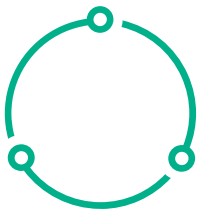


portus

A multi-chain open, trustless, permissionless API gateway

PORTUSNETWORK.ORG

19 MAY 2021

V 1.0.7

Contents

Introduction

APIs are used today by almost all applications, both web, and mobile. They provide social media services, location services, weather services, stock prices, language services, payment services, banking services, and much more. Companies and individuals usually provide APIs either for free or in exchange for a fee. This is great for software developers since they can build applications more efficiently without creating everything themselves. APIs are similar to Lego sets, where developers choose only the pieces they need to develop their applications.

While all of this sounds wonderful for traditional application developers, things are still at an early stage for dApps and Web 3.0. All fantastic APIs that are currently available out there are not compatible with these new, emerging technologies.

Blockchain Oracles

In Greek mythology, oracles were spiritual beings, sources of infinite wisdom and knowledge. A well-known oracle is the Oracle in Delphi. These oracles often provided information to people when consulted for critical decisions.

Blockchains, by design, do not have a way to access information outside of the chain because they cannot access data outside their network. Oracles aim to solve this exact problem: to allow smart contracts to interact with the outside world securely.

In the blockchain world, an oracle is a one-way digital agent that finds and verifies real-world data and submits it to a dApp in a secure way. An oracle is not the source of data but the layer that interfaces with off-chain data sources and the blockchain. It's a middleware that fetches information from off-chain sources and adds it to the blockchain. With oracles, smart contracts have a pathway to interact with data outside of the closed blockchain environment.

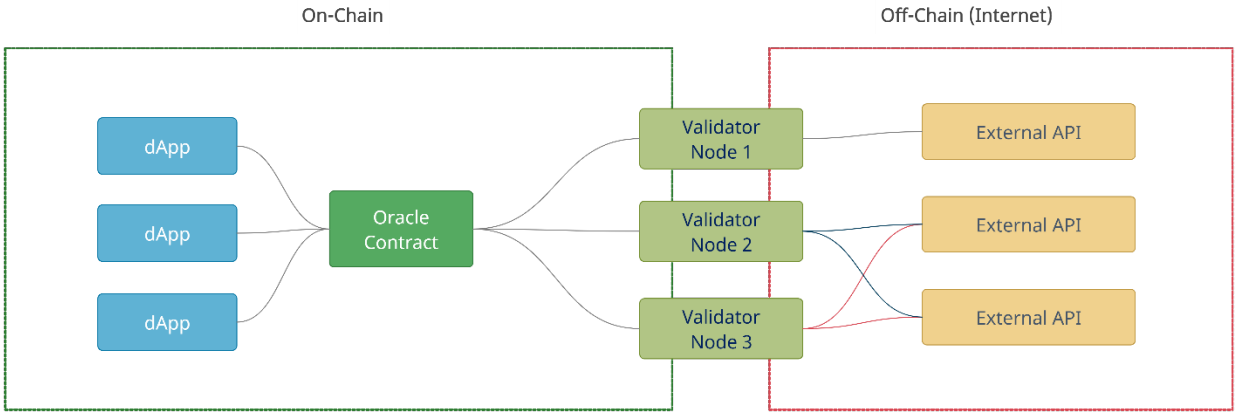
Oracles are defined by two main factors: type and direction. The oracle can either bring data inside the blockchain (inbound) or inform an entity outside the blockchain about a specific event (outbound).

Inbound oracles will provide smart contracts with data from the external world. An example use case is automated Ethereum sell orders based on the current price. Another example is gambling dApp payout based on a real-world event.

Outbound oracles use internal blockchain data to trigger an external event. An example use case is an external lottery payout based on the last published block.

Inbound Oracles

Until now, data oracles could provide data to dApps using an approach that involves a man-in-the-middle solution, and one of the best oracles is Chainlink. There are others like Band Protocol and Umbrella, but they all share the same principles. There is always a node that sits between the API provider and the dApp that requires data. It adds an intermediary in the process, which might not be needed. In a classic oracle network, the cost of everything is continually rising. In recent years, Chainlink has become the most extensive oracle network and is gaining a monopoly over data feeds, creating centralization. In Chainlink and similar oracles, validators don't know what data is provided to oracles. They just trust the data source but instead are punished for providing flawed data. The data feed itself has no penalty for doing so.



On traditional data oracles, requests are distributed across both oracles and data sources

PORTUS builds on the API3 proposal to allow API providers to run their on-chain nodes. dApps would talk directly to the API providers without requiring an intermediary. This approach creates competition that will lower inflation in the long run, promotes decentralization, and provides a way for data providers to self-manage without needing an intermediary to do so.

With the growth of the DeFi economy, a wide range of dApps must source trustworthy and reliable data. Since each oracle will own both the data and the services being provided, it will increase decentralization and allow data feeds to be curated transparently, an essential consideration for DeFi applications.

Data feeds provided by traditional data oracles only appear more decentralized than they are. When the data feeds do not provide transparency regarding the source of their data, one can no longer evaluate data feed integrity and is obligated to trust the governing entity, which is typically a validator node.

But without the middlemen to vouch for the data feeds, how can a dApp know the data it uses is safe and accurate? A traditional oracle uses a consensus algorithm that is susceptible to a specific

ratio of dishonest actors, as we find in all consensus mechanisms. A group of malicious actors can work to alter the outcome or even control it completely. A malicious actor can even fake multiple node operator identities, building a track record of honest operation, preparing the ground for a Sybil attack.

Data oracles must be accurate and reliable, but in the end, they are middlemen. dApp consumers interact with the middlemen, and the middlemen must ensure constant availability and reliability. You can find more oracles providing a single data feed to ensure redundancy, which is not always needed because it's provided by the API providers themselves, having a good reputation to maintain. Middlemen do not provide additional security at the data source level. Instead, they struggle to decrease vulnerabilities induced by using these third-party oracles.

Oracles are trying to solve in a complex manner a simple problem: to pull off-chain data and serve it to on-chain dApps. They are the equivalent of traditional APIs in the blockchain world, and their job is simple: deliver data to a consumer.

The current design of oracles is unnecessarily complex and expensive in the end. An ideal solution would be to allow dApps to use the original APIs directly, cut out the middlemen, excessive fees, and simplify the entire paradigm. Why pay an oracle a lot of money when you could pay the actual API provider for the real service? Even if an oracle charges a few pennies, it will still add up over time. Also, wouldn't it be better if the data comes directly from the source and not from an anonymous node?

This is what PORTUS is proposing. An open, trustless, permissionless network of genuine API providers that monetize their services directly to dApp consumers, opening the door to a new wave of blockchain use-cases and adoption.

Outbound Oracles

We now know that Inbound Oracles provide verified data feeds to on-chain dApps. But what about use cases where dApps need to trigger an external event, not just fetch data. Some examples would be initiating a bank account payment in a Web 3.0 dApp, selling a bond when a specific price is reached, or clearing an insurance claim when the payment is confirmed. These types of use-cases require dApps to tell the outside world something happened. They are known as outbound oracles.

In the same way as Inbound Oracle, Outbound Oracles need to guarantee several things to make them prime actors in a blockchain world:

- transactionality – a guarantee that actions are executed;
- traceability – actions can be audited and traced;
- security – actions must be secure;

Adoption

Due to the nature of blockchain technology and the fast pace at which it's evolving, providers might have difficulties onboarding and operating an oracle node. We selected a few of the concerns identified by other oracle providers:

API providers are not familiar with blockchain technology. Therefore, they typically cannot operate an oracle node with in-house resources. Portus does not require any specific knowledge to operate. The Portkey node needs a simple configuration file to start providing your API to blockchain dApps.

Running an oracle node needs maintenance and specialized people to maintain them, reducing scalability, increasing costs, and hurting adoption. The Portkey node does not require any day-to-day maintenance, such as updating the operating system or monitoring the node for availability. The node can run in cloud serverless and containerized environments. It's a stateless component, which makes it resilient against any problem that can cause permanent downtime and require intervention.

There is also a financial problem. Operating an oracle node requires a complex infrastructure, either on-premise or in a cloud. Unless one is guaranteed future profits, operating such a node might not be financially feasible. PORTUS allows its Portkey node to be run in cloud infrastructures that charge for on-demand resources, paying only for the resources it consumes. Once you have incoming requests, you start to consume resources and pay the infrastructure costs.

