarda el número de secuencia de las modificaciones hechas por p.

Las lecturas no afectan el número de secuencia, pues se realizan de manera local y de la misma manera que en la implementación anterior. Cuando el proceso p invoca $agrega(a, x)$, además de la validación pertinente y la modificación de su estado local, el proceso crea un mensaje, m, con la información de la modificación y el número de secuencia que le toca a esa modificación, sn. Para conservar la semántica de sn, se incrementa en 1 su valor. Finalmente, difunde el mensaje con ayuda del *Broadcast Confiable* e indica que la operación fue exitosa. El caso de la eliminación de un carácter es análogo.

Dado que los procesos difunden su modificación con información extra, ahora es posible tomar en cuenta esa información al momento de integrar las modificaciones remotas. Cuando el proceso p recibe una modificación remota del proceso i, consulta el arreglo seq en su entrada i para cotejar los números de secuencia. En cuanto coinciden, se procede a su procesamiento. El proceso p actualiza $seq[i]$, aumentando su valor en 1.

Algoritmo 5: Código ejecutado por cada proceso, p. Versión intermedia.

Variables Persistentes:
$seq[\,]$, arreglo de enteros inicializado con cada entrada en 0
Q, $\subset \Sigma \times \mathcal{I} \times \{0,1\}$, inicialmente $\emptyset$.
sn, entero inicializado en 0.

1 **Función** *lee(x)*:
2 **if** `existe(`x`,`Q`)` **then**
3 **return** `actual(`x`,`Q`)`
4 **return** *false*

5 **Función** *agrega(a,x)*:
6 **if** `existe(`x`,`Q`)` **then**
7 **return** *false*
8 $Q = Q \cup \{(a, x, 1)\}$
9 $m = ((a, x, 1), sn)$
10 $sn = sn + 1$
11 $seq[p] = seq[p] + 1$
12 `broadcast(`m`)`
13 **return** *true*

14 **Función** *borra(x)*:
15 **if** `existe(`x`,`Q`)` **then**
16 $a =$ `elemento(`x`,`Q`)`
17 $Q = Q \cup \{(a, x, 0)\}$
18 $m = ((a, x, 0), sn)$
19 $sn = sn + 1$
20 $seq[p] = seq[p] + 1$
21 `broadcast(`m`)`
22 **return** *true*
23 **return** *false*

24 **Función** *estado()*:
25 $IDS =$ `ids(`Q`)`
26 $E = \{\}$
27 **foreach** *id in IDS* **do** $E = E \cup \{$`actual(`*id*`,`Q`)`$\}$
28 **return** E

29 **Upon** *(deliver(m=((a,x,y), qsn),q)*
30 *and q != p*
31 *and seq[q] == snq)*:
32 $Q = Q \cup \{(a, x, y)\}$
33 $seq[q] = seq[q] + 1$

Teorema 4. *El algoritmo 5 cumple con la definición 8.*

Demostración. Consistencia Eventual Fuerte se justifica de la misma manera que en el algoritmo anterior. Esto puede observarse gracias a que no hemos modificado la capa de comunicación, ni la estructura que mantiene el estado. Además las instrucciones que agregamos terminan en un número finito de pasos.

Tomemos una ejecución finita, E, del algoritmo 5 cuya historia asociada es H. Sea p un proceso correcto del cuál tomaremos su historia secuencial, $Sp = H|p$. Podemos observar que en la línea 32 se integran las modificaciones remotas, por lo que sustituiremos esas instrucciones por $q.agrega(a, x) : true$ y $q.borra(x) : true$, si $y = 1$ y $y = 0$, respectivamente.

Ahora demostraremos por inducción que S_p es legal. Sea L_i la secuencia que consta de las primeras i invocaciones con sus respectivas respuestas de S_p. Podemos verificar que una historia vacía es trivialmente legal. Supongamos que L_m es legal, observemos que sucede con operación $m + 1$, op_{m+1}:

1. Si op_{m+1} es *p.lee(x):R* no se modifica el estado, lo cuál es correcto. Dado que L_m es legal, el resultado, R, que entrega la función es correcto, pues primero se verifica que x exista (línea 2) y posteriormente se regresa su condición actual (línea 3). Si no existe, se regresa falso (línea 4).

