Our position on performance and scalability

iov42 Limited
Regina House
124 Finchley Road
London NW3 5JS
United Kingdom

Table of Contents

Introduction

Much has been written about the performance of blockchains and Distributed Ledger Technologies (DLTs). The generally prevailing opinion is that these technologies are often "slow" — with slowness here measuring the average time it takes for a transaction to be processed in such a way that it cannot be reversed or revoked, and the average throughput of transactions over time.

In reality, it's hard to make statements of relative performance when DLTs are extremely varied in their approaches to achieving consensus. Many attempted comparisons made are actually incomparable and are often unfair or tell us very little about the reasons for a given performance profile. Worse still, many comparisons will seek to compare a DLT to a non-blockchain database technology, which generally have vastly superior performance profiles.

As with all such analyses, the reader must consider the trade-offs made and what use cases were in mind when making those trade-offs. Looking at DLTs through this lens should help us consider exactly which DLT will be suitable in which situations.

For example, public (permissionless) blockchains favour security over performance. It's actually extremely difficult to reason about the performance of such networks when tens of thousands of individual nodes may be participating in parallel. Current generation Proof of Work mechanisms essentially reduce that enormously available parallelism down to a single node for each block. Such approaches also waste enormous amounts of energy as nodes compete to 'win' the next block, each doing the same 'work'.

Improvements to these mechanisms exist, such as Proof of Stake, which elect smaller committees of validators within the network. These approaches to consensus in public DLTs can improve performance and decrease wasted energy, but there are unproven assumptions regarding the security of such networks.

Permissioned DLTs provide an alternative set of approaches to consensus because a permissioned network starts with a different set of security assumptions — namely a network of nodes in which some trusted set up has occurred and the nodes "know each other". Once we move from a permissionless setting and into a permissioned one we can leverage many well understood approaches to scaling a distributed technology.

Choosing the permissioned setting forces us to think carefully about the suitability of different business models. It's through designing a scalable DLT that iov42 deliberately chose to leverage data centre architectures and conceptually the idea that a small number of independent "node operators" would choose to come together to solve a business problem in which trust and security are paramount.

In the remainder of this paper we'll examine the iov42 approach to building a scalable DLT.

Why existing distributed ledgers are slow

At their core, DLTs require a consensus mechanism. The purpose of which is to decide a global ordering of transactions within the network. This is a key problem in computer science and is known by several names, the most common probably being the State Machine Replication or SMR problem. The "problem" with state machine replication is ensuring that all nodes in the network agree on exactly the same order for events within the network.

Provided your network has one single node deciding the ordering of events then this problem is easily solved. Indeed this is exactly how Proof of Work operates — nodes compete to solve an abstract mathematical problem by randomly searching for answers to it. The node that gets there first chooses the next 'block' of ordered transactions in the chain.

The problem with this approach is that you're always limited by the speed of the network to choose that single node for ordering. In more traditional models, such as Paxos based approaches, a "leader" is elected and it's this leader that decides the order of all transactions. In these approaches we're limited by the throughput of that single leader node. Paxos must also elect a new leader should the current one fail and this process is often slow

because transactions cease to process until it is completed. You can think of Proof of Work as randomly electing a leader for each block.

The search for high performance and failure tolerant consensus mechanisms has a long and rich history in computer science. There are several "impossibility results" in consensus research that give bounds on what is and isn't possible in a consensus mechanism. The findings are essentially that no perfect consensus mechanism can exist and that all consensus mechanisms must trade off several competing concerns, in particular security vs. performance and scalability.

How distributed systems can be scaled for performance

Fundamentally, the performance of any computer system comes down to the hardware of the machine on which it executes. The simplest way to improve performance is to increase the processing capability of that hardware, through faster or more processors, memory, bandwidth and so on. This is normally referred to as vertical scaling or "scale up". The problem with increasing performance through vertical scaling is that we eventually find it impossible to make a single hardware instance faster or more powerful. Ultimately we're limited by the laws of physics, but first and foremost we'll normally be limited by simple economic factors.

Instead, for a long time the approach has been instead to horizontally scale the hardware or "scale out". This means adding more individual machines to a cluster of machines all solving a given problem in a distributed manner.

If we consider a DLT network with tens of thousands of nodes, then we could consider allowing each node to process transactions from users in parallel. This would give enormous performance, but there's no reason why the nodes would agree to the same ordering of transactions amongst them. This is where a consensus protocol becomes necessary (see above).

A solution to this might be to divide the workload amongst the nodes in some deterministic manner, for example, Node A could process all requests for account 123 and Node B could process all requests for account 456. Any requests to these accounts will always process through the same node and so for a given account a single node will always enforce an ordering. Another way to think of this is that we make each node a leader, but only for a subset of the problem domain. This approach is used heavily in highly scalable systems such as databases and event stream processors and is called 'partitioning' or 'sharding'.