Portus built its protocol for easy adoption, especially from the API provider side. Current oracle solutions constrain providers to accept payment for their services in the native oracle cryptocurrency (e.g., LINK, BAND, etc.). However, due to compliance, accounting, and legal reasons, accepting cryptocurrency as payment is not an option for the vast majority of providers.

Portus allows providers to keep accepting payments as they are today. The PORT token is not involved in API service subscriptions. Providers are free to use any currency they want, and the subscription agreement between the provider and the consumer is made off-chain.

All transaction costs related to dAPI usage will be paid by the consumer. The consumer is usually proficient with blockchain technologies and has the required know-how and organizational capabilities to handle cryptocurrencies.

This is not currently possible with existing oracle solutions, making Portus a solution that can be adopted on a large scale.

Stateful vs. Stateless

Current dAPI providers invest significant resources to build highly-available infrastructures. An API gateway must not have single points of failure that may cause downtime or complicate the deployment infrastructure. Existing solutions that use third-party oracles depend on over-redundancy at the oracle level, which results in high costs. Also, dApps are evolving past the single API-call scenario and need more complex interactions with the outside world. With currently available oracle, this can be solved by performing multiple API calls. This approach has two main problems: transactionality and cost.

Transactionality

In computer science, transactions are intended to guarantee data validity despite errors, power failures, and other mishaps. For example, a bank transfer from one account to another involves operations like debiting one account and crediting another in a single transaction.

Transactions are often composed of multiple statements. Atomicity guarantees that each transaction is treated as a single "unit" that either succeeds or fails completely: if any of the statements in a transaction fails to complete, the entire transaction fails, and the system is left unchanged. An atomic system must guarantee atomicity in each situation, including power failures, errors, and crashes [1].

dApps by their nature cannot guarantee atomicity. Off-chain interactions are asynchronous and inherently indeterministic. The order of oracle API call fulfillments cannot be controlled because it depends on the availability, reliability, and response time of external systems.

Transactionality needs to be guaranteed by the API gateway in a way that's verifiable, and this is one of the main features of Portus.

Cost

dApps usually rely on smart contracts to implement their functionality. Smart contracts run in sandboxed virtual environments and consume resources for every executable instruction. These resources have a cost, and the larger the number of resources used by a smart contract method, the higher the cost will be for the caller. Implementing logic to orchestrate external interactions in a smart contract will be expensive and not feasible. Such orchestrations would also need to handle transactionality which will also add a lot to the cost.

With Portus, all external API calls can be packaged in off-chain atomic operations, preserving transactionality and keeping costs under control.

The Portus Oracle

Overview

Now that we know the significant problems connecting external APIs to on-chain dApps, let us look at a new approach that solves these problems more effectively than the currently available oracle-based solutions.

Portus is an open, trustless, permissionless API oracle. It is both an Inbound Oracle and an Outbound Oracle with features designed to ensure information validity and action traceability:

- **Open** because it is built from open-source software and made available as open-source software for full transparency.
- **Trustless** because the network allows participants to interact publicly or privately without requiring a trusted 3rd party.
- **Permissionless** because anyone can participate without authorization from a governing body. This is valid for both consumers and providers.

Portus is first a full-scale API Gateway. It's an API management tool that sits between a client and a collection of 3rd party services. An API gateway is a reverse proxy that defines API endpoints that aggregate various external services and return an aggregated result. API gateways are a standard in the enterprise landscape. It's common for them to handle everyday tasks such as user authentication, rate limiting, and statistics.

One component of the Portus network is the Portkey node. It's a piece of software that is both an Inbound and Outbound Oracle and sits at the API provider's premises.

As an Inbound Oracle (or Data Provider), we simplify the entire process, making the data provider itself an oracle node. This will make the middle layer obsolete, which is a tremendous achievement that solves many of the problems traditional data oracles face in terms of redundancy, performance, and scalability.

As an Outbound Oracle, the Portkey node acts as an on-chain information source for API providers. API providers can use blockchain data to trigger actions and trigger events in a way that's consistent with all blockchains Portus supports.

The API provider must have a reputation to keep, guaranteeing his commitment and notoriety. It must have a public identity, a description of its services, and price plans for the API services provided. **There are no middlemen in Portus**. The consumers are interacting directly with API providers. If the provided API or data is flawed, it should be immediately known, and there must be repercussions for the entity providing the service.

In oracle solutions, the middleman (validator) node is punished, but the data provider can continue providing false data with no penalty. And because nodes in traditional data oracles are

anonymous, no one ever knows which node was involved in delivering bad data. With Portus, the data providers are solely responsible for the truthfulness of their data.

This removes the possibility of "oracle bribing" in a cost-effective manner. Chainlink also solved the oracle bribing problem, but the solution they have used is prohibitively expensive. They designed their network for multiple nodes to deliver truthful data, but using numerous nodes can become very expensive over time.

In Portus, each data provider can establish an insurance deposit to vouch for his reputation and compensate dAPI consumers for quantifiable losses. The provider will have the option to alter the value of the insurance fund at their discretion, providing the fact there are no incurring penalties. Reducing the value of the insurance fund will negatively affect his reputation.

The API Gateway

The most basic function of an API Gateway is to accept a remote request and return a response. But real life is never that simple. Consider the various concerns when hosting large-scale APIs:

- the granularity of APIs provided by 3rd parties is often different than what clients need. 3rd parties provide fine-grained APIs, which means that clients need to interact with multiple services. For example, a client needing the details for a product needs to fetch data from numerous services.
- different clients need different data. For example, the desktop browser version of an application might require more data than the mobile version.
- the location and number of service instances changes frequently
- services use many protocols, some of which might not be web-friendly.
- APIs must be protected from overuse and abuse by using authentication and rate-limiting.
- adding new APIs and retiring others must be transparent to clients who want to find all the services in the same place.

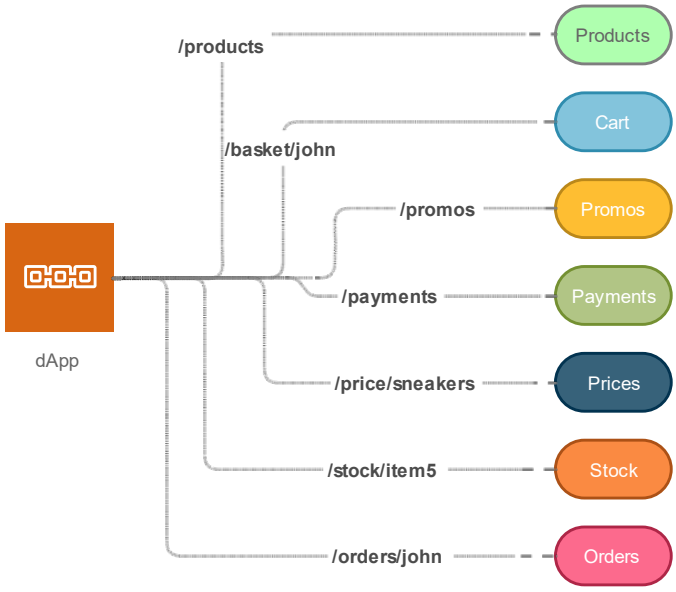
The challenge is to offer consumers a simple, consistent experience in the face of all this complexity, decoupling the client interface from an underlying implementation. When a consumer makes a request, the API gateway breaks it into multiple requests, routes them to the right places, produces a response, and keeps track of everything.

Using an API gateway has the following benefits:

- Reduces the number of requests/roundtrips. For example, the API gateway enables consumers to retrieve data from multiple services with a single round-trip. Fewer requests also mean fewer costs for dApps and improved user experience.
- Translates from a "standard" public web-friendly API protocol to whatever protocols are used internally.
- Provides the optimal API for each client.