2. Si op_{m+1} es *p.estado():R* no se modifica el estado, lo cuál es correcto. Dado que L_m es legal, el resultado, R, que entrega la función es correcto, pues se devuelve un conjunto con todos los identificadores presentes y su condición actual (línea 27).

3. Si op_{m+1} es *p.agrega(a,x):true* se modifica el estado, línea 8. Ya que L_m es legal, se puede garantizar que el identificador x no existía (línea 6), por lo cuál es correcto agregarlo al estado y responder $true$.

4. Si op_{m+1} es *p.agrega(a,x):false* no se modifica el estado. Como L_m es legal, se garantiza que x ya existía (línea 6), por lo tanto es correcto no modificar el estado y responder $false$.

5. Si op_{m+1} es *p.borra(x):true* se modifica el estado, línea 17. Ya que L_m es legal, se puede garantizar que el identificador x ya existía, por lo cuál es correcto borrarlo del estado y responder $true$.

6. Si op_{m+1} es *p.borra(x):false* no se modifica el estado. Como L_m es legal, se garantiza que x no existía, por lo tanto es correcto no modificar el estado y responder $false$.

7. Si op_{m+1} es *q.agrega(a,x):true* es correcto agregarlo pues el x no existe ya que la función generadora de identificadores garantiza unicidad de identificadores entre diferentes procesos. El estado refleja su inserción gracias a la línea 32.

8. Si op_{m+1} es *q.borra(x):true* se agrega al estado la modificación. Si x aun no está presente en el estado, se puede garantizar que eventualmente llegará el mensaje que la incluye pues todos los procesos correctos y si q borró x es porque ya existía y el mensaje dónde se crea x llegará a p por las garantías del BCR y de la corrección de los procesos.

Como podemos observar, en cada caso, se conserva la legalidad de la historia, por tanto S_p es legal.

Como último paso demostraremos que $Sp|q = A_{pM}(H)|q$. Supongamos que para algún q sucede que $Sp|q \neq A_{pM}(H)|q$ esto puede suceder por tres casos:

1. Hay una operación exitosa hecha por q que no fue recibida por p. Al ser un operación exitosa, q ejecutó la línea 12 o 21, según haya sido el caso, y por garantía del *Broadcast Confiable Regular* se entrega a todos los procesos correctos, p es un proceso correcto, por lo tanto p recibe la operación y eventualmente la integra. Con lo cual llegamos una contradicción.

2. Hay alguna operación que no aparece en orden correcto en p. Cuando q mandó la operación, agregó un número de secuencia. Según el código, en la línea 31, p verifica que la operación sea la siguiente de q. Por lo tanto p agrega en orden las operaciones de q. Con esto llegamos a una contradicción.

3. Hay una operación en S_p que no fue mandada por q. Por las propiedades del *Broadcast Confiable Regular*, no hay creación de mensajes, por lo tanto todo mensaje que reciben los procesos fueron efectivamente mandados por un emisor real. Por lo tanto llegamos a una contradicción.

En cada uno de los casos llegamos a una contradicción, por lo tanto podemos concluir que $Sp|q = A_{pM}(H)|q$.

De lo anterior podemos concluir que el algoritmo 5 cumple con la definición 8. ■

5.3.3. Acotaciones

Si bien el algoritmo impone un orden en la aplicación de las modificaciones por proceso, aún no se respeta el orden de tiempo real. Es decir, no hay orden con respecto a los otros procesos.