In the scaling of a distributed system there are really only the two approaches of vertical and horizontal scaling. And due to the economics of building ever more powerful computers we find ourselves with only **horizontal scaling as the solution to building systems at massive scale**. Once we accept this we realise that each machine can be built on quite modest commodity hardware. With a sufficiently well designed network we can tolerate failures in this hardware to build not only high performance, but also highly resilient systems.

Our approach to architecture

Principles

Some core principles that have guided our approach to architecture and protocol design:

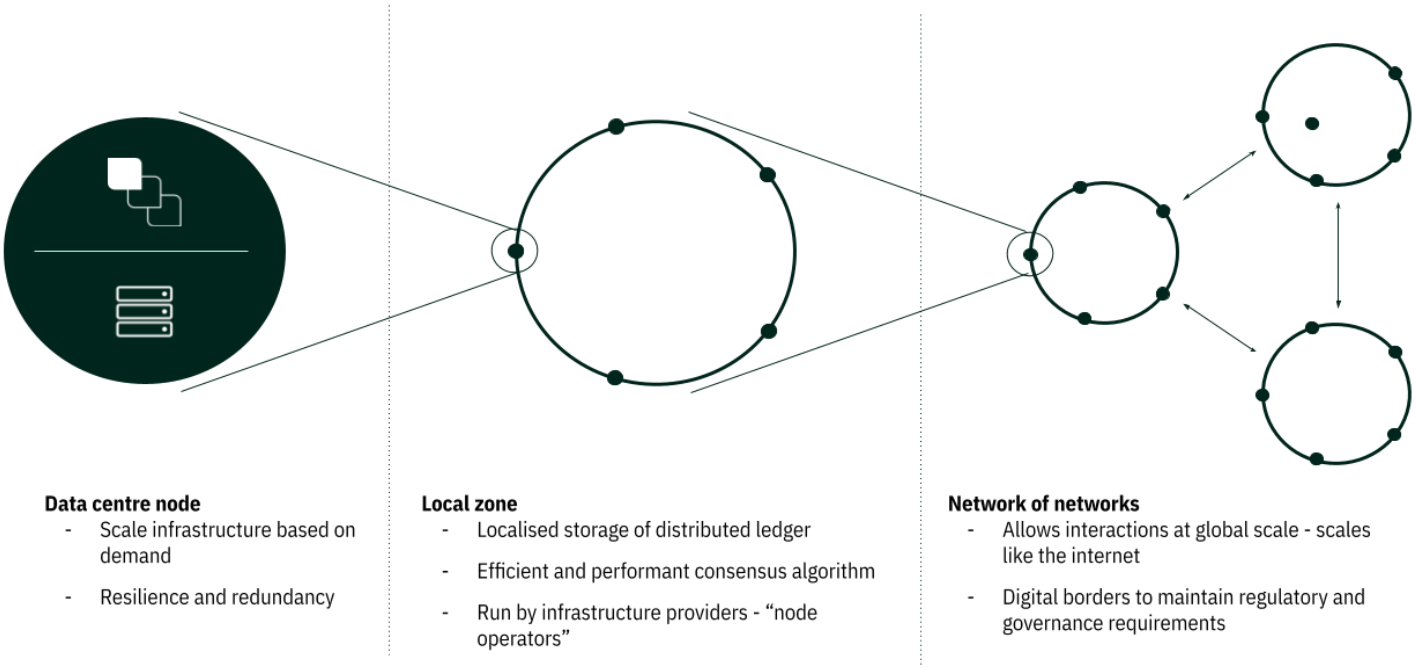
- We believe in permissioned DLTs for governments and enterprises, enabling them to form consortia to solve coordination problems that build trust
- Such organisations desire low latency and high throughput for their business cases
- Horizontal scaling approaches are how distributed systems can achieve high performance
- Governments and enterprises are used to running solutions at scale in data centres
- Technology exists to enable high scalability in data centre environments that we can leverage instead of reinventing
- Governments and enterprises exist in regulatory environments that require knowing who has access to which data and keeping that data private from anyone unauthorised

These constraints are very freeing in that they've allowed us to completely reconsider how a DLT can be built in order to serve the needs of governments and enterprises. In particular we aim to leverage as much low level data centre infrastructure technology as possible, so that the resultant stack is familiar and well understood by Site Reliability Engineers (SREs).