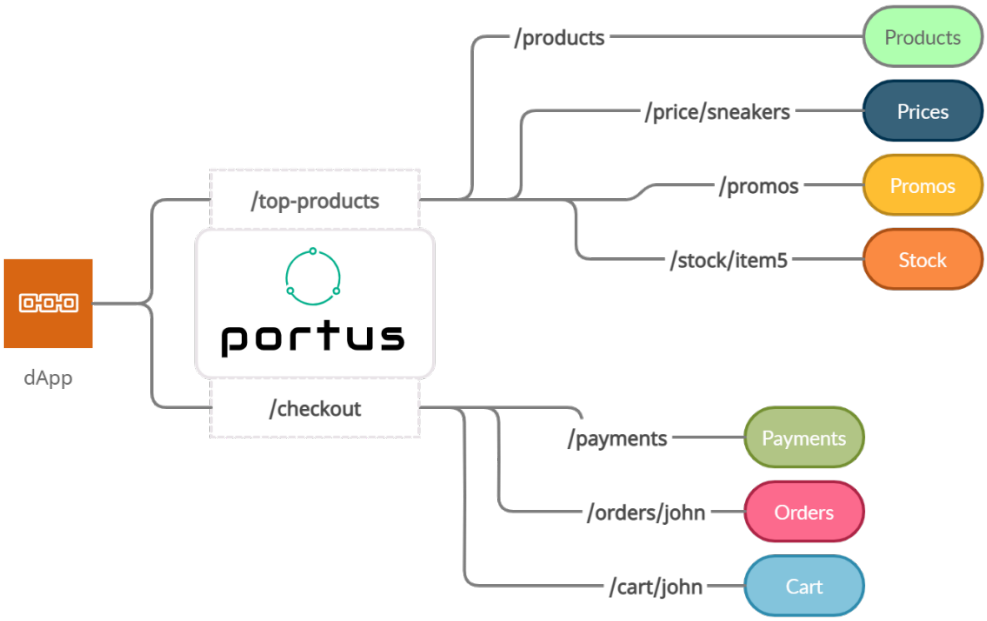
The API Gateway handles some requests by simply routing them to the appropriate backend service. It handles other requests by invoking multiple backend services and aggregating the results. To minimize response time, the API Gateway should perform independent requests concurrently. Sometimes, however, there are dependencies between requests. The API Gateway might first need to validate the request by calling an authentication service before routing the request to a backend service.

In a traditional environment, a typical API call flow would be similar to the one depicted in the image below.



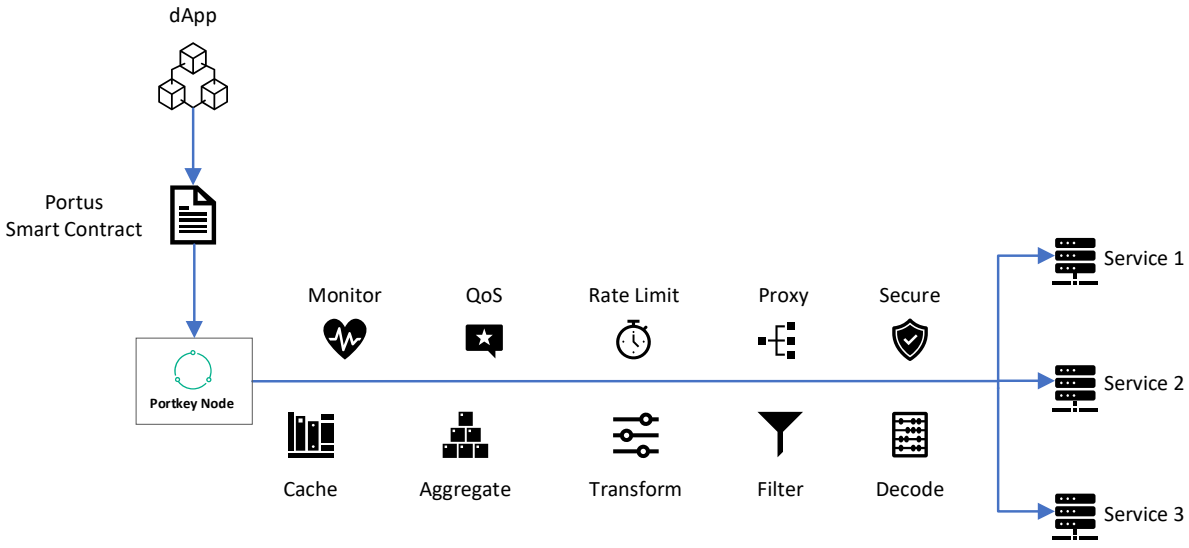
In this scenario, an application must know the location and details of each service, like host, port, request params, and response format. All API calls need to be treated independently for errors, and calling many APIs means the consumer will pay a lot of gas fees.

An API gateway resolves this problem by aggregating API calls into larger chunks. This means improved transactionality and speed, better error handling, and significantly lower costs.



Portus is a pure blockchain API Gateway that interacts with all your external backends providing consumers a single interface. It improves response times, saves bandwidth, delivers a better user experience, and saves developers precious time. With Portus, you can merge many calls into one, transform data, secure the transport, filter fields, shrink responses, throttle connections, establish quota usage limits, decode, enrich, and much more.

The API Gateway's core feature is to create a dAPI that acts as an aggregator of many external APIs into single endpoints, doing all the heavy lifting for you: aggregate, transform, filter, decode, throttle, authenticate, and more.



Portus needs no programming because it offers a declarative way to create endpoints. It is well structured, layered, and open to extending its functionality using a plug-and-play system for middleware components developed by the community or in-house.

It offers the following features:

- Aggregates the information from many sources
- Transforms responses and allows you to group, map, rename and delete fields
- Filters and shrinks responses, hiding or removing unwanted information
- Throttles connections against the gateway itself or backend services
- Protects your consumers with circuit breakers and implements security measures
- Speaks different encoding formats and protocols
- Fine control of transforms, validations, and filters

From a security point of view, Portus has the following features:

- User quota
- Support for SSL
- OAuth client credentials grants
- Restrict connections by host
- HTTP Strict Transport Security (HSTS)
- Clickjacking protection
- HTTP Public Key Pinning (HPKP)
- MIME-Sniffing prevention
- Cross-site scripting (XSS) protection

From an operations point of view, Portus is straightforward to use. It only requires you to create a configuration file that defines behaviors and endpoints.

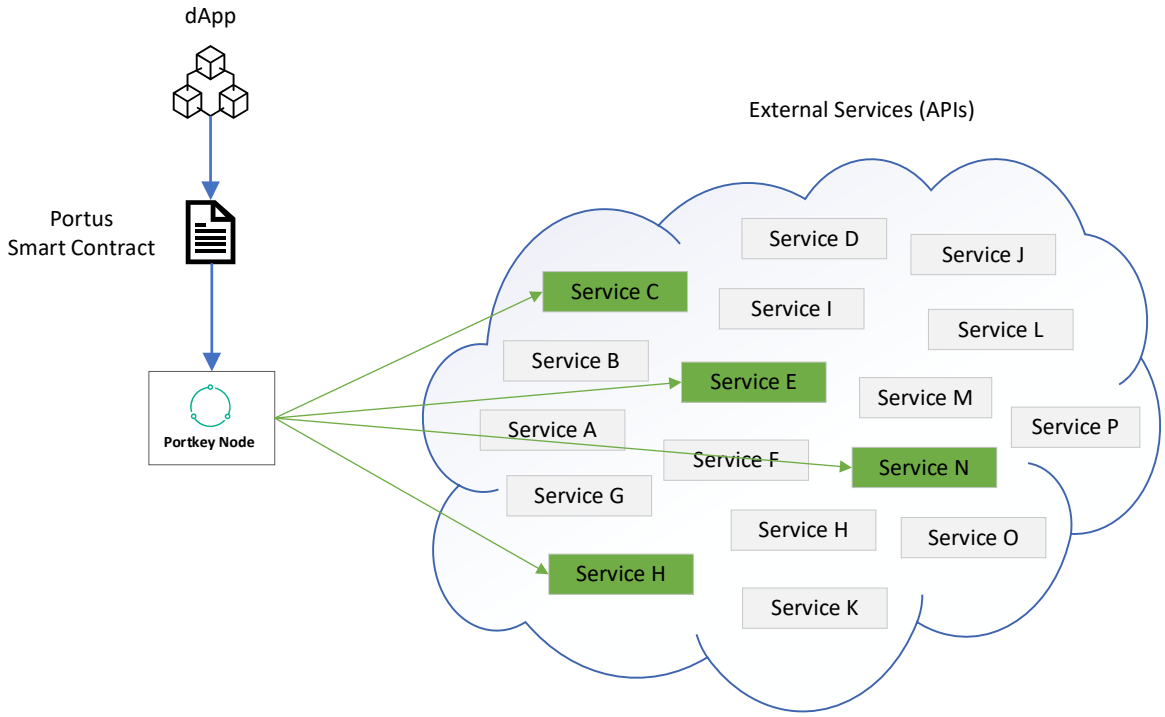
Transactionality

In distributed environments, some external services will inevitably fail. This is an issue for all distributed systems whenever one service calls another service that responds slowly or unavailable. Portus controls the interactions between distributed services by adding latency and fault tolerance logic. It does this by isolating access points between services, preventing cascading failures, and providing fallback options to improve your system's overall resiliency.