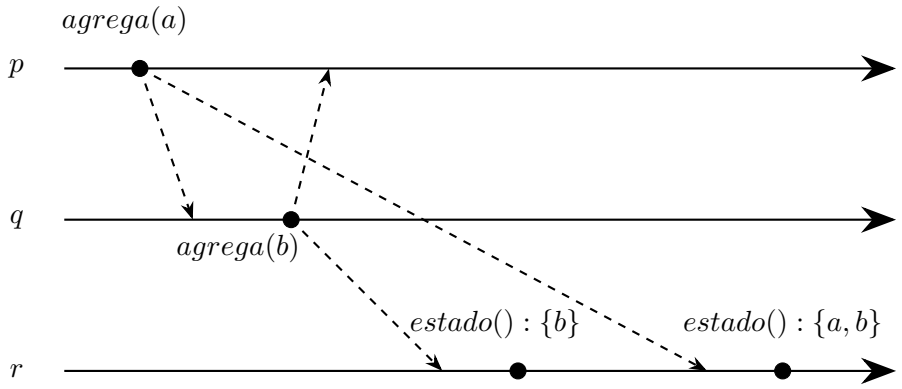

Figura 5.4: Ejecución que no respeta el orden de tiempo-real

En el diagrama anterior podemos observar como r procesa primero la modificación de q y posteriormente la de p, siendo que la de p fue primero, es decir, q ya conocía esa modificación cuando realizó la propia.

5.4. Implementación fuerte

El propósito de esta implementación es brindar un nivel de consistencia mayor entre las réplicas. Si bien las réplicas pueden tener estados diferentes, nos importa garantizar que se siga la misma secuencia en la integración de las modificaciones remotas. Para explicar mejor la idea mostraremos el siguiente ejemplo:

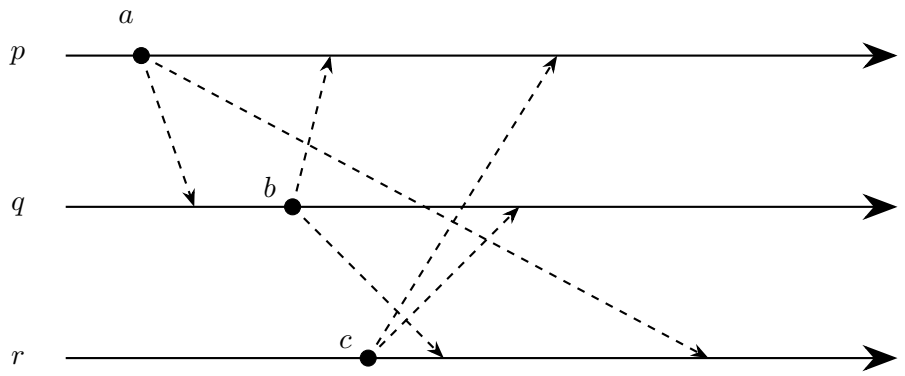

Figura 5.5: Concurrencia y causalidad

En el diagrama podemos observar que las operaciones de q y r son *concurrentes* pues cada una se hizo sin el conocimiento de la otra. En cambio, la operación de p precede a la de q, pues q realizó su operación posteriormente a recibir y aplicar la de p.

En esta implementación garantizamos que todas las relaciones de precedencia se respeten al momento de ser integradas a las réplicas. Este conjunto de precedencias forman un orden parcial de las modificaciones, que respetan todos los procesos.

Para lograr este comportamiento, utilizamos las ideas de [17] y relojes vectoriales [3]. Como mencionamos en la presentación de la implementación anterior, la solución que ofrece [17] era más sofisticada de lo que se requería en ese momento, pero para el nivel de consistencia que buscamos ahora, es posible retomarla. La corrección de su implementación se basa en la linealizabilidad de las transferencias exitosas hechas por procesos correctos. Para que un proceso correcto pueda decidir si una transferencia es exitosa o no, se le debe mandar información adicional que justifique esa transferencia, las *dependencias*. Gracias a lo anterior tenemos los indicios para definir la corrección de nuestra implementación y la certeza que necesitamos mandar, junto con la operación, información suficiente para que los procesos tengan la guía de cuándo incluir las actualizaciones y se respeten las precedencias. Uno de los mecanismos más conocidos para la causalidad, son los relojes vectoriales, y aprovechando que ya tenemos un vector, el que guarda los número de secuencia de cada proceso, fue natural adaptar ese mecanismo a nuestra implementación para representar la información extra que necesitamos.