Deductions

From the above basic facts we've deduced an architecture that:

1. Reimagines a DLT node as a data centre with various differing components that can be scaled horizontally, independently and elastically in accordance with demand and desired performance characteristics.
2. As with other DLTs, multiple of these nodes come together to form a secure permissioned consensus group which we call a "Zone". The nodes come together to provide security to the whole Zone by replicating the data and ensuring state changes only occur via consensus.
    a. Members of a Zone share public keys with each other as part of a commissioning phase.
    b. Zones exist within a geographical or regulatory environment so that they de facto come under the laws, jurisdictions and regulatory regimes relevant to their location or use cases.
    c. The purpose of a Zone varies, but they exist to create a common infrastructure on which digital services can be built and in which users can leverage protocols and tools in order to have confidence in the ecosystem that they are using.
3. Zones may further interact with each other in order to transact in an "Interzone" manner, giving rise to a network of networks



Note that in this network architecture we depart strongly from the concepts of a public, permissionless blockchain in which there could be tens of thousands of nodes arranged in a purely peer to peer fashion. Instead we see many Zones that contain in the region of 5-9 nodes, but where these nodes are built on horizontally scalable data centre technologies in order to scale arbitrarily. By connecting these Zones together our architecture attempts to replicate the design of the internet, by allowing distinct networks to connect to each other — except in our case, the purpose of that connection isn't the routing of data, but the routing of digital representations of value, so that an 'Internet of Value' can emerge.

Design

Given the high-level conceptual architecture described in the previous section it becomes clear that in order to achieve a highly scalable and performant architecture we need to carefully consider the software that runs within nodes and the consensus protocol that runs between nodes. In particular we're keen to achieve the following properties:

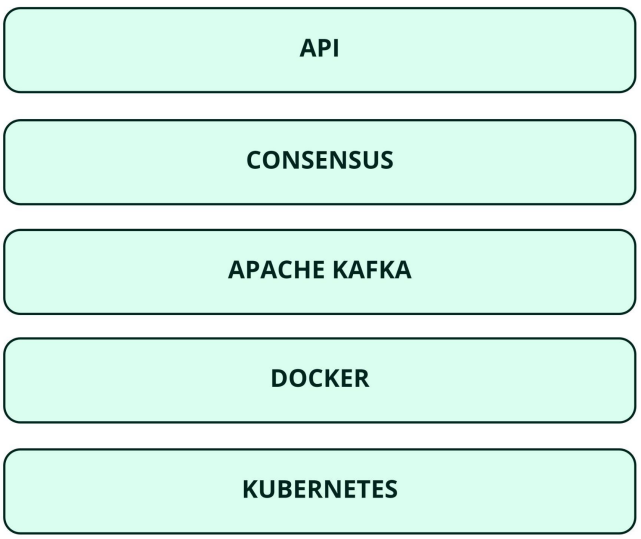
- Reliable
- Maintainable
- Resilient
- Performant

Reliability and maintenance

Our DLT is fundamentally different to many in the DLT space in that it has been designed to be run in multiple data centres using many standard cloud infrastructure components and tools. We made this decision precisely to bring a high performance DLT to the enterprise and government space, with the goal of making it easy to deploy and maintain — these are often blockers to adoption of other similar technologies.

Each data centre is run by a different stakeholder. These stakeholders come together to form a Zone that is able to agree, with high performance, on the outcome of every request to the network in a DLT model known as Proof of Authority. In this model, the nodes (data centres) are known to each other and choose to form a Zone in order to achieve their shared desired use cases.

Each data centre runs multiple components in a layered architecture that is conceptualised in the following diagram:



Logical architectural layers

Each data centre hosts the components within each layer, which we'll consider each starting from the bottom up:

- **Infrastructure Layers**
  These infrastructure layers are standard open source technologies used across industry to reduce the complexity of technologies. They are found in nearly all modern data centres and are the core of our strategy to build a consensus based technology using well understood infrastructure. We have consciously attempted to avoid re-inventing the wheel wherever possible so that our investment is in our application layer.

- **Kubernetes** — this is a low level data centre technology that is the industry standard for cloud and other data centre providers for running containerised workloads. The technology was originally developed by Google, is open source and is the modern standard for deployment of enterprise grade technologies. You can read more about Kubernetes here.

  Kubernetes simplifies maintenance by allowing Docker containers to be loaded and unloaded both manually and automatically. This allows operations engineers to scale and upgrade the technology as part of a regular update and maintenance schedule or in response to issues/incidents that may occur.

- **Docker** — is the industry standard for creating 'containers' for applications. Docker containers are scheduled for deployment by Kubernetes and are the fundamental execution components within our software architecture. You can read more about Docker here.

  The use of Docker containers works closely with Kubernetes as part of our maintenance strategy, by allowing operations engineers to load and unload containers as necessary. This can be part of the regular maintenance and upgrade schedule or in response to an issue or incident.

- **Apache Kafka** — is an extremely high performance open source streaming data technology that is used by tech industry titans wherever high performance is required. Kafka is used by organisations such as Netflix in order to deliver video content to customers and has been designed with resilience and maintenance in mind. You can read more about Apache Kafka here.

  Kafka was designed from the ground up to be highly maintainable and resilient. It does this by persisting messages between components to storage so that outages and upgrades can be performed without any data loss and through careful design, without the need for unnecessary downtime.

- **Application Layers**
  These layers compromise our primary intellectual property and industry innovations and are the secure application of our proprietary consensus architecture.