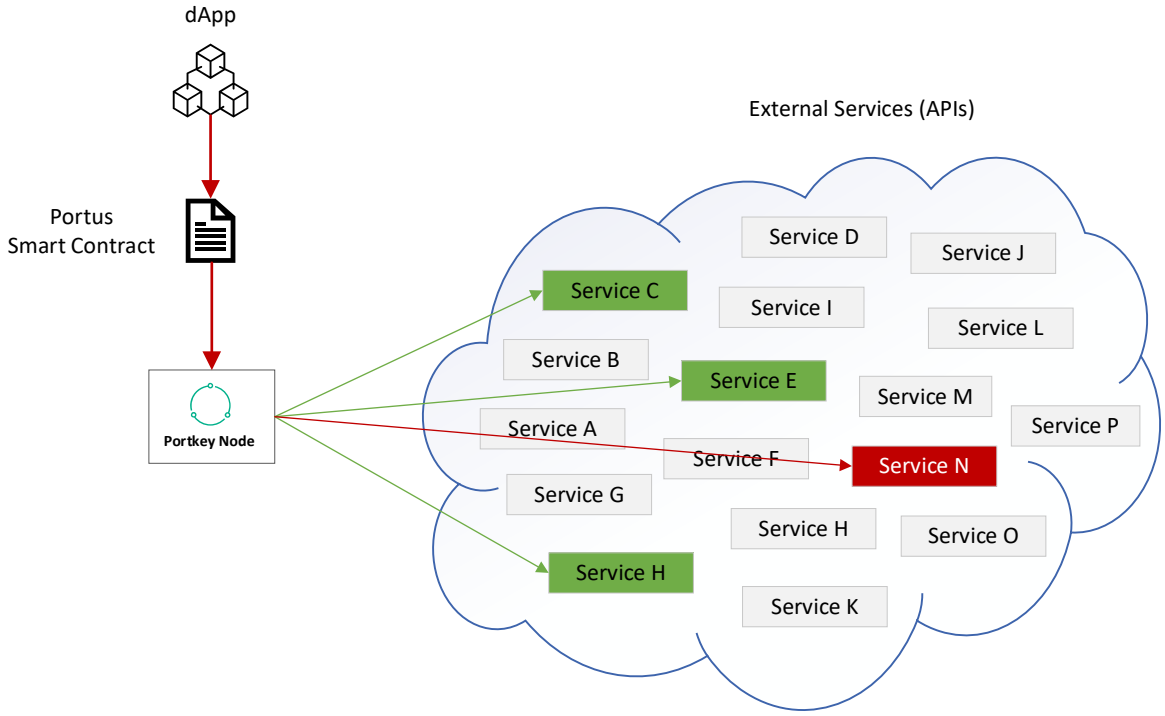
Complex dApps can have dozens of dependencies, each of which will inevitably fail at some point. If the dApp is not isolated from these external failures, blockchain transactions can fail, leading to unforeseen consequences. Even when all external services perform well, the aggregate impact

of even 0.01% downtime on each of the external services equates to hours of downtime in a month if you do not engineer the whole system to be resilient.

When everything is healthy, the request flow can look like this:



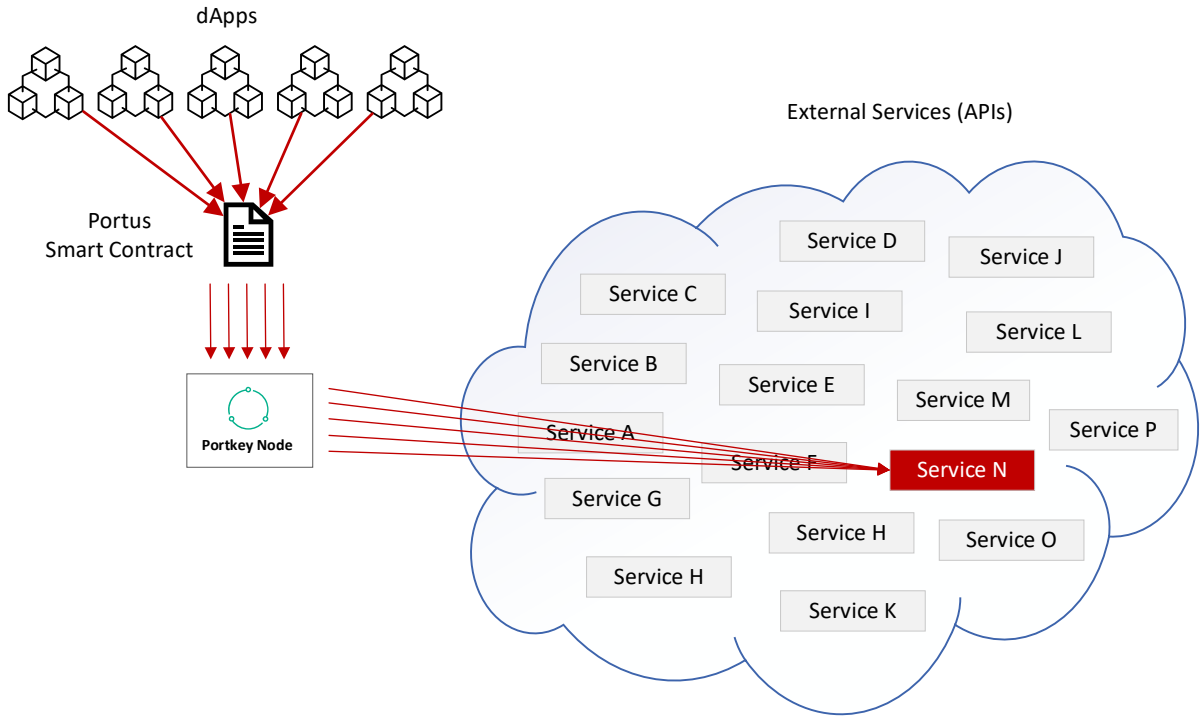
When one of many backend systems becomes latent, it can block the entire user request:



When you have more requests per second, all request threads can block for seconds.

With high volume traffic, a single backend dependency becoming latent can cause all resources to become saturated in seconds on all servers.

Every point in a system that reaches out over the network is a source of potential failure. Moreover, interactions between services can also result in increased latencies between, which fill up queues, threads, and other system resources, causing even more cascading failures across the entire system.



These problems become worse when calls are made to external APIs that are "black boxes", where implementation details are hidden and can change at any time. Even worse are transitive dependencies that perform potentially expensive or fault-prone network calls without explicitly invoking the original application.

Network connections fail or degrade. Services and servers fail or become slow. New service deployments can change behavior or performance characteristics. External APIs might also have bugs.

All these are sources of failure and latency. They need to be isolated and managed so that a single failing API call can't bring down the entire system.

Portus makes use of the Hystrix library to:

- Prevent any single external resource from using up all node user threads.
- Shed load and fail fast instead of queueing up indefinitely.

- Provide fallbacks wherever possible to protect consumers from failure.
- Use isolation techniques (such as the circuit breaker pattern) to limit the impact of any single failure.
- Optimize the time to recovery using immediate propagation of configuration changes in most aspects of Portus. This allows a node operator to make real-time operational changes with no downtime and fast feedback loops.
- Protect against failures in the entire dependency graph, not just network traffic.