5.4.1. Relojes Vectoriales

A continuación daremos una breve explicación de las relaciones de causalidad y concurrencia, qué son los relojes vectoriales, cómo se capturan esas relaciones y cómo los integramos a nuestra implementación.

Dada una ejecución, E, de un sistema concurrente, decimos que dos eventos de E, ϕ_1 y ϕ_2, guardan la relación de causalidad, $\phi_1 \rightarrow \phi_2$, si ϕ_1 influye en el cómputo de ϕ_2, es decir, ϕ_1 *precede* a ϕ_2. Ahora bien, si no es posible establecer $\phi_1 \rightarrow \phi_2$, ni $\phi_2 \rightarrow \phi_1$,

decimos que los eventos son *concurrentes*, $\phi_1 || \phi_2$.

Los *relojes vectoriales* son arreglos locales a cada proceso p, $VC_p[i]$, cuya entrada i corresponde al proceso i, es decir tiene tantas entradas como procesos hay en el sistema. Cada proceso actualiza su reloj cuando detecta un evento, ϕ, y asigna como identificador del evento el valor del reloj al término de la actualización, $VC(\phi)$. Las reglas que sigue p para mantener actualizado su reloj, son las siguientes:

1. Si el evento es producido en p, únicamente se incrementa en 1 el valor de $VC_p[p]$.
2. Si es un evento producido fuera de p, se incrementa en 1 el valor de $VC_p[p]$ y para el resto de las entrada, se coloca el máximo entre su valor actual y el correspondiente en el identificador del evento.

El siguiente es un ejemplo de cómo 3 procesos interactúan en una ejecución y los valores que van tomando sus relojes.

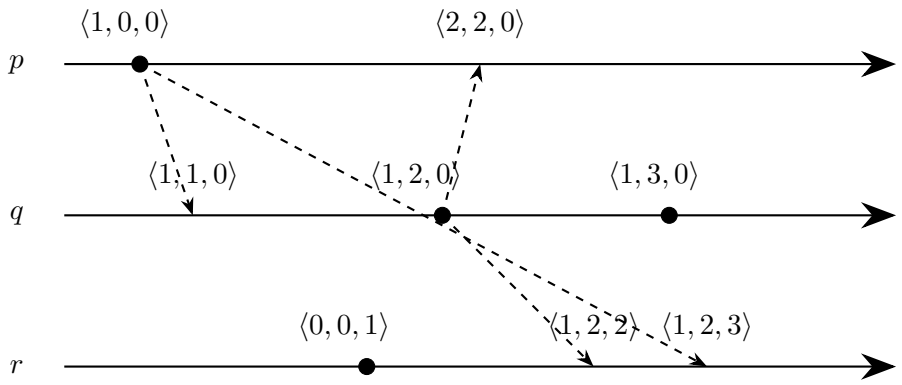

Figura 5.6: Eventos y relojes vectoriales

En general $VC_p[i], i \neq p$ es un estimado de p acerca de los pasos que ha dado i.

En las siguientes líneas explicaremos como integramos las ideas anteriores a la implementación intermedia 5:

- Cuando los procesos envían mensajes, deberán incluir su vector *seq* a modo de identificador.
- Como nuestro objetivo es respetar la causalidad de las modificaciones, seguiremos respetando el número de secuencia del proceso que realizó la modificación.
- Como lo anterior no basta, agregamos una restricción más para integrar modificaciones remotas: se integrarán siempre y cuando, el proceso local, p, tenga al menos el mismo conocimiento que tenía q cuando realizó la modificación ϕ. Es decir, $\forall i \neq q, VC(\phi)[i] \leq VC_p[i]$.
- Las modificaciones a los vectores *seq*, se harán únicamente del conocimiento nuevo adquirido, es decir, sólo se aumentará en 1 la entrada del proceso que realizó la

edición. De otro modo estaríamos registrando conocimiento que aún no ha llegado; sólo la noticia de su existencia.

A continuación formalizamos la intuición anterior.