  - **Consensus** — this layer consists of several Dockerised components that allow the individual data centres to form a Zone. These components have various roles such as 'coordinators', 'validators', 'aggregators' and various others. They form a microservices style architecture that is able to scale horizontally and reach consensus between data centre nodes.

    These components can be upgraded and replaced according to a maintenance schedule and rely heavily on the above explained lower level infrastructure layers. Our entire architecture has been designed with data centres in mind and therefore maintenance is a core principle. We have invested significant amounts in making it possible for upgrades to the underlying protocol to occur without any data-loss and in a data-preserving manner. This means that breaking changes are extremely rare and in most cases will never happen.

  - **API** — our API layer is separated from the core application consensus layer in order to maximise maintainability. By ensuring this decoupling in our design we gain points of scalability, resilience and maintenance.

In practice maintenance is performed using Helm Charts that describe to Kubernetes which Docker containers should be deployed to a running environment. This makes it straightforward for operations personnel within a data centre to upgrade and maintain our technology when new versions are released.

Furthermore, by designing our technology as multiple microservice style containers, maintenance can be performed on a single component with laser precision. Through the use of technologies such as Apache Kafka it is possible to maintain these microservices without unnecessary downtime.

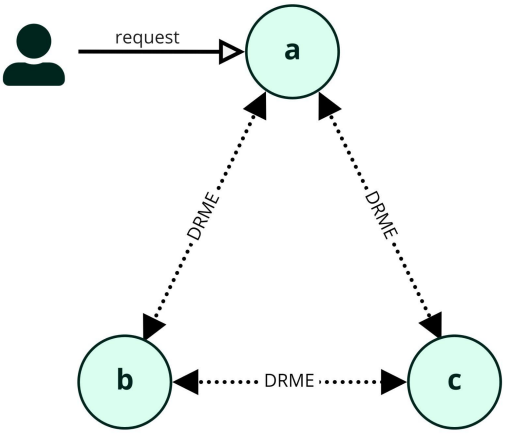
Resilience

Most of our technology choices have been made with resilience in mind. Our entire application layer is built on the principle of resilience, from the manner in which consensus is achieved in a Zone through to natural resilience properties of the infrastructure layers. We will consider resilience at each layer within the diagram 'Logical architecture layers' from the previous section:

- **Infrastructure Layers**
  Resilience starts with the infrastructural layers. Each of these layers adds properties to the overall application stack that bring about a highly resilient architecture.
    - **Kubernetes** — at its core this technology is designed to schedule and deploy Docker containers. Should a 'pod' fail then Kubernetes is able to automatically remove it and schedule a replacement. Kubernetes is the backbone of many of the world's most highly resilient architectures at the world's most valuable technology companies and is considered state of the art.
    - **Docker** — by containerising small microservice-like components we have been able to build a technology that achieves consensus while allowing containers to be taken online, offline and fail in a live running environment. A technology such as Docker works hand in hand with Kubernetes to bring about a highly resilient architecture.
    - **Apache Kafka** — was chosen as our underlying communication mechanism for several reasons, but its resilience properties are some of the most important. Kafka allows for the deployment of many parallel stream processors that automatically share the workload between them. Kubernetes can schedule more or less of these pods, live and on-the-fly and Kafka will automatically rebalance the workload between them. This creates a highly elastic and therefore resilient architecture.
- **Application Layers**
  In the infrastructure layers resilience is primarily about ensuring that the necessary number of each application layer component is able to run. At the application layer resilience takes on a fundamentally different meaning, which is more akin to: how can the Zone (network of data centre nodes) ensure that the consensus algorithm is fundamentally resilient.
    - **Consensus** — our patent pending consensus mechanism keeps the data centre nodes in perfect lockstep when it comes to the 'state of the world', through our Distributed Randomised Master Election (DRME) protocol. Changes are only allowed to occur when a simple majority of data centre nodes agree. Each node keeps an identical copy of the state of the world and so can be thought of as a resilient backup of the history of the zone. Assuming three nodes: **a**,**b** and **c** then DRME works in the following manner:

Incoming Request Phase
For the following phases, consider three data centre nodes: **a**, **b** and **c**. They are constantly exchanging control messages called DRME messages which are randomly generated noise (entropy).

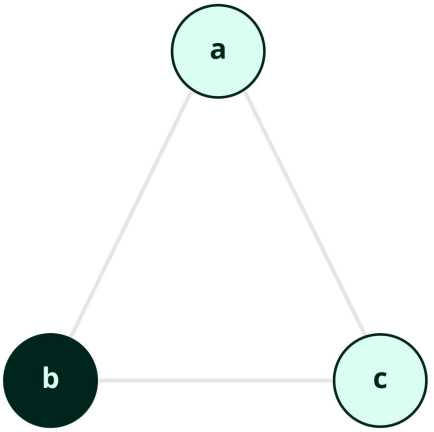
A user makes a request of the API layer hosted at data centre **a**.



