UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Departamento de Sistemas
Informáticos y Computación

Máster en Computación Paralela y Distribuida

Tesis de Máster

Consenso Bizantino y Blockchain

Valencia, a 17 de septiembre de 2018

Tesis de Máster presentada por: Fco. Javier Fernández-Bravo Peñuela

Dirigida por: José Manuel Bernabéu Aubán

Resumen

Los sistemas blockchain han despertado un gran interés desde su aparición por su utilización en las divisas criptográficas, de las que Bitcoin es la pionera y la más icónica. Sus aplicaciones se extienden a cualquier sistema que requiera mantener un registro replicado de documentos o transacciones, garantizando la consistencia eventual del estado compartido, la integridad de los datos y la tolerancia a fallos por medio de un algoritmo de consenso.

El presente trabajo versa acerca de Trebizond, un algoritmo de consenso tolerante a fallos bizantinos especialmente apropiado para su aplicación en sistemas blockchain permisionados. Su diseño está influenciado principalmente por Raft y PBFT. Del primero toma sus principios de diseño, que se fundamentan en la separación de problemas, la reducción del espacio de estados y la inteligibilidad del algoritmo. Del segundo toma algunos de los elementos que permiten dotar al algoritmo de tolerancia a fallos en un escenario bizantino.

Tras realizar un estudio detallado del problema del consenso distribuido y otros problemas estrechamente relacionados propios de los sistemas distribuidos, seguido por una crónica de las principales plataformas de blockchain permisionado actualmente en explotación, se enuncia la especificación funcional de Trebizond. Este algoritmo permite alcanzar consenso en un escenario eventualmente síncrono, autenticado y con presencia de fallos bizantinos. Respecto al estado actual de la investigación en este área, su principal contribución es la combinación de la detección de fallos activa y la validación semántica. Mientras que otros algoritmos utilizan la detección activa de fallos como una forma de mejorar el factor de tolerancia a fallos frente a la táctica de enmascaramiento habitualmente usada [HKD06], y otros algoritmos hacen uso de técnicas de validación semántica o externa a la hora de valorar si es lícito ejecutar una operación sobre el estado compartido [CKPS01][MNC12], Trebizond propone fusionar ambas técnicas. Su finalidad es utilizar la validación semántica, dependiente del protocolo de nivel superior que se ejecuta sobre el algoritmo de consenso, como parte del propio mecanismo de detección activa de fallos, de tal forma que se mejore su eficacia y completitud. Este refinamiento del detector de fallos permite, por ejemplo, deponer a un líder cuando los nodos detectan que éste está enviando operaciones que son coherentes con el propio algoritmo de consenso pero que violan la semántica del protocolo de nivel de aplicación, o aislar a una réplica del grupo de difusión cuando se tienen evidencias de comportamiento bizantino por su parte.

El establecimiento de las propiedades de seguridad y viveza del algoritmo y su diseño se detallan pormenorizadamente utilizando autómatas de entrada/salida, una técnica bien conocida para la elaboración de algoritmos distribuidos. Este diseño encuentra su correspondencia en una implementación, realizada como caso práctico con el ánimo de mostrar paradigmas y patrones que facilitan la codificación de algoritmos distribuidos a partir de un diseño bien especificado.

Abstract

Blockchain systems have been drawing a great deal of interest since their first appearance due to their use in cryptocurrencies, having Bitcoin as their pioneer and the most iconic of them. Their applications spread to any system which requires to keep a replicated log of documents or transactions, ensuring eventual consistency on the state shared among the replicas, data integrity, and fault tolerance by means of a consensus algorithm.

The present work deals with Trebizond, a fault tolerant algorithm particularly suitable for its application in permissioned blockchain systems. It is influenced by Raft and PBFT. From the former it takes its design principles, which lay on problem separation, state space reduction, and the algorithm's own intelligibility. From the latter it takes some of the elements which enable the algorithm to be provided with fault tolerance in the byzantine setting.

After carrying out a detailed study about the problem of distributed consensus and other typical problems of distributed systems which are tightly bound to it, followed by a summary of the main permissioned blockchain platforms currently in exploitation, Trebizond's functional specification is formulated. This algorithm tries to reach consensus in an eventually synchronous authenticated setting, where byzantine failures may arise. Regarding the current state of research in this area, Trebizond's main contribution is its combination of active failure detection and semantic validation. While other algorithms make use of active failure detection as a means of improving their fault tolerance degree in relation to the failure masking strategy commonly used [HKD06], and other algorithms rely on semantic or external validation when it comes to appreciate whether an operation is legal to be executed on the shared state [CKPS01][MNC12], Trebizond proposes to fuse both techniques. It aims to use semantic validation, this being dependent from the higher level protocol which executes on top of the consensus algorithm, as a part of the active failure detection system, thus improving its effectiveness and completeness. This polished failure detector makes possible to perform actions such as deposing a leader when it is detected for having been sending messages which are coherent with the consensus algorithm itself but violate the application level protocol, or isolating a replica from the communication group when evidences exist that it has featured byzantine behavior.

The algorithm's safety and liveness properties are first outlined and subsequently highlighted by means of input/output automatons, a well-known technique for crafting distributed algorithms. Such a design finds its match in an implementation case, whose goal is to show some paradigms and patterns which make easier to code distributed algorithms from a properly stated design.

Agradecimientos

En primer lugar quiero dedicar mi más sincero agradecimiento a José Bernabéu por prestarse de forma totalmente desinteresada a la dirección y revisión de este trabajo entregando su tiempo a la lectura y el comentario crítico del presente documento.

A mis padres, por educarme en el Bushidō inculcándome, junto con su cariño, los principios morales sobre los que he llegado a construir mi propia identidad. Y, sobre todo, por cuidar siempre de mí, especialmente cuando mi testarudez me lleva a ignorar los límites que mi cuerpo puede soportar.

A mi hermano Carlos, por ser un buen zagal de quien todos nos sentimos muy orgullosos.

A Edu, por ser un eterno compañero de batallas y siempre un espejo en el que mirarse.

A Alba, por su amistad desde que nos caíamos mal, sus verdades a la cara y las legendarias *albiñas*.

A Salva, por demostrarme su calidad humana, digna de Asgard, que es la que exhibimos no cuando llegamos hasta donde podría exigírsenos, sino cuando elegimos sacrificarnos a nosotros mismos para ir más allá de lo que nadie podría habernos pedido.

A Ainhoa y Raúl, por el aprecio y la comprensión que me han mostrado desde el primer día y por darme todas las facilidades, aun a su propia costa, para que pudiera cursar este máster y tener un tiempo para reencontrarme.

Finalmente, deseo expresar mi más profunda gratitud a Mónica y Verónica, mis doctoras, por la deferencia, la atención, la cercanía y la implicación con la que me han tratado a lo largo de todos estos años, sin importar las circunstancias o lo ajetreado que fuera el día. No es hasta que nos hallamos en la miseria más absoluta que aprendemos a valorar lo que significa poder llevar una vida normal. Por ello, tenemos el deber de vivir al máximo cada día y de dar lo mejor de nosotros mismos en cada aliento para transmitirles a los demás todo lo que se nos ha dado.

Javier Fernández-Bravo Peñuela

Para mis padres,
por velar por mí más allá
de mi propia insensatez

Índice general

Índice de cuadros

Índice de figuras

Índice de algoritmos

Listado de acrónimos

ASIC	Application-Specific Integrated Circuit
BCB	Byzantine Consistent Broadcast
BFT	Byzantine Fault Tolerance
BRB	Byzantine Reliable Broadcast
CAP	Consistency, Availability, Partition resilience
DLT	Distributed Ledger Technology
GST	Global Stabilization Time
KDC	Key Distribution Center
MAC	Message Authentication Code
NIST	National Institute of Standards and Techonology
PBFT	Practical Byzantine Fault Tolerance
PoET	Proof of Elapsed Time
PoS	Proof of Stake
PoW	Proof of Work
SGX	Software Guard eXtensions
SHA	Secure Hash Algorithm
SHARCS	Special-Purpose Hardware for Attacking Cryptographic Systems
TEE	Trusted Execution Environment
XFT	Cross Fault Tolerance
XMSS	eXtended Merkle Signature Scheme

Capítulo 1

Introducción

Los sistemas *blockchain*, como un caso particular de los DLT, se constituyen como un motor de computación distribuida que proporciona confianza descentralizada para mantener de forma segura un estado compartido entre los nodos de un sistema replicado. Estos sistemas han despertado gran interés desde sus comienzos, tanto por su utilización en las divisas criptográficas, de las que Bitcoin es el máximo exponente, como por su aplicación a la hora de mantener un registro replicado y consistente de documentos y transacciones. En este capítulo se realiza una introducción a los sistemas blockchain, describiendo los dos tipos principales de blockchains que existen: permisionados y no permisionados. En el caso de los blockchains permisionados, el registro replicado en la forma del estado compartido entre las réplicas se mantiene mediante algoritmos de consenso, que son descritos seguidamente. Dichos algoritmos, que constituyen la temática principal del presente trabajo, se sirven de la replicación para dotar al sistema de tolerancia a fallos bajo diversos modelos preservando la integridad y consistencia del estado compartido.

El presente trabajo versa sobre Trebizond, un algoritmo de búsqueda de consenso tolerante a fallos bizantinos especialmente apropiado para su aplicación en blockchains permisionados, cuyo diseño realiza algunas aportaciones interesantes respecto al estado actual de la investigación en esta materia.

1.1 Introducción a blockchain

Un blockchain es un sistema distribuido que proporciona un servicio de confianza a un grupo de nodos que no confíen completamente los unos en los otros. Se utiliza para registrar transacciones o documentos agrupados en bloques, los cuales mantienen un conjunto de los nodos que forman el sistema mediante un protocolo criptográfico distribuido que no depende de ninguna autoridad o componente centralizado. Cada bloque contiene un *hash* criptográfico del bloque anterior, integrando así una representación segura del historial de la cadena en cada bloque, que determina cuál es la secuencia de bloques precedente. El registro de los bloques conforma el estado compartido del sistema, que se mantiene a través de un protocolo de consenso que garantiza el establecimiento de un orden común y no ambiguo de las transacciones y los bloques en las que éstas se engloba, asegurando la integridad

y consistencia de la cadena de bloques en un dominio geográficamente distribuido. Así, un blockchain se constituye como un motor de computación distribuida que proporciona confianza descentralizada para mantener de forma segura un estado compartido entre los nodos del sistema. [CV17] [Bal17]

Dado que el sistema blockchain en su conjunto actúa como un proveedor de confianza distribuida, debería ser fiable, resistente y seguro, cumpliendo con las propiedades de disponibilidad, fiabilidad, seguridad, confidencialidad e integridad, tal como se describen en [ALRL04]. La adopción de la replicación, conformando el sistema distribuido, permite a los sistemas basados en blockchain garantizar las propiedades mencionadas, a diferencia de otros tipos de sistemas replicados que buscan mejorar su escalabilidad.

Aunque la tecnología blockchain se encuentra ligada desde sus orígenes a las divisas criptográficas, en particular Bitcoin, puede utilizarse en cualquier escenario en el que se necesite mantener un registro incremental de documentos que cumpla con un determinado modelo de consistencia e integridad. Por ello, normalmente se desvincula el protocolo de búsqueda de consenso, responsable de garantizar la consistencia e integridad del estado compartido, de la naturaleza de este estado y de las acciones a realizar por el sistema para añadir un nuevo elemento a la cadena de bloques. Muchos sistemas basados en blockchain implementan esta separación permitiendo la ejecución de tareas genéricas (denominadas *smart contracts*) implementadas en un lenguaje de programación, ya sea específico al dominio de la plataforma blockchain o de propósito general. [GKL15]

1.2 Tipos de blockchain

Existen dos tipos principales de blockchain de acuerdo a los privilegios necesarios para añadir nuevos bloques a la cadena: blockchains permisionados y blockchains no permisionados.

1.2.1 Blockchain no permisionados

En un blockchain no permisionado, como Bitcoin o Ethereum, cualquier entidad identificada por un par de claves (utilizando criptografía asimétrica) puede unirse al sistema, proponer bloques para su adición a la cadena y participar en el proceso de consenso que determina cuál es el estado válido del sistema. En un entorno no permisionado, se espera que el número de nodos que conformen el sistema sea elevado. Dado que los participantes son anónimos, únicamente identificados por su clave pública, y por lo tanto potencialmente no confiables a nivel individual, los mecanismos utilizados para alcanzar una situación de consenso sobre la cadena de bloques deben estar diseñados para resistir ataques que puedan pretender subvertir el sistema y condicionar cuáles son los bloques que se añaden.

Este tipo de sistemas, en los que una serie de participantes no autenticados e identificados únicamente por su clave pública determinan el estado del sistema, ya sea por votación directa o difundiendo el bloque de su elección mediante un algoritmo de *gossiping*, son vulnerables a ciertos ataques, de los que más característico es el ataque Sybil. [Dou02] describe este ataque, según el cual sin una autoridad centralizada de confianza que certifique una correspondencia unívoca entre identidades y entidades, siempre existe la posibilidad de que una entidad presente múltiples identidades, salvo en el caso de condiciones poco realistas que no se ajustan al escenario de un sistema distribuido de gran escala. Si una única entidad genera múltiples de pares de clave asimétrica, lo que puede llevar a cabo con mínimo esfuerzo, puede presentar tantas identidades diferentes como claves haya generado, representando a una fracción del sistema mucho mayor de la que constituye realmente por sí mismo y pudiendo por lo tanto influir o condicionar el resultado de las decisiones a tomar de forma colectiva. Ante la ausencia de una autoridad certificadora de confianza u otro mecanismo que proporcione una autenticación directa entre las partes, la capacidad de una entidad para diferenciar entre otras identidades remotas se basa en la suposición de que los recursos de un atacante son limitados. Una forma de validar la singularidad de una identidad es planteando desafíos que exijan el uso intensivo de recursos de cómputo ajustado a un determinado umbral. Además de la identificación única de entidades en los blockchains públicos, como se verá a continuación, otro escenario de aplicación de esta técnica es la protección frente a ataques de denegación de servicio, ya que obliga al atacante a realizar un mayor volumen de trabajo que el recurso atacado. [Ora03]

En el caso de Bitcoin, el mecanismo que se emplea para determinar las transacciones aceptadas y establecer su orden es el de prueba de trabajo (PoW, *Proof of Work*), exigiendo que los nodos consuman una cantidad significativa de energía resolviendo un problema criptográfico de complejidad NP-completo, lo que se conoce como proceso de minado [Nak08]. En este proceso, cada nodo tiene que encontrar un valor de *hash* menor que un determinado número, que se ajusta en función del tiempo que se pretende que dure el proceso de minado. El primer nodo que calcule un *hash* que cumpla con las condiciones lo añade a la cadena y reclama una recompensa en forma de bitcoins. Cada nodo ganador, tras añadir su bloque a la cadena, lo propaga al resto de la red. En caso de que más de un nodo haya encontrado una solución de forma simultánea se produce una bifurcación en la cadena, con el resultado de que cada nodo continúa añadiendo bloques a una de las dos ramas, condicionada por la cercanía en la red entre los nodos durante la difusión de los nuevos bloques. Sin embargo, el protocolo garantiza que, conforme las ramas crezcan, todos los nodos terminarán optando por la rama más larga, descartando las ramas restantes, lo que se corresponde con un modelo de consistencia eventual.

Aunque el mecanismo de prueba de trabajo de Bitcoin se basa en los incentivos que se otorgan por minar bloques y conseguir que éstos se propaguen y añadan a la cadena, dicho mecanismo es vulnerable a ataques de 51 % [ES18]. En este tipo de ataque, un porcentaje de la potencia computacional del sistema superior al 50 % se confabula para determinar qué bloques se añaden a la cadena, propagando sólo aquellos de su interés, y pudiendo así favorecer o penalizar a los participantes en base a sus intereses. Esto subvierte el propósito de la confianza distribuida, cuya principal virtud es su descentralización. Una de las formas de lograr un ataque de 51 % en un sistema con un alto número de participantes es la utilización de circuitos integrados de aplicación específica (ASIC, *Application-Specific Integrated Circuit*), diseñados especialmente para calcular *hashes* y aprovechar técnicas de paralelismo en hardware. Estos dispositivos proporcionan una amplia ventaja sobre los dispositivos de propósito general, creando una considerable brecha en función de los dispositivos utilizados por los mineros participantes que puede favorecer la consecución de un ataque de 51 %. El modelo de prueba de trabajo de Ethereum, denominado EthHash, permite tiempos de confirmación menores que Bitcoin y es menos susceptible a ataques de 51 % que Bitcoin, ya que la naturaleza de los problemas numéricos a calcular es resistente a su resolución mediante ASICs [ES18]. Además, el sistema de incentivos de Ethereum beneficia la inclusión de bloques huérfanos, pertenecientes a ramas descartadas, favoreciendo la participación en el proceso de minado frente a un monopolio de la cadena de bloques. Aunque estas técnicas dificultan realizar ataques de 51 % con éxito, el sistema sigue siendo vulnerable, por lo que Ethereum actualmente trabaja en los preparativos de su migración al mecanismo de prueba de participación (PoS, *Proof of Stake*) [Bal17]. En este mecanismo, la probabilidad de encontrar un bloque de transacciones es directamente proporcional a la cantidad de divisas acumuladas por el participante, si bien esto equivale a una cierta centralización sobre los principales propietarios de divisa, lo que tampoco es deseable. Otra de las desventajas del mecanismo de prueba de trabajo incorporado en Bitcoin es que, aunque facilita una participación abierta y altamente escalable incluso aunque los nodos se encuentren físicamente distribuidos a nivel global, no se adecúa a sistemas que requieran una alta productividad y completar transacciones de forma inmediata. Además, el minado en la resolución de estos problemas numéricos representa un enorme gasto energético cuya única finalidad es la obtención de confianza distribuida; de hecho, el estudio realizado en [OM14] estima que el volumen de energía eléctrica invertido en el minado de Bitcoins en 2014 era equivalente al consumo eléctrico medio Irlanda.

Además de las técnicas de prueba de trabajo y prueba de participación citados, otra técnica par alcanzar consenso distribuido en la ausencia de un mecanismo de autenticación directa de los participantes es la técnica de prueba de tiempo consumido (PoET, *Proof of Elapsed Time*). Este mecanismo es empleado por Sawtooth y su implementación depende de un entorno de ejecución seguro (TEE, *Trusted Execution Environment*), como el SGX (*Software Guard eXtensions*) de Intel, al que los participantes deben solicitar un temporizador aleatorio firmado, que determina cuándo el nodo puede propagar un nuevo bloque al resto.

1.2.2 Blockchain permisionados y consenso

Como contraste a los blockchains no permisionados, en los blockchains permisionados el proceso de adición de bloques a la cadena y búsqueda de consenso se lleva a cabo por entidades conocidas e identificadas, las cuales actúan como participantes. En el caso más típico, los miembros o partes interesadas de un consorcio operan en una red de blockchain permisionado. Estos sistemas blockchain cuentan con medios para identificar los nodos que pueden controlar y actualizar el estado compartido por el conjunto de nodos de la red, así como limitar quién puede ordenar transacciones, imponiendo también un control sobre las identidades de los clientes. Un caso particular es el de los blockchains privados, que se definen como un blockchain permisionado sobre el que sólo opera una entidad, en el contexto de un único dominio de confianza. [CV17]

En un blockchain permisionado todos los participantes conocen las identidades de los demás nodos de la red, y normalmente también las de los clientes autorizados a emitir peticiones de realización de transacciones sobre el sistema. Dado que cada una de estas entidades, nodos del sistema y clientes, se identifican por su clave pública, las claves públicas de todos los nodos se habrán distribuido previamente a la operación del sistema por algún medio, como un centro de distribución de claves. Este centro de distribución de claves (o el mecanismo utilizado por el sistema en cuestión) actúa como un tercero de confianza que enuncia las identidades de aquellas entidades que están autorizadas a operar y la forma en la que pueden hacerlo. La presencia de un tercero de confianza permite evitar las dificultades enunciadas en [Dou02], que en el caso de los blockchains no permisionados obligan a incluir mecanismos que eviten o inutilicen la realización de ataques Sybil que creen múltiples identidades asociándolas a una única entidad real.

Los blockchains permisionados se plantean como un escenario de aplicación de la búsqueda de consenso, uno de los problemas más ampliamente tratados en el contexto de los sistemas distribuidos, que permite la actualización segura de un estado replicado sobre un sistema distribuido, manteniendo en todo momento unas garantías de consistencia e integridad. A la hora de establecer un protocolo de búsqueda de consenso es importante especificar el *modelo de fallos* o las *suposiciones sobre el entorno*, sobre el que dicho protocolo está diseñado y preparado para operar, junto con las garantías de consistencia, seguridad y viveza que éste satisface. Las características del entorno sobre el que operan los sistemas blockchain, en el que la red puede ser inestable, los nodos pueden detenerse o ser corrompidos por un atacante, y las interacciones son inherentemente asíncronas, unido al potencial valor de los datos con los que opera un blockchain, implican que éste debe tolerar el modelo de fallos más estricto posible. Esto, unido al interés que la tecnología blockchain ha suscitado en los últimos años, ha provocado el resurgir de una corriente de investigación en el contexto de los sistemas de búsqueda de consenso, específicamente en el caso de los algoritmos de consenso tolerantes a fallos bizantinos, que se describirán en el capítulo 5. [CV17]

El estudio formal y el desarrollo de algoritmos que hacen uso de la replicación para construir sistemas distribuidos tolerantes a fallos se remonta a la introducción del concepto de acuerdo bizantino postulado por Lamport *et al.* [PSL80] [LSP82]. Tal como describe Schneider [Sch90], la tarea de alcanzar y mantener un estado de consenso entre un conjunto de nodos distribuidos consta de dos elementos: una máquina de estados determinista que implemente la lógica del servicio a replicar y un protocolo de consenso que difunda las peticiones entre los nodos, de forma que cada nodos ejecute la misma secuencia de peticiones en su propio ejemplar del servicio. En la literatura, tradicionalmente se define como «consenso» la tarea de alcanzar un acuerdo respecto a una única petición, mientras que la «difusión atómica» [HT94] proporciona un acuerdo sobre una secuencia de peticiones, tal como lo requiere una máquina de estados replicada. Puesto que existe una conexión muy estrecha entre ambos, ya que una secuencia de instancias de consenso permite alcanzar difusión atómica, el término «consenso» suele utilizarse para referirse a la difusión atómica, especialmente en el contexto de los blockchains.

Igualmente, para alcanzar consenso sobre un conjunto de nodos replicados en un sistema distribuido tolerante a fallos, garantizando que todas las réplicas se coordinan de tal forma que reciben y procesan la misma secuencia de peticiones, deben satisfacerse las dos siguientes condiciones: [Sch90]

- **Acuerdo:** Todas las réplicas correctas reciben todas las peticiones.
- **Orden:** Todas las réplicas correctas procesan las peticiones recibidas en el mismo orden.

1.3 Máquinas de estados y consenso distribuido

El enfoque de máquina de estados es un método bien conocido para implementar un servicio tolerante a fallos replicando los servidores y coordinando las interacciones de los clientes con las réplicas del servidor, de tal forma que el conjunto de las réplicas se comporte como un único servicio centralizado [Sch90]. A nivel lógico, una máquina de estados consiste en *variables de estado*, que contienen el estado del sistema, y *comandos*, que transforman dicho estado. Cada comando se implementa por medio de un programa determinista, que modifica las variables de estado y/o produce una salida, ejecutándose de forma atómica respecto a otros comandos. Los clientes de la máquina de estados realizan peticiones que se traducen en la ejecución de comandos. Una de las formas más comunes de implementar la ejecución de los comandos es mediante un modelo de programación reactiva, incluyendo manejadores de evento que ante la recepción de un comando siguen un modelo de comportamiento (en lo externo) similar a las rutinas asociadas a las interrupciones del kernel de un sistema operativo. Las peticiones de los clientes se procesan de forma secuencial en un orden consistente con las relaciones de orden causal, que reflejan los relojes lógicos que, junto con el propio concepto de replicación de una máquina de estados, fueron presentados en [Lam78b].

La implementación de una máquina de estados replicada debe cumplir las siguientes tres propiedades: [Sch90]

- **Estado inicial:** Todas las réplicas correctas comienzan en el mismo estado.
- **Determinismo:** Todas las réplicas correctas que reciban una misma entrada encontrándose en un mismo estado producen la misma salida y transitan al mismo estado común.
- **Coordinación:** Todas las réplicas correctas procesan la misma secuencia de comandos, en el mismo orden.

A partir del cumplimiento de estas propiedades, pueden enunciarse las siguientes propiedades de seguridad y viveza:

- **Seguridad:** Todas las réplicas correctas ejecutan la misma secuencia de operaciones, en el mismo orden.
- **Viveza:** Todas las operaciones emitidas por un cliente correcto y entregadas a los servidores son ejecutadas.

La propiedad de seguridad citada se encuentra estrechamente ligada al concepto de difusión atómica [HT94] que permite alcanzar consenso sobre una secuencia de operaciones, y permite la implementación de servicios acordes a un modelo de consistencia estricta, cumpliendo con la propiedad de consistencia denominada *linearizabilidad*. [HW90]

El propósito de utilizar replicación para implementar una máquina de estados sobre un sistema distribuido es el de ofrecer tolerancia a fallos. Se considera que un componente falla cuando su comportamiento deja de cumplir su especificación. Por consiguiente, se considera que un sistema es tolerante a fallos cuando la ocurrencia de un fallo enmascara el defecto en sus componentes y continúa exhibiendo un comportamiento bien definido y consistente con su especificación. Sin embargo, los componentes del sistema (que comprenden tanto las réplicas de los nodos servidores, los clientes y los canales de comunicación) pueden fallar de diferentes formas, que se corresponden con diferentes clases de modelos de fallos. Por ello, a la hora de diseñar un algoritmo de búsqueda de consenso sobre una máquina de estados replicada debe tenerse en cuenta el modelo de fallos tolerado por el algoritmo. [Cri91a]

Principalmente, se consideran dos modelos de fallos particularmente representativos:

- **Fallos de parada:** Ante un fallo, el componente cambia de estado, deteniéndose, de tal forma que este hecho puede ser detectado por otros componentes del sistema. [Sch84]
- **Fallos bizantinos:** El componente exhibe un comportamiento arbitrario y posiblemente malicioso, de una forma que puede no ser detectable por otros componentes del sistema. [LSP82]

Los fallos de naturaleza bizantina pueden ser los más disruptivos, debido a que pueden ocurrir por muchos motivos y manifestarse de cualquier forma, o incluso no manifestarse en absoluto, por lo que, a la hora de considerar el modelo de fallos que tolerará un sistema, el modelo de fallos bizantino es el más severo. Como se comentó en la sección 4.2.2, los sistemas basados en blockchain deben tolerar el modelo de fallos más severo posible, motivo por el que los algoritmos de búsqueda de consenso tolerantes a fallos bizantinos han despertado un creciente interés en los últimos años.

1.4 Las máquinas de estados replicadas como sistemas distribuidos tolerantes a fallos

Un sistema consistente en un conjunto de componentes se considera tolerante a f fallos si es capaz de exhibir un comportamiento que satisfaga sus especificaciones en caso de que un número de sus componentes menor o igual a f se encuentre simultáneamente en un estado fallido.

Además de la naturaleza de los fallos que pueden afectar a los procesos del sistema, en el contexto de un sistema distribuido también han de contemplarse los fallos que pueden afectar a los canales de comunicación utilizados para la interconexión de los procesos. Los principales modelos de fallos que se establecen sobre los canales de comunicación son los siguientes: [BA14]

- **Canal fiable:** Todo mensaje enviado será entregado.
- **Canal no fiable:** Los mensajes pueden ser alterados, generados espuriamente o perderse.
- **Canal con pérdidas:** Los mensajes pueden perderse, pero no alterarse.
- **Canal justo:** Si un mensaje se reenvía de forma indefinida, terminará alcanzando su destino.

En el caso de Internet, éste se constituye como una red de *mejor esfuerzo*, es decir, que tratará de que los mensajes alcancen su destino, pero sin poder ofrecer garantías; por lo tanto, los mensajes pueden perderse. La posibilidad de que los mensajes puedan ser alterados o creados de forma ilegítima depende de si las técnicas criptográficas utilizadas garantizan la autenticación del remitente y la integridad de los mensajes, como se verá en la sección 5.5.

En lo que se refiere a la tolerancia a fallos, es importante tener en cuenta la afirmación implícita de que no existe relación entre los fallos, es decir, que la probabilidad estadística de que un componente falle es independiente del fallo en otro componente del sistema. Para cumplir esta propiedad es conveniente desplegar las réplicas sobre diferentes plataformas y localizaciones, que dependan de diferentes fuentes de suministro eléctrico y diferentes infraestructuras de red y, si es posible, utilizando diferentes versiones de diseño e implementación

de los componentes. Este último aspecto es especialmente importante en el caso de los sistemas tolerantes a fallos bizantinos, ya que reduce las probabilidades de que un defecto o vulnerabilidad en el diseño o la implementación de un determinado componente comprometa todo el sistema.

Además del modelo de fallos asumido, la tolerancia a fallos de un sistema distribuido se sustenta sobre las premisas de las que se parta respecto a su modelo de sincronía y a las técnicas criptográficas utilizadas y aplicadas sobre los mensajes.

La sincronía hace referencia a los aspectos de temporización del sistema en relación al procesamiento de los mensajes y su envío a través de la red, estando por lo tanto ligado a la naturaleza de ésta. Si bien existe un número considerable de modelos de sincronía, los tres más relevantes en el contexto de las máquinas de estado replicadas son los siguientes: [AW04]

- En el **modelo de comunicación asíncrona**, no existen límites acotados sobre los tiempos de procesamiento y retardo en el envío de los mensajes. [FLP85]
- En el **modelo de comunicación síncrona**, existen límites acotados sobre los tiempos de procesamiento, retardo en el envío de los mensajes, y margen de error posible en los relojes que controlan los procesos. [Lyn96]
- El **modelo de comunicación parcialmente síncrona** es un modelo intermedio que considera que el sistema se comporta de forma asíncrona durante la mayor parte del tiempo, hasta que un cierto momento denominado GST (*Global Stabilization Time*) en el que el sistema se comporta de forma síncrona, estableciendo límites sobre los tiempos de procesamiento y envío de mensajes, durante un intervalo o la ejecución de un determinado protocolo. [DLS88]

Igualmente, como ya se ha mencionado, el escenario de operación de un sistema distribuido está condicionado por las técnicas criptográficas utilizadas, que definirán si se opera sobre canales de comunicación autenticados. Esto se consigue utilizando técnicas de clave pública que, mediante el uso de criptografía asimétrica o la existencia de secretos compartidos entre cada par de procesos, permitan firmar los mensajes enviados, garantizando autenticación, integridad y no repudio. En ambos casos las primitivas que definen el algoritmo de firma digital deben emplazarse sobre funciones de *hash* resistentes a colisiones.

Finalmente, a la hora de elaborar un protocolo que pretenda operar sobre una máquina de estados replicada han de tenerse en cuenta ciertas restricciones inherentes a los sistemas distribuidos que pueden condicionar el espacio de decisiones de diseño:

1. No se puede establecer una comunicación fiable utilizando el modelo de paso de mensajes sobre canales no fiables. Este resultado de imposibilidad se plantea en [AEH75], a través del *problema de los dos generales*.

2. El establecimiento de orden total equivale a alcanzar una situación de consenso [HT94]. Como ya se ha expuesto, en el enfoque de una máquina de estados replicada las peticiones de los clientes deben difundirse a los nodos que constituyen las réplicas del servicio. Las réplicas deben coordinarse de tal forma que reciban los mensajes correspondientes a las peticiones en el mismo orden, lo que se traslada a la necesidad de contar con un protocolo de difusión atómica. La implementación de este tipo de primitiva es equivalente a resolver el problema de consenso, en el que cada proceso propone un valor, entre los cuales todos los procesos deben alcanzar un acuerdo relativo al valor a elegir.
3. No se puede alcanzar consenso en un sistema totalmente asíncrono ante la presencia de fallos. Esta afirmación de imposibilidad resulta de la dificultad que implica detectar e identificar un fallo (por ejemplo, la parada de un nodo) en un entorno puramente asíncrono, ya que la ausencia de relojes o consideraciones temporales impide saber si un proceso ha fallado o simplemente está operando con extrema lentitud. Esto imposibilita alcanzar las condiciones de acuerdo y finalización (seguridad y viveza) necesarias para lograr consenso. [FLP85]
4. Como consecuencia de la imposibilidad de alcanzar consenso en un entorno puramente asíncrono, [DLS88] establece las condiciones de sincronía mínimas sobre las que un sistema tolerante a fallos puede alcanzar una situación de consenso. A partir de estas condiciones, la publicación citada define un modelo de sistema parcialmente síncrono, que permite alcanzar consenso realizando una separación entre las condiciones de seguridad y viveza del protocolo.
5. En función del modelo de sistema adoptado en el proceso de diseño, se requiere un número n diferente de procesos (nodos) para alcanzar consenso y difusión de orden total tolerando hasta f fallos. En [Lam00] y [Lam03], Lamport establece estos umbrales mínimos en función del modelo de sincronía, el modelo de fallos y la presencia de mecanismos criptográficos de autenficación y de integridad.