5.4.2. Definición Formal

Definición 9. *Una implementación del objeto de edición colaborativa cumple la propiedad de* Consistencia Eventual Fuerte *y además, para E, una ejecución finita con historia asociada H, existe una historia secuencial legal S_p para cualquier p, tal que:*

1. $S_p|q = A_{pM}(H)|q$ *para todo* q.

2. $\prec_{A_{pM}(H)} \subseteq \prec_{S_p}$

$A_{pM}(H)$ se definió en 7.

La definición anterior fuerza a que cada proceso agregue las modificaciones remotas en el orden *de tiempo real* en el que fueron realizadas. Es decir, brinda linealizabilidad a las operaciones que modifican el estado.

5.4.3. Algoritmo

De igual manera, basamos la comunicación del algoritmo en el *Broadcast Confiable Regular*.

Algoritmo 6: Implementación que respeta el orden de tiempo real.

Variables Persistentes:
seq[ ], arreglo de entero inicializado con cada entrada en 0
Q, $\subset \Sigma \times \mathcal{I} \times \{0,1\}$, inicialmente $\emptyset$.
sn, entero inicializado en 0.

```
1  Función lee(x):
2      if existe(x,Q) then
3          return actual(x,Q)
4      return false
5  Función agrega(a,x):
6      if existe(x,Q) then
7          return false
8      Q = Q ∪ {(a, x, 1)}
9      m = ((a, x, 1), sn, seq)
10     sn = sn + 1
11     seq[p] = seq[p] + 1
12     broadcast(m)
13     return true
14 Función borra(x):
15     if existe(x,Q) then
16         a = elemento(x,Q)
17         Q = Q ∪ {(a, x, 0)}
18         m = ((a, x, 0), sn, seq)
19         sn = sn + 1
20         seq[p] = seq[p] + 1
21         broadcast(m)
22         return true
23     return false
24 Función estado():
25     IDS = ids(Q)
26     E = {}
27     foreach id in IDS do  E = E ∪ {actual(id,Q)}
28     return E
29 Upon (deliver(m=((a,x,y), qsn, qseq),q)
30         and q != p
31         and seq[q] == qsn
32         and ∀i, qseq[i] ≤ seq[i]):
33     Q = Q ∪ {(a, x, y)}
34     seq[q] = seq[q] + 1
```

Teorema 5. *El algoritmo 6 cumple con la definición 9.*

Demostración. Consistencia Eventual Fuerte se justifica de la misma manera que en el primer algoritmo.

Tomemos una ejecución finita, E, del algoritmo 6 cuya historia asociada es H. Sea p un proceso correcto del cuál tomaremos su historia secuencial, $Sp = H|p$. Podemos observar que en la línea 33 se integran las modificaciones remotas, por lo que sustituiremos esa instrucción por $q.agrega(a, x) : true$, si $y = 1$, y $q.borra(x) : true$, si $y = 0$.

La demostración de legalidad es análoga al algoritmo anterior.

Dado que el número de secuencia también se toma en cuenta para agregar una modificación, la demostración del inciso 1 de la definición también se demuestra de manera análoga a la prueba anterior. Procederemos ahora a demostrar que el algoritmo respeta el orden de tiempo real.

Sea $(op_i, op_j) \in \prec_{A_{pM}(H)}$, es decir, la operación op_i precede a la operación op_j en el tiempo real de la ejecución. Supongamos que $(op_i, op_j) \notin \prec_{S_p}$. Como S_p es una historia secuencial, tiene un orden total, por lo tanto en S_p, op_j precede a op_i. Sean r y q los procesos que realizaron op_i y op_j respectivamente. Tenemos ahora dos casos:

1. $q = r$

   Como en el tiempo real, op_i precede a op_j, el número de secuencia del primero es menor que el de la segunda operación y por el código, línea 32, S_p procesa primero op_i y después op_j. Por lo que llegamos a una contradicción.

2. $q \neq r$

   Cuando r mandó la operación op_i la acompañó de un vector, $rseq$, correspondientemente q acompañó su operación op_j con el vector $qseq$. De ambos vectores nos fijaremos en sus entradas q y r:

   - $rseq[r] = rsn$, $rseq[q] = qsn$
   - $qseq[r] = rsn + l$, $qseq[q] = qsn + m$, con $1 \leq l$, $0 \leq m$.

Para que p integre los cambios remotos tiene que comparar su vector seq con los vectores de los mensajes que llegan. Dado que el vector $qseq$ tiene en la entrada de r un número mayor a $seq[r]$ (si no ha integrado al menos el vector $rseq$) debe esperar (línea 32) a integrar el cambio de r, op_i. Por lo tanto en la historia secuencial de S_p sí aparecería primero op_i y después op_j. También llegamos a una contradicción.

Como en ambos casos surge una contradicción de suponer que op_i no precede a op_j en Sp, podemos concluir que S_p sí respeta el orden de tiempo real de la ejecución.

Por tanto el algoritmo cumple con la definición 9.

■

Para apreciar mejor las diferencias entre las implementaciones, presentamos el siguiente cuadro comparativo. n respresenta el número de procesos y *CEF*, consistencia eventual fuerte. Las comparaciones están hechas bajo el número de palabras que se mandan en los mensajes, no la longitud de ellas. Por ejemplo, un identificador de carácter lo tomaremos como una palabra sin importad su longitud, que en nuestro caso es no acotada, es decir podría ser muy grande, pero esta complejidad la delegamos al problema de la generación de identificadores con las características que los requerimos.

Implementación	**Tamaño mensaje**	**Consistencia**	**Dependencias**
Básica	$O(1)$	CEF	Ninguna
Intermedia	$O(1)$	CEF + Orden por fuente	Todos los mensajes previos que mandó ese proceso
Fuerte	$O(n)$	CEF + linealizabilidad	Todos los mensajes que conocía el proceso, al momento de enviar ese mensaje

Tabla 5.1: Cuadro comparativo de los implementaciones

Capítulo 6

Conclusiones

Bitcoin mostró la importancia de la descentralización de los sistemas, pues brinda privacidad y empodera a los usarios. La comunidad científica del cómputo distribuido ha ayudado a que esa idea mejore y se diversifique. Por lo que creemos relevante la costumbre de tomar problemas típicamente de la industria y observarlos bajo la lupa de la teoría. Si bien, para fines prácticos, el análisis teórico no es del todo palpable, sí lo es para la formulación de los algoritmos en un ambiente distribuido. Tener un conocimiento profundo del tema, permite manipular más fácilmente los algoritmos para agregarles propiedades, en este caso, diferentes niveles de consistencia. Por ello podemos comprobar que la metodología sugerida por el artículo [17], aunque implica más trabajo, sí es recomendable para abordar problemas distribuidos y proponer soluciones.

Los algoritmos presentados pueden tomarse como una extensión de los CRTS's, en el sentido en que su implementación es siemple, intuitiva y su corrección radica en la conmutatividad de las operaciones, pero ofrecen un grado mayor de consistencia. No sobra mencionar que, si bien los algoritmos son simples, la implementación de un conjunto denso y ordenado de identificadores no es tarea fácil. El que se describe en el capítulo 3 funciona bien en la teoría, pero en la práctica pueden llegar a tener identificadores de gran tamaño que, sumado a todos los caracteres agregados, puede ser difícil de mantener.

La contribuciones pricipales de este trabajo son:

1. El análisis teórico del problema de la edición colaborativa

2. Los algoritmos, totalmente descentralizados, que permiten la edición de un documento entre pares y que ofrecen mayor grado de consistencia que los actuales. Así como la prueba teórica de su corrección.

Como trabajo a futuro consideramos ahondar en la creación de un conjunto denso y ordenado de identificadores cuya implementación sea factible y eficiente, así como revisar las propuestas existentes [28]. Por otro lado, agregar a las implementaciones un mecanismo para reducir el tamaño del conjunto que representa el estado, de modo que el espacio requerido por cada réplica se reduzca.