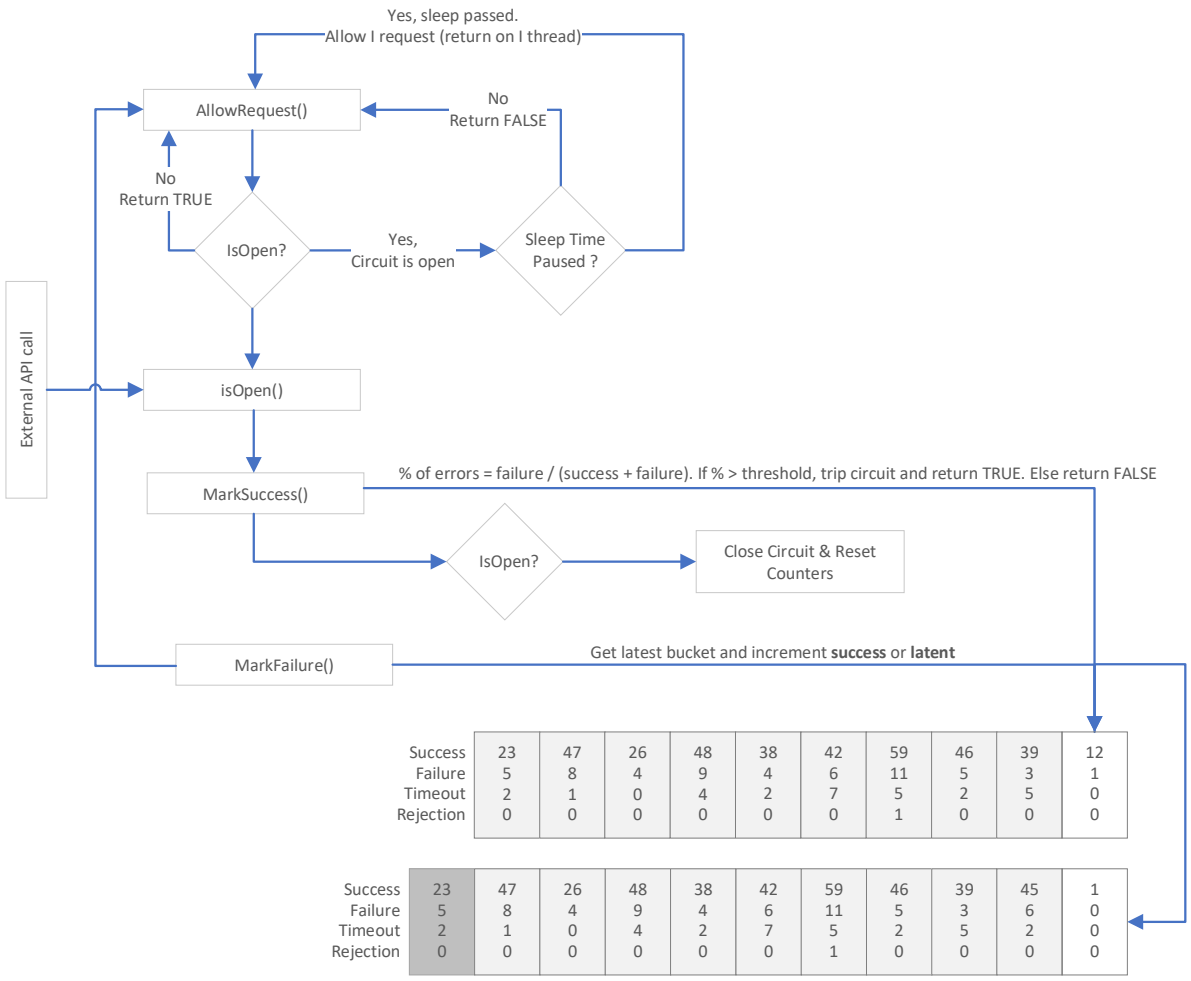
To achieve all the above concerns, Portus is designed to:

- Wrap all calls to external systems in objects which typically execute in threads
- Timeout calls that take longer than configured thresholds.
- Run a thread pool for each external system; if the thread pool becomes full, requests routed to that service will be rejected immediately instead of being queued.
- Measure success and failure rates (error thrown by the external API), timeouts, and thread rejections.
- Trip a circuit-breaker to stop all requests to a particular service for a configured period of time. This can be done manually or automatically (i.e., if the error percentage for the service reaches a threshold).
- Execute fallback logic when a request fails, is rejected, times-out, or short-circuits.
- Monitor metrics and configuration changes and apply them immediately.

When you use Portus to wrap each underlying external service, each external service is isolated from, restricted in the resources it can saturate, and covered with fallback logic that decides what response to return in case of failure.

Portus uses the circuit breaker pattern to detect failures and encapsulate the logic of preventing a failure from constantly recurring during maintenance, temporary external system failure, or unexpected system difficulties.

The following diagram shows how an API call interacts with the circuit breaker and its flow of logic and decision-making, including how the counters behave.



On *GetLatestBucket()* if the I-second window passed, a new bucket is created. The rest slide over and the oldest one is dropped.

The way the circuit-breaker works is:

1. If the volume across the circuit meets a threshold
2. And if the error percentage exceeds the error threshold
3. Then the circuit-breaker transitions from the Closed state to the Open state
4. As long as it's Open, it will short-circuit all requests made against it
5. After some time, a single request is let through the circuit (Half-Open state). If the request fails, the circuit returns to the Open state for the duration of the sleep window. If the request succeeds, the circuit-breaker transitions to Closed and the logic in 1. and starts again.

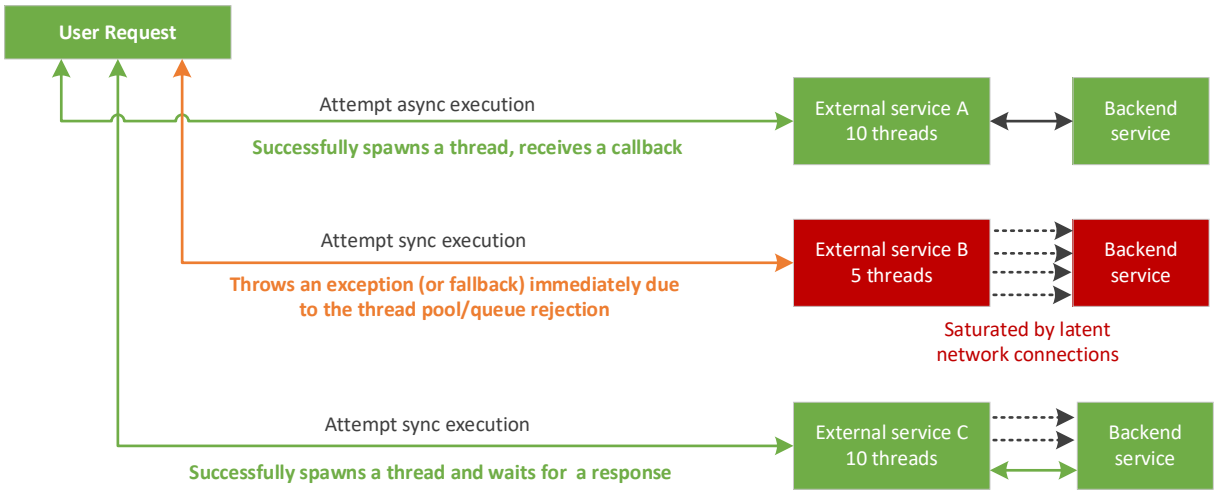
Isolation

Portus employs the Bulkhead pattern to isolate dependencies from each other and to limit concurrent access to any one of them.

The Bulkhead pattern is a type of application design that is tolerant of failure. In a bulkhead architecture, elements of an application are isolated into pools so that if one fails, the others will continue to function. It's named after the sectioned partitions (bulkheads) of a ship's hull. If a ship's hull is compromised, only the damaged section fills with water, which prevents the ship from sinking. [3]

Threads & Thread Pools

Portus uses separate thread pools for each backend service to prevent it from saturating the main thread pool and affect other services running on the Portkey node.



The use of threads and thread pools is an excellent way to achieve isolation for the following reasons:

- A Portkey node can run dozens of different backend services to proxy dozens of external service providers.
- External services can change.
- External service logic can change by adding, updating, or removing different pieces.
- External services can contain logic such as retries, data parsing, caching (in-memory or across the network), and other such behavior.
- External services are "black boxes". They provide no details to their users about implementation, network access patterns, configuration defaults, infrastructure, etc.

- Even if an external service contract doesn't change, the underlying implementation can change, impacting performance characteristics, causing the dAPI configuration to become invalid.
- Network access is mostly performed synchronously.

Why use Thread Pools

The benefits of external service isolation in their thread pools are many:

- The node is fully protected from misbehaving external services. The pool for a given service can fill up without impacting the health of the node.
- When a failed service is restored, the thread pool clears up, and the application immediately resumes performance.
- If an external service is misconfigured, the health of a thread pool will quickly report it through errors, latency, timeouts, rejections, etc., so you can take immediate action (via config properties) without affecting application functionality.
- If an external service alters its performance characteristics (which happens often enough to be an issue), it might need some config property tuning like increasing/decreasing timeouts, changing retries, etc. It becomes immediately visible through thread pool monitoring/alerting metrics like errors, latency, timeouts, and rejections.
- Apart from isolation benefits, dedicated thread pools also provide built-in concurrency, which can be used to build asynchronous facades on top of synchronous services.

To summarize, the isolation provided by thread pools allows external services to be handled gracefully without causing outages.

