

Suscríbete a DeepL Pro para poder editar este documento.  
Entra en www.DeepL.com/pro para más información.

## PORTADA DE LA REVISTA

**CAP Doce**

**Eric Brewer, *Universidad de California, Berkeley***

# Años después: Cómo han cambiado las "reglas"

**El teorema CAP afirma que cualquier sistema de datos compartidos en red sólo puede tener dos de las tres propiedades deseables. Sin embargo, al manejar explícitamente las particiones, los diseñadores pueden optimizar la consistencia y la disponibilidad, logrando así alguna compensación de las tres.**

**I**

n la década transcurrida desde su introducción, los diseñadores e investigadores han utilizado (y a veces abusado) del teorema CAP como motivo para explorar una amplia variedad de sistemas distribuidos novedosos. El movimiento NoSQL

también lo ha aplicado como argumento contra las bases de datos tradicionales.

El teorema CAP establece que cualquier sistema de datos compartidos en red puede tener como máximo dos de las tres propiedades deseables:

* *consistencia* (C) equivalente a tener una única copia actualizada de los datos;
* *alta disponibilidad* (A) de esos datos (para las actualizaciones); y
* tolerancia a las *particiones de* la red (P).

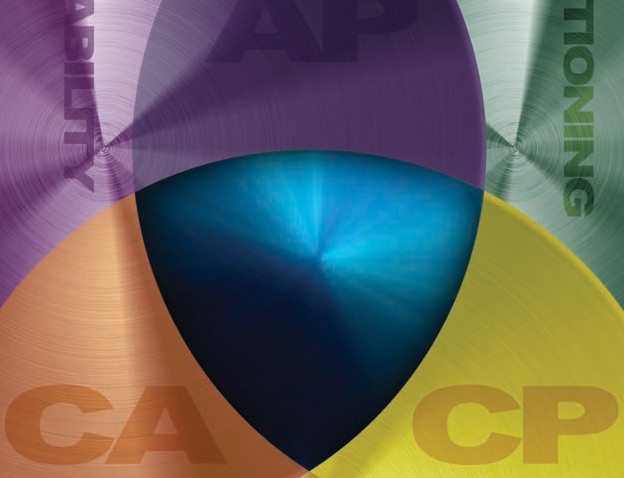
Esta expresión de la PAC cumplió su propósito, que era abrir la mente de los diseñadores a una gama más amplia de sistemas y compensaciones; de hecho, en la última década, ha surgido una amplia gama de nuevos sistemas, así como un gran debate sobre los méritos relativos de la consistencia y la disponibilidad. La formulación "2 de 3" siempre fue engañosa porque tendía a simplificar en exceso las tensiones entre propiedades. Ahora tales nu-

anzas importan. CAP sólo prohíbe una pequeña parte del espacio de diseño: la disponibilidad y consistencia perfectas en presencia de particiones, que son raras.

Aunque los diseñadores todavía tienen que elegir entre consistencia y disponibilidad cuando hay particiones, hay una increíble gama de flexibilidad para manejar las particiones y recuperarse de ellas. El objetivo moderno de CAP debería ser maximizar las combinaciones de consistencia y disponibilidad que tengan sentido para la aplicación específica. Este enfoque incorpora planes para el funcionamiento durante una partición y para la recuperación posterior, ayudando así a los responsables a pensar en CAP más allá de sus limitaciones históricas.

### POR QUÉ "2 DE 3" ES ENGAÑOSO

La forma más fácil de entender CAP es pensar en dos nodos en lados opuestos de una partición. Permitir que al menos un nodo actualice su estado hará que los nodos se vuelvan inconsistentes, perdiendo así C. Asimismo, si la elección es preservar la consistencia, un lado de la partición debe actuar como si no estuviera disponible, perdiendo así A. Sólo cuando los nodos se comunican es posible preservar tanto la consistencia como la disponibilidad, perdiendo así P. La creencia general es que para los sistemas de área amplia, los diseñadores no pueden renunciar a P y, por tanto, tienen que elegir difícilmente entre C y A. En cierto sentido, el movimiento NoSQL consiste en crear opciones que se centran en la disponibilidad en primer lugar y en la consistencia en segundo lugar; las bases de datos que se adhieren a las propiedades ACID (atomidad, consistencia, aislamiento y durabilidad) hacen lo contrario. La barra lateral "ACID, BASE y CAP" explica esta diferencia con más detalle.



#### 0018-9162/12/$31.00 © 2012 IEEE

Publicado por la IEEE Computer Society

#### FEBRERO DE 201223

**PORTADA DE LA REVISTA**

**ÁCIDO, BASE Y TAPA**



**A**

**CID y BASE representan dos filosofías de diseño en los extremos opuestos del espectro consistencia-disponibilidad. Las propiedades ACID se centran en la consistencia y son el enfoque tradicional de las bases de datos. Mis colegas y yo creamos BASE a finales de la década de 1990 para capturar los enfoques de diseño emergentes para la alta disponibilidad y para hacer explícita tanto la elección como el espectro. Los sistemas modernos de área amplia a gran escala, incluida la nube, utilizan una mezcla de ambos**

**enfoques.**

**Aunque ambos términos son más mnemotécnicos que precisos, el acrónimo BASE (al ser el segundo) es un poco más incómodo: Básicamente Disponible, Estado Suave, Eventualmente Consistente. El estado suave y la consistencia eventual son técnicas que funcionan bien en presencia de particiones y, por tanto, promueven la disponibilidad.**

**La relación entre CAP y ACID es más compleja y a menudo se malinterpreta, en parte porque la C y la A de ACID representan conceptos diferentes que las mismas letras en CAP y en parte porque la elección de la disponibilidad afecta sólo a algunas de las garantías de ACID. Las cuatro propiedades de ACID son:**

**Atomicidad (A). Todos los sistemas se benefician de las operaciones atómicas. Cuando el objetivo es la disponibilidad, ambos lados de una partición deberían utilizar operaciones atómicas. Además, las operaciones atómicas de alto nivel (el tipo que implica ACID) realmente simplifican la recuperación.**

**Consistencia (C). En ACID, la C significa que una transacción cumple con todas las reglas de la base de datos, como las claves únicas. En cambio, la C de CAP se refiere sólo a la consistencia de una sola copia, un subconjunto estricto de la consistencia ACID. La consistencia ACID tampoco puede ser mantenida a través de las particiones - la recuperación de la partición necesitará restaurar la consistencia ACID. En general, mantener las invariantes durante las particiones puede ser imposible, por lo que es necesario pensar cuidadosamente en qué operaciones no permitir y cómo restaurar las invariantes durante la recuperación.**

**Aislamiento (I). El aislamiento es el núcleo del teorema CAP: si el sistema requiere aislamiento ACID, puede operar como máximo en un lado durante una partición. La serializabilidad requiere comunicación en general y, por lo tanto, falla a través de las particiones. Las definiciones más débiles de la corrección son viables a través de las particiones mediante la compensación durante la recuperación de la partición.**

**Durabilidad (D). Al igual que con la atomicidad, no hay ninguna razón para renunciar a la durabilidad, aunque el desarrollador podría optar por evitar su necesidad a través del estado suave (al estilo de BASE) debido a su costo. Un punto sutil es que, durante la recuperación de la partición, es posible revertir las operaciones duraderas que, sin saberlo, violaron un invariante durante la operación. Sin embargo, en el momento de la recuperación, dado un historial duradero de ambos lados, tales operaciones pueden ser detectadas y corregidas. En general, la ejecución de transacciones ACID en cada lado de una partición facilita la recuperación y permite un marco para compensar las transacciones que se pueden utilizar para la recuperación de una partición.**

De hecho, esta misma discusión condujo al teorema CAP. A mediados de la década de 1990, mis colegas y yo estábamos construyendo una variedad de sistemas de área amplia basados en clústeres (esencialmente la primera computación en nube), incluyendo motores de búsqueda, cachés proxy y sistemas de distribución de contenidos. 1 Debido a los objetivos de ingresos y a las especificaciones de los contratos, la disponibilidad del sistema era muy importante, por lo que optábamos regularmente por optimizar la disponibilidad mediante estrategias como el empleo de

cachés o registrando las actualizaciones para su posterior reconciliación. Aunque estas estrategias aumentaron la disponibilidad, la ganancia se produjo a costa de una menor consistencia.

La primera versión de este argumento de consistencia frente a disponibilidad apareció como ACID frente a BASE,2 que no fue bien recibida en su momento, principalmente porque la gente adora las propiedades de ACID y duda en renunciar a ellas. El objetivo del teorema CAP era justificar la necesidad de explorar un espacio de diseño más amplio, de ahí la formulación "2 de 3". El teorema apareció por primera vez en otoño de 1998. Se publicó en 19993 y en el discurso de apertura del Simposio sobre Principios de la Computación Distribuida del año 2000,4 que condujo a su demostración.

Como se explica en el recuadro "Confusión de la PAC", la visión "2 de 3" es engañosa en varios frentes. En primer lugar, dado que las particiones son raras, hay pocas razones para renunciar a C o A cuando el sistema no está particionado. En segundo lugar, la elección entre C y A puede producirse muchas veces dentro del mismo sistema con una granularidad muy fina; no sólo los subsistemas pueden hacer elecciones diferentes, sino que la elección puede cambiar según la operación o incluso los datos específicos o el usuario implicado. Por último, las tres propiedades son más continuas que binarias. La disponibilidad es obviamente continua de 0 a 100 por ciento, pero también hay muchos niveles de consistencia, e incluso las particiones tienen matices, incluyendo el desacuerdo dentro del sistema sobre si una partición existe.

Explorar estos matices requiere forzar la forma tradicional de tratar las particiones, que es el reto fundamental. Dado que las particiones son poco frecuentes, CAP debería permitir un C y un A perfectos la mayor parte del tiempo, pero cuando las particiones están presentes o se perciben, es necesaria una estrategia que detecte las particiones y las tenga en cuenta explícitamente. Esta estrategia debería tener tres pasos: detectar las particiones, entrar en un modo de partición explícito que pueda limitar algunas operaciones, e iniciar un proceso de recuperación para restaurar la consistencia y compensar los errores cometidos durante una partición.

### CONEXIÓN DE LATENCIA MÁXIMA

En su interpretación clásica, el teorema CAP ignora la latencia, aunque en la práctica, la latencia y las particiones están profundamente relacionadas. Operativamente, la esencia de CAP tiene lugar durante un tiempo de espera, un periodo en el que el programa debe tomar una decisión fundamental: la *decisión de partición*:

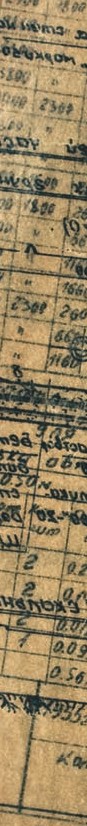
* cancelar la operación y así disminuir la disponibilidad,

*o*

* proceder a la operación y así arriesgarse a la incoherencia.