- In this first phase of any incoming request, resilience is achieved by being written to Apache Kafka. This makes the request durable with respect to the node that receives it. The node could be taken offline at this moment, but the request is safely stored using the native durability guarantees of the underlying Kafka technology and will be processed once the node comes back online.
- DRME messages are constantly exchanged between nodes and also stored using Apache Kafka. They interleave the request messages that come into the zone with the API layers at each node and so therefore are resiliently stored with all of the incoming requests at every node.
- Incoming requests are cryptographically signed by the requesting user identity. These signatures are impossible to forge without access to the secured private key, making requests resilient against forgery attacks.
- A user could supply the same request to multiple nodes at once in order to de-risk the very tiny chance that a single node might go offline. Requests contain a unique identifier and so will not be processed twice.
- By containing a unique identifier selected by the user, the requests cannot be ‘replayed’ by an attacker, making them resilient to replay attacks.

Election Phase
The DRME messages are combined in a deterministic but completely unpredictable manner.

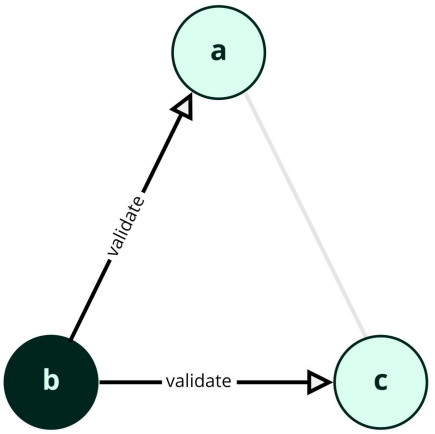
This combined DRME message is used to randomly elect one of the nodes to coordinate the request. All nodes will deterministically select the same coordinator node for each given request.



- The election process is entirely deterministic because each node can see the DRME messages and requests and the exact ordering of them stored durably within Apache Kafka. As the combination of these messages follows an agreed approach this means each node knows exactly which node is coordinating a given incoming request. This data is stored at each node and so is resilient to the failure of nodes within the zone — processing can always continue provided a majority quorum is available.
- Elections are per-request as unique components of the request are used as entropy in the randomised election process.
- A timeout mechanism exists within the DRME protocol that adds a further layer of resilience, allowing a request to be processed even if a given elected coordinator goes down during processing.

Validation Phase
The elected coordinator broadcasts instructions to each node for them to validate the various parts of a request.

On receipt of these instructions the nodes will reserve any state that might change as part of this request.



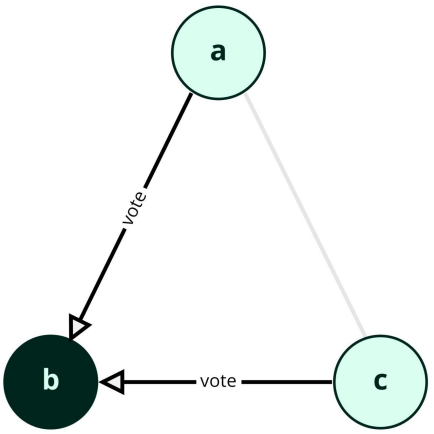
- The validation instructions are communicated over the Apache Kafka mechanism and so are durably written to storage in a resilient manner.

- When a node receives an instruction it immediately reserves any necessary state information. Messages are processed in sequence and so therefore state is locked before any subsequent messages are processed. This means that two requests attempting to affect the same state information cannot both simultaneously lock the same state making the mechanism resilient to the double spend problem.
- The state in the validators uses Kafka Streams which provides automatic handling should a validator fail. It redistributes the state and the processing to other live validators automatically.

Voting Phase
All nodes, including the node coordinating the transaction, generate votes on whether the request should be committed (success) or rolled back (fail).

These votes are sent back to the master coordinator for collation into a final outcome.

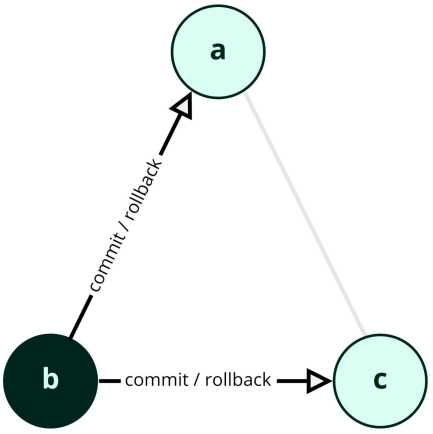
The master coordinator assembles a final 'proof' that justifies the outcome of the request.



- The nodes cryptographically sign their votes, so that the consensus mechanism is resilient to the forging of messages by an attacker.
- The master coordinator collates the signed votes and assembles them into a proof, which is a document containing all of the evidence, including signatures, that justifies the outcome of the request. The coordinator cannot forge these votes and so cannot influence the outcome of the request. The consensus mechanism is therefore resilient to a compromised coordinator data centre node.
- The master coordinator signs the proof so that it can be proven that the proof was generated by it.
- Each vote includes the signature of the previous committed transaction that referred to the entity being validated.

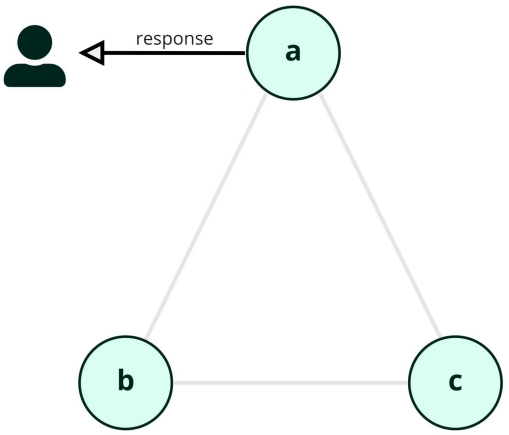
Commit / Rollback Phase
The coordinator node broadcasts the assembled proof to the nodes within the zone.

On receiving the proof all nodes either commit reserved data changes or roll them back, depending on the outcome of the proof.



- All commit and rollback messages are durably stored by the underlying Apache Kafka communication technology and so are available resiliently at each data centre node.
- The commit / rollback messages are cryptographically signed and so cannot be forged by an attacker and are resilient to such forgery attacks.
- The original unique request identifier is incorporated into the proof messages and so the consensus mechanism is resilient against replay attacks.
- Once nodes have committed or rolled back then the consensus protocol is completed.
- By bringing all the votes together it brings all the links to previous transactions together in the proof and in doing so creates a mesh of proofs. The signature of this proof will be referenced by subsequent transactions, extending the mesh.

Response Phase
As a final step after consensus the user is able to retrieve the status of their request and proof that justifies the outcome.



- Each node contains a copy of every proof generated within a Proofmesh™ which is akin to many blockchains woven together simultaneously. This data structure cannot be

modified or forged unless the private keys of all users and nodes are compromised. Because this is very unlikely the network is resilient to forgery attacks.

- The user can retrieve proof of their request from any available node, meaning that the network is resilient to nodes going offline.

- **API** — the API layer is decoupled from the consensus layer and connected via Apache Kafka. A data centre node can choose to run zero or many instances of this stateless layer making it resilient against load and attacks such as Distributed Denial of Service (DDoS) attacks.

Performance

One of the key goals behind our design is to increase performance in terms of requests per second, reduce latency in processing an individual request and the reduction of energy consumption per request. Many of our design decisions with regard to performance also have the knock on effect of resilience in the face of failures that ensure a Zone can recover as fast as possible.

There are two key design decisions that enable high performance and resilience:

1. **Proof of Authority** — is a network in which a small number of nodes are the 'trusted authority' with regard to which changes are allowed in relation to the data stored. This is in direct contrast to a fully public network that relies on Proof of Work. In a Proof of Work network, anyone can run a node and so in order to guarantee that no one node can control the network, a lottery-like mechanism is employed where nodes compete in a randomised puzzle to be the next node to add a block to the chain.

   Proof of Work is widely reported as being hugely wasteful with regard to the energy consumed. These networks are also poorly performing in terms of both throughput and latency; however this is a feature of these networks and not necessarily a bug.

   Our technology starts from a fundamentally different set of foundational assumptions and is therefore not constrained in the same way. We assume:

   a. That a small group of organisations can come together to govern and implement a use-case in a consortium that is aligned either geographically, regulatorially or along some other axis of cooperation.

   b. That this consortium has a shared set of users that trust the consortium to hold each other to account.

   c. That the members of the consortium have the capability to each run a data centre node within the consortium's zone.

   Once the concept of a consortium is defined as a starting axiom it follows that these data centre nodes already trust each other and can validate messages from one another using cryptographic signatures. Adding and removing nodes from a Zone is unlike a public blockchain network and is a relatively rare event. These Zones are long lived and legal agreements may be in place between the node operators in the consortium with regard to Service Level Agreements (SLAs) and the model of governance employed within the Zone.

   By using cryptographic signatures to validate messages within the consensus protocol, our DLT is able to perform extremely fast as there is no "wasted computation" in order to secure the network. Similarly, by not wasting computational resources and only doing the work necessary, power consumption is kept to an absolute minimum and the sustainability profile of our DLT is very compelling when compared to others.

We have performed calculations against our current Amazon Web Services deployment to determine the energy consumption requirements. Our current deployment requires between $2.6x10^{-7}$ and $1x10^{-6}$ kilowatt hours per transaction, depending on the load on the compute resources. This compares well with existing solutions such as Hyperledger and is multiple orders of magnitude better than Ethereum and Bitcoin. We have performed only limited performance tuning / optimisation so far on the DLT and this setup. When we increase performance the number of transactions per second will increase on the same hardware and hence decrease the per transaction energy consumption.

2. **Horizontal Scalability** — as described above, each node within our network is in fact a data centre that runs multiple microservice style components connected together using a streaming data technology called Apache Kafka.

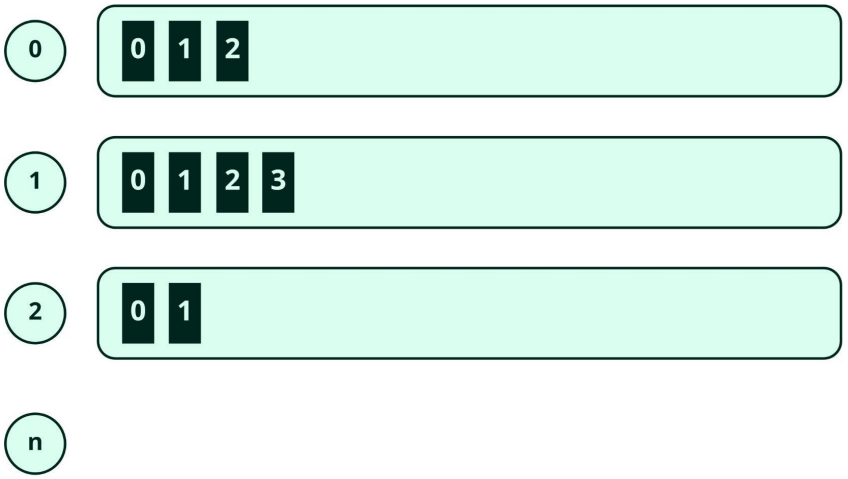
   The choice of Apache Kafka is key as it provides a highly resilient and highly performant underlying communication infrastructure trusted and used by the world's largest technology companies.

   Kafka achieves this through the concept of a partitioned log of messages. A message log is simply a queue of messages to be processed by the microservices within your architecture:

   In this model a service will process messages one-by-one starting with the head of the log. New messages are appended at the tail of the log. This model guarantees the order of processed messages meaning that all consumers see all messages in the same order, greatly simplifying the consensus protocol down to an algorithm called a two-phase commit.

   However, the performance is constrained by how quickly you can process each individual message and is not dissimilar to the performance bottleneck of public blockchain networks, in which work is done in series rather than in parallel (although nodes compete in parallel, only one, the winner, can add a block at a time). Also, this model adds a single point of failure — if only one instance of a service can process this log, then it crashing causes all work to cease until it can be recovered.

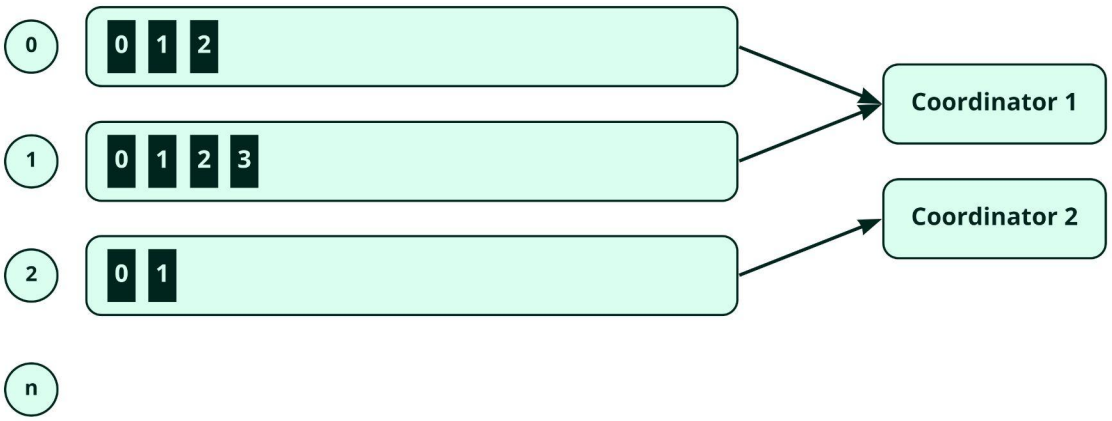
   For these reasons Apache Kafka introduced the concept of a partitioned log. This splits the above log into n-many individual partitions. Each partition is processed in parallel and the overall performance is limited by (a) how many partitions exist, (b) how many underlying brokers manage those partitions and (c) how many stream processors are sharing those partitions. A partitioned log can be conceptualised in the following manner:



Each partition is numbered individually and provides exactly the guarantees as with a single partition above, however now the workload is shared amongst the partitions. Care has to be taken when designing a protocol using this model that related messages are always assigned to the same partition, so that related messages are processed in the same order and it is always guaranteed to be the same. Apache Kafka makes this simple through the use of a 'partition key', which allows related messages to always end up in the same partition.

Apache Kafka supports up to 200,000 partitions per cluster, which allows for an almost embarrassingly high level of parallelism in processing. In practice there is a tradeoff to be made in choosing the number of partitions because with every partition comes a small overhead in processing.

Partitions are processed by the 'stream processors' in an architecture. In the case of our DLT the primary components building consensus are known as 'coordinators' and 'validators'. The number of each of these components can be scaled elastically and dynamically, for example:

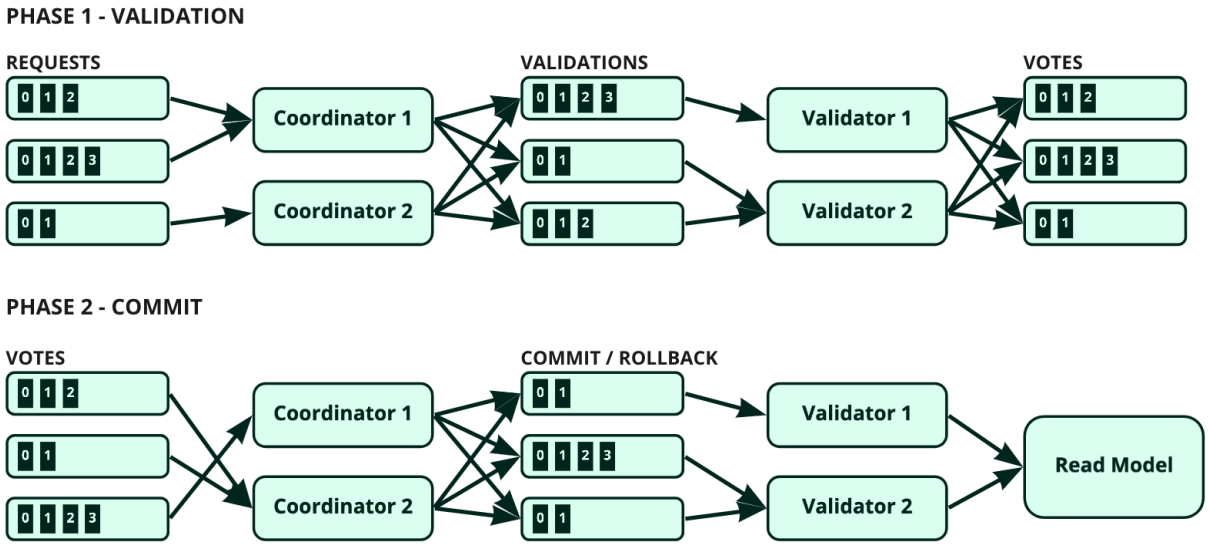


Each coordinator is assigned partitions to process. If we dynamically add more or less coordinators then the processing is automatically rebalanced amongst those available to make the best use of the available coordinators. Should a coordinator crash, then another one will be assigned its partition and continue processing where the crashed coordinator left off. This results in a high level of resilience and rapid recovery in the face of failures.

This design pattern is a horizontally scalable architecture, in that we add more processing nodes (normally depicted horizontally on a diagram) in order to increase throughput and resilience. This also gives

complete control to node operators to decide exactly how many resources in terms of compute and storage they allocate to their data centre, giving control over costs.

Our DRME consensus protocol works on the basis of a two-phase commit and the flow between partitioned logs and processing nodes looks a little like the following:



In the first phase of this flow, requests are picked up by coordinators for processing. Recall that the coordinating node is randomly assigned via the DRME protocol described previously. Coordinators split these requests into a number of individual validations that are required. For example, an individual request might specify that a unique asset representing an individual product be swapped for some amount of Euros between two identities within the zone. Validations would need to be carried out to ensure that both identities currently possess the assets in question, that their signatures match their public keys and any other aspects with respect to permissions to perform the transaction.

The validators pick up the validation requests created by the coordinators and lock any assets necessary. This locking prevents any conflicting operations occurring while the transaction is in progress. The validators issue their votes on whether they believe the validation is successful or not and write these to the votes log.

In the commit phase, the same coordinator that started a request tallies the votes from each node in the network. They create a cryptographic proof and link it into the Proofmesh™ so that it becomes tamper-evident. These proofs are then written to the commit / rollback log.

The same validators pick up these proofs and either commit or rollback their locked state changes in accordance with the outcome of the proof. Changes then flow downstream to a read model that the API uses to serve read requests.

Conclusion

Our dual approach of Proof of Authority built on top of a horizontally scalable partitioned collection of data streams guarantees high availability in the face of failure and an elastically flexible scalability model that increases throughput while minimising latency and energy consumption.

Using our DRME consensus mechanism we're able to load-balance the coordination of requests in a deterministic yet securely randomised manner.

Our reliance on extremely well understood data centre infrastructure technologies such as Docker, Kubernetes and Kafka ensures we leverage the many thousands of person-hours of research and development applied to these products to build highly reliable and scalable distributed systems.