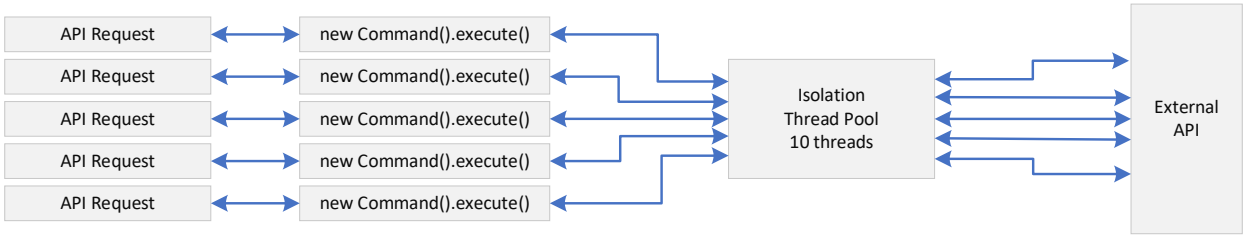
Request Collapsing

You can front an API call with a request collapser to collapse multiple requests into a single external service call.

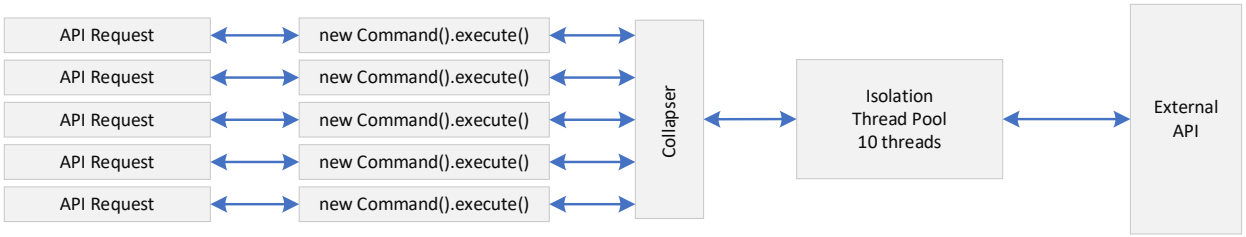
The image below shows the number of threads and network connections in two scenarios:

1. without request collapsing.
2. with request collapsing, assuming all connections are "concurrent" in a window of 10ms.

Without Collapsing: 1 Request = 1 Thread = 1 Network connection



With Collapsing: Requests in a Window = 1 Thread = 1 Network connection



Request collapsing can be used to reduce the number of threads, and network connections needed to perform concurrent external API calls. Portus does this in an automated manner which does not require additional coding or manually batching requests.

Ideally, request collapsing should be done at the node level, so requests from *any user* on any dApp can be collapsed together.

If you configure an API call only to handle batch requests for *a single user*, Portus can collapse requests from a single consumer request.

For example, if a dApp wants to load prices 500 coins, instead of executing 500 network calls, Portus can combine them all into one.

For example, given a list of 500 coins, iterating over them and calling getCurrentPrice() on each one is an obvious method, but if implemented naively, it can result in 500 API calls all being made sequentially, milliseconds distance of each other (and very likely saturating resources).

This could be done manually, such as before allowing the user to call getCurrentPrice(), require them to declare what coins they want to get the price for, so they can be pre-fetched.

Or you could divide the requests so a user must get a list of coins from one place, then ask for the price for that list of coins from another API.

This kind of approach can lead to awkward APIs and interaction models. These, in turn, can lead to simple mistakes and inefficiencies.

Letting Portus handle request collapsing, it doesn't matter how you create the object model or in what order the API calls are made. The getCurrentPrice () method can be put wherever it fits best and be called in whatever manner that suits the usage pattern. Portus will automatically batch calls into time windows.

Enabling request collapsing might increase latency before the actual request is executed. The maximum cost for the original dAPI call is the size of the batch window.

If you have an external request that takes 5ms to execute and a 10ms batch window, the execution time could maximally become 15ms. Typically, a request will not be submitted to the batch window immediately, so the median penalty is half the window time, 5ms in our case.

Suppose a particular command is heavily utilized concurrently and can batch dozens or even hundreds of calls together. In that case, the cost is typically far outweighed by the increased throughput achieved because Portus will reduce the number of threads and the number of network connections to external APIs.

Request Caching

API implementations can define a cache key which is then used to de-dupe calls within a request context in a concurrent-aware manner. The benefits of request caching are that different code paths can execute API calls without concern for duplicate work.

Traceability

dApp API calls are made inside blockchain transactions. The dApp will issue an API call to the Portus smart contract, which will fire an event that the Portkey node will intercept. The node will do its magic and will fulfill or fail the request by calling the Portus smart contract with the result of the API request.

This adds traceability for the consumer both for the API call and for the result of the call. Portus has a block explorer available to trace on-chain transactions for complete traceability.

The Insurance Pool

Decentralized oracle networks should be trusted to a certain extent, rather than being treated as unconditionally trustless. The ideal solution must provide quantifiable security guarantees which can only be assessed using off-chain information.

High-value smart contracts that depend on accurate data from an oracle present a lucrative target for attackers. To protect a contract with economic guarantees, a provider can establish an

insurance amount payable to consumers if something malfunctions (an on-chain insurance policy).

Insurance is mandatory for the API provider and establishing an insurance fund will seriously boost the confidence of the API provider.

We, therefore, make the following assumptions:

- A solution to service malfunctions is to have insurance that pays for damages
- The governing entity is responsible for service malfunctions.
- It is possible to determine service malfunctions, their causes, and the resulting damages in a matter of days.

Insurance is not often used in the crypto space because it naturally requires a third party to resolve claims. Using a mutually trusted third party for this purpose is against the concept of decentralization.

This insurance service will protect the API consumer against damages caused by certain dAPI malfunctions up to a specific limit.

Each dAPI provider will establish an insurance pool contract for the service it provides. The insured amount is placed in an insurance pool contract and will be held there as long as the dAPI provider offers his services. The provider establishes the value of the insurance pool and the arbitration fee he is willing to pay to the claim arbitration team for each resolved claim.

Transaction Fees

Each dAPI transaction between a provider and a consumer is subject to a fee. This includes the underlying blockchain gas fee and the protocol fee. Collected protocol fees are distributed to stakers in the network, proportionally with their staked amount of PORT tokens, every 1 hour.

Staking

Portus is designed with decentralization in mind. To incentivize network participation by token holders, they can opt to stake their coins in insurance pools rather than solely focusing on trading or doing nothing with them at all. The dAPI provider can choose to let network participants stake their tokens in the insurance pool, to cover the insured amount. Participants that stake their tokens in a pool will earn inflationary rewards from the network, based on their staked amount.

Participants that stake their tokens in an insurance pool will also share the risks of the insurance pool. It is imperative for potential stakers to understand those risks and that being a staker is not a passive task. Some actions that a staker should perform are:

- **Perform due diligence on the providers you wish to stake on before committing**. If a provider you staked on misbehaves, a portion of the insurance pool will be distributed to consumers. Stakers should carefully consider the staking choices.
- **Actively monitor the providers you've committed to.** Stakers should ensure the dAPI providers behave correctly, meaning that they have good uptime and don't get compromised.

Claims Arbitration

Arbitrators is responsible for resolving claims using a voting mechanism designed to provide an incentive for people who cast honest votes and a strong disincentive for people who cast fraudulent votes to prevent fraudulent voting.

To prevent fraudulent voting and maliciously defraud the insurance pool, arbitrators must have a stake in the insurance pool. The stake is deposited for a specific voting round, and, provided claims are assessed honestly, it is returned.

In addition, the following other incentive structures will be put in place:

- Voting in agreement with the consensus entitles aribtrators to a share of the arbitration fee in the insurance pool.
- Voting against the consensus will lock the staked amount for a period of time. Each time a vote is cast against the consensus, the locked period will increase.
- To prevent the no-consensus (tie) scenario, a voting round will select an odd number of registered aribtrators.
- For the voting round to be valid, at least 75% of the participants must agree.

Dishonest voters will have their stake locked up in the insurance pool for a long period of time.

A voting round becomes valid once at least three claim arbitrators are registered, to be able to achieve a majority.

To prevent misuse of the claims process by the consumer, before registering a claim, a consumer needs to lock an amount of PORT tokens equal to the arbitration fee set by the dAPI provider. If the claim is rejected, the consumer will lose locked amount and it will be distributed to claim arbitrators. The arbitration fee must be at least 3% of the claimed amount.

Claim arbitrators will have a portal where they will see all open claims and they will choose the ones they want. They will always be anonymous. If a claim does not meet the required number of arbitrators, the arbitration fee will automatically increase, and the claim will be posted again. This process will be repeated one more time up to a total of three. If there are no registered arbitrators in the end, the claim will be cancelled, the locked amount will be returned to the claimant, and he must update the details of the claim to make it clearer.