Reintentar la comunicación para lograr la consistencia, por ejemplo, a través de Paxos o un commit de dos fases, sólo retrasa la decisión. En algún momento el programa debe tomar la decisión; reintentar la comunicación indefinidamente es, en esencia, elegir C en lugar de A.

|  |  |  |
| --- | --- | --- |
|  | |  |
| **CONFUSIÓN DE LA TAPA** | | |
| **A menudo se malinterpretan aspectos del teorema CAP, en particular En la práctica, la mayoría de los grupos asumen que un centro de datos (un solo sitio) no tiene el alcance de la disponibilidad y la consistencia, lo que puede llevar a particiones dentro, y por lo tanto el diseño para CA dentro de un solo sitio; tal**  **resultados indeseables. Si los usuarios no pueden acceder al servicio en absoluto, hay nodosignos , incluidas las bases de datos tradicionales, son la opción por defecto pre-CAP. entre C y A, excepto cuando parte del servicio se ejecuta en elSin embargo , aunque las particiones son menos probables dentro de un centro de datos, sí son posibles. Esta excepción, comúnmente conocida como operación desconectada o son efectivamente posibles, lo que hace que un objetivo de CA sea problemático. Por último, dado el modo sin conexión,1 es cada vez más importante. Algunos HTML5 la alta latencia a través de la zona amplia, es relativamente común renunciar a las características -en particular, el almacenamiento persistente en el cliente- facilitan la consistencia desconectada a través de la zona amplia para mejorar el rendimiento. operación desconectada en el futuro. Estos sistemas suelen Otro aspecto de la confusión de CAP es el coste oculto de renunciar a elegir A en lugar de C y, por tanto, deben recuperarse de las particiones largas. consistencia, que es la necesidad de conocer las invariantes del sistema. El**  **El alcance de la consistencia refleja la idea de que, dentro de algún límite, la belleza sutil de un sistema consistente es que las invariantes tienden a mantener el estado es consistente, pero fuera de ese límite todas las apuestas están fuera. Porque incluso cuando el diseñador no sabe cuáles son. En consecuencia, por ejemplo, dentro de una partición primaria, es posible asegurar completa una amplia gama de invariantes razonables funcionará bien. Por el contrario, la consistencia y la disponibilidad, mientras que fuera de la partición, el servicio no es cuando los diseñadores eligen A, que requiere restaurar invariantes después de un disponible. Paxos y los sistemas de multidifusión atómica suelen coincidir con esta partición, deben ser explícitos sobre todos los invariantes, que son ambos escenarios. 2 En Google, la partición primaria suele residir en un desafiante y propenso a errores. En el núcleo, este es el mismo centro de datos concurrente; sin embargo, Paxos se utiliza en el área amplia para garantizar actualizaciones globales problema que hace que el multithreading sea más difícil que el consenso secuencial, como en Chubby,3 y el almacenamiento duradero altamente disponible, como en la programación.**  **Megastore. 4**  **Los subconjuntos independientes y autoconsistentes pueden avanzarReferencias**  **mientras está particionado, aunque no es posible asegurar la invariabilidad global 1. J. Kistler y M. Satyanarayanan, "Disconnected Operation in the Coda File**  **antos. Por ejemplo, con la fragmentación, en la que los diseñadores preparan el sistema de datos " *ACM Trans. Computer Systems*, Feb. 1992, pp. 3-25.**  **2. K. Birman, Q. Huang y D. Freedman, "Overcoming the 'D' in CAP: Using ress during a partition. Por el contrario, si el estado relevante se reparte entre unaIsis2 para construir servicios en la nube con respuesta local", *Computer*, feb. 2011, pp. Se necesitan particiones o invariantes globales, entonces, en el mejor de los casos, sólo un lado 50-58.**  **puede progresar y, en el peor de los casos, no es posible avanzar. 3. M. Burrows, "The Chubby Lock Service for Loosely-Coupled Distributed**  **Does choosing consistency and availability (CA) as the "2 of 3 " Systems", *Proc. Symp. Operating Systems Design and Implementation* (OSDI**  **¿tiene sentido? Como señalan acertadamente algunos investigadores, exactamente lo que06 ), Usenix, 2006, pp. 335-350.**  **significa renunciar a P no está claro. 5,6 ¿Puede un diseñador optar por no tener parti- 4. J. Baker y otros, "Megastore: Providing Scalable, Highly Available Storage for tions? Si la elección es CA, y luego hay una partición, la elección debeServicios Interactivos ," *Proc. 5th Biennial Conf. Innovative Data Systems* volver a C o A. Es mejor pensar en esto de forma probabilística: elegirInvestigación (CIDR 11), ACM, 2011, pp. 223-234.**  **CA debería significar que la probabilidad de una partición es mucho menor que la 5. D. Abadi, "Problems with CAP, and Yahoo's Little Known NoSQL System", de otros fallos sistémicos, como catástrofes o múltiples simultáneasDBMS *Musings*, blog, 23 abr. 2010;** [**http://dbmsmusings.blogspot.**](http://dbmsmusings.blogspot/) **faults. com/2010/04/problems-with-cap-and-yahoos-little.html.**  **Este punto de vista tiene sentido porque los sistemas reales pierden tanto C como A 6. C. Hale, "You Can't Sacrifice Partition Tolerance", 7 oct. 2010; http://**  **bajo algunos conjuntos de fallos, por lo que las tres propiedades son una cuestión de grado. codahale.com/you-cant-sacrifice-partition-tolerance.** | | |
| Así, pragmáticamente, una partición es un límite de tiempo en la comunicación. Si no se consigue la coherencia dentro del límite de tiempo, se produce una partición y, por tanto, una elección entre C y A para esta operación. Estos conceptos captan la cuestión central del diseño con respecto a la latencia: ¿avanzan las dos partes sin comunicación?  Esta visión pragmática da lugar a varias con- secuencias importantes. La primera es que no hay una noción global de partición, ya que algunos nodos pueden detectar una partición y otros no. La segunda consecuencia es que los nodos pueden detectar una partición y entrar en un *modo de partición, una* parte central de la optimización de C y A.  Por último, este punto de vista significa que los diseñadores pueden establecer límites de tiempo de forma intencionada en función de los tiempos de respuesta deseados; los sistemas con límites más estrictos probablemente entrarán en el modo de partición con más frecuencia y en momentos en los que la red es simplemente lenta y no se particiona realmente.  A veces tiene sentido renunciar a una C fuerte para evitar la alta latencia de mantener la coherencia en una zona amplia. El sistema PNUTS de Yahoo incurre en inconsistencia al mantener | de copias remotas de forma asíncrona. 5 Sin embargo, hace que la copia maestra sea local, lo que disminuye la latencia. Esta estrategia funciona bien en la práctica porque los datos de un solo usuario se dividen naturalmente según la ubicación (normal) del usuario. Lo ideal es que el maestro de datos de cada usuario esté cerca.  Facebook utiliza la estrategia contraria:6 la copia maestra está siempre en una ubicación, por lo que un usuario remoto suele tener una copia más cercana pero potencialmente obsoleta. Sin embargo, cuando los usuarios actualizan sus páginas, la actualización va directamente a la copia maestra al igual que todas las lecturas del usuario durante un breve periodo de tiempo, a pesar de la mayor latencia. Después de 20 segundos, el tráfico del usuario vuelve a la copia más cercana, que en ese momento debería reflejar la actualización.  **GESTIÓN DE LAS PARTICIONES**  El reto para los diseñadores es mitigar los efectos de una partición sobre la consistencia y la disponibilidad. La idea clave es gestionar las particiones de forma muy explícita, incluyendo no sólo la detección, sino también un proceso de recuperación específico y un plan para todas las invariantes que puedan ser violadas durante una partición. Este enfoque de gestión tiene tres pasos: |  |



**PORTADA DE LA REVISTA**

Estado: S Operaciones sobre S

Tiempo

Comienza la partición

Estado: S1

Estado: S2 Modo de partición

Recuperación de particiones

Estado: S'

de la tabla. Por ejemplo, para la invariante de que las claves de una tabla son únicas, los diseñadores suelen decidir arriesgar esa invariante y permitir claves duplicadas durante una partición. Las claves duplicadas son fáciles de detectar durante la recuperación y, suponiendo que puedan fusionarse, el diseñador puede restaurar fácilmente la invariante.

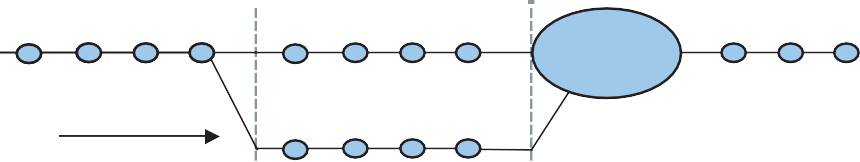
**Figura 1. El estado comienza siendo consistente y permanece así hasta que se inicia una partición. Para permanecer disponibles, ambos lados entran en el modo de partición y continúan ejecutando operaciones, cre**

**de los estados concurrentes S1 y S2, que son inconsistentes. Cuando la partición termina, la verdad se aclara y comienza la recuperación de la partición. Durante la recuperación, el sistema fusiona S1 y S2 en un estado consistente S*'* y también compensa cualquier error cometido durante la partición.**

Para una invariante que debe ser

Sin embargo, si se mantiene durante una partición, el diseñador debe prohibir o modificar las operaciones que puedan violarla. (En general, no hay forma de saber si la operación violará realmente la invariante, ya que

* detectar el inicio de una partición,



* entrar en un modo de partición explícito que puede limitar algunas

operaciones, y

* iniciar la recuperación de la partición cuando se restablece la comunicación.

El último paso tiene como objetivo restaurar la consistencia y compensar los errores que el programa cometió mientras el sistema estaba particionado.

La figura 1 muestra la evolución de una partición. El funcionamiento normal es una secuencia de operaciones atómicas, por lo que las particiones siempre se inician entre operaciones. Una vez que el sistema se agota, detecta una partición, y el lado que la detecta entra en el modo de partición. Si efectivamente existe una partición, ambos lados entran en este modo, pero son posibles las particiones unilaterales. En estos casos, el otro lado se comunica según sea necesario y, o bien este lado responde correctamente o no se requirió ninguna comunicación; en cualquier caso, las operaciones siguen siendo consistentes. Sin embargo, como el lado que detecta podría tener operaciones incongruentes, debe entrar en el modo de partición. Los sistemas que utilizan un quórum son un ejemplo de esta partición unilateral. Un lado tendrá quórum y podrá proceder, pero el otro no. Los sistemas que soportan la operación desconectada tienen claramente una noción de modo de partición, al igual que algunos sistemas de multidifusión atómica, como los JGroups de Java.

Una vez que el sistema entra en modo de partición, son posibles dos estrategias. La primera es limitar algunas operaciones, reduciendo así la disponibilidad. La segunda es registrar información adicional sobre las operaciones que será útil durante la recuperación de la partición. Seguir intentando la comunicación permitirá al sistema discernir cuándo termina la partición.

### ¿Qué operaciones deben realizarse?

Decidir qué operaciones limitar depende principalmente de las invariantes que debe mantener el sistema. Dado un conjunto de invariantes, el diseñador debe decidir si mantener o no una invariante particular durante el modo de partición o arriesgarse a violarla con la intención de restaurarla durante la recov-

el estado de la otra parte no es conocible). Los eventos externalizados, como el cobro de una tarjeta de crédito, suelen funcionar así. En este caso, la estrategia es registrar la intención y ejecutarla después del cobro. Tales transacciones suelen formar parte de un flujo de trabajo más amplio que tiene un estado de procesamiento de órdenes explícito, y no hay mucho inconveniente en retrasar la operación hasta que la partición termine. El diseñador pierde A de una manera que los usuarios no ven. Los usuarios sólo saben que han hecho un pedido y que el sistema lo ejecutará más tarde.

En términos más generales, el modo de partición da lugar a un desafío fundamental para la interfaz de usuario, que consiste en comunicar que las tareas están en curso pero no se han completado. Los investigadores han explorado este problema con cierto detalle para el funcionamiento desconectado, que no es más que una partición larga. La aplicación de calendario de Bayou, por ejemplo, muestra las entradas potencialmente inconsistentes (tentativas) en un color diferente. 7 Este tipo de notificaciones son regularmente visibles tanto en las aplicaciones de flujo de trabajo, como el comercio con notificaciones por correo electrónico, como en los servicios en la nube con un modo desconectado, como Google Docs.

Una de las razones para centrarse en las operaciones atómicas explícitas, en lugar de sólo en las lecturas y escrituras, es que es mucho más fácil analizar el impacto de las operaciones de nivel superior en las invariantes. Esencialmente, el diseñador debe construir una tabla que mire el producto cruzado de todas las operaciones y todos los invariantes y decidir para cada entrada si esa operación podría violar el invariante. En caso afirmativo, el diseñador debe decidir si prohíbe, retrasa o modifica la operación. En la práctica, estas decisiones también pueden depender del estado conocido, de los argumentos o de ambos. Por ejemplo, en los sistemas con un nodo de origen para determinados datos5 , las operaciones suelen poder realizarse en el nodo de origen pero no en otros nodos.

La mejor manera de seguir la historia de las operaciones en ambos lados es utilizar vectores de versiones, que capturan las dependencias causales entre las operaciones. Los elementos del vector son un par (nodo, tiempo lógico), con una entrada por cada nodo que haya actualizado el objeto y la hora de su última actualización. Dadas dos versiones de un objeto, A y B, A es más nueva que B si, para cada nodo en común en sus vectores, los tiempos de A

|  |
| --- |
| El sistema concatena los registros, los ordena y los ejecuta. La conmutatividad implica la capacidad de reordenar las operaciones en un orden global preferentemente coherente. Desgraciadamente, utilizar sólo operaciones conmutativas es más difícil de lo que parece; por ejemplo, la suma es conmutativa, pero la suma con una comprobación de límites no lo es (un balance cero, por ejemplo). |
| **Los diseñadores pueden optar por restringir el uso de ciertas operaciones durante la partición para que el sistema pueda fusionar automáticamente el estado durante la recuperación.** |
| Los recientes trabajos de Marc Shapiro y sus colegas del INRIA12,13 han mejorado mucho el uso de las operaciones conmutativas para la convergencia de estados. El equipo ha desarrollado *tipos de datos replicados conmutativos* (CRDT), una clase de estructuras de datos que convergen de forma demostrable después de una partición, y describe cómo utilizar estas estructuras para   * garantizar que todas las operaciones durante una partición sean com- portantes, *o* * representan valores en un entramado y garantizan que todas las operaciones durante una partición son monotónicamente crecientes con respecto a ese entramado.   Este último enfoque hace converger el estado moviéndose hacia el máximo de los valores de cada lado. Es una formalización y mejora de lo que hace Amazon con su carrito de la compra:14 tras una partición, el valor convergente es la unión de los dos carritos, siendo la unión una operación de conjunto monótona. La consecuencia de esta elección es que los artículos borrados pueden reaparecer.  Sin embargo, los CRDT también pueden implementar conjuntos tolerantes a las particiones que añaden y eliminan elementos. La esencia de este enfoque consiste en mantener dos conjuntos: uno para los elementos añadidos y otro para los eliminados, siendo la diferencia el número de miembros del conjunto. Cada conjunto simplificado converge y, por lo tanto, también lo hace la diferencia. En algún momento, el sistema puede limpiar las cosas simplemente eliminando los elementos eliminados de ambos conjuntos. Sin embargo, esta limpieza sólo es posible mientras el sistema no esté particionado. En otras palabras, el diseñador debe prohibir o posponer algunas operaciones durante una partición, pero se trata de operaciones de limpieza que no limitan la disponibilidad per- cibida. Por lo tanto, al implementar el estado a través de CRDTs, un diseñador puede elegir A y todavía asegurar que el estado converge automáticamente después de una partición.  **Compensación de errores**  Además de calcular el estado posterior a la partición, existe el problema algo más difícil de corregir los errores cometidos |

son mayores o iguales a los de B y al menos uno de los tiempos de A es mayor.

Si es imposible ordenar los vectores, entonces las actualizaciones fueron concurrentes y posiblemente inconsistentes. Así, dado el historial de vectores de versiones de ambas partes, el sistema puede decir fácilmente qué operaciones están ya en un orden conocido y cuáles se ejecutaron de forma concurrente. Un trabajo reciente8 ha demostrado que este tipo de consistencia causal es el mejor resultado posible en general si el diseñador decide centrarse en la disponibilidad.

### Recuperación de particiones

En algún momento, la comunicación se reanuda y la partición termina. Durante la partición, cada lado estaba disponible y, por tanto, avanzando, pero la partición ha retrasado algunas operaciones y ha violado algunas invariantes. En este punto, el sistema conoce el estado y la historia de ambos lados porque ha mantenido un cuidadoso registro durante el modo de partición. El estado es menos útil que el historial, a partir del cual el sistema puede deducir qué operaciones violaron realmente los invariantes y qué resultados se externalizaron, incluyendo las respuestas enviadas al usuario. El diseñador debe resolver dos problemas difíciles durante la recuperación:

* el estado de ambas partes debe ser coherente, y
* debe haber una compensación por los errores cometidos

durante el modo de partición.

Por lo general, es más fácil arreglar el estado actual empezando por el estado en el momento de la partición y haciendo retroceder ambos conjuntos de operaciones de alguna manera, manteniendo un estado consistente a lo largo del camino. Bayou hizo esto explícitamente haciendo retroceder la base de datos a un momento correcto y reproduciendo el conjunto completo de operaciones en un orden bien definido y determinista para que todos los nodos alcanzaran el mismo estado. 9 Del mismo modo, los sistemas de control de código fuente, como el Sistema de Verificación Concurrente (CVS), parten de un punto consistente compartido y hacen retroceder las actualizaciones para fusionar ramas.

La mayoría de los sistemas no siempre pueden fusionar conflictos. Por ejemplo, CVS tiene ocasionalmente conflictos que el usuario debe resolver manualmente, y los sistemas wiki con modo offline suelen dejar conflictos en el documento resultante que requieren edición manual. 10

Por el contrario, algunos sistemas *siempre* pueden fusionar conflictos eligiendo determinadas operaciones. Un ejemplo es la edición de texto en Google Docs,11 que limita las operaciones a la aplicación de un estilo y a la adición o eliminación de texto. Por lo tanto, aunque el problema general de la resolución de conflictos no tiene solución, en la práctica, los diseñadores pueden optar por restringir el uso de ciertas operaciones durante la partición para que el sistema pueda fusionar automáticamente el estado durante la recuperación. Retrasar las operaciones arriesgadas es una implementación relativamente fácil de esta estrategia.

El uso de operaciones conmutativas es la aproximación más cercana a un marco general para la convergencia automática de estados.

## PORTADA DE LA REVISTA

**PROBLEMAS DE COMPENSACIÓN EN UN CAJERO AUTOMÁTICO**



**I**

**n el diseño de un cajero automático (ATM), la consistencia fuerte parecería ser la opción lógica, pero en la práctica,**

**La A supera a la C. La razón es bastante sencilla: una mayor disponibilidad significa mayores ingresos. En cualquier caso, el diseño de la ATM sirve como un buen contexto para revisar algunos de los desafíos involucrados en la compensación de las violaciones de invariantes durante una partición.**

**Las operaciones esenciales del cajero automático son depositar, retirar y comprobar el saldo. La invariante clave es que el saldo debe ser cero o superior. Como sólo la retirada puede violar el invariante, necesitará un tratamiento especial, pero las otras dos operaciones siempre pueden ejecutarse.**

**El diseñador del sistema de cajeros podría optar por prohibir las retiradas durante una partición, ya que es imposible conocer el saldo real en ese momento, pero eso comprometería la disponibilidad. En su lugar, utilizando el modo de *espera* (modo de partición), los cajeros modernos limitan la retirada neta a un máximo de *k*, donde *k* puede ser de 200 dólares. Por debajo de este límite, las retiradas funcionan completamente; cuando el saldo alcanza el límite, el sistema deniega las retiradas. Así, el cajero elige un límite sofisticado de disponibilidad que permite las retiradas pero limita el riesgo.**

**Cuando la partición termina, debe haber alguna forma de restaurar la consistencia y compensar los errores cometidos mientras el sistema estaba particionado. Restaurar el estado es fácil porque las operaciones son conmutativas, pero la compensación puede tomar varias formas. Un saldo final inferior a cero viola el invariante. En el caso normal, el cajero automático dispensó el dinero, lo que provocó el error externo. El banco compensa cobrando una comisión y esperando el reembolso. Dado que el riesgo está acotado, el problema no es grave. Sin embargo, supongamos que el saldo era inferior a cero en algún momento de la partición (desconocido para el cajero), pero que un depósito posterior lo devolvió. En este caso, el banco podría cobrar una comisión por sobregiro con carácter retroactivo, o podría ignorar la infracción, puesto que el cliente ya ha realizado el pago necesario. En general, debido a los retrasos en la comunicación, el sistema bancario no depende de la coherencia para ser correcto, sino de la auditoría y la compensación. Otro ejemplo de esto es el "check kiting", en el que un cliente retira dinero de varias sucursales antes de que puedan comunicarse y luego huye. El sobregiro se detecta más tarde, lo que puede dar lugar a una compensación en el**

**forma de acción legal.**

durante la partición. El seguimiento y la limitación de las operaciones de modo de partición garantizan el conocimiento de qué invariantes podrían haberse violado, lo que a su vez permite al diseñador crear una estrategia de restauración para cada uno de esos invariantes. Normalmente, el sistema descubre la violación durante la recuperación y debe implementar cualquier corrección en ese momento. Hay varias formas de arreglar las invariantes, incluyendo formas triviales como "el último escritor gana" (que ignora algunas actualizaciones), enfoques más inteligentes que fusionan operaciones y la escalada humana. Un ejemplo de esto último es el overbooking en los aviones: subir al avión es, en cierto sentido, una recuperación de particiones con la invariante de que debe haber al menos tantos asientos como pasajeros. Si hay demasiados pasajeros, algunos perderán sus asientos, e ide-

almente el servicio de atención al cliente compensará a esos pasajeros de alguna manera.

El ejemplo del avión también exhibe un error externalizado: si la aerolínea no hubiera dicho que el pasajero tenía un asiento, solucionar el problema sería mucho más fácil. Esta es otra razón para retrasar las operaciones de riesgo: en el momento de la recuperación, se conoce la verdad. La idea de la compensación es realmente el núcleo de la reparación de estos errores; los diseñadores deben crear operaciones de compensación que restauren una variante interna y corrijan de forma más amplia un error externalizado. Técnicamente, los CRDT sólo permiten invariantes *localmente* verificables

-una limitación que hace innecesaria la compensación pero que disminuye un poco la potencia del enfoque. Sin embargo, una solución que utilice CRDTs para la convergencia del estado podría permitir la violación temporal de un invariante global, hacer converger el estado después de la partición, y luego ejecutar cualquier compensación necesaria.

La recuperación de los errores externalizados suele requerir algo de historia sobre los resultados externalizados. Consideremos el escenario de la "marcación" en estado de embriaguez, en el que una persona no recuerda haber realizado varias llamadas telefónicas mientras estaba intoxicada la noche anterior. El estado de esa persona a la luz del día podría ser sólido, pero el registro sigue mostrando una lista de llamadas, algunas de las cuales podrían haber sido errores. Las llamadas son los efectos externos del estado de la persona (intoxicación). Como la persona no recuerda las llamadas, podría ser difícil compensar los problemas que ha causado.

En un contexto de máquina, un ordenador podría ejecutar órdenes dos veces durante una partición. Si el sistema puede distinguir dos órdenes intencionadas de dos órdenes duplicadas, puede cancelar una de las duplicadas. Si se externaliza, una estrategia de compensación sería autogenerar un correo electrónico para el cliente explicando que el sistema ejecutó accidentalmente la orden dos veces pero que el error se ha corregido y adjuntando un cupón de descuento para la siguiente orden. Sin embargo, sin el historial adecuado, la carga de detectar el error recae en el cliente.

Algunos investigadores han explorado formalmente las transacciones de compensación como una forma de tratar las transacciones de larga duración. 15,16 Las transacciones de larga duración se enfrentan a una variación de la decisión de partición: ¿es mejor mantener los bloqueos durante mucho tiempo para garantizar la coherencia, o liberarlos antes y exponer los datos no comprometidos a otras transacciones pero permitir una mayor concurrencia? Un ejemplo típico es tratar de actualizar todos los registros de los empleados como una sola transacción. La serialización de esta transacción en la forma normal bloquea todos los registros y evita la concurrencia. Las transacciones de compensación adoptan un enfoque diferente al dividir la transacción grande en una saga, que consiste en múltiples subtransacciones, cada una de las cuales se compromete en el camino. Así, para abortar la transacción más grande, el sistema debe deshacer cada subtransacción ya comprometida emitiendo una nueva transacción que corrija sus efectos: la transacción de compensación. En general, el objetivo es evitar abortar otras transacciones.

|  |
| --- |
| *10º Ann. ACM Symp. User Interface Software and Technol-gy* (UIST 97), ACM, 1999, pp. 119-128.   1. P. Mahajan, L. Alvisi y M. Dahlin, *Consistency, Avail- ability, and Convergence,* tech. report UTCS TR-11-22, Univ. of Texas at Austin, 2011. 2. D.B. Terry et al., "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System", *Proc. 15th ACM Symp. Operating Systems Principles* (SOSP 95), ACM, 1995, pp. 172-182. 3. B. Du y E.A. Brewer, "DTWiki: A Disconnection and Intermittency Tolerant Wiki", *Proc. 17th Int'l Conf. World Wide Web* (WWW 08), ACM, 2008, pp. 945-952. 4. "Qué hay de diferente en el nuevo Google Docs: Resolución de conflictos", blog; <http://googledocs.blogspot.com/2010/09/>whats-different-about-new-google-docs\_22.html. 5. M. Shapiro et al., "Conflict-Free Replicated Data Types," *Proc. 13th Int'l Conf. Stabilization, Safety, and Security of Distributed Systems* (SSS 11), ACM, 2011, pp. 386-400. 6. M. Shapiro et al., "Convergent and Commutative Replicated Data Types", *Bulletin of the EATCS*, nº 104, junio de 2011, pp. 67-88. 7. G. DeCandia y otros, "Dynamo: Amazon's Highly Available Key-Value Store", *Proc. 21st ACM SIGOPS Symp. Operating Systems Principles* (SOSP 07), ACM, 2007, pp. 205-220. 8. H. García-Molina y K. Salem, "SAGAS", *Proc. ACM SIGMOD Int'l Conf. Management of Data* (SIGMOD 87), ACM, 1987, pp. 249-259. 9. H. Korth, E. Levy, y A. Silberschatz, "A Formal Approach to Recovery by Compensating Transactions", *Proc. VLDB Endowment* (VLDB 90), ACM, 1990, pp. 95-106.   ***Eric Brewer es*** *profesor de informática en la Universidad de California, Berkeley, y vicepresidente de infraestructuras en Google. Sus intereses de investigación incluyen la computación en nube, los servidores escalables, las redes de sensores y la tecnología para las regiones en desarrollo. También ayudó a crear USA.gov, el portal oficial del gobierno federal. Brewer se doctoró en ingeniería eléctrica y ciencias de la computación en el MIT. Es miembro de la Academia Nacional de Ingeniería. Póngase en contacto con él en brewer@cs. berkeley.edu.* |
| **ICDE 2012**  **28ª Conferencia Internacional del IEEE sobre Ingeniería de Datos**  **1-5 de abril de 2012**  Washington, DC, Estados Unidos  La ICDE aborda cuestiones de investigación en el diseño, la construcción, la gestión y la evaluación de sistemas y aplicaciones avanzadas de uso intensivo de datos.  *Más información en:*  [**http://www.icde12.org/**](http://www.icde12.org/) |
|  |

acciones que utilizaron los datos comprometidos incorrectamente (no hay abortos en cascada). La corrección de este enfoque no depende de la serializabilidad o el aislamiento, sino del efecto neto de la secuencia de transacciones sobre el estado y las salidas. Es decir, después de las compensaciones, ¿acaba la base de datos esencialmente en un lugar equivalente a donde habría estado si las subtransacciones nunca se hubieran ejecutado? La equivalencia debe incluir las acciones externalizadas; por ejemplo, reembolsar una compra duplicada no es lo mismo que no cobrar a ese cliente en primer lugar, pero podría decirse que es equivalente. La misma idea es válida en la recuperación de particiones. Un proveedor de servicios o productos no siempre puede deshacer los errores directamente, pero pretende admitirlos y tomar nuevas medidas compensatorias. La mejor manera de aplicar estas ideas a la recuperación de particiones es un problema abierto. La barra lateral "Cuestiones de compensación en un cajero automático" describe algunas de las preocupaciones en un área de aplicación.

Los diseñadores de sistemas no deben sacrificar ciegamente la consistencia o la disponibilidad cuando existen particiones. Utilizando el enfoque propuesto, pueden optimizar ambos prop

**S**

eridades a través de la gestión cuidadosa de invariantes durante las particiones. A medida que las técnicas más nuevas, como los vectores de versión y los CRDT, pasan a marcos que simplifican su

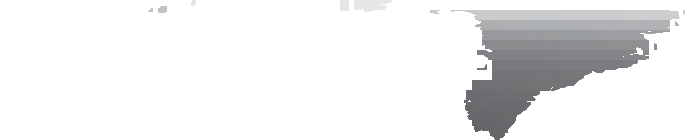
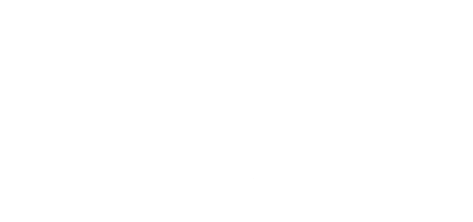
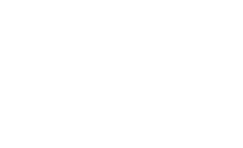
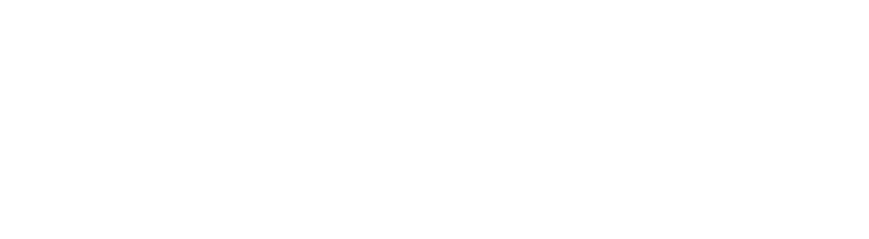
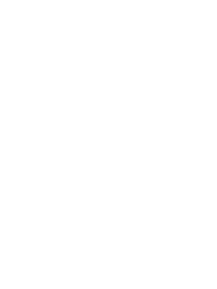
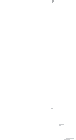
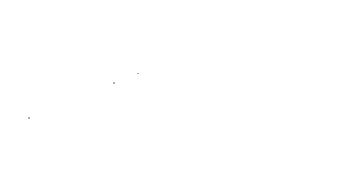
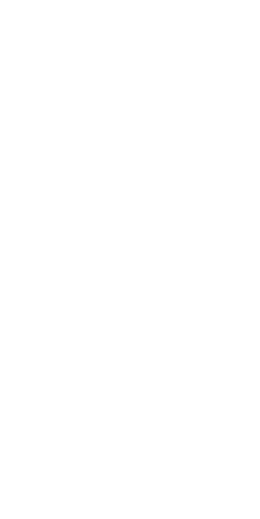
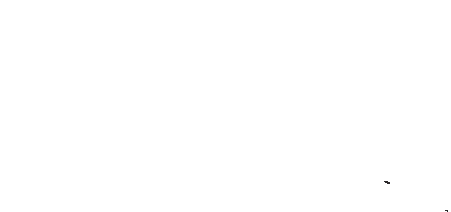
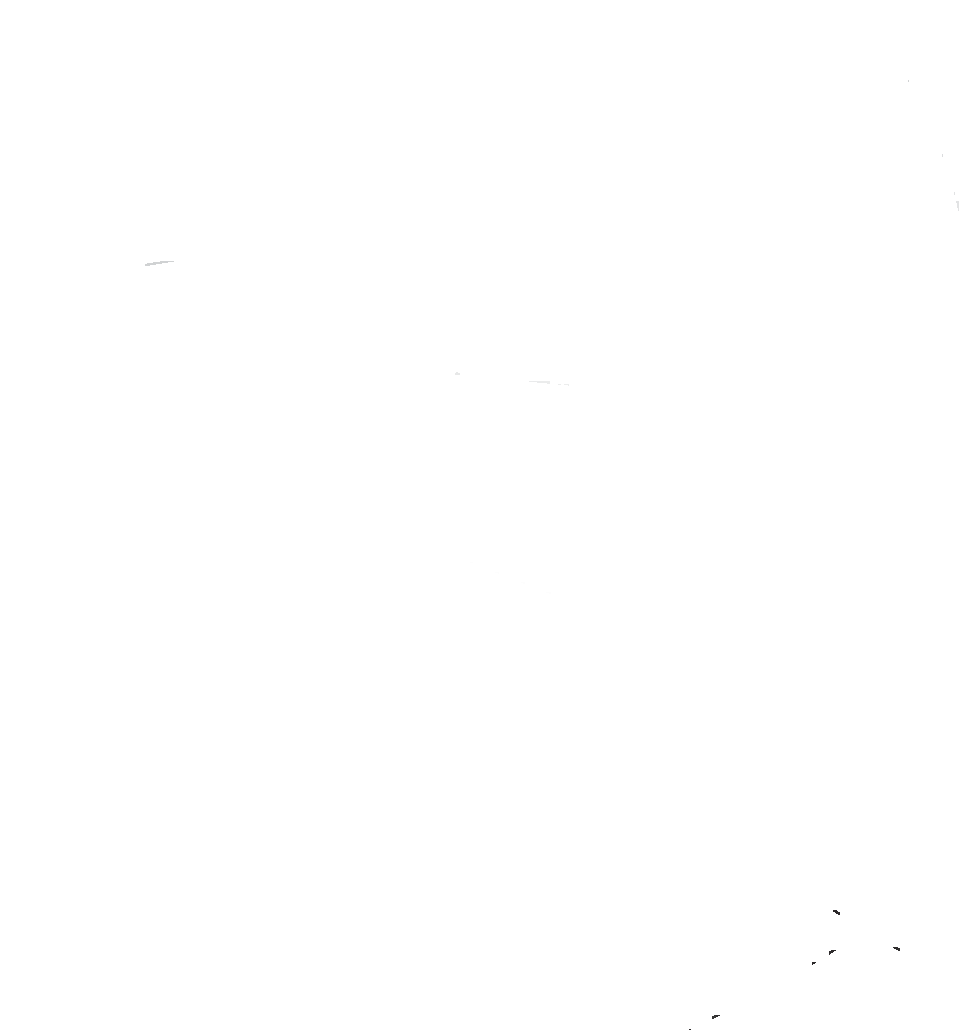
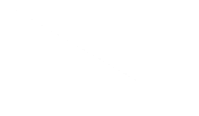
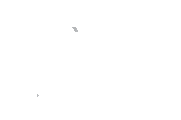
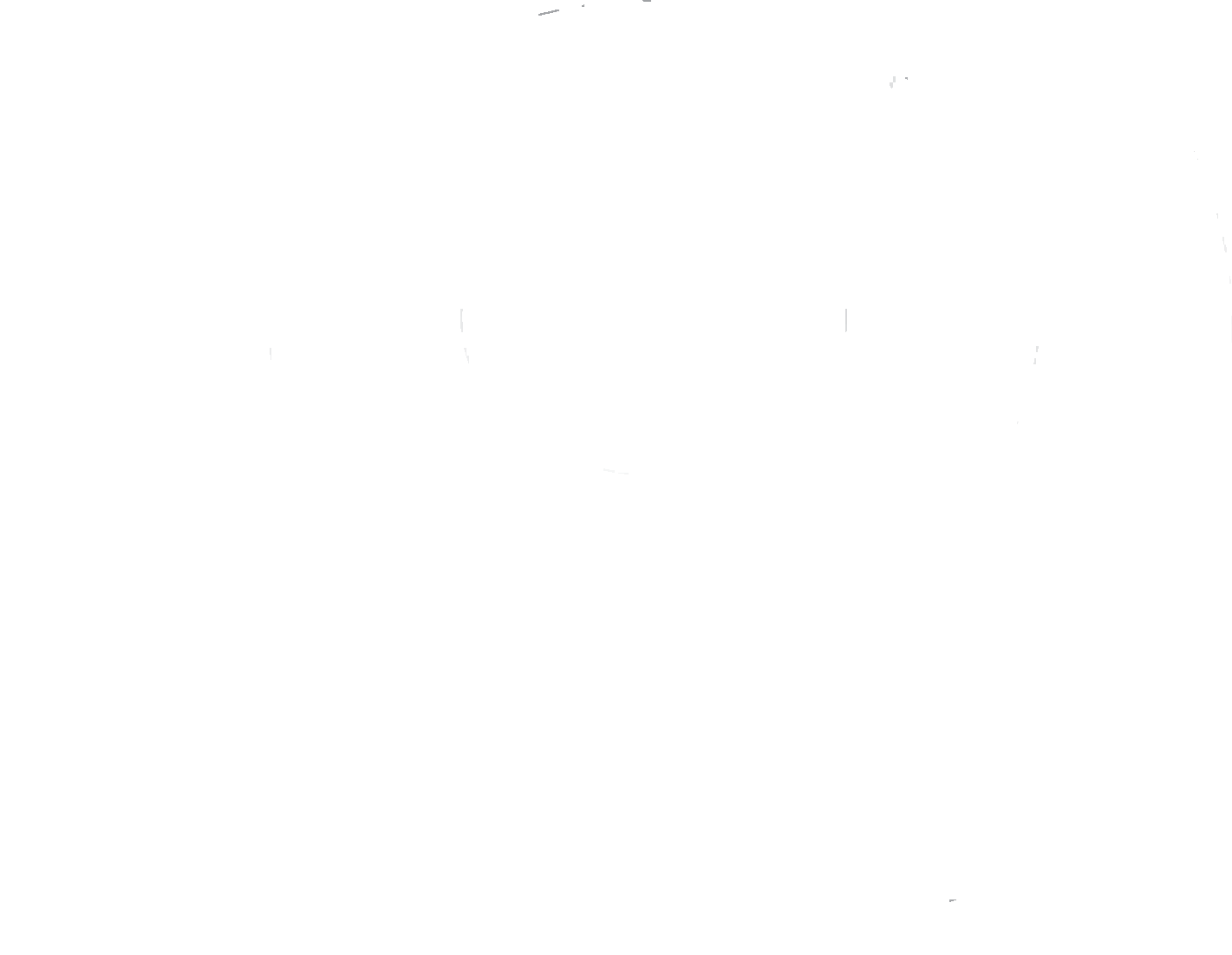
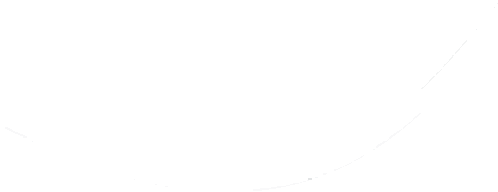
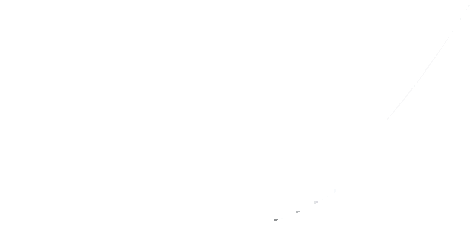
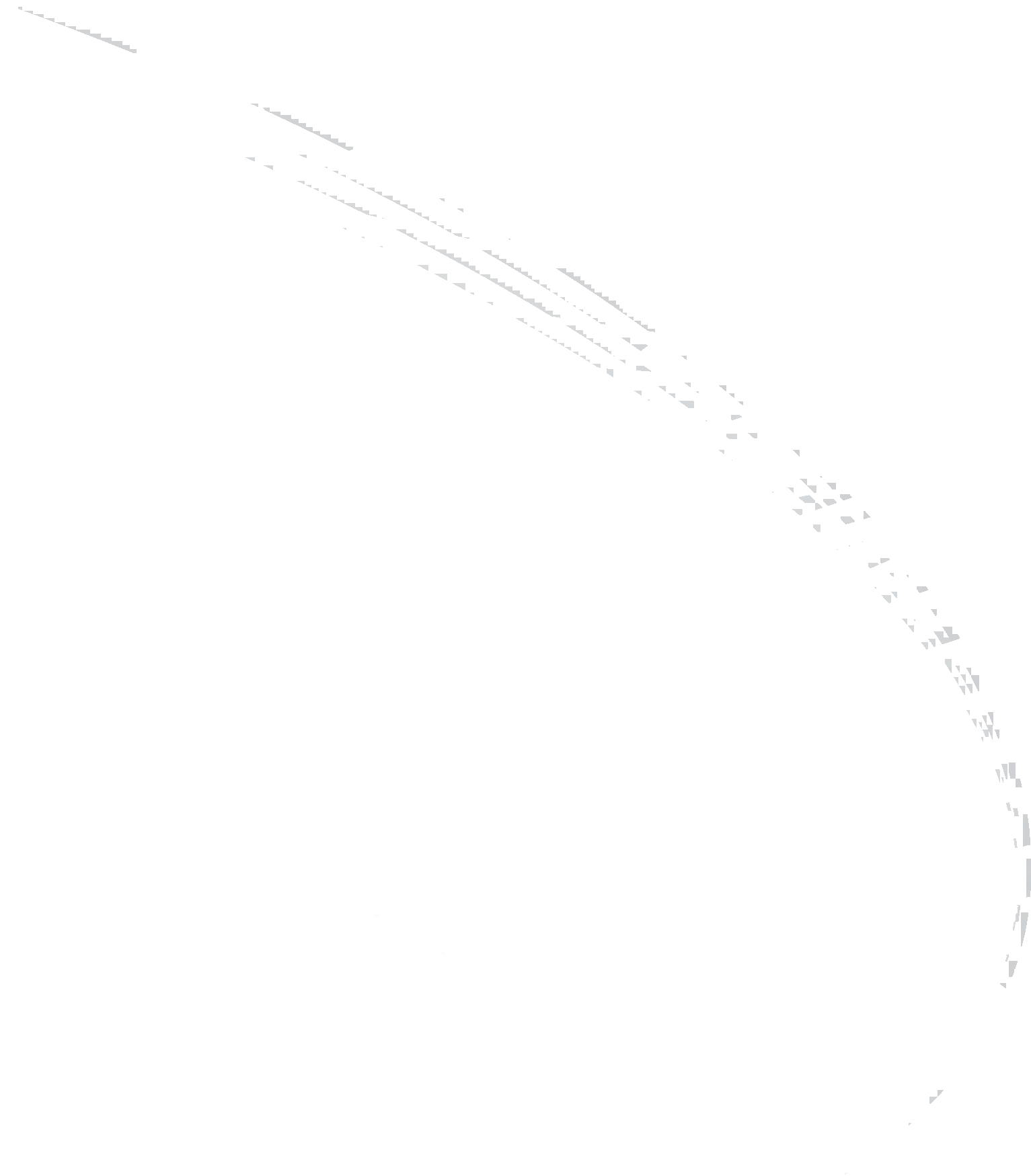
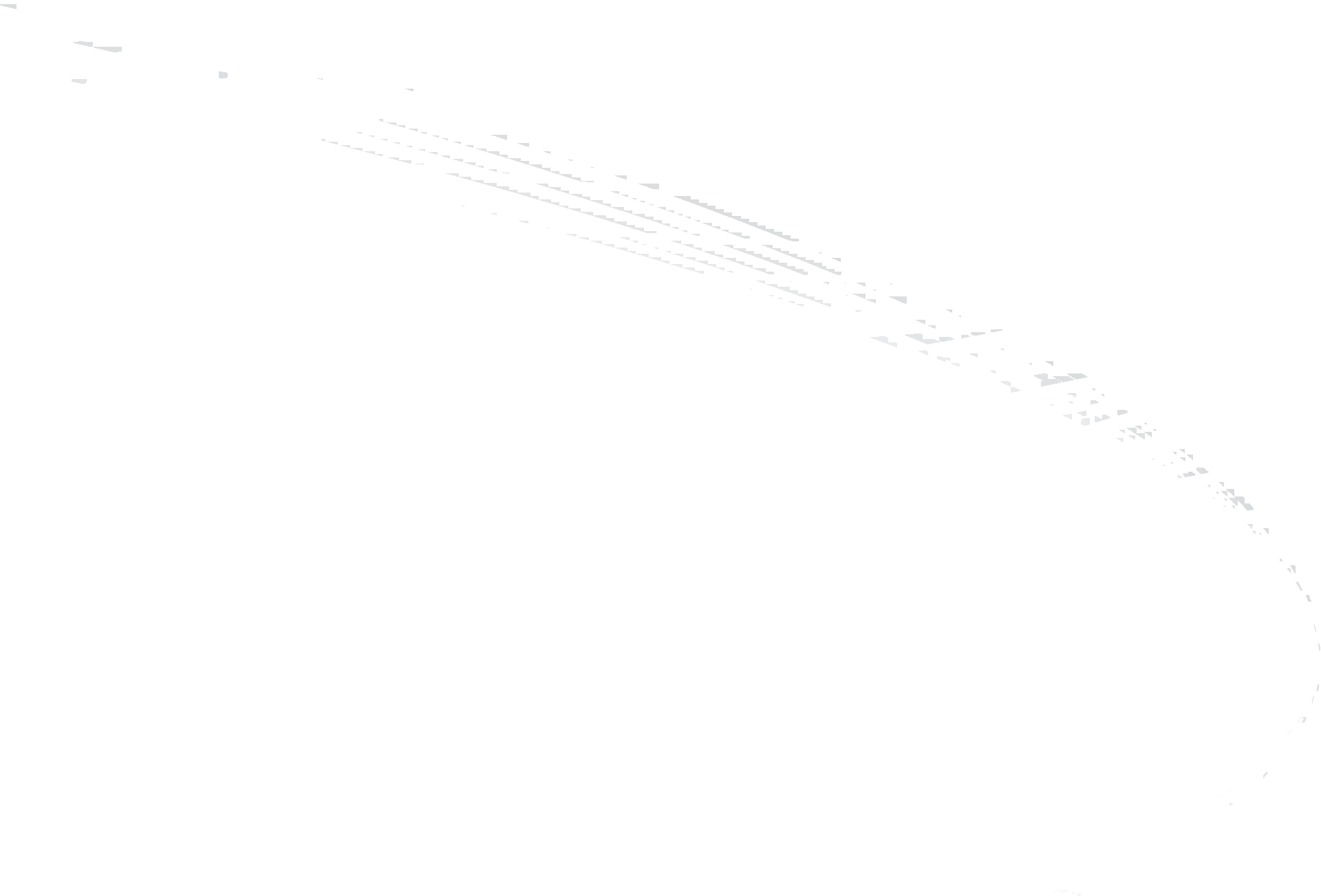
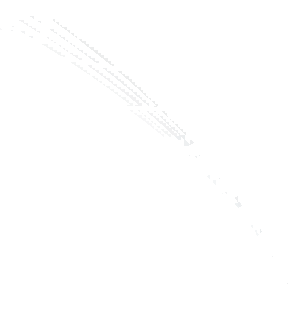
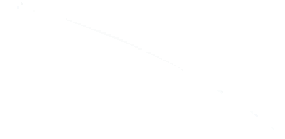
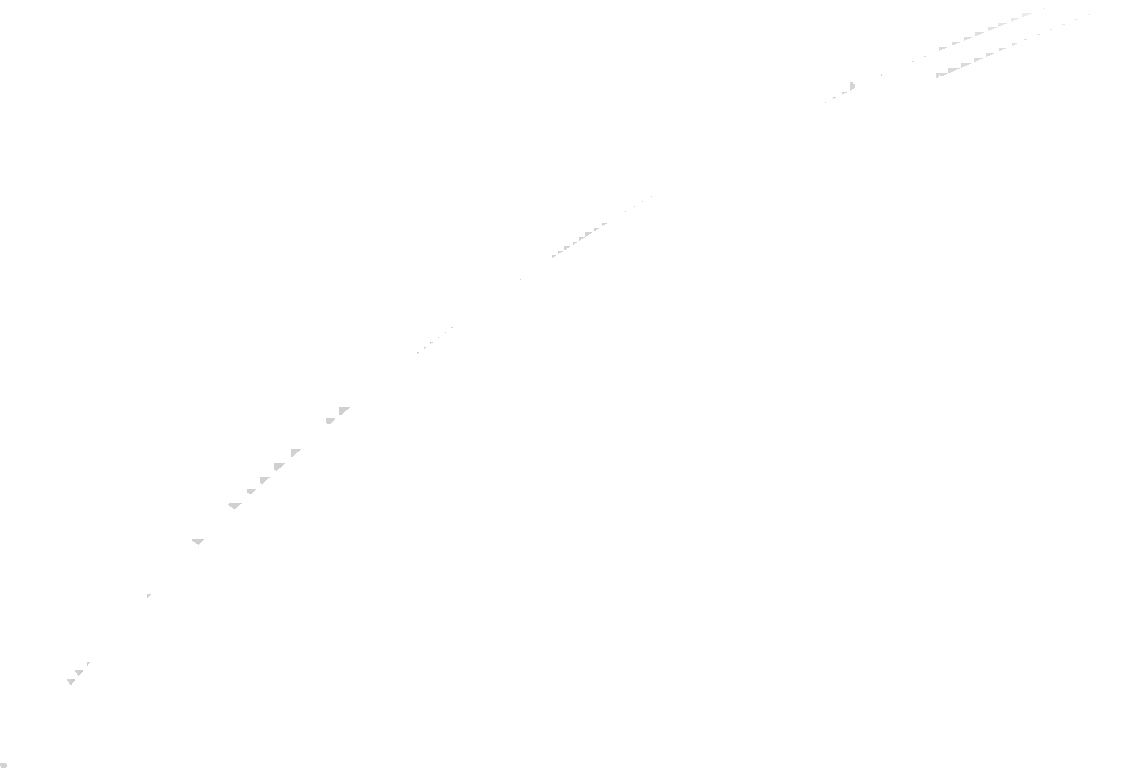
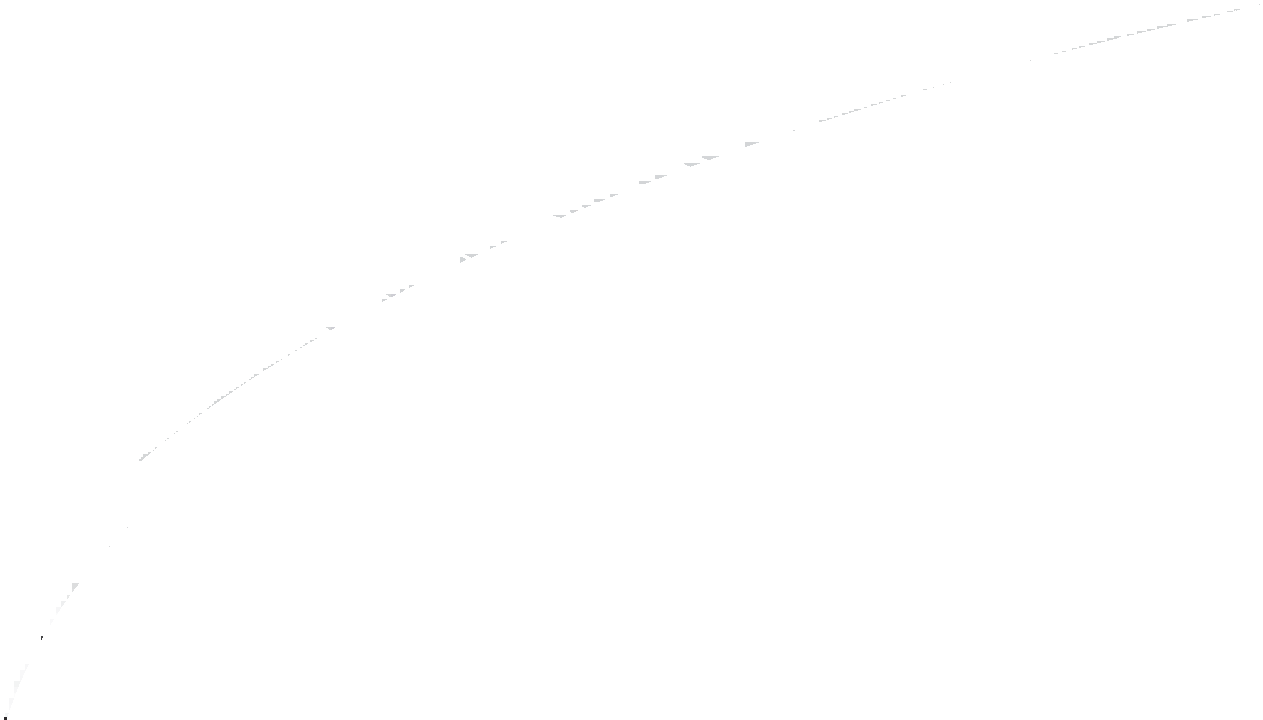
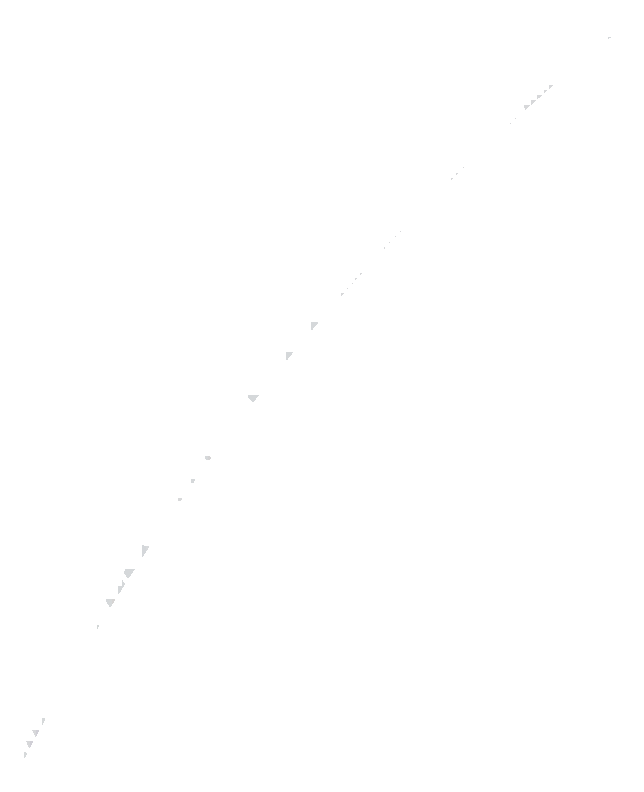
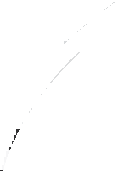
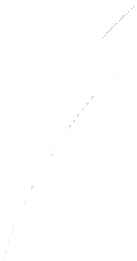
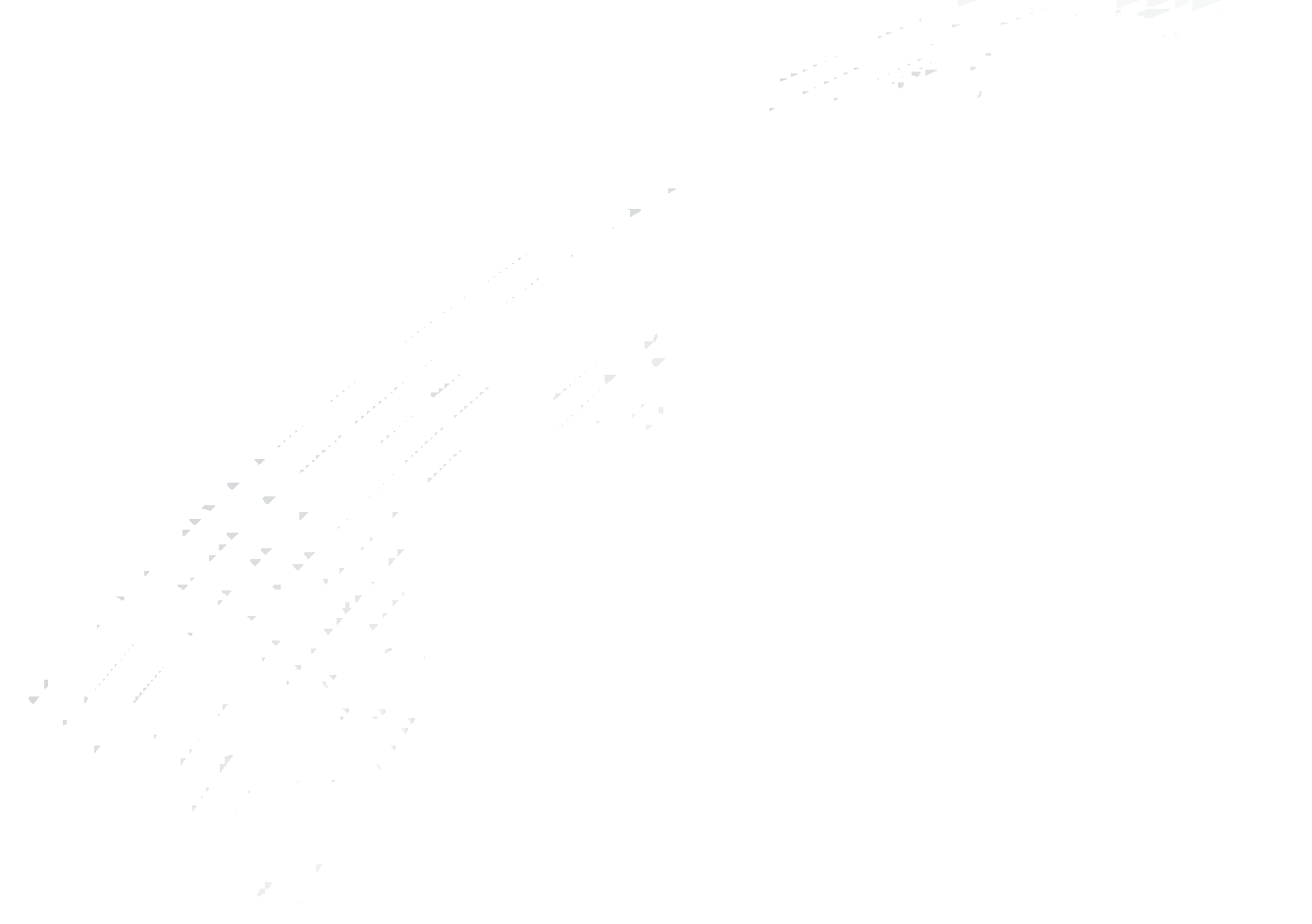
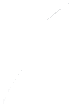
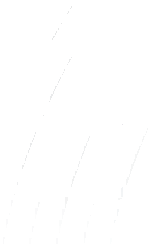
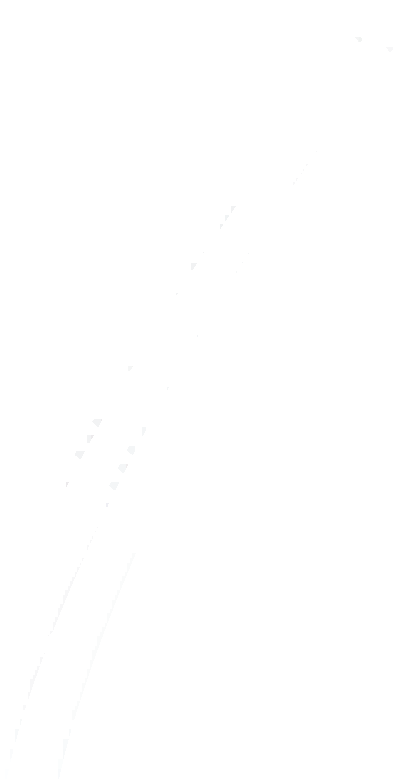
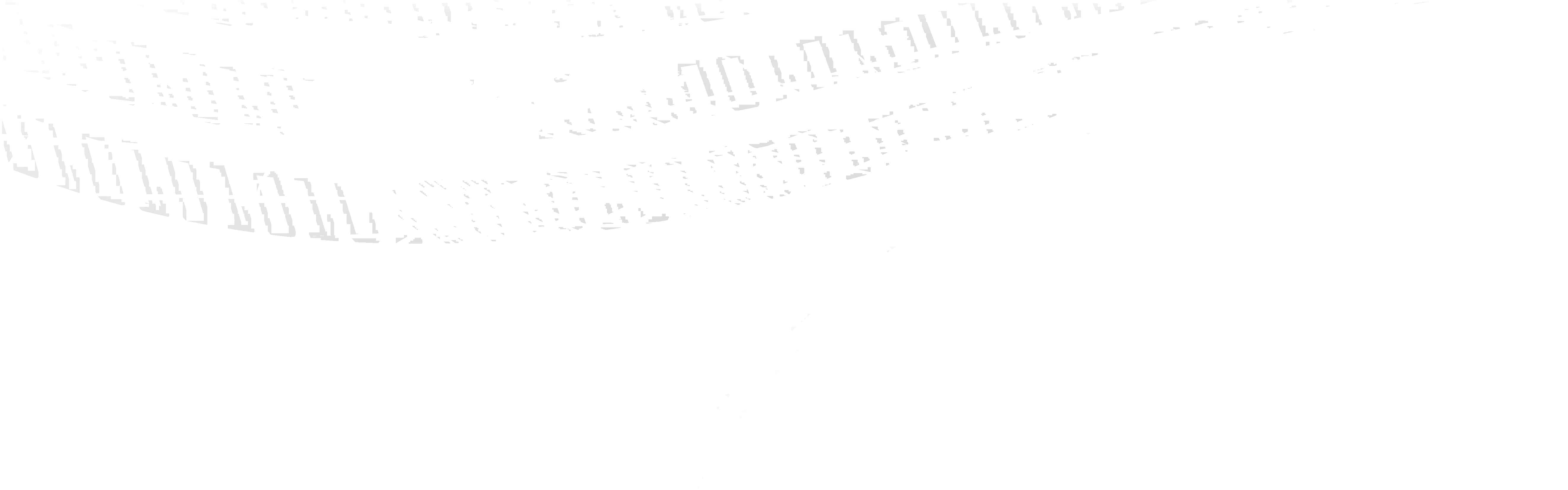
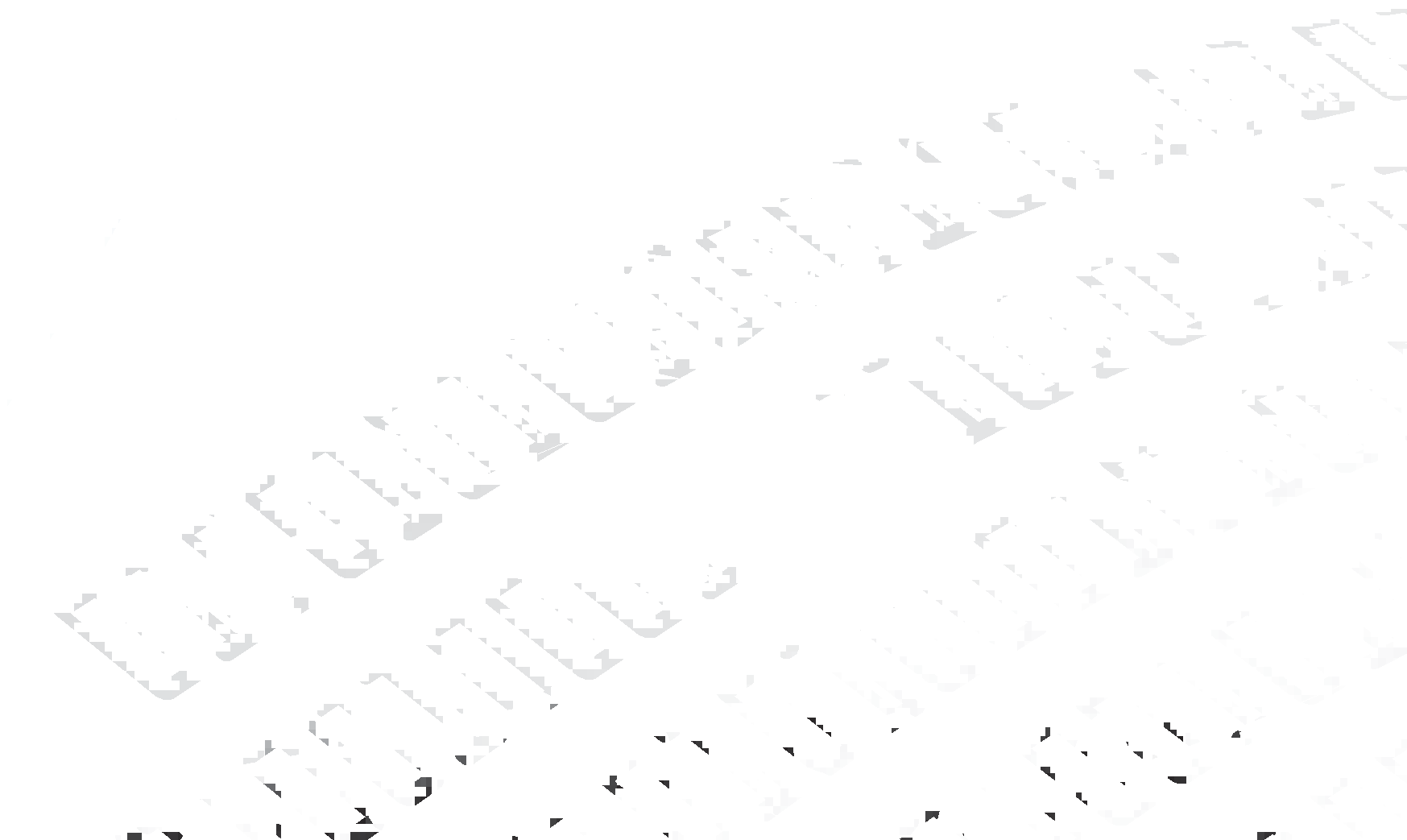
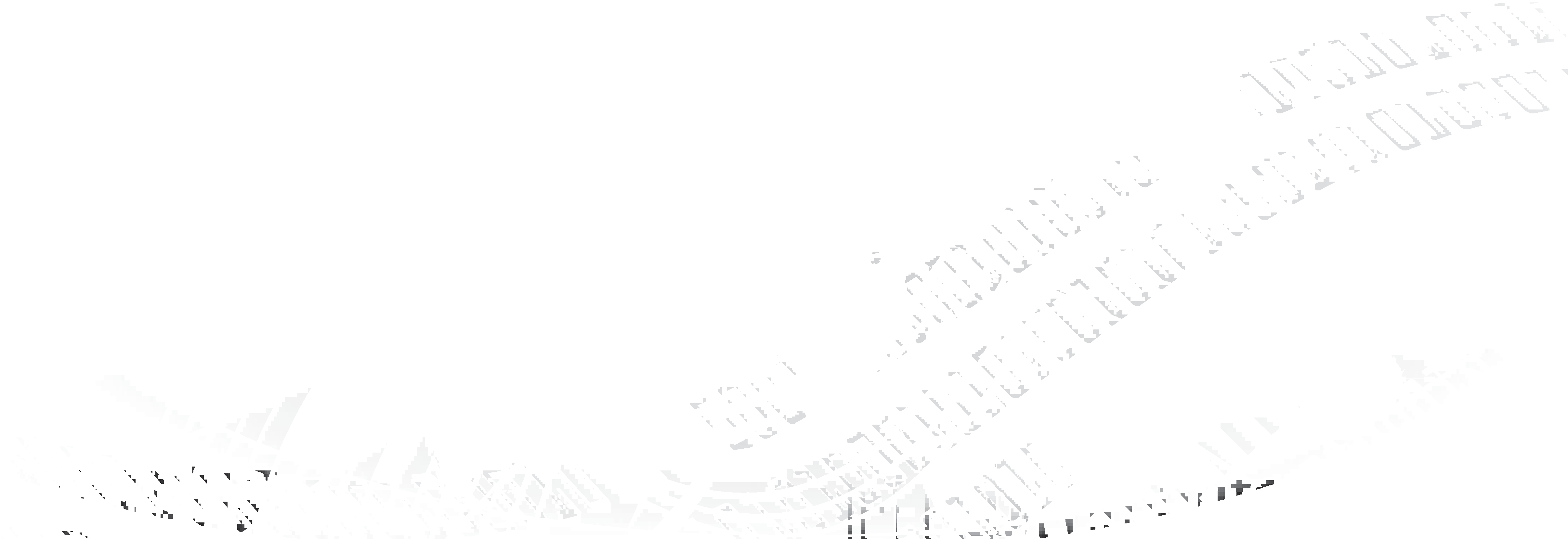
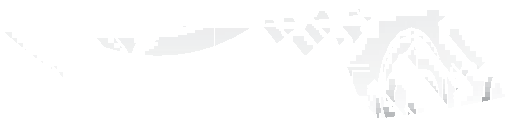
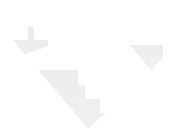
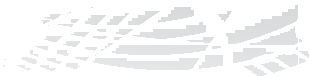
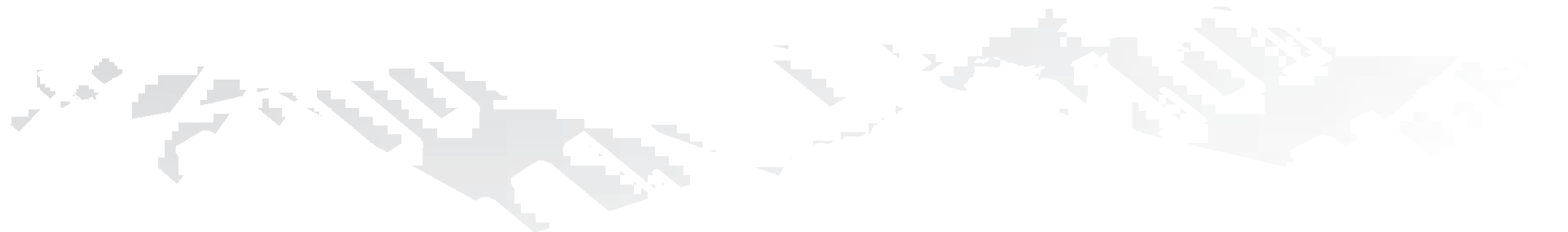
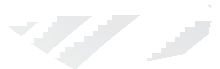
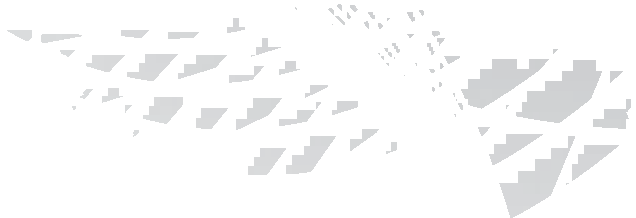
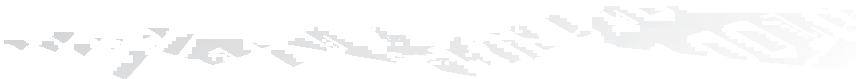
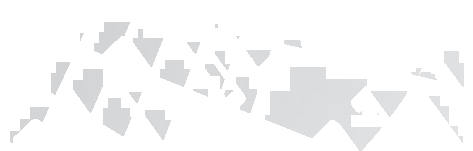
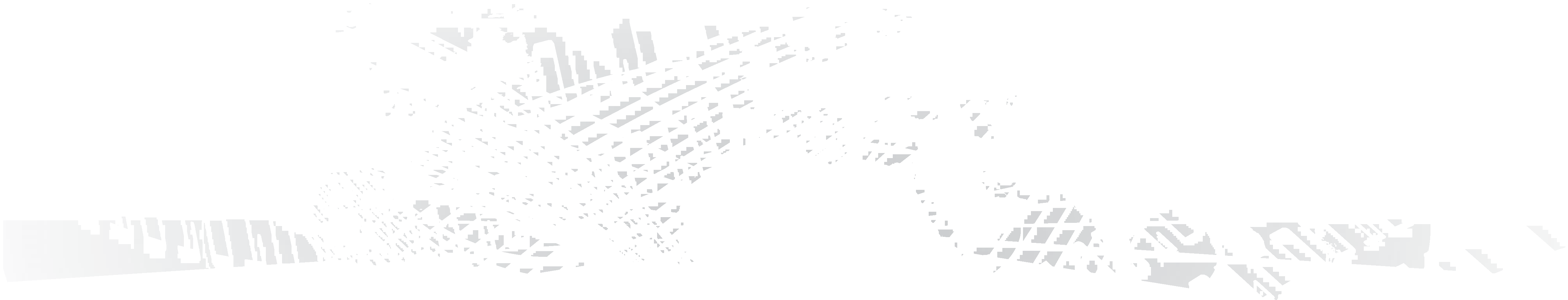
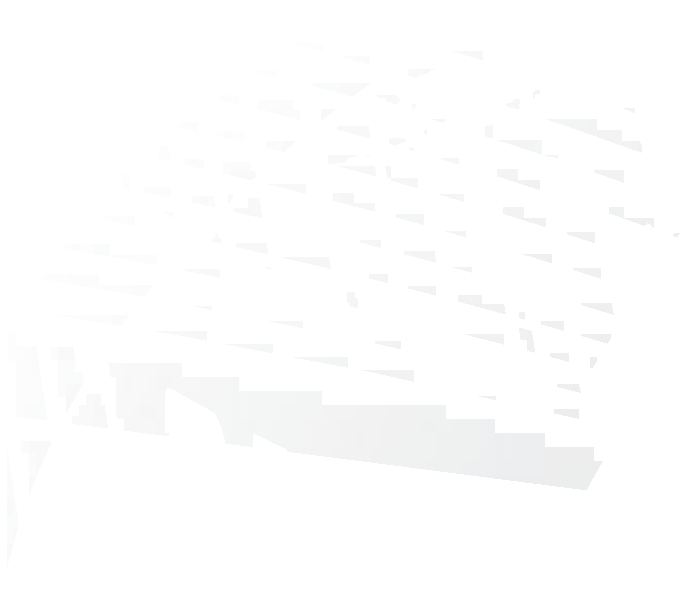
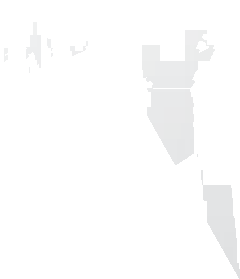
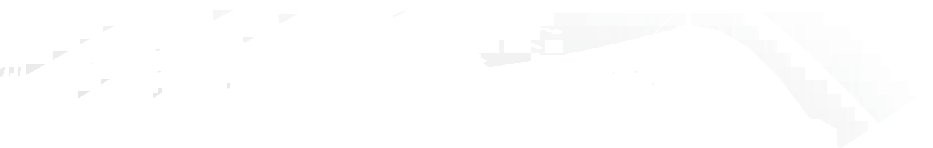
uso, este tipo de optimización debería extenderse más. Sin embargo, a diferencia de las transacciones ACID, este enfoque requiere un despliegue más reflexivo en relación con las estrategias anteriores, y las mejores soluciones dependerán en gran medida de los detalles sobre las invariantes y las operaciones del servicio.