1.5 Estructura del documento

El presente trabajo trata el diseño y la construcción de Trebizond, un algoritmo de búsqueda de consenso tolerante a fallos bizantinos para su aplicabilidad en blockchains permisionados. Este documento sigue la siguiente estructura:

1. **Introducción.** El capítulo actual ha realizado un recorrido por los sistemas blockchain. Tras establecer una clasificación entre los principales tipos de plataformas que hacen uso de esta tecnología, se evidencia cómo los sistemas blockchain se fundamentan sobre la existencia de una técnica que permita alcanzar consenso distribuido. Centrando el análisis en el caso de los blockchains permisionados, se realiza un resumen introductorio sobre las características de los algoritmos de consenso que operan en este tipo de plataformas, puesto que el objetivo del presente trabajo es el diseño y desarrollo de un algoritmo de tal naturaleza.

2. **Antecedentes y estado del arte.** Tomando el relevo del resumen realizado en la introducción acerca de los algoritmos de consenso en el ámbito de los sistemas blockchain permisionados, se realiza un estudio cronológico detallado sobre el consenso, uno de los problemas fundamentales de los sistemas distribuidos, citando las principales publicaciones y aportaciones. Este estudio, junto con el de otros aspectos de los sistemas distribuidos estrechamente ligados al problema del consenso, permiten establecer las bases sobre las que justificar las decisiones tomadas en el resto del trabajo y argumentar sus aportaciones respecto al estado actual de la investigación de este problema.

3. **Trebizond, algoritmo de consenso bizantino para sistemas blockchain permisionados.** Tras presentar las bases sobre las que se asientan los algoritmos distribuidos de consenso tolerante a fallos bizantinos y el estado actual de la investigación en este área, se define la meta del trabajo: el desarrollo de Trebizond, un algoritmo de consenso tolerante a fallos bizantinos y apropiado para su utilización en plataformas blockchain permisionadas. Una vez que se han detallado los objetivos del algoritmo, su entorno de operación, aportaciones y limitaciones, el capítulo procede a documentar su diseño.

4. **Modelo de implementación.** Una vez que se ha definido, descrito y argumentado el algoritmo diseñado, se aborda una implementación del algoritmo realizada como caso práctico. La motivación de este capítulo no es abordar los detalles de bajo nivel de dicha implementación, sino exponer modelos y técnicas de programación que contribuyen a simplificar el proceso de trasladar el diseño de un algoritmo distribuido a una implementación funcional y a establecer una frontera bien definida y a la vez versátil entre el algoritmo y el sistema de alto nivel que se sirve de éste.

5. **Conclusiones y trabajo futuro.** Finalmente, se detallan las conclusiones que se extraen tras la realización del trabajo expuesto, los principales aspectos a destacar *a posteriori* y se enumeran posibles líneas de investigación futura a partir del trabajo realizado, así como elementos que se hayan quedado fuera del ámbito de este trabajo y cuya materialización pudiera resultar útil e interesante.

Capítulo 2

Antecedentes y estado del arte

El análisis de la investigación realizada hasta el momento actual respecto al objeto de estudio del presente trabajo comienza con una breve crónica superficial de las técnicas empleadas para alcanzar acuerdo y consenso tolerante a fallos tanto bajo el modelo de parada como bajo el modelo de fallos bizantinos. La obtención de consenso es uno de los problemas fundamentales de los sistemas distribuidos, y su búsqueda bajo diferentes escenarios y contextos es una materia de estudio en sí misma, que se sustenta sobre otros problemas típicos de este tipo de sistemas. En este capítulo se han escogido cuatro de los problemas más relevantes que deben tomarse en consideración a la hora de alcanzar consenso distribuido en la presencia de fallos bizantinos: sincronía, consistencia, detección de fallos y criptografía. Por ello, las subsiguientes secciones analizan cada uno de ellos desde una perspectiva cronológica, destacando las principales aportaciones en sus respectivas áreas que han repercutido en un avance en los algoritmos de búsqueda de consenso y las máquinas de estado replicadas. Estos problemas se presentan de forma independiente con el objetivo de poder detallarlos con mayor claridad que si se expusieran de forma conjunta, estudiándolos en el orden que se considera que más contribuye a facilitar su comprensión incremental. Aun así, tales problemas se encuentran estrechamente relacionados, sobre todo en el caso de los tres primeros, por lo que el lector encontrará abundantes referencias cruzadas entre ellos conforme avance en el texto. Tras dicho estudio, el capítulo concluye ofreciendo un recorrido más detallado sobre los algoritmos tolerantes a fallos bizantinos aplicables a blockchains permisionados.

2.1 Breve historia: De los relojes lógicos al consenso bizantino

El concepto de máquina de estados replicada surgió en 1978 a partir de la descripción de los relojes lógicos por parte de Lamport [Lam78b]. No obstante, aunque dicha publicación es una de las más relevantes en la historia de los sistemas distribuidos, el algoritmo presentado en ella no es tolerante a fallos. Por este motivo, en el mismo año Lamport presentó el primer algoritmo para una máquina de estados replicada tolerante a fallos en un entorno síncrono [Lam78a]. Esto propició durante los años ochenta y noventa una corriente de investigación dirigida a los protocolos de difusión [BSS91] [HT94] y sistemas de comunicación a grupos [CKV01] [Pow96], utilizados entre otras aplicaciones para la implementación de

máquinas de estados replicadas. Estas aportaciones sentaron las bases para la publicación de Schneider [Sch90], que constituye un texto fundamental acerca de las máquinas de estados replicadas en el que se detalla este enfoque describiendo los elementos de los que se compone y los protocolos requeridos para su implementación. Tales esfuerzos cristalizaron en dos de los algoritmos clásicos más representativos dirigidos a máquinas de estados replicadas: el algoritmo Paxos [Lam98] (diseñado inicialmente en 1989) y el algoritmo *Viewstamped Replication* [OL88].

Aunque la finalidad con la que los algoritmos referidos utilizan las técnicas de replicación es la de obtener tolerancia a fallos y alta disponibilidad, ninguno de ellos proporciona tolerancia a fallos bizantinos [LSP82]. En este modelo de fallos, un número de componentes acotado superiormente puede exhibir un comportamiento arbitrario y potencialmente malicioso, no consistente con su especificación. Ante la posibilidad de que este comportamiento se deba a una intrusión, debe considerarse que un componente fallido trate de sabotear el algoritmo, impidiendo su progreso o subvirtiendo su funcionamiento (violando sus condiciones de seguridad), por ejemplo, suplantando la identidad de otros componentes en ausencia de mecanismos que lo impidan.

Pese a la publicación de numerosos trabajos dedicados a alcanzar acuerdo en presencia de fallos bizantinos o implementar máquinas de estados replicadas tolerantes a ellos, la mayor parte de los algoritmos presentados en las publicaciones iniciales en este campo resultan poco prácticos, en unos casos por estar completamente orientados a una demostración teórica de posibilidad y factibilidad, pero ser demasiado ineficientes como para ser utilizados en la práctica [CR92] [MR97b] [GM98], y en otros casos por operar bajo suposiciones irreales en el mundo real, como basar su corrección en el hecho de operar sobre entornos completamente síncronos [Rei95] [KMMS98].

La primera publicación que presenta un algoritmo que permite implementar una máquina de estados replicada que ofrece tolerancia a fallos en un escenario bizantino, de forma probada, sobre un escenario aplicable y con una pérdida de rendimiento asumible corresponde a PBFT (*Practical Byzantine Fault Tolerance*), creado por Castro y Liskov [CL+99]. Al igual que Paxos, este algoritmo no requiere de un entorno síncrono para asegurar su corrección, aunque sí se basa en la sincronía eventual para asegurar su progreso como una forma de evitar el resultado de imposibilidad referente a los sistemas puramente asíncronos presentado en [FLP85].

El gran impacto que supuso la publicación del algoritmo PBFT provocó el advenimiento de un gran número de sucesores [AEMGG+05] [CML+06] [KAD+09] optimizados para diversos escenarios. Algunas de las optimizaciones propuestas en [CL+99] han tenido amplia acogida, como la posibilidad de implementar soporte especial para las operaciones de sólo lectura, cuya resolución puede realizarse más rápidamente que las operaciones que implican una modificación del estado compartido. Otras optimizaciones propuestas posterior-

mente han contribuido a refinar el diseño de este tipo de algoritmos, como la división del acuerdo (establecimiento de orden total) y la ejecución de peticiones en dos capas separadas [YMV+03], que pueden implementarse con un número diferente de réplicas, conllevando importantes beneficios [KJ10].

En los últimos años se ha producido un resurgir del interés en los algoritmos de búsqueda de consenso debido a su aplicación en el caso particular de sistema replicado que constituyen los blockchains.

Un ejemplo de estos algoritmos que ha atraído considerable atención desde su publicación es Raft [OO14], un algoritmo que se presenta como una alternativa al dominio esgrimido por Paxos en el ámbito de la implantación de algoritmos de búsqueda de consenso. Raft pretende solventar los problemas más significativos que presenta Paxos, principalmente por la difícil comprensión del algoritmo debido a su gran dificultad, como al hecho de que para adaptarse a casos reales el modelo de implementación suele alejarse de la especificación original de la arquitectura, lo que propicia la aparición de errores y casos no probados. Para ello, Raft opta por descomponer el algoritmo de búsqueda de consenso en los problemas de elección de líder, replicación del estado compartido y de la secuencia de operaciones ejecutada y cumplimiento de las condiciones de seguridad. Además de esta descomposición, el algoritmo trata de reducir en la medida de lo posible la dimensión del espacio de estado a considerar, simplificando su comprensión e implementación, especialmente en comparación con Paxos.

La arquitectura de Raft es considerablemente similar a la de *Viewstamped Replication*, presentando en contraposición algunos elementos distintivos, como el mayor protagonismo del líder como figura centralizadora en la toma de decisiones que permiten alcanzar consenso, el uso de temporizadores aleatorios a la hora de proponer la elección de posibles líderes y la posibilidad de añadir o eliminar nodos de la red de consenso sin detener la operación del algoritmo. Dado que Raft está diseñado para operar con el modelo de fallos de parada, al igual que la versión clásica de Paxos, permite alcanzar consenso con una mayoría simple de nodos honestos, debiendo contar con $2f + 1$ réplicas para tolerar hasta f fallos.

El hecho de que Raft no ofrezca tolerancia ante fallos bizantinos es una de sus mayores desventajas. Sus dos principales debilidades devienen de la forma en la que se postula y elige un nuevo líder, así como de la excesiva centralización de la operativa que se delega en la figura de un líder, puesto que el sistema necesita la presencia de un líder electo para progresar. En un hipotético escenario bizantino, un líder bizantino comprometería la seguridad y el progreso de todo el sistema, mientras que un único nodo bizantino, aunque no fuera elegido como líder, podría detener el progreso de todo el sistema proclamando nuevas elecciones de líder continuamente. Un algoritmo que trata de proporcionar tolerancia a fallos bizantinos heredando el espíritu de Raft en cuanto a separación de problemas, inteligibilidad y reducción del espacio de estados es BFT-Raft, también conocido como Tangaroa [CZ14], que combina los

aspectos positivos de este algoritmo con algunos de los conceptos presentados en PBFT. Si bien este algoritmo refleja algunas buenas ideas que pueden servir de inspiración a la hora de construir algoritmos de consenso tolerantes a fallos bizantinos, también presenta errores de diseño de notable gravedad que hacen que fracase en su empeño. Pese a estas deficiencias, BFT-Raft se postula como un punto de partida interesante para el diseño de un algoritmo de búsqueda de consenso tolerante a fallos bizantinos especialmente aplicable para sistemas blockchain permisionados, motivo por el cual será estudiado en mayor detalle en la sección 5.6.

2.2 Sincronía

Tal como se enunció en la introducción, uno de los aspectos más importantes a la hora de clasificar la tolerancia de un sistema distribuido de acuerdo a un determinado modelo de fallos es el modelo de sincronía sobre el que opera dicho sistema. Dicho modelo de sincronía se establece a partir de las afirmaciones que se pueden realizar respecto a los aspectos temporales del sistema, principalmente derivados de las comunicaciones, aunque también se consideran a este efecto los tiempos de computación de los procesos que componen el sistema distribuido.

Explorando ambos extremos del estudio de la sincronía se podrían encontrar los siguientes dos modelos: [AW04]

- **Modelo síncrono:** Existen límites acotados sobre los tiempos de procesamiento, el retardo en el envío de los mensajes y el margen de error posible en los relojes que controlan los procesos. Este modelo se corresponde con la representación de un sistema de tiempo real, como la solución propuesta por Lamport en [Lam84], que implementa un sistema distribuido síncrono tolerante a fallos bizantinos.
- **Modelo asíncrono:** No existen límites acotados sobre los tiempos de procesamiento y retardo en el envío de los mensajes, operándose con independencia de cualquier noción temporal.

El resultado de imposibilidad presentado por Fischer, Lynch y Patterson demuestra que los problemas de computación distribuida tolerante a fallos, como es el caso de la búsqueda de consenso, no pueden resolverse en un escenario puramente asíncrono. Par alcanzar esta imposibilidad no es necesario considerar los modelos de fallos más exigentes, como el modelo bizantino, sino basta con considerar el modelo de fallos de parada sobre canales de comunicación fiables. En este contexto, la caída de un único proceso puede comprometer la viveza del algoritmo, causando que éste sea incapaz de progresar o avanzar en su desempeño. La ausencia de cualquier referencia temporal, es decir, la imposibilidad de realizar afirmación alguna en cuanto a las velocidades relativas de cómputo de los procesos o a la latencia en el envío de los mensajes, implica que los procesos no pueden recurrir a ninguna solución basa-

da en *timeouts*, como las que se presentan en [DS82]. Esto impide detectar la parada de un proceso, ya que la ausencia de referencias temporales imposibilita determinar si la ausencia de respuesta por parte de un proceso se debe a que éste se ha detenido por completo o que simplemente se está comportando con extrema lentitud. [FLP85]

Los sistemas a los que afecta esta imposibilidad suelen circunvalarla mediante dos técnicas diferentes: adoptando un enfoque probabilista basado en aleatorización o relajando las restricciones de sincronía del sistema.

La primera opción consiste en incorporar una componente aleatoria en el algoritmo, de tal forma que éste no sea completamente determinista y, mediante una serie de selecciones aleatorias, se reduzca a cero la probabilidad de que el algoritmo sea incapaz de progresar. [BO83]

Por su parte, la segunda opción permite que el algoritmo principal mantenga su operativa en base a un modelo de comunicación asíncrono, pero dispone de un detector de fallos que encapsula la componente síncrona del sistema y se encarga de informar al nivel superior cuando se sospecha que un proceso determinado puede haberse detenido [CT96]. Este tipo de mecanismos se estudiarán en mayor detalle en la sección 5.4.

Esta última opción, que se fundamenta sobre la relajación de las condiciones de sincronía del sistema hasta un punto que permita un funcionamiento principalmente asíncrono pero sobre el que resulte posible evitar el resultado de imposibilidad de acuerdo en sistemas asíncronos, da lugar a la formulación de un nuevo modelo de sincronía que satisfaga estas necesidades, denominado modelo de sincronía eventual. Este modelo, englobado en la categoría de modelos de sincronía parcial, considera que el sistema se comporta de forma asíncrona durante la mayor parte del tiempo, hasta un determinado momento en el que, durante un intervalo o la ejecución de un determinado protocolo, se establecen límites sobre los tiempos de procesamiento y envío de mensajes, adquiriendo un comportamiento síncrono [DLS88]. Expresado formalmente, en un sistema eventualmente síncrono existe un límite temporal τ, finito pero desconocido, a partir del cual existe un límite superior en la latencia comprendida entre el envío de un mensaje y su recepción por parte del destinatario correspondiente. En caso de que dicho límite no existiera, se consideraría que el sistema es completamente asíncrono, mientras que en caso de que dicho límite se conozca y $\tau = 0$, el sistema sería síncrono [BMR11]. El modelo de sincronía eventual es especialmente relevante ya que se corresponde con el modelo de sincronía que ofrece Internet, como una red de *entrega de mejor esfuerzo*. En este tipo de infraestructuras se espera que la red exhiba un comportamiento estable, ofreciendo un escenario sobre el que emplazar un sistema síncrono, pero que ocasionalmente pueda verse sujeta a perturbaciones que hagan que su comportamiento resulte impredecible, como ocurre en los sistemas asíncronos. [BA14]

Algoritmos como Paxos [Lam05] [Lam06] y *Viewstamped Replication* [OL88] permiten alcanzar consenso ofreciendo tolerancia a f fallos de parada contando con n réplicas, tal que $n \geqslant 2f + 1$. El algoritmo PBFT [CL+99] proporciona consenso en presencia de fallos bizantinos con hasta f réplicas potencialmente maliciosas de un total de $n \geqslant 3f + 1$. Los tres algoritmos mantienen sus condiciones de seguridad incluso bajo el modelo asíncrono, mientras que su progreso de acuerdo con las condiciones de viveza se lleva a cabo durante los periodos síncronos, por lo que estos algoritmos operan bajo un modelo de sincronía parcial.

Mientras que algunos trabajos han logrado reducir el número de réplicas necesarias para tolerar f nodos potencialmente bizantinos a $n \geqslant 2f + 1$ incluyendo componentes hardware de confianza en el sistema [CNV04] [VCB+13] o relajando las restricciones de consistencia [LM07], otros sistemas como XFT [LVC+16] y el trabajo presentado en [ADD+17] logran alcanzar este factor adoptando un modelo síncrono e incluyendo mecanismos criptográficos de firma digital para proporcionar autenticación e integridad sobre los mensajes. En el caso de XFT, su propuesta es la de un modelo de fallos cruzado, que afirma ofrecer tolerancia a fallos de parada bajo condiciones de asincronía (en realidad, bajo condiciones de sincronía eventual), mientras que a la par ofrece tolerancia a fallos bizantinos bajo condiciones de sincronía total.

Dado que la adopción de un modelo totalmente síncrono exige fijar límites superiores sobre los tiempos de procesamiento y envío de los mensajes, estas suposiciones y afirmaciones han de realizarse sobre un entorno totalmente controlado y fiable, siendo el caso ideal de ejemplo el de una red de área local con enlaces replicados. Por lo general, es difícil afirmar que el despliegue del sistema se vaya a realizar sobre un escenario de estas características, que permita asegurar que se va a operar de forma totalmente síncrona respetando los límites establecidos sobre el procesamiento y el envío de los mensajes en todo momento. Sin embargo, también es difícil pensar que, en una red de uso común, la presencia de un adversario bizantino vaya a suponer incurrir en una asincronía total. Incluso ante el modelo de fallos bizantino es altamente improbable que el adversario sea capaz de corromper y controlar un número de máquinas y la red al completo de forma coordinada, así como crear particiones sobre la red a voluntad, lo que ha llevado a algunos autores a afirmar que el poder disruptivo que se le otorga a un adversario en el modelo bizantino clásico es excesivo por su inadecuación a la realidad [KR10]. Por ejemplo, los fallos severos accidentales no suelen dar lugar a particiones en la red. Incluso ante fallos que introduzcan un comportamiento malicioso en una fracción del sistema, es difícil provocar la caída completa de la red en sistemas geo-replicados o redes de área global, como las abarcadas por Internet. Se asume que, además de tomar el control de un número de réplicas acotado superiormente, un atacante puede introducir un incremento en la latencia de transmisión de los mensajes, pero no puede retrasar su entrega indefinidamente (es decir, impedirla).

En base a estos motivos es sensato afirmar que el modelo de sincronía más apropiado para sistemas de búsqueda de consenso que operen en presencia de fallos bizantinos, sobre Internet o sobre redes a las que no se puedan atribuir propiedades más estrictas, es el modelo de sincronía eventual. En este modelo, la red se puede comportar de forma asíncrona durante ciertos intervalos. Sin embargo, dado que un atacante no puede inutilizar la red de forma indefinida, la condición de que las propiedades de seguridad del sistema se mantengan en todo momento y las propiedades de viveza se cumplan durante los intervalos síncronos suele ser suficiente para garantizar un correcto desempeño del sistema en este escenario.

2.3 Consistencia

Cuando se replica información entre diferentes nodos, un modelo de consistencia especifica qué divergencias se llegarán a admitir entre el estado de cada una de las réplicas de un mismo dato. Aunque los modelos de consistencia se definieron inicialmente para arquitecturas multiprocesador con memoria compartida,también son aplicables a entornos distribuidos con información replicada.

Tradicionalmente existen dos categorías de modelos de consistencia, que se complementan entre sí en función de la perspectiva que se adopte a la hora de visualizar el sistema: los modelos de consistencia centrados en datos, que operan desde la perspectiva del estado replicado en cada uno de los nodos, y los modelos de consistencia centrados en el cliente [TDP+94], que se limitan a contemplar la forma en la que se procesan las peticiones de cada cliente individual del sistema. La especificación de un modelo de consistencia establece el conjunto de condiciones que el sistema ha de cumplir. [Mos93]

Los sistemas blockchain normalmente adoptan un modelo de consistencia en función de su naturaleza. En los blockchains no permisionados, en la mayor parte de los casos se admite la conformación de bifurcaciones temporales, consecuencia de los retardos en la propagación y difusión de las escrituras sobre el estado compartido. Cuando varios nodos añaden un nuevo bloque a la cadena, lo propagan a los demás nodos de la red mediante un protocolo de difusión. Sin embargo, la red puede constar de una alta cantidad de nodos con una gran separación geográfica entre sí. En caso de que varios nodos generen un nuevo bloque y procedan a su difusión prácticamente al mismo tiempo, los diferentes retardos en la propagación de mensajes pueden propiciar que los nodos de la red añadan bloques diferentes a su copia de la cadena, en función del bloque que reciban primero. Dado que en los sistemas blockchain cada bloque contiene un *hash* criptográfico del bloque anterior, estableciendo la secuencia ordenada de elementos que conforman la cadena [MB02], esta situación daría lugar a la creación de bifurcaciones en la cadena de bloques, que continuarían creciendo de forma separada. Ante esta situación, plataformas como Bitcoin optan por hacer que, cuando un nodo recibe un bloque que pertenece a una subcadena diferente a la suya y la longitud de dicha cadena es superior a la suya propia, el nodo descarte su propia cadena y adopte la de mayor

longitud. Este comportamiento se incentiva haciendo que la remuneración por el minado y la adición de un bloque no se produzca cuando se logra generar dicho bloque, sino cuando se ratifica que dicho bloque (y, por lo tanto, todos los que lo preceden) ha sido aceptado por la mayoría de los nodos de la red. Este es el motivo de que se recomiende esperar un lapso de tiempo desde la realización de una petición de transacción a Bitcoin hasta que se pueda considerar que ésta se ha ratificado, tanto por la duración del proceso de minado en el que la transacción se verá incluida en un nuevo bloque como por el tiempo necesario para garantizar que un bloque que contenga la transacción haya sido ratificado por la red como parte de la cadena de bloques que actúa como estado compartido. Como se expuso en la sección 4.2.1, si la plataforma se ve sometida a un ataque de 51 % se podría subvertir la funcionalidad del sistema forzando a la mayor parte del poder computacional de la red a la ratificación de una cadena determinada que no fuera necesariamente la más larga. Como ya se ha comentado, plataformas de divisa criptográfica como Ethereum incorporan mecanismos para proporcionar mayor fortaleza ante ataques de 51 %, como el hecho de que su sistema de incentivos beneficie la inclusión de bloques huérfanos, pertenecientes a ramas descartadas, favoreciendo escisiones frente a un monopolio de la cadena de bloques. El modelo de consistencia con el que se corresponde este comportamiento se denomina consistencia eventual [TDP+94], que permite la formación de divergencias en el estado compartido bajo la condición de que, tarde o temprano, se produzca una convergencia a un estado común. El principal motivo de que los sistemas blockchain no permisionados suelan optar por este modelo de consistencia se debe a la buena escalabilidad que permite alcanzar [Vog09], ya que el número de participantes que conforman la red de consenso sobre la que debe replicarse el estado compartido es potencialmente alto. Dada la pobre escalabilidad que presentan los algoritmos de búsqueda de consenso que permiten adoptar un modelo de consistencia más estricto, como el modelo de consistencia linearizable, en este tipo de blockchains la solución pasa por alcanzar una situación de consenso respecto al estado compartido relajando las restricciones de consistencia y adoptando un modelo menos estricto, que sí permita una buena escalabilidad ante un alto número de participantes en el proceso de búsqueda de consenso.

Profundizando en los modelos de consistencia relajada más allá de la que suele ser su aplicación en los sistemas blockchain no permisionados y la buena escalabilidad que su correcta implantación puede facilitar, la adopción de modelos de consistencia derivados de la consistencia eventual en el proceso de búsqueda de consenso en un escenario bizantino puede proporcionar ciertas ventajas, como aumentar el número máximo de réplicas cuyo fallo puede tolerarse. En el escenario tradicional de búsqueda de consenso ante fallos bizantinos, el sistema falla si, en un sistema compuesto de $3f + 1$, más de f réplicas fallan simultáneamente. [LM07] presenta un modelo de consistencia más relajada, denominado *fork**, que permite aumentar el número de réplicas fallidas en presencia de las cuales se puede alcanzar consenso, operando sobre un escenario bizantino autenticado. La concepción de este modelo se emplaza sobre un modelo anterior, denominado *fork* [MS02], una de cuyas principales

virtudes es que facilita la gestión de la trazabilidad y la auditoría de las operaciones realizadas, permitiendo detectar violaciones de la consistencia producidas con anterioridad por parte de nodos potencialmente maliciosos.

Para que un sistema se ajuste al modelo de consistencia *fork*, ha de cumplir las siguientes propiedades:

- Para que un cliente correcto acepte un resultado, dicho resultado debe contener una lista de resultados previos L que únicamente albergue operaciones bien formadas legítimamente emitidas por otros clientes.
- Cada cliente correcto puede ver todas sus operaciones previas: si un cliente α emite una petición de operación op^i antes que otra op^j, entonces op^i figurará en la lista de resultados de op^j. Esta propiedad garantiza que un cliente tiene una visión del estado consistente con su propia secuencia de operaciones. Por ejemplo, en el caso de un sistema de archivos distribuido, esto significaría que los datos leídos por un cliente siempre contendrán los cambios incorporados por sus operaciones de escritura previas.
- Toda lista de resultados aceptada por cualquier cliente que contenga una operación op emitida por un cliente correcto será idéntica por lo menos hasta la aparición de dicha operación op.

Dado que toda lista de resultados mantiene el orden temporal de las peticiones no concurrentes emitidas por clientes correctos, cuando dos clientes ven las operaciones previas del otro el modelo de consistencia garantiza que están viendo dichas operaciones en el orden correcto.

Dos listas de resultados son consistentes si una es un prefijo (es decir, una subcadena desde el principio hasta un determinado punto) de la otra. Por el contrario, dos listas de resultados que no sean consistentes son producto de una bifurcación, tal como muestra la figura 5.1.

En caso de que ocurra una bifurcación, se dice que dos nodos se encuentran en diferentes conjuntos de bifurcación si poseen listas de resultados bifurcadas y, por lo tanto, no consistentes entre sí. Un conjunto de bifurcación se compone de un conjunto de servidores que devuelven la misma lista de resultados consistentes entre sí a todos los clientes, aunque, en el caso de los sistemas asíncronos, en el momento en el que los clientes reciben los resultados éstos puedan no reflejar el estado actual del sistema. Como se puede inferir de forma intuitiva, un nodo honesto no puede encontrarse en más de un conjunto de bifurcación. Por el contrario, tal como refleja la figura 5.2, un nodo malicioso puede encontrarse simultáneamente en múltiples conjuntos de bifurcación, pretendiendo albergar un estado diferente del sistema en función del nodo o cliente con el que interactúe. Un atacante que controle la red y una cantidad suficiente de réplicas puede provocar que las réplicas honestas restantes se adhieran a conjuntos de bifurcación diferentes, alcanzando un punto en el que la consistencia desde la perspectiva del cliente no se mantenga.

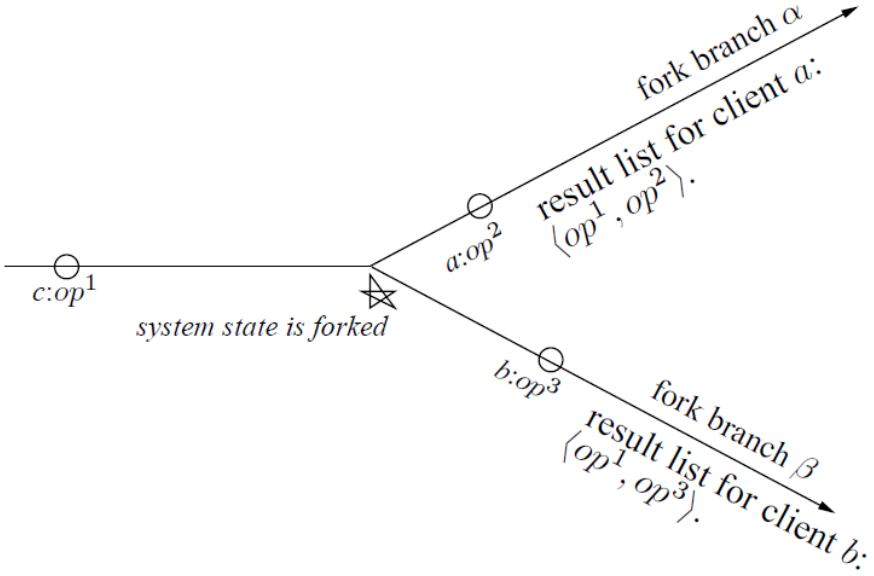

Figura 2.1: Ejemplo de consistencia según el modelo *fork* [LM07]

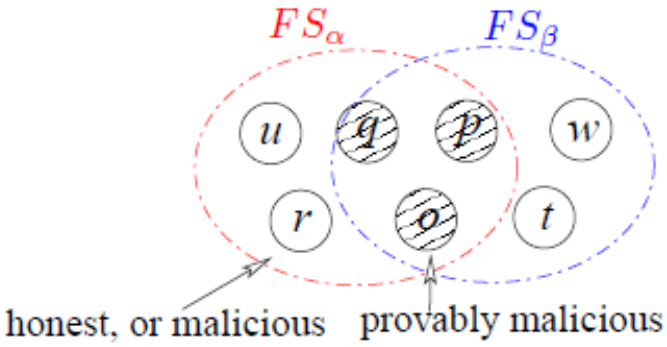

Figura 2.2: Conjuntos de bifurcación [LM07]

Desafortunadamente, para que un sistema pueda garantizar un comportamiento consistente con el modelo cuando el número de réplicas es menor del necesario para proporcionar consistencia lineal, los clientes deben enviar dos mensajes por cada operación que realicen, forzando a que el consenso se alcance en un protocolo de dos rondas, como el que muestra la figura 5.3.

Como consecuencia de esta problemática, que supone asumir la pérdida de rendimiento que implica emplear un protocolo de dos rondas, [LM07] define un modelo de consistencia más relajada que el modelo *fork*, denominado *fork**, que garantiza consistencia en una única ronda ante un número de réplicas maliciosas mayor que el necesario para proporcionar consistencia lineal. En el protocolo seguido para alcanzar este modelo cada petición debe especificar el orden exacto en el que la petición anterior del mismo cliente debía haberse

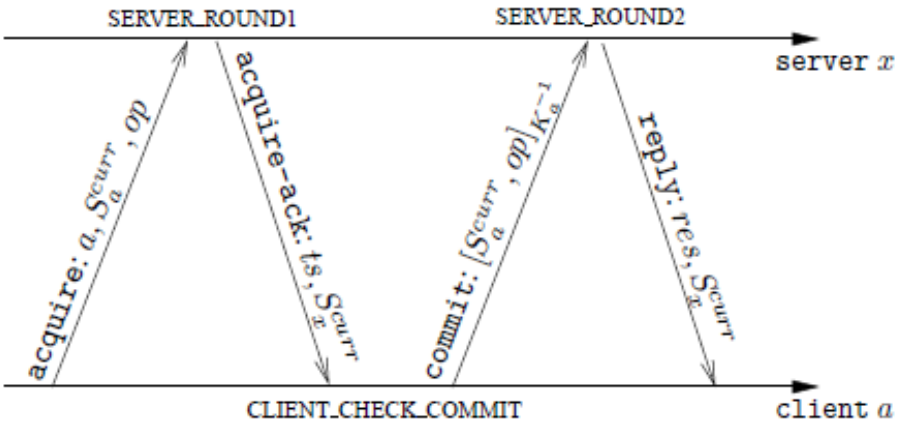

Figura 2.3: Protocolo de dos rondas para alcanzar consistencia según el modelo *fork* [LM07]

ejecutado. Así, en caso de que un nodo ejecute una operación sin respetar este orden, este hecho se detectará en el momento en el que se reciba la siguiente petición por parte del mismo cliente.

Para ello, el modelo de consistencia *fork** relaja las condiciones de consistencia exigidas por el modelo *fork*, reemplazando la tercera propiedad de su predecesor por la siguiente:

- Si dos listas de resultados aceptadas por clientes correctos contienen sendas operaciones op y op' del mismo clientes, entonces ambas listas de resultados contendrán dichas operaciones en el mismo orden. Si op precede a op', entonces ambas listas de resultados serán idénticas por lo menos hasta la posición de la operación op.

Si se comprueba periódicamente el conocimiento de las secuencias de operaciones que poseen los clientes, la propiedad introducida permite comprobar que en un determinado instante de tiempo t^n el sistema no ha violado la consistencia por lo menos hasta el instante t^{n-1}. Los modelos de consistencia estricta requieren intersección sobre el quórum, es decir, que dos quórums intersecciones en al menos una réplica correcta, mientras que el modelo de consistencia *fork** únicamente requiere inclusión en el quórum, es decir, que cada quórum contenga al menos una réplica correcta. Más información sobre los modelos de consistencia *fork* y *fork**, incluyendo las justificaciones de las propiedades sobre las que operan y el protocolo utilizado en cada caso, puede encontrarse en [MS02] y [LM07].