Reputation

API providers must also support some penalty when something goes wrong. The Insurance Pool will reimburse consumers, but if the API provider has no stake in the pool, he must support the consequences in another way.

A reputation mechanism for providers is needed to give transparency with regards to how reliable a provider is. A token holder can use this information to decide if he will stake his coins in the provider's insurance pool. An initial reputation is established for each provider when he joins the network, and it has a value of 100, which is also the maximum value a provider can have.

Each time a claim is approved, it will directly impact the provider's reputation, decreasing it by 1. If the reputation drops below **ReputationSuspendThreshold** (default 50) points, the provider will be suspended by the protocol, and his services will no longer be available for a configured period.

Reputation can also increase if the provider has no incidents after **ReputationRestoreWindow** blocks (default 1 month). For each **ReputationRestoreWindow** interval that passed, the provider will be given back 1 reputation point.

A history of reputation increases/decreases will be available for stakers to evaluate the performance and reliability of the provider.

Multi-chain

Blockchains are pluggable in Portus. For the protocol to work, a blockchain must support Smart Contracts.

The first implementation is for the Ethereum blockchains, with contracts available in the Solidity language.

Binance Smart Chain (BSC) is planned next, and other chains will follow based on developer requests.

The Portkey node

Portus is a charm used to turn an ordinary object into a Portkey. A Portkey is a magical object enchanted to bring anyone touching it to a specific location instantly.

The main component of Portus is the Portkey node.

It is an on-chain deployable component that providers will use to make their API available to blockchain applications. The node is designed to be a stateless component that does not need a database, and that can run on any operating system, server, and cloud provider. Its primary

purpose is to provide logic to bridge calls between data consumers, protocol smart contracts, and the final API endpoints.

The stateless nature of the Portkey node makes it ideal to be deployed in a container or serverless environment, which directly translates to low maintenance costs, reliability, and availability. The node does not need to be co-located with the actual API endpoint, but it needs to have fast access to the final API endpoint, either via the Internet or private networks. It can be deployed both in public clouds and private infrastructures using Docker containers and serverless functions. Since it does not have a state, one can run multiple instances of a Portkey node, ideally distributed geographically, to serve blockchain dApps in a fast, reliable way instantly.

Existing decentralized API solutions raise significant challenges to traditional API providers who are relatively unfamiliar with blockchain architectures and dApps. The entire ecosystem of various blockchains, smart contracts, data oracles, and tokens is quite complex and can be a barrier for API providers. One would need to acquire knowledge and expertise to help them access these new technologies.

The Portkey node is designed with simplicity in mind, with the primary purpose of making the onboarding process extremely easy for API providers. It's a fire-and-forget solution requiring little to no maintenance on the part of the provider. The providers can use a simple and elegant solution to make their service immediately available to blockchain dApps without having deep technical knowledge.

Allowing dAPI providers to run their own oracle nodes becomes easier for them to service blockchain applications and gain additional monetization of their services and data. Top Chainlink node operators earn as much as $100,000 per month in the blockchain oracle system since DeFi became very popular.

If those rewards were extended directly to the API providers, it could open a whole new market for API providers and decreased costs for dApps that use the external data.

Architecture

When the node is initially being prepared to serve the requests, it prepares all the middlewares and creates request pipelines. Each request pipeline is bound to an endpoint via their common outer interface, resulting in an HTTP handler function. The initializing state transitions to the Running state after the configuration is parsed and the service is prepared.

In the running state, the router maps every request to an HTTP handler function and triggers pipeline execution. Portus doesn't care how many endpoints you have. The performance won't suffer because endpoints are prepared at node startup.

In the initializing stage, all the pipes, tasks, helpers, and generators are created. Each pipeline will be bound to an endpoint, resulting in an HTTP handler function.

The routing layer is responsible for configuring HTTP services, connecting endpoints, and transforming HTTP requests into proxy requests before submitting the task to the proxy layer. When the proxy layer returns a response, the router layer converts it into an HTTP response and sends it to the caller. This layer can be extended to use any HTTP router, framework, or middleware. Adding transport layer adapters for other protocols can be done via plugins, with gRPC and AMQP on the roadmap.

The proxy layer is where most of the Portus magic happens. It has two stacked components:

- The proxy component - a function that converts a given request and its context into a response.
- The middleware component - a function that accepts one or more proxies and returns a new proxy that wraps them.

The proxy layer transforms the request received from the router into a single or several requests to your external services, aggregates responses, and returns a single response.

Middlewares generate custom proxies that are chained together based on the configured workflows. Each generated proxy can transform, clone, filter, and aggregate input data and pass it to the next proxy in the chain, which can modify the received response, adding all kinds of features to the generated pipeline.

Security

Portus is a hybrid solution that must guarantee data integrity to on-chain dApps, preventing 3rd parties from tampering with data. We propose a simple approach where the Portkey node signs their response with the node's private key and then posts it on-chain along with the response. The authenticity of the data can be verified on-chain using the public key of the provider. API consumers will have access to the provider's public key to be able to verify API responses.

Note that signing must be performed by the Portkey node and not by the API provider itself. This would limit adoption as it would require API providers to perform changes to their APIs to implement the signing of responses.

Features

The Portkey node is designed to be a fully stateless component that dAPI providers can deploy to run their own oracle nodes. It must provide features to ensure data consistency, resiliency, performance, reliability, security, and scalability.

Portus is more than a typical proxy that forwards consumers to backend services. It's a pure API Gateway that interacts with different microservices to provide consumers a single interface that improves response times, saves bandwidth, and delivers a better user experience.

Self-service

The Portkey node requires no coding because all data transformations, picking, merging, definitions, and transforms are declared in a single configuration file. All the API gateway behavior can be edited in the JSON file by hand or using the visual designer, a GUI that allows you visually design your API from scratch or edit an existing one.

Performance

It's been built with performance in mind. Expect massive performance and speed in every running instance. Because the Portkey node has been coded in Go using minimal dependencies and the best architecture practices, the node can support a massive amount of traffic with low memory consumption.

Linear Scalability

Unlike other solutions, Portkey's stateless design is the only architecture that can provide true linear scalability. The node does not require coordination or centralization. There is no concept of a cluster because all nodes are autonomous and can keep running even if all other nodes are shut down.

Low Operational Cost

The node can run in very small machines, serverless, and containerized environments. With an average memory usage of 100MB RAM on heavy-traffic machines, the costs can be kept to a minimum compared to existing solutions.

Run it everywhere!

Portkey natively runs in any cloud or private datacenter and is designed to run both in Kubernetes, Docker Swarm, Mesos, and Nomad but also directly installed on bare metal machines and serverless environments.

Extensible

You can extend the node functionality and your business logic with plugins. Many operations can be done directly from the configuration file using Google CEL for validators and Martian DSL for HTTP requests and responses.

Secure

Portus is designed with security in mind. It provides rate-limiting, OAuth, SSL certificates, and protection from clickjacking, XSS, MIME-Sniffing, HSTS, HPKP, and much more.

QoS

The node allows you to control the throttling and usage quotas based on the criteria you specify. The limits can be set both for end-users and at the infrastructure level. When settings limits against your backends, bursting is controlled to allow certain spikes without affecting end users.

Multi-platform

The node is written in Go and runs natively on Linux, BSD, Windows, and Mac OS.

Multi-chain

New blockchains can be added easily. Just migrate the smart contract to the new chain and write a Go plugin to talk to the smart contract.

Open-Source

Portus is entirely open-source under the MIT license. You are free to fork the repos, create pull requests and contribute your ideas. Just make sure you have a look at the contribution guidelines.

Authorization

The Portkey node operator can attach authorization smart contracts to their oracle endpoints, which essentially decide if a consumer should be allowed to query the dAPI based on any criteria that can be implemented on-chain or off-chain. The authorization contracts can define rules, whitelists and blacklists.

PORT tokens

This is the main currency of the network. Decentralized governance requires well-balanced incentive mechanisms to accurately model positive and negative outcomes, rewarding entities for good results and penalizing them for bad ones.

The PORTUS token is designed to have three principal utilities:

- Staking: grants transaction and inflationary rewards to stakers.
- Insurance: backs insurance claims that protect users from damages caused by dAPI malfunction.
- Governance: grants representation in the DAO when the project will be handed over to the community.

Insurance pools are funded using PORT tokens, and claim aribtrators must also stake PORT tokens to cast their vote on consumer claims.

The consumer will support blockchain transaction fees for each dAPI request. It makes onboarding a lot easier for API providers since their only responsibility will be to ensure their dAPI is available and reliable.