### Agradecimientos

Agradezco a Mike Dahlin, Hank Korth, Marc Shapiro, Justin Sheehy, Amin Vahdat, Ben Zhao y a los voluntarios de la IEEE Computer Society sus útiles comentarios sobre este trabajo.

### Referencias

1. E. Brewer, "Lessons from Giant-Scale Services", *IEEE In- ternet Computing*, julio/agosto de 2001, pp. 46-55.
2. A. Fox et al., "Cluster-Based Scalable Network Services", *Proc. 16th ACM Symp. Operating Systems Principles* (SOSP 97), ACM, 1997, pp. 78-91.
3. A. Fox y E.A. Brewer, "Harvest, Yield and Scalable Toler- ant Systems", *Proc. 7th Workshop Hot Topics in Operating Systems* (HotOS 99), IEEE CS, 1999, pp. 174-178.
4. E. Brewer, "Towards Robust Distributed Systems", *Proc. 19th Ann. ACM Symp.Principles of Distributed Com- puting* (PODC 00), ACM, 2000, pp. 7-10; [www.cs.berkeley.](http://www.cs.berkeley/) edu/~brewer/PODC2000.pdf.



1. B. Cooper et al., "PNUTS: Yahoo!'s Hosted Data Serving Platform", *Proc. VLDB Endowment* (VLDB 08), ACM, 2008, pp. 1277-1288.
2. J. Sobel, "Scaling Out", *Facebook Engineering Notes*, 20 de agosto de 2008; [www.facebook.com/note.](http://www.facebook.com/note) php?note\_id=23844338919&id=9445547199.
3. W. K. Edwards et al., "Designing and Implementing Asyn- chronous Collaborative Applications with Bayou", *Proc.*



Los mejores artículos, podcasts y más.

**computingnow.computer.org**

Este artículo apareció en

Para acceder a más contenidos de la IEEE Computer Society, consulte computingnow.computer.org.



Los mejores artículos, podcasts y más.

**computingnow.computer.org**

Este artículo apareció en

Para acceder a más contenidos de la IEEE Computer Society, consulte computingnow.computer.org.