Con el objetivo de determinar el modelo de consistencia que más se adecúa a la operación de los sistemas blockchain permisionados, [DAAB+18] realiza un estudio de este tipo de sistemas en términos del teorema CAP, empleando a PBFT como exponente más representativo de los algoritmos tradicionales de búsqueda de consenso en escenarios bizantinos.

El teorema CAP establece que en un almacén de datos distribuido sólo pueden mantenerse dos de las tres siguientes propiedades: consistencia (C, *Consistency*), disponibilidad (A, *Availability*) y tolerancia a particiones (P, *Partition resilience*). Esto quiere decir que cualquier almacén de datos distribuido se puede clasificar en base a las dos propiedades que como máximo puede garantizar, ya sea CA, CP O AP. [GL02]

Un blockchain alcanza una situación de consistencia cuando no existen bifurcaciones en la cadena de bloques, cumpliendo las propiedades de orden total y acuerdo que caracterizan a los protocolos de difusión atómica [HT94], tal como señala [CV17]. Cuando no se puede alcanzar una situación de consistencia, debe determinarse si las bifurcaciones se resolverán tarde o temprano (consistencia eventual) o no llegarán a resolverse (no consistencia). Un blockchain cumple con el concepto de disponibilidad si las transacciones emitidas por los clientes se atienden y finalmente son confirmadas, es decir, añadidas a la cadena de bloques de forma permanente. Un blockchain es tolerante a particiones cuando, al surgir una partición en la red, las réplicas se dividen en grupos disjuntos que no pueden comunicarse entre sí.

En el caso de los sistemas blockchain permisionados desplegados para su operación sobre Internet, éstos se ven obligados a asumir las restricciones impuestas por un sistema eventualmente síncrono, tal como se vio en la sección dedicada al estudio de los modelos de sincronía, debiendo tolerar situaciones adversas como un número acotado de nodos bizantinos que traten de actuar contra la disponibilidad y la consistencia y periodos en los que la red se comporte de forma asíncrona. Dado que un blockchain debe tolerar la formación de particiones, no puede optar a la vez por mantener consistencia y disponibilidad, debiendo elegir una de las dos. La relación entre las elecciones condicionadas por el teorema CAP y los diferentes modelos de consistencia se refleja en la figura 5.4.

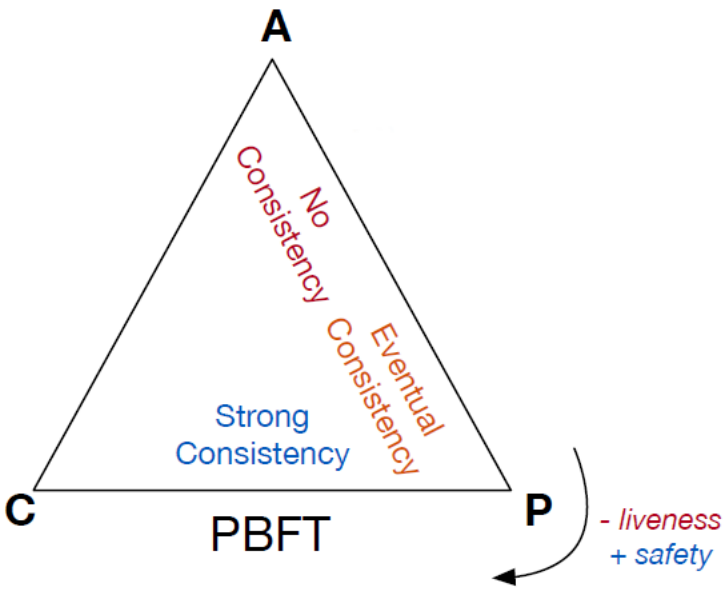

Figura 2.4: Clasificación de PBFT y los modelos de consistencia en base al teorema CAP [DAAB+18]

En escenarios en los que resulte adecuado asumir un modelo de consistencia relajada, su adopción puede conllevar las ventajas referidas; sin embargo, en las máquinas de estados replicadas utilizadas para conseguir consenso distribuido en los sistemas blockchain permisionados es necesario respetar un modelo de consistencia más estricto, que cumpla con las propiedades de linearizabilidad y viveza. De hecho, la principal motivación para utilizar DLTs como los blockchain permisionados frente a otros tipos de sistemas distribuidos se debe a la consistencia estricta y la integridad de los datos que los primeros permiten mantener, junto con su alta productividad en el procesamiento de operaciones por parte del sistema replicado. La linearizabilidad [HW90], también denominada consistencia ideal e inspirada por el modelo de consistencia atómica [Lam86], representa las propiedades de seguridad del modelo, exigiendo que ante los clientes el servicio ofrezca una impresión similar a un servicio centralizado, procesando una secuencia de peticiones idéntica y preservando el orden temporal de las operaciones no concurrentes. Las propiedades de viveza del modelo exigen que el sistema avance en su estado procesando las peticiones remitidas por los clientes. Desde el punto de vista de la consistencia centrada en el cliente, los clientes deben tener una visión del estado compartido que sea consistente con sus propias acciones, aunque lean y escriban en diferentes réplicas del servicio.

2.4 Detección de fallos

En sistemas distribuidos tolerantes a fallos como las máquinas de estados replicadas es conveniente que los datos compartidos por las diferentes réplicas no sólo incluyan la información relativa específicamente al estado actual del servicio y el historial de peticiones, sino que también se posea información referente al conjunto de nodos que actualmente se encuentran activos y exhibiendo un comportamiento correcto, y por lo tanto pueden participar en el proceso de búsqueda de consenso para la toma de decisiones colectivas. Sin embargo, no es suficiente con que cada proceso tenga una noción individual y propia acerca del conjunto de procesos activos, sino que conviene que todos los nodos activos coincidan en dicha información, alcanzando un acuerdo respecto a la conformación del grupo en cada momento. Los servicios que, mediante la ejecución de un algoritmo, permiten alcanzar esta situación de acuerdo se denominan servicios de pertenencia a grupo. La principal ventaja de este tipo de servicios es que permiten que todos los componentes del sistema replicado tomen acciones uniformes ante la caída o reincorporación de un proceso, a la par que simplifican los algoritmos necesarios para tratar la situaciones de fallo o reincorporación (exclusión e inclusión, respectivamente, puesto que se tiene la garantía de que todos los procesos correctos reciben la misma secuencia de eventos de este tipo.

Los primeros servicios de pertenencia a grupo fueron expuestos para el caso de los sistemas distribuidos síncronos [Cri91b]. Aunque posteriormente se publicaron otros trabajos referentes a servicios de pertenencia a grupo en sistemas asíncronos [JFR93], trabajos posteriores demostraron la imposibilidad de implantar estos servicios en sistemas puramente asíncronos [CHTCB96], como consecuencia de la imposibilidad de alcanzar una situación de acuerdo distribuido tolerante a fallos en un sistema distribuido expuesta por Fischer, Lynch y Patterson [FLP85].

Pese a esta dificultad, los servicios de pertenencia a grupo suelen encontrarse en cualquier sistema moderno de comunicación a grupos [CKV01], como los utilizados para realizar difusión de orden total [HT94], empleada en la búsqueda de consenso. Estos servicios de pertenencia se apoyan sobre los resultados de detección de fallos presentados en [CT96], que expone que estos servicios no podrán ser perfectos en un entorno asíncrono, pero sin embargo sí que podrán ser utilizables y de provecho si los algoritmos que los emplean conocen y tienen en cuenta sus limitaciones.

Los detectores de fallos fueron definidos por Chandra y Toueg en [CT96] como la abstracción de un mecanismo que reside en cada uno de los procesos que componen el sistema distribuido, recopilando la lista de los procesos los cuales se sospecha que puedan haber fallado. La publicación citada los presenta en un escenario de operación en el que se asume un modelo de fallos de caída sin recuperación, con canales de comunicación fiables y en el contexto de un sistema asíncrono.

La calidad de un detector de fallos puede medirse en base a su cumplimiento de dos propiedades, sobre cada una de las cuales se distinguen varios grados:

- **Completitud:** Los procesos que han fallado deben ser considerados sospechosos.
  - **Completitud fuerte:** Llega un momento en el que todos los procesos correctos sospechan permanentemente de todos los procesos fallidos.
  - **Completitud débil:** Llega un momento en el que, para todo proceso fallido, existe un proceso correcto que sospecha permanentemente de él.
- **Precisión:** Los procesos correctos no deben ser considerados sospechosos.
  - **Precisión fuerte:** No se sospecha de ningún proceso antes de que éste falle.
  - **Precisión débil:** Como mínimo, hay un proceso correcto del que jamás se sospecha.
  - **Precisión fuerte eventual:** Llega un momento en el que no se sospecha de ningún proceso correcto.
  - **Precisión débil eventual:** Llega un momento en el que hay al menos un proceso correcto del que no se sospecha.

La combinación de estos ocho grados de cumplimiento permite clasificar los detectores de fallos en los ocho tipos presentados en el cuadro 5.1.

Completitud	**Precisión**			
	Fuerte	Débil	Eventualmente fuerte	Eventualmente débil
Fuerte	Perfecto P	Fuerte S	Eventualmente perfecto $\Diamond P$	Eventualmente fuerte $\Diamond S$
Débil	Q	Débil W	$\Diamond Q$	Eventualmente débil $\Diamond W$

Cuadro 2.1: Ocho clases de detectores de fallos definidos en términos de su completitud y precisión [CT96]

En base a esta clasificación y además de ella, [CT96] propone un algoritmo de reducción con el que se puede demostrar que a partir de un detector de fallos con completitud débil se puede implantar otro con completitud fuerte. Su planteamiento se basa en que los procesos que sospechen de la caída de algún otro difundirán sus sospechas, que serán aceptadas por todos los demás, lo que permite alcanzar un nivel de completitud fuerte a la vez que se mantiene la precisión del algoritmo original.

La aplicación de los detectores de fallos que Chandra y Toueg realizan es la resolución de dos problemas tradicionalmente irresolubles en sistemas distribuidos asíncronos donde los procesos pueden fallar: el acuerdo distribuido y la difusión atómica.

Realizando un breve retorno al problema del consenso distribuido, éste tiene lugar un contexto en el que un proceso propone un valor, que se corresponde con una acción ejecutable por parte de los procesos, y todos los procesos correctos han de ponerse de acuerdo respecto a la acción a ejecutar, optando todos por el mismo valor. En estos términos, una solución correcta debe cumplir las siguientes cuatro propiedades:

- **Terminación:** Todo proceso correcto eventualmente decide algún valor.
- **Integridad uniforme:** Todo proceso correcto decide como máximo una vez.
- **Acuerdo:** Ningún par de procesos correctos decide de manera diferente.
- **Validez uniforme:** Si un proceso decide un valor v, entonces dicho valor v fue originalmente propuesto por algún proceso.

Puesto que el algoritmo de reducción posibilita que cualquier tipo de detector de fallos con completitud débil pueda llegar a ofrecer completitud fuerte, el problema de la búsqueda de consenso puede resolverse con cualquiera de los detectores de fallos presentados anteriormente [CHT96]. No obstante, cada tipo soporta un número máximo de fallos diferente y necesitará un número de rondas de difusión fiable diferente para alcanzar la situación de consenso. Así, siendo n el número de procesos que componen el sistema y recordando que se opera bajo el modelo de fallos de parada, con las clases P y S (*perfecto* y *fuerte*, respectivamente) se toleran $n-1$ fallos, mientras que con las clases $\Diamond P$ y $\Diamond S$ (*eventualmente perfecto* y *eventualmente fuerte*, respectivamente) se necesita una mayoría de procesos correctos.

Es sencillo construir un detector de fallos perfecto en un entorno síncrono. Sin embargo, en un escenario totalmente asíncrono el resultado de imposibilidad de [FLP85] impide alcanzar ningún grado de completitud. Tal como se expuso en la sección dedicada al estudio de los modelos de sincronía, una de las soluciones más utilizadas para evitar esta imposibilidad es relajar las restricciones del modelo de sincronía, adhiriéndose a un modelo menos estricto, como el modelo de sincronía eventual. Bajo este modelo, la existencia de un límite superior desconocido sobre la latencia en el envío de los mensajes permite que la implementación de una solución basada en *timeouts* dé como resultado un detector de fallos eventualmente perfecto. El encapsulamiento de esta componente síncrona en el detector de fallos, que constituye una abstracción de cara a los niveles superiores, fundamentalmente el servicio de pertenencia a grupo, permite que el algoritmo principal del sistema siga operando desde una perspectiva aparentemente asíncrona. Un ejemplo de algoritmo que permite alcanzar consenso gracias a esta técnica es Paxos [Lam05] [Lam06], así como el propio algoritmo de búsqueda de consenso tolerante a fallos presentado en [CT96].

Los detectores de fallos descritos hasta ahora operan bajo el modelo de fallos de parada, y cumplen su propósito en los sistemas que se ciñen a las condiciones que impone este modelo. Por el contrario, para alcanzar una situación de consenso en un escenario bajo el modelo de fallos bizantino es necesario complementar los detectores de fallos con otros me-

canismos más sofisticados. El ejemplo más demostrativo los constituye PBFT [CL+99], que proporciona tolerancia a fallos bizantinos complementando los detectores de fallos (basados en *timeouts* sobre el modelo de sincronía eventual) con técnicas criptográficas para dotar de autenticación e integridad a los mensajes, la posible ejecución de un mayor número de rondas para alcanzar consenso y la necesidad de que el quórum para confirmar la ejecución de acciones sobre la máquina de estados o realizar un cambio de vista sea consistente con el factor máximo de réplicas fallidas que el modelo permite tolerar. En este caso, considerando un sistema bizantino con sincronía eventual y consistencia lineal, el número óptimo de réplicas con el que se puede alcanzar consenso es $n = 3f + 1$; dado que es necesario un quórum de $f + 1$ réplicas para confirmar una orden o realizar un cambio de vista, dicho quórum debe ser de al menos $\frac{n-1}{3} + 1$ réplicas.

Además, PBFT emplea recuperación proactiva, una técnica que consiste en restaurar periódicamente las réplicas, incluso en caso de que éstas parezcan ser honestas, para eliminar potenciales intrusiones no detectadas [CL02]. La principal ventaja de esta técnica es que el adversario sólo puede estar en control de una réplica durante una ventana de vulnerabilidad, hasta la siguiente restauración

Dado que la funcionalidad principal de los detectores de fallos es proporcionar información sobre los nodos de los que se sospecha, de tal forma que puedan excluirse del grupo de pertenencia y propiciar acciones como la deposición de un líder y un cambio de vista, en un escenario bizantino los detectores de fallos pueden perfeccionarse para apercibirse de más situaciones que puedan suscitar la posibilidad de que un nodo esté fallando, y cuya naturaleza escapa del ámbito del modelo de fallos de parada.

El paradigma de detector de fallos presentado hasta el momento se amplía en [MR97b] al abarcar entornos en los que los procesos puedan sufrir intrusiones por parte de un atacante (fallos independientes de carácter potencialmente malicioso), considerando el problema de la búsqueda de consenso ante estos detectores de fallos bizantinos. Sin embargo, la mayor dificultad a la hora de diseñar detectores de fallos para escenarios bizantinos consiste en identificar qué tipos posibles de comportamiento arbitrario deberían detectarse, definiendo su completitud y su precisión. Sin embargo, realizar esta clasificación es más complejo que en el modelo de fallos de parada. Por ejemplo, para que un detector de fallos fuera completo en un escenario bizantino debería poder señalar a los nodos corruptos, una condición difícil de mantener puesto que un nodo corrupto puede seguir manteniendo el comportamiento de un nodo correcto, no exhibiendo ningún malfuncionamiento que pueda conducir a su detección. Para ello, los mecanismos de detección de fallos en escenarios bizantinos se fundamentan en su capacidad para identificar comportamiento que trate de impedir el progreso del sistema. En este contexto se establece la completitud en base a la detección (eventual) de comportamientos que puedan tratar de impedir el progreso del algoritmo, la no recepción de mensajes por parte de nodos que, según el modelo del algoritmo, deberían enviarlos (omisión de envíos

[HT94]), definiendo ambas propiedades de la siguiente forma:

- **Completitud fuerte:** Todo proceso del que no se reciban mensajes será eventualmente objeto de sospecha por parte de todo proceso correcto.
- **Precisión débil eventual:** Existe un límite de tiempo a partir del cual un proceso correcto deja de ser objeto de sospecha por parte de otros procesos correctos.

Así, [MR97b] define una clase de detectores de fallos $S(bz)$ para escenarios bizantinos con completitud fuerte y precisión eventualmente débil, demostrando que este tipo de detectores de fallos pueden resolver el problema del consenso en un sistema asíncrono sobre un escenario bizantino en el que menos de un tercio de las réplicas del sistema hayan sido comprometidas. Dicho algoritmo se basa en la disponibilidad de una primitiva de difusión fiable, pero cuyas restricciones de orden causal son más laxas que las de las funciones de difusión atómica utilizadas tradicionalmente para alcanzar consenso [BSS91]. En base a esta definición y a sus características asociadas, los autores construyen un algoritmo de búsqueda de consenso en entornos bizantinos que adopta un modelo híbrido, combinando el tipo de detectores de fallos bizantinos descritos con técnicas de aleatorización para circunvalar el resultado de imposibilidad de [FLP85] en escenarios asíncronos y alcanzar una situación de consenso con la mayor certeza posible, ya sea por la vía determinista o por la probabilística.

Las técnicas de tolerancia a fallos bizantinos expuestas hasta el momento se basan en *enmascarar* un número acotado superiormente de fallos, lo que supone el inconveniente de que obliga a desplegar un número de réplicas suficiente para proporcionar el umbral de tolerancia a fallos deseado. En particular, y como ya se ha referido anteriormente, para tolerar f fallos bizantinos simultáneos se necesitan al menos $3f + 1$ réplicas [BT85]. La pobre escalabilidad que por norma general presentan los algoritmos utilizados para alcanzar consenso tolerante a fallos con consistencia lineal provoca que la eficiencia del sistema decrezca conforme se añaden nuevas réplicas a la red, debido a la necesidad de intercambiar un mayor número de mensajes para alcanzar una situación de acuerdo entre los nodos de la red.

Una forma alternativa de proceder es reaccionar de forma activa ante la detección de fallos, en lugar de limitarse a enmascarar el comportamiento bizantino, tal como se propone en [HKD06]. El procedimiento base es similar al tradicional; cada nodo cuenta con un detector de fallos que supervisa la comunicación con los otros nodos, tratando de detectar síntomas de comportamiento defectuoso, en cuyo caso notifica este hecho al nivel superior, el cual decidirá realizar la acción oportuna, como proponer aislar del grupo de pertenencia al nodo supuestamente afectado. El hecho de operar en un entorno autenticado, es decir, empleando técnicas criptográficas que doten a los mensajes de autenticación e integridad que permitan demostrar tanto ante uno mismo como ante terceros qué identidad fue la remitente de un mensaje, posibilita exponer ante los otros nodos los indicios de comportamiento malicioso.

A partir de estas pruebas, el grupo podría decidir, por ejemplo, aislar de la operación del sistema al nodo afectado.

En este contexto, un detector de fallos perfecto detectaría de forma inmediata un fallo bizantino de cualquier naturaleza. En un escenario práctico, la capacidad y eficacia de los detectores de fallos se encuentran limitadas. Partiendo de la base de que en el escenario del artículo [HKD06] la capacidad de un detector de fallos en un nodo correcto se restringe a poder monitorizar todos los mensajes enviados y recibidos por dicho nodo, existen tipos de fallos bizantinos que no son observables y, por lo tanto, no pueden ser detectados. Bajo el modelo que establece esta premisa, se pueden distinguir dos tipos de fallos. De acuerdo a una definición formal, un nodo i *falla detectablemente* si el comportamiento que exhibe ante los nodos correctos diverge del que el nodo i mostraría en caso de ser correcto, y un nodo i es *detectablemente ignorante* si ignora un mensaje dirigido a él por parte de un nodo correcto. Por ejemplo, si un nodo correcto solicita un servicio que el nodo i debería prestar de acuerdo a su especificación funcional, i falla detectablemente si rechaza la solicitud, y es clasificado como detectablemente ignorante si actúa como si ni siquiera hubiera recibido dicha solicitud.

En este contexto, si un módulo detector de fallos B_i en un nodo correcto i presencia indicios de comportamiento malicioso en otro nodo j, envía una indicación de fallo a su proceso de aplicación local. Estas indicaciones se dividen en tres tipos *confianza*$_j$, *sospecha*$_j$ y *exposición*$_j$. Si B_i emite *sospecha*$_j$, existen indicios de que j está ignorando determinadas entradas, por ejemplo, al negarse a aceptar una solicitud de servicio enviada por un nodo correcto. Si B_i emite *exposición*$_j$, existen pruebas de que j ha sido comprometido, por ejemplo, al haberse desviado de la especificación funcional de su máquina de estados A_j. En caso de que B_i emita *confianza*$_j$, se asume que ninguna de las situaciones anteriores ha tenido lugar.

Estableciendo una relación entre estas definiciones y las propiedades de completitud y precisión fuerte eventual enunciadas en [KMMS03] como una adaptación de las propiedades establecidas en [CT96] al modelo de fallos bizantino, los sistemas detectores de fallos presentados en [HKD06] exhiben las siguientes propiedades:

- **Completitud fuerte eventual:** En algún momento, todo nodo detectablemente ignorante es objeto de sospecha permanente por parte de todo nodo correcto, y si un nodo i falla detectablemente con respecto a un mensaje m, entonces en algún momento (eventualmente) algún cómplice malicioso de i (involucrado en m) será objeto de sospecha permanente por parte de todo nodo correcto.
- **Precisión fuerte eventual:** Ningún nodo correcto es objeto de sospecha permanente por parte de otro nodo correcto, y ningún nodo correcto es catalogado como expuesto (comprometido) por parte de otro nodo correcto.

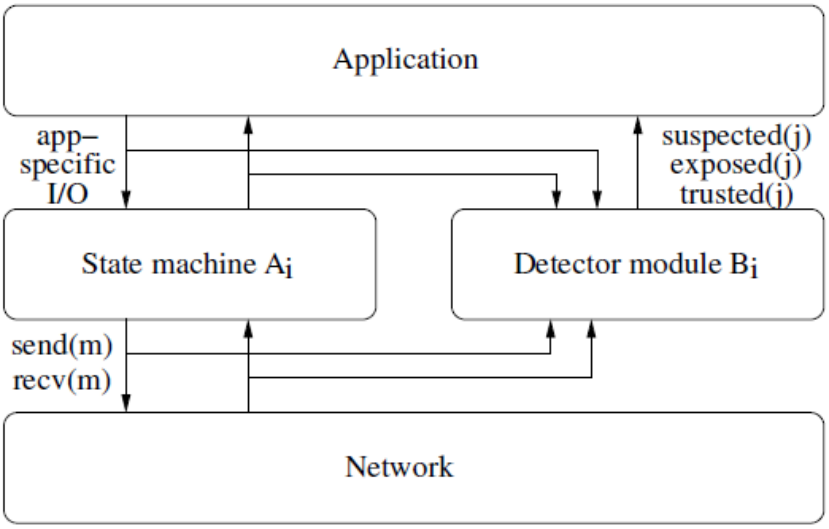

Figura 2.5: Flujo de información entre aplicación, protocolo y detector de fallos en el nodo i [HKD06]

Debe destacarse que, cumpliendo las propiedades descritas, el detector no garantiza que un nodo correcto sea objeto de confianza permanente por parte de otro nodo correcto, ya que podría ser objeto de sospecha momentánea, por ejemplo, debido a una alta latencia transitoria en el envío de los mensajes. Además, es posible que un nodo fallido detectablemente nunca sea detectado, pero en tal caso algún nodo fallido será calificado como comprometido u objeto de sospecha permanente. Así, si el número de nodos comprometidos en el sistema es finito, el tiempo durante el que los nodos correctos puedan verse afectados por su comportamiento malicioso también lo será.

Asumiendo independencia entre los fallos, el planteamiento de detección de fallos presenta varias ventajas destacables frente al enfoque de enmascaramiento de fallos tradicionalmente empleado en los sistemas tolerantes a fallos bizantinos, como PBFT [CL02] o el planteado en [MR97b]. La principal ventaja es que se requieren menos réplicas para alcanzar una situación de acuerdo tolerante a fallos. En caso de que el conjunto de réplicas determine que un nodo en particular ha sido comprometido, se puede excluir a dicho nodo del grupo de pertenencia, de forma que éste deje de tomar parte en el proceso de intercambio de mensajes utilizado para alcanzar consenso. Frente al enfoque de enmascaramiento de fallos, en el que se utilizan $3f + 1$ réplicas para tolerar hasta f fallos bizantinos, un supuesto detector de fallos perfecto (es decir, con completitud y precisión fuertes) podría reducir el número de réplicas necesarias para alcanzar consenso en un escenario bizantino a $2f + 1$, ya que sólo sería necesaria una mayoría de nodos honestos para evitar una subversión del sistema. En función de la naturaleza del sistema y su entorno de operación, esta ventaja podría em-

plearse para disminuir el número de réplicas utilizadas manteniendo el umbral de tolerancia a fallos, o tolerar un mayor número de fallos con un número de réplicas constante, teniendo presente en todo caso la relación inversa entre número de réplicas y productividad que los algoritmos de búsqueda de consenso con consistencia estricta suelen presentar debido a su mala escalabilidad.

Al realizar un recorrido sobre las bondades de la detección activa de fallos enunciando sus ventajas, [HKD06] realiza algunas afirmaciones sin mencionar que éstas habrían de sustentarse sobre ciertos planteamientos especiales. Por ejemplo, el artículo mantiene que el número de réplicas necesarias para proporcionar tolerancia a fallos bizantinos podría reducirse a $f + 1$, omitiendo la aclaración de que dicho nivel de tolerancia a fallos habría de alcanzarse complementando el servicio de pertenencia a grupo con un mecanismo como un componente incorruptible de confianza, que impidiera que una mayoría de nodos bizantinos pudiera aislar a una minoría de nodos honestos. Igualmente, el artículo afirma que la carga de trabajo de cada nodo podría disminuirse si cada petición de los clientes sólo fuera procesada por un nodo, mediante la propagación de las modificaciones del estado compartido a las demás réplicas, siguiendo un esquema similar al modelo de replicación multi-máster [ABKW98]. Sin embargo, se obvia que esto tendría como consecuencia la necesidad de adoptar modelos de consistencia menos estrictos que la consistencia lineal, a no ser que las operaciones a ejecutar sobre la máquina de estados fueran conmutativas o igualmente se hiciera uso de un protocolo de búsqueda de consenso para establecer de forma colectiva orden total en la ejecución de las operaciones vinculadas a las peticiones (y aplicación de las actualizaciones de estado generadas) sobre el estado compartido.

El modelo de fallos bizantino es el menos restrictivo, y por lo tanto comprende el abanico más amplio de los fallos que puede sufrir un nodo, ya que abarca desde defectos en el software hasta cualquier tipo de comportamiento malicioso que pueda acontecer como consecuencia de una intrusión. Esto dificulta su detección, debido a sus múltiples orígenes y naturalezas posibles. Los autores de [MNC12] estudian el problema de la búsqueda de consenso tolerante a fallos bizantinos en redes *ad-hoc* inalámbricas. Debido a la poca fiabilidad del medio de transmisión de las comunicaciones en tal escenario, la resolución de este problema puede beneficiarse en gran medida de técnicas que faciliten la detección de fallos. Si bien los autores de [HKD06] no concretan ante qué sucesos un nodo podría llegar a sospechar de otro nodo (indicando únicamente el caso de la expiración de un temporizados, lo que se ajusta al modelo de fallos de parada pero se queda corto frente a la gran variedad causas posibles de un fallo en el modelo bizantino) o esgrimir pruebas de que éste se estuviera comportando de forma maliciosa, los autores de [MNC12] añaden una fase de validación al tratamiento de los mensajes recibidos con anterioridad a su procesamiento efectivo. Esta fase de validación se compone de dos etapas, validación de autenticación y validación semántica. La etapa de validación de autenticación verifica que el nodo remitente de un mensaje es

efectivamente quien dice ser. La etapa de validación semántica comprueba que el contenido del mensaje recibido (tipo y datos) es congruente de acuerdo con el algoritmo que se ejecuta en el nivel superior al protocolo de búsqueda de consenso y al estado actual de éste, compartido entre las réplicas. Ambas etapas deben superarse con éxito para que el mensaje sea considerado como válido.

La adición de una fase de validación semántica puede ser de gran utilidad para facilitar la detección de fallos bizantinos, especialmente en el caso del comportamiento malicioso derivado de una intrusión. Uno de los aspectos más llamativos de la validación semántica es que no se fundamente únicamente en la información de la que el algoritmo de consenso dispone por sí mismo, siendo un protocolo de propósito general. Por el contrario, la inclusión de un módulo validador cuya implementación depende específicamente del algoritmo de nivel superior que se ejecuta sobre la máquina de estados replicada permite integrar los protocolos de aplicación y consenso, compartiendo así información entre ambos niveles sin difuminar las fronteras bien definidas existentes entre ellos, tal como lo refleja el concepto de *validación externa* presentado por Cachin *et al.* [CKPS01]. En el caso de la validación de autenticación, ésta es intrínseca a un escenario en el que se utilicen técnicas criptográficas que doten de autenticación e integridad a los mensajes. En ambos casos, en [MNC12] estas validaciones se incluyen en el flujo de ejecución del protocolo de consenso, sin integrarse en el módulo detector de fallos como en el modelo de funcionamiento que sugiere [HKD06].

2.5 Criptografía

Uno de los elementos clave en los algoritmos de búsqueda de consenso sobre máquinas de estado replicadas, si éstas pretenden exhibir tolerancia a fallos de naturaleza bizantina, es el uso de técnicas criptográficas sobre los mensajes transmitidos y, en función del algoritmo, sobre el propio estado mantenido por cada una de las réplicas. Su aplicación generalmente no es la de ofrecer confidencialidad mediante encriptación, sino el uso de primitivas que permiten obtener autenticación, integridad y no repudio, dando lugar a lo que se conoce como un escenario autenticado.

Los resultados presentados en [DLS88] establecen una diferenciación explícita entre un escenario bizantino autenticado, en el que se aplican técnicas criptográficas sobre los mensajes que permiten verificar la identidad de su remitente, y un escenario bizantino sin autenticación, donde no existen garantías de que el remitente de un mensaje sea realmente quien dice ser, o de que el mensaje no haya sido alterado en su tránsito. En un escenario bizantino no autenticado, un nodo bizantino podría tratar de subvertir el protocolo suplantando la identidad de otros nodos y enviando mensajes no válidos o incorrectos en su nombre. Dado que gran parte de los escenarios de búsqueda de consenso emplean la figura de un líder para centralizar parte de la toma de decisiones, un nodo bizantino incluso podría tratar de alterar el estado de los demás nodos asumiendo la identidad del líder y difundiendo comandos en

su nombre. El uso de técnicas que posibilitan la verificación de la identidad del autor de un mensaje limita el daño que un nodo bizantino puede causar.

Como se ha comentado, en este tipo de sistemas se suele proporcionar autenticación pero no confidencialidad, pues esto último requeriría encriptar los mensajes. Aunque es posible combinar autenticación y confidencialidad en un único algoritmo encriptando un mensaje y ñadiéndole una etiqueta (*tag*) de autenticación, normalmente estas dos funciones se proporcionan como primitivas independientes. Tal como se refleja en [DP89], esto resulta conveniente en aplicaciones en los que la confidencialidad no sea una preocupación fundamental y convenga evitar la sobrecarga computacional producida por ejecutar las funciones de cifrado y descifrado ante cada envío y recepción de un mensaje. Algunos escenarios ilustrativos de esta situación son los sistemas en los que cada mensaje se difunda a múltiples destinatarios, lo que provoca que cada uno de los destinatarios haya de asumir el coste computacional de descifrar cada mensaje recibido, o en aplicaciones en las que el receptor se vea sometido a una gran carga de trabajo y el hecho de tener que descifrar cada mensaje recibido pueda degradar su rendimiento.

Para proporcionar autenticación e integridad a los mensajes transmitidos entre los nodos de un sistema replicado se utilizan fundamentalmente dos mecanismos: MACs y funciones de firma digital.

Un código de autenticación de mensaje (MAC, *Message Authentication Code*) es un pequeño bloque de datos que se añade al mensaje. Esta técnica asume que ambos interlocutres poseen un secreto o clave privada compartida. Al enviar un mensaje, el remitente calcula el código de autenticación del mensaje mediante una función *hash* que se ejecuta a partir del propio mensaje y del secreto compartido, y anexa dicho bloque de datos al mensaje. Al recibirlo, el destinatario aplica la misma función *hash* utilizando el secreto compartido con el remitente y el cuerpo del mensaje recibido. Ambos códigos de autenticación, el calculado por el destinatario y el adjunto al mensaje, se comparan; la figura 5.6 muestra este proceso. En caso de que coincidan, se pueden realizar las siguientes afirmaciones: [SBBB12]

- El receptor tiene la certeza de que el mensaje proviene efectivamente del legítimo emisor. Puesto que se supone que el secreto compartido no ha sido comprometido, sólo emisor y receptor pueden haber generado el código de autenticación del mensaje.

- El receptor tiene la certeza de que el mensaje no ha sido alterado. En caso de que un atacante hubiera alterado el mensaje pero no el código adjunto al mismo, el código calculado por el receptor habría diferido del código recibido. Puesto que se supone que el secreto compartido no ha sido comprometido, con lo que únicamente lo conocen el emisor legítimo y el receptor, un atacante no puede alterar el código para reflejar las modificaciones que pueda haber realizado sobre el mensaje.

- En caso de que el mensaje incluya un número de secuencia (como en X.25, HDLC o TCP), el receptor puede detectar mensajes enviados fuera de orden, puesto que un atacante no podría alterar el número de secuencia del mensaje.

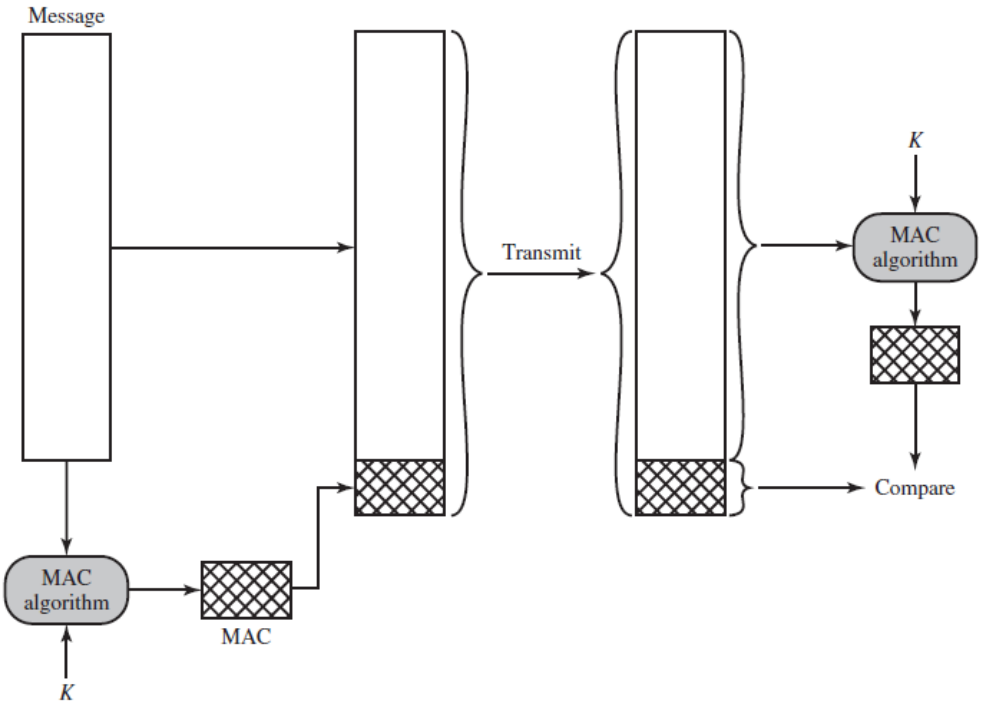

Figura 2.6: Autenticación distribuida mediante MAC [SBBB12]

Las técnicas de firma digital proporcionan autenticación basada en los mecanismos de encriptación de clave pública. En este caso, el emisor del mensaje utiliza una función para generar un valor de *hash* del mensaje y seguidamente cifra dicho valor de *hash* utilizando su clave privada, creando así una firma digital que se anexa al mensaje. Cuando el destinatario del mensaje lo recibe junto con la firma, calcula un valor de *hash* del mensaje, descifra la misma utilizando la clave pública del emisor y compara el valor de *hash* calculado con el valor de *hash* descifrado. En caso de que ambos valores de *hash* coincidan, el receptor tiene la garantía de que el mensaje efectivamente fue firmado por el emisor legítimo, tal como muestra la figura 5.7. El hecho de que sólo el emisor se halle en posesión de su propia clave privada permite garantizar que el mensaje no ha sido alterado, puesto que forzosamente el valor de *hash* del mensaje debe haber sido generado utilizando dicha clave privada. [SBBB12].

Tanto en el caso de los códigos de autenticación de mensaje, cuyas premisas de seguridad se basan en un secreto compartido, como en la firma digital, basada en pares de clave asimétrica, es necesario contar con algún mecanismo que, de forma previa a la operación del algoritmo, asegure que cada nodo cuente con los elementos necesarios: un secreto compartido con cada uno de los demás nodos en un caso, y la clave pública de los posibles interlocutores en el otro. Normalmente, el establecimiento de las identidades de los participantes se realiza

mediante la figura de un tercero de confianza, como una autoridad certificadora reconocida o un centro de distribución de claves (KDC, *Key Distribution Center*) [Mer80]. Otra opción que podría plantearse sería la generación de secretos compartidos entre cada par de participantes en el algoritmo distribuido [Mer78], mediante un algoritmo como Diffie-Hellman [DH76]. No obstante, en el caso de Diffie-Hellman, a lo poco práctico de tener que ejecutar el algoritmo para cada par de nodos se une el hecho de que en ausencia de un mecanismo de autenticación previa como certificados digitales validados por una autoridad certificadora el algoritmo de Diffie-Hellman es vulnerable a ataques de *man-in-the-middle* durante el intercambio de claves y a otros tipos de ataques [ABD+15], lo que lo convierte en una opción insegura.

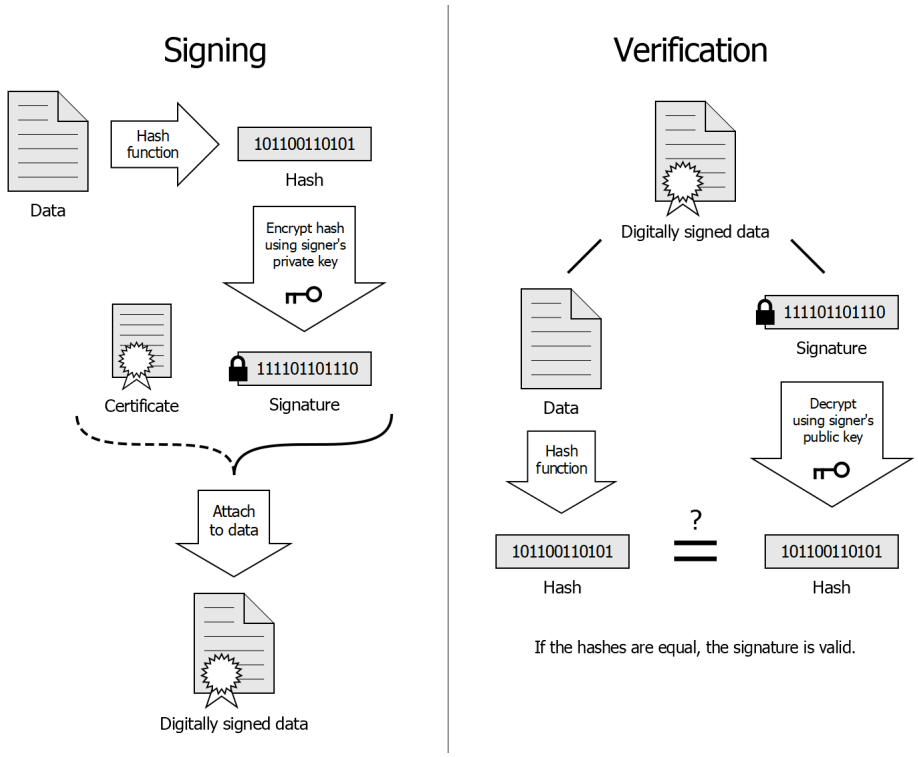

Figura 2.7: Autenticación e integridad mediante firma digital [SBBB12]

Ambas técnicas, MACs y firmas digitales se han aplicado en algoritmos de búsqueda de consenso tolerantes a fallos bizantinos como mecanismos capaces de dotar de autenticación e integridad a los mensajes, tal como se ha descrito. Mientras que en el caso de las firmas digitales cada mensaje se encuentra firmado con la clave privada de su emisor, en el caso de los códigos de autenticación de mensajes cada mensaje contiene un vector de MACs, lo que permite difundir un mismo mensaje a un grupo de destinatarios, con cada uno de los cuales posee un secreto compartido.

En el caso de los vectores de MACs la principal desventaja estriba, como ya se ha comentado, en la necesidad de establecer un secreto compartido entre cada par de procesos que toma parte en el algoritmo, mientras que la delegación de la distribución de las claves públicas de cada proceso a sus interlocutores en un KDC resulta más cómoda, segura y flexible. Además, otra característica que puede constituir una desventaja en determinados escenarios es el hecho de que las MACs permiten establecer la autenticidad e integridad de un mensaje a los poseedores del secreto compartido, pero no permiten certificar dichas propiedades ante terceros, como sí lo permiten las firmas digitales en conjunción con los certificados de clave pública. Por el contrario, la principal desventaja del uso de técnicas de criptografía asimétrica radica en el mayor coste computacional requerido por los procesos de cifrado y descifrado [Tsu92]. Al no cifrar el mensaje completo (salvo que se pretenda ofrecer confidencialidad), sino sólo el valor de *hash*, se reduce la carga computacional introducida, pero aun así el coste sigue siendo notablemente superior al requerido por las funciones de *hash* utilizadas en el mecanismo basado en códigos de autenticación de mensaje.

Los autores de PBFT [CL+99] consideran que el uso de vectores de MACs en los mensajes del protocolo (salvo en los cambios de vista, en los que sí se emplean firmas digitales para poder certificar la autoridad del nuevo líder) constituye una de las causas de su alta eficiencia y una de sus principales ventajas frente a otros algoritmos de búsqueda de consenso previos, como [MR97a] y [Rei94]. Sin embargo, es importante reseñar que dicha contribución es significativa en el contexto de la potencia computacional y la eficiencia algorítmica de los mecanismos de firma disponibles en 1999, fecha de publicación del artículo; pero, tal como señala [ADD+17], la existencia de plataformas de cómputo más potentes y a un coste más reducido en la actualidad [M+75] [Pre00], dos décadas más tarde, junto con la existencia de esquemas de firma más eficientes han hecho que el uso de firmas digitales deje de constituir un cuello de botella en el rendimiento ofrecido por el sistema que implementa el protocolo. Por este motivo, la mayoría de los algoritmos de búsqueda de consenso tolerantes a fallos bizantinos actuales, como [20117] y [CZ14] utilizan firmas digitales sobre los resúmenes de los mensajes durante todo el transcurso del protocolo para poder garantizar la autenticidad del origen de los mensajes y su integridad ante terceros.

Tanto los códigos de autenticación de mensajes como los algoritmos de firma digital se fundamentan sobre el uso de funciones de *hash* no reversibles, que a partir de un determinado texto de entrada producen un resumen de una longitud fija. Para ello, normalmente la longitud del mensaje se amplía hasta una determinada longitud (por ejemplo, 1024 bits) y en la ampliación del mensaje se incluye su tamaño original, como una forma de dificultar ataques de búsqueda de colisiones.

Además, los sistemas blockchain incluyen las funciones de *hash* como una parte fundamental de su operación. Por una parte, para enlazar los bloques se utiliza una función de *hash* que opera a partir del bloque y del *hash* del bloque anterior, construyendo de esta forma la cadena de bloques vinculados como una estructura de *hash* incremental. Por otra parte, el problema criptográfico que deben abordar los participantse para alcanzar cosnenso bajo el modelo de *Proof of Work* de sistemas como Bitcoin supone la resolución de un problema NP-completo consistente en hallar un valor de *hash* menor que un determinado número. [Bal17]

Dada la importancia de las funciones de *hash*, las funciones utilizadas han de ser seguras y robustas. Para ello, una función H que aplicada a un bloque de longitud arbitraria m produce una salida h de longitud fija debe cumplir las siguientes propiedades: [SBBB12]

- **Camino único:** Para un h dado, debe ser computacionalmente imposible encontrar un m tal que $H(m) = H(n)$.
- **Resistencia a colisión débil:** Para un m dado, debe ser computacionalmente imposible encontrar un n, tal que $H(m) = H(n)$.
- **Resistencia a colisión fuerte:** Debe ser computacionalmente imposible encontrar un par (m, n) tal que $H(m) = H(n)$.

Existen dos formas de atacar una función de *hash* segura: el criptoanálisis y la fuerza bruta. El criptoanálisis supone explotar vulnerabilidades lógicas o matemáticas del algoritmo. La fortaleza de una función de *hash* contra los ataques de fuerza bruta reside únicamente en la longitud del código de *hash* producido por el algoritmo. Para una función de *hash* de longitud n, el nivel de esfuerzo requerido se sitúa en los órdenes que muestra el cuadro 5.2.

Función resistente a preimagen	2^n
Función resistente a segunda preimagen	2^n
Función resistente a colisiones	$2^{n/2}$

Cuadro 2.2: Resistencia según longitud de función de *hash*

En caso de requerir resistencia a colisiones, lo que resulta esencial para cualquier función de *hash* de propósito general que pretenda presentarse como segura, el valor que tome el factor $2^{n/2}$ determina la resistencia planteada por la función ante ataques de fuerza bruta.

Las funciones de *hash* más utilizadas son la familia de algoritmos SHA (*Secure Hash Algorithm*). Su primera versión fue desarrollada por el NIST (*National Institute of Standards and Technology*) y publicada en 1993. El descubrimiento de vulnerabilidades en el algoritmo llevó a la presentación de una nueva versión en 1995, referida como SHA-1, la cual genera un valor de *hash* de 160 bits. En 2002, el NIST produjo una versión revisada del estándar que definía tres nuevas versiones del algoritmo SHA, que permiten generar valores de *hash* de longitudes de 256, 384 y 512 bits, conocidas respectivamente como SHA-256, SHA-384 y SHA-512. Estas tres versiones se conocen de forma colectiva como SHA-2, y hacen uso de la misma estructura y operaciones aritméticas y lógicas que SHA-1. Aunque SHA-512 parecía suficientemente segura, el NIST decidió estandarizar una nueva función denominada SHA-3, muy diferente en términos estructurales a SHA-1 y a SHA-2, siendo publicada en 2012 como una alternativa a SHA-2. [ADMG+16]

Una de las mayores amenazas a la seguridad de los mecanismos criptográficos actuales es la llegada de los procesadores cuánticos. Existen algoritmos cuánticos, como el algoritmo de Shor, que permiten realizar en tiempo polinomial operaciones como la factorización de números enteros, logaritmos discretos de cuerpo fiito y logaritmos discretos de curva elíptica [Sho99] [BL95], mediante su reducción a un problema de subgrupo oculto abeliano. Otro algoritmo cuántico es el algoritmo de Grover, capaz de reducir en orden cuadrático el número de consultas requeridas para la resolución de problemas de búsqueda en un escenario de caja negra [Gro97] [BBHT98] [BHMT02].

Esto supone una grave amenaza a técnicas criptográficas como el intercambio de claves de Diffie-Hellman y la encriptación y firmas digitales basadas en el algoritmo RSA, cuya seguridad podría verse gravemente comprometida en un escenario en el que los computadores cuánticos se encontraran perfeccionados y disponibles, debido a que se basan en operaciones vulnerables según el enfoque seguido por algoritmos similares al de Shor. Otras técnicas criptográficas de diferente naturaleza, como las basadas en funciones de *hash* y cifradores simétricos se verían afectadas en menor grado. Puesto que su nivel de seguridad se basa en la longitud empleada (el tamaño de la clave en el caso de los cifradores simétricos y el tamaño del valor de salida en el caso de las funciones de *hash*), y el algoritmo de Grover en este caso sólo logra reducir a la mitad el número de consultas necesarias para resolver este tipo de problemas, un mecanismo de defensa de carácter conservador para protegerse ante ataques por parte de computadores cuánticos consistiría en aumentar la longitu de la clave utilizada en este tipo de sistemas criptográficos. No obstante, la comparación del número de consultas como modelo de medida sólo es adecuanda en caso de que el coste de las consultas en cada caso sea equiparable.

En las arquitecturas de los procesadores cuánticos actuales, la mayor sobrecarga deriva del hecho de que el tiempo durante el que los qubits físicos pueden mantener un estado coherente es finito, lo que actualmente requiere el uso de técnicas de computación clásica de cálculo y corrección del error para estabilizar y mantener un valor reflejado por dicho estado cuántico [FWH12]. La introducción forzosa de un componente de computación clásica en el modelo de computación cuántica con esta fnalidad supone un cuello de botella respecto a lo que podría ser un sistema computacional puramente cuántico. No obstante, se espera que el perfeccionamiento de los computadores cuánticos en los próximos años reduzca esta sobrecarga [KBF+15], idealmente llegando a eliminarla.

[ADMG+16] analiza un ataque de pre-imagen cuántico mediante el algoritmo de búsqueda de Grover sobre las funciones criptográficas de *hash* SHA-256 y SHA3-256 en función del coste computacional necesario para llevar a cabo el ataque con éxito. Los resultados obtenidos muestran que para realizar un ataque contra la función SHA-256 se necesitan aproximadamente $2^{153,8}$ ciclos de código de superficie [FMMC12], mientras que para realizar un ataque contra la función SHA3-256 se necesitan aproximadamente $2^{146,5}$ ciclos de código de superficie, lo que en ambos casos y según el modelo de las arquitecturas cuánticas actuales y el componente de computación clásica que incluyen se traduce en aproximadamente 2^{166} operaciones básicas. Esto permite concluir que el número de operaciones actualmente necesarias en un escenario cuántico para realizar un ataque contra este tipo de funciones de *hash* está acotado inferiormente, y aumentando la longitud del valor de salida (que define el nivel de seguridad proporcionado por la función) en base a la tecnología disponible podría obtenerse un nivel de seguridad en principio suficiente para este tipo de funciones en un escenario post-cuántico.

Brassard, Høyer y Tapp presentan una extensión al algoritmo de Grover que, en un escenario específico, puede reducir en orden cúbico el número de consultas necesarias en el proceso de búsqueda [BHT98], frente a la mejora de orden cuadrático obtenida por el algoritmo de Grover en su forma clásica. La investigación realizada en [Ber09] muestra cómo los SHARCS (*Special-Purpose Hardware for Attacking Cryptographic Systems*), el hardware tradicionalmente utilizado para realizar ataques a sistemas criptográficos, en particular los circuitos de búsqueda de colisiones ideados inicialmente por Oorschot y Wiener [VOW94] hace más de dos décadas, pueden ser más efectivos en términos de coste para hallar colisiones en funciones de *hash* que los computadores cuánticos, incluso operando bajo suposiciones optimistas relativas a la velocidad de éstos últimos. Especialmente si operan haciendo uso de técnicas de paralelismo, el hardware clásico para la detección de colisiones supone una mayor amenaza para la seguridad de las funciones de *hash* que los computadores cuánticos.

Los datos ofrecidos hasta el momento permiten mostrar cómo muchas de las técnicas criptográficas comúnmente utilizadas, basadas en operaciones como la factorización de enteros, el cálculo de logaritmos o la criptografía de curva elíptica [JMV01] pueden ver gravemente comprometida su seguridad ante el advenimiento de los computadores cuánticos. Aunque es reseñable la existencia de esquemas de firma específicamente diseñados para ser seguros en un escenario en el que los computadores cuánticos se encuentren disponibles y operando bajo suposiciones muy optimistas, como XMSS (*eXtended Merkle Signature Scheme*) [BDH11], las técnicas de cifrado simétrico y las funciones de *hash* utilizadas para proporcionar autenticación e integridad en los sistemas replicados tolerantes a fallos bizantinos, debido a su diferente naturaleza presentan un nivel de vulnerabilidad mucho menor a este tipo de amenaza, siendo incluso más susceptibles a ataques realizados con hardware de propósito específico que ha existido durante más de dos décadas. En estos casos, el incremento del factor de seguridad de la función, como el tamaño del campo de salida, puede ser una medida suficiente para garantizar la fortaleza y resistencia a colisiones de este tipo de funciones.

2.6 Consenso tolerante a fallos bizantinos en blockchains permisionados

El interés suscitado por los protocolos de consenso, especialmente en los últimos dos años debido a su aplicabilidad para las plataformas blockchain, ha dado lugar a una gran variedad de algoritmos de este tipo. Algunos ejemplos de plataformas blockchain permisionadas, tanto comerciales como de código abierto, se enumeran a continuación junto a los algoritmos de búsqueda de consenso de los que se sirven. Una descripción de cada una de estas plataformas junto con sus respectivos algoritmos de consenso puede encontrarse en el estudio realizado en [CV17].

- Hyperledger Fabric[1] (PBFT [CL02])
- Tendermint[2]
- Symbiont Assembly[3] (BFT-SMaRT [SAB13])
- R3 Corda[4] (Raft [OO14] y BFT-SMaRT)
- Iroha[5] (Sumeragi)
- Kadena[6] (Juno[7] y ScalableBFT [Mar16])
- Chain[8] (Federated Consensus [20117]

[1] Hyperledger Fabric `https://github.com/hyperledger/fabric`
[2] Tendermint `https://github.com/tendermint/tendermint`
[3] Symbiont Assembly `https://symbiont.io/technology`
[4] Corda `https://github.com/corda/corda`
[5] Iroha `http://iroha.readthedocs.io/en/latest/`
[6] Kadena `http://kadena.io/`
[7] Juno `https://github.com/kadena-io/juno`
[8] Chain Core `https://chain.com/`

- Quorum[9] (QuorumChain[10] y Raft)
- Multichain[11]

Con la excepción de las versiones de Hyperledger posteriores a 2016, cuyo caso particular se comentará más adelante, todas estas plataformas mantienen una filosofía de diseño similar a la de los blockchains no permisionados, que fueron pioneros en este tipo de plataformas. Sin embargo, un diseño adecuado para blockchains públicos y no permisionados, construidos para operar sobre divisas criptográficas, no tiene por qué ser adecuado para aplicaciones de negocio que busquen aprovechar los beneficios de un registro de documentos incremental y distribuido sin depender de ningún tipo de divisa criptográfica. Los principales aspectos de diseño heredados de los blockchains no permisionados que actúan como un factor limitante en los blockchains permisionados son los siguientes:

- El fuerte acoplamiento entre el algoritmo de consenso utilizado y la plataforma (ya se trate de un algoritmo de tipo BFT o una técnica del estilo de *Proof of Work*), de una forma difícilmente modularizable, provoca que las plataformas de blockchain permisionado suelan extrapolar a su funcionamiento general restricciones innecesariamente heredadas del algoritmo de consenso utilizado a nivel subyacente. En los blockchains permisionados suelen utilizarse algoritmos de consenso tolerantes a fallos bizantinos que operan en un escenario asíncrono, requiriendo al menos $3f + 1$ réplicas. El hecho de que la consecución de consenso se encuentre acoplada con la ejecución de las operaciones provoca que esta restricción se traslade a fases ajenas a la propia búsqueda de consenso. En aquellas plataformas en las que la ejecución de las operaciones a aplicar sobre el estado compartido dista de ser computacionalmente trivial, una elección inadecuada respecto al modelo de ejecución de estas operaciones puede erigirse en el principal cuello de botella del sistema. Por el contrario, la separación de las fases de acuerdo y ejecución en niveles separados escogiendo estrategias adecuadas para cada uno de ellos, como sugiere [YMV+03], puede aumentar la productividad del sistema notablemente.
- Igualmente, la falta de modularización en el diseño del sistema crea dependencias indeseadas entre los componentes. Un ejemplo es el del algoritmo de consenso del sistema. Si el sistema se diseña de tal forma que su funcionamiento a alto nivel tenga dependencias estrictas con un algoritmo de consenso en concreto, que a su vez puede encontrarse acoplado con el modelo de ejecución, reemplazar el algoritmo de consenso utilizado por la plataforma se convierte en una labor prácticamente imposible.
- El uso de técnicas basadas en el modelo de replicación activa provoca que la ejecución de operaciones no deterministas puedan anular el efecto del consenso alcanzado,

[9] Quorum `https://chain.com/`
[10] Quorum Chain `https://github.com/jpmorganchase/quorum`
[11] MultiChain `https://github.com/MultiChain/multichain`

produciendo divergencias en el sistema que incumplan el modelo de consistencia establecido.

- El hecho de que las operaciones deban ejecutarse en todas las réplicas contribuye a degradar la productividad del sistema, al aumentar la carga por nodo. En caso de contar con un subconjunto de nodos de confianza de dimensión suficiente, las operaciones podrían ejecutarse en dichos nodos y propagar los cambios sobre el estado compartido a las demás réplicas, lo que constituiría una transición desde el modelo de replicación activa a una variante del modelo de replicación pasiva.

Todas estas consideraciones llevan a pensar que una plataforma blockchain permisionada de propósito general debería diseñarse siguiendo un enfoque de consenso modular, de tal forma que sus componentes puedan reemplazarse atendiendo a unas interfaces de operación bien definidas. Hyperledger Fabric es un caso paradigmático de las observaciones realizadas. En sus versiones iniciales, esta plataforma de blockchain permisionado auspiciada por la Linux Foundation presentaba un diseño similar al que se ha descrito inicialmente, heredado de las plataformas blockchain no permisionadas. Sin embargo, la detección de estas deficiencias en el diseño, principalmente el acoplamiento entre los módulos de consenso y ejecución unido a la dependencia de todo el sistema respecto al algoritmo de consenso específico utilizado, condujo a que el sistema sufriera un rediseño completo en 2016. El nuevo diseño se concibió especialmente para evitar la mayor parte de estos inconvenientes y adoptar un enfoque modular, estructurado en las capas de validación, orden y ejecución. [Vuk17]

Es reseñable el hecho de que, aunque estas plataformas suelen presentar un componente de validación externa que comprueba la corrección de las operaciones antes de ejecutarlas y aplicar sus cambios sobre el estado compartido, como es el caso de Hyperledger Fabric o el algoritmo de búsqueda de consenso Turquois [MNC12], ninguno de los algoritmos empleados por dichas plataformas hace un uso más extensivo del componente de validación, en aplicaciones como la detección de nodos maliciosos.

Las consideraciones realizadas corresponden a aspectos relativos al diseño de plataformas blockchain permisionadas y su arquitectura, aunque en cualquier caso estas plataformas deben disponer de un algoritmo que les permita alcanzar consenso sobre la secuencia de operaciones cuyos cambios resultantes se aplicarán en el estado compartido entre las réplicas. Como se detallará en el capítulo 6, el presente trabajo versa sobre el diseño y desarrollo de un algoritmo de búsqueda de consenso tolerante a fallos bizantinos inspirado principalmente en Raft [OO14]. Los principios de diseño de Raft, comentados en la sección 5.1, son la separación de problemas, su inteligibilidad y la reducción del espacio de estados. Un trabajo ya existente en esta línea es BFT-Raft [CZ14], que combina los principios de diseño de Raft con elementos extraídos de PBFT [CL+99] para proporcionar tolerancia a fallos bizantinos. Este algoritmo posee graves fallos de diseño que comprometen su seguridad y viveza, pero también proporciona algunas ideas cuyo aprovechamiento puede sr de utilidad a la hora de

diseñar un algoritmo de similar naturaleza. Por este motivo, este capítulo, dedicado al estudio del estado actual de la investigación en el ámbito de los algoritmos de consenso tolerantes a fallos bizantinos y su aplicación a sistemas blockchain permisionados, concluye con el análisis del algoritmo BFT-Raft en sus aspectos tanto positivos como negativos.

Los principales elementos incorporados en BFT-Raft para ampliar el modelo de Raft proporcionándole tolerancia a fallos bizantinos son los siguientes:

- **Firmado de mensajes:** El uso de técnicas de firma digital permite proporcionar autenticación e integridad a los mensajes. BFT-Raft opera en un entorno autenticado mediante criptografía de clave asimétrica, en el que tanto clientes como réplicas servidoras se identifican ante los demás mediante su clave pública. Cada réplica almacena las claves públicas de las demás réplicas y de los clientes, pero ambos almacenes de claves se encuentran separados, de tal forma que cada entidad sólo pueda enviar los tipos de mensajes que le corresponden según su naturaleza, evitando situaciones como que una réplica envíe una petición de operación pretendiendo ser un cliente. Dado que las funciones de firma digital se utilizan en varias partes del protocolo, todos los mensajes se encuentran firmados por su remitente.

- **Intervención activa de los clientes:** BFT-Raft permite que los clientes fuercen un cambio de líder, en caso de que el sistema no esté avanzando con el líder actual. La finalidad de este mecanismo es evitar que un líder bizantino impida el progreso del sistema, provocando inanición en un conjunto de clientes.

- **Hashing incremental:** Cada réplica de BFT-Raft calcula un *hash* criptográfico al añadir una nueva entrada a la lista de operaciones resultante del consenso distribuido, denominada *log* en el ámbito de Raft. Este es el procedimiento empleado por los sistemas blockchain para asegurar integridad, consistencia y trazabilidad de las secuencias de operaciones. Cada *hash* se calcula incluyendo la entrada actual junto con el *hash* de la entrada anterior, lo que permite a la réplica dejar constancia de la cadena de operaciones que ha replicado, y a otras réplicas comprobar dicha cadena con sólo comprobar la firma del último elemento.

- **Verificación de elección:** Cuando un nodo se convierte en líder, su primer mensaje a las réplicas restantes incluye un quórum de votos positivos recibidos por parte de otros nodos, con el fin de certificar su liderazgo ante toda la red. Dichos votos se encuentran firmados por sus emisores, lo que permite a cada contarlos y validarlos, verificando su legitimidad.

- **Verificación de confirmación:** Para evitar delegar una excesiva responsabilidad en el líder, lo que podría comprometer las condiciones de seguridad del sistema ante un líder bizantino, antes de confirmar la adición de una operación a la lista replicada no sólo se requiere la confirmación del líder, sino un quórum de confirmaciones por parte de las demás réplicas. Esta descentralización de la toma de decisiones se aleja del modelo de Raft, asemejándose más a la forma de alcanzar consenso distribuido en PBFT.

- **Voto diferido:** Los nodos no otorgan su voto a un candidato a no ser que sospechen que el líder actual ha caído o ha sido comprometido. Un nodo puede tener indicios de este hecho si el *timeout* que monitoriza la comunicación con el líder expira (idealmente, este comportamiento estaría encapsulado en un detector de fallos que abstrajera al algoritmo de las consideraciones relativas a la sincronía) o si recibe un mensaje de un cliente notificándole que el líder está fallando. Si sucede lo segundo pero no lo primero, es posible que un líder bizantino esté marginando a un sector determinado de los clientes. Este mecanismo impide que un nodo bizantino que dé lugar a un proceso de elección de forma innecesaria se convierta en líder y pueda corromper o subvertir el sistema.

Las propiedades de firmado de mensajes, *hashing* incremental y verificación de confirmación pretenden mantener las condiciones de seguridad del sistema, mientras que las propiedades de intervención activa de los clientes, verificación de elección y voto diferido buscan mantener las condiciones de viveza o progreso. BFT-Raft descompone el problema de consenso en dos sub-problemas relativamente independientes, de forma similar a Raft: elección de líder y replicación incremental de una secuencia ordenada de operaciones. BFT-Raft asimismo busca ofrecer las mismas garantías de seguridad que Raft, adaptando las condiciones que permiten cumplir con dichas garantías al escenario de tolerancia a fallos bizantinos:

- En cada etapa sólo se puede elegir a un líder como máximo.
- Un líder honesto nunca sobrescribe o elimina elementos de su secuencia de operaciones, se limita a añadir nuevos elementos.
- Si dos nodos tienen el mismo *hash* incremental en la misma posición de la lista, sus *logs* son idénticos en todos los elementos propios hasta dicha posición.
- Si una operación se confirma en una determinada etapa, entonces la entrada asociada se encontrará en los *logs* de los líderes de de todas las etapas posteriores (numeradas con un índice mayor).
- Si un nodo honesto ha ejecutado una operación almacenada en una determinada posición de su *log*, ningún nodo honesto ejecutará una operación diferente en dicha posición.

Pese a que BFT-Raft proporciona algunas interesantes, como la verificación de elecciones o el voto diferido, posee fallos de diseño de considerable magnitud y notable gravedad. Por ejemplo, la propiedad de voto diferido obra con el afán de evitar que un nodo bizantino pueda evitar el progreso del sistema propiciando nuevos procesos de elección de líder continuamente. Sin embargo, el hecho de delegar responsabilidad a los clientes permitiendo que éstos sean quienes puedan dar comienzo a un proceso de elección de líder desvirtúa totalmente este propósito. En realidad, la probabilidad de que un cliente haya sido comprometido es asumiblemente mayor que la de que una réplica servidora pudiera haberlo sido.

Cachin y Vukolić detallan otros fallos de diseño de de BFT-Raft en [CV17], como situaciones en las que un líder bizantino podría crear deliberadamente una bifurcación en la cadena de operaciones, incumpliendo el modelo de consistencia lineal que debe mantener el sistema. Como demuestran los autores, el protocolo de BFT-Raft incluso incumple las garantías de acuerdo, al emplear la primitiva de comunicación conocida como difusión consistente bizantina (BCB, *Byzantine Consistent Broadcast*) [ST87], que mantiene las restricciones de consistencia pero no garantiza que la situación de acuerdo distribuido llegue a alcanzarse. Una primitiva de difusión que, en cambio, sí permite alcanzar acuerdo distribuido conforme a las garantías del protocolo de consenso es la difusión fiable bizantina (BRB, *Byzantine Reliable Broadcast*) [Bra87], utilizada en BFT con este fin y que se sirve de una segunda ronda de intercambio de mensajes para confirmar que se alcanza la situación de acuerdo.

Todos los trabajos presentados poseen aspectos de concepción y diseño que resultan enormemente interesantes a la hora de diseñar un algoritmo de búsqueda de consenso distribuido, ya sea por su innovación, aplicación práctica o incluso, en algunos casos, como muestra de un compendio de errores a evitar a lo largo del proceso de desarrollo.

Capítulo 3

Trebizond, algoritmo de consenso bizantino para sistemas blockchain permisionados

Los antecedentes expuestos hasta el momento conducen a que en este capítulo se proceda a presentar el objetivo principal del presente trabajo, el diseño de Trebizond, un algoritmo de consenso tolerante a fallos bizantinos especialmente apropiado para su aplicación en sistemas blockchain permisionados.

El objetivo de Trebizond es mantener un estado compartido entre una serie de nodos distribuidos, que actúan como réplicas del sistema. El algoritmo mantiene garantías de consistencia estricta de acuerdo con la propiedad de linearizabilidad y opera en condiciones de sincronía eventual. Al tratarse de un algoritmo especialmente apropiado para sistemas blockchain permisionados, su despliegue se realiza en un escenario autenticado mediante criptografía de clave asimétrica, asumiendo que de forma previa a la ejecución del algoritmo todos los nodos cuentan con las claves públicas de los demás nodos y de los clientes que pueden enviarles peticiones, emplazadas desde un centro confiable de distribución de claves, como Kerberos[1]. La presencia de claves criptográficas se utiliza para firmar digitalmente los mensajes que cada entidad de la red (tanto clientes como nodos servidores) envía a los demás, y el hecho de que cada nodo conozca las claves públicas de todos los demás permite que un nodo pueda comprobar la autenticidad de un mensaje enviado por otro nodo o por un cliente, aunque dicho mensaje no estuviera originalmente dirigido a él. Se asume que un atacante no puede generar una firma válida en nombre de un nodo que no ha corrompido y del que, por tanto, no conoce su clave privada.

Este algoritmo está influenciado principalmente por PBFT [CL02] y por Raft [OO14], el cual a su vez incorpora mecanismos inspirados en *Viewstamped Replication* [OL88]. De Raft se toman sus principios de diseño, que se fundamentan en la separación de problemas, la reducción del espacio de estados y la inteligibilidad del algoritmo. De PBFT se toman algunos de los elementos que permiten dotar al algoritmo de tolerancia a fallos en un escenario bizantino. Como se comentó en el capitulo 5, existe una variante de Raft que afirma proporcionar tolerancia a fallos bizantinos, denominada PBFT-Raft o Tangaroa [CZ14]. Aunque, como ya se expuso, este algoritmo tiene notables fallos de concepción que hacen que fracase en su empeño, también incorpora algunos elementos interesantes que se aprovechan en Trebizond,

[1] Kerberos `http://web.mit.edu/kerberos/`

como el *hashing* incremental de la secuencia de operaciones ejecutadas sobre la máquina de estados replicada y la verificación de situaciones que han de ser respaldadas por un cierto número de nodos, a través del uso intensivo de funciones de firma digital en el algoritmo. Además, Trebizond incorpora una separación explícita entre las capas de consenso y ejecución, tal como propone [YMV+03], lo que permite que la ejecución de las operaciones se pueda llevar a cabo por un número de réplicas menor que el necesario para obtener consenso distribuido, como en el caso de las versiones recientes de Hyperledger [Vuk17], aumentando así la productividad del sistema replicado.

Respecto al estado actual de la investigación en el área de los algoritmos de consenso tolerantes a fallos bizantinos, la principal contribución de Trebizond es la combinación de la detección de fallos activa y la validación semántica. Mientras que otros algoritmos utilizan la detección activa de fallos como una forma de mejorar el factor de tolerancia a fallos frente a la táctica de enmascaramiento habitualmente empleada [HKD06], y otros algoritmos hacen uso de técnicas de validación semántica o externa a la hora de valorar si es lícito ejecutad una operación sobre el estado compartido [MNC12], Trebizond propone fusionar ambas técnicas. Su finalidad es utilizar la validación semántica, dependiente del protocolo de nivel superior que se ejecuta sobre el algoritmo de consenso, como parte del propio mecanismo de detección activa de fallos, de tal forma que se mejore su eficacia y completitud. Este refinamiento del detector de fallos permite, por ejemplo, deponer a un líder cuando los nodos detectan que éste está enviado operaciones que son coherentes con el propio algoritmo de consenso pero que violan la semántica del protocolo de nivel de aplicación, o aislar a una réplica del grupo de difusión cuando se tienen evidencias de comportamiento bizantino por su parte.

El establecimiento de las propiedades de seguridad y viveza del algoritmo se realiza mediante lógica de primer orden, mientras que sus propios diseño y funcionamiento se especifican utilizando autómatas de entrada/salida, una técnica bien conocida para el modelado formal de sistemas concurrentes asíncronos y, en particular, sistemas distribuidos.

Esta especificación encuentra su correspondencia en el caso práctico de una implementación, realizada con el ánimo de mostrar paradigmas y patrones que facilitan la codificación de algoritmos distribuidos a partir de una especificación modelada formalmente, aplicando buenas prácticas para la implementación de máquinas de estados replicadas como las que se sugieren en [Sch90].

3.1 Método de especificación: Autómatas de entrada salida

El diseño de Trebizond se plasmará modelando la especificación de su funcionamiento mediante autómatas de entrada/salida.

Los autómatas de entrada/salida son una técnica que proporciona un modelo formal que permite describir la mayor parte de los sistemas concurrentes asíncronos, adaptándose especialmente bien al caso de los sistemas distribuidos. Fue propuesto por Lynch y Tuttle en la tesis de máster de éste último. [LT87]

Una de las características típicas de los sistemas concurrentes es que, en lugar de realizar una labor de cómputo ante una determinada entrada y seguidamente detenerse, prosiguen en su ejecución, recibiendo sucesivos eventos de entrada ante los que operan de forma reactiva. Aunque los autómatas de entrada/salida también pueden utilizarse para modelar sistemas síncronos, son particularmente apropiados para modelar sistemas cuyos componentes se comportan de forma asíncrona.

En este modelo, cada componente del sistema se representa como un autómata de entrada/salida, que en esencia es un autómata con una cantidad de estados posiblemente infinita y en el que cada transición se etiqueta como una acción. En el modelo se realiza una distinción muy clara entre diferentes tipos de acciones atendiendo a su origen, clasificándolas en acciones de entrada, de salida e internas. Un autómata genera acciones de salida e internas de forma autónoma, transmitiendo las acciones de salida a su entorno de forma instantánea. Las acciones de entrada son generadas por el entorno y transmitidas instantáneamente al autómata. La distinción explícita entre las acciones de entrada y las acciones restantes es fundamental en el modelo, ya que el origen de las acciones condiciona el momento de su ejecución: un autómata puede establecer restricciones respecto a cuándo ejecutar una acción interna o de salida, pero no puede bloquear una acción de entrada y su consiguiente procesamiento, propiedad heredada del modelo de memoria compartida presentado en [LF79].

Los autómatas de entrada/salida pueden ser no deterministas, puesto que la capacidad para tolerar el indeterminismo es una de las principales señas de la capacidad descriptiva del modelo. Puesto que un indeterminismo elevado puede provocar que los resultados obtenidos en relación al algoritmo sean demasiados generales, normalmente se suelen aplicar ciertas restricciones sobre los componentes indeterministas del sistema con el fin de discretizar o reducir el espacio de decisiones.

Los autómatas de entrada/salida pueden combinarse para posibilitar su interacción. Cuando se componen una serie de autómatas se mantiene la misma denominación sobre las acciones de cada uno de ellos. Esta composición garantiza que si un autómata tiene una acción de salida π, entonces π será una acción de entrada de todos los demás autómatas que tengan una acción con dicho nombre π. Como resultado, un autómata que genere una acción de salida lo hará de forma autónoma, y esta salida se transmitirá instantáneamente a todos los otros autómatas que la tengan como una acción de entrada.

Además, varios autómatas pueden componerse en uno solo mediante adición. Dicha composición afecta específicamente tanto a sus acciones, de los tres tipos descritos, como a sus variables internas de estado. Este mecanismo permite descomponer el algoritmo que se ejecuta en un nodo de un sistema distribuido en varios problemas diferentes y resolverlos por separado. Mientras que el proceso alternativo, abordar la resolución de todos los problemas que componen el algoritmo como una única entidad, es notablemente más propensa a error, esta separación de los problemas individuales que conforman el algoritmo permite diseñar la resolución de cada problema de una forma más clara, dando lugar a una solución global más robusta.

La operación de los autómatas de entrada/salida genera *ejecuciones*, secuencias que alternan estados y acciones. De entre todas las ejecuciones de un autómata, las más relevantes son las ejecuciones *justas*, aquellas que permiten que cada componente primitivo del autómata tenga infinitas oportunidades de ejecutar sus acciones internas o de salida. Las ejecuciones justas de un autómata dan lugar a los *comportamientos justos* del autómata, que son las sub-secuencias de ejecuciones justas que se componen de acciones externas (de entrada y de salida). Este conjunto de secuencias puede consistir en secuencias finitas e infinitas de acciones, y abarca el comportamiento interesante de un autómata de entrada/salida.

El modelo de autómatas de entrada/salida constituye una buena base formal para realizar pruebas de corrección, determinando si un algoritmo efectivamente soluciona unos problemas determinados de acuerdo a su especificación. La modularidad que otorga la posibilidad de descomponer el algoritmo que se ejecuta en un autómata, junto con el nivel de abstracción que proporciona el hecho de que el modelo se centre en las acciones de los autómatas, equivalentes semánticos de las interacciones entre los componentes del sistema, lo hace especialmente apropiado para este fin. La formalización del modelo podría posibilitar automatizar las pruebas sobre algunos problemas bien conocidos y especificados, mediante herramientas software de verificación formal. [LT89]

3.2 Especificación del algoritmo

La especificación de Trebizond se compone de varios algoritmos que permiten resolver cada uno de los problemas individuales que aborda el algoritmo general, siguiendo el enfoque de separación de problemas aplicado en Raft. Esta descomposición permite que cada uno de los algoritmos parciales obtenidos tenga una complejidad menor que favorezca su inteligibilidad y reduzca la probabilidad de cometer errores durante el proceso de diseño o de implementación, puesto que cada algoritmo parcial está dirigido a la resolución de un único problema. La especificación del algoritmo se realiza con autómatas de entrada/salida, cuyos mecanismos de composición permiten la obtención del algoritmo general mediante la superposición de los algoritmos parciales, obteniendo de forma trivial un algoritmo que soluciona simultáneamente los problemas abordados.

A la hora de descomponer el problema principal que el algoritmo a diseñar pretende resolver, es importante alcanzar un nivel de granularidad adecuado. Una descomposición con un nivel de granularidad excesivamente bajo daría lugar a problemas de alto nivel, cuya resolución completa sigue siendo difícil de abordar. En el otro extremo, una descomposición con un nivel de granularidad excesivamente alto daría lugar a un número elevado de problemas de bajo nivel, que requerirían un gran número de algoritmos para su resolución; aunque estos algoritmos fueran sencillos, su elevada cantidad también añadiría complejidad innecesaria al proceso de diseño y dificultaría la capacidad para realizar abstracciones sobre los componentes del algoritmo.

El problema de búsqueda de consenso distribuido tolerante a fallos bizantinos que Trebizond pretende resolver se ha descompuesto en los siguientes problemas, cuyas especificaciones individuales se presentan a continuación mediante el modelo de autómatas de entrada/salida: algoritmo de sucesión rotatoria de líder, algoritmo de detección activa de fallos, algoritmo de difusión atómica y algoritmo del cliente.

3.2.1 Algoritmo de sucesión rotatoria de líder

El primer algoritmo que compone el modelo de operación de los nodos de la máquina de estados replicada en Trebizond es el algoritmo de designación de líder. La elección del líder se basa en el número de etapa que todas las réplicas incluyen como una variable de su estado. Mientras que algoritmos como Raft implementan un algoritmo de elección de líder donde cualquier nodo puede postularse como candidato y dar lugar a una elección, el modelo de Trebizond es más similar al de PBFT, en el que el líder de cada etapa se calcula a través del operador módulo entre el número de la etapa, el número total de réplicas y el identificador de cada réplicas. Uno de los defectos de Tangaroa es que exhibe un mecanismo de elección de líder inspirado directamente en el de su predecesor Raft, que le añade complejidad y situaciones que pueden dotar al algoritmo de vulnerabilidad ante fallos bizantinos si no se controlan adecuadamente. Trebizond sí incorpora el mecanismo de validación de votos de Tangaroa, que se basa en mensajes firmados para poder atestiguar la mayoría de votos obtenida ante terceros.

Frente al algoritmo de PBFT, que realiza un reemplazo de líder cuando los nodos sospechan que el actual líder ha fallado, en Trebizond todos los nodos tienen un temporizador fijo ante cuya expiración cada uno envía su correspondiente voto firmado al nodo que considera que debe ser el líder en base al número de la siguiente etapa. La duración de una etapa es suficientemente prolongada como para esperar que el nuevo líder reciba una cantidad de votos suficiente antes de una nueva expiración del temporizador. Cuando un nodo cuenta con votos suficientes para una etapa posterior, envía dichos votos a los demás nodos, que admitirán su liderazgo para dicha etapa. Hasta que un líder no se confirma, se mantendrá al líder anterior. En caso de que en una etapa no se pueda confirmar un nuevo líder, en la siguiente expiración del temporizador se incrementará el número de etapa y se enviará el voto correspondiente a la nueva etapa al siguiente nodo, puesto que una etapa posterior siempre tendrá prevalencia sobre una etapa anterior.

El hecho de forzar una sucesión de líder en lugar de mantenerlo hasta que demuestre un comportamiento defectuoso, como en PBFT, se debe a que en dicho modelo un líder con un comportamiento bizantino indetectable puede mantenerse en el puesto de forma indefinida, mientras que en el modelo implementado ningún nodo podría acaparar el liderazgo. La detección activa de fallos permite expulsar del grupo de comunicación a aquellos nodos que exhiban un comportamiento bizantino detectable, lo que contribuiría a reducir el número de nodos bizantinos que pueden acceder al liderazgo. En cualquier caso, en Trebizond la figura del líder es la de un centralizador del envío de mensajes que pretende reducir el número de mensajes entre los clientes y las réplicas, evitando la congestión en la red, además de facilitar la tarea de establecer orden total sobre los mensajes difundidos en el escenario de operación normal. Más allá de este comportamiento, las réplicas se comportan como las entidades integrantes de una red *peer-to-peer*. Por lo tanto, el líder no posee un papel tan crítico en la toma

de decisiones o la confirmación de las operaciones como en otros algoritmos como Raft, por lo que la existencia de un líder bizantino tiene un impacto mucho menor, que generalmente se limita a la capacidad de retrasar el procesamiento de los mensajes enviados desde los clientes.

La principal vulnerabilidad que sufre el algoritmo de sucesión rotatoria de líder diseñado es que cada nodo almacena todos los votos que recibe para las etapas posteriores. Un atacante que haya corrompido una o varias réplicas puede enviar votos a todos los nodos para número de etapa elevados con el fin de agotar su espacio de almacenamiento. Aunque el algoritmo sólo almacena el primer voto que recibe de una réplica para una determinada etapa, el impacto que un nodo bizantino podría causar mediante este tipo de ataque puede acotarse superiormente mediante la implementación de dos medidas complementarias. Por una parte, el establecimiento de un límite superior sobre el número de la etapa (incluyendo algún mecanismo para su reinicio cíclico), de tal forma que los votos para etapas posteriores se descarten. Por otar parte, cada vez que se produzca un incremento o actualización de la etapa actual conviene purgar la estructura de datos en la que se almacenan los votos de todo el contenido correspondiente a etapas previas, minimizando el espacio de almacenamiento utilizado para la gestión de los votos del líder.

A continuación se presenta la especificación del algoritmo de sucesión rotatoria de líder de Trebizond, modelado mediante autómatas de entrada/salida.

Nomenclatura:

Nombre de variable: *tipo de variable* $\leftarrow$ *valor inicial*
$\varsigma \rightarrow$ *firma*

Tipos de datos:

`VotoLiderazgo`: `voto: int, etapa: int`
`ConfirmacionLider`: `id: int, etapa: int, votos:` $[\varsigma($`liderId, etapa`$)]$ de $\frac{2n+1}{3}$ réplicas, cada uno firmado por su emisor

Variables de estado:

`id`: `int` $\leftarrow$ identificador de la réplica
`replicaIds`: `Lista<int>` $\leftarrow$ identificadores de las restantes réplicas
`etapaActual`: `int` $\leftarrow 0$
`liderActual`: `int` $\leftarrow 0$
`registroVotos`: `[nEtapa: int] [VotoLiderazgo]`
`n`: `int` $\leftarrow$ número de réplicas
$\mathtt{q}_{id,j} \forall j \in$ *replicaIds*: `Cola<mensaje>` $\leftarrow []$
$\mathtt{q}_{j,id} \forall j \in$ *replicaIds*: `Cola<mensaje>` $\leftarrow []$

Condiciones de seguridad:

1. Un nodo no envía más de un voto para una misma etapa.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot voto \cdot \alpha_2 \cdot voto \cdot \alpha_3 \rightarrow \alpha_2 = \alpha_{2,1} \cdot timeout \cdot \alpha_{2,2}$
2. No se envía un voto a un nodo si no es ante la recepción de un *timeout*.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot voto \cdot \alpha_2 \rightarrow \alpha_1 = \alpha_{1,1} \cdot timeout \cdot \alpha_{1,2}$

Condiciones de viveza:

1. La recepción de un *timeout* siempre supone el envío de un voto.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot timeout \cdot \alpha_2 \rightarrow \alpha_2 = \alpha_{2,1} \cdot voto \cdot \alpha_{2,2}$
2. La recepción de una cantidad suficiente de votos para una etapa igual o posterior a la actual supone el envío de un quórum de votos verificables a todos los otros nodos.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot \alpha_2 \wedge (|voto \subseteq \alpha_1| \geqslant \frac{2n+1}{3}) \wedge voto$ `.etapa` $\geqslant$ `etapa` $\rightarrow$
   $\rightarrow \alpha_2 = \alpha_{2,1} \cdot (\forall j \in replicaIds\, Confirmacion Lider_{i,j}) \cdot \alpha_{2,2}$

Entradas:

Voto de liderazgo$_i$: `VotoLiderazgo`, procedente de la réplica i
Confirmación de líder$_i$: `ConfirmacionLider`$_i$, procedente de la réplica i
Timeout Líder

Salidas:

Voto de liderazgo: `voto: int`, dirigido a la réplica votada
Confirmación de líder$_i$: `ConfirmacionLider`$_i$, dirigido a la réplica i

Operaciones:

(E) *Timeout* Líder. El nodo vota para la siguiente etapa a la réplica que considera que debe ser el siguiente líder en base al número de etapa.

acciones:
`etapaActual` $\leftarrow$ `etapaActual` $+1$
$q_{id,replicaIds[(etapaActual)\,\%n]} \leftarrow q_{id,replicaIds[(etapaActual)\,\%n]} \cdot$
$\cdot$ `VotoLiderazgo`$\{$ `etapaActual` $\%n$, `etapaActual` $\}$

(E) Voto de liderazgo, procedente de la réplica $j \rightarrow$ `VotoLiderazgo`

acciones:
$q_{j,id} \leftarrow q_{j,id} \cdot$ `VotoLiderazgo`

(E) Confirmación de líder, procedente de la réplica $j \rightarrow$ `ConfirmacionLider`

acciones:

$q_{j,id} \leftarrow q_{j,id} \cdot$ `ConfirmacionLider`

(S) Voto de liderazgo

precondiciones:

$q_{id,j} =$ `VotoLiderazgo` $\cdot\, q'_{id,j}$

(S) Confirmación de líder

precondiciones:

$q_{id,j} =$ `ConfirmacionLider` $\cdot\, q'_{id,j}$

(I) Confirmación de líder relativa a una etapa igual o posterior a la actual. Se actualizan la etapa y el líder actuales.

precondiciones:

$q_{id,j} =$ `ConfirmacionLider` $\cdot\, q'_{id,j}$

`ConfirmacionLider.etapa` $\geqslant$ `etapaActual`

acciones:

`liderActual` $\leftarrow j$

`etapaActual` $\leftarrow$ `ConfirmacionLider.etapa`

$q_{id,j} \leftarrow q'_{id,j}$

(I) Confirmación de líder relativa a una etapa anterior a la actual. Se ignora el mensaje.

precondiciones:

$q_{id,j} =$ `ConfirmacionLider` $\cdot\, q'_{id,j}$

`ConfirmacionLider.etapa` $<$ `etapaActual`

acciones:

$q_{id,j} \leftarrow q'_{id,j}$

(I) Voto de liderazgo relativo a una etapa igual o posterior a la actual y dirigida a esta réplica; no se ha recibido un voto previo del nodo de origen para la etapa indicada. Se almacena el voto.

precondiciones:

$q_{id,j} = \texttt{VotoLiderazgo} \cdot q'_{id,j}$

$\texttt{VotoLiderazgo.etapa} \geqslant \texttt{etapaActual}$

$\texttt{VotoLiderazgo.voto} = \texttt{id}$

$\nexists\ \texttt{voto} \mid \texttt{voto} \in \texttt{registroVotos[VotoLiderazgo.etapa]}[j]$

acciones:

$\texttt{registroVotos}[j] \leftarrow \texttt{registroVotos}[j] \cdot \texttt{VotoLiderazgo}$

$q_{id,j} \leftarrow q'_{id,j}$

(I) Voto de liderazgo relativo a una etapa igual o posterior a la actual y dirigida a esta réplica; sí se ha recibido un voto previo del nodo de origen para la etapa indicada. Se ignora el nuevo voto, descartándolo.

precondiciones:

$q_{id,j} = \texttt{VotoLiderazgo} \cdot q'_{id,j}$

$\texttt{VotoLiderazgo.etapa} \geqslant \texttt{etapaActual}$

$\texttt{VotoLiderazgo.voto} = \texttt{id}$

$\exists\ \texttt{voto} \mid \texttt{voto} \in \texttt{registroVotos[VotoLiderazgo.etapa]}[j]$

acciones:

$q_{id,j} \leftarrow q'_{id,j}$

(I) Voto de liderazgo relativo a una etapa anterior a la actual, o dirigido a otra réplica. Se ignora el mensaje.

precondiciones:

$q_{id,j} = \texttt{VotoLiderazgo} \cdot q'_{id,j}$

$\texttt{VotoLiderazgo.etapa} < \texttt{etapaActual} \vee \texttt{VotoLiderazgo.voto} \neq \texttt{id}$

acciones:

$q_{id,j} \leftarrow q'_{id,j}$

(I) Se ha alcanzado un número de votos suficiente para proclamarse líder en una etapa igual o superior a la actual. Se actualizan el líder y la etapa actuales y se envían los votos conseguidos como testimonio y confirmación del liderazgo adquirido a las demás réplicas.

precondiciones:

$\exists$ `etapa` $\mid$ `etapa` $\geqslant$ `etapaActual` $\wedge$ |`registroVotos[etapa]`| $\geqslant \frac{2n+1}{3}$

acciones:

`etapaActual` $\leftarrow$ `etapa`

`liderActual` $\leftarrow$ `id`

$\forall j \in replicaIds\, q_{id,j} \leftarrow q_{id,j} \cdot$ `registroVotos[etapa]`

Argumentación de las condiciones de seguridad:

1. Un nodo no envía más de un voto para una misma etapa.
   La etapa actual se incrementa siempre al recibir un *timeout*, con anterioridad al envío del voto, con lo que únicamente se envía un voto para cada etapa, y éste siempre es consecuencia de la recepción de un *timeout*.
2. No se envía un voto a un nodo si no es ante la recepción de un *timeout*.
   El envío de un voto sólo se realiza ante la recepción de un *timeout*, cuando se envía un voto referente a la etapa actual.

Argumentación de las condiciones de viveza:

1. La recepción de un *timeout* siempre supone el envío de un voto.
   La recepción de un *timeout* es una acción de entrada, que no se puede precondicionar, y entre cuyos consecuentes está incondicionalmente la emisión de un mensaje de voto de liderazgo para la etapa actual.
2. La recepción de una cantidad suficiente de votos para una etapa igual o posterior a la actual supone el envío de un quórum de votos verificables a todos los otros nodos.
   Esta circunstancia se controla como una acción interna cuyas precondiciones son que el registro de votos alberga una cantidad de votos referidos al presente nodo igual o superior a $\frac{2n+1}{3}$, para una etapa igual o superior a la etapa actual del nodo. Cuando esto ocurre, se actualiza la etapa actual a aquella para la que se ha obtenido el liderazgo, se actualiza el identificador del líder actual al del propio nodo y se envía a todos los nodos un mensaje de confirmación de líder que contiene el quórum de votos recibidos por parte de diferentes nodos para la etapa, firmados por sus respectivos remitentes. Dado que los precondicionantes de esta acción interna, aparte del número de etapa actual (que se actualiza al recibir un *timeout*, un quórum de votos o un mensaje de confirmación de líder para una etapa posterior), se basan en haber almacenado votos para la etapa, debe analizarse cómo se obtienen y gestionan éstos.

El voto de otro nodo para una etapa igual o posterior a la actual (en el momento de emitir el voto) se almacena ante su recepción si la etapa que figura en el voto es igual o superior a la actual y no se ha recibido un voto previo del mismo nodo para la misma etapa, comprobando la integridad de las firmas digitales en los mensajes de voto recibidos. Por lo tanto, únicamente se detectará que existe un quórum de votos suficiente para enviar una confirmación de líder si se han recibido votos suficientes de nodos diferentes, dirigidos al nodo actual para la etapa, y dicha etapa es igual o posterior a la actual en el momento de interpretar el quórum de votos.

3.2.2 Algoritmo de detección activa de fallos

Tal como se ha expuesto con anterioridad, la principal característica distintiva de Trebizond es su combinación de las técnicas de detección activa de fallos y validación semántica. Para ello, se propone un algoritmo que se ejecuta en el nivel inferior al protocolo de consenso, e intercepta todos los mensajes dirigidos al proceso. El algoritmo se estructura en una serie de colas, de acuerdo con los diferentes niveles de validación que los mensajes deben superar antes de su entrega al nivel superior, en el que se ejecuta el protocolo de consenso.

En primer lugar, los mensajes recibidos se sitúan en la cola de red, donde deben superar la validación de firma, que certifica la integridad de los datos del mensaje y la autenticación (y no repudio) de su remitente. En caso de superar estas comprobaciones, los mensajes se trasladan a la cola de validación, mientras que si no las superan son descartados directamente (puesto que, si la firma del mensaje no es válida, no servirá como prueba ante otras réplicas).

Los mensajes de la cola de validación deben obtener un resultado positivo ante la función de validación semántica cuyo pseudocódigo se presenta en el algoritmo 1. Si el mensaje no es una acusación, la comprobación de validez semántica se delega a la validación semántica externa, específica del sistema en el que se haya integrado el algoritmo. En caso de que sí se trate de un mensaje de acusación (o, posiblemente, varios mensajes de acusación anidados), la función de validación se ejecuta de forma recursiva hasta alcanzar el mensaje base.

Algoritmo 1 Función de validación semántica

```
function ValidaciónSemántica(m: Mensaje)
    if m.tipo = Acusación then
        comprobaciónAnidada ← ValidaciónSemántica(m.msj)        ▷ Llamada recursiva
        resultadoValidación ← ¬ comprobaciónAnidada
    else
        resultadoValidación ← ValidaciónSemánticaExterna(m.msj)
    end if
    return resultadoValidación
end function
```

Si un mensaje que no consiste en una acusación supera la validación semántica, este mensaje es propagado a la cola de entrega. Si un mensaje de acusación supera la validación semántica, indicando por lo tanto que el mensaje encapsulado en la acusación no la supera, el nodo remitente del mensaje original es añadido a la lista de nodos sospechosos. En caso de que se haya emitido una acusación referente a un mensaje que sí supera la validación semántica, se considera que el mensaje de acusación no la supera y podrá ser utilizado como prueba de comportamiento bizantino ante las demás réplicas. Así, ante la recepción de un mensaje que no supere la función de validación semántica, se añade el remitente del mensaje

a la lista local de nodos sospechosos y el mensaje recibido se encapsula en un mensaje de acusación que se envía a las demás réplicas.

Los mensajes depositados en la cola de entrega, que actúa como una salida del algoritmo, son delegados al protocolo del nivel superior, mientras que los mensajes de acusación generados son enviados a sus destinatarios (las restantes réplicas) a través de las correspondientes colas de salida del nivel de red.

Este algoritmo está inspirado en el algoritmo presentado en [CT96] que permite transformar un detector de fallos con completitud débil en un detector de fallos con completitud estricta, mediante la propagación a las demás réplicas de las listas locales de nodos sospechosos. El algoritmo especificado en la publicación referida está dirigido a sistemas que operan bajo el modelo de fallos de parada. En este caso se opera en un entorno bizantino autenticado, por lo que al algoritmo original se incorporan los niveles de validación de autorización y semántica descritos. Igualmente, en el algoritmo original las acusaciones de las demás réplicas no se validan, lo que podría ser aprovechado por una réplica bizantina para lanzar acusaciones falsas contra las réplicas honestas, comprometiendo el progreso del algoritmo de consenso. Por ello, en este caso el mensaje de acusación encapsula al mensaje denunciado, convenientemente firmado por su remitente, de tal forma que las demás réplicas puedan certificar la legitimidad de la acusación y, ante una acusación infundada, tomar las medidas pertinentes y crear un nuevo mensaje de acusación referente a la acusación recibida.

A continuación se presenta la especificación del algoritmo de detección activa de fallos de Trebizond, modelado mediante autómatas de entrada/salida.

Nomenclatura:

Nombre de variable: *tipo de variable* $\leftarrow$ *valor inicial*
$\delta \rightarrow$ *digest*
$\varsigma \rightarrow$ *firma*

Tipos de datos:

`Acusación: m: mensaje, origen: int`

Variables de estado:

`id`: `int` $\leftarrow$ identificador de la réplica
`replicaIds`: `Lista<int>` $\leftarrow$ identificadores de las restantes réplicas
`n`: `int` $\leftarrow$ número de réplicas
`qRed`$_{id,j} \forall j \in$ *replicaIds*: `Cola<mensaje>` $\leftarrow []$
`qRed`$_{j,id} \forall j \in$ *replicaIds*: `Cola<mensaje>` $\leftarrow []$
`qEntrega`$_{j,id} \forall j \in$ *replicaIds*: `Cola<mensaje>` $\leftarrow []$
`qValidacion`$_{j,id} \forall j \in$ *replicaIds*: `Cola<mensaje>` $\leftarrow []$
`nodosSospechosos`: `Lista<int>` $\leftarrow []$

Condiciones de seguridad:

1. Todo envío de un mensaje al nivel de aplicación se produce como consecuencia de la recepción previa de un mensaje en el nivel de red que hubiera superado las validaciones de firma y semántica.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot m \cdot \alpha_2 \rightarrow (\alpha_1 = \alpha_{1,1} \cdot \varsigma(m) \cdot \alpha_{1,2}) \wedge$
   $\wedge$ `superaValidaciónFirma(`$\varsigma(m)$`)` $\wedge$ `superaValidaciónSemántica(`m`)`
2. Todo envío de un mensaje de acusación será consecuencia de la recepción previa de un mensaje de acusación o de oro tipo, que no haya superado la validación semántica.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot \varsigma($`Acusación(`$\varsigma(m)$`)`$) \cdot \alpha_2 \rightarrow ((\alpha_1 = \alpha_{1,1} \cdot \varsigma(m) \cdot \alpha_{1,2}) \vee$
   $\vee ((\alpha_1 = \alpha_{1,1} \cdot \varsigma($`Acusación(`$\varsigma(m')$`)`$) \cdot \alpha_{1,2}) \wedge (m =$`Acusación(`$\varsigma(m')$`)`$))) \wedge$
   $\wedge \neg$`superaValidaciónSemántica(`m`)`
3. La recepción de un mensaje de acusación nunca supondrá el envío de un mensaje al nivel superior de aplicación.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot \varsigma($`Acusación(`$\varsigma(m)$`)`$) \cdot \alpha_2 \rightarrow m \notin \alpha_2$
4. Un mensaje que no supere las validaciones de firma digital (autenticación del remitente e integridad de los datos) y/o la validación semántica nunca será enviado al nivel superior de aplicación.
   $\forall \alpha \,|\, (\alpha = \alpha_1 \cdot \varsigma(m) \cdot \alpha_2) \wedge$
   $\wedge (\neg$`superaValidaciónFirma(`$\varsigma(m)$`)` $\vee \neg$`superaValidaciónSemántica(`m`)`$) \rightarrow m \notin \alpha_2$

Condiciones de viveza:

1. Ante la recepción de un mensaje que no sea de acusación y supere las validaciones de firma, eventualmente dicho mensaje se enviará al nivel superior de aplicación o se encapsulará en un mensaje de acusación que será enviado a través de la red.
   $\forall \alpha \,|\, (\alpha = \alpha_1 \cdot \varsigma(m) \cdot \alpha_2) \wedge (m \neq$ `Acusación`$) \wedge$ `superaValidaciónFirma(`$\varsigma(m)$`)` $\rightarrow$
   $\rightarrow (\alpha_2 = \alpha_{2,1} \cdot m \cdot \alpha_{2,2}) \vee (\alpha_2 = \alpha_{2,1} \cdot \varsigma($`Acusación(`$\varsigma(m)$`)`$) \cdot \alpha_{2,2})$
2. Un mensaje que no supere la validación semántica tendrá como consecuencia el envío de un mensaje de acusación referido al mensaje originalmente recibido.
   $\forall \alpha \,|\, (\alpha = \alpha_1 \cdot \varsigma(m) \cdot \alpha_2) \wedge \neg$`superaValidaciónSemántica(`$m$`)` $\rightarrow$
   $\rightarrow \alpha_2 = \alpha_{2,1} \cdot \varsigma($`Acusación(`$\varsigma(m)$`)`$) \cdot \alpha_{2,2}$

Entradas:

Mensaje firmado entrante `m` procedente de la réplica j: $(m, \varsigma(\delta(m')))_{j,id}$

Salidas:

Entrega al nivel superior del sistema local id de un mensaje `m` procedente de la réplica j que ha superado las validaciones: $m_{j,id}$
Envío de un mensaje firmado `m` a la réplica j a través de la red: $(m, \varsigma(\delta(m')))_{id,j}$

Operaciones:

(E) Mensaje firmado entrante, procedente de la réplica $j \rightarrow (m, \varsigma(\delta(m')))$

acciones:
$qRed_{j,id} \leftarrow qRed_{j,id} \cdot (m, \varsigma(\delta(m')))$

(S) Se entrega al nivel superior del sistema local un mensaje procedente de la réplica j que ha superado las validaciones $\rightarrow$ `m`

precondiciones:
$qEntrega_{j,id} = m \cdot qEntrega'_{j,id}$

acciones:
$qEntrega_{j,id} \leftarrow qEntrega'_{j,id}$

(S) Envío de un mensaje firmado a la réplica j a través de la red $\rightarrow (m, \varsigma(\delta(m')))$

precondiciones:
$qRed_{id,j} = (m, \varsigma(\delta(m'))) \cdot qRed'_{id,j}$

acciones:
$qRed_{id,j} \leftarrow qRed'_{id,j}$

(I) El nodo recibe un mensaje y comprueba que su creador es efectivamente la réplica remitente y que se ha mantenido la integridad de los datos. El mensaje se propaga a la cola de validación.

precondiciones:
$qRed_{id,j} = (m, \varsigma(\delta(m'))) \cdot qRed'_{j,id}$
`m.origen` = j
$\delta(m) = \delta(m')$
`firmaValida`($\varsigma(\delta(m'))$, j)

acciones:
$qValidacion \leftarrow qValidacion \cdot (m, \varsigma(\delta(m')))$
$qRed_{j,id} \leftarrow qRed'_{j,id}$

(I) El nodo recibe un mensaje, pero la firma no es correcta o no se ha mantenido la integridad de los datos. Se descarta el mensaje.

precondiciones:
$qRed_{j,id} = (m, \varsigma(\delta(m')) \cdot qRed'_{j,id}$
(`m.origen` $\neq j$) $\vee$ ($\delta(m) \neq \delta(m')$) $\vee \neg$ `firmaValida`($\varsigma(\delta(m'))$, j)

acciones:
$qRed_{j,id} \leftarrow qRed'_{j,id}$

(I) Se recibe un mensaje de otra réplica que, habiendo superado previamente las validaciones de firma e integridad, supera el proceso de validación semántica. Se activa la salida de entrega al nivel superior de la aplicación.

precondiciones:
$qValidacion_{j,id} = (m, \varsigma(\delta(m))) \cdot qValidacion'_{j,id}$
`m.tipo` $\neq$ `Acusación`
`superaValidaciónSemántica(m)`

acciones:
$qEntrega_{j,id} \leftarrow qEntrega_{j,id} \cdot m$
$qValidacion_{j,id} \leftarrow qValidacion'_{j,id}$

(I) Se recibe un mensaje de acusación de otra réplica que, habiendo superado previamente las validaciones de firma e integridad, supera también el proceso de validación semántica, lo que certifica la legitimidad de la acusación. Se añade a la lista de nodos sospechosos a la réplica que creó el mensaje encapsulado en el mensaje de acusación.

precondiciones:
$qValidacion_{j,id} = (m, \varsigma(\delta(m))) \cdot qValidacion'_{j,id}$
`m.tipo` $=$ `Acusación`
`superaValidaciónSemántica(m)`

acciones:
`nodosSospechosos` $\leftarrow$ (`nodosSospechosos` $\cup$ `m.m.origen`) $- id$
$qValidacion_{j,id} \leftarrow qValidacion'_{j,id}$

(I) Se recibe un mensaje de otra réplica (ya sea una acusación o cualquier otro tipo de mensaje) que, habiendo superado previamente las validaciones de firma e integridad, no supera el proceso de validación semántica. Se añade a la réplica remitente a la lista de sospechosos, se encapsula el mensaje recibido en un mensaje de acusación (como evidencia demostrable del comportamiento bizantino, dado que el mensaje recibido se encuentra firmado por su emisor) y se envía a las demás réplicas.

precondiciones:
$qValidacion_{j,id} = (m, \varsigma(\delta(m))) \cdot qValidacion'_{j,id}$
$\neg$ `superaValidaciónSemántica(m)`

acciones:
`nodosSospechosos` $\leftarrow$ (`nodosSospechosos` $\cup\, j) - id$
$\forall k \in$ `replicaIds` $qRed_{i,k} \leftarrow qRed'_{i,k} \cdot$ `Acusación(`$m, \varsigma(\delta(m))$`)`
$qValidacion_{j,id} \leftarrow qValidacion'_{j,id}$

Argumentación de las condiciones de seguridad:

1. La salida de entrega de un mensaje al nivel de aplicación sólo se activa si se recibe un mensaje en la cola de validación que supera las comprobaciones de validación semántica. A su vez, sólo se inserta un mensaje en la cola de validación si éste se recibe en la cola de red y supera las validaciones de firma, certificándose su autenticación y la integridad de los datos.
2. La salida de envío de un mensaje de acusación sólo se activa si se recibe un mensaje en la cola de validación que no supera las comprobaciones se validación semántica. A su vez, igual que se señala en la primera condición de seguridad, sólo se inserta un mensaje en la cola de validación si éste se recibe en la cola de red y supera las validaciones de firma, certificándose su autenticación y la integridad de los datos.
3. La salida de entrega de un mensaje al nivel de aplicación sólo se activa si se recibe un mensaje en la cola de validación que supere las comprobaciones de validación semántica y cuyo tipo no sea el de un mensaje de acusación. En caso de que el mensaje sí fuera una acusación (superando la validación semántica), no se entregará ningún mensaje al nivel de aplicación, realizándose en cambio una inserción en la lista de nodos sospechosos.
4. Como se argumenta en la primera condición de seguridad, sólo se inserta un mensaje en la cola de validación si éste supera las validaciones de firma, certificándose su autenticación y la integridad de los datos. Continuando la argumentación anterior, e igualmente como se argumenta en la primera condición de seguridad, la salida de entrega de un mensaje al nivel de aplicación sólo se activa si se recibe un mensaje en la cola de validación que supere las comprobaciones de validación semántica.

Argumentación de las condiciones de viveza:

1. Si se recibe un mensaje en la cola de red que supere las validaciones de firma (autenticación e integridad de los datos), éste se trasladará a la cola de validación. Una vez en la cola de validación, si el mensaje supera la validación semántica se trasladará al nivel de aplicación, mientras que en caso de no superar la validación se encapsulará en un mensaje de acusación que será enviado a las demás réplicas.
2. Se parte de la premisa de que el mensaje referido por la condición supera las validaciones de firma, y por tanto se deposita en la cola de validación. Tras recoger este mensaje de la cola y comprobar que no supera la validación semántica, el mensaje se encapsulará en un mensaje de acusación (incluso aunque el mensaje recibido a su vez fuera un mensaje de acusación procedente de otra réplica, puesto que la función de validación semántica es capaz de resolver el anidamiento de acusaciones) que se enviará a las réplicas restantes.

3.2.3 Algoritmo de difusión atómica

Este algoritmo es el que permite a las réplicas ponerse de acuerdo sobre un determinado valor propuesto por una de ellas, que se corresponde con una operación cuyos efectos se aplicarán sobre el estado compartido por las réplicas en un orden especifico y común a todas ellas. Tal como expone [HT94], el concepto de consenso es equivalente a una sucesión de difusiones atómicas.

El algoritmo está basado en el algoritmo de difusión fiable bizantina presentado en [Bra87], adaptándolo para su integración con el resto de los algoritmos que, mediante su composición, forman Trebizond. La adaptación realizada del algoritmo de difusión se describe en el pseudocódigo del algoritmo 2 y su modelo de funcionamiento responde a la siguiente descripción.

Algoritmo 2 Difusión fiable bizantina

function DIFUSIÓNFIABLEBIZANTINA(m: Mensaje)

Etapa 0: ▷ ejecutada por la réplica que inicia la difusión
Enviar un mensaje *Inicio(m)* a todas las réplicas

Etapa 1: Esperar hasta la recepción de
un mensaje *inicial(m)*
o $\frac{n+f}{2}$ mensajes *Eco(m)*
o $f + 1$ mensajes *Preparado(m)*
para algún mensaje *m*
Enviar un mensaje *Eco(m)* a todas las réplicas

Etapa 2: Esperar hasta la recepción de
$\frac{n+f}{2}$ mensajes *Eco(m)*
o $f + 1$ mensajes *Preparado(m)*
▷ (incluyendo los mensajes recibidos en la etapa 1)
para algún mensaje *m*
Enviar un mensaje *Preparado(m)* a todas las réplicas

Etapa 3: Esperar hasta la recepción de
$2f + 1$ mensajes *Preparado(m)*
▷ (incluyendo los mensajes recibidos en las etapas 1 y 2)
para algún mensaje *m*
Aceptar *m* para su ejecución
end function

El algoritmo hace uso de tres tipos de mensajes: `Inicio`, `Eco` y `Preparado`. Un mensaje $\texttt{Inicio}_p(v)$ implica que la réplica p desea difundir el valor v. Un mensaje `Eco`(v) implica que su emisor es consciente de que la réplica p está difundiendo el valor v porque ha recibido un mensaje `Inicio`(v) de la réplica p o suficientes mensajes `Eco`(v) o `Preparado`(v) que confirman dicha difusión. Un mensaje `Preparado`(v) implica que su emisor es consciente de que v es el único valor enviado en esta difusión por la réplica p y que se encuentra preparado para aceptar v porque ha recibido suficientes mensajes, bien de tipo `Eco`(v) o bien de tipo `Preparado`(v). Cuando un proceso recibe suficientes mensajes `Preparado`(v), acepta v como el valor enviado por la réplica p, con la certeza de que todas las otras réplicas correctas también llegarán a aceptar dicho valor.

El algoritmo se divide en etapas que se corresponden con los tipos de mensaje. En cada etapa la réplica espera hasta haber recibido una cantidad suficiente de mensajes que le permita enviar el siguiente tipo de mensaje (incluyendo los recibidos en etapas anteriores), envía el mensaje a todas las otras réplicas y transita a la siguiente etapa. Así, una réplica correcta envía un mensaje de cada tipo (un mensaje por cada etapa) a todas las otras réplicas correctas. La difusión se completará con éxito cuando todas las réplicas *acepten* el valor v, con lo que el efecto de la ejecución de la operación contenida en el mensaje llegará a aplicarse sobre todas las réplicas del estado compartido.

Tal como enuncia [Bra87], un algoritmo constituye un protocolo fiable de difusión si cumple las siguientes dos propiedades, en este caso en un entorno bizantino autenticado:

1. Si la réplica p es correcta, entonces todas las réplicas correctas coinciden en el valor del mensaje que ha enviado.
2. Si la réplica p ha sido comprometida, entonces o bien todas las réplicas correctas coinciden en el mismo valor o bien ninguna acepta valor alguno de la réplica p.

La argumentación de las condiciones de seguridad y viveza que se establecen sobre el algoritmo lleva a concluir que éste cumple ambas propiedades, pudiéndose afirmar que el algoritmo especificado constituye un algoritmo de difusión fiable bizantina.

A continuación se presenta la especificación del algoritmo de difusión atómica de Trebizond, modelado mediante autómatas de entrada/salida.

Nomenclatura:

Nombre de variable: *tipo de variable* $\leftarrow$ *Valor inicial*
$\delta \rightarrow$ *digest*
$\varsigma \rightarrow$ *firma*

Tipos de datos:

`Operación: op: int, uuid: string, timestamp, origen: int, difusión: boolean`

`Resultado: opId: int, result`

`RespuestaIndividual: resultado: Resultado`

`RespuestaColectiva: resultado: Resultado, confirmaciones: ς(δ(Resultado))[]`

`Inicio: op: Operación, estadoActual:` $\delta($`estado`$)$

`Eco: op: Operación, estadoActual:` $\delta($`estado`$)$

`Preparado: op: Operación, estadoActual:` $\delta($`estado`$)$

Variables de estado:

`id: int` ← identificador de la réplica

`replicaIds: Lista<int>` ← identificadores de las restantes réplicas

`n: int` ← número de réplicas

`f: int` ← $\lceil \frac{n-1}{3} \rceil$ (número máximo de réplicas comprometidas)

$\texttt{q}_{id,j} \forall j \in$ *replicaIds*: `Cola<mensaje>` ← $[]$

$\texttt{q}_{j,id} \forall j \in$ *replicaIds*: `Cola<mensaje>` ← $[]$

`ultimaOpCliente`$_c$`{uuid, resultado}` $\forall c \in clientes \leftarrow \varnothing$

`logMensajes` ← `[]`

Condiciones de seguridad:

1. Si dos réplicas correctas *s* y *t* envían los mensajes `Preparado(v)` y `Preparado(u)`, respectivamente, entonces $u = v$.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot \texttt{Preparado}_s(v) \cdot \alpha_2 \cdot \texttt{Preparado}_t(u) \cdot \alpha_3 \rightarrow u = v$
2. Si dos réplicas correctas *q* y *r* aceptan los valores *v* y *u*, respectivamente, entonces $u = v$.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot \texttt{Aceptar}_q(v) \cdot \alpha_2 \cdot \texttt{Aceptar}_r(u) \cdot \alpha_3 \rightarrow u = v$

Condiciones de viveza:

1. Si una réplica correcta *q* acepta el valor *v*, entonces toda otra réplica correcta eventualmente aceptará dicho valor *v*.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot \texttt{Aceptar}_q(v) \cdot \alpha_2 \rightarrow \forall p \in replicasCorrectas \,;\, p \neq q \rightarrow$
   $\rightarrow (\ \texttt{Aceptar}_p(v)) \in \alpha_1 \wedge \texttt{Aceptar}_p(v) \notin \alpha_2) \vee$
   $\vee(\texttt{Aceptar}_p(v) \notin \alpha_1 \wedge \texttt{Aceptar}_p(v) \in \alpha_2)$
2. Si una réplica correcta *p* difunde el valor *v*, entonces todas las réplicas correctas llegarán a aceptar *v*.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot \texttt{Inicio}_p(v) \cdot \alpha_2 \rightarrow \forall q \in replicasCorrectas\ \texttt{Aceptar}_q(v) \in \alpha_2$

Entradas:

Recepción de solicitud de operación procedente de un cliente *c*: `Operación`
Recepción de un mensaje `Inicio` de otra réplica *j*, habiendo superado las validaciones del algoritmo de detección activa de fallos: `Inicio`
Recepción de un mensaje `Eco` de otra réplica *j*, habiendo superado las validaciones del algoritmo de detección activa de fallos: `Eco`
Recepción de un mensaje `Preparado` de otra réplica *j*, habiendo superado las validaciones del algoritmo de detección activa de fallos: `Preparado`
Recepción de un resultado individual procedente de un nodo *j* para una operación: `RespuestaIndividual`

Salidas:

Envío de un mensaje `Inicio` a otra réplica *j*: `Inicio`
Envío de un mensaje `Eco` a otra réplica *j*: `Eco`
Envío de un mensaje `Preparado` a otra réplica *j*: `Preparado`
Envío de un mensaje de resultado individual a otro nodo *j*: `RespuestaIndividual`
Envío de un mensaje de respuesta individual al cliente *c*: `RespuestaIndividual`
Envío de un mensaje de respuesta colectiva al cliente *c*: `RespuestaColectiva`

Operaciones:

(E) Recepción de solicitud de operación procedente de un cliente $c \rightarrow$ `{op, uuid, ts}`

acciones:

$q_{c,id} \leftarrow q_{c,id} \cdot$ `op`

(I) Solicitud de operación cliente recibida; no supera las validaciones, se ignora.

precondiciones:

$q_{c,id} = (op, \varsigma(\delta(op'))) \cdot q'_{c,id}$
$((\delta(op) = \delta(op')) \vee$
$\vee$ `superaValidacionFirma(`$\varsigma(\delta(op')), c$`)` $\vee$
$\vee$ `superaValidaciónSemántica(`op`)` $\vee$
$\vee$ (`op.origen` $\neq c$))

acciones:

$q_{c,id} \leftarrow q'_{c,id}$

(I) Solicitud de operación cliente recibida, supera las validaciones de firma. El identificador coincide con el de la última operación del cliente aceptada, se devuelve el resultado almacenado.

precondiciones:
$q_{c,id} = (op, \varsigma(\delta(op'))) \cdot q'_{c,id}$
`op.remitente` $\cdot = c$
$\delta(op) = \delta(op')$
`superaValidaciónFirma`$(\varsigma(\delta(op')), c)$
`op.uuid = ultimaOpCliente`$_c$`.op.uuid`

acciones:
$q_{c,id} \leftarrow q'_{c,id}$
$q_{id,c} \leftarrow q_{id,c}\cdot$ `ultimaOpCliente`$_c \cdot$ `result`

(I) Se recibe una nueva operación cliente de sólo lectura, se ejecuta y se devuelve el resultado inmediatamente.

precondiciones:
$q_{c,id} = (op, \varsigma(\delta(op'))) \cdot q'_{c,id}$ `op.origen =` c
$\delta(op) = \delta(op')$
`superaValidaciónFirma`$(\varsigma(\delta(op')), c$
`superaValidaciónSemántica`(op)
`esOperaciónSóloLectura`(op)

acciones:
$q_{c,i} \leftarrow q'_{c,id}$
$q'_{id,c} \leftarrow q_{id,c}\cdot$ `RespuestaIndividual(op.uuid, ejecutarOpLectura(op))`

(I) Se recibe una solicitud de operación cliente. Nueva operación de escritura.

precondiciones:
$q_{c,id} = (op, \varsigma(\delta(op'))) \cdot q'_{c,id}$
`op.origen`$= c$
$\delta(op) = \delta(op')$
`superaValidacionFirma`$(\varsigma(\delta(op')), c)$
$\neg$`esOperaciónSóloLectura`(op)
`op.uuid` $\neq$ `ultimaOpCliente`$_c$`.op.uuid`

acciones:
$q_{c,id} \leftarrow q'c, id$
$\forall j \in replicas; q_{id,j} \leftarrow q_{id,j} \cdot$ `Inicio`$((op, \varsigma(\delta(op'))))$

(E) Recepción de un mensaje `Inicio` de otra réplica *j*, habiendo superado las validaciones del algoritmo de detección activa de fallos $\rightarrow$ `Inicio(op,` δ`(estado))`

acciones:

$q_{j,id} \leftarrow q_{j,id} \cdot$ `Inicio(op,` δ`(estado))`

(E) Recepción de un mensaje `Eco` de otra réplica *j*, habiendo superado las validaciones del algoritmo de detección activa de fallos $\rightarrow$ `Eco(op,` δ`(estado))`

acciones:

$q_{j,id} \leftarrow q_{j,id} \cdot$ `Eco(op,` δ`(estado))`

(E) Recepción de un mensaje `Preparado` de otra réplica *j*, habiendo superado las validaciones del algoritmo de detección activa de fallos $\rightarrow$ `Preparado(op,` δ`(estado))`

acciones:

$q_{j,id} \leftarrow q_{j,id} \cdot$ `Preparado(op,` δ`(estado))`

(I) Se recibe un mensaje `Inicio`. Se almacena si no existía ya un mensaje `Inicio` para la misma operación y la misma réplica.

precondiciones:

$q_{j,id} =$ `Inicio(op,` δ`(estado))` $\cdot\ q'_{j,id}$

acciones:

`logMensajes[op.uuid].inicio` $\leftarrow$ `logMensajes[op.uuid].inicio` $\cup\ j$

$q_{j,id} \leftarrow q'_{j,id}$

(I) Se recibe un mensaje `Eco`. Se almacena si no existía ya un mensaje `Eco` para la misma operación y la misma réplica.

precondiciones:

$q_{j,id} =$ `Eco(op,` δ`(estado))` $\cdot\ q'_{j,id}$

acciones:

`logMensajes[op.uuid].eco` $\leftarrow$ `logMensajes[op.uuid].eco` $\cup\ j$

$q_{j,id} \leftarrow q'_{j,id}$

(I) Se recibe un mensaje `Preparado`. Se almacena si no existía ya un mensaje `Preparado` para la misma operación y la misma réplica.

precondiciones:

$q_{j,id} =$ `Preparado(op,` $\delta($`estado`$)$`)` $\cdot\ q'_{j,id}$

acciones:

`logMensajes[op.uuid].preparado` $\leftarrow$ `logMensajes[op.uuid].preparado` $\cup\ j$

$q_{j,id} \leftarrow q'_{j,id}$

(I) Se alcanzan las condiciones de la etapa 1. Se envían mensajes `Eco` a todas las réplicas.

precondiciones:

$\exists\, op \mid$ `(|logMensajes[op.uuid].inicio|` $\geqslant 1\ \vee$

$\vee$ `|logMensajes[op.uuid].eco|` $\geqslant \frac{n+f}{2}\ \vee$

$\vee$ `|logMensajes[op.uuid].preparado|` $\geqslant f+1$`)` $\wedge\ id \notin$ `logMensajes[op.uuid].eco`

acciones:

$\forall j \in replicas\,;\ q_{id,j} \leftarrow q_{id,j}$ $\cdot$ `Eco(op)`

`logMensajes[op.uuid].eco` $\leftarrow$ `logMensajes[op.uuid].eco` $\cup\ id$

(I) Se alcanzan las condiciones de la etapa 2. Se envían mensajes `Preparado` a todas las réplicas.

precondiciones:

$\exists\, op \mid$ `(|logMensajes[op.uuid].eco|` $\geqslant \frac{n+f}{2}\ \vee$

$\vee$ `|logMensajes[op.uuid].preparado|` $\geqslant f+1$`)` $\wedge\ id \notin$ `logMensajes[op.uuid].preparado`

acciones:

$\forall j \in replicas\,;\ q_{id,j} \leftarrow q_{id,j}$ $\cdot$ `Preparado(op)`

`logMensajes[op.uuid].preparado` $\leftarrow$ `logMensajes[op.uuid].preparado` $\cup\ id$

(I) Se alcanzan las condiciones de la etapa 3, con lo que se puede aceptar una operación. Ésta se recibió de forma individual.

precondiciones:

$\exists\, op \mid$ `|logMensajes[op.uuid].preparado|` $\geqslant 2f+1 \ \wedge$

$\wedge$ `¬logMensajes[op.uuid].aceptado` $\wedge$ `¬op.difusión`

acciones:

`estado` $\leftarrow$ `ejecutarOperación(op)`

`logMensajes[op.uuid].resultado` $\leftarrow$ `estado`

`logMensajes[op.uuid].aceptado` $\leftarrow$ `true`

$\texttt{ultimaOpCliente}_{op.origen} \leftarrow$ `(op.uuid, estado)`

$q_{id,liderActual} \leftarrow q_{id,liderActual} \cdot$ `RespuestaIndividual(op.uuid,` $\delta($`estado`$)$`)`

(I) Se alcanzan las condiciones de la etapa 3, con lo que se puede aceptar una operación. Ésta se recibió por medio de una difusión a las réplicas (colectivamente).

precondiciones:

$\exists\, op \mid$ `|logMensajes[op.uuid].preparado|` $\geqslant 2f+1 \ \wedge$

$\wedge$ `¬logMensajes[op.uuid].aceptado` $\wedge$ `op.difusión`

acciones:

`estado` $\leftarrow$ `ejecutarOperación(op)`

`logMensajes[op.uuid].resultado` $\leftarrow$ `estado`

`logMensajes[op.uuid].aceptado` $\leftarrow$ `true`

$\texttt{ultimaOpCliente}_{op.origen} \leftarrow$ `(op.uuid, estado)`

$q_{id,op.origen} \leftarrow q_{id,op.origen} \cdot$ `RespuestaIndividual(op.uuid, estado)`

(S) Envío de un mensaje `Inicio` a otra réplica $j \rightarrow$ `Inicio(op)`

precondiciones:

$q_{id,j} =$ `Inicio(op)` $\cdot\, q'_{id,j}$

acciones:

$q_{id,j} \leftarrow q'_{id,j}$

(S) Envío de un mensaje `Eco` a otra réplica $j \rightarrow$ `Eco(op)`

precondiciones:

$q_{id,j} =$ `Eco(op)` $\cdot\, q'_{id,j}$

acciones:

$q_{id,j} \leftarrow q'_{id,j}$

(S) Envío de un mensaje `Preparado` a otra réplica *j* $\rightarrow$ `Preparado(op)`

precondiciones:

$q_{id,j} =$ `Preparado(op)` $\cdot\ q'_{id,j}$

acciones:

$q_{id,j} \leftarrow q'_{id,j}$

(E) Recepción del resultado individual de otra réplica *j* para una operación con identificador `opId` $\rightarrow$ `RespuestaIndividual(opId,` δ`(estado))`

acciones:

$q_{j,id} \leftarrow$ `RespuestaIndividual(opId,` δ`(estado))`

(I) Un nodo recibe un resultado individual de otro nodo. Almacena dicho resultado.

precondiciones:

$q_{j,id} = \varsigma($ `RespuestaIndividual(opId,` δ`(resultado))` $\cdot\ q'_{j,id}$

acciones:

$q_{j,id} \leftarrow q'_{j,id}$

`logMensajes[opId].respuestas[`j`]` $\leftarrow \varsigma($`RespuestaIndividual(opId,` δ`(resultado)))`

(I) Un nodo reúne suficientes respuestas individuales para una operación que ha aceptado. Se compone una respuesta colectiva y se envía al cliente.

precondiciones:

$\exists$ `(opId,` δ`(resultado)` $|$

`|RespuestaIndividual(opId,` δ`(resultado))|` $\in$ `logMensajes[opId].respuestas` $\geqslant \frac{2n+1}{3}$

`logMensajes[opId].respuestas[`id`]` $\neq$ `RespuestaColectiva`

acciones:

`logMensajes[opId].respuestas[`id`]` $\leftarrow$

$\leftarrow$ `RespuestaColectiva(opId, logMensajes[opId].resultado,`

$\varsigma($`RespuestaIndividual(opId,` δ`(resultado)))` $|$

δ`(resultado)` $= \delta$`(logMensajes[opId].resultado)))`

$q_{id,logMensajes[opId].origen} \leftarrow q_{id,logMensajes[opId].origen}$ $\cdot$ `logMensajes[opId].respuestas[`id`]`

(S) Envío de un mensaje de respuesta individual a otra réplica $j \rightarrow$ `RespuestaIndividual(opId,` $\delta($`estado`$)$`)`

precondiciones:

$q_{id,j} =$ `RespuestaIndividual(opId,` $\delta($`estado`$)$`)` $\cdot\ q'_{id,j}$

acciones:

$q_{id,j} \leftarrow q'_{id,j}$

(S) Envío de un mensaje de respuesta colectiva al cliente $c \rightarrow$ `RespuestaColectiva(opId, resultado,` $\varsigma($`RespuestaIndividual(opId,` $\delta($`resultado`$)$`)`$)$`[])`

precondiciones:

$q_{id,j} =$ `RespuestaColectiva(opId, resultado,` $\varsigma($`RespuestaIndividual(opId,` $\delta($`resultado`$)$`)`$)$`[])`

acciones:

$q_{id,j} \leftarrow q'_{id,j}$

Argumentación de las condiciones de seguridad:

1. Partiendo del caso contrario, se supone que una réplica *q* es la primera en enviar un mensaje `Preparado(v)`, y *r* es la primera réplica que envía un mensaje `Preparado(u)`. La réplica *q* debe haber recibido más de $\frac{n+f}{2}$ mensajes `Eco(v)`, y la réplica *r* debe haber recibido más de $\frac{n+f}{2}$ mensajes `Eco(u)`. Por lo tanto, alguna réplica correcta debe haber enviado ambos mensajes `Eco(u)` y `Eco(v)`. Puesto que las réplicas correctas sólo envían un mensaje de cada tipo durante una misma difusión, esto es una contradicción.
2. Si la réplica *q* acepta el valor *v*, entonces previamente debe haber recibido $2f + 1$ mensajes `Preparado(v)`, habiendo recibido por lo tanto al menos $f + 1$ mensajes `Preparado(v)` procedentes de réplicas correctas. De forma similar, la réplica *r* debe haber recibido al menos $f + 1$ mensajes `Preparado(u)` procedentes de réplicas correctas. De acuerdo con la primera condición de seguridad, en tales circunstancias se puede concluir que $u = v$.

Argumentación de las condiciones de viveza:

1. Si la réplica correcta *q* acepta el valor *v*, entonces *q* habrá recibido $2f + 1$ mensajes `Preparado(v)` con anterioridad, de los cuales al menos $f+1$ fueron enviado por réplicas correctas. Por lo tanto, todas esas réplicas habrán recibido al menos $f + 1$ mensajes `Preparado(v)` y enviado su propio mensaje `Preparado(v)`. De acuerdo con la primera condición de seguridad, es imposible que en la misma difusión una réplica correcta envíe un mensaje `Preparado` referente a un valor diferente. Así, al menos $n - f$ réplicas llegan a enviar un mensaje `Preparado(v)`. Toda réplica correcta *r* recibe eventualmente (de acuerdo con las condiciones de sincronía) al menos $2f + 1 \geqslant n - f$ mensajes `Preparado(v)` y acepta el valor *v*.
2. Partiendo de que la réplica correcta *p* difunde el valor *v*, toda réplica correcta *q* recibe un mensaje `Inicio(v)` y responde enviando un mensaje `Eco(v)`. Toda réplica correcta *q* recibirá $(n - f > \frac{n+f}{2})$ mensajes `Eco(v)` de las réplicas correctas, y posiblemente $f < \frac{n+f}{2}$ mensajes diferentes de las réplicas comprometidas. Por lo tanto, la réplica *q* enviará un mensaje `Preparado(v)`. En el tercer paso del algoritmo, toda réplica correcta *q* recibirá $(n - f \geqslant 2f + 1)$ mensajes `Preparado(v)`, y posiblemente f mensajes `Preparado` diferentes procedentes de las réplicas comprometidas. Así, la réplica *q* aceptará el valor *v*.

Argumentación:
El algoritmo propuesto describe un protocolo de difusión fiable.

Si una réplica *p* difunde un mensaje con el valor *v*:

1. Si la réplica *p* es correcta, entonces, en base a la segunda condición de viveza, todas las réplica correctas aceptarán el valor *v*.
2. Si la réplica *p* es bizantina y alguna réplica correcta *q* acepta el valor *v*, en base a la primera condición de viveza todas las réplica correctas aceptarán dicho valor *v*. En caso contrario, ninguna réplica correcta aceptará valor alguno.

3.2.4 Algoritmo del cliente

El cliente es responsable de enviar peticiones de operación al servicio y recibir las respuestas correspondientes. Al operar bajo el modelo de fallos bizantinos con condiciones de sincronía eventual, el cliente debe recopilar un número suficiente de respuestas de las réplicas servidoras para asegurarse de que efectivamente se ha alcanzado consenso sobre la operación solicitada. Se distinguen dos tipos de operaciones, operaciones de sólo lectura y operaciones que implican escritura de datos, puesto que el servicio puede solventar con mayor facilidad las primeras y el cliente necesita la confirmación del resultado por parte de un número menor de réplicas. En todos los casos, el cliente espera a tener confirmación suficiente del consenso sobre una petición de operación emitida antes de emitir la siguiente petición de operación, por lo que las operaciones pendientes de emitir por el cliente se encolan. En el momento de enviarse al servicio, cada petición de operación incluye la propia operación, un identificador unívoco y una marca de tiempo. Todos los mensajes enviados por el cliente se encuentran firmados por éste, y al recibir un mensaje procedente de una réplica del servicio se comprueba que el nodo correspondiente es efectivamente el remitente mediante la comprobación de la firma del mensaje recibido; en caso de que no sea así, se descarta el mensaje.

Resumiendo el funcionamiento del cliente mediante una explicación textual y ciñéndose al caso general, su comportamiento sigue el siguiente esquema:

- Se envía una petición de operación al nodo que se piensa que actualmente es el líder de la red.
- Si se recibe una redirección de líder por parte de dicho nodo, se actualiza la variable local al nuevo líder.
- Si tras realizar el intento de emisión de petición individual se produce la expiración del temporizador de inactividad, se envía la petición de operación a todas las réplicas del servicio. Tras realizar una primera difusión, nuevas expiraciones del temporizador supondrán la realización de subsiguientes difusiones.
- Pueden recibirse dos tipos de respuestas: En primer lugar, puede recibirse un único mensaje de respuesta colectiva procedente del líder actual. Dicho mensaje contendrá el identificador de la petición (que debe corresponder con la última petición emitida), el resultado de la operación y un conjunto de *hashes* procedentes de un número de réplicas suficiente para proclamar consenso, que estarán firmados por sus respectivos remitentes y contendrán el identificador de la petición y el resultado. El cliente calculará el *hash* del identificador de la petición junto con el resultado de la operación y comprobará que coincide con los *hashes* recibidos de las diferentes réplicas, así como que todas las firmas son legítimas. En caso de superar esta validación se darán por válidas la respuesta y el resultado de la operación. El motivo de incluir los *hashes* en lugar de las respuestas completas en el mensaje es reducir el tamaño de dicho mensaje y la

consiguiente complejidad espacial del algoritmo. Por otra parte, en lugar de un único mensaje de respuesta colectiva puede recibirse una serie de mensajes de respuesta individual, conteniendo el identificador de la petición y el resultado de la operación, cada uno de ellos emitido por una única réplica. En este caso, el cliente deberá acumular un número suficiente de mensajes de respuesta, procedentes de distintos remitentes, cuyo resultado coincida, ante lo que podrá concluirse que se ha alcanzado consenso sobre la operación.

Este algoritmo está principalmente inspirado en el algoritmo cliente de PBFT, aplicando algunas optimizaciones sobre el mismo, como el tratamiento de operaciones de sólo lectura. El algoritmo cliente de PBFT realiza una petición de operación a una única réplica y, si su temporizador expira sin haber obtenido respuesta, realiza una difusión a todas las réplicas.

Este modelo ha sido adaptado, ampliándolo para que no sólo posea modos de envío individual y colectivo, sino para que estos dos modos también estén presentes en la recepción de las respuestas.

El elemento de redirección de líder es propio de Raft, en este caso se incorpora para actualizar la noción local de líder, pero no se produce un segundo envío individual ante la recepción de un mensaje de redirección de líder. Desde la perspectiva del cliente, el funcionamiento dual de este algoritmo (*unicast*/*broadcast*) deja intuir que en el algoritmo del servidor la figura del líder es más débil que en otros algoritmos similares, limitándose principalmente a ser un mero punto de centralización de mensajes de petición o respuesta cuyo objetivo es reducir el número de mensajes que se intercambian entre los clientes y las réplicas del servicio, y reducir el tamaño de las respuestas mediante la incorporación de *hashes* firmados en la respuesta colectiva. La posibilidad de difundir las peticiones a todas las réplicas o bien recibir respuestas individuales de cada una de éstas sirve al caso en el que se detecta que el líder no está funcionando como debería, por lo que se opta por evitar su intermediación en el envío de mensajes.

En caso de que un cliente detecte que el líder ha fallado, bien porque no responda a la petición o bien porque envíe mensajes incorrectos, optará por difundir la petición a todos los nodos en el primer caso y a descartar los mensajes incorrectos recibidos en el segundo. A diferencia de Tangaroa, un cliente no puede acusar a un nodo de haber fallado ante el resto de réplicas del servicio, puesto que un cliente bizantino podría hacer uso de esta capacidad para forzar al resto de los nodos a ejecutar acciones adicionales, como una nueva designación de líder, disminuyendo su productividad o incluso excluyendo a un nodo correcto del grupo de difusión. Sin embargo, una posible implantación de esta medida podría llevarse a cabo en un escenario en el que la acusación procedente del cliente pudiera respaldarse con pruebas contrastables y verificables.

A continuación se presenta la especificación del algoritmo cliente de Trebizond, modelado mediante autómatas de entrada/salida.

Nomenclatura:

nombre de variable: *tipo de variable* $\leftarrow$ *valor inicial*
$\delta \rightarrow$ *digest*
$\varsigma \rightarrow$ *firma*

Tipos de datos:

`Operacion`: `op`, `uuid`, `esSoloLectura`, `timestamp`
`RespuestaIndividual`: `uuid`, `result`
`RespuestaColectiva`: `uuid`, `result`, `digest`: $[\varsigma(\delta(\texttt{uuid, result}))]$ de $\frac{2n+1}{3}$ réplicas, cada uno firmado por su emisor

Variables de estado:

`n`: `int` $\leftarrow$ número de réplicas
$\texttt{q}_{i,c} \forall i \in$ *réplicas*: `Cola<mensaje>` $\leftarrow []$
$\texttt{q}_{c,i} \forall i \in$ *réplicas*: `Cola<mensaje>` $\leftarrow []$
`qOp`: `Cola<Operacion>` $\leftarrow []$
`opActual`: `Operacion` $\leftarrow \varnothing$
`liderActual`: `int` $\leftarrow$ `rand(n)`
`intentosUnicast`: `int` $\leftarrow 0$
`respuestasActuales`: `result[]` $\leftarrow []$
`resultado`: `result` $\leftarrow \varnothing$
`temporizadorExpirado`: `bool` $\leftarrow$ `falso`

Condiciones de seguridad:

1. Si se envía una petición de operación a alguna réplica del servicio, previamente el cliente habrá recibido una orden para dicha operación.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot request(op) \cdot \alpha_2 \rightarrow \alpha_1 = \alpha_{1,1} \cdot op \cdot \alpha_{1,2}$
2. El cliente únicamente emitirá salidas de resultado para aquellas órdenes de operación que haya recibido, pudiendo emitir un máximo de una salida de resultado para cada orden de operación reibida.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot resultado(op) \cdot \alpha_2 \rightarrow (\alpha_1 = \alpha_{1,1} \cdot op \cdot \alpha_{1,2}) \wedge (resultado(op) \notin \alpha_2)$
3. En caso de recibir una orden de operación que hubiera sido precedida por otra anterior, no se envía ninguna solicitud al servicio para la nueva operación hasta no haber emitido una salida de resultado relativa a la orden de operación precedente.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot op \cdot \alpha_2 \cdot op' \cdot \alpha_3 \cdot resultado(op') \cdot \alpha_4 \rightarrow \alpha_2 = \alpha_{2,1} \cdot resultado(op) \cdot \alpha_{2,2}$

Condiciones de viveza:

1. Toda orden de operación recibida termina siendo trasladada a las réplicas del servicio.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot op \cdot \alpha_2 \rightarrow \alpha_{2,1} \cdot request_{c,i}(op) \cdot \alpha_{2,2} \;\; ; \;\; i \in replicas$
2. Toda difusión de peticiones de operación termina recibiendo una respuesta colectiva o una cantidad suficiente de respuestas individuales, procedentes de diferentes réplicas.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot (\forall_{i \in replicas}\, request_{c,i}(op)) \cdot \alpha_2 \;\; \rightarrow$
   $\rightarrow \;\; (\alpha_2 = \alpha_{2,1} \cdot fullReply(op) \cdot \alpha_{2,2} \;\; ; \;\; i \in replicas) \;\; \vee$
   $\vee \;\; (\exists r \,|\, r \subset replicas \wedge |r| \geqslant \frac{2n+1}{3} \;\; ; \;\; \forall_{i \in r}\, reply_{i,c}(op) \in \alpha_2)$
   $\alpha 2 = \alpha_{2,1} \cdot \;\; \cdot reply_i(op)$
3. Toda difusión de peticiones de operación termina produciendo una salida de resultado.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot op \cdot \alpha_2 \rightarrow \alpha_2 = \alpha_{2,1} \cdot resultado(op) \cdot \alpha_{2,2}$
4. Toda orden de operación termina produciendo una salida de resultado.
   $\forall \alpha \,|\, \alpha = \alpha_1 \cdot (\forall_{i \in replicas}\, request_{c,i}(op)) \cdot \alpha_2 \rightarrow \alpha_2 = \alpha_{2,1} \cdot resultado(op) \cdot \alpha_{2,2}$

Entradas:

Operación cliente: `op`
Redirección líder$_i$: `int`, procedente de la réplica i
Respuesta colectiva$_i$: `RespuestaColectiva`$_i$, procedente de la réplica i
Respuesta individual$_i$: `RespuestaIndividual`$_i$, procedente de la réplica i
Timeout

Salidas:

Petición$_i$: `Operacion`
Resultado: `result`

Operaciones:

(E) Operación cliente { `op` }

acciones:
$qOp \leftarrow qOp \cdot$ `Operacion: {op, generarUUID, verificarSoloLectura, timestampActual}`

(E) Redirección líder$_i$ { `idLider` }

acciones:
$q_{i,c} \leftarrow q_{i,c} \cdot$ `idLider`

(E) Respuesta colectiva$_i$ { `RespuestaColectiva` }

acciones:
$q_{i,c} \leftarrow q_{i,c} \cdot$ `RespuestaColectiva`

(E) Respuesta individual$_i$ { `RespuestaIndividual` }

acciones:

$q_{i,c} \leftarrow q_{i,c} \cdot$ `RespuestaIndividual`

(S) Petición$_i$: `Operacion`

precondiciones:

$q_{c,i} =$ `Operacion` $\cdot\, q'_{c,i}$

(S) Resultado Operación: `resultado`

precondiciones:

`resultado` $\neq \varnothing$

(I) Expiración de temporizador (*timeout*)

acciones:

`temporizadorExpirado` $\leftarrow$ `verdadero`

(I) Nueva operación { `op` }

precondiciones:

`opActual`$= \varnothing$

`qOp` $=$ { `op, uuid, ts` } $\cdot$ `qOp'`

acciones:

`opActual` $=$ { `op, uuid, ts` }

`intentosUnicast` $\leftarrow 0$

`qOp` $\leftarrow$ `qOp'`

`resultado` $\leftarrow \varnothing$

(I) Primer intento de envío de una nueva operación.

precondiciones:

`opActual` $\neq \varnothing$

`intentosUnicast` $= 0$

acciones:

`resetTimeout`

$q_{c,liderActual} \leftarrow q_{c,liderActual} \cdot$ `opActual`

`intentosUnicast` $\leftarrow$ `intentosUnicast` $+1$

(I) Tratamiento erróneo de líder: Se recibe un mensaje de un nodo indicando que el líder es él mismo. Se ignora el mensaje.

precondiciones:
$q_{i,c} = \texttt{Lider}(i) \cdot q'_{i,c}$
$i \neq \texttt{liderActual}$

acciones:
$q_{i,c} \leftarrow q'_{i,c}$

(I) Durante el procesamiento de una operación, el supuesto líder actual dice que el líder es él mismo. Este comportamiento es potencialmente bizantino, se cambia de líder aleatoriamente y se realiza un *broadcast*.

precondiciones:
$q_{i,c} = \texttt{Lider}(i) \cdot q'_{i,c}$
$i = \texttt{liderActual}$
$\texttt{intentosUnicast} \geqslant 0$
$\texttt{opActual} \neq \varnothing$

acciones:
$q_{i,c} \leftarrow q'_{i,c}$
$\texttt{liderActual} \leftarrow \texttt{rand}(n)$
$\texttt{intentosUnicast} \leftarrow -1$
$\forall_j q_{c,j} \leftarrow q_{c,j} \cdot \texttt{opActual}$
resetTimeout

(I) Si no se está procesando una operación o ya se ha realizado *broadcast*, y el supuesto líder actual dice que el líder es el mismo. Se ignora el mensaje.

precondiciones:
$q_{i,c} = \texttt{Lider}(i) \cdot q'_{i,c}$
$i = \texttt{liderActual}$
$\texttt{intentosUnicast} < 0 \vee \texttt{opActual} = \varnothing$

acciones:
$q_{i,c} \leftarrow q'_{i,c}$

(I) El supuesto líder actual redirige a otro nodo, tras haber realizado *broadcast* o no haber llegado a enviar una nueva operación. Se actualiza el líder actual.

precondiciones:

$q_{liderActual,c} = \texttt{Lider}\ (j) \cdot q'_{liderActual,c}$

$\texttt{liderActual} \neq j \wedge \texttt{intentosUnicast} \leqslant 0$

acciones:

$q_{liderActual,c} \leftarrow q'_{liderActual,c}$

$\texttt{liderActual} \leftarrow j$

(I) Mientras se espera respuesta a una petición de operación tras un intento de *unicast*, se recibe una redirección de líder. Se actualiza el líder.

precondiciones:

$q_{liderActual,c} = \texttt{Lider}\ (j) \cdot q'_{liderActual,c}$

$\texttt{intentosUnicast} \geqslant 1$

acciones:

$q_{liderActual,c} \leftarrow q'_{liderActual,c}$

$\texttt{liderActual} \leftarrow i$

$\texttt{intentosUnicast} \leftarrow \texttt{intentosUnicast} + 1$

(I) Mientras se espera respuesta a una petición de operación, se produce un *timeout*. Se realiza un *broadcast*.

precondiciones:

$\texttt{temporizadorExpirado} = \texttt{verdadero}$

$\texttt{intentosUnicast} > 0$

acciones:

$\texttt{intentosUnicast} \leftarrow -1$

$\texttt{liderActual} \leftarrow \texttt{rand}(n)$

$\forall_j q_{c,j} \leftarrow q_{c,j} \cdot \texttt{opActual}$

$\texttt{temporizadorExpirado} \leftarrow \texttt{falso}$

resetTimeout

(I) Se produce un *timeout* sin haber realizado una petición de operación. Se ignora.

precondiciones:

`temporizadorExpirado` $=$ `verdadero`

`intentosUnicast` $\leqslant 0$

acciones:

`temporizadorExpirado` $\leftarrow$ `falso`

(I) Respuesta colectiva; el identificador coincide con el de la operación actual. Se habilita la salida de resultado.

precondiciones:

$q_{i,c} =$ `RespuestaColectiva` $\cdot\, q'_{i,c}$

`opActual` $\neq \varnothing$

`RespuestaColectiva.uuid` $=$ `opActual.uuid`

acciones:

`opActual` $\leftarrow \varnothing$

`liderActual` $\leftarrow i$

`respuestasActuales` $\leftarrow []$

$q_{i,c} \leftarrow q'_{i,c}$

`resultado` $\leftarrow$ `RespuestaColectiva.result`

(I) Respuesta colectiva; el identificador no coincide con el de la operación actual. Se ignora la respuesta.

precondiciones:

$q_{i,c} =$ `RespuestaColectiva` $\cdot\, q'_{i,c}$

`opActual` $= \varnothing \vee$ `RespuestaColectiva.uuid` $\neq$ `opActual.uuid`

acciones:

$q_{i,c} \leftarrow q'_{i,c}$

(I) Respuesta individual; el identificador coincide con el de la operación actual y no se ha recibido una respuesta previa del mismo nodo para dicha operación. Se almacena el resultado.

precondiciones:

$q_{i,c} = \texttt{RespuestaIndividual} \cdot q'_{i,c}$

$\texttt{opActual} \neq \varnothing$

$\texttt{RespuestaIndividual.uuid} = \texttt{opActual.uuid}$

$\texttt{respuestasActuales}[i] = \varnothing$

acciones:

$\texttt{respuestasActuales}[i] \leftarrow \texttt{RespuestaIndividual.result}$

$q_{i,c} \leftarrow q'_{i,c}$

(I) Respuesta individual; el identificador coincide con el de la operación actual, pero se había recibido una respuesta previa del mismo nodo para dicha operación. Se ignora el mensaje.

precondiciones:

$q_{i,c} = \texttt{RespuestaIndividual} \cdot q'_{i,c}$

$\texttt{opActual} \neq \varnothing$

$\texttt{RespuestaIndividual.uuid} = \texttt{opActual.uuid}$

$\texttt{respuestasActuales}[i] \neq \varnothing$

acciones:

$q_{i,c} \leftarrow q'_{i,c}$

(I) Respuesta individual; el identificador no coincide con el de la operación actual. Se ignora la respuesta.

precondiciones:

$q_{i,c} = \texttt{RespuestaIndividual} \cdot q'_{i,c}$

$\texttt{opActual} = \varnothing \vee \texttt{RespuestaIndividual.uuid} \neq \texttt{opActual.uuid}$

acciones:

$q_{i,c} \leftarrow q'_{i,c}$

(I) Existe un número de respuestas individuales similares suficiente para considerar que se ha alcanzado consenso sobre la operación, en caso de que la operación implique escritura de datos. Se habilita la salida de resultado.

precondiciones:

$\exists$ `replies, result` $\mid$ `replies` $\subseteq$ `respuestasActuales` $\wedge$ $|$`replies`$| \geqslant \frac{2n+1}{3} \wedge$

$\wedge \forall$ `reply` $\in$ `replies`, `reply.result` $=$ `result`

`opActual.esSoloLectura` $=$ `verdadero`

acciones:

`opActual` $\leftarrow \varnothing$

`respuestasActuales` $\leftarrow []$

`resultado` $\leftarrow$ `result`

(I) Existe un número de respuestas individuales similares suficiente para considerar que se ha alcanzado consenso sobre la operación, en caso de que la operación sólo implique lectura de datos. Se habilita la salida de resultado.

precondiciones:

$\exists$ `replies, result` $\mid$ `replies` $\subseteq$ `respuestasActuales` $\wedge$ $|$`replies`$| \geqslant \frac{n-1}{2} \wedge$

$\wedge \forall$ `reply` $\in$ `replies`, `reply.result` $=$ `result`

`opActual.esSoloLectura` $=$ `falso`

acciones:

`opActual` $\leftarrow \varnothing$

`respuestasActuales` $\leftarrow []$

`resultado` $\leftarrow$ `result`

(I) La distribución actual de respuestas individuales impide alcanzar consenso, en caso de que la operación implique escritura de datos. Se descartan todas las respuestas recibidas y se realiza un nuevo *broadcast*.

precondiciones:

$\forall v \in |\pi$ `respuestasActuales` $|, \frac{2n+1}{3} - v > \mathtt{n} - \sum v$

`opActual.esSoloLectura` $=$ `falso`

acciones:

`respuestasActuales` $\leftarrow []$

`intentosUnicast` $\leftarrow -1$

$\forall_j q_{c,j} \leftarrow q_{c,j} \cdot$ `opActual`

`resetTimeout`

(I) La distribución actual de respuestas individuales impide alcanzar consenso, en caso de que la operación sólo implique lectura de datos. Se descartan todas las respuestas recibidas y se realiza un nuevo *broadcast*.

precondiciones:

$\forall v \in |\pi\ \texttt{respuestasActuales}\ |, \frac{n+1}{2} - v > \texttt{n} - \sum v$

$\texttt{opActual.esSoloLectura} = \texttt{verdadero}$

acciones:

$\texttt{respuestasActuales} \leftarrow []$

$\texttt{intentosUnicast} \leftarrow -1$

$\forall_j q_{c,j} \leftarrow q_{c,j} \cdot \texttt{opActual}$

resetTimeout

Argumentación de las condiciones de seguridad:

1. Si se envía una petición de operación a alguna réplica del servicio, previamente el cliente habrá recibido una orden para dicha operación.
   Un envío de solicitud de operación a las réplicas del servicio siempre se realiza en primer lugar con independencia de re-emisiones o difusiones posteriores. Este primer envío se realiza cuando existe una operación actualmente en curso y no se han realizado intentos de envío previos para dicha operación. Tal situación se produce cuando no hay ninguna operación en curso y existe al menos una orden de operación en cola, en cuyo caso la primera operación de la cola se convierte en la operación actualmente en curso y el número de intentos de envío inicial para dicha operación se fija a cero, cumpliendo las precondiciones del primer intento de envío.
2. El cliente únicamente emitirá salidas de resultado para aquellas órdenes de operación que haya recibido, pudiendo emitir un máximo de una salida de resultado por cada orden de operación recibida.
   El cliente sólo emite una salida de resultado al recibir una respuesta colectiva válida referente a la operación en curso o bien una cantidad suficiente de respuestas individuales concordantes. Tras emitir dicha salida de resultado, se actualiza la operación en curso y se pierde toda constancia de la salida cuyo resultado se ha emitido, por lo que no se puede emitir una nueva salida de resultado para la operación anterior.
   Complementando la argumentación anterior, las respuestas sólo se consideran válidas si hacen referencia a la operación en curso, a través de un identificador unívoco de la operación cuya generación es resistente a colisiones. Por lo tanto, dichas respuestas válidas serán consecuencia de peticiones de operación enviadas con anterioridad al servicio. Tal como argumenta la primera condición de seguridad de este algoritmo, el envío de una petición de operación a las réplicas del servicio será consecuencia de la recepción en el cliente de una orden para dicha operación.

Combinando esas tres argumentaciones, se obtiene que toda salida de resultado emitida por el cliente es consecuencia de una orden de operación y que para cada orden de operación recibida no se emita más de una salida de resultado.

3. En caso de recibir una orden de operación que hubiera sido precedida por otra anterior, no se envía ninguna solicitud al servicio para la nueva operación hasta no haber emitido una salida de resultado relativa a la orden de operación precedente.
   Cuando se emite una salida de resultado, ésta se corresponde inequívocamente con la operación actualmente en curso. Al emitir la salida de resultado se desasigna la operación actualmente en curso (limpiando el valor de la variable correspondiente). Esto, junto con la disponibilidad de más operaciones en la cola, supone el procesamiento de una nueva operación, que eventualmente alcanzará un resultado consensuado. No obstante, dado que la nueva orden de operación no se comienza a procesar hasta no haber emitido la salida de resultado de la operación anterior, la salida de resultado de la segunda operación se realizará con posterioridad a la de la primera.

Argumentación de las condiciones de viveza:

1. Toda orden de operación recibida termina siendo trasladada a las réplicas del servicio.
   Una orden de operación se traslada a alguna de las réplicas del servicio cuando la operación en curso se encontraba desasignada y la operación actual estaba en la cabeza de la cola de órdenes de operación. La condición se cumple trivialmente si la operación corresponde a la primera orden que se recibe en la ejecución del algoritmo cliente. Sin embargo, en los demás casos esto sólo se cumple cuando el cliente obtiene consenso en las respuestas por parte de las réplicas del servidor para la operación en curso. Si no se recibe una respuesta consensuada para una operación en curso, no se llegará a atender una nueva operación y las órdenes de operación recibidas se acumularán en la cola indefinidamente. Por lo tanto, el cumplimiento de esta condición está condicionado al de la segunda condición de viveza, según la cual toda difusión de peticiones de operación termina recibiendo una respuesta colectiva o una cantidad suficiente de respuestas individuales procedentes de diferentes réplicas. Ante esto, se emitirá una salida de resultado, se desasignará la operación en curso y podrá procesarse la siguiente orden de operación de la cola.

2. Toda difusión de peticiones de operación termina recibiendo una respuesta colectiva o una cantidad suficiente de respuestas individuales, procedentes de diferentes réplicas.
   El cumplimiento de esta condición no depende de este algoritmo, sino del algoritmo de consenso ejecutado por las réplicas del servicio, por lo que no se puede realizar afirmación alguna al respecto contando únicamente con el algoritmo cliente, siendo el algoritmo del servicio el que debe garantizar el cumplimiento de las condiciones asociadas correspondientes.

Esta condición se establece aquí para que sirva como base a otras condiciones que se asientan sobre un correcto funcionamiento del algoritmo de consenso del servicio.

3. Toda difusión de peticiones de operación termina produciendo una salida de resultado.
   Una difusión de peticiones de operación garantiza la obtención de un resultado consensuado bajo las premisas expresadas en la segunda condición de viveza. La obtención de esta respuesta consensuada se traducirá en la emisión de una salida de resultado.

4. Toda orden de operación termina produciendo una salida de resultado.
   Por una parte, la primera condición de viveza establece que toda orden de operación recibida termina siendo trasladada a las réplicas del servicio. La obtención de una respuesta consensuada por parte de las réplicas del servicio se traduce en la emisión de una salida de resultado. La recepción de dicha respuesta consensuada puede producirse como consecuencia de un envío individual de petición de operación o como resultado de una difusión. Una emisión simple de petición de operación no garantiza la obtención de un resultado consensuado, pero una difusión sí lo hace, tal como establece la segunda condición de viveza. Si pasado un tiempo acotado tras la emisión simple de una petición de operación no se obtiene un resultado consensuado por las réplicas del servicio, se realizará una difusión de la petición de operación. Esto termina coincidiendo con el escenario descrito en la tercera condición de viveza y que garantiza la emisión de una salida de resultado para una operación de la que se ha realizado una difusión a las réplicas del servicio, de acuerdo con las premisas expresadas en la segunda condición de viveza.

3.3 Caso práctico de implementación

Finalmente, se presenta un caso práctico demostrativo mediante la implementación de Trebizond, trasladando su especificación a código fuente en un programa. Esta implementación se ha realizado en el lenguaje de programación Typescript, un súperconjunto de Javascript sobre el que proporciona construcciones adicionales y tipos estáticos. Esto permite al desarrollador adoptar una aproximación al paradigma de orientación a objetos con una sintaxis similar a la de otros lenguajes populares orientados a objetos, como Java y C#, manteniendo la flexibilidad que proporciona Javascript. Los lenguajes basados en Javascript presentan un comportamiento reactivo sobre un único hilo de ejecución, lo que conduce a que la implementación del algoritmo se diseñe mediante un enfoque dirigido por eventos. Este tipo de modelo y los de programación reactiva resultan especialmente apropiados para implementar algoritmos dirigidos a sistemas concurrentes, en este caso particular un algoritmo distribuido como lo es Trebizond.

La interacción entre las entidades participantes en el algoritmo a través de la red se implementa utilizando ZeroMQ, una biblioteca de mensajería asíncrona de alto rendimiento, especialmente apropiada para su uso en aplicaciones concurrentes. Esta biblioteca se encuentra disponible para un gran número de lenguajes y entornos, proporcionando un conjunto de *sockets* especializados para la implementación de diferentes patrones de comunicación basados en mensajes, permitiendo definir protocolos de comunicación de nivel de aplicación sobre tecnologías de comunicación de diversa naturaleza, desde TCP/IP hasta IPC (*Inter-Process Communication*). Las capacidades de ZeroMQ relativas a la gestión de la asincronía se emplean en conjunción con los mecanismos de gestión de eventos de Typescript (heredados de Javascript), con la finalidad de dar lugar a un modelo de implementación reactiva que pueda codificarse directamente a partir de un diseño realizado mediante autómatas de entrada/salida, puesto que este modelo determina las acciones a ejecutar ante la recepción de un evento, en función de dicho evento y del estado actual de la entidad modelada. En este caso práctico se optó por materializar la máquina de estados replicada por el algoritmo mediante una base de datos Redis, teniendo cada nodo de la red una instancia de la base de datos que ejemplifica su réplica del estado compartido. Las figuras 6.1 y 6.2 muestran los modelos de conexión utilizados por los nodos de la red del algoritmo y por sus clientes, haciendo uso de las abstracciones facilitadas por los diferentes tipos de *sockets* que proporciona ZeroMQ.

Respecto al diseño seguido a la hora de abordar la implementación del algoritmo en el contexto de un sistema que hiciera uso del mismo para alcanzar consenso distribuido sobre un estado replicado, es conveniente realizar varias observaciones que destacan aspectos importantes de dicho diseño. Como se expuso en la sección 6.2, el algoritmo se especificó utilizando autómatas de entrada/salida, permitiendo realizar una implementación reactiva codificando las entradas como eventos ante los que se ejecuta una serie de acciones en función del estado actual y del evento recibido.

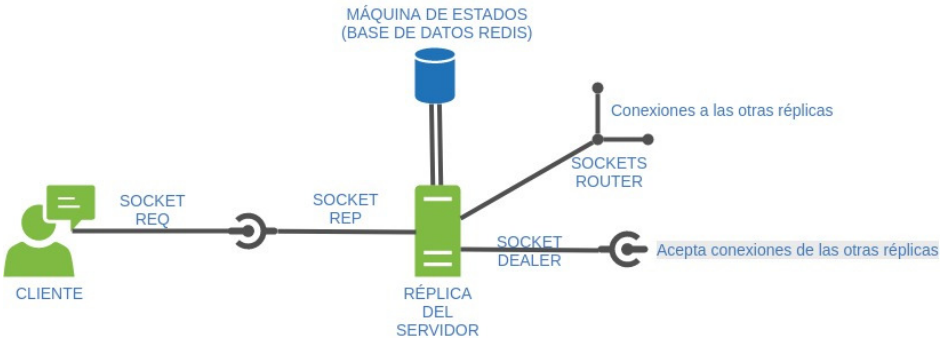

Figura 3.1: Interfaces de red de un nodo

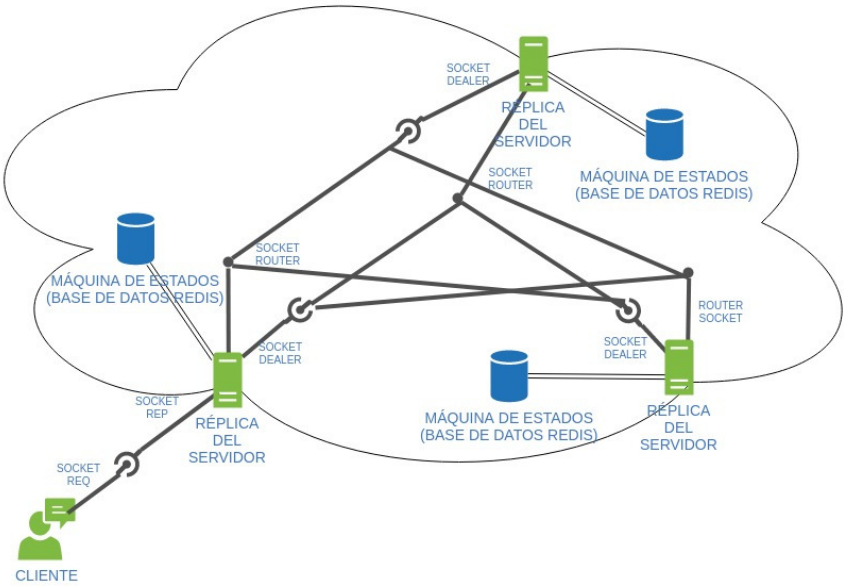

Figura 3.2: Arquitectura de red del sistema

De acuerdo con el principio de separación de responsabilidades, se ha adoptado una arquitectura estructurada en dos capas, delegando la gestión de la comunicación a la capa inferior, denominada *controlador de red*. De esta forma, la capa superior se especializa en proporcionar una implementación limpia del comportamiento modelado, enviando mensajes a través de funciones de alto nivel y abstrayéndose de cómo la comunicación se realiza en los niveles inferiores. La capa inferior además incorpora el detector de fallos del nodo, por lo que puede notificar al nivel superior en caso de sospechar de otro nodo, incluyendo las validaciones semántica y de autorización. Ambas capas se comunican mediante los mecanismos de control de la asincronía que proporcionan los lenguajes basados en Javascript, como las promesas, objetos que representan la finalización (o fallo) eventual de una operación asíncrona junto con su valor resultante.

Cuando un nodo recibe una solicitud de mensaje de otro nodo, la capa de red deserializa el mensaje y separa sus cabeceras, tras lo que lo transfiere a la capa superior a través de un método de callback. Este método clasifica el mensaje de acuerdo a su tipo (y, en función del caso, de acuerdo a algunos de sus atributos) e invoca al método responsable de tratar mensajes de dicho tipo y actualizar las variables de estado del nodo en consecuencia. Cuando un nodo envía un mensaje a otro nodo, la capa del controlador de red aplica la función de firma con la clave privada del nodo sobre el mensaje y lo dirige al nodo destinatario a través del *socket* correspondiente. Puesto que la capa que controla las interacciones con la red proporciona una interfaz de gestión de las entradas y salidas del nodo, también proporciona soporte al tratamiento de los temporizadores, permitiendo su programación y comportándose como si se hubiera recibido un mensaje de un determinado tipo al cumplirse el lapso marcado, presentando una correspondencia con el papel de una acción de entrada en el modelo de autómatas. De esta forma, la capa superior sólo debe asumir la responsabilidad de implementar el comportamiento conveniente para cada combinación de tipo de mensaje, su contenido y el estado actual, encargándose de lanzar la invocación al método correspondiente, adecuándose a los principios del paradigma de programación dirigida por eventos. [Hib76]

Con el fin de separar el algoritmo de búsqueda de consenso de su aplicación específica y de la naturaleza del sistema en el que se integre, la implementación de Trebizond se plantea como una biblioteca estructurada en diferentes módulos. Dado que el entorno de ejecución de la implementación realizada es NodeJS, el establecimiento de varios módulos separados se realiza mediante NPM, el gestor de paquetes de NodeJS. La implementación realizada se divide en el módulo servidor, que implementa el algoritmo de consenso de Trebizond, y el módulo cliente, que implementa el correspondiente algoritmo homónimo. Cada uno de los módulos proporciona una interfaz al desarrollador de aplicaciones distribuidas, haciendo uso de clases abstractas y tipos de datos genéricos, que declaran las operaciones que una máquina de estados replicada debe soportar pero se mantiene agnóstica a la naturaleza de dicha máquina de estados, de las operaciones que se ejecutan sobre ella y del contenido de las funciones de validación semántica utilizadas por el algoritmo, lo que se representa en la figura 6.3.

Por lo tanto, la implementación de la máquina de estados y la integración del conocimiento que incorpora la validación semántica no se concreta hasta la instanciación de los objetos que actúan como puntos de entrada y configuración de cada uno de los módulos. En el caso práctico implementado, la máquina de estados se plasma en la forma de una conexión a una instancia individual de la base de datos Redis. El hecho de que esta base de datos se constituya como un almacén de pares clave/valor permite una implementación sencilla de operaciones de asignación, actualización, consulta, aritméticas y lógicas, suponiendo un caso de ejemplo notablemente ilustrativo. Como ya se ha puntualizado, tanto las peticiones que contienen las operaciones a ejecutar sobre la máquina de estados al alcanzar consenso como

la forma en la que dichas operaciones se ejecutan sobre la máquina de estados son transparentes para el módulo servidor que implemento el algoritmo, pues que éstas se declaran en la forma de tipos de datos genéricos que se resuelven al inicializar el servicio. Esta independencia del algoritmo de consenso respecto a las operaciones que se ejecutan sobre la máquina de estados y la propia naturaleza de ésta contribuye a implementar la separación entre consenso y ejecución, como se propone en [YMV+03], así como a aplicar otras buenas prácticas para la implementación de servicios replicados tolerantes a fallos referidas en [Sch90].

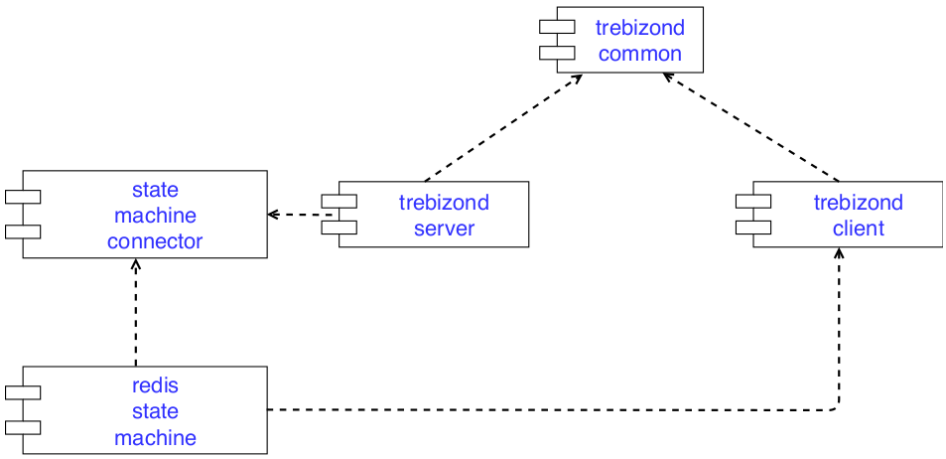

Figura 3.3: Módulos de la implementación de Trebizond

Buscando facilitar los procesos de desarrollo y pruebas, el despliegue de la red de réplicas de Trebizond se distribuye en contenedores Docker. La orquestación y activación de los recursos necesarios se ha automatizado mediante Docker Compose, que también proporciona un servicio interno de resolución de nombres que permite a los componentes de la red comunicarse entre sí a partir de los nombres de sus interlocutores, sin necesidad de conocer sus direcciones IP. Las instancias de la base de datos Redis también se despliegan de esta forma, encontrándose cada una en una subred sólo accesible al nodo correspondiente. Cada contenedor del servicio expone un *endpoint* público a la máquina anfitriona a través de la red virtual de Docker, mapeándose a números de puerto correlativos, puesto que todos los contenedores se despliegan físicamente sobre la misma máquina. Durante el desarrollo, con anterioridad a la obtención de una versión estable del software, se empleó un volumen virtual de Docker de tal forma que un directorio de la máquina anfitriona que contuviera el código fuente en desarrollo pudiera ser accedido por todos los contenedores Docker, que en esta parte del proceso ejecutaban una imagen Docker base de NodeJS. Una vez que se alcanzó el hito en el que se pudo considerar que se había logrado obtener una versión estable del software, se generó una imagen Docker propia, la cual contiene la distribución del código fuente a desplegar (al tratarse de un lenguaje de programación interpretado) y tiene instalado todo el software complementario necesario. El hecho de haber usado Git como sistema de control de versiones durante el desarrollo del código fuente permitió hacer uso de la funcionalidad

de construcción automática proporcionada por DockerHub, que se vincula a un repositorio de Git y se lanza de forma autónoma cuando se inserta un *commit*, reconstruyendo la imagen Docker a partir de los contenidos actuales del repositorio y las especificaciones del archivo Dockerfile, de acuerdo con las prácticas de integración continua.

El siguiente paso lógico en este proceso es dirigirse a una solución ubicua, migrando a plataformas distribuidas de gestión de contenedores, como Docker Swarm y Kubernetes, que permiten el despliegue de una arquitectura de aplicación basada en contenedores sobre múltiples máquinas (físicas o virtuales), conectadas por medio de una red virtual que se superpone a la red física. Usando las herramientas apropiadas, como Rancher en el caso de Kubernetes, el traspaso a una solución de contenedores ubicua sería relativamente sencillo.

En lo que se refiere a las pruebas realizadas sobre la implementación del algoritmo, con el fin de comprobar que ésta es fiel al comportamiento que ha de reproducir, se diseñó un conjunto de casos de prueba unitarios que se activaban conforme se añadía nueva funcionalidad al código de forma sucesiva. A la hora de realizar un *commit* al repositorio de código fuente se ejecutaban todos los casos de prueba activados hasta el momento, debiendo superarse todos ellos para permitir que el nuevo código se incorporara a la distribución del repositorio. Estos casos de prueba unitarios se complementan con pruebas de integración que verifican el correcto comportamiento colectivo del programa y con pruebas de caja blanca que sitúan a los nodos de la red en un determinado estado (mediante la configuración manual de los valores de sus variables internas) y comprueban los cambios de estado producidos ante la recepción y el envío de determinados tipos de mensajes.

El código fuente de la implementación de Trebizond puede encontrarse en la siguiente ubicación, perteneciente a un repositorio de Git accesible a los miembros del DSIC:
`https://gitlab.dsic.upv.es/fraferp9/trebizond`

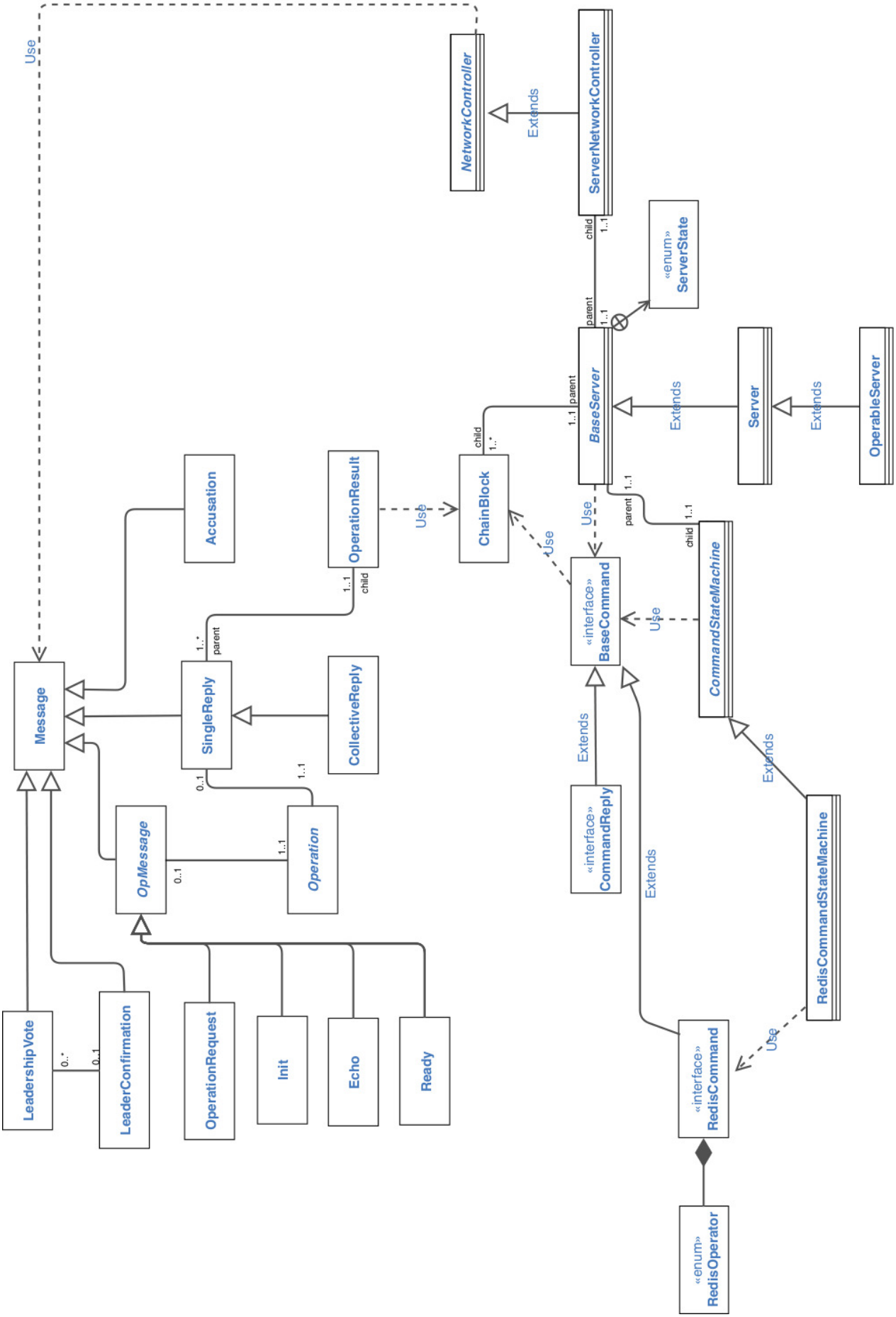

Figura 3.4: Diagrama de clases de la implementación de Trebizond

Capítulo 4

Conclusiones y propuestas

FINALMENTE, en este capítulo se presenta una síntesis acerca del trabajo realizado, resaltando las aportaciones que realiza al estado del arte precedente. Se detallan algunas de las líneas de trabajo futuro que pueden resultar interesantes de cara a la mejora o ampliación del algoritmo desarrollado y de su caso práctico de implementación, y se concluye con una apreciación personal sobre el presente Trabajo Fin de Máster como punto final a la titulación del Máster Universitario en Computación Paralela y Distribuida.

4.1 Síntesis del trabajo realizado

El presente trabajo ha realizado un estudio de la tecnología blockchain, los diferentes tipos de plataformas que hacen uso de esta tecnología, sus principales aplicaciones y los métodos utilizados para alcanzar consenso en cada una de ellas. La capacidad para obtener consenso distribuido sobre un estado compartido y replicado permite a las plataformas blockchain actuar como garantes de confianza descentralizada en un registro incremental de documentos o transacciones, asegurando la integridad de los datos.

Profundizando en el ámbito de las plataformas blockchain permisionadas y del tipo de algoritmos que suelen emplear para alcanzar consenso, se ha realizado un estudio cronológico relativo al problema del consenso distribuido. Dicho estudio se ha estructurado de forma que aborde los algoritmos propuestos para resolver el problema del consenso bajo diferentes escenarios y condiciones, junto con otros problemas fundamentales de los sistemas distribuidos que se encuentran estrechamente ligados a éste.

Los resultados ofrecidos permiten comprobar que la validación semántica es una herramienta muy poderosa en el caso de los algoritmos de consenso dirigidos a plataformas blockchain, puesto que permite condicionar el comportamiento del algoritmo al resultado de operaciones de verificación directamente dependientes de la lógica de negocio en la que éste se integra. Sin embargo, los algoritmos actuales que hacen uso de técnicas de validación semántica las emplean únicamente a la hora de decidir si una operación es válida para su aplicación sobre el estado replicado. Por otra parte, la mayor parte de los algoritmos de consenso emplean técnicas de enmascaramiento de fallos, destinada a proporcionar tolerancia ante un número de fallos simultáneos acotado superiormente. En cambio, la detección

activa de fallos busca identificar a aquellos nodos que presentan un comportamiento susceptible de ser malicioso, pudiendo tomar acciones contra ellos tales como su exclusión de las comunicaciones entre los miembros del grupo.

La principal aportación del presente trabajo al estado del arte actual es la utilización conjunta de las técnicas de validación semántica y detección activa de fallos, con la finalidad de aumentar la efectividad del detector de fallos. Además de otorgar al algoritmo las ventajas que cada una de estas técnicas supone individualmente, su combinación permite detectar un comportamiento malicioso de origen bizantino en otros nodos no sólo por la ausencia de respuesta o por la recepción de mensajes inconsistentes con el propio algoritmo de consenso, sino incorporando conocimiento propio de la lógica de negocio de la aplicación.

Estas observaciones toman forma mediante la especificación de Trebizond, un algoritmo de consenso tolerante a fallos bizantinos especialmente apropiado para su aplicación en sistemas blockchain, que hace uso de las técnicas de validación semántica y detección activa de fallos de forma combinada. El algoritmo general se divide en varios algoritmos de menor dimensión, siguiendo el principio de separación de problemas y con el objetivo de reducir la complejidad del algoritmo, disminuyendo con ello la probabilidad de introducir errores en su diseño y aumentando su inteligibilidad. La especificación del algoritmo se realiza mediante autómatas de entrada/salida, un modelo de especificación formal para sistemas concurrentes, como los sistemas distribuidos, y que facilita la transición de la especificación diseñada a una implementación de acuerdo con el paradigma de programación dirigida por eventos.

Como caso práctico demostrativo, se ha realizado una implementación de Trebizond. Su principal motivación, además de complementar la especificación del algoritmo diseñado plasmándolo en un programa real, es la de mostrar mecanismos, técnicas, buenas prácticas y patrones de diseño que facilitan el proceso de desarrollo y permiten dar lugar a un producto software de calidad. Por una parte, se hace énfasis en la forma de implementar un algoritmo mediante el paradigma de programación dirigida por eventos aprovechando los mecanismos de control de la asincronía del lenguaje de programación. Por otra parte, se muestra cómo la aplicación del paradigma de programación orientada a objetos se puede utilizar para estructurar las dependencias entre clases de tal forma que no exista acoplamiento entre el algoritmo y la aplicación en la que se integra. Esto se logra estableciendo interfaces bien definidas para aquellos aspectos externos al algoritmo con los que éste ha de operar, como la propia máquina de estados, la naturaleza de las operaciones y cómo se ejecutan, y el conocimiento de la lógica de negocio incluido en las funciones de validación semántica. La aplicación de estos principios permite obtener un software bien estructurado, con una separación de responsabilidades clara y una interfaz de operación flexible que le permite integrarse con mínimo esfuerzo en aplicaciones pertenecientes a distintos modelos de negocio.

4.2 Propuestas y líneas de trabajo futuro

Dada la principal componente investigadora de la labor realizada, cuya motivación es la de asimilar el trabajo ya existente y realizar una aportación sobre el estado previo del campo de conocimiento, se contemplan una serie de puntos que podrían servir como líneas de investigación y desarrollo futuro. Estas vías de trabajo, que no se han llevado a cabo por situarse fuera del alcance acotado o por restricciones temporales, se plantean como una extensión al trabajo expuesto o como una forma de complementarlo y son las siguientes:

Demostración formal del algoritmo: El hecho de haber usado un método formal para la especificación del algoritmo, como los autómatas de entrada/salida, favorece la tarea de abordar su demostración. No obstante, la gran complejidad de esta tarea, basada en recursos como la inducción matemática, y el elevado plazo de tiempo que exige han supuesto que no se llegue a abordar. En su lugar se ha realizado una argumentación textual respecto a las condiciones de seguridad y viveza expresadas para cada uno de los algoritmos individuales que componen Trebizond. La realización de una demostración formal del algoritmo sería de gran interés como una forma de verificar que el algoritmo efectivamente cumple con su cometido de acuerdo a sus condiciones de operación, manteniendo sus condiciones de seguridad en todo momento y progresando siempre que el número de nodos bizantinos no supere el umbral establecido.

Cálculo de complejidad del algoritmo: En el diseño y la especificación del algoritmo se han tratado de aplicar técnicas que minimizaran su complejidad, tanto algorítmica como en términos del número de mensajes enviados y el tamaño de éstos. Aun así, sería conveniente realizar un análisis de la complejidad del algoritmo en términos computacionales, espaciales y relativos a las comunicaciones, de tal forma que Trebizond se pudiera comparar objetivamente en términos de complejidad con otros algoritmos de consenso tolerantes a fallos bizantinos en condiciones de sincronía eventual.

Externalización de la validación semántica: Trebizond es deliberadamente ambiguo respecto a la forma en la que se realiza la validación semántica, que se presenta a través de la interfaz de una función que recibe una operación y devuelve un valor lógico. Esto permite flexibilizar la implementación de la validación semántica, que puede ser desde una función que se ejecuta en el propio proceso local del algoritmo hasta una consulta a un servicio externo. La externalización explícita (parcial o total) de la validación semántica en otras máquinas, ya fueran un subconjunto de los nodos en los que se ejecuta el algoritmo o máquinas ajenas, podría dar lugar a una afirmación sobre la presencia de un componente de cómputo incorruptible y de confianza, con las ventajas que esto puede aportar. Su inclusión podría ser una dirección de investigación interesante a la hora de desarrollar otros algoritmos de consenso o de proponer mejoras al aquí propuesto.

Integración en Hyperledger Fabric: Buena parte de las decisiones de diseño a la hora de valorar cómo estructurar el algoritmo han sido influenciadas por el estudio del proceso de rediseño que sufrió Hyperledger Fabric con el fin de desacoplar la propia plataforma blockchain del algoritmo de consenso utilizado. Junto con el propósito de separar las capas de consenso y ejecución, esto ha dado lugar a un algoritmo especialmente diseñado para facilitar su integración en plataformas blockchain de diferente naturaleza. Por lo tanto, sería interesante estudiar cómo podría llevarse a cabo la integración de Trebizond en la plataforma de Hyperledger Fabric, como una forma de constatar a efectos prácticos la flexibilidad de Trebizond y su independencia respecto a la plataforma blockchain subyacente.

4.3 Conclusión personal

El presente Trabajo Fin de Máster constituye la culminación del plan de estudios del Máster Universitario en Computación Paralela y Distribuida. Al tratarse de un programa de posgrado, todos los que hemos podido acceder a él habíamos obtenido con anterioridad una titulación universitaria, la de Ingeniero en Informática en mi caso. En su momento ya sabía que alcanzar el título de Ingeniero no era más que el comienzo del camino y que de ninguna forma podría pretender haber adquirido y asimilado ya más que las bases de un campo del saber mucho más amplio. Este máster me ha servido no sólo para continuar avanzando en ese eterno camino de aprendizaje, adquiriendo conocimientos que considero enormemente interesantes y útiles en el área de la Informática que más me atrae, sino para tener la oportunidad de orientar este trabajo en una vertiente investigadora. A diferencia de lo que suelen ser los Proyectos Fin de Carrera, generalmente orientados a una aplicación eminentemente práctica, el hecho de haber dotado a este trabajo de una aproximación más teórica, realizando un estudio extensivo del estado del arte de acuerdo con el área que pretendía explorar y a la que esperaba tener algo que aportar me ha servido para encontrarme con el ámbito de la investigación científica universitaria, en la que hasta ahora no había tenido el placer de trabajar.

Cuando solicité al profesor Bernabéu que dirigiera mi Trabajo Fin de Máster y él me realizó su propuesta temática, muchos de los términos de los que me hablaba me resultaban poco familiares o directamente desconocidos. Poco a poco, a medida que mis compañeros y yo avanzábamos en las asignaturas del máster y complementaba la formación magistral con la lectura de artículos y documentos de investigación, fui comprendiendo todos esos conceptos: algoritmos de consenso, modelos de replicación, la tecnología blockchain, contenedores de procesos... Sobre todo en la recta final del curso era bastante frecuente que de pronto reconociera algún concepto del que el profesor Bernabéu me hubiera hablado un par de meses atrás y que a partir de ese momento fuera consciente de su relación con el tema de este trabajo y la cabida que podía tener en él.

La cristalización de ese proceso es este trabajo, en el que, haciendo acopio de la información recopilada, he tratado de aplicar los conocimientos adquiridos y el propio afán de innovación para dar lugar a algo que supusiera un avance sobre el estado actual del área.

Al finalizar este máster sólo puedo afirmar que mi motivación para seguir aprendiendo, formándome e investigando es altamente escalable y está más viva que nunca.

Referencias

[20117] Chain protocol. Whitepaper. *Available at* `https://chain.com/docs/1.2/protocol/papers/whitepaper`, 2017.

[ABD+15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, et al. Imperfect forward secrecy: How Diffie-Hellman fails in practice. En *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, páginas 5–17. ACM, 2015.

[ABKW98] Todd Anderson, Yuri Breitbart, Henry F Korth, y Avishai Wool. Replication, consistency, and practicality: are these mutually exclusive? *ACM SIGMOD Record*, 27(2):484–495, 1998.

[ADD+17] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, y Ling Ren. Efficient Synchronous Byzantine Consensus. *arXiv preprint arXiv:1704.02397*, 2017.

[ADMG+16] Matthew Amy, Olivia Di Matteo, Vlad Gheorghiu, Michele Mosca, Alex Parent, y John Schanck. Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3. En *International Conference on Selected Areas in Cryptography*, páginas 317–337. Springer, 2016.

[AEH75] Eralp A Akkoyunlu, Kattamuri Ekanadham, y Richard V Huber. Some constraints and tradeoffs in the design of network communications. En *ACM SIGOPS Operating Systems Review*, volume 9, páginas 67–74. ACM, 1975.

[AEMGG+05] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, y Jay J Wylie. Fault-scalable Byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74, 2005.

[ALRL04] Algirdas Avizienis, J-C Laprie, Brian Randell, y Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[AW04] Hagit Attiya y Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

[BA14] Alysson Neves Bessani y Eduardo Alchieri. A guided tour on the theory and practice of state machine replication. En *Tutorial at the 32nd Brazilian symposium on computer networks and distributed systems*, 2014.

[Bal17] Arati Baliga. Understanding blockchain consensus models. *Persistent*, 2017.

[BBHT98] Michel Boyer, Gilles Brassard, Peter Høyer, y Alain Tapp. Tight bounds on quantum searching. *Fortschritte der Physik: Progress of Physics*, 46(4-5):493–505, 1998.

[BDH11] Johannes Buchmann, Erik Dahmen, y Andreas Hülsing. XMSS-a practical forward secure signature scheme based on minimal security assumptions. En *International Workshop on Post-Quantum Cryptography*, páginas 117–129. Springer, 2011.

[Ber09] Daniel J Bernstein. Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete. *SHARCS*, 9:105, 2009.

[BHMT02] Gilles Brassard, Peter Hoyer, Michele Mosca, y Alain Tapp. Quantum amplitude amplification and estimation. *Contemporary Mathematics*, 305:53–74, 2002.

[BHT98] Gilles Brassard, Peter Høyer, y Alain Tapp. Quantum cryptanalysis of hash and claw-free functions. En *Latin American Symposium on Theoretical Informatics*, páginas 163–169. Springer, 1998.

[BL95] Dan Boneh y Richard J Lipton. Quantum cryptanalysis of hidden linear functions. En *Annual International Cryptology Conference*, páginas 424–437. Springer, 1995.

[BMR11] Olivier Baldellon, Achour Mostéfaoui, y Michel Raynal. A necessary and sufficient synchrony condition for solving Byzantine consensus in symmetric networks. En *International Conference on Distributed Computing and Networking*, páginas 215–226. Springer, 2011.

[BO83] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. En *Proceedings of the second annual ACM symposium on Principles of distributed computing*, páginas 27–30. ACM, 1983.

[Bra87] Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[BSS91] Kenneth Birman, Andre Schiper, y Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)*, 9(3):272–314, 1991.

[BT85] Gabriel Bracha y Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, y Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, 1996.

[CHTCB96] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, y Bernadette Charron-Bost. On the impossibility of group membership. En *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, páginas 322–330. ACM, 1996.

[CKPS01] Christian Cachin, Klaus Kursawe, Frank Petzold, y Victor Shoup. Secure and efficient asynchronous broadcast protocols. En *Annual International Cryptology Conference*, páginas 524–541. Springer, 2001.

[CKV01] Gregory V Chockler, Idit Keidar, y Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR)*, 33(4):427–469, 2001.

[CL+99] Miguel Castro, Barbara Liskov, et al. Practical Byzantine fault tolerance. En *OSDI*, volume 99, páginas 173–186, 1999.

[CL02] Miguel Castro y Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[CML+06] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, y Liuba Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. En *Proceedings of the 7th symposium on Operating systems design and implementation*, páginas 177–190. USENIX Association, 2006.

[CNV04] Miguel Correia, Nuno Ferreira Neves, y Paulo Verissimo. How to tolerate half less one Byzantine nodes in practical distributed systems. En *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, páginas 174–183. IEEE, 2004.

[CR92] R Canneti y Tal Rabin. Optimal asynchronous byzantine agreement. Technical report, Technical Report, 1992.

[Cri91a] Flavin Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[Cri91b] Flaviu Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4(4):175–187, 1991.

[CT96] Tushar Deepak Chandra y Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[CV17] Christian Cachin y Marko Vukolić. Blockchain consensus protocols in the wild. *arXiv preprint arXiv:1707.01873*, 2017.

[CZ14] Christopher Copeland y Hongxia Zhong. Tangaroa: a byzantine fault tolerant raft, 2014.

[DAAB+18] Stefano De Angelis, Leonardo Aniello, Roberto Baldoni, Federico Lombardi, Andrea Margheri, y Vladimiro Sassone. PBFT vs proof-of-authority: applying the CAP theorem to permissioned blockchain. 2018.

[DH76] Whitfield Diffie y Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[DLS88] Cynthia Dwork, Nancy Lynch, y Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[Dou02] John R Douceur. The sybil attack. En *International workshop on peer-to-peer systems*, páginas 251–260. Springer, 2002.

[DP89] Donald Watts Davies y Wyn L Price. *Security for computer networks: an introduction to data security in teleprocessing and electronic funds transfer*. Wiley, 1989.

[DS82] D Dolev y HR Strong. *Distributed commit with bounded waiting*. IBM Thomas J. Watson Research Division, 1982.

[ES18] Ittay Eyal y Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. *Communications of the ACM*, 61(7):95–102, 2018.

[FLP85] Michael J Fischer, Nancy A Lynch, y Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[FMMC12] Austin G Fowler, Matteo Mariantoni, John M Martinis, y Andrew N Cleland. Surface codes: Towards practical large-scale quantum computation. *Physical Review A*, 86(3):032324, 2012.

[FWH12] Austin G Fowler, Adam C Whiteside, y Lloyd CL Hollenberg. Towards practical classical processing for the surface code: timing analysis. *Physical Review A*, 86(4):042313, 2012.

[GKL15] Juan Garay, Aggelos Kiayias, y Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. En *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, páginas 281–310. Springer, 2015.

[GL02] Seth Gilbert y Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[GM98] Juan A Garay y Yoram Moses. Fully polynomial byzantine agreement for processors in rounds. *SIAM Journal on Computing*, 27(1):247–290, 1998.

[Gro97] Lov K Grover. Quantum mechanics helps in searching for a needle in a haystack. *Physical review letters*, 79(2):325, 1997.

[Hib76] Peter Hibbard. Parallel processing facilities. *New Directions in Algorithmic Languages*, páginas 1–7, 1976.

[HKD06] Andreas Haeberlen, Petr Kouznetsov, y Peter Druschel. The Case for Byzantine Fault Detection. En *HotDep*, 2006.

[HT94] Vassos Hadzilacos y Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

[HW90] Maurice P Herlihy y Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[JFR93] Farnam Jahanian, Sameh Fakhouri, y Ragunathan Rajkumar. Processor group membership protocols: Specification, design and implementation. En *Reliable Distributed Systems, 1993. Proceedings., 12th Symposium on*, páginas 2–11. IEEE, 1993.

[JMV01] Don Johnson, Alfred Menezes, y Scott Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International journal of information security*, 1(1):36–63, 2001.

[KAD+09] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, y Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 27(4):7, 2009.

[KBF+15] Julian Kelly, R Barends, AG Fowler, A Megrant, E Jeffrey, TC White, D Sank, JY Mutus, B Campbell, Yu Chen, et al. State preservation by repetitive error detection in a superconducting quantum circuit. *Nature*, 519(7541):66, 2015.

[KJ10] Manos Kapritsos y Flavio Paiva Junqueira. Scalable Agreement: Toward Ordering as a Service. En *HotDep*, 2010.

[KMMS98] Kim Potter Kihlstrom, Louise E Moser, y P Michael Melliar-Smith. The SecureRing protocols for securing group communication. En *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*, volume 3, páginas 317–326. IEEE, 1998.

[KMMS03] Kim Potter Kihlstrom, Louise E Moser, y P Michael Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, 2003.

[KR10] Petr Kuznetsov y Rodrigo Rodrigues. BFTW 3: why? when? where? workshop on the theory and practice of byzantine fault tolerance. *ACM SIGACT News*, 40(4):82–86, 2010.

[Lam78a] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer networks*, 2(2):95–114, 1978.

[Lam78b] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Lam84] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):254–280, 1984.

[Lam86] Leslie Lamport. On interprocess communication. *Distributed computing*, 1(2):86–101, 1986.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[Lam00] Leslie Lamport. Lower bounds on consensus. *Unpublished manuscript, March*, 2000.

[Lam03] Leslie Lamport. Lower bounds for asynchronous consensus. En *Future Directions in Distributed Computing*, páginas 22–23. Springer, 2003.

[Lam05] Leslie Lamport. Generalized consensus and Paxos. 2005.

[Lam06] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[LF79] Nancy A Lynch y Michael J Fischer. On describing the behavior and implementation of distributed systems. En *Semantics of Concurrent Computation*, páginas 147–171. Springer, 1979.

[LM07] Jinyuan Li y David Maziéres. Beyond One-Third Faulty Replicas in Byzantine Fault Tolerant Systems. En *NSDI*, 2007.

[LSP82] Leslie Lamport, Robert Shostak, y Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[LT87] Nancy A Lynch y Mark R Tuttle. Hierarchical correctness proofs for distributed algorithms. En *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, páginas 137–151. ACM, 1987.

[LT89] Nancy A. Lynch y Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[LVC+16] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, y Marko Vukolic. XFT: Practical Fault Tolerance beyond Crashes. En *OSDI*, páginas 485–500, 2016.

[Lyn96] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

[M+75] Gordon E Moore et al. Progress in digital integrated electronics. En *Electron Devices Meeting*, volume 21, páginas 11–13, 1975.

[Mar16] Will Martino. Kadena—the first scalable, high performance private blockchain. Whitepaper. *Available at* `http://kadena.io/docs/Kadena-ConsensusWhitePaper-Aug2016.pdf`, 2016.

[MB02] Petros Maniatis y Mary Baker. Secure history preservation through timeline entanglement. *arXiv preprint cs/0202005*, 2002.

[Mer78] Ralph C Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(4):294–299, 1978.

[Mer80] Ralph C Merkle. Protocols for public key cryptosystems. En *Security and Privacy, 1980 IEEE Symposium on*, páginas 122–122. IEEE, 1980.

[MNC12] Henrique Moniz, Nuno F Neves, y Miguel Correia. Byzantine fault-tolerant consensus in wireless ad hoc networks. *IEEE Transactions on Mobile Computing*, (1):1, 2012.

[Mos93] David Mosberger. Memory consistency models. *ACM SIGOPS Operating Systems Review*, 27(1):18–26, 1993.

[MR97a] Dahlia Malkhi y Michael Reiter. A high-throughput secure reliable multicast protocol. *Journal of Computer Security*, 5(2):113–127, 1997.

[MR97b] Dahlia Malkhi y Michael Reiter. Unreliable intrusion detection in distributed computations. En *csfw*, página 116. IEEE, 1997.

[MS02] David Mazieres y Dennis Shasha. Building secure file systems out of Byzantine storage. En *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, páginas 108–117. ACM, 2002.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[OL88] Brian M Oki y Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. En *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, páginas 8–17. ACM, 1988.

[OM14] Karl J O'Dwyer y David Malone. Bitcoin mining and its energy footprint. 2014.

[OO14] Diego Ongaro y John K Ousterhout. In search of an understandable consensus algorithm. En *USENIX Annual Technical Conference*, páginas 305–319, 2014.

[Ora03] Andy Oram. Peer-to-peer: Harnessing the power of disruptive technologies. *SIGMOD Record*, 32(2):57, 2003.

[Pow96] David Powell. Group communication. *Communications of the ACM*, 39(4):50–54, 1996.

[Pre00] I Present. Cramming more components onto integrated circuits. *Readings in computer architecture*, 56, 2000.

[PSL80] Marshall Pease, Robert Shostak, y Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[Rei94] Michael K Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. En *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, páginas 68–80. ACM, 1994.

[Rei95] Michael K Reiter. The Rampart toolkit for building high-integrity services. En *Theory and Practice in Distributed Systems*, páginas 99–110. Springer, 1995.

[SAB13] João Sousa, Eduardo Alchieri, y Alysson Bessani. State machine replication for the masses with BFT-SMaRt. 2013.

[SBBB12] William Stallings, Lawrie Brown, Michael D Bauer, y Arup Kumar Bhattacharjee. *Computer security: principles and practice*. Pearson Education, 2012.

[Sch84] Fred B Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems (TOCS)*, 2(2):145–154, 1984.

[Sch90] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[Sho99] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[ST87] TK Srikanth y Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

[TDP+94] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, y Brent B Welch. Session guarantees for weakly consistent replicated data. En *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, páginas 140–149. IEEE, 1994.

[Tsu92] Gene Tsudik. Message authentication with one-way hash functions. *ACM SIGCOMM Computer Communication Review*, 22(5):29–38, 1992.

[VCB+13] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, y Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2013.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[VOW94] Paul C Van Oorschot y Michael J Wiener. Parallel collision search with application to hash functions and discrete logarithms. En *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, páginas 210–218. ACM, 1994.

[Vuk17] Marko Vukolić. Rethinking permissioned blockchains. En *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, páginas 3–7. ACM, 2017.

[YMV+03] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, y Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. *ACM SIGOPS Operating Systems Review*, 37(5):253–267, 2003.

Este documento fue editado y tipografiado con LaTeX empleando
la clase **esi-tfm** (versión 0.20180917) que se puede encontrar en:
`https://bitbucket.org/arco_group/esi-tfg`