Regarding price fluctuations and the value of insurance funds, providers will have the option to update the value of the insurance fund value based on the PORT token price. It will only affect their reputation if they decide to lower the fund's value, with the same percentage as the reduction in funds.

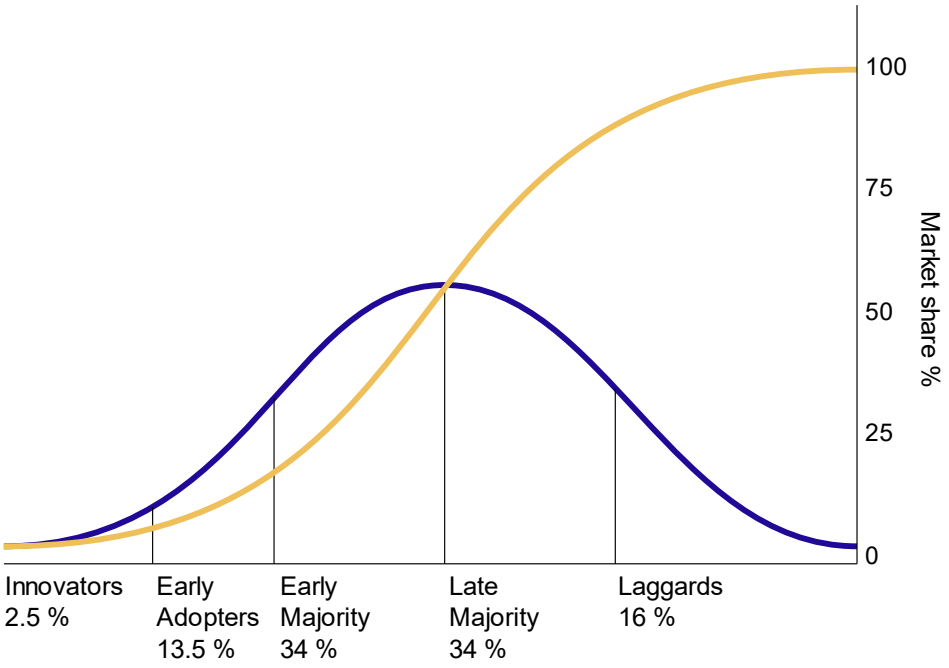
The total supply of tokens will be 300M (five hundred million). Initially, 100M of tokens will be minted to kick-off the project and drive adoption. The final objective of Portus is to completely decentralize the project to a DAO after three years, named further the **Continuity DAO**. It is decentralized and open, meaning that all stakeholders will be able to participate directly in the project's governance. The Portus organization will keep and mediation role if needed until the Continuity DAO is fully operational and self-managed.

Inflation

Portus employs an inflationary model to incentivize token holders. The objective is to make holders stake their tokens on the network, rather than focusing solely on trading or doing nothing with it at all.

Minted tokens need to be distributed to a wide base of holders to grow the network proportionally and avoid large holders in a natural way. We employ the model initially designed by Everett Rogers in his theory Diffusion of innovations. Rogers argues that "*diffusion is the process by which an innovation is communicated over time among the participants in a social system*. *The innovation must be widely adopted in order to self-sustain. The categories of adopters are innovators, early adopters, early majority, late majority, and laggards*."

According to Rogers, the diffusion of innovations is as follows:



With successive groups of consumers adopting Portus (blue), the market cap (yellow) will eventually reach the saturation level. Token distribution happens nfaturally.

The annual inflation rate ranges from 7% to 20% and is adjusted to target to have a 67% of the total supply of PORT tokens staked in insurance pools. This means that when 67% of PORT tokens are staked, the inflation rate will stop changing, based on the model adopted by the Cosmos blockchain.

Since the inflation goes to those staking, it serves as an incentive to secure the network. If the staked ratio is lower than 67%, the inflation rate gradually increases to a maximum of 20%, to incentivize more to stake. Similarly, when more than 67% ok tokens are staked, the inflation rate gradually decreases to a minimum of 7%.

In a staked insurance pool, the rewards are split between stakers. The more you stake, the bigger the reward will be.

A quick way to do calculate the rewards rate is with this equations:

Staked Ratio (%) = *Staked Tokens / Circulating Supply*

Reward Rate (%) = Inflation Rate / Staked Ratio

Example:

For an inflation rate of 20% and 34% of the 100M circulating PORT tokens staked in insurance pools, the reward rate is:

*Reward Rate = 0.12 * 34M/100M = 4%*

Distribution

The PORTUS organization will distribute all tokens in the 3-year period under a fixed schedule for complete transparency. Tokens are organized into six main categories, each with a different purpose, lock-up period, and release schedule.

1. Private Sale: **20,000,000 PORT.** When Private Sale tokens are sold to a buyer, they will be locked for six months, unlocking 1/6 each month to that buyer.
2. Exchange Sale: **20,000,000 PORT**, not locked.
3. Development Fund: **20,000,000 PORT**, gradually released over a period of three years. The tokens will be locked at release time for six months, unlocking 1/6 of the amount each month.
4. Ecosystem Partner Incentives: **15,000,000 PORT**, released when needed and time-locked for six months, unlocking 1/6 of the amount each month to the owner.
5. Team: **15,000,000 PORT,** gradually released over a period of three years. The tokens will be locked at release time for six months, unlocking 1/6 of the amount each month.
6. Continuity Fund: **10,000,000 PORT**. Transferred to the Continuity DAO at fixed intervals and time-locked until Week 162.

Release schedule

PORT tokens will be released on the market at regular intervals to fund project activities and incentivize network participants.

All released tokens will have a total vesting period of 6 months, and each month 1/6 of the vested tokens will be released. This prevents large sales on exchanges and promotes a healthy ecosystem.

Schedule*	**Private Sale**	**Exchange Sale**	**Development Fund**	**Ecosystem Partners**	**Team**	**Continuity Fund**	**Circulating Supply**
T+2W	20				1		21
T+6W		20		1.5	1		43.5
T+10W				1.5	1		46
T+18W				1.5	1	1	49.5
T+30W			2	1.5	1	1	55
T+42W			2	1	1	1	60
T+54W			2	1	1	1	65
T+66W			2	1	1	1	70
T+78W			2	1	1	1	75
T+90W			2	1	1	1	80
T+102W			2	1	1	1	85

T+114W			2	0.75	1	1	89.75
T+126W			2	0.75	1	1	94.5
T+138W			2	0.75	1		98.25
T+150W				0.75	1		100
T+162W							100
TOTAL	**20**	**20**	**20**	**15**	**15**	**10**	**100**

** All values are in millions of PORT tokens. T+**n**W means project start date (T) plus **n** weeks (W).*

Closing Statement

PORTUS will connect decentralized applications with the rest of the world, data, and services offered by traditional APIs, expanding the applicability of blockchain technology without sacrificing decentralization. We envision dAPIs that are fully decentralized and available at scale for all industries and use cases: banking, marketing, media, retail, telecom, and much more.

Simplifying adoption for API providers will increase available options, create competitivity and technological evolution.

dAPIs created with PORTUS do not depend on third-party oracles, have no hidden costs, and are just as secure as any other oracle. Additionally, you have the dAPI insurance poll that provides quantifiable, trustless security guarantees to dAPI consumers, establishing PORTUS as the most secure solution to deliver dAPI services to decentralized applications (dApps).

PORTUS eliminates middlemen, which brings scalability, performance, and cost-efficiency. Consumers do not have to pay fees associated with middlemen to incentivize them against attempting an attack. dAPIs do not require over-redundancy and over-engineering, achieving the same decentralization and significant gas cost reductions.

Finally, PORTUS will evolve into a decentralized organization composed of parties with real skin in the game. The project will constantly evolve to meet new needs and challenges beyond the scope of this whitepaper. This is what blockchain technology is about: decentralization.

The first generation of decentralized applications was limited to the sandbox of the various blockchains. Today, we have decentralized applications that interact with the off-chain world in a limited, pseudo-decentralized way. PORTUS will be at the heart of the next evolution, a new generation of dApps that interact with the off-chain world in an open, trustless, permissionless way.

References

[1] ACID, Wikipedia. https://en.wikipedia.org/wiki/ACID_transactions

[2] Burak Benligiray, Sasa Milic, Heikki Vanttinen, September 2020API3 "Decentralized APIs for Web 3.0", Whitepaper v1.0.1 https://github.com/api3dao/api3-whitepaper/blob/master/api3-whitepaper.pdf

[3] Bulkhead pattern - Cloud Design Patterns, Microsoft Docs. https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead