
Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

# Implementación y estudio de un protocolo basado en replicación de datos

Tesis de Licenciatura en Ciencias de la Computación

Nicolás Len

Director: Christian Roldán

Buenos Aires, 2020

# RESUMEN

Muchas de las aplicaciones que utilizamos hoy les aseguran a sus usuarios que siempre van a estar disponibles, aunque la red por momentos se encuentre lenta o incluso fuera de servicio. Para alcanzar esto, los programadores escriben aplicaciones donde el estado se propaga de forma asincrónica a través de distintos dispositivos. Una implementación posible consiste en clientes (dispositivos) que mantienen una copia de los datos y un líder o servidor que decide un orden sobre las operaciones realizadas por los usuarios. La literatura ofrece distintos modelos, y sus diferencias están asociadas a si la propagación entre clientes-servidor y servidor-clientes es sincrónica o asincrónica.

En esta tesis estudiamos e implementamos GSP (por sus siglas en inglés *Global Sequence Protocol*), un modelo operacional que propaga operaciones de forma asincrónica en ambas direcciones, es decir, clientes-servidor y servidor-clientes. Para esto, la implementación se construye sobre una capa de broadcast llamada RTOB (por sus siglas *Reliable Total Order Broadcast*) que garantiza que todas las escrituras siempre son entregadas a cada cliente, en el mismo orden y sin perderse. Concretamente, desarrollamos una librería open-source de GSP, haciendo foco en estudiar cuáles garantías de consistencia, tales como *read my writes*, causalidad o prefijos, son alcanzadas por utilizar RTOB. Nuestros casos de estudio muestran que, en la práctica, GSP depende del protocolo de broadcast para asegurar ciertas garantías de consistencia.

**Palabras claves:** Global Sequence Protocol, Sistemas distribuidos, sistemas replicados, consistencia eventual, Reliable Total Order Broadcast, OCaml

# ABSTRACT

Many of the applications we use today assure their users that they will always be available, even if the network is slow or even out of order at times. To achieve this, developers write applications where the state propagates asynchronously across different devices. A possible implementation consists of clients (devices) that keep a copy of the data and a leader or server that decides an order on the operations performed by the users. The literature offers different models, and their differences are associated with whether the spread between client-servers and server-clients is synchronous or asynchronous.

In this thesis we study and implement GSP (*Global Sequence Protocol*), an operational model that propagates operations asynchronously in both directions, that is, client-servers and server-clients. For this, the implementation is built on a broadcast layer called RTOB (*Reliable Total Order Broadcast*) that guarantees that all the writes are always delivered to each client, in the same order and without getting lost. Specifically, we developed an open-source GSP library, focusing on studying which consistency guarantees, such as *read my writes*, causality, or prefixes, are achieved by using RTOB. Our case studies show that, in practice, GSP depends on the broadcast protocol to ensure certain guarantees of consistency.



**Keywords:** Global Sequence Protocol, Distributed Systems, Replicated Systems, Eventual Consistency, Reliable Total Order Broadcast, OCaml

# AGRADECIMIENTOS

A Christian, por darme la oportunidad de hacer este trabajo, por ofrecerme su tiempo hasta cuando no lo tenía, y por la paciencia y la voluntad para acompañarme en el desarrollo.
A todxs lxs profesores y ayudantes por su increíble vocación y dedicación para enseñar y transmitir conocimientos de tópicos tan complejos como los que presenta esta ciencia.
A todxs mis compañerxs de estudio con quienes nos complementamos para afianzar los conocimientos y transitar esta etapa de la mejor manera posible.
A mi novia Nicky, a mis amigos y amigas, a toda la banda de Fortu, y a mis padres por acompañarme, bancarme y apoyarme durante todo este largo camino lleno de sensaciones buenas y no tan buenas.
A todas las personas que hacen posible y defienden nuestro derecho de contar con una universidad pública y gratuita.
Y especialmente a mi hermano Juli, a quien le abrí las puertas a esta carrera y quien terminó siendo mi principal motor, apoyo y motivación para terminarla.

# Índice general

# 1. INTRODUCCIÓN

## 1.1. Motivación

El mundo del software está en constante cambio y evolución. Uno de los cambios más novedosos es la incorporación de la computación en la nube, o *cloud computing* [18]. La definición de cloud computing es ofrecer servicios a través de Internet a gran escala. Cuando hablamos de la nube, nos referimos al software y a los servicios que se ejecutan en internet, en lugar de en una computadora. Usamos el término *cloud* dado que las aplicaciones no se encuentran alojadas en un único servidor, sino en una red de servidores, es decir, algo tan difuso como una nube.

Para entender cómo funciona cloud computing, tomemos la definición de los *Sistemas Distribuidos* [14]. Un sistema distribuido es un sistema en donde los recursos de la red no se encuentran centralizados en una sola máquina, sino en varias máquinas dentro de la red, que incluso pueden estar en lugares físicos diferentes. Cloud computing es esto pero a gran escala, los recursos se encuentran en uno o varios servidores distribuidos en todo el mundo, y se pueden combinar los servicios de uno o varios proveedores y utilizarlos todos juntos. Por lo que podemos decir que Cloud computing es un gran sistema distribuido.

Previo a la creación de Cloud computing, ya existían las aplicaciones conocidas como *Web-based applications* (o aplicaciones basadas en la web) cuyas principales características son:

- En la Figura 1.1 se representa su arquitectura, que cuenta con (i) un servidor con almacenamiento persistente en la nube, y (ii) dispositivos de usuarios que realizan consultas a un servicio en la nube para comunicarse entre ellos.
- Para utilizarla se requiere de un navegador web.
- Debe tener una conexión a Internet continua e ininterrumpida para funcionar.
- Los datos utilizados se almacenan exclusivamente en el servidor.
- Tiene escalabilidad y disponibilidad limitadas.

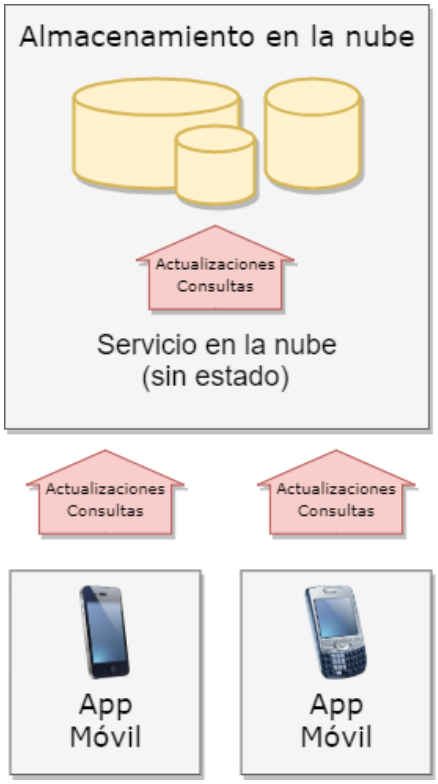


*Fig. 1.1:* Arquitectura de las Web-based applications.

Dentro del mundo de Cloud computing, se encuentran las *Cloud-based applications* (o aplicaciones basadas en la nube). En [32] se define a las Cloud-based applications como la evolución de las *Web-based applications* ya que:

- Se puede utilizar con o sin un navegador web.
- Los datos se pueden almacenar en caché localmente, por lo que puede funcionar sin conexión a internet y sincronizarse con los demás dispositivos cuando se restaura la conexión.
- Los datos utilizados podrían almacenarse en cualquier lugar, es decir, en la nube.
- Por lo general, es escalable on-demand y tiene poco o ningún tiempo de inactividad.

De estas características, la más novedosa es la capacidad de seguir funcionando sin conexión a internet, ya que la conectividad de los dispositivos puede llegar a perderse. Para esto, es necesario adoptar una estrategia que garantice que la carga y descarga de los datos compartidos entre todos los dispositivos, eventualmente converjan a un mismo estado consistente. Como solución se adopta la estrategia de **replicar en cada dispositivo la información compartida**, es decir, se usan réplicas con una copia del mismo estado en cada dispositivo.

En la Figura 1.2 se representa la arquitectura que adopta un sistema replicado. Por un lado se puede ver que se cuenta con almacenamiento en la nube. Por otro lado, cada dispositivo almacena una réplica de la información compartida que la aplicación usa

para efectuar lecturas y escrituras. Las flechas negras refieren a la sincronización que se hace entre los dispositivos y la nube. La flecha punteada refiere a una dispositivo que se encuentra sin conexión, pero que podrá sincronizarse una vez que se restablezca.

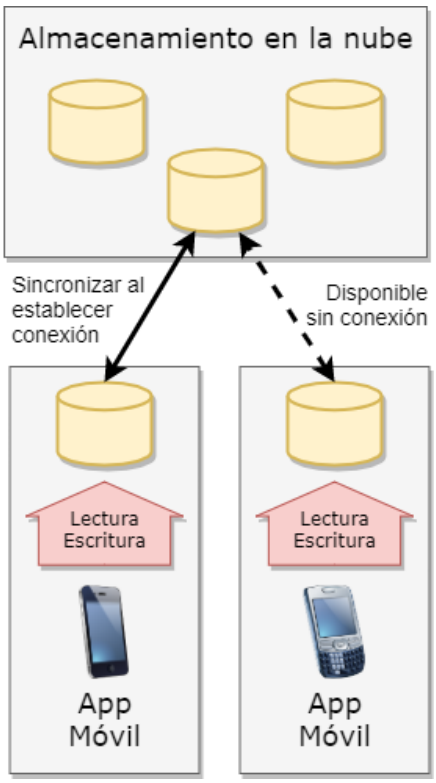


*Fig. 1.2:* Arquitectura para Estados Compartidos Replicados.

Por otro lado, para trabajar con sistemas distribuidos debemos conocer sus limitaciones. El teorema CAP [19] establece que no es posible garantizar simultáneamente las siguientes propiedades:

- Consistencia (**C**onsistency) atómica o linealizable, es decir, que cualquier lectura reciba como respuesta la escritura más reciente. En otras palabras, significa que debe existir un orden total de todas las operaciones, de manera que cada operación parezca completarse en un solo instante. Esto es equivalente a que las operaciones sobre los datos compartidos distribuidos, actúen como si se estuvieran ejecutando en un solo dispositivo, invocándose de a una a la vez [24].
- Disponibilidad (**A**vailability), es decir, que cualquier solicitud enviada por un dispositivo debe recibir una respuesta, aunque no se garantice que incluya el valor de la escritura más reciente.
- Tolerancia a fallas (**P**artition Tolerance), es decir, que el sistema siga funcionado incluso si un número arbitrario de mensajes enviados entre dispositivos de la red, se pierden o son retrasados.

Como consecuencia, una de estas tres propiedades debe ser descartada.

En la actualidad, se estudian estrategias para relajar la consistencia y para que un sistema distribuido ofrezca nociones más débiles de consistencia, para seguir garantizando disponibilidad y tolerancia a fallas. Para eso se definen *modelos de consistencia*. Un modelo de consistencia básicamente especifica un contrato entre el programador y el sistema, asegurando un determinado comportamiento por cada operación ejecutada en el sistema. Algunos modelos de consistencia estudiados, ordenados de más fuerte a más débil [7, 39, 27], son *Consistencia Secuencial*, *Consistencia Causal*, *Consistencia PRAM* y *Consistencia Eventual*.

En la actualidad, muchas bases de datos replicadas, como Cassandra [22] o Dynamo [15], son construidas ofreciendo Consistencia Eventual. La consistencia eventual garantiza que cualquier escritura será entregada a cada réplica del sistema y que eventualmente todas las réplicas convergerán a un mismo estado [6]. En otras palabras, si no se invocan nuevas operaciones de escritura, eventualmente todas las lecturas devolverán el mismo valor. La consistencia eventual es especialmente adecuada en contextos donde la coordinación entre dispositivos no es práctica o es demasiado costosa, como el caso de los entornos móviles [34].

Los sistemas replicados adoptan diferentes estrategias para alcanzar la consistencia eventual, y estas estrategias impactan directamente en las garantías que estos sistemas ofrecen, es decir, el tipo de inconsistencias o anomalías que pueden ocurrir [7] de cara al usuario. Las anomalías son caracterizadas en términos de *garantías de consistencia* como el caso de *read-your-writes* o *monotonic read* [41]. Para razonar con garantías de consistencia, necesitamos modelar sistemas de ejecuciones, y para eso usamos el concepto de *Ejecuciones Abstractas* [7], cuya definición incluye el término *Historia*. Una historia es un conjunto de operaciones que están ordenadas según el momento en el que son invocadas. Una ejecución abstracta es construida a partir de una historia y relaciones de orden que capturan el no determinismo del entorno asincrónico (entorno en el que pueden ejecutarse varias operaciones concurrentemente). Por ejemplo, esas relaciones de orden especifican el orden en el que se van a entregar los mensajes a los dispositivos.

Cuando las réplicas intentan converger a un mismo estado, hay distintas estrategias que una base de datos puede realizar. Una estrategia es que la misma base de datos sea responsable de resolver conflictos entre escrituras concurrentes [36] (por ejemplo, usando el timestamp de las operaciones), mientras que otra estrategia es que las aplicaciones sean las responsables de resolver el problema de la convergencia (como por ejemplo, el caso de Cassandra [22]).

En los últimos años se realizaron distintos trabajos para estudiar cómo las aplicaciones resuelven el problema de la convergencia. Por ejemplo, en [20] se define una clase de modelos de consistencia, llamada *Global Sequence Consistency* (GSC), que garantizan que los clientes eventualmente acuerden una secuencia global de operaciones. Previo a esto, cada cliente podrá leer una subsecuencia de esta secuencia final. Esa clase incluye los siguientes modelos:

- Total Store Order (TSO) [31, 30] se define como un modelo de memoria basado en buffers locales. Cada proceso tiene un buffer local en el que se almacenan las escrituras que invoca. Luego, las escrituras se van volcando asincrónicamente en la memoria principal. Por lo tanto, TSO no garantiza que una lectura invocada por un

proceso, devuelva lo que otro proceso escribió anteriormente.

- Dual TSO [2] presenta una semántica alternativa para TSO, que es más adecuada con la verificación de programas, es decir, de ciertas propiedades de seguridad que requieren los programas para poder ser ejecutados. Básicamente, a diferencia de TSO, los buffers intermedios entre los procesos y la memoria principal, en lugar de ser de escritura, son de lectura. Por lo tanto, las escrituras se hacen atómicamente a la memoria principal, y los valores se propagan asincrónicamente a todos los buffers permitiendo que cada proceso invoque lecturas a su buffer.

- **Global Sequence Protocol** (GSP) [11] propone un modelo operacional para razonar sobre aplicaciones que corren en sistemas distribuidos. GSP fue implementado en TouchDevelop, un entorno de programación para Windows 8 y Windows Phone diseñado por Microsoft Research, y cuenta con una semántica muy específica que asegura consistencia eventual. En GSP, el estado del sistema es representado como una secuencia de actualizaciones y cada cliente posee una copia local del mismo que no es necesariamente igual. Los cambios de estado son propagados de manera asincrónica mediante el uso de un protocolo llamado RTOB (*reliable total order broadcast*). RTOB es una primitiva de *broadcast* que garantiza que (i) todos los mensajes siempre son entregados a cada cliente en el mismo orden y (ii) ningún mensaje se pierde.

Aunque la propagación asincrónica de actualizaciones y la consistencia eventual ofrecen beneficios claros, también son más difíciles de entender, tanto para los desarrolladores de sistemas como para los programadores de aplicaciones, lo que motiva la necesidad de modelos de programación simples.

En particular, GSP es tan simple que sirve como alternativa para ayudar al programador a lidiar con la consistencia eventual, mientras que RTOB nos permite abstraernos de la propagación asincrónica de actualizaciones.

En [25] se propone un modelo operacional para GSP llamado GSP-CALCULUS, y a partir de este modelo se estudian formalmente las garantías de consistencia que GSP ofrece. Para esto, se reconstruyen las ejecuciones abstractas asociadas a las computaciones que suceden en el sistema y se verifica que toda ejecución cumpla con los modelos de consistencia descritos en [39, 7]. Dado que [25] estudia el protocolo formal asumiendo que: (i) los mensajes no se pierden y (ii) todos los mensajes siempre llegan en orden (es decir, garantías intrínsecas del protocolo de broadcast) nos propusimos estudiar el impacto de estas propiedades sobre las garantías de consistencia ofrecidas por GSP. Sin embargo, la plataforma TouchDevelop sobre la cual fue implementado GSP fue retirada[1] por Microsoft en Junio de 2019, y eso significó desarrollar una versión open-source del protocolo para luego responder esta pregunta.

## 1.2. Contribuciones

En este trabajo, desarrollamos una librería open-source[2] que implementa un servicio llamado *Global Sequence Protocol* (GSP) [11] con el objetivo de poder observar de forma empírica las garantías de consistencia que el protocolo ofrece y respaldar así, el estudio

[1] `https://makecode.com/touchdevelop`
[2] `https://git.exactas.uba.ar/nlen/tesis`

formal presentado en [25]. Para esto, implementamos nuestro propio programa CORE GSP, una versión básica de GSP que no incluye transacciones ni sincronización, y luego la extendimos a su versión robusta. Para lograr esto, fue necesario encontrar un programa que implemente la primitiva *Realiable Total Order Broadcast* (RTOB), es decir, que desde un dispositivo nos permita transmitir mensajes a todos los dispositivos de la red, con las garantías que esta primitiva ofrece.

Por último, pensamos y simulamos distintos escenarios de uso que nos permitieron ver en la práctica, cuáles son las garantías de consistencia que se cumplen y cuáles no, y cuál es la influencia de la primitiva de *broadcast* sobre ellas, respaldando lo demostrado a través de GSP-CALCULUS.

## 1.3. Organización

Introducimos nuestra implementación de GSP, la influencia de la primitiva de *broadcast* y el estudio de garantías de consistencia en cuatro etapas. Primero, explicamos el modelo GSP, la primitiva de *broadcast* que usa y las garantías de consistencia estudiadas (sección 2). Luego, explicamos nuestra implementación de CORE GSP y su extensión a una versión robusta, exponiendo sus arquitecturas y desarrollando acerca del programa que usamos para garantizar la primitiva de *broadcast* RTOB. Además, listamos las librerías que usamos en nuestro desarrollo y algunas mejoras que realizamos sobre la especificación del modelo GSP (sección 3).

En la sección 4 contamos cómo adaptamos nuestra implementación GSP para correr experimentos, explicamos los escenarios de uso que planteamos para analizar la influencia de la primitiva de *broadcast* utilizada y para demostrar de forma empírica las garantías de consistencia, y mostramos los resultados que obtuvimos a partir de los experimentos.

Por último, en la sección 5 contamos las conclusiones que obtuvimos con nuestro trabajo.

# 2. PRELIMINARES

## 2.1. Conjunto, Relaciones y Órdenes

Esta sección resume nociones básicas usadas en la tesis para razonar sobre *garantías de consistencia* y *modelos de consistencia*.

**Conjunto.** Un *conjunto* es una colección de objetos, llamados elementos, que tiene la propiedad que dado un objeto cualquiera, se puede decidir si ese objeto es un elemento del conjunto o no. Sea $A$ un conjunto. Se dice que un conjunto $B$ está contenido en $A$, y se nota $B \subseteq A$ (o también $B \subset A$), si todo elemento de $B$ es un elemento de $A$. En ese caso decimos también que $B$ está incluido en $A$, o que $B$ es un subconjunto de $A$. Si $B$ no es un subconjunto de $A$ se nota $B \nsubseteq A$ (o $B \not\subset A$). El producto cartesiano de $A$ con $B$, que se nota $A \times B$, es el conjunto de pares ordenados:

$$A \times B := \{(x, y) : x \in A, y \in B\}$$

**Relación binaria.** Sean $A$ un conjunto. Una *relación binaria* $\mathcal{R}$ en $A$ es un subconjunto cualquiera $\mathcal{R}$ del producto cartesiano $A \times A$. Es decir, $\mathcal{R}$ es una relación binaria en $A$ si $\mathcal{R} \subseteq A \times A$. Sea $x, y \in A$, se dice que $x$ está relacionado con $y$ por la relación $\mathcal{R}$ en $A$ si $(x, y) \in \mathcal{R}$, y lo notamos:

$$x \xrightarrow{\mathcal{R}} y$$

Sean $A$ un conjunto y $\mathcal{R}$ una relación binaria en $A$.

- Se dice que $\mathcal{R}$ es *reflexiva* si $\forall x \in A, x \xrightarrow{\mathcal{R}} x$.
- Se dice que $\mathcal{R}$ es *simétrica* si siendo $x, y \in A$, cada vez que $x \xrightarrow{\mathcal{R}} y$, entonces también $y \xrightarrow{\mathcal{R}} x$.
- Se dice que $\mathcal{R}$ es *antisimétrica* si siendo $x, y \in A$, cada vez que $x \xrightarrow{\mathcal{R}} y$ con $x \neq y$, entonces también $y \not\xrightarrow{\mathcal{R}} x$.
- Se dice que $\mathcal{R}$ es *transitiva* si para toda terna de elementos $x, y, z \in A$ tales que $x \xrightarrow{\mathcal{R}} y$ e $y \xrightarrow{\mathcal{R}} z$, se tiene también que $x \xrightarrow{\mathcal{R}} z$.
- Se dice que $\mathcal{R}$ es una *relación de equivalencia* cuando es una relación reflexiva, simétrica y transitiva. Las relaciones de equivalencia clasifican a los elementos del conjunto en subconjuntos donde se los considera "iguales" en algún sentido.
- Se dice que $\mathcal{R}$ es una *relación de orden*, o "orden en $\mathcal{R}$", cuando es una relación reflexiva, antisimétrica y transitiva. Una relaciones de orden formalizan la idea intuitiva de ordenación de los elementos de un conjunto, es decir, ayudan a la creación del orden del mismo.

- Se dice que $\mathcal{R}$ es una *relación de orden total* cuando todos los elementos de $A$ se relacionan entre sí, es decir,

$$\forall x, y \in A, (x \xrightarrow{\mathcal{R}} y) \vee (y \xrightarrow{\mathcal{R}} x)$$

- Se dice que $\mathcal{R}$ es una *relación de orden parcial* cuando al menos un par de elementos de $A$ se relacionan entre sí, es decir,

$$\exists x, y \in A, (x \xrightarrow{\mathcal{R}} y) \vee (y \xrightarrow{\mathcal{R}} x)$$

**Definición 2.1.1** (**Clausura transitiva**)**.** Sea $\mathcal{R}$ una relación binaria. La *clausura transitiva* de $\mathcal{R}$ es la relación más pequeña que siendo transitiva contiene al conjunto de pares de $\mathcal{R}$, y se nota $\mathcal{R}^+$. En otras palabras, $\mathcal{R}^+$ es la relación que verifica:

- $\mathcal{R} \subseteq \mathcal{R}^+$ (contiene al conjunto de pares de $\mathcal{R}$)
- $\mathcal{R}^+$ es transitiva
- Si $\mathcal{R}'$ es una relación transitiva tal que $\mathcal{R} \subseteq \mathcal{R}'$, entonces $\mathcal{R}^+ \subseteq \mathcal{R}'$ (es la relación más pequeña que cumple las otras dos condiciones).

**Definición 2.1.2** (**Composición de relaciones**)**.** Sean $A$ un conjunto, $\mathcal{R}$ y $\mathcal{S}$ dos relaciones binarias en $A$. La composición de $\mathcal{R}$ y $\mathcal{S}$ en $A$ es la relación binaria formada por los pares $(x, y)$ para los que existe un $z$ tal que $(x, z) \in \mathcal{R}$ y $(z, y) \in \mathcal{S}$ y lo notamos $\mathcal{R}; \mathcal{S}$. En otras palabras:

$$\mathcal{R}; \mathcal{S} = \{(x, y) \in \mathcal{A} \times \mathcal{A} \mid \exists z \in \mathcal{A} : (x, z) \in \mathcal{R} \wedge (z, y) \in \mathcal{S}\}$$

## 2.2. Ejecución abstracta

Para interpretar *ejecuciones abstractas* definimos una *historia* y las posibles relaciones que pueden haber entre sus elementos:

**Definición 2.2.1** (**Historia**)**.** Una historia $H$ es el conjunto de operaciones invocadas en una ejecución determinada. Las operaciones pueden ser escrituras o lecturas.

Definimos, además, las siguientes relaciones en elementos de una historia:

- **returns-before (*rb*)** es una relación de orden parcial natural en $H$ basado en precedencia en tiempo real. En otras palabras, $rb$ captura el orden de las operaciones que no se superponen. Sean $e, f \in H, e \xrightarrow{rb} f$ denota que $e$ fue invocada antes que $f$.
- **same-session (*ss*)** es una relación de equivalencia en $H$ que agrupa pares de operaciones invocadas por el mismo cliente.
- **session-order (*so*)** es una relación de orden parcial definida como: $so = rb \cap ss$. Sean $e, f \in H$, $e \xrightarrow{so} f$ denota que ambas operaciones fueron invocadas por un mismo cliente, y que $e$ fue invocada antes que $f$.

- **visibility (*vis*)** es una relación natural acíclica que explica cómo se propagan las operaciones de escritura. Sean $e$ y $f$ dos operaciones de $H$, $e \xrightarrow{vis} f$ denota que los efectos de la operación $e$ son visibles para la operación $f$ (por ejemplo, $f$ leerá el valor escrito por $e$).
- **arbitration (*ar*)** es un orden *total* de las operaciones de $H$. Sean $e$ y $f$ dos operaciones de $H$, $e \xrightarrow{ar} f$ denota que la operación $e$ precede a la operación $f$ en el registro del servidor. Este orden especifica cómo el sistema resuelve conflictos que ocurren por operaciones concurrentes. En la práctica, el orden total puede ser construido de diferentes maneras, como por ejemplo usando timestamps en las operaciones.
- **happens-before (*hb*)** es la clausura transitiva de la unión de $so$ y $vis$, es decir, $hb = (so \cup vis)^+$. Sean $e, f, g$ tres operaciones, si ocurre $e \xrightarrow{so} f$ ó $e \xrightarrow{vis} f$, y $f \xrightarrow{so} g$ ó $f \xrightarrow{vis} g$, entonces ocurre $e \xrightarrow{hb} f$, $f \xrightarrow{hb} g$ y $e \xrightarrow{hb} g$. Explicaremos para qué sirve esta relación al definir *Garantías de Causalidad* en la Sección 2.3.

**Definición 2.2.2.** (**Ejecución abstracta**) Sea $H$ una historia, y $vis$ y $ar$ relaciones en elementos de $H$, una ejecución abstracta es un multigrafo dirigido $A = (H, vis, ar)$ donde los vértices representan las operaciones de $H$, y los ejes representan las relaciones $vis$ y $ar$ entre esas operaciones. Mientras que las historias describen los resultados observables de las ejecuciones, $vis$ y $ar$ capturan el no determinismo del entorno asincróno. En otras palabras, $vis$ y $ar$ determinan las relaciones entre pares de operaciones de $H$ que explican y justifican sus resultados.

## 2.3. Modelos y Garantías de consistencia

En esta sección definimos las garantías de consistencia y los modelos de consistencia, y explicamos cómo se relacionan.

**Definición 2.3.1.** (**Garantía de consistencia**) Una *garantía de consistencia* es una propiedad de una ejecución abstracta que especifica el conjunto de resultados aceptables que puede devolver cada operación de lectura. El tamaño de este conjunto determina la "fuerza" de la consistencia: los conjuntos de resultados aceptables más pequeños implican una consistencia más fuerte, y viceversa.

Para definir un modelo de consistencia, agrupamos las garantías de consistencia necesarias y luego especificamos que una ejecución abstracta, tal que satisface a todas esas garantías, debe validar la correctitud de las historias.

**Definición 2.3.2.** (**Historia correcta**) Sea $H$ una historia, y $\mathcal{P}_1, \cdots, \mathcal{P}_n$ una conjunto de garantías de consistencias. Decimos que $H$ satisface las garantías $\mathcal{P}_1, \cdots, \mathcal{P}_n$ si puede ser extendida (agregando visibilidad y arbitrariedad) a una ejecución abstracta que las satisface.

**Definición 2.3.3.** (**Modelo de consistencia**) Un modelo de consistencia es una combinación de garantías de consistencia que especifica un contrato entre el programador y el sistema, que asegura un determinado comportamiento por cada operación ejecutada en el sistema. En otras palabras, el sistema garantiza que si el programador sigue determinadas reglas, los datos almacenados en la memoria compartida serán consistentes y los resultados de leer, escribir o actualizar la memoria compartida serán predecibles.

Los modelos de consistencia se dividen en dos grandes grupos: Las consistencias fuertes (tales como *Consistencia Linealizable* y *Consistencia Secuencial*) y las consistencias débiles (tales como *Consistencia Causal*, *Consistencia PRAM* y *Consistencia Eventual*).

Por otro lado, los modelos de consistencia pueden incluir tres tipos de garantías de consistencia:

- **Semánticas de tipos de datos** que dan sentido a las operaciones.
- **Garantías de convergencia** que aseguran que eventualmente los datos serán consistentes.
- **Garantías de orden** que descartan anomalías causadas por los diferentes órdenes en los que pueden ocurrir las operaciones. Las garantías de orden se divide en dos grupos:
  - **Garantías de sesión** Cuando un cliente invoca varias operaciones, espera que el orden en el que las invoca se preserve. Por ejemplo, *read my writes*, *monotonic reads* y *consistent prefix* son garantías de sesión.
  - **Garantías de causalidad** En sistemas distribuidos usamos la noción de $hb$ para operaciones que tienen una relación potencialmente causal. Por un lado, si dos operaciones son invocadas por el mismo cliente, pueden estar causalmente relacionadas: el cliente pudo haber decidido invocar una operación basada en los valores que retornó una operación anterior. Por otro lado, si una operación $A$ es visible para una operación $B$, el valor devuelto por $B$ puede depender de $A$. Por eso se define $hb = (so \cup vis)^+$. *No circular causality*, *causal arbitration* y *causal visibility* son ejemplos de garantías de causalidad.

A continuación, explicamos los modelos de consistencia mencionados, ordenados de más fuerte a más débil [7, 39, 27]:

- **Consistencia Linealizable** (también conocida como consistencia atómica)
- **Consistencia Secuencial** La consistencia secuencial es una garantía de orden que especifica que el resultado de una ejecución es el mismo si las operaciones (lecturas y escrituras) de todos los dispositivos sobre el dato compartido fueron ejecutadas en algún orden secuencial, y las operaciones de cada dispositivo individual aparecen en esta secuencia en el orden en que fueron invocadas por su dispositivo. Esto quiere decir que en este modelo sólo importa que las operaciones invocadas por un dispositivo sean vistas por otro en el mismo orden, sin importar que se intercalen con las operaciones de otros dispositivos.
- **Consistencia Causal** La consistencia causal captura las posibles relaciones causales entre las operaciones y garantiza que todos los procesos observen las operaciones causalmente relacionadas en un orden común. En otras palabras, todos los procesos del sistema acuerdan el orden de las operaciones causalmente relacionadas, pero pueden disentir sobre el orden de las operaciones que causalmente no están relacionadas.
- **Consistencia PRAM** La consistencia Pipeline RAM (PRAM), también conocida como Consistencia del Procesador, determina que todos los dispositivos ven las operaciones de escritura emitidas por un dispositivo en el mismo orden en que fueron

invocadas por ese dispositivo. Por otro lado, los dispositivos pueden observar escrituras emitidas por diferentes dispositivos en diferentes órdenes. Por lo tanto, no se requiere un orden total global. Sin embargo, las escrituras de cualquier dispositivo deben ser serializadas (o transmitidas) en orden, como si estuvieran en una tubería (pipeline), de ahí el nombre.

- **Consistencia Eventual** La consistencia eventual garantiza que cualquier escritura será entregada a cada réplica del sistema y que eventualmente todas las réplicas convergerán a un mismo estado [6]. En otras palabras, si no se invocan nuevas operaciones de escritura, eventualmente todas las lecturas devolverán el mismo valor. La consistencia eventual es especialmente adecuada en contextos donde la coordinación entre dispositivos no es práctica o demasiado costosa (por ejemplo, en entornos móviles) [34]. Por ejemplo, en la actualidad muchas bases de datos replicadas, como Cassandra [22] o Dynamo [15], son construidas ofreciendo Consistencia Eventual.

Por último, definimos y explicamos las garantías de orden mencionadas:

- **Read My Writes** $\stackrel{def}{=} (so \subseteq vis)$: *Read My Writes* es una garantía de sesión que dice que por cada par de operaciones $e, f$ que un cliente invoca, si $e$ es invocada antes que $f$, entonces los efectos de $e$ serán visibles para $f$.

- **Monotonic Reads** $\stackrel{def}{=} (vis; so) \subseteq vis$: *Monotonic Reads* es una garantía de sesión que dice por cada terna de operaciones $e, f, g$ si ocurre simultáneamente que: los efectos de $e$ son visibles para $f$, $f$ y $g$ son invocadas por un mismo cliente y $f$ es invocada antes que $g$. Entonces los efectos de $e$ también serán visibles para $g$.

- **Consistent Prefix** $\stackrel{def}{=} (ar; (vis \cap \neg ss)) \subseteq vis$: *Consistent Prefix* es una garantía de sesión que dice que por cada terna de operaciones $e, f, g$, si ocurre simultáneamente que:

  - $e$ ocurre antes que $f$ en la secuencia global,
  - $f$ fue invocada por el cliente $i$,
  - $g$ fue invocada por el cliente $j$ $(i \neq j)$, y
  - los efectos de $f$ son visibles para $g$.

  Entonces los efectos de $e$ serán visibles para $g$.

- **No Circular Causality** $\stackrel{def}{=} acyclic(hb)$: *No Circular Causality* es una garantía de causalidad que dice que la relación $hb$ es acíclica. Por ejemplo, sean $e, f, g$ tres operaciones tales que $f$ fue invocada basada en los valores que retornó $e$ y $g$ fue invocada basada en los valores que retornó $f$, no puede pasar que $e$ haya sido invocada basada en los valores que retornó $g$.

- **Causal Arbitration** $\stackrel{def}{=} (hb \subseteq ar)$: *Causal Arbitration* es una garantía de causalidad que dice que el orden de causalidad de las operaciones está contenido en el orden de la secuencia global. Por ejemplo, sean $e, f$ dos operaciones tales que $f$ fue invocada basada en los valores que retornó $e$, entonces la secuencia global tendrá a la operación $f$ ordenada después de $e$.

- **Causal Visibility** $\stackrel{def}{=}$ $(hb \subseteq vis)$: *Causal Visibility* es una garantía de causalidad que dice que el orden de causalidad de las operaciones, está contenido en el orden de visibilidad. Por ejemplo, sean $e, f, g$ tres operaciones tales que $f$ fue invocada basada en los valores que retornó $e$. Si $g$ visualiza los efectos de $f$, entonces también visualizará los efectos de $e$, en el orden: $e \longrightarrow f$.

## 2.4. Global Sequence Protocol

*Global Sequence Protocol* (GSP) [11], es un modelo operacional que sirve para replicar información compartida, y que describe el comportamiento del sistema con tanta precisión que nos sirve como un modelo simple para entender cómo funciona el almacenamiento replicado.

En GSP el estado del sistema es representado como una secuencia de actualizaciones, llamada *secuencia global de actualizaciones*. Cada cliente posee una copia local con una subsecuencia del mismo, y eventualmente todas esas copias convergen a un mismo estado.

Además, en GSP cada actualización es propagada de manera asincrónica mediante el uso de un protocolo llamado *Reliable Total Order Broadcast* (RTOB) que explicaremos en la sección 2.5.

La Figura 2.1 describe sencillamente el mecanismo que realiza GSP: (i) las operaciones de lectura acceden únicamente a la réplica local y (ii) las operaciones de actualización se almacenan en la réplica local y además se copian en un *buffer* al cual RTOB accede para propagarlas a todos los clientes y actualizar sus réplicas locales.

Una ventaja de este protocolo es que ofrece un nivel de **abstracción** muy alto para razonar sobre estados compartidos replicados, ya que se basa en un modelo de datos abstracto. Esto significa que la información compartida que replicamos en cada dispositivo cliente, tiene un tipo de datos genérico que puede ser instanciado con cualquier tipo de datos particular, como por ejemplo: registros, contadores, *key-values* y *cloud types* [9]. Estos tipos de datos cuentan con sus propias operaciones que definen la semántica de las actualizaciones que generan los clientes, y por ende, que contiene la secuencia global que define al estado del sistema.

Otra ventaja que presenta GSP es que en caso de que deseemos modificar el tipo de datos con sus operaciones, este modelo de datos abstracto nos evita tener que modificar la implementación del servidor, ya que el cliente es el responsable de almacenar la semántica de las operaciones.

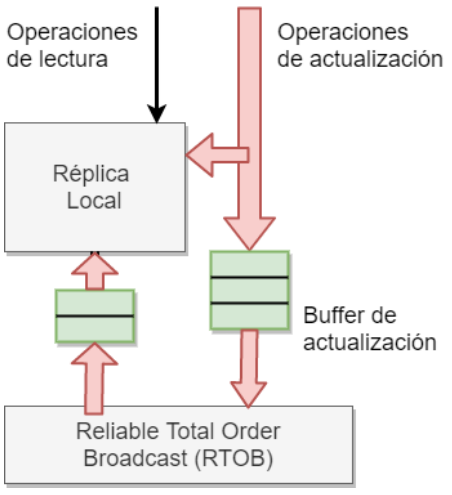


*Fig. 2.1:* Global Sequence Protocol.

Por otra parte, GSP es **robusto**, lo que significa que su funcionamiento contempla que los dispositivos puedan estar desconectados, o tener conexiones que fallan o que son muy lentas. Esto se logra principalmente gracias a que GSP especifica que el servidor debe comunicarle a cada cliente si sus operaciones ya fueron recibidas. De esta manera,

si un cliente se entera que alguna de sus operaciones no fueron recibidas, deberá reenviarla.

Como ejemplo, supongamos que para un evento se necesita contar la cantidad de ingresos entre varios responsables que se encuentran situadas en distintos accesos, de manera que un mismo ingreso no podrá ser contabilizado dos veces. Cada responsable tendrá en su dispositivo un botón para incrementar un contador compartido. Supongamos que durante un intervalo de tiempo, un responsable se queda sin conexión e ingresan tres nuevos invitados al evento. Luego de ese intervalo, este responsable recupera la conexión.

Cuando el cliente restablece la conexión, envía tres operaciones al servidor que indican que se debe incrementar al contador, independientemente del valor que tenga anteriormente. Además, si al conectarse nuevamente, se producieran fallas en la conexión, en algún momento al cliente recibirá la información que dirá si sus operaciones enviadas fueron recibidas. En caso contrario, el cliente sabrá que deberá reenviarlas.

Este mecanismo, nos evita preocuparnos por los cambios que puedan haber realizado otros dispositivos en ese contador mientras no estábamos conectados, y por fallas que pudieron haber ocurrido con la conexión.

Otra ventaja a destacar de GSP es que brinda la capacidad de realizar operaciones de manera sincrónica, aún siendo un modelo asincrónico, es decir, que si nos encontramos en alguna situación en la que necesitamos que la propagación de la actualización sea sincrónica (por ejemplo, finalizando una reserva de un pasaje de avión), GSP permite recuperar consistencia fuerte y obtener un comportamiento sincrónico. Durante el lapso de tiempo que se procesa la operación sincrónica, el cliente queda bloqueado esperando que finalice, por lo que pierde disponibilidad (siguiendo con lo explicado por el teorema de CAP [19]).

GSP se introduce en dos etapas. Primero se introduce CORE GSP y luego se lo extiende a una versión robusta de GSP:

### *Core GSP*

Por un lado, CORE GSP es una versión simplificada de GSP que incluye lecturas y actualizaciones de datos, pero no cuenta con transacciones ni con sincronización. Considerar los tipos abstractos *Update*, *Read* y *Value*, y la función abstracta *rvalue*:

abstract type $Update, Read, Value$;
function $rvalue : Read \times Update\ * \rightarrow Value$

La implementación que se elija para ellos, determina el tipo de la información que se comparte, y la semántica de sus operaciones. Esto define el modelo de datos en el que se basa GSP para esta versión. Los ejemplos 2.4.1 y 2.4.2 muestran posibles instancias para estos tipos abstractos, y para la función *rvalue*.

**Ejemplo 2.4.1** (Contador)**.** *Podemos definir un modelo de datos para un contador de la siguiente manera:*

$$Update = \{\, inc \,\}$$
$$Read = \{\, rd \,\}$$
$$rvalue(rd, s) = s.length$$

*donde una lectura simplemente cuenta los números de actualizaciones.*

**Ejemplo 2.4.2** (Key-Value)**.** *También podemos definir un modelo de datos para el tipo de datos key-value de la siguiente manera:*

$$
\begin{aligned}
&\mathit{Update} = \{\ \mathit{wr}(k,v) \mid k,v \text{ in } \mathit{Value}\ \}\\
&\mathit{Read} \quad = \{\ \mathit{rd}(k) \mid k \text{ in } \mathit{Value}\ \}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{rvalue}(\mathit{rd}(k), s) = \ &\text{match } s \text{ with}\\
&[] \qquad\qquad \to \text{undefined}\\
&s_0 \cdot \mathit{wr}(k_0,v) \to \text{if } (k = k_0) \text{ then } v \text{ else } \mathit{rvalue}(\mathit{rd}(k), s_0)
\end{aligned}
$$

**Ejemplo 2.4.3** (Registro)**.** *También podemos definir un modelo de datos para un registro de la siguiente manera:*

$$
\begin{aligned}
&\mathit{Update} = \{\ \mathit{wr}(v) \mid v \text{ in } \mathit{Value}\ \}\\
&\mathit{Read} \quad = \{\ \mathit{rd}\ \}\\
&\mathit{rvalue}(\mathit{rd}, s) = \ \text{match } s \text{ with}\\
&\qquad\qquad [] \qquad\quad \to \text{undefined}\\
&\qquad\qquad s_0 \cdot \mathit{wr}(v) \to v
\end{aligned}
$$

Además, las actualizaciones en CORE GSP se encapsulan en objetos de tipo *round* para ser propagadas usando el protocolo de broadcast RTOB:

$$\text{class } \mathit{Round}\ \{\ \mathit{origin} : \mathit{Client},\ \mathit{number} : \mathbb{N},\ \mathit{update} : \mathit{Update}\ \}$$

donde *origin* contiene el identificador del cliente, *number* el número de round y *update* la actualización.

### *Versión extendida de GSP*

Por otro lado, CORE GSP se extiende a una versión robusta con varias mejoras entre las cuales destacamos:

- **Transacciones y primitivas de sincronización** significa agregar las siguientes operaciones:
  - **Operación PUSH** A diferencia de CORE GSP, en la versión robusta se pueden propagar múltiples escrituras atómicamente gracias a la operación explícita *push* (las escrituras son propagadas en un único *Round* únicamente cuando el cliente llama a *push*).
  - **Operación PULL** En CORE GSP las actualizaciones entrantes realizadas por otros clientes, pueden generar carreras indeseadas en el programa. En la versión robusta esto se soluciona controlando cuándo se hacen efectivas esas actualizaciones gracias a la operación explícita *pull*.
  - **Operación CONFIRMED** A diferencia de CORE GSP, se cuenta con la propiedad *confirmed* que retorna verdadero cuando no hay ninguna actualización local esperando que el servidor confirme su recepción.

- **Operación FLUSH** Aunque no forme parte de la especificación del modelo, la operación *flush* demuestra que las operaciones *push*, *pull* y *confirmed* permiten realizar actualizaciones de manera sincrónica, como habíamos mencionado:

  **function** FLUSH
  *push*();
  **while** !*confirmed*() **do**
  *pull*();
  **end while**
  **end function**

  Cuando un cliente realiza una actualización que requiere propagarse de manera sincrónica, basta con invocar a la operación *flush* para bloquearse hasta ser confirmada.

- **Estructuras de datos: States y Deltas**. Mientras que en CORE GSP las actualizaciones que aún no han sido enviadas se almacenan en distintas secuencias (sin ningún límite de capacidad), en la versión robusta se reducen tales secuencias y se almacenan en forma reducida en objetos de tipo *State* o de tipo *Delta*:

  abstract type *State*
  abstract type *Delta*

  Los *Deltas* representan segmentos (o intervalos) de la secuencia global de actualizaciones y son creados añadiendo actualizaciones, o reduciendo varios *Deltas*, mientras que los *States* representan prefijos de la secuencia global de actualizaciones y son creados aplicando *Deltas* al *State* inicial. Esto se puede ver especificado en la definición de cada función:

  $$\begin{array}{lll} \text{const} & \mathit{initialstate} & : \mathit{State} \\ \text{function} & \mathit{read} & : \mathit{Read} \times \mathit{State} \to \mathit{Value} \\ \text{function} & \mathit{apply} & : \mathit{State} \times \mathit{Delta}^* \rightharpoonup \mathit{State} \\ \text{const} & \mathit{emptydelta} & : \mathit{Delta} \\ \text{function} & \mathit{append} & : \mathit{Delta} \times \mathit{Update} \rightharpoonup \mathit{Delta} \\ \text{function} & \mathit{reduce} & : \mathit{Delta}^* \rightharpoonup \mathit{Delta} \end{array}$$

  Estas funciones deben ser instanciadas desde el programa cliente con la semántica del tipo de datos que se desee usar.

  **Ejemplo 2.4.4** (Key-Value)**.** *Podemos definir un modelo de datos para el tipo de datos Key-Value representando State y Delta como mapeos de claves a valores, e instanciando* reduce*,* append *y* apply *como funciones que simplemente combinan esos mapeos (donde la última escritura gana).*

  El Ejemplo 2.4.4 describe cómo deberían instanciarse los tipos abstractos *State* y *Delta*, para implementar el tipo Key-Value.

- **Canales** En CORE GSP los clientes se comunican directamente usando las funciones que brinda RTOB. En cambio, en la versión robusta, la comunicación entre cada

cliente y el servidor se realiza usando *sockets*. Para esto, se usa un objeto *Channel* por cada canal establecido entre un cliente y el servidor, que contiene dos listas: (i) *clientstream* en la que se agregan las actualizaciones que el servidor desea enviarle a ese cliente, y (ii) *serverstream* en la que se agregan las actualizaciones que ese cliente desea enviarle al servidor. Para propagar una actualización, cada cliente agrega a la lista (ii) un objeto de tipo *Round* con la actualización encapsulada:

$$\mathsf{struct}\ \mathit{Round}\,\{\mathit{origin}:\ \mathit{Client};\ \mathit{number}:\ \mathbb{N};\ \mathit{delta}:\ \mathit{Delta};\}$$

El tipo *Round* que se usa en esta versión, difiere de la que se usa en CORE GSP en que en lugar de contener una actualización, contiene un objeto *Delta*.

- **Servidor** Mientras que en CORE GSP la conexión se hace únicamente entre clientes, sin ningún servidor que intervenga, en la versión robusta se cuenta con un servidor que cumple la función de almacenar el estado persistente, establecer conexión con cada cliente nuevo, y comunicar a todos los clientes conectados las actualizaciones que va recibiendo. Para lograr esto, el servidor usa objetos de tipo *GSPrefix* y *GSSegment*:
  - **GSPrefix** El servidor almacena la secuencia global de actualizaciones en forma reducida en un *state*. Cuando un nuevo cliente desea establecer conexión con el servidor, el servidor le envía ese *state*, encapsulado en un objeto de tipo *GSPrefix*.
  - **GSSegment** Para propagar las actualizaciones recibidas, el servidor las reduce en un único *delta*, lo encapsula en un objeto de tipo *GSSegment* y, por último, lo agrega en la lista (i) de cada canal establecido con cada cliente.

  *GSPrefix* y *GSSegment*, también contienen un diccionario que por cada cliente, el servidor guarda el número de *round* más alto que recibió. De esta manera, cada cliente se mantiene informado acerca de las actualizaciones que ya fueron recibidas por el servidor.

## 2.5. Reliable Total Order Broadcast

Otro beneficio que ofrece este modelo, es que permite abstraernos de la propagación de las actualizaciones, es decir, del envío de mensajes en los que se comunican las actualizaciones que realiza cada cliente. Esta abstracción no sólo nos permite prescindir de un mecanismo para garantizar la recepción de los mensajes por parte de todos los clientes, sino que también nos garantiza que a todos ellos les van a llegar las actualizaciones en el mismo orden.

Esta abstracción es posible gracias a *Reliable Total Order Broadcast* (RTOB), una primitiva de comunicación grupal que puede ser implementada para topologías client-server o peer-to-peer. Esta primitiva nos garantiza que todos los mensajes que enviamos desde un dispositivo hacia toda la red, se entregan con confianza y en un mismo orden, a todos los clientes. En otras palabras, RTOB es quien crea el orden arbitrario entre todos los mensajes de todos los clientes, respetando el orden de los mensajes enviados por cada cliente en particular y garantizando que todos reciban los mensajes en ese mismo orden arbitrario.

Destacamos que RTOB fue bien estudiada en la literatura de los sistemas distribuidos, y es usada frecuentemente para construir sistemas con estados compartidos [12, 17].

# 3. IMPLEMENTACIÓN EN OCAML

En este capítulo presentamos nuestra propia implementación de GSP escrita en OCaml [26], un lenguaje funcional que ofrece primitivas para programación asíncrona y distribuida. Esta implementación se basa en el cálculo de procesos presentado en [25], cuyos términos son definidos de forma inductiva y cuya semántica operacional fue desarrollada usando mecanismos tradicionales de la programación funcional como las funciones de alto orden. Entre las ventajas de usar OCaml, queremos remarcar:

- **No es un lenguaje puramente funcional**. Esto nos permitió, en el contexto adecuado, poder trabajar con funciones que generan efectos. En consecuencia, nuestra implementación cuenta con *tipos de datos mutables*, es decir, tipos de datos cuyos valores puedan ser modificados. En particular, esto será útil para compartir variables entre distintos hilos de ejecución, y que cada hilo pueda modificar una variable e impacte al resto de los procesos. En la Sección 4.1.1 revisaremos este aspecto.

- **Es *portable***. No necesita realizar ninguna adaptación para llamar a funciones de un programa escrito en lenguaje C. En particular, será útil para hacer pruebas de envíos de mensajes entre los procesos del programa GSP y del programa BROADCAST escrito en C. En la Sección 3.1.2, detallaremos estas pruebas de comunicación.

- **El uso de *Mónadas***. Es compatible con estructuras que representan cálculos definidos como una secuencia de pasos. Estas estructuras se conocen como mónadas. Estos cálculos pueden pensarse como funciones, que además de tener una entrada y una salida, también producen un efecto. Las mónadas proporcionan una abstracción de los efectos y ayudan a garantizar que los efectos sucedan en un orden controlado. En particular, usaremos la mónada `Maybe` para implementar aquellas funciones que son parciales, es decir, funciones que a veces devuelven un valor `a` (`Some a`) y a veces ninguno (`None`).

## 3.1. Core GSP

Presentamos una versión *naive* llamada CORE GSP, donde cada cliente interactúa con su estado local realizando operaciones de lectura y escritura. A pesar de ser un modelo simple, este presenta varios de los problemas con los que nos encontramos al trabajar con sistemas que tienen datos replicados como por ejemplo: (i) no garantiza que dos lecturas sucesivas devuelvan los mismos valores, ni (ii) que varias escrituras se ejecuten de forma atómica (es decir, otro cliente puede observar parcialmente los efectos de una secuencia de escrituras).

Para crear esta versión, diseñamos su arquitectura que consta de un servidor y tantos clientes como se desee, creamos distintos módulos de OCaml para implementarla de forma organizada, y por último adaptamos una librería que implementa la primitiva RTOB para la propagación de mensajes.

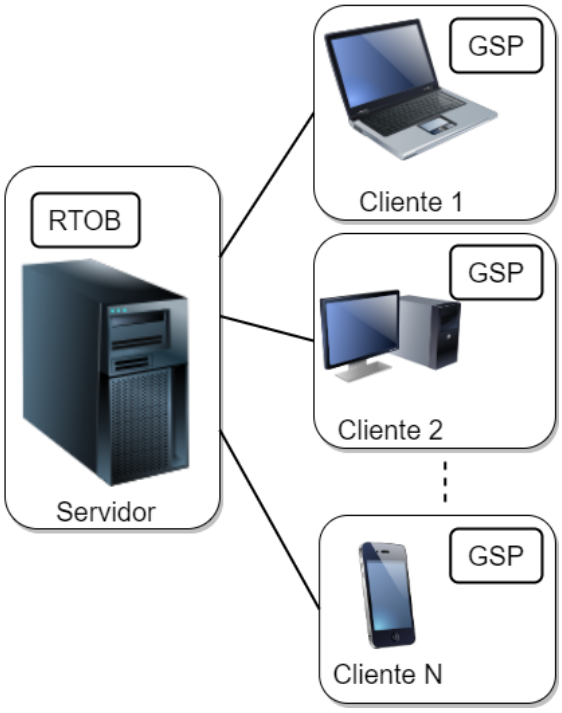


*Fig. 3.1:* Arquitectura de CORE GSP: servidor - clientes.

### 3.1.1. Arquitectura

En la figura 3.1 se representa la arquitectura de CORE GSP. Por un lado contiene un número de clientes tan grande como se desee, que contienen la implementación de CORE GSP. Por otro lado, un servidor que mantiene en ejecución la implementación de RTOB para propagar las actualizaciones de los clientes. Por último, las líneas representan las conexiones, es decir, todos los clientes se encuentran conectados con el servidor, y no entre ellos.

A pesar de que en la Sección 2.4 mencionamos que en CORE GSP no hay ningún servidor que interviene en la conexión, el programa que usamos para implementar RTOB requiere ser ejecutado en un servidor al que todos los clientes tengan acceso.

Simulamos esta arquitectura desde una única computadora, utilizando el software de virtualización Oracle VM VirtualBox [40]. Para esto, creamos una máquina virtual para el servidor, y una para cada cliente, cargando en todas ellas una imagen de disco con Ubuntu[1] (un sistema operativo open-source basado en Linux) descargada de OSBoxes[2] (un sitio que ofrece instancias de sistemas operativos basados en Linux).

Luego, las configuramos para restringir las conexiones entre ellas, de manera que los clientes no puedan conectarse entre sí, sin pasar por el servidor. Para esto, creamos una sola interfaz de red para cada cliente, conectada únicamente con una interfaz de red del servidor. En otras palabras, el servidor tiene tantas interfaces de red como clientes haya, y cada una de ellas está conectada con un único cliente.

Esta simulación demuestra que nuestra implementación puede ser utilizada en múltiples

[1] `https://ubuntu.com`
[2] `https://osboxes.org`

dispositivos. Sin embargo, nuestra versión fue implementada para ser ejecutada en una sola computadora, lanzando por cada cliente un proceso de CORE GSP y un proceso de RTOB.

### 3.1.2. Implementación

La implementación de CORE GSP está dada a partir de los módulos *client.ml*, *helpers.ml* con sus respectivas interfaces *client.mli*, *helpers.mli*, de la interfaz *types.mli* y del módulo *main.ml*.

La interfaz *types.mli* contiene la definición de los distintos tipos de datos que se usan en el programa, *client.ml* contiene las funciones necesarias para que el cliente pueda realizar lecturas y escrituras, y *helpers.ml* contiene funciones auxiliares que usamos en *client.ml*. Por último, el módulo principal *main.ml* contiene las funciones que permiten la interacción entre el usuario y CORE GSP para poder observar el funcionamiento del protocolo, realizando pruebas y obteniendo sus resultados.

Instanciamos el modelo de datos abstracto con un registro que almacena un número entero, basándonos en el Ejemplo 2.4.3, donde cada vez que un cliente realiza una operación de escritura se pisa el valor anterior.

A continuación, detallaremos la interfaz *types.mli* y el módulo *client.ml* para comprender el funcionamiento de CORE GSP.

Primero creamos la interfaz *types.mli* donde definimos los siguientes tipos de datos:

```
type client = int
type value = int
type update = value
type read = unit
type round = { origin: client; number: int; update: update }
type clientGSP = { id: client; known: round list; pending: round list;
    round: int }
```

donde:

- `client` es el tipo de datos que representa al número identificador del cliente. En nuestra versión usamos un número entero.
- `value`, `update` y `read` son los tipos de datos abstractos que definen el tipo de la información que se comparte entre los clientes:
  - `value` define el tipo de la información compartida. Como en nuestra versión creamos un registro que almacena un número entero, lo instanciamos con el tipo `int`.
  - `update` define las operaciones de escritura. Como en nuestra versión las actualizaciones se realizan escribiendo un nuevo número entero (pisando el valor anterior), lo instanciamos con el tipo `int`.
  - `read` define las operaciones de lectura. Como en nuestra versión las lecturas no requieren de ningún parámetro, ignoramos este valor (lo instanciamos con el tipo `unit` que contiene únicamente la constante vacía). Si instanciáramos el modelo de datos abstracto con el almacenamiento *key-value* (como se muestra en el Ejemplo 2.4.2), necesitaríamos instanciar `read` con el tipo de las claves.

De esta manera, al realizar una operación de lectura, el valor de tipo `read` especificaría el valor que deseamos leer.

- `round` es el tipo de los objetos que encapsulan las actualizaciones para ser propagadas a todos los clientes, donde `origin` contiene el identificador del cliente, `number` el número de round enviado por el cliente, y `update` la actualización.
- `clientGSP` es el tipo de los objetos que identifican el estado de un cliente, donde `id` es el número identificador del cliente, `known` es la lista de rounds que contiene el prefijo de la secuencia global conocido por el cliente, `pending` es la lista de rounds que contiene aquellos enviados pero aún no confirmados por el servidor, y `round` contiene el número de rounds enviados al servidor por el cliente.

Luego, para crear el modulo *client.ml*, comenzamos definiendo las siguientes funciones en la interfaz *client.mli*:

```
val client_create: client -> clientGSP
val client_read: clientGSP -> read -> value option
val client_update: clientGSP -> update -> clientGSP
val onReceive: clientGSP -> round -> clientGSP
```

donde:

- `client_create` crea un nuevo objeto de tipo `clientGSP` con el número identificador del cliente pasado por parámetro.
- `rvalue` es una función local del módulo *client.ml* que determina el comportamiento del tipo de datos, es decir, especifica lo que retornará una lectura a partir del conjunto de escrituras conocidas por el cliente y de un valor de tipo `read`. Para implementar un registro, definimos esta función de manera que devuelva el último valor escrito (la última escritura entre las realizadas por el cliente, y las recibidas de otros clientes) ignorando el parámetro `read`.
- `client_read` devuelve el resultado de invocar una lectura a partir del estado actual del cliente y de un parámetro de tipo `read`. Cuando un cliente ejecuta la función `client_read` combinamos las escrituras de las listas `pending` y `known`, y retornamos el resultado de aplicar `rvalue` a esa combinación de listas. De esta manera, un cliente puede obtener sus escrituras desde el momento en el que las invoca, y no debe esperar a que el servidor las confirme (*Read My Writes* [7]). En nuestra versión, si hay algún elemento en `pending` devolvemos el primero, y sino, devolvemos el primer elemento de `known`.
- `client_update` crea un nuevo estado para el cliente a partir del estado anterior, encapsulando la actualización pasada por parámetro en un `round`, agregando ese `round` en la lista `pending`, enviando el `round` por RTOB, e incrementando el contador `round` de rounds enviados.
- `onReceive` crea un nuevo estado para el cliente a partir del estado anterior, agregando una actualización recibida a la lista de actualizaciones conocidas, es decir, a la lista `known`. Además, si el `round` que recibe fue enviado por él mismo, lo eliminamos

de la lista `pending` para tener registrado que ya fue confirmado por el servidor. En nuestra implementación, ejecutamos la función `onReceive` cuando un cliente recibe alguna actualización.

### 3.1.3. Broadcast

Presentamos BROADCAST, un programa implementado por [29] que provee una interfaz con funciones de envío y recepción de mensajes, implementado en C sobre OPEN-MPI [21]. OPEN-MPI es una librería diseñada para ser usada en programas que funcionan con múltiples procesos, cuya función es el envío de mensajes entre ellos.

A continuación describimos los cambios que realizamos sobre BROADCAST para adaptarlo a nuestra implementación:

- **Compilación y ejecución del programa BROADCAST** Para compilar un programa implementado sobre OPEN-MPI, necesitamos ejecutar el compilador `mpicc`. Análogamente, para ejecutarlo, necesitamos ejecutar el programa `mpirun` enviando su ruta como parámetro e indicando la cantidad de procesos que deseamos ejecutar, entre los cuales se enviarán los mensajes. Por lo tanto, desarrollamos un archivo `makefile` para automatizar la instalación de dependencias y compilación del programa, y un archivo `run.sh` para automatizar su ejecución. Destacamos que BROADCAST corre un proceso por cada cliente con el que desea interactuar, y cada cliente debe comunicarse con ese proceso para la propagación de mensajes.
- **Comunicación entre BROADCAST y CORE GSP** Como BROADCAST está desarrollado en C sobre OPEN-MPI, su ejecución debe ser independiente de la ejecución de CORE GSP. Por lo tanto, la comunicación entre ambos programas no puede depender de las librerías de OCaml que resuelven la comunicación entre distintos procesos. A continuación, detallaremos las pruebas que realizamos para lograr esta comunicación:
  - **Comunicación directa entre procesos** En C, `scanf()` es una función que se bloquea esperando recibir datos de un origen determinado. Probamos usar esta función para bloquear BROADCAST esperando recibir los mensajes a propagar de alguno de los procesos de CORE GSP. Sin embargo, al enviar mensajes desde CORE GSP observamos errores de memoria que provocaban el cierre del programa, debido a la incompatibilidad que genera OPEN-MPI (al trabajar con múltiples procesos) con la entrada de datos estándar de C. A partir de esto, resolvimos implementar una memoria compartida entre BROADCAST y CORE GSP que se pueda leer y escribir en cualquier momento.
  - **Memoria compartida** Para implementar una memoria compartida entre BROADCAST y CORE GSP, por cada proceso que lanza BROADCAST creamos dos archivos de texto para que cada cliente envíe y reciba mensajes:
    - Para el envío de mensajes, desde el proceso $i$ que lanza BROADCAST leemos constantemente el archivo *send_i.txt*, y cuando detectamos una nueva línea, la leemos y la enviamos como un mensaje (con origen $i$) a todos los clientes. Por lo tanto, si el cliente $i$ desea propagar un `round` desde CORE GSP, basta con agregarlo al archivo *send_i.txt*.

- Para la recepción de mensajes, si desde el proceso $i$ de BROADCAST recibimos un mensaje, lo agregamos al archivo *receive_i.txt*. Por lo tanto, si el cliente $i$ desea recibir `rounds` propagados desde CORE GSP, basta con leer constantemente el archivo *receive_i.txt*, y procesar como un nuevo `round` cada línea nueva que se detecte.

- **Logueo de actividades en BROADCAST** Inicialmente BROADCAST consumía mucho procesamiento registrando todas las actividades que realizaba. Para un mejor rendimiento, reducimos las operaciones que se registraban, dejando las necesarias para controlar el envío y la recepción de los mensajes.

- **Cierre correcto de BROADCAST** Observamos que todos los procesos lanzados por BROADCAST, luego de cerrarlo, se mantenían en ejecución consumiendo innecesariamente recursos de la computadora. Para evitar esto, modificamos la implementación para capturar el cierre del programa y cerrar todos los procesos en ejecución lanzados por BROADCAST.

- **De procesos a dispositivos** Como mencionamos anteriormente (Sección 3.1.1), simulamos una red con distintos dispositivos creando máquinas virtuales y conectándolas entre sí. Para continuar garantizando que nuestra implementación puede ser utilizada en múltiples dispositivos, demostramos que BROADCAST (que incluye comunicación entre procesos que se ejecutan localmente) también puede enviar y recibir mensajes entre procesos que se encuentren ejecutándose en distintos dispositivos (que se encuentren en una misma red). Para esto, configuramos BROADCAST para que OPEN-MPI envíe y reciba mensajes entre procesos de distintos dispositivos[3], creando una carpeta compartida entre los dispositivos para almacenar los archivos que usamos como memoria compartida.

- **Limitaciones de los mensajes** Cada mensaje que BROADCAST puede enviar y recibir, está limitado a un número entero de 32 bits. Como los mensajes que se envían y reciben desde CORE GSP son `rounds` que incluye tres valores (el número identificador del cliente, el número de round enviado por el cliente y la escritura invocada por el cliente), codificamos el número de round enviado por el cliente, y la escritura invocada por el cliente en un único número entero para enviarlos atómicamente (el número identificador del cliente no fue necesario dado que BROADCAST por cada mensaje que envía, también envía su origen). Comenzamos codificando los dos valores elevando cada valor por un número primo distinto (mayor que 1), y luego multiplicando los dos resultados entre sí, obteniendo un único valor por tupla de valores a codificar. Es decir, siendo $r$ el número de round enviado por el cliente, y $u$ la escritura invocada por el cliente, la siguiente función codifica $r$ y $u$ en un único número entero:

  $$f(r, u) = 2^r * 3^u$$

  Esta función es conocida como la *numeración de Gödel*, y al ser inversible permite obtener $r$ y $u$ a partir del número codificado. Sin embargo, al codificar valores pequeños, la función devuelve valores que exceden el límite de un número de tipo entero y, por lo tanto, debimos descartar esta opción. Finalmente, limitamos a 16

[3] `https://mpitutorial.com/tutorials/running-an-mpi-cluster-within-a-lan`

bits cada valor a enviar, y los concatenamos en un único valor de 32 bits para enviarlos atómicamente. Análogamente, al recibir un mensaje de 32 bits, lo separamos en dos valores de 16 bits obteniendo los valores que habíamos enviado.

## 3.2. Versión robusta

Para extender nuestra implementación de CORE GSP a una versión robusta, implementamos distintas mejoras (que especificamos en la sección 2.4). Para esto, diseñamos una nueva arquitectura basada en la de CORE GSP, con algunas modificaciones necesarias para agregar transacciones y primitivas de sincronización. Por último, dado que esta versión requiere el envío de estructuras más grandes que un `round`, realizamos algunas modificaciones en el programa BROADCAST.

Mostraremos con esta versión que GSP no sólo es un modelo simple, sino que también (i) resuelve fácilmente situaciones en las que ocurren fallas de comunicación, gracias a un servidor que almacena el estado de la información compartida entre todos los clientes, (ii) requiere menor cantidad de memoria al almacenar las secuencias de escrituras en forma reducida y (iii) evita condiciones de carrera gracias a que los clientes cuentan con operaciones para decidir cuándo recibir y enviar las escrituras.

### 3.2.1. Arquitectura

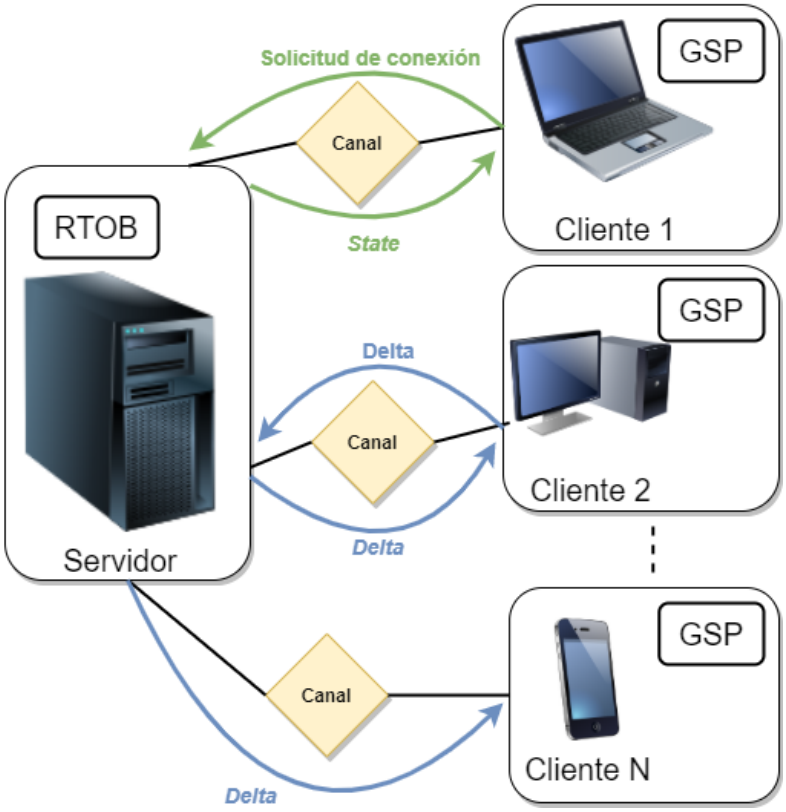


*Fig. 3.2:* Arquitectura de GSP: servidor - clientes.

La Figura 3.2 describe la arquitectura para la versión robusta de GSP. Partimos de la

arquitectura de CORE GSP (Figura 3.1), ya que también contiene un servidor y un número de clientes tan grande como se desee.

Las conexiones entre cada cliente y el servidor las manejamos con canales. Cada vez que un cliente desea enviarle una escritura al servidor, debemos agregarla en el canal establecido con el servidor. Análogamente, cuando el servidor desea propagar una escritura a todos los clientes, debemos agregarla en el canal establecido con cada cliente.

Las flechas verdes representan los mensajes que ocurren para que un cliente nuevo establezca conexión con el servidor. El cliente comienza enviando una solicitud de conexión y el servidor responde un `state` con el prefijo de la secuencia global de actualizaciones que contiene hasta el momento.

A partir de ese momento, el cliente podrá enviar sus escrituras, y recibirá todas las que hagan los clientes. Las flechas azules representan estos dos flujos. Las primeras dos representan el envío de un `delta` con un conjunto de escrituras por parte de un cliente, y la posterior respuesta del servidor, confirmando la recepción con un `delta` que contiene ese conjunto y posiblemente a otras escrituras de otros clientes en forma reducida. La tercera flecha azul representa la recepción de un `delta` por parte de un cliente, que contiene escrituras en forma reducida que enviaron otros clientes.

### 3.2.2. Implementación

Presentamos la implementación de la versión extendida de GSP, para la cual partimos de la implementación de CORE GSP. Esta implementación está dada a partir de los módulos *statedelta.ml*, *channel.ml*, *gsprefix.ml*, *gssegment.ml*, *streamingServer.ml*, *streamingClient.ml*, *helpers.ml* y *maps.ml* con sus respectivas interfaces *statedelta.mli*, *channel.mli*, *gsprefix.mli*, *gssegment.mli*, *streamingServer.mli*, *streamingClient.mli*, *helpers.mli* y *maps.mli*, de la interfaz *types.mli* y del módulo *main.ml*.

La interfaz *types.mli* contiene la definición de los distintos tipos de datos que usamos en el programa, *statedelta.ml* contiene la instancia del modelo de datos abstracto, *channel.ml* contiene las funciones que utilizamos para el envío y recepción de mensajes por parte del servidor y de los clientes, *gsprefix.ml* y *gssegment.ml* contienen las funciones que usamos para el manejo de las formas reducidas en las que el servidor envía la información, *streamingServer.ml* contiene las funciones que necesitamos para que el servidor procese y propague las escrituras que recibe, incorpore nuevos clientes a la red y se recupere frente a situaciones de fallas de conexión, y *streamingClient.ml* contiene las funciones que usamos para que un cliente solicite incorporarse a la red al servidor, para enviar y recibir escrituras.

Además, los módulos *helpers.ml* y *maps.ml* contienen funciones auxiliares que usamos en todos los módulos. Por último, *main.ml* contiene las funciones que necesitamos para garantizar la interacción entre el usuario y GSP, para poder observar el funcionamiento del protocolo realizando pruebas y obteniendo sus resultados.

Para esta versión, definimos dos tipos de datos distintos para instanciar el modelo de datos abstracto: un registro y un acumulador.

A continuación, detallaremos los módulos mencionados para comprender el funcionamiento de la versión extendida de GSP.

#### State y Delta

Primero creamos la interfaz *statedelta.mli* donde definimos los siguientes tipos de datos:

```
type read
type state
type delta
type value
type update

val initialstate: state
val emptydelta: delta

val rvalue: read -> state -> value
val apply: state -> delta list -> state
val append: delta -> update -> delta
val reduce: delta list -> delta
```

donde:

- `value`, `update` y `read` son los tipos de datos abstractos que definimos en CORE GSP, y en esta versión agregamos los tipos de datos `state` y `delta`. Tanto para implementar un acumulador, como para implementar un registro, definimos estos tipos como enteros (salvo `read` que, al igual que en CORE GSP, es un tipo que se ignora).
- `initialstate` es el valor inicial de un nuevo `state`. Tanto para implementar un acumulador, como para implementar un registro, asignamos un 0 al definir este valor.
- `emptydelta` es el valor inicial de un nuevo `delta`. Tanto para implementar un acumulador, como para implementar un registro, asignamos un 0 al definir este valor.
- `rvalue`, al igual que en CORE GSP, especifica lo que retornará una lectura a partir del conjunto de escrituras conocidas por el cliente. A diferencia de CORE GSP, en lugar de recibir como parámetro a ese conjunto en una secuencia, lo recibe en forma reducida en un objeto de tipo `state`. Tanto para implementar un acumulador, como para implementar un registro, definimos esta función para que retorne el mismo `state` recibido por parámetro.
- `apply` es la función que crea un nuevo prefijo de escrituras a partir del prefijo de escrituras anterior y de una secuencia de nuevos segmentos de escrituras. Por un lado, para implementar un acumulador definimos esta función para que genere un nuevo prefijo sumando todos los `delta` de la secuencia recibida por parámetro y el `state` anterior. Por otro lado, para implementar un registro definimos esta función para que retorne la primera escritura de la secuencia de deltas recibida por parámetro, es decir, el último valor escrito.
- `append` es la función que crea un nuevo segmento de escrituras a partir del segmento de escrituras anterior y de una nueva escritura. Por un lado, para implementar un acumulador definimos esta función para que devuelva el resultado de la suma de los dos valores recibidos por parámetro. Por otro lado, para implementar un registro definimos esta función para que devuelva el valor de la escritura recibida por parámetro.

- `reduce` es la función que reduce un conjunto de segmentos de escrituras en un único segmento de escrituras. Por un lado, para implementar un acumulador definimos esta función para que devuelva el resultado de sumar todos los valores de la secuencia de elementos de tipo `delta` recibida por parámetro. Por otro lado, para implementar un registro definimos esta función para que devuelva el primer `delta` de la secuencia recibida por parámetro, es decir, el último `delta` generado.

A continuación, en la implementación usaremos los tipos abstractos `value`, `update`, `read`, `state` y `delta`, y para operar con ellos usaremos las funciones abstractas `rvalue`, `apply`, `append` y `reduce`, y a las constantes `initialstate` y `emptydelta`. Es decir, la instanciación del modelo de datos abstracto se encuentra únicamente en el módulo *statedelta.ml*. De esta manera, para elegir el modelo de datos para usar en el programa (un acumulador o un registro), únicamente debemos cambiar la implementación de ese módulo.

Luego, creamos la interfaz *types.mli* donde definimos los tipos de datos que se observan en la Figura 3.3, donde:

- `round` es igual que en CORE GSP, pero en lugar de contener un `update`, contiene un `delta`.
- `gsSegmentGSP` y `gsPrefixGSP` son estructuras que definen los paquetes que envía el servidor a los clientes. Encapsulan un `delta` o un `state` respectivamente y un diccionario en el que, por cada cliente, almacenamos el máximo número de round que envió y que confirmó el servidor.
- `channelGSP` es el tipo de los objetos que identifican el estado de un canal entre un cliente y el servidor, donde `client` es el identificador del cliente, `clientstream` es la lista en la que agregamos las escrituras que el servidor desea enviarle al cliente, `serverstream` es la lista en la que agregamos las escrituras que el cliente desea enviarle al servidor, `accepted` contiene un valor que maneja el servidor que indica si la conexión con el cliente ya fue aceptada, `receivebuffer` es la lista en la que el cliente almacena los paquetes que recibe del servidor (que pueden ser de tipo `gsSegmentGSP` o `gsPrefixGSP`) y `established` contiene un valor que maneja el cliente que indica si ya procesó el primer paquete que envía el servidor al aceptar la conexión.
- `gs` es un tipo de datos que contiene dos constructores: `GSSegment` y `GSPrefix`. Usamos este tipo de datos para definir los elementos que puede contener la lista `receivebuffer` (de esta manera, puede contener elementos de tipo `gsSegmentGSP` y de tipo `gsPrefixGSP`).
- `streamingServerGSP` es el tipo de los objetos que identifican el estado de un servidor, donde `serverstate` guarda el estado persistente de la información compartida entre los clientes, es decir, el prefijo conocido por el servidor construido a partir de las escrituras emitidas por los clientes, que ya fueron propagadas hacia todos. Por otro lado, `connections` contiene un diccionario en el que, por cada cliente, almacenamos el canal establecido con él.
- `streamingClientGSP` es el tipo de los objetos que identifican el estado de un cliente, donde `id` es el número identificador del cliente, `known` contiene el prefijo de escrituras

conocidas por el cliente, `pending` contiene los rounds enviados por el cliente pero aún no confirmados por el servidor (son confirmados con el diccionario `maxround` que envía el servidor en cada paquete, indicando el número máximo de round que ya procesó de cada cliente), `round` es el contador de rounds enviados por el cliente, `transactionbuf` contiene los rounds con las escrituras del cliente que conforman una transacción y que serán enviados atómicamente al servidor, `tbuf_empty` indica si `transactionbuf` se encuentra vacío, `channel` almacena el canal entre el cliente y el servidor (en caso de existir una conexión establecida), `pushbuf` contiene las transacciones (en forma reducida) en espera de ser enviadas al servidor, y `rds_in_pushbuf` indica el número de transacciones que contiene `pushbuf`.

```
type round = {
        origin : client;
        number : int;
        delta : delta
}

type gsPrefixGSP = {
        state : state;
        maxround : maxround;
}
(* representa un prefijo de la
secuencia de actualización global *)
```

```
type gsSegmentGSP = {
        delta : delta;
        maxround : maxround;
}
(* representa un intervalo de la
secuencia de actualización global *)

type streamingServerGSP = {
        serverstate : gsPrefixGSP;
            (* estado persistente *)
        connections : connections;
            (* estado volátil *)
}
```

```
type gs = GSPrefix of gsPrefixGSP | GSSegment of gsSegmentGSP

type channelGSP = {
        client : client; (* inmutable *)

        (* duplex streams *)
        clientstream : round list; (* de cliente a servidor *)
        serverstream : gs list; (* de servidor a cliente *)

        (* estado de conexión del lado del servidor *)
        accepted : bool; (* si el servidor ha aceptado la conexión *)

        (* estado de conexión del lado del cliente *)
        receivebuffer : gs list;  (* paquetes almacenados localmente *)
        established : bool; (* si el cliente procesó el primer paquete *)
}

type streamingClientGSP = {
        id : client;
        known : state; (* prefijo conocido *)
        pending : round list; (* rounds enviados, pero no confirmados *)
        round : int; (* contador de rounds enviados *)
        transactionbuf : delta;
        tbuf_empty : bool;
        channel : channelGSP option;
            (* conexión actual (o None si no hay ninguna) *)
        pushbuf : delta;
            (* actualizaciones que se pushearon
               pero que aún no se enviaron *)
        rds_in_pushbuf : int;
            (* contador de rounds en el pushbuffer *)
}
```

**Fig. 3.3:** Estructuras que utilizamos en nuestra implementación. Estos tipos se encuentran definidos en el archivo *types.mli*

**Canales**

Luego, para crear el módulo *channel.ml*, primero definimos las siguientes funciones en la interfaz *channel.mli*:

```
val channel_create: client -> channelGSP
val channel_appendToServerstream: channelGSP -> gs -> channelGSP
val channel_appendToClientstream: channelGSP -> round -> channelGSP
```

donde:

- `channel_create` crea un nuevo objeto de tipo `channelGSP` con el número identificador del cliente pasado por parámetro.
- `channel_appendToServerstream` crea un nuevo estado para el canal para que el servidor envíe un paquete a clientes. Para esto, agregamos un objeto de tipo `gs` a la lista `serverstream`, y luego lo enviamos a clientes usando el programa BROADCAST.
- `channel_appendToClientstream` crea un nuevo estado para el canal para que un cliente envíe un paquete al servidor. Para esto, agregamos un `round` a la lista `clientstream`, y luego lo enviamos al servidor usando el programa BROADCAST.

**Prefijo y Segmento**

Continuamos creando los módulos *gsprefix.ml* y *gssegment.ml*. Para esto, comenzamos definiendo las siguientes funciones en la interfaz *gsprefix.mli*:

```
val gsprefix_create: unit -> gsPrefixGSP
val gsprefix_apply: gsPrefixGSP -> gsSegmentGSP -> gsPrefixGSP
```

Y estas otras funciones en la interfaz *gssegment.mli*:

```
val gssegment_create: unit -> gsSegmentGSP
val gssegment_append: gsSegmentGSP -> round -> gsSegmentGSP
```

donde:

- `gsprefix_create` crea un nuevo objeto de tipo `gsPrefixGSP`.
- `gsprefix_apply` crea un nuevo estado para el prefijo para aplicarle nuevas escrituras. Para esto, actualizamos el `state` (aplicando el `delta` recibido por parámetro al `state` del estado anterior, usando la función `StateDelta.apply`) y el diccionario `maxround` (uniendo el del estado anterior con el recibido por parámetro, dejando los valores más altos).
- `gssegment_create` crea un nuevo objeto de tipo `gsSegmentGSP`.
- `gssegment_append` crea un nuevo estado para el segmento para anexarle un nuevo round. Para esto, actualizamos el `delta` (añadiendo el `round` recibido por parámetro al `delta` del estado anterior, usando la función `StateDelta.reduce`) y el diccionario `maxround` (uniendo el del estado anterior con el recibido por parámetro, dejando los valores más altos).

**Servidor**

Luego, para crear el módulo *streamingServer.ml*, definimos las siguientes funciones en la interfaz *streamingServer.mli*:

```
val streamingServer_create: unit -> streamingServerGSP
val streamingServer_acceptConnection: streamingServerGSP -> channelGSP ->
    streamingServerGSP
val streamingServer_processBatch: streamingServerGSP -> streamingServerGSP
val streamingServer_dropConnection: streamingServerGSP -> client ->
    streamingServerGSP
val streamingServer_crashAndRecover: streamingServerGSP -> streamingServerGSP
```

donde:

- `streamingServer_create` crea un nuevo objeto de tipo `streamingServerGSP`.
- `streamingServer_acceptConnection` crea un nuevo estado para el servidor para aceptar una solicitud de conexión de un cliente. Para esto, incluimos un nuevo canal con el cliente en las conexiones del servidor, y le enviamos el prefijo del estado persistente de la información compartida que almacenamos en `known`.
- `streamingServer_processBatch` crea un nuevo estado para el servidor, procesando los rounds que enviaron los clientes. Para esto, se combinan y reducen todos los segmentos recibidos en un único `delta` para agregarlo al prefijo de la secuencia global, y propagarlo a todos los clientes con el programa BROADCAST. Esta función está pensada para que el servidor la invoque constantemente para procesar y propagar todos los paquetes recibidos.
- `streamingServer_dropConnection` crea un nuevo estado para el servidor para modelar la desconexión (o falla) de un canal del lado del servidor (no del lado del cliente, quien puede seguir enviando y recibiendo paquetes hasta notar la pérdida de conexión). Para esto, eliminamos el canal que almacena la conexión activa con el cliente cuyo identificador es recibido por parámetro.
- `streamingServer_crashAndRecover` crea un nuevo estado para el servidor para modelar una falla y recuperación del servidor en la que pierde todo el estado temporal con los clientes (escrituras recibidas pero aún no procesadas), pero conserva el estado persistente (escrituras procesadas). Para esto, eliminamos todos los canales de `connections` en los que almacenamos las conexiones activas con los clientes.

**Cliente**

Por último, para crear el módulo *streamingClient.ml*, empezamos definiendo las siguientes funciones en la interfaz *streamingClient.mli*:

```
val streamingClient_create: client -> streamingClientGSP
val streamingClient_read: streamingClientGSP -> read -> value
val streamingClient_update: streamingClientGSP -> update ->
    streamingClientGSP
val streamingClient_confirmed: streamingClientGSP -> bool
val streamingClient_push: streamingClientGSP -> streamingClientGSP
```

```
val streamingClient_send: streamingClientGSP -> streamingClientGSP
val streamingClient_pull: streamingClientGSP -> streamingClientGSP
val streamingClient_receive: streamingClientGSP -> streamingClientGSP
val streamingClient_dropConnection: streamingClientGSP ->
    streamingClientGSP
val streamingClient_sendConnectionRequest: streamingClientGSP ->
    streamingClientGSP
```

donde:

- `streamingClient_create` crea un nuevo objeto de tipo `streamingClientGSP` con el número identificador pasado por parámetro.
- `streamingClient_read` devuelve el resultado de invocar una lectura a partir del estado actual del cliente. Para esto, combinamos los segmentos contenidos en `pending`, `pushbuf` y `transactionbuf`, y los aplicamos al prefijo `known`, generando un `state` que combina las actualizaciones confirmadas por el servidor, con las invocadas por el cliente pero aún no confirmadas por el servidor. Al igual que en CORE GSP, se recibe como parámetro un valor definido por el tipo abstracto `read` que especifica lo que se desea obtener en la lectura. Por último, retornamos el valor que obtenemos al invocar a la función `Statedelta.rvalue` enviando como parámetros al `state` generado y al `read` recibido por parámetro.
- `streamingClient_update` crea un nuevo estado para el cliente para agregar una nueva escritura. Para esto, obtenemos un nuevo `transactionbuf` invocando a la función `Statedelta.append` enviando como parámetros al `transactionbuf` anterior y al nuevo `update`.
- `streamingClient_confirmed` notifica si todas las actualizaciones realizadas por el cliente fueron confirmadas por el servidor. Para esto, comprobamos si hay actualizaciones pendientes en `pending`, en `transactionbuf` o en `pushbuf`. Si no hay ninguna, retornamos `true`.
- `streamingClient_push` crea un nuevo estado para el cliente para enviar la transacción `transactionbuf` (que contiene escrituras del cliente) atómicamente al servidor. Para esto, asignamos en `pushbuf` el resultado de invocar a la función `Statedelta.reduce` enviando como parámetro una lista que contiene los deltas de `transactionbuf` y de `pushbuf`.
- `streamingClient_send` crea un nuevo estado para el cliente para enviar las transacciones en espera de ser enviadas al servidor (aquellas "pusheadas"). Para esto, agregamos a la lista `clientstream` del canal un nuevo `round` con el `delta` que contiene `pushbuf` (para enviarlo al servidor), agregamos ese mismo `delta` a `pending` (para indicar que se espera confirmación del servidor), y vaciamos `pushbuf` (asignando `Statedelta.emptydelta`). Luego, propagamos el nuevo `round` usando el programa BROADCAST.
- `streamingClient_pull` crea un nuevo estado para el cliente para procesar todos los paquetes recibidos del servidor. Para esto, verificamos la propiedad `receivebuffer` del canal. Si hay algún paquete, verificamos su tipo leyendo el atributo `established`:

- Si no es el primer paquete (`established = true`), es un `gsSegmentGSP` que contiene un `delta` con nuevas escrituras. Lo agregamos al prefijo conocido `known`, y leemos su propiedad `maxround` para determinar los rounds del cliente que ya confirmó el servidor. Si contiene rounds que el cliente está esperando que el servidor confirme, los eliminamos de `pending`.
- Si es el primer paquete (`established = false`), es un `gsPrefixGSP` que contiene el último estado del prefijo que almacena el servidor. Lo asignamos a `known` y establecemos `established` en verdadero. Como el cliente pudo haber estado conectado anteriormente (enviando y recibiendo paquetes del servidor), debemos asegurarnos de reanudar la transmisión de rounds a partir del round correcto (para evitar perder o duplicar rounds). Para esto, leemos su propiedad `maxround` para determinar los rounds del cliente que ya confirmó el servidor. Si contiene rounds que el cliente está esperando que el servidor confirme, los eliminamos de `pending`. Si el cliente ya tenía rounds en `pending` esperando confirmación y no llegaron en el prefijo, los reenviamos al servidor.

- `streamingClient_receive` crea un nuevo estado para el cliente para recibir un nuevo paquete del servidor. Para esto, movemos un paquete enviado por el servidor (contenido en la lista `serverstream` del canal) a `receivebuffer`.

- `streamingClient_dropConnection` crea un nuevo estado para el cliente para modelar la desconexión (o falla) de un canal del lado del cliente (no del lado del servidor, quien puede seguir enviando y recibiendo paquetes hasta notar la pérdida de conexión). Para esto, eliminamos el canal que almacena la conexión activa con el servidor.

- `streamingClient_sendConnectionRequest` crea un nuevo estado para el cliente, para solicitarle conectarse a la red al servidor. Para esto, creamos un nuevo canal y le enviamos una solicitud al servidor usando el programa BROADCAST.

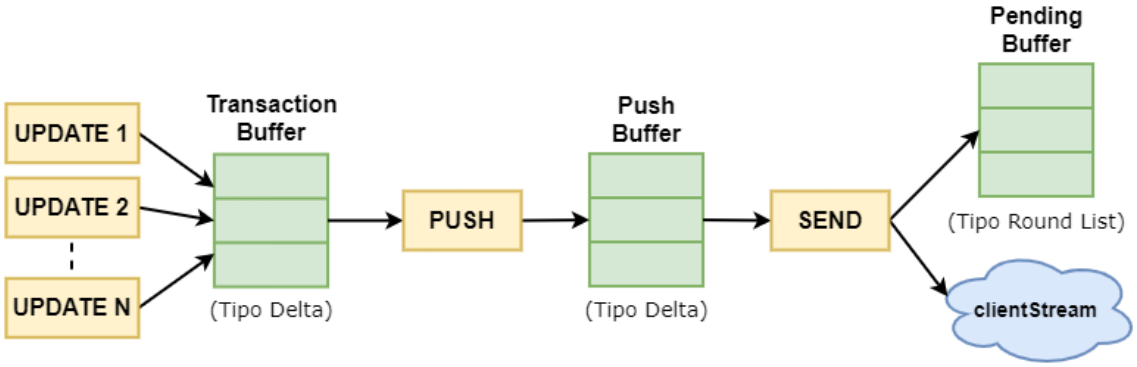


Fig. 3.4: Diagrama de flujo de una actualización hecha por un cliente en GSP.

Para facilitad la comprensión, mostramos en la Figura 3.4 el flujo de operaciones que realiza un cliente desde que crear una escritura hasta que la envía al servidor. Los rectángulos amarillos representan las operaciones, mientras que los verdes son los *buffers* por los que pasan las escrituras. Debajo de cada *buffer* indicamos el tipo de las operaciones

que almacena. Por cada operación, cada escrituras es almacenada en un *buffer* distinto. Por lo tanto, el nombre del *buffer* en el que se encuentra la escritura indica su estado:

- En *Transaction Buffer* (`transactionbuf`) almacenamos en forma reducida (tipo `delta`) las escrituras que realiza el cliente (invocando la función `streamingClient_update`).
- En *Push Buffer* (`pushbuf`) almacenamos en forma reducida (tipo `delta`) las transacciones que "pushea" el cliente (invocando la función `streamingClient_push`) que se encontraban en *Transaction Buffer* (`transactionbuf`).
- En *Pending Buffer* (`pending`) almacenamos los rounds (tipo `round list`) que envía el cliente (invocando la función `streamingClient_send`) que se encontraban en *Push Buffer* (`pushbuf`), y los agregamos al canal (`clientstream`) para enviarlos al servidor usando BROADCAST. Por último, cuando el servidor confirma haber recibido esos rounds, los borramos de *Pending Buffer* (`pending`).

### 3.2.3. Broadcast

Dado que BROADCAST sólo puede enviar y recibir un número entero de 32 bits, y que a diferencia de CORE GSP el servidor envía paquetes de tipo `gsPrefixGSP` y `gsSegmentGSP` que cuentan con atributos distintos que un `round`: un `state` o un `delta` (que en nuestra versión es un entero) y un diccionario (que en nuestra versión es un par de enteros `(id del cliente, máximo round)` por cada cliente), limitamos nuestra versión a dos clientes para que cada paquete que envía el servidor contenga (i) un entero para el `state` o `delta` y (ii) un par de enteros `(máx round del cliente 1, máx round del cliente 2)` para el diccionario `maxround`.

Además, a diferencia de CORE GSP donde todos los mensajes eran enviados a todos los clientes, en esta versión los mensajes del servidor pueden estar destinados a un único cliente (en caso de ser de tipo `gsPrefixGSP`).

Por lo tanto, codificamos los mensajes que envía el servidor de la siguiente manera:

- 1 bit para indicar si el paquete es un `gsPrefixGSP` o un `gsSegmentGSP`.
- 1 bit para indicar el cliente destino. Tomamos en cuenta este bit si el paquete enviado es de tipo `gsPrefixGSP` ya que se lo enviamos al cliente que solicita conectarse a la red. Si el paquete es de tipo `gsSegmentGSP`, lo ignoramos.
- 10 bits para almacenar el `delta` o `state`.
- 10 bits para almacenar el máximo `round` del cliente 1.
- 10 bits para almacenar el máximo `round` del cliente 2.

De esta manera, enviamos 5 valores distintos en un entero de 32 bits.

Por otro lado, como BROADCAST tiene la función de propagar los mensajes hacia todos los clientes, para usar esta versión de GSP debemos tener en cuenta las siguientes limitaciones:

- Un cliente sólo debe aceptar aquellos mensajes cuyo emisor es el servidor. Si el mensaje es un segmento, debe aceptarlo siempre. Si el mensaje es un prefijo, debe aceptarlo únicamente si el destino del mensaje.

- El servidor debe aceptar todos y únicamente los mensajes de los clientes (solicitudes de conexión y actualizaciones).

- El servidor, al aceptar una solicitud de conexión de un cliente, debe propagar un prefijo indicando que en el mensaje que ese cliente es el único destino.

## 3.3. Librerías y dependencias

Para implementar ambas versiones de GSP, usamos las librerías Opam, Inotify, Merlin y Oasis. A continuación detallamos el rol de cada una:

- **Opam** es un gestor de paquetes para OCaml. Gracias a este gestor, pudimos instalar fácilmente todas las demás librerías.

- **Inotify** es un paquete que permite acceder a la interfaz de monitoreo del sistema de archivos de Linux. Este paquete nos permitió detectar cuándo se realizaban cambios en los archivos de texto que usamos como memoria compartida entre BROADCAST y GSP.

- **Merlin** es un servicio de edición de código que proporciona funciones IDE modernas para OCaml. Este servicio nos permitió realizar la programación en lenguaje OCaml con fluidez.

- **Oasis** es una herramienta para construir bibliotecas y aplicaciones OCaml. Esta herramienta genera un sistema completo de configuración e instalación a partir de un archivo *_oasis* que contiene datos de configuración. Oasis nos permitió automatizar la generación del archivo *makefile* para la compilación de ambas versiones de GSP.

## 3.4. Correcciones sobre la especificación de GSP

A continuación detallamos distintas correcciones sobre la especificación de CORE GSP y su extensión robusta en [11] que realizamos en nuestra implementación:

### 3.4.1. Correcciones para Core GSP

- En el módulo *Client*, en el método `update`, se especifica concatenar en `pending` un `update` cuando `pending` está definido como una lista de elementos de tipo `round`. Por lo tanto decidimos concatenarle el mismo `round` que se construye en la siguiente instrucción para enviarlo usando RTOB.

### 3.4.2. Correcciones para la versión robusta de GSP

- En el módulo *stateDelta*, se define una función abstracta y un tipo abstracto con un mismo nombre: `read`. Tener un tipo y una función con el mismo nombre puede generar problemas en la implementación. Por lo tanto, renombramos la función `read` a `rvalue`, tomando en cuenta que cumple el mismo rol que la función `rvalue` de CORE GSP.

- En el módulo *streamingClient*, en el método `adjust_pending_queue`, se especifica obtener el número de `round` recuperando el valor `pending[0].round`. Sin embargo, como `pending` almacena una lista de elementos de tipo `round`, y un `round` almacena su número en el atributo `number`, cambiamos `round` por `number`.

- En el módulo *GSPrefix*, en el método `apply`, se especifica enviar como segundo parámetro el valor de `s.delta` que es de tipo `delta`, cuando para el segundo parámetro se espera una lista de elementos de tipo `delta`. Por lo tanto, cambiamos `s.delta` a `[s.delta]`, es decir, una lista que únicamente contenga el `delta` que necesitamos enviar como parámetro.

- En el módulo *streamingClient*, en el método `send`, se especifica crear un nuevo `round` asignándole su número en su atributo `round` cuando el nombre del atributo que almacena el número de un `round` es llama `number`. Por lo tanto, cambiamos `round` por `number`.

- En el módulo *streamingClient*, en el método `pull`, se especifica asignar `known = known.append(s)`. Para esto, la función `append` debería ser de tipo `state → gsSegmentGSP → state`. Como no hay ninguna función `append` definida así, llamamos a la función `Statedelta.apply` que es de tipo `state → delta list → state`, enviando como parámetro una lista que contiene únicamente a `delta` del segmento, es decir, `segment.delta`.

- En el módulo *streamingClient*, en el método `deltas`, se especifica que se devuelve un valor de tipo `delta`. Sin embargo, esa función devuelve una lista de elementos de tipo `delta`. Por lo tanto, cambiamos el tipo de retorno de la función a una lista de elementos de tipo `delta`.

- En el módulo *streamingClient*, en el método `push`, se especifica asignar `pushbuf = pushbuf.append(transactionbuf)`. Para esto, la función `append` debería ser una función de tipo `delta → delta → delta`. Como no hay ninguna función `append` definida así, llamamos a la función `reduce` que es de tipo `delta list → delta`, enviando como parámetro una lista que contiene a `pushbuf` y a `transactionbuf`.

# 4. ESCENARIOS DE USO

En este capítulo, presentamos las experimentaciones sobre las garantías de consistencia mencionadas en la sección 2.3 que corrimos en nuestra implementación de GSP con el objetivo de agregarle valor al estudio realizado a partir del modelo formal GSP-CALCULUS en [25].

Para correrlas, desarrollamos el archivo principal *main.ml* (Sección 4.1.1) de nuestra implementación de GSP, que contiene la configuración necesaria para poder usarla ingresando una simple línea de comandos (Sección 4.1.3).

Luego, por cada garantía de consistencia que estudiamos, creamos un archivo donde implementamos los distintos casos de prueba (Sección 4.1.2).

Por último, mostramos empíricamente que GSP cumple con cada una de las garantías de consistencia mencionadas, proponemos escenarios en los que el protocolo, en caso de no cumplir alguna garantía, inmediatamente falla alternando el comportamiento del programa y retornando valores inesperados.

Una vez planteados los casos de prueba, los corrimos en nuestra versión de GSP y exportamos los resultados (Sección 4.2).

## 4.1. Configuración

Para simular los escenarios de uso en nuestra implementación de GSP, a nivel hardware usamos una única computadora con Ubuntu instalado (sistema operativo basado en Linux) simulando cada nodo de la red con un proceso distinto.

A nivel software, desarrollamos el archivo principal *main.ml* con el objetivo de que los clientes y el servidor se encuentren constantemente enviando y recibiendo las escrituras especificadas en cada experimento. Además, para personalizar la ejecución de GSP, realizamos las configuraciones necesarias para que el funcionamiento del programa dependa de ciertos parámetros indicados en la ejecución.

### 4.1.1. Main

En el archivo principal *main.ml* configuramos el siguiente formato para ejecutar GSP con sus parámetros:

```
gsp hostType clientId testType testId.
```

O más detalladamente:

```
gsp [client|server] clientId [readmywrites|monotonicreads|
    nocircularcausality|causalarbitration|causalvisibility] testId
```

Destacamos el uso de *Thread*, un módulo provisto por las librerías de OCaml, con el que creamos un hilo de ejecución para correr constantemente la función `waitForNewMessagesToReceive`. Esta función detecta la llegada de nuevos mensajes en la memoria principal y avisa al hilo principal para que los procese.

Para esto, los hilos de ejecución comparten un objeto de tipo `status`:

```
type status = { mutex : Mutex.t; mutable currentLine : int;
   mutable lines : int; client : string }
```

donde:

- El atributo `client` es una constante que contiene el número identificador del cliente (en caso de ser el servidor, tendrá un 0).
- El atributo `mutex` es una variable de exclusión mutua que usamos para evitar que más de un hilo puedan escribir sobre los atributos del recurso compartido `mutexHandler` (un objeto de tipo `status`).
- El atributo `lines` es un contador que guarda la cantidad de líneas que ya fueron detectadas en la memoria compartida.
- El atributo `currentLine` es un contador que guarda cuántas líneas ya fueron leídas, es decir, cuántos mensajes de la memoria compartida ya fueron procesado.

Cuando la función `waitForNewMessagesToReceive` detecta cambios en el archivo *receive_X.txt* (donde $X$ es el valor del atributo `client`), incrementamos el contador `lines`. Cuando el hilo principal procesa mensajes recibidos, incrementamos el contador `currentLine`.

Usamos estos últimos dos atributos en el hilo principal para detectar si hay nuevos mensajes en la memoria compartida (cuando `lines` es mayor que `currentLine`, hay nuevos mensajes para procesar). Además, vale la pena mencionar que ambos son mutables para lograr que ambos hilos puedan modificar sus valores.

Por otro lado, creamos las funciones `serverPolling` y `clientPolling` que el servidor y los clientes, respectivamente, se encuentran ejecutando a lo largo del funcionamiento del programa. La función `serverPolling` ejecuta constantemente la función `streamingServer_processBatch`, salvo cuando detectamos la llegada de un nuevo mensaje (en este caso, procesamos el mensaje y luego continuamos ejecutando constantemente la misma función). La función `clientPolling` ejecuta constantemente la función `streamingClient_pull`, salvo cuando detectamos la llegada de un nuevo mensaje (al igual que el servidor), o que el parámetro `testType` que ingresa el usuario al ejecutar el programa contenga algún valor (en este caso, ejecutamos el test completo leyendo también el valor del parámetro `testId`, y luego continuamos ejecutando constantemente la misma función).

Por lo tanto, cuando un servidor ejecuta el programa, creamos una instancia del objeto `streamingServerGSP` con la función `streamingServer_create`, y luego llamamos a la función `serverPolling` usando esa instancia del servidor como parámetro. Cuando un cliente ejecuta el programa, creamos una instancia del objeto `streamingClientGSP` con la función `streamingClient_create` con el número identificador del cliente pasado por parámetro, y luego invocamos a la función `clientPolling` usando esa instancia del cliente como parámetro.

### 4.1.2. Tests

Para desarrollar los experimentos, creamos los siguientes módulos:

`test_readmywrites.ml, test_monotonicreads.ml,`

`test_consistentprefix.ml, test_nocircularcausality.ml, test_causalarbitration.ml, test_causalvisibility.ml`

en los que implementamos funciones cuyos nombres comparten el siguiente formato:

`[CONSISTENCIA]_test[NUMTEST]_c[CLIENTE]_[PASO]`

donde:

- `CONSISTENCIA` es la garantía de consistencia que estudiamos con ese test.
- `NUMTEST` es el número de test para esa garantía de consistencia.
- `CLIENTE` es el número del cliente que debe invocar ese test.
- `PASO` indica el orden de las funciones que debe invocar ese cliente para ese test.

Por ejemplo:

`readMyWrites_test1_c1_1`

es la primera función que debe invocar el cliente 1, para correr el test 1 que sirve para estudiar la garantía de consistencia *Read My Writes*.

### 4.1.3. Uso del programa

Dentro de la carpeta `release` creamos las carpetas `broadcast` y `gsp` para organizar los archivos que ejecutamos para correr los tests. La carpeta `broadcast` contiene el ejecutable de BROADCAST, y la carpeta `gsp` contiene el ejecutable GSP y un script ejecutable que nombramos `run.sh`. Cuando ejecutamos este script, se abren distintas terminales para correr ambos ejecutables. En la primera terminal corremos BROADCAST. En otras dos terminales corremos dos instancias de GSP representando dos clientes.

El formato para ejecutar `run.sh` es el siguiente:

```
run.sh testType testId
```

O más detalladamente:

```
run.sh [readmywrites|monotonicreads|nocircularcausality|
    consistentprefix|causalarbitration|causalvisibility] testId
```

donde:

- `testType` es la garantía de consistencia que vamos a estudiar. Esta opción puede ser `readmywrites` para demostrar *read my writes*, `monotonicreads` para demostrar *monotonic reads*, `consistentprefix` para demostrar *consistent prefix*, `nocircularcausality` para demostrar *no circular causality*, `causalarbitration` para demostrar *causal arbitration* y `causalvisibility` para demostrar *causal visibility*.
- `testId` es el número del cliente que debe invocar a ese test.

## 4.2. Experimentos

Presentamos los distintos escenarios de uso que desarrollamos y ejecutamos con GSP para observar empíricamente si cada garantía de consistencia se cumple, y cuál es su dependencia con respecto a la primitiva de *broadcast* RTOB.

Observamos que en los resultados de los experimentos, siempre ocurre que ambos clientes comienzan imprimiendo `ACCEPT` y `PULL`. Esto ocurre para indicar que el cliente fue aceptado por el servidor al solicitar establecer conexión, recibiendo el prefijo de la secuencia global. Por lo tanto, debe invocar la operación `PULL` para procesarlo y comenzar a transmitir y recibir nuevas escrituras.

Por otro lado, aclaramos que en los resultados imprimimos `PULL` únicamente cuando el cliente invoca a esa operación y encuentra información para procesar. Si un cliente invoca `PULL` y no hay información nueva para procesar, no se imprime en los resultados del experimento.

Además, usaremos como notación `C1` y `C2` para referirnos al cliente 1 y 2 respectivamente.

### 4.2.1. Read My Writes

Recordemos la definición de esta garantía de consistencia:

$$ReadMyWrites \stackrel{def}{=} (so \subseteq vis)$$

Cuando realizamos una operación de lectura invocando la función `streamingClient_read`, construimos el estado de la información compartida del cliente a partir del prefijo conocido por el cliente contenido en `known`, de los elementos de tipo `delta` de la lista `pending`, del `delta` contenido en `pushbuf` y del `delta` contenido en `transactionbuf`. De esta manera, la lectura se basa en un estado que incluye las escrituras realizadas aún no "pusheadas", las "pusheadas" aún no enviadas al servidor y las confirmadas por el servidor. Esto nos asegura que la garantía de consistencia *Read My Writes* siempre se cumple.

A modo de ejemplo, planteamos un experimento en el que `C1` realiza una escritura y la envía al servidor. Desde que hace la escritura, entre cada par de operaciones necesarias para enviarla al servidor, invoca lecturas para observar si siempre lee el valor que escribió.

Por otro lado, `C2` invoca las operaciones `PULL` y `READ` reiteradas veces para poder observar en qué momento el servidor propagó la escritura de `C1`. De esta manera obtenemos un intervalo de tiempo que comienza en el instante en el que `C1` realizó la escritura y termina en el instante en el que `C2` recibió esa escritura, y observamos que durante todo ese intervalo `C1` leyó su escritura.

La tabla 4.1 muestra las ejecuciones que realizamos en el experimento (en el archivo *test_readmywrites.ml* se encuentra la implementación), mientras que la tabla 4.2 muestra su resultado. Como esperábamos, `C1` lee su escritura desde el momento en el que la invoca, y las lecturas de `C2` muestran que la propagación se realizó luego de que `C1` lea su propia escritura:

*Tab. 4.1:* Experimento N° 1

| C1 | C2 |
|---|---|
| W10 | PULL |
| READ | READ |
| PUSH | PULL |
| READ | READ |
| SEND | ⋮ |
| READ | ⋮ |
| | ⋮ |
| | ⋮ |

*Tab. 4.2:* Resultado N° 1

| C1 | C2 |
|---|---|
| | ACCEPTED |
| | PULL |
| ACCEPTED | |
| PULL | |
| W10 | |
| R10 | |
| PUSH | |
| R10 | |
| SEND | |
| R10 | |
| | R0 |
| | R0 |
| | PULL |
| | R10 |

### 4.2.2. Monotonic Reads

Recordemos la definición de esta garantía de consistencia:

$$MonotonicReads \stackrel{def}{=} (vis; so) \subseteq vis$$

Como acabamos de mencionar, a partir de que un cliente realiza una escritura, sus efectos serán visibles en todas las lecturas que invoque. Por otro lado, cuando un cliente invoca la operación `PULL`, obtiene las escrituras emitidas por otros clientes (que fueron propagadas por el servidor) y las adjunta al prefijo conocido contenido en `known`. En resumen, las escrituras emitidas por ese cliente y las emitidas por otros son almacenadas en las listas a las que accede el cliente para realizar una lectura. Eso nos garantiza que una vez que un cliente reciba una escritura y observe sus efectos, no existe escenario en el que los pierda. En otras palabras, la garantía de consistencia *Monotonic Reads* siempre se cumple.

A modo de ejemplo, planteamos un experimento usando un acumulador como modelo de datos, en el que `C1` realiza una escritura (sumando 5 en el acumulador), la "pushea" y la envía al servidor.

Por otro lado, `C2` obtiene la escritura de `C1` y luego realiza dos pasos distintos. Primero realiza una escritura (sumando 10 en el acumulador), "pusheándola" y enviándola al servidor, intercalando lecturas entre cada par de operaciones. Y luego repite esta operación, pero sumando 20, en lugar de 10, en el acumulador. Observamos en este experimento que las escrituras que `C2` va leyendo (incluyendo las propias) se van acumulando, y nunca se pierde ninguna, cumpliendo con la garantía de consistencia *Monotonic Reads*.

La tabla 4.3 muestra las ejecuciones que realizamos en el experimento (en el archivo *test_monotonicreads.ml* se encuentra la implementación), mientras que la tabla 4.4 muestra su resultado:

*Tab. 4.3:* Experimento N° 2

| **C1** | **C2** |
|---|---|
| W5 | WHILE R=0: PULL |
| PUSH | W10 |
| SEND | READ |
| | PUSH |
| | READ |
| | SEND |
| | READ |
| | W20 |
| | READ |
| | PUSH |
| | READ |
| | SEND |
| | READ |

*Tab. 4.4:* Resultado N° 2

| **C1** | **C2** |
|---|---|
| | ACCEPTED |
| | PULL |
| ACCEPTED | |
| PULL | |
| | R0 |
| W5 | |
| PUSH | |
| | R0 |
| PULL | |
| | PULL |
| | R5 |
| | W10 |
| | R15 |
| | PUSH |
| | R15 |
| | SEND |
| | R15 |
| | W20 |
| | R35 |
| | PUSH |
| | R35 |
| | SEND |
| | R35 |

### 4.2.3. Consistent Prefix

Recordemos la definición de esta garantía de consistencia:

$$ConsistentPrefix \stackrel{def}{=} (ar; (vis \cap \neg ss)) \subseteq vis$$

Si desarrollamos la definición, esta garantía nos dice que por cada terna de operaciones $e, f, g$, si ocurre simultáneamente que:

1. $e$ ocurre antes que $f$ en la secuencia global,
2. $f$ fue invocada por el cliente $i$,
3. $g$ fue invocada por el cliente $j$ $(i \neq j)$, y
4. los efectos de $f$ son visibles para $g$.

Entonces los efectos de $e$ serán visibles para $g$.

Para estudiar esta garantía de consistencia creamos el siguiente escenario que cumple con la hipótesis:

Supongamos que un cliente $C_k$ realiza una escritura $W_e$, la "pushea" y la envía al servidor, y que también otro cliente $C_i$ realiza otra escritura $W_f$, la "pushea" y la envía al servidor. El servidor eventualmente las recibe en ese orden y las guarda en `serverstate` donde almacena la secuencia global en forma reducida. Hasta acá se cumplen las primeras dos de las cuatro propiedades de la hipótesis.

Para las últimas dos propiedades, necesitamos que un tercer cliente $C_j$ realice una lectura $R_g$ que vea los efectos de $W_f$. Lo que buscamos responder es: ¿existe algún escenario en el que $C_j$ vea los efectos de $W_f$, pero no los de $W_e$?

La única manera de que $C_j$ vea los efectos de cualquier escritura realizada por otro cliente, es invocando la operación `PULL` en la que recorremos en orden los paquetes recibidos (que contiene `receivebuffer`) y los agregamos a `known` donde el cliente guarda el prefijo conocido. Por lo tanto, podemos cambiar la pregunta que buscamos responder a: ¿existe algún escenario en el que el `receivebuffer` de $C_j$ contenga $W_f$, pero no $W_e$?

Sabemos que el servidor guarda en `serverstate` todas las escrituras que recibe y las propaga usando RTOB, primitiva de broadcast que garantiza que las escrituras son propagadas en orden y que ninguna se pierde. Para este escenario, el servidor propaga $W_e$ y $W_f$ con RTOB, garantizándonos que $C_j$ eventualmente las recibirá y las guardará en `receivebuffer` en ese orden.

Supongamos el mismo escenario pero sin usar RTOB, es decir, sin la garantía de que todas las escrituras se propaguen en orden. Entonces podemos suponer que cuando el servidor propaga la escritura $W_e$, por algún motivo (por ejemplo, por problemas en la conexión) se retrasa su entrega a $C_j$. Por otro lado, podemos suponer que el servidor propaga $W_f$ y $C_j$ lo recibe de inmediato. En este caso, si $C_j$ invoca la operación `PULL` y realiza la lectura $R_g$ verá los efectos de $W_f$, pero no verá los de $W_e$, cumpliendo así todas las propiedades de la hipótesis que plantea *Consistent Prefix* pero negando la consecuencia.

En conclusión, este escenario demuestra que GSP depende de RTOB para cumplir con la garantía de consistencia *Consistent Prefix*.

A modo de ejemplo, planteamos un experimento simulando este escenario para mostrar los resultados, en el que $C_k$ y $C_i$ son el mismo cliente `C1` (notar que esto no contradice la hipótesis) y $C_j$ es `C2`. La operación $W_e$ es `W5` de `C1`, $W_f$ es `W10` de `C1` y $R_g$ es el `READ` de `C2` que ve los efectos de `W10`. Para este experimento usamos un registro como modelo de tipos de datos.

La tabla 4.5 muestra las ejecuciones que realizamos en el experimento (en el archivo *test_consistentprefix.ml* se encuentra la implementación), mientras que la tabla 4.6 muestra su resultado. Además, la figura 4.1 muestra el estado del servidor luego de ejecutar el experimento, donde vemos que `W5` ocurre antes que `W10` en la secuencia global:

*Tab. 4.5:* Experimento N° 3

| C1 | C2 |
|---|---|
| W5 | PULL |
| PUSH | READ |
| SEND | PULL |
| SLEEP | READ |
| W10 | PULL |
| PUSH | READ |
| SEND | ⋮ |

*Tab. 4.6:* Resultado N° 3

| C1 | C2 |
|---|---|
| ACCEPTED | |
| PULL | |
| | ACCEPTED |
| | PULL |
| W5 | |
| PUSH | |
| | PULL |
| | R0 |
| | R0 |
| W10 | |
| PUSH | |
| PULL | |
| | R0 |
| PULL | |
| | R0 |
| | PULL |
| | R10 |
| | R10 |
| | PULL |
| | R5 |
| | R5 |

```
server: {
  serverstate : { state : 10, maxround : [1: 1] },
  connections : [
    1: { client : 1, clientstream : [],
         serverstream : [
           { state : 0, maxround : [] },
           { delta : 5, maxround : [[1: 0]] },
           { delta : 10, maxround : [[1: 1]] }
         ],
         accepted : true, receivebuffer : [], established : false },
    2: { client : 2, clientstream : [],
         serverstream : [
           { state : 0, maxround : [] },
           { delta : 5, maxround : [[1: 0]] },
           { delta : 10, maxround : [[1: 1]] }
         ],
         accepted : true, receivebuffer : [], established : false }
  ]
}
```

Fig. 4.1: Estado del servidor.

### 4.2.4. No Circular Causality

Recordemos la definición de esta garantía de consistencia:

$$NoCircularCausality \stackrel{def}{=} acyclic(hb)$$

Independientemente del orden en el que el servidor y cada cliente envíe y reciba las escrituras, el estado del sistema es representado como una secuencia de actualizaciones almacenada por el servidor que contiene un único orden. A pesar de las anomalías que puedan ocurrir debido a problemas en la conexión, el servidor establece un orden total de las operaciones, y por lo tanto, una operación $W_e$ que fue invocada antes de $W_f$ no puede depender de los valores de $W_f$ ni visualizar sus efectos, y por lo tanto, la relación de causalidad no puede contener ciclos cumpliendo con la garantía de consistencia *No Circular Causality*.

A modo de ejemplo, planteamos un experimento simulando un escenario, usando como modelo de datos un registro, en el que `C1` lee y luego invoca a `W10`, mientras que `C2` lee y luego invoca a `W5`. Para que se cumpla esta garantía de consistencia, alguna de las dos lecturas deben obtener un 0. Si ambas leen 5, 10 o 15, no se estaría cumpliendo esta garantía ya que significaría que existe causalidad circular.

La tabla 4.7 muestra las ejecuciones que realizamos en el experimento (en el archivo *test_nocircularcausality.ml* se encuentra la implementación), mientras que la tabla 4.8 muestra su resultado. Como era de esperarse, uno de los dos clientes obtiene un 0 al invocar una lectura (en particular, en este caso `C1` leyó un 0):

*Tab. 4.7:* Experimento N° 4

| **C1** | **C2** |
|---|---|
| PULL | PULL |
| READ | READ |
| W10 | W5 |
| PUSH | PUSH |
| SEND | SEND |
| SLEEP | SLEEP |
| PULL | PULL |
| READ | READ |
| W10 | W5 |
| PUSH | PUSH |
| SEND | SEND |
| SLEEP | SLEEP |

*Tab. 4.8:* Resultado N° 4

| **C1** | **C2** |
|---|---|
| PULL | |
| | PULL |
| R0 | |
| W10 | |
| PUSH | |
| | PULL |
| | R10 |
| | W5 |
| | PUSH |

### 4.2.5. Causal Arbitration

Recordemos la definición de esta garantía de consistencia:

$$CausalArbitration \stackrel{def}{=} (hb \subseteq ar)$$

Para simular una relación de causalidad, supongamos que contamos con un registro como modelo de datos y con una operación de escritura $W_f$ que fue invocada basada en los valores que escribió otra operación de escritura $W_e$.

Por un lado, si ambas operaciones fueron invocadas por distintos clientes, sabemos que el cliente que realizó $W_f$ tuvo que haber recibido $W_e$. Por ende, sabemos que $W_e$ se encontraba almacenada en el servidor. Por lo tanto, cuando se envíe $W_f$ al servidor, la secuencia global tendrá $W_e$ y luego $W_f$. Es decir, la relación de causalidad estará incluida en la secuencia global del servidor, y por lo tanto, la garantía de consistencia se cumplirá.

Por otro lado, si ambas operaciones fueron invocadas por un mismo cliente, al enviarla al servidor pueden pasar dos cosas:

- Que el cliente envíe ambas operaciones atómicamente en forma reducida, y por lo tanto, el servidor reciba un único paquete con el resultado de ambas escrituras. En este caso, la relación de causalidad estará incluida en la secuencia global del servidor, y por ende, la garantía de consistencia se cumplirá.

- Que el cliente envíe ambas operaciones en dos paquetes distintos. Es decir, en un paquete envía la operación $W_e$, luego invoca la operación $W_f$ cuyo resultado depende del valor que escribió $W_e$ y luego envía otro paquete con la operación $W_f$. En este caso, como RTOB garantiza que el servidor recibirá $W_e$ y $W_f$ en orden, podemos decir que la relación de causalidad se verá incluida en la secuencia global del servidor, y por lo tanto, la garantía de consistencia se cumplirá. Pero, al igual que mencionamos al estudiar *Consistent Prefix*, si no contáramos con la primitiva RTOB, podría ocurrir que el servidor reciba $W_e$ y $W_f$ en orden opuesto, es decir, primero $W_f$ y después $W_e$. En consecuencia, la relación de causalidad no estaría incluida en la secuencia global del servidor, y por ende, la garantía de consistencia no se cumpliría.

A modo de ejemplo, planteamos un experimento simulando este escenario para mostrar los resultados, en el que `C1` invoca la operación de escritura `W5` ($W_e$), luego realiza una lectura y a partir del resultado realiza otra escritura duplicando el valor leído, es decir, `W(READ*2)` ($W_f$). El servidor recibe ambas escrituras en el orden opuesto al que fue enviado por `C1` produciendo que el orden de causalidad no esté incluido en la secuencia global, y por ende, negando la garantía de consistencia. Por otro lado, `C2` invoca las operaciones `PULL` y `READ` reiteradas veces para poder observar que el servidor propaga las escrituras de `C1` en el orden opuesto al que las invocó.

La tabla 4.9 muestra las ejecuciones que realizamos en el experimento (en el archivo *test_causalarbitration.ml* se encuentra la implementación), mientras que la tabla 4.10 muestra su resultado. Además, la figura 4.2 muestra el estado del servidor luego de ejecutar el experimento, donde vemos que `W5` ocurre antes que `W10` en la secuencia global:

*Tab. 4.9:* Experimento N° 5

| C1 | C2 |
|---|---|
| W5 | PULL |
| PUSH | READ |
| SEND | PULL |
| SLEEP | READ |
| W(READ*2) | PULL |
| PUSH | READ |
| SEND | ⋮ |

*Tab. 4.10:* Resultado N° 5

| C1 | C2 |
|---|---|
| ACCEPTED | |
| PULL | |
| | ACCEPTED |
| | PULL |
| W5 | |
| PUSH | |
| | PULL |
| | R0 |
| | R0 |
| W10 | |
| PUSH | |
| PULL | |
| | R0 |
| PULL | |
| | R0 |
| | PULL |
| | R10 |
| | R10 |
| | PULL |
| | R5 |
| | R5 |

```
server: {
  serverstate : { state : 5, maxround : [1: 1] },
  connections : [
    1: { client : 1, clientstream : [],
         serverstream : [
           {state : 0, maxround : [] },
           {delta : 10, maxround : [[1: 0]] },
           {delta : 5, maxround : [[1: 1]] }
         ],
         accepted : true, receivebuffer : [], established : false },
    2: { client : 2, clientstream : [],
         serverstream : [
           {state : 0, maxround : [] },
           {delta : 10, maxround : [[1: 0]] },
           {delta : 5, maxround : [[1: 1]] }
         ],
         accepted : true, receivebuffer : [], established : false }
  ]
}
```

Fig. 4.2: Estado del servidor.

### 4.2.6. Causal Visibility

$$CausalVisibility \stackrel{def}{=} (hb \subseteq vis)$$

Análogamente al caso planteado con *Causal Arbitration*, si un cliente envía dos paquetes distintos con las escrituras $W_e$ y $W_f$ al servidor y este los recibe en orden opuesto, al propagarlos puede ocurrir que otro cliente observe $W_f$ pero no aún $W_e$. En consecuencia, la relación de causalidad no estaría incluida en el orden de visibilidad, y por ende, la garantía de consistencia no se cumpliría.

En el mismo experimento que planteamos con *Causal Arbitration* vemos que el orden de causalidad es $W_e \xrightarrow{hb} W_f$, mientras que el servidor y `C2` visualizan los efectos de esas operaciones en el orden $W_f \xrightarrow{vis} W_e$. Por lo tanto, no se cumple con la garantía de consistencia *Causal Visibility*.

## 4.3. Observaciones

Para concluir esta sección remarcamos que el diseño del protocolo propuesto no es suficiente para garantizar niveles de consistencia como consistent-prefix. Remarcamos entonces, que algunas garantías de consistencia están ligadas al tipo de broadcast que se utiliza:

| **Modelo de Consistencia** | **Dependencia con RTOB** |
|---|---|
| **Read my writes** | No |
| **Monotonic reads** | No |
| **Consistent prefix** | Sí |
| **No circular causality** | No |
| **Causal arbitration** | Sí |
| **Causal visibility** | Sí |

Por lo tanto, a pesar de que la versión robusta de GSP contiene mejoras que permiten: evitar el uso excesivo de memoria, controlar la asincronicidad con las operaciones `PUSH`, `PULL` y `CONFIRMED`, manejar fácilmente las fallas de conexiones tanto de los clientes como del servidor, entre otras, la elección del broadcast juega un rol central en la implementación de GSP.

# 5. CONCLUSIONES

En esta tesis presentamos e implementamos un modelo operacional llamado GSP que permite razonar en términos de aplicaciones que corren en sistemas distribuidos tan grandes como *cloud computing*. La implementación de GSP está directamente ligada al estudio formal llamado GSP-CALCULUS presentado en [25]. Este trabajo propone una versión naive de GSP, llamada CORE GSP, la cual fue comparada con una versión robusta del protocolo. Esta implementación significa una librería open-source desarrollada para trabajar con protocolos que ofrecen niveles de consistencia al menos tan fuertes como la consistencia eventual.

La pregunta inicial, previa a comenzar el trabajo era: *¿Qué consecuencias existen si modificamos el protocolo de broadcast?*, es decir, *¿cómo cambia el comportamiento de* GSP *al cambiar la implementación de broadcast?* Sin embargo, al revisar que la versión robusta de GSP modela la comunicación entre nodos usando sockets (en lugar de mencionarse RTOB y la dependencia sobre esta primitiva), adaptamos la librería de BROADCAST para simular el comportamiento de esos sockets. Por lo tanto, en lugar de estudiar otras primitivas de broadcast, propusimos un análisis empírico sobre la dependencia de las garantías de consistencia con las propiedades que RTOB garantiza.

Aunque en la actualidad contamos con conexiones muy rápidas y eficientes, no podemos fiarnos de que no ocurrirán comportamientos inesperados. Por eso uno de los objetivos de la extensión de CORE GSP a una versión robusta, fue lograr la independencia entre las propiedades de GSP y la primitiva de broadcast que se utilice. Entendimos que para distintas situaciones, como por ejemplo la desconexión del servidor, GSP cuenta con mecanismos para recuperarse fácilmente sin producir errores. Sin embargo, luego de estudiar las garantías de consistencia que busca asegurar GSP y observar que efectivamente dependen de RTOB, creemos que se pueden realizar distintas modificaciones en la especificación de GSP para que todas esas garantías de consistencia se cumplan independientemente de la primitiva de broadcast que se utilice. Por ejemplo, el servidor y los clientes podrían contar con un mecanismo que detecte y resuelva la pérdida o demora de los paquetes enviados que podrían modificar el orden en el que son recibidos.

Estos resultados muestran que las hipótesis planteadas a partir del modelo formal responden a una dependencia de un protocolo directamente ligado al mecanismo de envío de mensajes.

### 5.0.1. Trabajos relacionados

- En los últimos años se han propuesto distintos modelos de consistencia para bases de datos replicadas [5, 23, 10, 38, 3, 4] que hacen diferentes balances entre consistencia y rendimiento, y que sirven para compararlos con GSP.

- También, se han propuesto diferentes alternativas para ayudar al programador a

lidiar con la consistencia eventual. Tal es el caso de QUELEA[37][1], un modelo de programación declarativo que por medio de contratos permite especificar y verificar garantías de consistencia.

- En GSP usamos tipos de datos replicados simples. Sin embargo, si deseamos usar tipos de datos replicados que presenten mayor complejidad, puede resultar difícil poder razonar sobre ellos, entender su semántica, resolver la consistencia eventual e implementarlos. En [36] se propone una plataforma que sirve para especificar tipos de datos replicados usando relaciones entre eventos y para verificar sus implementaciones.
- En [1, 8, 13, 16, 35, 33, 38] se implementan distintos tipos de datos replicados como registros, contadores, conjuntos o listas, con distintas estrategias para resolución de conflictos.
- El sistema Jupiter [28] tiene una estructura de sistema similar (cliente-servidor con streaming bidireccional) al modelo de streaming que usamos para la versión robusta de GSP. Sin embargo, utiliza un algoritmo de transformación operativa (OT) para transformar actualizaciones conflictivas entre sí, en lugar de simplemente ordenar actualizaciones secuencialmente como en GSP.

### 5.0.2. Trabajo futuro

- No encontramos ninguna librería para el uso de primitivas de broadcast que pueda ser descargada y utilizada fácilmente por un programador. Por eso creemos que deben desarrollarse librerías para el uso de distintas primitivas de broadcast (como RTOB). Aunque en nuestra versión usamos la librería BROADCAST, debimos adaptarnos a varias limitaciones como la de trabajar únicamente dos clientes.
- Realizar las correcciones en la especificación de GSP mencionadas en la sección 3.4.
- Ajustes en la especificación GSP para que todas esas garantías de consistencia se cumplan independientemente de la primitiva de broadcast que se utilice.

[1] `http://gowthamk.github.io/Quelea/`

## Bibliografía


[1] *Riak key-value store.* `http://basho.com/products/riak-overview/`.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. The benefits of duality in verifying concurrent programs under tso. In *27th International Conference on Concurrency Theory (CONCUR 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[3] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, pages 163–172. IEEE, 2013.

[4] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Highly available transactions: virtues and limitations (extended version). *arXiv preprint arXiv:1302.0309*, 2013.

[5] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Scalable atomic visibility with ramp transactions. *ACM Transactions on Database Systems (TODS)*, 41(3):1–45, 2016.

[6] Peter Bailis and Ali Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *Commun. ACM*, 56(5):55–63, 2013.

[7] Sebastian Burckhardt. Principles of eventual consistency. *Foundations and Trends in Programming Languages*, 1(1-2):1–150, 2014.

[8] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P Wood. Cloud types for eventual consistency. In *ECOOP 2012*, pages 283–307. Springer, 2012.

[9] Sebastian Burckhardt, Daan Leijen, and Manuel Fahndrich. Cloud types: Robust abstractions for replicated shared state. Technical report, Technical Report MSRTR-2014-43, Microsoft, 2014. `https://research.microsoft.com/apps/pubs/default.aspx?id=211340`.

[10] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually consistent transactions. In *European Symposium on Programming*, pages 67–86. Springer, 2012.

[11] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In *ECOOP 2015*, pages 568–590, 2015.

[12] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.

[13] Neil Conway, William R Marczak, Peter Alvaro, Joseph M Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–14, 2012.

[14] Julieth Paola Barrientos Matute Damian E. Barrios Castillo. Los sistemas distribuidos, bajo el enfoque de cloud computing y sus aplicaciones. 2015.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07*, pages 205–220. ACM, 2007.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[17] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4):372–421, 2004.

[18] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: issues and challenges. In *2010 24th IEEE international conference on advanced information networking and applications*, pages 27–33. Ieee, 2010.

[19] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[20] Alexey Gotsman and Sebastian Burckhardt. Consistency models with global operation sequencing and their composition. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[21] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. Open mpi: A flexible high performance mpi. In *International Conference on Parallel Processing and Applied Mathematics*, pages 228–239. Springer, 2005.

[22] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[23] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Scalable causal consistency for wide-area storage with cops.

[24] Nancy A Lynch. Distributed algorithms. chapter 13, pages 397–450. Elsevier, 1996.

[25] Hernán Melgratti and Christian Roldán. A formal analysis of the global sequence protocol. In *International Conference on Coordination Languages and Models*, pages 175–191. Springer, 2016.

[26] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional programming for the masses*. .ºReilly Media, Inc.", 2013.

[27] David Mosberger. Memory consistency models. *ACM SIGOPS Operating Systems Review*, 27(1):18–26, 1993.

[28] David A Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 111–120, 1995.

[29] Nicoló Rivetti. A project assignement from the 2nd year Master Course "Concurrent and Parallel Programming" at DIAG / Sapienza University of Rome. *Total Order Broadcast implementation over Open MPI.* `https://github.com/Helrohir/tob`.

[30] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *International Conference on Theorem Proving in Higher Order Logics*, pages 391–407. Springer, 2009.

[31] Seungjoon Park and David L Dill. An executable specification, analyzer and verifier for rmo (relaxed memory order). In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 34–41, 1995.

[32] Liladhar R Rewatkar and UL Lanjewar. Implementation of cloud computing on web application. *International Journal of Computer Applications*, 2(8):28–32, 2010.

[33] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.

[34] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys (CSUR)*, 37(1):42–81, 2005.

[35] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer, 2011.

[36] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *SSS 2011*, pages 386–400, 2011.

[37] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *PLDI 2015*, pages 413–424. ACM, 2015.

[38] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400, 2011.

[39] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys (CSUR)*, 49(1):19, 2016.

[40] VirtualBox is a powerful x86 and AMD64/Intel64 virtualization product for enterprise as well as home use. *Oracle VM VirtualBox 6.0.* `https://www.virtualbox.org/`.

[41] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.