Bibliografía

[1] Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., and Shavit, N. (1993). Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890. 20, 36

[2] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA. Association for Computing Machinery. 13

[3] Attiya, H. and Welch, J. (2004). *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley amp; Sons, Inc., Hoboken, NJ, USA. 42, 43, 57

[4] Auvolat, A., Frey, D., Raynal, M., and Taïani, F. (2020). Money transfer made simple: a specification, a generic algorithm, and its proof. *arXiv preprint arXiv:2006.12276*. 42, 50, 51

[5] Bach, L., Mihaljevic, B., and Zagar, M. (2018). Comparative analysis of blockchain consensus algorithms. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1545–1550. IEEE. 4

[6] Back, A. (1997). Hash cash postage implementation. 3

[7] Bailis, P. and Ghodsi, A. (2013). Eventual consistency today: limitations, extensions, and beyond. *Communications of the ACM*, 56(5):55–63. 2

[8] Brewer, E. A. (2000). Towards robust distributed systems. In *PODC*, volume 7, pages 343477–343502. Portland, OR. 2

[9] Cachin, C., Guerraoui, R., and Rodrigues, L. (2011). *Introduction to reliable and secure distributed programming*. Springer Science & Business Media. 43

[10] Chaudhry, N. and Yousaf, M. M. (2018). Consensus algorithms in blockchain: Comparative analysis, challenges and opportunities. In *2018 12th International Conference on Open Source Systems and Technologies (ICOSST)*, pages 54–63. IEEE. 4

[11] Collins, D., Guerraoui, R., Komatovic, J., Kuznetsov, P., Monti, M., Pavlovic, M., Pignolet, Y.-A., Seredinschi, D.-A., Tonkikh, A., and Xygkis, A. (2020). Online payments by merely broadcasting messages. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38. IEEE. 5

[12] Dijkstra, E. W. (1965). Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569. 12

[13] Dwork, C. and Naor, M. (1992). Pricing via processing or combatting junk mail. In *Annual international cryptology conference*, pages 139–147. Springer. 3

[14] Ellis, C. A. and Gibbs, S. J. (1989). Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 399–407. 5

[15] Eric, H. (1993). A cypherpunk's manifesto. *Satoshi Nakamoto Institute*. 3

[16] Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382. 4

[17] Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovič, M., and Seredinschi, D.-A. (2019). The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 307–316. 5, 6, 7, 42, 50, 57, 64

[18] Herlihy, M. (1991). Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149. 5, 13

[19] Herlihy, M. and Shavit, N. (2011). On the nature of progress. In *International Conference On Principles Of Distributed Systems*, pages 313–328. Springer. 18

[20] Herlihy, M. and Shavit, N. (2012). The art of multiprocessor programming, revised reprint. elsevier. 14, 20

[21] Herlihy, M. P. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492. 16, 36

[22] Lamport, L. (1974). A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455. 25, 36

[23] Lynch, N. A. (1996). *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 43

[24] Molli, P., Urso, P., Imine, A., et al. (2006). Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *2006 International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 1–10. IEEE. 5

[25] Nakamoto, S. and Bitcoin, A. (2008). A peer-to-peer electronic cash system. *Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf*, 4. 4

[26] Pervez, H., Muneeb, M., Irfan, M. U., and Haq, I. U. (2018). A comparative analysis of dag-based blockchain architectures. In *2018 12th International Conference on Open Source Systems and Technologies (ICOSST)*, pages 27–34. 4

[27] Preguica, N., Marques, J. M., Shapiro, M., and Letia, M. (2009a). A commutative replicated data type for cooperative editing. In *2009 29th IEEE International Conference on Distributed Computing Systems*, page nil. 2, 46

[28] Preguica, N., Marques, J. M., Shapiro, M., and Letia, M. (2009b). A commutative replicated data type for cooperative editing. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 395–403. 32, 64

[29] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. (2011). Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer. 6

[30] Stoll, C., Klaaßen, L., and Gallersdörfer, U. (2019). The carbon footprint of bitcoin. *Joule*, 3(7):1647–1661. 4

[31] Turing, A. M. (1937). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265. 10

[32] von Neumann, J. (1993). First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75. 10